PLANNING

PLANNING

PLANNING WITH FORWARD SEARCH

Planning with Resources and Concurrency A Forward Chaining Approach*

Fahiem Bacchus
Dept. Of Computer Science
University Of Toronto
Toronto, Ontario
Canada, M5S 1A4
fbacchus@cs.toronto.edu

Michael Ady
Winter City Software
Edmonton, Alberta
Canada, T5R 2M2
winter.city@v-wave.com

Abstract

Recently tremendous advances have been made in the performance of AI planning systems. However increased performance is only one of the prerequisites for bringing planning into the realm of real applications; advances in the scope of problems that can be represented and solved must also be made. In this paper we address two important representational features, concurrently executable actions with varying durations, and metric quantities like resources, both essential for modeling real applications. We show how the forward chaining approach to planning can be extended to allow it to solve planning problems with these two features. Forward chaining using heuristics or domain specific information to guide search has shown itself to be a very promising approach to planning, and it is sensible to try to build on this success. In our experiments we utilize the TLPLAN approach to planning, in which declaratively represented control knowledge is used to guide search. We show that this extra knowledge can be intuitive and easy to obtain, and that with it impressive planning performance can be achieved.

1 Introduction

For a long time AI planning systems were either capable of solving only trivial problems, or required extensive engineered knowledge to solve problems that were still relatively simple. Recently, however, tremendous performance gains have been made. These gains have come from the development of new approaches to planning, and most recently from the improvement of the old idea of forward chaining. As a result, the fastest planning system in the recent AIPS-2000 planning competition [AIPS, 2000] was a forward chaining planner that was able to generate plans containing 2000 steps in less than 2 seconds. This level of performance was achieved on simple test domains. Nevertheless, if performance within one or two orders of magnitude of this can be achieved in real application domains, tremendous possibilities for the practical application of AI planning will be created.

However, performance is not the only impediment to the practical application of AI planning systems. The scope of problems they can represent and solve is also a problem. The planners in the competition were restricted to problems in which actions could be modeled by a set of simultaneous updates to the predicates describing the world. This is the model of planning inherited from the STRIPS action representation; it is also the model used by the ADL [Pednault, 1989] representation. ADL simply provides more flexibility in specifying the set of predicate updates that an action generates, so that, e.g., this set can be conditional on the current world.

Real world applications require modeling a number of more sophisticated features, including uncertainty, sensing, varying action durations, delayed action effects, concurrently executing actions, and metric quantities. In this paper we address the last four issues. In particular, we present an approach to modeling and solving planning problems containing metric quantities, actions of varying duration, actions with delayed effects, and concurrently executing actions. Our approach is based on extending the forward chaining approach to planning. In our experiments we demonstrate that, in particular, the TLPLAN approach to planning [Bacchus & Kabanza, 2000] can be successfully extended to deal with such problems.

Metric quantities have never been a significant problem for forward chaining planners, and in fact the TLPLAN system has been able to deal with metric quantities since its original 1996 implementation. Hence, the ability to deal with metric quantities is not a contribution of this paper. However, the manner in which TLPLAN deals with metric quantities is unique and has many advantages that help support the other extensions that are new to this paper.

In the sequel we first review and motivate the manner in TLPLAN extends the STRIPS/ADL representation to deal with metric quantities. Then we develop our approach to modeling actions with delayed effects from which the ability to model concurrent actions follows naturally. The approach we present can be used in any forward chaining planner. We compare our approach to some of the other work in this area, and then present some empirical results to demonstrate the potential of our approach and the capabilities of the planner

*This research was supported by the Canadian Government through their NSERC and NCE-IRIS programs.

we have developed. Unfortunately it proved to be impossible to run controlled experiments to compare our planner with other systems that have been developed. So we have instead made an effort to present a suite of experiments and provide all of the necessary data sets so that a reusable experimental basis can be established for future work.

2 Functions

We take planning problems as including a fully specified initial state, a goal, and a set of actions for transforming states to new states. A solution to the problem is a sequence of actions that when applied to the initial state yields a sequence of states satisfying the goal. The goal might simply be a condition on the final state of the sequence, or it might place conditions on the entire sequence of states.[1]

A forward chaining planner is one that searches in the space generated by applying to each state s all actions whose preconditions are satisfied by s, starting at the initial state I. A forward chaining planner expands this space as it searches for a sequence of actions that transform I to a state (or sequence of states) satisfying the goal. In other words, forward chaining planners treat the planning problem as a state-based search problem.

The key difference between the planning problem and a generic search problem, however, is that planning assumes a particular representation of states and operators. In planning states are represented as databases of predicate instances, and operators are represented by specifying the set of updates they make to the database (state) to generate a new state (database). In other words, planning uses a factored representation of the state in which each transition updates only a few of the state's components. It is this factored representation that has allowed the development of planning specific notions such as goal-regression.

In planning, the closed world assumption is standard: any predicate instance not in the database is assumed to be false. Under this assumption the state databases become first-order models against which arbitrary first-order formulas can be efficiently evaluated [Halpern & Vardi, 1991]. Given a first-order formula $\phi(\vec{x})$ containing some set of free variables $\vec{x}$, we can efficiently find all tuples of bindings for $\vec{x}$ that make $\phi(\vec{x})$ true in a state: this is the same problem as computing the relation specified by an SQL query in databases.

A natural semantics for operators specified in the STRIPS or ADL notation is to view them as being update queries. Each operator has a precondition $\pi(\vec{x})$ that is a first-order formula containing the free variables $\vec{x}$. Every binding $\vec{c}$ for the variables $\vec{x}$ such that $\pi(\vec{c})$ is true in a state s generates an action that can be applied to s to yield a new state. The STRIPS/ADL action representations have the property that a set of fully instantiated predicates to add and delete from s can be computed simply by evaluating formulas in s. For example, if an ADL operator `(drive ?t ?l ?l')` (drive truck `?t` from location `?l` to `?l'`) contains the conditional update[2]

```
(forall (?o) (in ?o ?t)
  (and (add (at ?o ?l')) (del (at ?o ?l))))
```

(i.e., update the `at` property of all objects `?o` in `?t`), then given a binding for `?t`, `?l`, and `?l'`, i.e., a fixed action instance, by computing the set of bindings for `?o` that satisfy `(in ?o ?t)` in s all instances of `at` that must be changed can be determined. Notice that the specific predicate instances in the update are determined by replacing the terms `?o`, `?l`, and `?l'` by their values. In this case these terms are variables and their values (interpretations) are determined by the current variable bindings.

First-order languages typically include functions. Terms can then be constructed by applying functions to other terms. Thus a *natural* extension to the STRIPS/ADL representation is to remove its function-free restriction. For every function `f` the state can include in its database a relation specifying the value of `f` on its various arguments. First-order formulas can be evaluated just as before: whenever we encounter a term like `(f t1 ... tk)` in a formula we replace it with its value by recursively evaluating each of the `ti` and then looking up the resulting tuple of values in the relation specifying `f`. We specify updates to these function values by asserting equalities that must hold in the next state. Now, e.g., instead of describing the location of objects `?x` with an `(at ?x ?l)` relation, we could describe their location with a `(loc ?x)` function. Then the operator `(drive ?t ?l ?l')` could contain the conditional update

```
(forall (?o) (in ?o ?t) (add (= (loc ?o) ?l)))
```

We use the convention that the function that is the first argument of the equality (`loc`) is the function to be updated, its arguments (the variable `?o`) are evaluated in the current state to determine which arguments of `loc` are to be updated, and the second argument (the term `?l`) is evaluated in the current state to determine the new value.

Functions whose values are numbers, and numeric functions like $+$ can now be accommodated in the same way. Furthermore, the standard numeric functions like $+$ can be computed using existing hardware or software: we do not need to have a table of its values as part of the state's database.

For example, if `(capacity ?t)` is the fuel capacity of truck `?t`, `(fuel ?t)` is its current level of fuel, and `(fuel-used)` is a (0-ary) function whose value in any state is the total amount of fuel used, then we can write an operator like `(`**`refuel`** `?t)` with the update

```
(and
 (add (= (fuel-used)
         (+ (fuel-used)
         (- (capacity ?t) (fuel ?t)))))
 (add (= (fuel ?t) (capacity ?t))))
```

In the new state truck `?t` will have a full tank and we would have accounted for the amount of fuel put into its tank.

[1] See [Bacchus & Kabanza, 1998] for more about such "temporally-extended" goals. In this paper we will confine our attention to "final-state" goals. However, the extensions we describe here could also be realized in the context of temporally-extended goals.

[2] The update asserts the truth or falsity of a collection of predicate instances in the next state, and due to the closed world assumption these assertions can be realized by adding or deleting these instances from s.

Adding functions to the action representation in this way, motivated directly by viewing operators as database updates, provides all the flexibility needed to model complex resource usage. For example, the following operators model FIFO access to a fixed resource and also track the number of times the resource is used. `(qhead)` and `(qtail)` are 0 in the initial state, `(queue i)` is a function whose value is the `i`'th request in the queue, and `(serve ?x)` is a predicate true of `?x` if `?x` is currently being served. `(qtail)` will always be the total number of times the resource is used, and `(qhead)` − `(qtail)` is always the number of items currently in the queue.

```
(def-adl-operator (enqueue-access ?x)
  (and (add (= (queue (qtail)) ?x))
       (add (= (qtail) (+ (qtail) 1)))))
(def-adl-operator (dequeue-and-serve)
  (pre (> 0 (- (qtail) (qhead))))
  (and
    (del (serve (queue (qhead))))
    (add (serve (queue (+ (qhead) 1))))
    (add (= (qhead) (+ (qhead 1))))))
```

In the literature addressing metric quantities, specialized notation has been developed for expressing resources (e.g., [Wolfman & Weld, 1999; Kvarnström, Doherty, & Haslum, 2000]). Our argument is that such notation is not required. The natural extension of making functions first-class citizens along with standard operator preconditions provides a better solution.[3] What we have just described is the manner in which TLPLAN has implemented functions since its original 1996 version. Functions as first-class citizens were also present in the original ADL formalism [Pednault, 1989], and [Geffner, 2000] provides some other arguments in support of using functions.

3 Modeling Concurrent Actions

Forward chaining has proved itself to be a very fruitful basis for implementing high-performance planners. For example, the two fastest planners in the recent AIPS-2000 planning competition (TALPlanner [Doherty & Kvarnström, 1999], a planner that uses the TLPLAN approach, and Fast-Forward [Hoffmann, 2000] a planner using domain independent heuristics to guide its search) were both forward chaining planners. However, there is at least one aspect of forward chaining planners that seems to be problematic: they explore totally ordered sequences of actions. This is where they get their power: such sequences provide complete information about the current state and that information can provide powerful guidance for search. But modeling concurrent actions with linear sequences seems to be problematic. However it turns out that there is a surprisingly simple way of modeling concurrency with linear actions sequences.

We associate with every state a time stamp, starting with a fixed start time in the initial state. The time stamp denotes the actual time the state will occur during the execution of a plan. In a linear sequence of states a number of successive states may have the same time stamp. Intuitively this means that the transitions between these states occur instantaneously, so the intermediate states are never physically realized. Their existence is simply a convenient computational fiction.

Additionally, each state has an event queue. The event queue contains a set of updates (events) each scheduled to occur at some time in the future (of the state's time stamp). Along any fixed sequence of states generated by a sequence of actions, each state inherits the pending events of its parent state. It might also queue up some additional events to be passed to its children. Thus if we arrive at the same state via two different actions sequences we could generate two different event queues. We regard two states as being equal only if they have both the same database and the same event queue. Thus, when the planner backtracks it backtracks to a state with a prior event queue, in effect backtracking the state of the event queue.

As before, an action a can be executed in a state s only if its preconditions are satisfied by s. Applying a to s generates a new successor state s^+. Standard actions do not advance the world clock, so s^+ will have the same time stamp as s. Typically, it will have a different event queue (i.e., what will happen in the future has changed), and a different database (i.e., what is true "now" has changed). Updates to s's database are used to model a's instantaneous effects, and updates to the event queue are used to model a's delayed effects.

For example, consider the action of driving a truck `?t` from `?l` to location `?l'`:

```
(def-adl-operator (drive ?t ?l ?l')
  (pre (?t)  (truck ?t)
       (?l)  (loc ?l)
       (?l') (loc ?l')
     (at ?t ?l))
  (del (at ?t ?l))
  (delayed-effect
    (/ (dist ?l ?l') (speed ?t))
    (arrived-driving ?t ?l ?l')
    (add (at ?t ?l'))))
```

We can execute this action in s if `?t` is a truck, both `?l` and `?l'` are locations, and `?t` is at location `?l` in s. The instantaneous effect of the action is to delete the current location of the truck, and the delayed effect is to add the new location of the truck. The delayed effect is realized by adding an item to the event queue. The first argument of the **`delayed-effect`** specification is the event's time delta, the number of time units from the current time the event is scheduled to occur. This time delta is a term that will be evaluated in the current state. In this case it is the distance between the two locations divided by the speed of the vehicle. The next argument is simply a label for the event (designed to make the final plan more readable). The delayed effects are the subsequent arguments. In this case it is the addition of the new location of the truck. In general, delayed effects can be any kind of effect allowed in a normal action, including, e.g., conditional effects.

In addition to the standard actions there is one special action that advances the world clock: the **`unqueue-event`**

[3] Computing plans in the presence of metric quantities might require restrictions on how these quantities can be updated. But such restrictions should be imposed on a general representation, not used to determine the representation.

```
Plan(<s,Q>,Goal)
 if (s ⊨ Goal and Q = {})
   return(s)
 else
   s+ := s
   s+.prev := s
   Q+ := Q
 choice a ∈ {act : s ⊨ pre(act)}
   s+.action := a
   if a != unqueue-event
     s+ := ApplyInstantaneousUpdates(s,a)
     Q+ := AddDelayedEvents(Q,s,a)
   else
        newTime := eventTime(front(Q))
        s+.time := newTime
        while eventTime(front(Q+)) == newTime
          e := removeFront(Q+)
          s+ := ApplyEffect(s+,e)
   Plan(s+,Q+)
```

Figure 1: Forward Chaining Search

action.[4] This action moves time forward to the next scheduled event, removes all events scheduled for that new time and uses them to update the state's database. This realizes various delayed effects of previous actions. For example, eventually the arrival of `?t` at `?l'` will reach the front of the queue and will be dequeued. This will cause a transition to a new state in which the fact `(at ?t ?l')` is added and the time is updated. If a set of events have been scheduled for the same time, they will all be dequeued and applied sequentially in FIFO order.

Figure 1 specifies more precisely forward chaining search in this enhanced search space. The non-deterministic ***choice*** operator is realized by search. `AddDelayedEvents` examines the action, and for each **`delayed-effect`** in `a` evaluates the term specifying the delay of that effect. Adding the time of the current state, `s.time`, gives the absolute time of the effect, and the effect is merged into the queue so as to keep the queue in time sorted order. The current variable bindings for the free variables in the effect are also stored along with the effect. If the chosen action is **`unqueue-event`** time is moved forward, and all effects scheduled for that time are removed from the queue and applied sequentially to the state. A goal state is a state whose database satisfies the goal and that has an empty event queue; we can find the sequence of actions leading to that state by following the state's `prev` pointers.

Search for a plan is started by calling **Plan** on the initial state. Note that the queue need not initially be empty. Instead it could contain some set of events that are going to occur in the future. The planner will then have to find a plan that negotiates around these future events. This also facilitates replanning where some previous actions cannot be canceled. This feature is similar to the ability of temporal refinement planners like IxTeT [Ghallab & Laruelle, 1994] and RAX [Jónsson *et al.*, 2000] to flesh out an initial set of temporal constraints.

The choice of which action to try next is where heuristic or domain specific control comes into play. In the TLPLAN approach we restrict the set of possible action choices by requiring that the next state s^+ satisfy the temporal control formula (see [Bacchus & Kabanza, 2000] for details).

With delayed effects, concurrent actions are automatic. When an action is executed it generates a successor state in which its immediate effects have been made. This state "marks" the start of the action, and since it has the same time stamp as the previous state the action can be viewed as starting at the current time. After some stream of delayed effects have been executed the final delayed effect generates a state that "marks" the end of the action. Depending on how we interleave the **`unqueue-event`** action with the ordinary actions we can start a whole series of actions at the same time, these actions can execute concurrently and some can end before others. Thus at any particular time any number of actions can be executing concurrently. If we only choose **`unqueue-event`** when there is no other action available, we will maximize the number of concurrent actions at each stage: each ordinary action whose precondition is satisfied will be started before the world clock is advanced. Or we can achieve finer control over the degree of concurrency by controlling (via, e.g., a temporal control formula) when **`unqueue-event`** is chosen.

In our approach all concurrency control is handled by action preconditions. Typically the instantaneous effects of an action are used to modify the state so as to achieve concurrency control, while the delayed effects of an action are used to model physical achievements in the world. This is a low level but very powerful approach to concurrency control. For example, in the previous **`drive`** operator, the current location of the truck is immediately deleted. Since this is also a precondition of **`drive`**, any attempt to concurrently drive the same truck to another location is blocked. More sophisticated situations are also quite straightforward to model, in part because of our general approach to functions and numeric computations.

For example, consider a gas station with 6 refueling bays and a limited amount of fuel shared among these bays. Let `(station-fuel)` be the current amount of fuel at the station, `(bays-free)` the number of bays currently free, `(capacity ?v)` the fuel capacity of a vehicle, and `(fuel ?v)` the fuel in the vehicle. Then the following actions model resource bounded concurrent access to the gas station:

```
(def-adl-operator (refuel ?v ?amount)
 (pre
   (?v)        (vehicle ?v)
   (?amount)   (= ?amount (- (capacity ?v)
                             (fuel ?v)))
  (and (> 0 (bays-free))
       (> (station-fuel) ?amount)))
 (add (= (station-fuel)
         (- (station-fuel) ?amount)))
 (add (= (bays-free) (- (bays-free) 1)))
 (delayed-effect 10
  (fueled ?v ?amount)
```

[4] Dead time can be inserted into the plan by including a "wait" action in the domain. This action would have no instantaneous effects and would enqueue a null delayed effect, delayed by the wait period, into the event queue. The presence of such an event on the queue would allow **`unqueue-event`** to advance the world time by the wait period.

```
(and (add (= (bays-free)
             (+ (bays-free) 1)))
     (add (= (fuel ?v) (capacity ?v))))))
```

The operator also demonstrates our system's ability to use functions to bind operator arguments. In this case `?amount` is bound to a function value computed from the binding of `?v`.

Suppose in a state s there are 12 vehicles and that various sets of these vehicles require more fuel that the station has. From s a number of different sequences of concurrent **`refuel`** actions can be initiated. But in each of these sequences no more than 6 vehicles will be concurrently fueled, due to the instantaneous update of the `(bays-free)` resource, and no set of vehicles needing more fuel than available will be concurrently fueled, due to the instantaneous update of the `(station-fuel)` resource. In fact, it can be that after the first batch of 6 vehicles is concurrently fueled an additional batch of vehicles enter the station after 10 units of time have elapsed and the bays have become free.[5] Exactly which sequence of concurrent refueling appears in the plan will be determined by what sequences allow the goal to be achieved and the search strategy.

The final plan will be a linear sequence of actions grouped into subsequences of actions each with the same time stamp. However, the linear sequencing is not a limitation of our approach. For example, a simple post analysis of these subsequences can be used to determine if the actions have to be started in the supplied order or if some other ordering can be used. For example, if no action in the subsequence affects the preconditions of another then they can be started in any order, or simultaneously. However, if starting the actions is near instantaneous in practice then there will be little to gain from such a post analysis. For example, say that one subsequence of actions to be executed at the same time is `(drive truck1 locA locB)` followed `(drive truck2 locC locD)` (start driving two trucks concurrently) then it typically will make very little difference if we tell `truck1` to start driving before telling `truck2`—the command to start driving takes negligible time in comparison to the actual drive.

Often what is required in these kinds of planning problems are goals that specify conditions over time. That is, specifying conditions on the final state is not sufficient. Temporal refinement planners like IxTeT [Ghallab & Laruelle, 1994] allow one to, e.g., enforce that a predicate holds without interruption over a particular interval of time. In our approach if one specifies only a condition on the final state, there is no way of stopping the planner from inserting actions produce undesired intermittent effects—there is no way of telling the planner that these intermittent effects are undesirable. Although we have not implemented it, there is no conceptual difficulty with combining our approach with our previous work on specifying temporally-extended goals [Bacchus & Kabanza, 1998]. With such a combination, an extremely rich set of extended conditions can be enforced on the final plan, including the typical conditions supported by temporal refinement planners.

[5] It would be easy have a more complex model of the time required to complete the fueling.

4 Empirical Results

We have implemented the above event-queue mechanism as an extension of the TLPLAN system, and tested our implementation using different versions of the metric logistics domain developed by [Wolfman & Weld, 1999].

Using this domain allows us to make some empirical comparisons with previous work. Unfortunately, it proved to be impossible to run controlled experiments with other planning systems (the systems and problems sets were not readily available, or the systems were not easily ported to our machine). Therefore, the results we report are simply to demonstrate that our approach can efficiently solve large planning problems containing the features we are concerned with. However, we are reporting a range of results, and making the test sets available [Bacchus, 2001] so as to provide an experimental base for future work.

In the logistic domain there are a collection of packages that need to be transported to their final destination, trucks for moving packages between points in a city, and planes for moving packages between airports located in different cities. Packages can be loaded and unloaded from vehicles and the vehicles can be moved between compatible locations.

[Wolfman & Weld, 1999] added a fuel capacity for each vehicle, and a refueling action that fills up a vehicle given that the vehicle is located at a depot. In addition, each `drive-truck` operator consumes a fixed amount of fuel, each `fly-airplane` operator consumes an amount of fuel based on the (fixed) fuel efficiency of planes and the distance between the two airports, and one is not allowed to move a vehicle to a location unless it contains enough fuel to get there.

We take the TLPLAN approach to planning in which domain specific information is declaratively encoded in a temporal logic. Unlike standard heuristics which try to measure the worth of a state, TLPLAN typically uses negative information that tells it that certain kinds of action sequences are flawed [Kibler & Morris, 1981]. This information is checked against the sequences generated during forward chaining, and any sequence satisfying a bad property is pruned from the search space. This approach to planning has proved to be extremely successful: in yields a level of planning performance that is an order of magnitude better than any other approach, and the approach has been applied to a wide range of different domains. In [Bacchus & Kabanza, 2000] an extensive set of examples and empirical results are presented to demonstrate both the performance of this approach and the fact that the requisite knowledge for many different planning domains is easily obtained and represented in the formalism.

In the standard logistic world the control information needed is very simple.

1. Don't move a vehicle to a location unless it needs to go there to pickup or drop off a package.
2. Don't move a vehicle from a location while it still contains a package that needs to be dropped off at that location.
3. A package needs to be picked up by a truck if it needs to be moved to another destination in the same city.

4. A package needs to be picked up by a plane if it needs to be moved to another city.
5. A package needs to be dropped off from a truck if the truck is at its goal destination or if the truck is at an airport and its goal destination is in another city.
6. A package needs to be dropped off from a plane if the plane is in the package's destination city.

Each of these assertions can be easily encoded as a temporal logic formula [Bacchus & Kabanza, 2000], and with this collection of assertions the planner finds plans very efficiently. Furthermore, this control knowledge is of such a simple form that it becomes possible, by looking at each action's effects, to predict whether or not an action will extend the current plan in such a way as to violate one of these assertions. As a result one can systematically convert the control rules into extra action preconditions [Bacchus & Ady, 1999]. This has the effect of blocking an action first, rather than executing it, generating the plan extension, and then determining that the extension is invalid. Due to the extremely high branching factor in larger logistic problems (over a 1,000 applicable actions in each state on the harder problems), this "compiled one-step" look ahead improves planning performance by a couple orders of magnitude. In our experiments, we used a precondition encoding of these control rules.

We used two sets of test problems. A set of four small problems, loga, logb, logc, and logd, utilized by Wolfman in testing his LPSAT system, and a collection of 30 much larger problems used in the AIPS98 planning competition.[6] All of the experiments were run on a 500MHz PIII machine with 512MB of memory. All times are reported in CPU seconds.

In Table 1 the first set of columns gives the time it takes the current version of TLPLAN to solve the original version of these problems, and the number of steps in the resulting plan. We then encoded Wolfman's metric version, with fuel consumption, and ran the problems again. We used Wolfman's loga–logd problems directly, and for the AIPS98 suite we added distances between the locations, fuel consumption rates and fuel tank capacities for the planes and trucks. We found that TLPLAN could solve these metric logistics problems very efficiently with the same control knowledge as used in the standard logistics world, along with the extra information

1. Allow a vehicle to move to a depot if it needs fuel.
2. Don't refuel a vehicle unless it needs more fuel to make a pickup or drop off.
3. Don't move a truck or plane to a location in order to pickup an object if there is already exists a similar vehicle at that location with sufficient fuel capacity to take it to its destination.

The times required to solve the fuel version of the logistic problems are shown in the second set of columns of the table. The data shows that our approach finds the metric problems not that much more difficult than the standard problems.

[6] This test suite is not to be confused with the 30 problem ATT logistics suite which are much easier.

[Wolfman & Weld, 1999] present a SAT encoding approach to solving these metric logistic problems. Their planner utilizes a combination of SAT solving and linear programming: the SAT solver finds the plan while the linear program solver ensures that the plan satisfies the (linear) metric constraints. The solution times they report are approximately 10 sec. for loga, 300 sec. for logb, 500 sec. for logc and 5000 sec. for logd (they also point out that their approach was faster than previous approaches). These times indicate that their approach scales poorly. Interestingly, in examining the plans their system generated[7] it was found that these plans used much more fuel, e.g., 6426.67 units for the loga solution, and also contained many unnecessary moves, e.g., moving a truck back and forth without using it to transport any packages. Their system does not use domain specific control knowledge but some of the knowledge used here might be useful in improving the performance of their system.

Then, we took the metric logistics domain and made it concurrent, so that the domain contained both metric quantities and concurrent actions. In the concurrent version loads and unloads take 1 unit of time to complete, and multiple concurrent loads/unloads into the same vehicle are allowed (the same object cannot be manipulated concurrently). Refueling a truck takes 1 unit of time, and an airplane takes 10 units. Finally, driving a truck takes 5 units of time, and flying an airplane takes time that is dependent on the distance between the two locations. To the above control knowledge we added the extra information

1. A vehicle can be moved to a location if there is an object en route to that location (in a different type of vehicle) that can be transported by the vehicle.

This allows the planner to get the vehicles moving so that they can make progress towards the pickup location concurrently with the object they are to pickup. For example, the planner can start flying a plane to an airport while a truck is transporting an object to the airport that needs to be transported to another city. This decreases the duration of the plan. The last set of columns shows our results for this domain. In this case we show the duration of the plan as well as the length of the plan (number of actions). Since the actions take at least 1 unit of time, it can be seen that highly concurrent plans are being found. The time to find a solution has also climbed, but not by much, and so has fuel consumption. This makes sense, as the concurrent plan will try to utilize more vehicles in order to maximize concurrency, and more vehicles means more fuel.

Another planning system that is capable of dealing with concurrent actions of different durations is the TGP system [Smith & Weld, 1999] that is based on GraphPlan. However, its underlying algorithms are considerably more complex than the approach we suggest here, and it cannot deal with metric quantities. We were able to run the TGP system on some simpler logistic problems involving varying action durations but without fuel consumption. We found that TGP could not solve any of loga–logd problems (when we removed the fuel consumption component) even when given an hour of CPU time. These results and the performance of Wolfman's LPSAT system, demonstrate that adding domain

[7] Thanks to Steve Wolfman for supplying us with these solutions.

Problem	Standard		Metric Fuel			Fuel+Concurrent			
	CPU	Len.	CPU	Len.	Fuel	CPU	Dur.	Len.	Fuel
loga	0.02	51	0.06	60	2558	0.06	35.00	84	2518
logb	0.10	42	0.06	49	1392	0.08	35.25	90	2425
logc	0.02	51	0.08	60	3158	0.09	42.75	92	3158
logd	0.07	70	0.15	80	4384	0.19	67.25	154	4960
x-1	0.01	26	0.03	26	846	0.07	24.50	53	1994
x-2	0.03	33	0.11	33	2005	0.18	18.00	62	2187
x-3	0.15	55	0.39	55	3364	0.54	26.75	94	3963
x-4	0.22	59	0.53	59	3694	1.17	28.50	114	4562
x-5	0.02	22	0.03	22	1442	0.06	23.00	39	1825
x-6	0.33	72	0.83	74	5549	1.30	38.50	118	5886
x-7	0.04	34	0.22	34	2356	0.31	21.75	64	2855
x-8	0.16	41	0.77	41	3941	1.23	34.25	85	5571
x-9	0.41	85	1.17	85	3923	2.14	28.50	152	5595
x-10	0.50	105	1.77	106	11285	1.69	60.50	184	14554
x-11	0.05	31	0.12	32	859	0.20	16.75	49	1062
x-12	0.35	41	1.06	41	3415	3.16	25.50	73	3528
x-13	0.68	67	3.20	68	6736	2.95	27.25	110	8235
x-14	0.37	94	2.27	94	4641	1.87	24.00	142	5774
x-15	0.12	94	0.48	97	1406	0.43	47.75	151	1786
x-16	0.26	58	1.07	58	1937	2.01	29.00	93	2319
x-17	0.08	45	0.39	45	945	0.72	21.75	76	1497
x-18	3.23	170	10.24	174	15914	7.12	43.50	275	24111
x-19	2.24	153	5.17	159	4373	7.40	48.25	270	7229
x-20	2.34	150	7.35	156	7527	7.02	56.00	226	7999
x-21	1.52	104	4.03	105	4621	9.05	37.00	184	5803
x-22	17.95	296	34.75	305	25121	33.87	79.00	498	35278
x-23	0.25	115	1.20	115	3796	1.08	28.75	183	5728
x-24	0.30	41	1.40	41	1356	7.99	26.00	77	1759
x-25	7.66	190	16.77	196	14209	16.65	50.00	284	19203
x-26	4.89	194	11.65	203	39616	17.40	241.50	361	44291
x-27	2.90	149	16.44	155	7041	14.03	54.00	263	11293
x-28	26.10	274	63.86	283	13855	197.38	58.75	478	21899
x-29	20.36	330	43.08	339	26236	22.03	58.75	531	44308
x-30	4.60	136	10.57	139	10648	63.97	55.00	265	15265

Table 1: Test results on versions of Logistics

specific control knowledge and utilizing our forward chaining approach allows us to move to another level of planning performance.

There are two other planning systems that are quite similar to ours, [Pirri & Reiter, 2000] and [Kvarnström, Doherty, & Haslum, 2000]. Both of these planner utilize the TLPLAN approach and display good performance. However, both of these systems utilize a rich logical representation for actions and states, whereas the approach we present here can be utilized by any forward chaining planner with the much simpler STRIPS/ADL action representation.

Finally, temporal refinement planners are an alternate approach to planning with concurrent actions. The IxTeT planner is an impressive system capable of dealing with resources and concurrent actions [Ghallab & Laruelle, 1994]. NASA's remote agent project RAX also utilized a refinement planner. Temporal refinement planners operate by taking an initial plan that typically specifies the initial state and various goal conditions, and refining that plan by adding additional actions to achieve open conditions or constraints to protect other conditions. A key component of these planners is the use of constraint propagation to maintain the temporal constraints imposed on the plan during the refinement process. Thus these planner search in a space of partially specified plans using constraint propagation to detect deadends, rather than in a space of fully specified worlds as in our approach. The RAX planner also utilized extensive search control knowledge in order to achieve its good level of performance. However, the control knowledge was at a much lower level and was more procedural in style than that utilized by our approach. Many of the standard concurrency control paradigms are easier to specify with a temporal refinement planner than in our approach. However, it should be possible to develop macros for our approach to encapsulate many of these paradigms thus

easing the specification problem.

5 Conclusion

Forward chaining's ability to deal with metric quantities was already documented, but in this paper we have demonstrated that there is also a simple way of extending forward chaining to deal with concurrent actions. These two features can then be combined with other ideas like search control knowledge to yield a powerful approach to planning in the presence of concurrent actions of differing durations and resources.

Our empirical results show that complex and lengthy plans can be generated with our approach, and serve to demonstrate the potential of our approach. Further verifying that potential is the subject of future work.

Other items of future work include (a) higher level constructs for concurrency control implemented as macros that can are expanded to the very general lower level constructs already supported by our system, and (b) access by actions to the event queue. This last is worth further explanation. Consider a situation where a truck is being driven from location A to location B. At the start of the drive its arrival at location B is entered into the event queue. Now it could be that location B only has capacity for one truck, thus a subsequent action should not be scheduled that would cause another truck to arrive at location B at the same time. Checking this "precondition" involves querying the event queue. A different example is when via some other action or event the truck gets a flat tire en route. This will delay its arrival time. Thus the "flat-tire" event must not only change the current state of the world but it must also alter events in the event queue. Simply put actions must be able to treat the event queue just like the state's database: they must be able to query and update it. With this ability it becomes easier to model on-going processes that can be interrupted and restarted, something that is cumbersome in our current model.

Acknowledgments The referees provided some very useful suggestions that helped improve the paper.

References

[AIPS, 2000] AIPS 2000. Artificial Intelligence Planning & Scheduling 2000 planning competition. http://www.cs.toronto.edu/aips2000/

[Bacchus & Ady, 1999] Bacchus, F., and Ady, M. 1999. Precondition control. available at http://www.cs.toronto.edu/~fbacchus/on-line.html

[Bacchus & Kabanza, 1998] Bacchus, F., and Kabanza, F. 1998. Planning for temporally extended goals. *Annuals of Mathematics and Artificial Intelligence* 22:5–27.

[Bacchus & Kabanza, 2000] Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116:123–191.

[Bacchus, 2001] Bacchus, F. 2001. On line experimental data sets. http://www.cs.toronto.edu/~fbacchus/tlplan.html

[Doherty & Kvarnström, 1999] Doherty, P., and Kvarnström, J. 1999. Talplanner: An empirical investigation of a temporal logic-based forward chaining planner. In *Proceedings of TIME '99, IEEE Computer Society*, 47–54.

[Geffner, 2000] Geffner, H. 2000. Functional strips: a more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer. in press.

[Ghallab & Laruelle, 1994] Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proceedings of the International Conference on Artificial Intelligence Planning*, 61–67. AAAI Press.

[Halpern & Vardi, 1991] Halpern, J. Y., and Vardi, M. Y. 1991. Model checking vs. theorem proving: a manifesto. In Allen, J. A.; Fikes, R.; and Sandewall, E., eds., *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*. San Mateo, CA: Morgan Kaufmann, San Mateo, California. 325–334.

[Hoffmann, 2000] Hoffmann, J. 2000. Fast-forward. http://www.informatik.uni-freiburg.de/~hoffmann/ff.html.

[Jónsson *et al.*, 2000] Jónsson, A. K.; Morris, P. H.; Muschettola, N.; and Rajan, K. 2000. Planning in interplanetary space: Theory and practice. In *Proceedings of the International Conference on Artificial Intelligence Planning*, 177–186. AAAI Press.

[Kibler & Morris, 1981] Kibler, D., and Morris, P. 1981. Don't be stupid. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, 345–347.

[Kvarnström, Doherty, & Haslum, 2000] Kvarnström, J.; Doherty, P.; and Haslum, P. 2000. Extending TALplanner with concurrency and resources. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*.

[Pednault, 1989] Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 324–332.

[Pirri & Reiter, 2000] Pirri, F., and Reiter, R. 2000. Planning with natural actions in the situation calculus. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer Press. in press.

[Smith & Weld, 1999] Smith, D. E., and Weld, D. S. 1999. Temporal planning with mutual exclusion reasoning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 326–337.

[Wolfman & Weld, 1999] Wolfman, S. A., and Weld, D. S. 1999. The lpsat engine and its application to resource planning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 310–317.

Total-Order Planning with Partially Ordered Subtasks

Dana Nau
Univ. of Maryland, College Park, MD, USA
nau@cs.umd.edu

Héctor Muñoz-Avila
Univ. of Maryland, College Park, MD, USA
munoz @cs.umd.edu

Yue Cao
Solers Inc.
Arlington, VA
JasonC@homemail.com

Amnon Lotem
Estimotion, Ltd.
Glil Yam, Israel
lotem@cs.umd.edu

Steven Mitchell
Lockheed Martin
Manassas, VA, USA
steve.mitchell@lmco.com

Abstract

One of the more controversial recent planning algorithms is the SHOP algorithm, an HTN planning algorithm that plans for tasks in the same order that they are to be executed. SHOP can use domain-dependent knowledge to generate plans very quickly, but it can be difficult to write good knowledge bases for SHOP.

Our hypothesis is that this difficulty is because SHOP's total-ordering requirement for the subtasks of its methods is more restrictive than it needs to be. To examine this hypothesis, we have developed a new HTN planning algorithm called SHOP2. Like SHOP, SHOP2 is sound and complete, and it constructs plans in the same order that they will later be executed. But unlike SHOP, SHOP2 allows the subtasks of each method to be partially ordered.

Our experimental results suggest that in some problem domains, the difficulty of writing SHOP knowledge bases derives from SHOP's total-ordering requirement—and that in such cases, SHOP2 can plan as efficiently as SHOP using knowledge bases simpler than those needed by SHOP.

1 Introduction

One of the more controversial recent planning systems is the SHOP system [Nau *et al.*, 1999], an HTN planning system that plans for tasks in the same order that they will later be executed. Like any HTN planner, SHOP uses domain knowledge in order to plan more efficiently—but unlike other HTN planners, SHOP always generates the steps of its plans in the same order that those steps will later be executed.

On one hand, the SHOP algorithm makes it possible to generate plans quite efficiently. For example, in the experiments reported in [Nau *et al.*, 1999, 2000], SHOP ran orders of magnitude faster than the Blackbox, IPP, Tlplan, and UMCP planners. Furthermore, the SHOP algorithm is suitable for use as an embedded planning sytem in complex applications [Munoz *et al.*, 2000].

On the other hand, creating a SHOP knowledge base can require significantly more "programming effort" than is needed for action-based planners. For example, in two of the planning domains in Track 2 of the AIPS-2000 planning competition, SHOP was disqualified because we did not finish debugging the knowledge bases in time.

We believe that although the total-order HTN-decomposition technique used in SHOP has some significant benefits, the SHOP planning algorithm provides too restrictive a way of achieving these benefits. In particular, SHOP requires the subtasks of each method to be totally ordered, which makes it impossible for SHOP to interleave subtasks of different tasks. In Section 2 we describe how this can complicate the job of the knowledge-base author by requiring him/her to introduce "global planning" instructions into SHOP's knowledge bases that would not otherwise be needed.

In this paper we introduce the SHOP2 planning algorithm, which has the following properties:

- Like SHOP, SHOP2 is a sound and complete HTN planning algorithm that generates the steps of each plan in the same order that those steps will later be executed. Thus, like SHOP, SHOP2 knows the current state at each step of the planning process.
- Unlike SHOP, SHOP2 allows each method to decompose into a partially ordered set of subtasks, and allows the creation of plans that interleave subtasks from different tasks.
- SHOP2 is upward-compatible with SHOP. Our SHOP2 implementation can run SHOP knowledge bases with only minor syntactical changes, and in fact runs them more efficiently than SHOP does.

We have done experimental comparisons of SHOP and SHOP2 in problem domains exemplifying situations (1) where domain-specific global-reasoning knowledge seems necessary for efficient plan-generation regardless of SHOP's total-ordering requirement, and (2) where such knowledge is necessitated only by SHOP's partial-ordering requirement. In the latter case, we could easily create a knowledge base much simpler than SHOP's, that enabled SHOP2 to create plans more efficiently than SHOP and with plan quality similar to SHOP's.

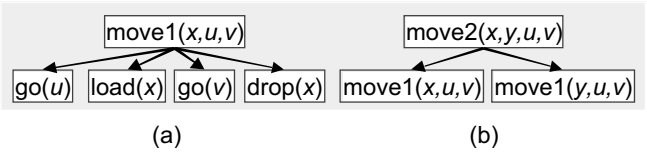


Figure 1. First set of methods for moving packages.

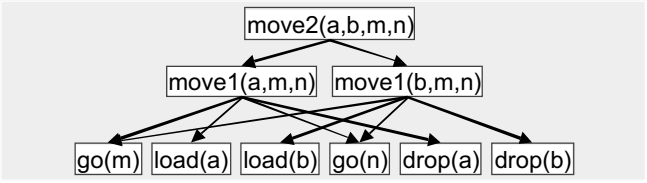


Figure 2: The plan that we want for two packages.

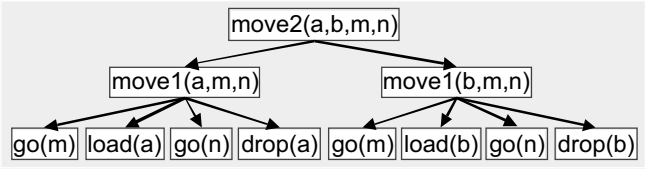


Figure 3: The plan we have actually told SHOP to generate.

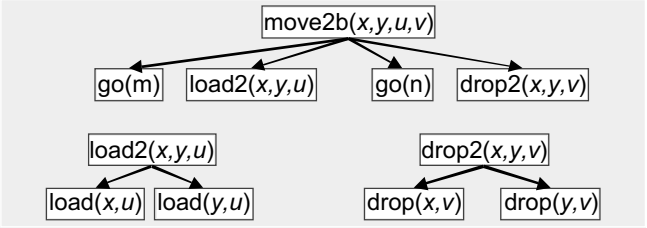


Figure 4. Second set of methods to move two packages.

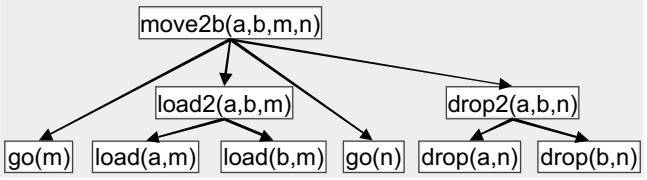


Figure 5. Plan generated using the methods in Figure 4.

2 Motivation

As example of the kind of difficulty that can result from SHOP's requirement that all tasks be totally ordered, consider the task of moving a package from one location to another. For SHOP to generate a plan for this task, it needs to have a method telling it how to move a package, such as the method shown in Figure 1(a). This method says that one way to move a package is to go to the package, pick the package up, go to the destination, and drop the package off.

If all we want to do is to deliver one package, then the method in Figure 1(a) will work fine. However, suppose we want to move two packages at once. Although the two methods shown in Figure 1 might at first glance seem satisfactory for this task, they will not always do what we want. If the two packages both have the same initial location and the same destination, then we probably would like to deliver both packages at once, as shown in Figure 2—but the methods in Figure 1 will tell SHOP to deliver the packages one at a time, as shown in Figure 3.

In this simple example, it is not hard to write methods telling SHOP to generate a plan for delivering both packages at once. For example, the methods shown in Figure 4 will tell SHOP to generate the plan shown in Figure 5. However, to do this we had to tell SHOP explicitly how to reason about both packages at once.

The need to give SHOP such "global planning" instructions can occur frequently. Each of the SHOP knowledge bases described in [Nau *et al.*, 1999] contains instructions for reasoning globally about the planning problem—and the same is true in most of the knowledge bases that we created during the AIPS-2000 planning competition. To write and debug such instructions can require significant time and effort.

3 SHOP2

3.1 Preliminaries

As with most AI planners, a logical atom in SHOP2 consists of a predicate name followed by a list of arguments, and a state of the world consists of a collection of ground logical atoms.

As with most HTN planners, a task in SHOP2 consists of a task name followed by a list of arguments. Although tasks are similar syntactically to logical atoms, they are different semantically since they denote activities that need to be performed rather than logical conditions [Erol *et al*, 1994; Barret, 1997].

A SHOP2 knowledge base contains domain-specific knowledge that SHOP2 will need in order to do planning in some domain. It consists of axioms, methods, and operators, as described below.[1]

Axioms. SHOP2's Horn-clause axioms are identical to those used in SHOP. As in SHOP, SHOP2 uses these axioms to infer whether a method's preconditions are satisfied in the current state of the world, using Horn-clause inference.

Also like SHOP, SHOP2's methods and Horn clauses can contain calls to the Lisp evaluator. This allows SHOP2 to evaluate preconditions that contain, for example, numeric computations or queries to external information sources.

Methods. SHOP2's methods are similar to those of SHOP except that methods can produce partially ordered sets of subtasks. As in SHOP, the basic form of a method is

[1] SHOP2 generalizes the M-SHOP algorithm described in [Nau *et al.*, 2000]. M-SHOP takes partially ordered task lists as input—but like SHOP, its methods must produce totally ordered sets of subtasks. In contrast, SHOP2 allows methods to produce partially ordered sets of subtasks.

(:method *task-atom precondition-atoms decomposition*)

The *task-atom* tells what kind of task the method can be used for. The method cannot be applied to some task *t* unless *task-atom* is unifiable with *t*).

The *precondition-atoms* tell what things must be true in the current state of the world in order for the method to be applicable to *t*. The previous paragraphs described how SHOP2 infers whether these preconditions are true in the current in the current state.

The *decomposition* tells what subtasks to decompose the task into. In SHOP this set of subtasks was totally ordered, but in SHOP2 it can be partially ordered.

As in SHOP, additional preconditions and decompositions can be appended to the method for SHOP2 to use in an "if-then-else" manner:

(:method
task-atom
precondition-atoms-1
decomposition-1
precondition-atoms-2
decomposition-2 ...)

The idea here is that if *precondition-atoms-1* is true then the method will produce *decomposition-1*; otherwise if *precondition-atoms-2* is true then the method will produce *decomposition2*, and so forth.

Operators. Since the subtasks of a method can be partially ordered, this means that subtasks of different methods can be interleaved in a plan. Thus in order to prevent deleted-condition interactions, we need a way to specify protected conditions. However, since SHOP2 will plans for tasks in the order that they will later be executed, we can get by with a rather simple protection mechanism, rather than the more sophisticated mechanisms used in partial-order HTN planners such as O-PLAN [Currie and Tate, 1991; Tate, 1994], SIPE [Wilkins, 1990], and UMCP [Erol *et al.*, 1994]. To accomplish this, SHOP2's operator syntax is modified beyond that of SHOP, to include the following way to specify protected conditions.

Like SHOP's operators, each SHOP2 operator includes a task atom (which must be unifiable with a task in order for the operator to be applicable to that task), a "delete list" (which tells what atoms to delete from the current state), and an "add list" (which tells what atoms to add to the current state). However, SHOP2 operators can also include *protection requests* (to tell SHOP2 that certain conditions should not be deleted) and *protection cancellations* (to tell SHOP2 that it is now permissible to delete those conditions).

For example, suppose we want to tell SHOP2 to drive a truck from location *p* to location *q*, and prevent the truck from being moved away from *q*. Then we would write an operator that deletes "(at-truck *p*)", adds "(at-truck *q*)", and adds a protection request for "(at-truck *q*)". Once we are ready to move the truck, another operator can delete the protection request.

3.2 The SHOP2 Algorithm

The SHOP2 planning algorithm is as follows, where *S* is the current state, *M* is a partially ordered set of tasks, and *L* is a list of protected conditions:

procedure SHOP2(*S*,*M*,*L*)
  if *M* is empty **then return** NIL **endif**
  nondeterministically choose a task *t* in *M* that has no predecessors
  <*r*,*R'*> = reduction(*S*,*t*)
  if *r* = FAIL **then return** FAIL **endif**
  nondeterministically choose an operator instance *o* applicable to *r* in *S*
  S' = the state produced from *S* by applying *o* to *r*
  L' = the protection list produced from *L* by applying *o* to *r*
  M' = the partially ordered set of tasks produced from *M* by replacing *t* with *R'*
  P = SHOP2(*S'*,*M'*,*L'*)
  return cons(*o*,*P*)
end SHOP2

procedure reduction(*S*,*t*)
  if *t* is a primitive task **then return** <*t*,NIL>
  else if no method is applicable to *t* in *S* **then return** <FAIL,NIL> **endif**
  nondeterministically let *m* be any method applicable to *t* in *S*
  R = the decomposition (partially ordered set of tasks) produced by *m* from *t*
  r = any task in *R* that has no predecessors
  <*r'*,*R'*> = reduction(*S*,*r*)
  if *r'* = FAIL **then return** <FAIL,NIL> **endif**
  R'' = the partially ordered set of tasks produced from *R* by replacing *r* with *R'*
  return <*r'*,*R''*>
end reduction

The proof that the SHOP2 algorithm is both sound and complete is too long to include in this paper. However, it is a relatively straightforward induction proof, proceeding from the usual kind of definition of what it means for something to be an HTN plan.

Probably the only complicating factor worth mentioning here is the reason why REDUCE calls itself recursively until it finds a primitive task. This is needed in order to ensure that for all of the methods used to produce that primitive task, the preconditions are evaluated in the correct state of the world.

4 Implementation and Experiments

We implemented the SHOP2 algorithm by modifying the Common-Lisp coding for the SHOP planning system. As we did with SHOP, we intend to make SHOP2 available as freeware under the GNU public license.

For our experiments, we wanted to compare SHOP2 and SHOP on problem domains exemplifying two different cases for the role of domain-dependent "global reasoning" knowledge:

- Cases where such knowledge is somehow an intrinsic requirement for generating plans efficiently. We chose the blocks world [Nilsson, 1980] as a problem domain where such cases might be likely to occur.
- Cases where such knowledge is necessitated only by SHOP's total ordering requirement. As a domain in which such cases would be likely, we chose the logistics domain [Veloso, 1992].

For our comparisons, we built knowledge bases for SHOP2 in the logistics and blocks-world domains, and compared them to the knowledge bases that come with the SHOP code. For our tests we used a 400-MHz Power Macintosh G4 with 256 MB of RAM, using Macintosh Common Lisp 4.3.

In order to do these comparisons properly, one concern was that in modifying the SHOP code to create the SHOP2 code, we also made some significant optimizations. Because of these optimizations, SHOP2 runs SHOP knowledge bases faster than SHOP does, so we felt that running the SHOP2 code against the SHOP code would be unfairly favorable to SHOP2.

To solve this problem, we utilized SHOP2's upward-compatibility with SHOP. With some minor syntactical changes, any SHOP knowledge base will run in SHOP2—and running such a knowledge base in SHOP2 is equivalent to running the SHOP planning algorithm. Thus for our tests, we used SHOP2 to run both the SHOP knowledge base and the SHOP2 knowledge base.

4.1 Logistics Problems

The SHOP knowledge base contains a complicated set of instructions that tell SHOP how to reason globally about the logistics-planning domain. We could not figure out a way to make any significant simplifications to this knowledge base and still have it run in SHOP, except by removing the domain knowledge and forcing SHOP to do a brute-force search. However, it was relatively easy for us to create a much simpler SHOP2 knowledge base, consisting of instructions for how to transport an individual package to its destination. As can be seen from Table 1, this resulted in a SHOP2 knowledge base whos size was only about 26% of the size of the SHOP knowledge base.

We compared the SHOP knowledge base with the SHOP2 knowledge base on 110 randomly generated logistics problems. The problems involved *N* packages to be delivered, for *N* = 10, 15, …, 60. There were 10 problems for each *N*, for a total of 110 problems. In each problem, the number of cities was no larger than N/2. Each city contained three locations, one truck, and *N*/5 or fewer airports. For each package, the original location and the destination location were randomly chosen and were guaranteed to be different from each other.

As shown in Figure 1, SHOP2 ran about 4 times as fast with the SHOP2 knowledge base as it did with the SHOP knowledge base. As shown in Figure 2, the two knowledge bases created plans of nearly the same size, but the ones generated by the SHOP knowledge base were slightly shorter.

Table 1: Sizes of the SHOP and SHOP2 knowledge bases for the logistics domain (counting each if-then decomposition of a method as a separate method).

	SHOP knowledge base	SHOP2 knowledge base
Methods	50	10
Operators	7	7
Axioms	10	1

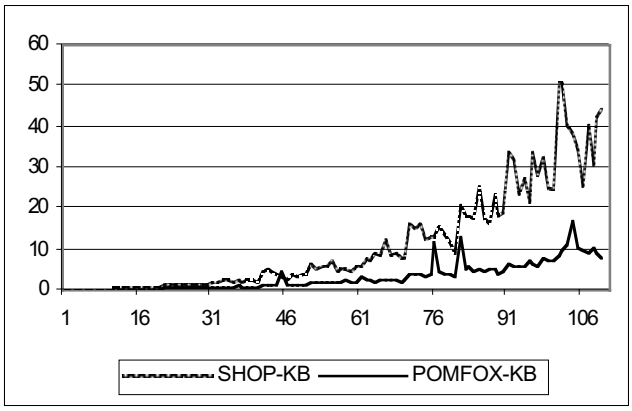


Figure 1. CPU times for SHOP2 using the SHOP knowledge base and the SHOP2 knowledge base, on 110 randomly generated logistics problems. The x-axis gives the problem number, and the y-axis gives the CPU time.

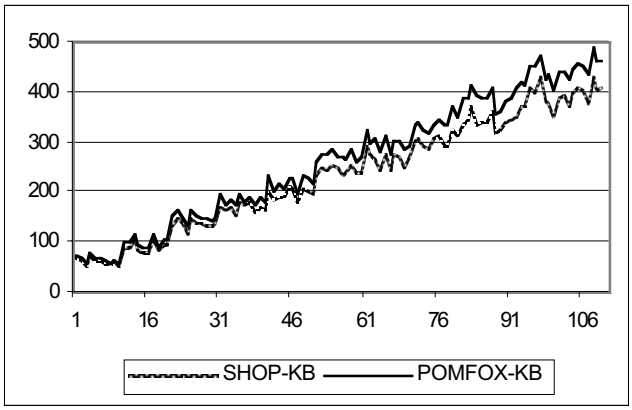


Figure 2. Sizes of the plans of Figure 1. The x-axis gives the problem number, and the y-axis gives the number of actions.

Table 2: Sizes of the SHOP and SHOP2 knowledge bases for the blocks world (counting each if-then decomposition of a method as a separate method).

	SHOP knowledge base	SHOP2 knowledge base
Methods	10	13
Operators	7	8
Axioms	1	5

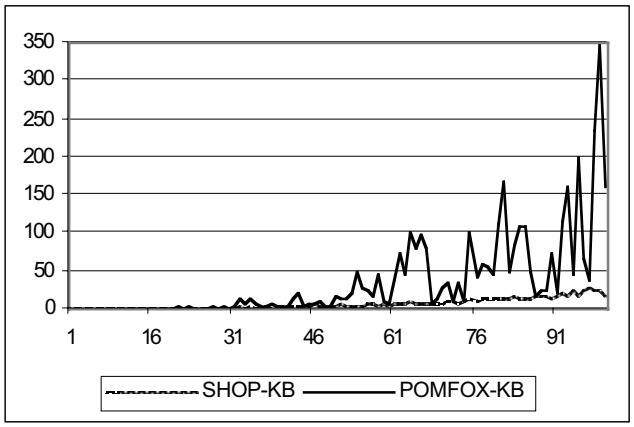


Figure 3. CPU times for SHOP2 using the SHOP knowledge base and the SHOP2 knowledge base, on randomly generated blocks-world problems. The x-axis gives the problem number, and the y-axis gives the CPU time.

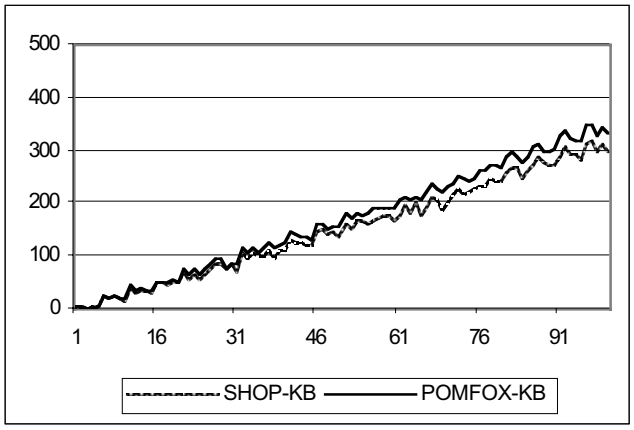


Figure 4. Sizes of the plans of Figure 3. The x-axis gives the problem number, and the y-axis gives the number of actions.

4.2 Blocks-World Problems

Just as before, the blocks-world knowledge base for SHOP contains instructions for how to reason globally about the planning process. Just as before, we could not think of any way to significantly reduce the size of this knowledge base and still have it run in SHOP, other than by removing the domain knowledge and forcing SHOP to do a brute-force search.

As before, our objective for SHOP2 was to simplify the knowledge base giving SHOP2 instructions for how to move individual blocks. However, we could not figure out any way to do this without forcing SHOP2 to do a brute-force search. In fact, we suspect that "global" domain-specific algorithms such as the ones discussed in [Chenoweth, 1991; Gupta and Nau, 1991] may be the only way to achieve efficient planning in the blocks world.

We tried creating a SHOP2 knowledge base by removing some of the total-ordering constraints and bookkeeping operations from the SHOP knowledge base, but this required us to add additional coding such as protection requests and protection cancellations to handle interleaving correctly. As shown in Table 2, the resulting knowledge base was about 44% larger than the original SHOP knowledge base.

We compared the performance of the two knowledge bases on randomly blocks-world problems consisting of N= 5,10,…,100 blocks to be relocated. We generated five problems for each value of N, for a total of 100 problems. To build the initial and final states, we generated configurations of blocks as follows:

- First, put a block onto the table (thereby creating a new tower).
- For each block after the first one, if t is the number of existing towers, then there are t+1 possible locations for the new block: on top of any of the existing towers, or on the table (thereby creating a new tower). Choose one of those locations at random, with an equal probability for each choice.

As shown in Figure 3, the time taken by SHOP2 using the SHOP2 knowledge base varied greatly from problem to problem. On the average, SHOP2 needed about 2.4 times as much time to generate plans with this knowledge base as it did with the SHOP knowledge base. As shown in Figure 4, the two knowledge bases created plans of nearly the same size, but the ones generated by the SHOP knowledge base were slightly shorter.

Discussion and Conclusions

We have described a new HTN planning algorithm, the SHOP2 algorithm. Like the SHOP planning algorithm, the SHOP2 algorithm generates the steps of a plan in the same order in which those steps are to be executed—but unlike SHOP, SHOP2 allows the subtasks of each method to be partially ordered.

SHOP2 runs SHOP knowledge bases faster than SHOP does; and our test results show that in some cases one can create knowledge bases for SHOP2 that are much simpler than the ones needed by SHOP yet still allow SHOP2 to run more quickly than SHOP.

We believe that the primary impact of our results is to provide a way to obtain the same advantages ascribed to the SHOP planning algorithm, while alleviating one of SHOP's primary drawbacks. Below, we summarize what those advantages and drawbacks are:

1. Planning for tasks in the order that those tasks will be performed makes it possible to know the current state of the world at each step in its planning process, which makes it possible to incorporate significant reasoning power into the planner's precondition-evaluation mechanism. Rather than just unifying preconditions against current-state atoms, the SHOP2 system (like the SHOP system) can perform Horn-clause inferences to evaluate preconditions that are not directly mentioned

in the current state, and its preconditions can incorporate calls to the Lisp evaluator (e.g., to do numeric computations or make queries to external sources of information).

2. The combination of HTN decomposition (to focus the search on the goal) and reasoning power in the preconditions (to prune inapplicable methods and operators from the search space) makes it possible to write domain-dependent knowledge bases that provide very efficient planning performance. As an illustration of what this means, when we tried to run Blackbox and IPP on the suites of logistics problems and blocks-world problems described above, we could not get them to solve any of the problems in the test suites. In each case, either they ran out of memory or else we had to terminate them after they had run for more than 30 minutes of CPU time without finding solutions.

3. The primary drawback of any HTN planning system is the effort needed to create a knowledge base of domain-dependent information for the domain we want it to do planning in. In SHOP, this drawback is sometimes worsened by SHOP's restriction that the subtasks of each method must be totally ordered, because this can require the knowledge-base author to introduce global reasoning into the planning domain that would not otherwise be needed. Our experimental results suggest that in these cases, SHOP2 can plan more efficiently than SHOP using knowledge bases much simpler than those needed by SHOP. In cases where such knowledge bases cannot be created, SHOP2 can run SHOP knowledge bases quicker than SHOP.

Some of our topics for future work include the following: investigating additional ways to make the SHOP2 algorithm more powerful and easier to use, releasing the coding for SHOP2 as open-source software, and using the SHOP2 algorithm as an embedded planning algorithm in a real-world application.

Acknowledgements

This work was supported in part by the following grants, contracts, and awards: AFRL F306029910013 and F30602-00-2-0505, ARL DAAL0197K0135, and U. of Maryland General Research Board. Opinions expressed here do not necessarily reflect those of the funders.

References

[Barrett, 1997] A. Barrett. *Frugal Hierarchical Task-Network Planning*. Computer Science and Engineering Dept., University of Washington, 1997.

[Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. "Using Temporal Logics to Express Search Control Knowledge for Planning,." *Artificial Intelligence*, 116(1-2):123-191, January, 2000.

[Chenoweth, 1991] S. V. Chenoweth. On the {NP}-hardness of blocks world. *AAAI-91*, July 1991, pages 623–628.

[Currie and Tate, 1991] K. Currie and A. Tate. O-Plan: The Open Planning Architecture. *Artificial Intelligence*, 52(1):49-86, 1991.

[Erol *et al.*, 1994] K. Erol, J. Hendler and D. S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In *Proc. Second International Conf. on AI Planning Systems (AIPS-94)*, June, 1994, pages 249-254.

[Gupta and Nau, 199] N. Gupta and D. S. Nau. Complexity Results for Blocks-World Planning. *AAAI-91*, July 1991, pages 629-633.

[Kautz and Selman, 1999] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. *IJCAI-99*, 1999, pages 318-325.

[Koehler *et al.*, 1997] J. Koehler, B. Nebel, J. Hoffman and Y. Dimopoulus. Extending planning graphs to an ADL subset. In *Proc. ECP-97*, 1997. Toulouse, France.

[Munoz-Avila *et al.*, 2000] H. Munoz-Avila, D. W. Aha, L. A. Breslow, D. S. Nau and R. Weber. Integrating Conversational Case Retrieval with Generative Planning. In *EWCBR-2000*, 2000. Trento, Italy: Springer-Verlag.

[Nau *et al.*, 1999] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *IJCAI-99*, 1999.

[Nau *et al.*, 2000] D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP and M-SHOP: Planning with Ordered Task Decomposition. Tech report TR 4157, University of Maryland, College Park, MD, June 2000.

[Nilsson, 1980] N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

[Tate, 1994] A. Tate. Mixed Initiative Planning in O-Plan2. In *Proceedings of the ARPA/Rome Laboratory Knowledge-Based Planing and Scheduling Initiative*, 1994, pages 512-516. Tuscon, AR: Morgan Kaufmann.

[Veloso, 1992] M. Veloso. "Learning by Analogical Reasoning in General Problem Solving." Tech. Report CMU-CS-92-174, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.

[Wilkins, 1990] D. Wilkins. "Can AI Planners Solve Practical Problems?" *Computational Intelligence*, 6(4):232-246, 1990.

Conditional progressive planning under uncertainty

Lars Karlsson
Center for Applied Autonomous Sensor Systems
Örebro University, SE-701 82 Örebro, Sweden
http://www.aass.oru.se/
lars.karlsson@tech.oru.se

Abstract

In this article, we describe a possibilistic/probabilistic conditional planner called PTLplan. Being inspired by Bacchus and Kabanza's TLplan, PTLplan is a progressive planner that uses strategic knowledge encoded in a temporal logic to reduce its search space. Actions effects and sensing can be context dependent and uncertain, and the information the planning agent has at each point in time is represented as a set of situations with associated possibilities or probabilities. Besides presenting the planner itself — its representation of actions and plans, and its algorithm — we also provide some promising data from performance tests.

1 Introduction

In this article, we describe a conditional planner called PTLplan. A conditional planner does not presuppose that there is complete information about the state of its environment at planning time, but that more information may be available later due to sensing, and that this new information can be used to make choices about how to proceed. Hence, a conditional planner generates plans that may contain conditional branches. In addition, PTLplan can handle degrees of uncertainty, either in possibilistic or in probabilistic terms. The "P" in PTLplan stands for exactly that.

PTLplan is a progressive (forward-chaining) planner; it starts from an initial situation and applies actions to that and subsequent resulting situations, until a situation where the goal is satisfied is reached. Being progressive, PTLplan has the disadvantage that the search space is potentially very large even for small problems The advantage is that PTLplan can reason from causes to effects, and always in the context of a completely specified situation. The latter makes it possible to apply a technique that can reduce the search space considerably: the use of strategic knowledge that helps pruning away unpromising plan prefixes.

PTLplan builds on a conditional planner ETLplan [Karlsson, 2001], which in turn built on a sequential (non-conditional) planner called TLplan [Bacchus and Kabanza, 1996; 2000] ("TL" stands for "temporal logic"). Two of the features that makes TLplan interesting are its fairly expressive first-order representation and its good performance due to its use of strategic knowledge. The latter is indicated by its outperforming most other comparable planners in empirical tests, as has been documented in [Bacchus and Kabanza, 2000]. ETLplan added two features to TLplan:

- The representation of actions and situations used in ETLplan permitted actions that have sensing effects, that is the agent may observe certain fluents (state variables).
- Based on these observations, the planning algorithm could generate conditional plans.

In addition to these features, PTLplan also incorporates:

- Assignments of degrees of uncertainty, either in possibilistic or probabilistic terms, to effects, observations and situations/states.
- Based on these degrees, measures of how likely a plan is to succeed or fail. In the absence of plans that are completely certain to succeed, PTLplan is still capable of finding plans that, although they may fail, are likely enough to succeed.

PTLplan is the first progressive planner utilizing strategic knowledge to incorporate the features mentioned above, and as indicated by tests, it does so successfully. In the rest of the article, we describe the plan representation of PTLplan, the use of strategic knowledge and the planning algorithm, and we also briefly provide some data from performance tests.

2 Related work

Planning in partially observable domains has been a hot topic since the mid 90's. There has been a good amount on work on POMDPs (partially observable Markov decision processes), see e.g. [Kaebling *et al.*, 1998]. POMDPs typically utilize explicit enumerations of states, but there are also results on the use of more compact propositional representations [Boutilier and Poole, 1996]. Note that this work is on the more general problem of generating policies that maximize utilities, and not plans with sequences and branches that achieve goals.

One of the earliest conditional probabilistic systems originating from classical planning was the partial-order planner C-BURIDAN [Draper *et al.*, 1994], which had a performance

which made in impractical for all but the simplest problems. The partial-order planner MAHINUR [Onder and Pollack, 1999] improved considerably on C-BURIDAN by handling contingencies selectively. MAHINUR focuses its efforts on those contingencies that are estimated to have the greatest impact on goal achievement. C-BURIDAN and MAHINUR are regressive planners: they perform means-ends reasoning from effects to causes.

WEAVER [Blythe and Veloso, 1997] is a system that iterates between a classical planner and a probabilistic evaluator. It can model stochastic exogenous events but not sensing.

C-MAXPLAN and ZANDER [Majercik and Littman, 1999] are two conditional planners based on transforming propositional probabilistic planning problems to stochastic satisfiability problems (E-MAJSAT and S-SAT, respectively). The latter can be solved in a highly efficient manner, in particular when certain pruning techniques are used. But, as pointed out in [Majercik and Littman, 1999], the translation from planning problems to satisfiability problems can lead to blow-ups in size and obscures the problem structure.

On the possibilistic side, there is Guéré's and Alami's conditional planner [Guéré and Alami, 1999], which also has been demonstrated to have a practical level of performance. Their planner is based on Graph Plan [Blum and Furst, 1995]. It assumes that sensing actions are perfectly reliable and do not have preconditions. Another Graph Plan derivative is Sensing Graph Plan (SGP) [Weld *et al.*, 1998], which can deal with incomplete information but has no means for quantifying uncertainty.

3 Representation

3.1 Uncertainty

PTLplan can represent uncertainty either using possibility theory [Dubois and Prade, 1988] or probability theory, simply by interpreting the connectives used as follows.[1]

Connective	Possibility	Probability	Comment
$\otimes$	min	$\cdot$	"and"
$\oplus$	max	$+$	"or"

Thus, PTLplan can be used both when one is only interested in the relative likelihood of different facts and outcomes, and when one has precise information about probabilities.

3.2 Syntax: Fluents

The state of the world is described in terms of fluents (state variables) and their values. A fluent-value formula has the form f=v, denoting that the fluent f has the value v; if the value is omitted, it is implicitly assumed to be `T` (true). Examples of fluent-value formulae are `door(d1)` and `robot-at(table1)=F`. A fluent formula is a logical combination of fluent-value-formulae using the standard connectives and quantifiers. A fluent-assignment formula has the form f`:=`v denoting that f is caused to have the value v; e.g. `robot-at(table1):=F`.

[1] The probabilistic versions of the connectives are justified by the fact that they are only applied to mutually exclusive and independent situations or branches and Markovian transitions.

3.3 Syntax: Actions

An action schema consists of a tuple $\langle a, P, R\rangle$ where a is the action name, P is a precondition (a fluent formula) and R is a set of result descriptions. Each result description $r \in R$ is a tuple $\langle C, p, E, O\rangle$ where

- C is a context condition (a fluent formula) that determines when (in what states) the result is applicable.
- p is the possibility or probability of the result.
- E is a set of fluent assignment formulae (f`:=T` or f`:=F`) which specifies the effects of the result. We let E^+ denote the fluents with positive assignments in E and we let E^- be those with negative ones.
- O is a set of fluents f, denoting that the current value of f is observed, and/or fluent-value-formulae f=v denoting that f is (correctly or incorrectly) observed to have the value v. These observations are assumed to be made by the agent executing the plan.

Note that restrictions apply to the p values; the sum of the p in applicable results must always be 1 for the probabilistic case, and the maximum of the p must always be 1 for the possibilistic case.

Example 1 The following are action schemas for the tiger scenario (the scenario is due to [Kaebling *et al.*, 1998]). There are two doors, one to the left (`ld`) and one to the right (`rd`). Behind one of them lurks a tiger, and behind the other there is a reward. The fluent `tiger(`d`)` stands for that the tiger is behind door d. Probabilities are used.

The agent can listen at the doors, and this gives a indication of behind which door the tiger lurks. Unfortunately, there is a 15% chance of error. This is an action that yields observations but no concrete effects.

act: `listen()`
pre: `true`

	context	*p*	*effects*	*observations*
res:	`(tiger(ld),`	`0.85,`	`{},`	`{tiger(ld)=T}),`
	`(tiger(ld),`	`0.15,`	`{},`	`{tiger(rd)=T}),`
	`(tiger(rd),`	`0.85,`	`{},`	`{tiger(rd)=T}),`
	`(tiger(rd),`	`0.15,`	`{},`	`{tiger(ld)=T})`

The agent can open one of the doors, which either leads to the reward (`rew`), or to tiger-induced death (`dead`).

act: `open(?d)`
pre: `door(?d)`

	context	*p*	*effects*	*observations*
res:	`(¬tiger(?d),`	`1.0,`	`{rew:=T},`	`{rew=T}),`
	`(tiger(?d),`	`1.0,`	`{dead:=T},`	`{dead=T})`

3.4 Semantics: Situations and epistemic situations

We use a model of knowledge and action which is based on the concepts of a situation and an epistemic situation. In short, a situation describes one possible state of the world at a given point in time, where time is defined in terms of what actions has occurred. An epistemic situation, or e-situation for short, is essentially a set of situations with associated possibilities/probabilities that describes the agent's knowledge at a point in time. In this PTLplan is similar

to e.g. C-BURIDAN [Draper *et al.*, 1994] which employs a probability distribution over states. A transition relation res over situations provides the temporal dimension; the transitions are due to applications of actions. The global possibility/probability of a situation s is denoted $p(s)$. This represents the possibility/probability that some given choice of actions (plan) will lead to s. The global possibility/probability $p(\overline{s})$ of an e-situation is computed from those of its constituents: $p(\overline{s}) = \bigoplus_{s \in \overline{s}} p(s)$. The local possibility/probability $p_l(s)$ of a situation s is obtained by normalizing relative to its containing e-situation $\overline{s}$: $p_l(s) = p(s)/p(\overline{s})$. This represents the possibility/probability the agent assigns to s inside $\overline{s}$.

The state $state(s)$ of a situation s is a mapping from fluents to values (i.e. a set of fluents), and the observation set $obs(s)$ is a set of fluent-value-pairs $\langle f, v \rangle$. We impose the constraint that all situations in an e-situation $\overline{s}$ should have the same observation set; we denote this set by $obs(\overline{s})$.

Example 2 The initial e-situation for the tiger scenario has no observations (*obs*= { }), but contains two situations with associated probabilities and states. In one situation where $p = 0.5$, the tiger is behind the left door, and in the other, where $p = 0.5$, it is behind the right door.

obs=	{ }
0.5:	{ tiger(ld) , door(ld) , door(rd) }
0.5:	{ tiger(rd) , door(ld) , door(rd) }

3.5 Semantics: results of actions

The results of an operator/action A in a situation is defined as follows: for each result description $\langle C_i, p_i, E_i, O_i \rangle$, if context condition C_i is true in s then there is a situation s' resulting from s (i.e. $res(s, s')$) where the effects in E_i occur (i.e. $state(s') = (state(s) \cup E_i^+) \setminus E_i^-$) and the observations in O_i are made (if $f \in O_i$ and f has value v in s, or if "f=v" $\in O_i$, then $\langle f, v \rangle \in obs(s')$). Finally, $p(s') = p(s) \otimes p_i$. For an epistemic situation $\overline{s}$, the result of an action A is determined by applying A to the different situations $s \in \overline{s}$ as above, and then partitioning the resulting situations into new e-situations according to their observation sets.

Example 3 We attempt to apply some different actions to the initial e-situation. Applying `open(ld)` yields the following two new e-situations ("..." stands for "`door(ld) , door(rd)`"):

obs=	{ dead=T }
0.5:	{ tiger(ld) , dead,... }

obs=	{ rew=T }
0.5:	{ tiger(rd) , rew,... }

Applying `listen()` to the initial e-situation yields the following two new e-situations:

obs=	{tiger(ld)=T}
0.425:	{ tiger(ld),... }
0.075:	{ tiger(rd),... }

obs=	{tiger(rd)=T}
0.425:	{ tiger(rd),... }
0.075:	{ tiger(ld),... }

Note how the resulting situations are partitioned into e-situations based on their observations.

3.6 Syntax: Conditional plans

Plans in PTLplan are conditional. This means that there can be points in the plans where the agent can choose between different ways to continue the plan depending on some explicit condition. Therefore, in addition to the sequencing plan operator (`;`), we introduce a conditional operator (`cond`). The syntax of a conditional plan is as follows:

plan ::= `success` | `fail` | *action* `;` *plan* |
`cond` *branch*`*`
branch ::= (*cond* : *plan*)
action ::= *action-name* (*args*)

A condition *cond* is a conjunction of fluent-value formulae. The conditions for a branch should be exclusive and exhaustive relative to the potential e-situations at that point in the plan. `Success` denotes predicted plan success, and `fail` denotes failure.

Example 4 The following is a plan which is a solution to the tiger scenario (see example 6 for how it could be generated).

```
listen();
cond(tiger(rd)=T : open(ld);
     cond(rew=T:success) (dead=T:fail))
     (tiger(ld)=T : open(rd);
     cond(rew=T:success) (dead=T:fail))
```

3.7 Semantics: Application of conditional plans

The application of a plan to an epistemic situation $\overline{s}_i$ results in a set of new e-situations. First, the application of an action a to $\overline{s}_i$ is defined as in section 3.5. Next, the application of a sequence $a\,;p$ is defined as applying the first action a to $\overline{s}_i$, and then the rest of the plan p to each of the resulting e-situations. Finally, the application of a conditional plan element `cond` $(c_1 : p_1) \ldots (c_n : p_n)$ is defined as an application of the branch p_j whose context condition c_j matches with $obs(\overline{s}_i)$ (there should only be one such branch).

This concludes how actions and plans are represented in PTLplan. Next, we proceed to investigate how strategic knowledge can be represented and used.

4 Strategic knowledge

In order to eliminate unpromising plan prefixes and reduce the search space, PTLplan (and TLplan and ETLplan before it) utilizes strategic knowledge. This strategic knowledge is encoded as expressions (search control formulae) in an extension of first-order linear temporal logic (LTL) [Emerson, 1990] and is used to determine when a plan prefix should not be explored further. One example could be the condition "never pick up an object and then immediately drop it again". If this condition is violated, that is evaluates to `false` in some e-situation, the plan prefix leading there is not explored further and all its potential continuations are cut away from the search tree. A great advantage of this approach is that one can write search control formulae without any detailed knowledge about how the planner itself works; it is sufficient to have a good understanding about the problem domain.

LTL is based on a standard first-order language consisting of predicate symbols, constants and function symbols and the

Algorithm *Progress(f,$\overline{s}$)*
Case:

1. $f = \mathcal{K}f_1 :\ f^+ := \begin{cases} \texttt{true} \text{ if } K \leq (1 - \bigoplus_{s \in S^-} p_l(s)) \\ \text{where } S^- = \{s \in \overline{s} | s \models \neg f_1\}, \\ \texttt{false} \text{ otherwise} \end{cases}$
2. $f = \mathcal{O}(f_1{=}v) :\ f^+ := \begin{cases} \texttt{true} \text{ if } \langle f_1, v\rangle \in obs(\overline{s}), \\ \texttt{false} \text{ otherwise} \end{cases}$
3. $f = \mathcal{G}f_1 :\ f^+ := \begin{cases} \texttt{true} \text{ if } s \models f_1 \text{ for all } s \in G, \\ \texttt{false} \text{ otherwise} \end{cases}$
4. $f = f_1 \wedge f_2 :\ f^+ := \ Progress(f_1, \overline{s}) \wedge Progress(f_2, \overline{s})$
5. $f = \neg f_1 :\ f^+ := \neg Progress(f_1, \overline{s})$
6. $f = \bigcirc f_1 :\ f^+ := f_1$
7. $f = f_1 \mathcal{U} f_2 :\ f^+ := \ Progress(f_2, \overline{s}) \vee (Progress(f_1, \overline{s}) \wedge f)$
8. $f = \Diamond f_1 :\ f^+ := Progress(f_1, \overline{s}) \vee f$
9. $f = \Box f_1 :\ f^+ := Progress(f_1, \overline{s}) \wedge f$
10. $f = \forall x[f_1] :\ f^+ := \bigwedge_{c \in U} Progress(f_1(x/c), \overline{s})$
11. $f = \exists x[f_1] :\ f^+ := \bigvee_{c \in U} Progress(f_1(x/c), \overline{s})$

Return f^+

Figure 1: The PTLplan progression algorithm.

usual connectives and quantifiers. In addition, there are four temporal modalities: $\mathcal{U}$ (until), $\Box$ (always), $\Diamond$ (eventually), and $\bigcirc$ (next). In LTL, these modalities are interpreted over a sequence of situations, starting from the current situation. For the purpose of PTLplan, we can interpret them over a sequence/branch of epistemic situations $B = \langle \overline{s}_1, \overline{s}_2, \ldots \rangle$ and a current epistemic situation $\overline{s}_i$ in that sequence. The expression $\phi_1 \mathcal{U} \phi_2$ means that ϕ_2 holds in the current or some future e-situation, and in all e-situations inbetween ϕ_1 holds; $\Box\phi$ means that ϕ holds in this and all subsequent e-situations; $\Diamond\phi$ means that ϕ holds in this or some subsequent e-situation; and $\bigcirc\phi$ means that ϕ holds in the next e-situation $\overline{s}_{i+1}$.

In addition to the temporal modal operators from LTL, PTLplan also uses a goal operator $\mathcal{G}$, which is useful for referring to the goal in search control formulae. We let $\mathcal{G}\varphi$ denote that it is among the agent's goals to achieve the fluent formula φ. Semantically, this modality will be interpreted relative to a set of goal states G, i.e. the set of states that satisfy the goal. We also introduce two new modal operators: $\mathcal{K}\varphi$ ("knows") means that the necessity/probability that the fluent formula φ is true in the current e-situation exceeds some prespecified threshold K, given the information the agent has; and $\mathcal{O}f{=}v$ denotes that f is observed to have the value v in the current e-situation. We restrict fluent formulae to appear only inside the $\mathcal{K}$, $\mathcal{G}$ and (for fluent-value formulae) $\mathcal{O}$ operators.

We can now define the semantics of these modal operators relative to a branch B of epistemic situations, a current epistemic situation $\overline{s}_i$, a variable assignment V, and a set of goal states G, as follows.

- $(B, \overline{s}_i, V, G) \models \phi_1 \mathcal{U} \phi_2$ iff there exists a $j \geq i$ such that $(B, \overline{s}_j, V, G) \models \phi_2$ and for all k such that $i \leq k < j$, $(B, \overline{s}_k, V, G) \models \phi_1$.
- $(B, \overline{s}_i, V, G) \models \Box\phi$ iff for all $j \geq i$, $(B, \overline{s}_j, V, G) \models \phi$.

Algorithm PTLplan($\overline{s}, f, g, A, \sigma$)

1. If $\overline{s} \models g$ then return $\langle \texttt{success}, p(\overline{s}), 0\rangle$.
2. Let $f^+ :=$ *Progress*$(f, \overline{s})$;
   if $f^+ = \texttt{false}$ then return $\langle \texttt{fail}, 0, p(\overline{s})\rangle$.
3. For the actions $a_i \in A$ whose preconditions are satisfied in all $s \in \overline{s}$ do:
   a. Let $S^+ = Apply(a_i, \overline{s})$.
   b. For each $\overline{s}^+_j \in S^+$,
      let $\langle P'_j, succ'_j, fail'_j\rangle :=$ PTLplan($\overline{s}^+_j, f, g, A, \sigma$).
   c. If $(\bigoplus_j fail'_j) \geq 1 - \sigma$ then
      let $cont_i = \langle \texttt{fail}, 0, p(\overline{s})\rangle$.
   d. If $|S^+| = 1$, let $cont_i = \langle a_1 \texttt{ ; } P'_1, succ'_1, fail'_1\rangle$.
      Otherwise let $cont_i = \langle P_i, succ_i, fail_i\rangle$ where
      $P_i = a_i \texttt{ ; (cond } (c'_1 : P'_1) \ldots (c'_n : P'_n))$,
      $succ_i = (\bigoplus_j succ'_j)$, $fail_i = (\bigoplus_j fail'_j)$
      and each $c'_i = \bigwedge\{f{=}v \mid \langle f, v\rangle \in obs(\overline{s}^+_i)\}$.
4. Return the $cont_i = \langle P_i, succ_i, fail_i\rangle$ with the lowest $fail_i$, provided $fail_i < (1 - \sigma)$, or otherwise return $\langle \texttt{fail}, 0, p(\overline{s})\rangle$.

Figure 2: The PTLplan planning algorithm.

- $(B, \overline{s}_i, V, G) \models \Diamond\phi$ iff there exists a $j \geq i$ such that $(B, \overline{s}_j, V, G) \models \phi$.
- $(B, \overline{s}_i, V, G) \models \bigcirc\phi$ iff $(B, \overline{s}_{i+1}, V, G) \models \phi$.
- $(B, \overline{s}_i, V, G) \models \mathcal{G}\varphi$ iff for all $s \in G$, $(s, V) \models \varphi$.
- $(B, \overline{s}_i, V, G) \models \mathcal{K}\varphi$ iff $K \leq 1 - (\bigoplus_{s \in S^-} p_l(s))$ where $S^- = \{s \in \overline{s}_i | (s, V) \models \neg\varphi\}$.
- $(B, \overline{s}_i, V, G) \models \mathcal{O}f{=}v$ iff $\langle f, v\rangle \in obs(\overline{s}_i)$.

The interpretation of $\mathcal{K}\varphi$ is motivated by the fact that in possibility theory, necessity is defined as $Nec(\varphi) = 1 - Pos(\neg\varphi)$, and in probability theory $P(\varphi) = 1 - P(\neg\varphi)$.

In order to efficiently evaluate control formulas, PTLplan incorporates a progression algorithm (similar to the ones of TLplan and ETLplan) that takes as input a formula f and an e-situation and returns a formula f^+ that is "one step ahead", i.e. corresponds to what remains to evaluate of f in subsequent e-situations. It is shown in figure 1. (The goal states G in case 3 are fixed for a given planning problem and need not be passed along.) Note that the algorithm assumes that all quantifiers range over a finite universe U.

Example 5 The following is a control formula stating that a robot should never pick an object up and then drop it immediately again, i.e. hold it only one moment.

$$\Box\neg(\mathcal{K}(\neg\exists x[\texttt{robot-holds}(x)]) \wedge \bigcirc(\mathcal{K}(\exists x[\texttt{robot-holds}(x)]) \wedge \bigcirc\mathcal{K}(\neg\exists x[\texttt{robot-holds}(x)]))) \quad (1)$$

5 The planning algorithm

PTLplan is a progressive planner, which means that it starts from an initial e-situation and then tries to sequentially apply actions until an e-situation where the goal is satisfied is reached. The algorithm is shown in figure 2. It takes as input an e-situation $\overline{s}$, a search control formula f, a goal formula g (with a $\mathcal{K}$), a set of actions A and a success threshold σ (the failure threshold is $1 - \sigma$). It returns a triple

$\langle plan, succ, fail\rangle$ containing a conditional plan, a degree (possibility/probability) of success, and a degree of failure. It is initially called with the given initial e-situation and a search control formula.

Step 1 checks if the goal is satisfied in $\overline{s}$, if this is the case it returns the possibility/probability $p(\overline{s})$ of $\overline{s}$. Step 2 progresses the search control formula; if it evaluates to `false`, the plan prefix leading to this e-situation is considered unpromising and is not explored further. In step 3, we consider the different applicable actions. For each such action, we obtain a set of new e-situations (a). We then continue to plan from each new e-situation separately (b). For those subplans thus obtained, if the combined possibility/probability of failure is above the failure threshold, we simply return a fail plan (c). If there was a single new e-situation, we return a simple sequence starting with the chosen action (d). Otherwise, we add to the chosen action a conditional plan segment where branches are constructed as follows. The conditions are derived from the different observations of the different e-situations, and the sub-plans are those obtained in (b). The success and failure degrees of this new plan are computed by combining those of the individual sub-plans. In step 4, finally, we return the best of those plans (i.e. the one with the least failure degree) found in step 3, or a fail plan if the degree of failure is too high.

Example 6 To give a feel for the operation of PTLplan, we go through the tiger scenario. The actions are defined as in example 1, and the initial e-situation is as in example 2. There is one search control formula:[2]

$$f = \Box(\mathcal{K}(\neg \texttt{dead})) \tag{2}$$

Finally, the goal is:

$$g = \mathcal{K}(\texttt{rew} \wedge \neg \texttt{dead}) \tag{3}$$

This goal should be achieved with a probability of at least $\sigma = 0.8$, giving a failure threshold of 0.2.

1. From the initial e-situation, the applicable actions are `open(ld)`, `open(rd)` and `listen()`.
2. We first apply `open(ld)`, resulting in the e-situations described in the first half of example 3.

   (a) Continuing planning from the first of these e-situations, the condition $\Box(\mathcal{K}(\neg \texttt{dead}))$ after progression evaluates to false, so this branch yields $\langle \texttt{fail}, 0, 0.5\rangle$ (plan, success degree,failure degree); see step 2 in the algorithm.

   (b) In the second one, the goal is achieved, yielding $\langle \texttt{success}, 0.5, 0\rangle$.

   Thus, the combined probability of failure is $0 \oplus 0.5 = 0.5$, which is above the failure threshold (step 3.c).
3. Applying `open(rd)` gives a similar outcome.
4. Applying `listen()` to the initial e-situation yields the two e-situations from the second half of example 3.

   (a) Continuing from the first new e-situation, we eventually find that choosing `open(rd)` leads to success with a degree of 0.425: $\langle$`open(rd); cond(rew=T : success)(dead=T : fail)`$, 0.425, 0.075\rangle$

   (b) Continuing from the second one, we find that choosing `open(ld)` leads to success with a probability of 0.425: $\langle$`open(ld); cond(rew=T : success)(dead=T : fail)`$, 0.425, 0.075\rangle$.

   Combining the branches from (a) and (b) and appending them to `listen()`, finally, yields the plan shown in example 4, with 0.85 and 0.15 as probabilities of success/failure.

[2]The same effect could be achieved by including ¬dead in the preconditions of the actions, but we encode it as a search control formula to have the opportunity to see one in action.

Problem	SC	No SC	Other planners	
Tiger 0.5	0.002	0.008	0.01	ZANDER
Tiger 0.85	0.008	0.05	0.02	d:o
Tiger 0.939	0.07	0.20	0.08	d:o
Coffee	0.05	11.82	2.73	MAHINUR
Ask-coffee	0.16	13.06	769.20	ZANDER
Cold-room	0.56	556.80	11.78	Guéré-Alami

Figure 3: Some solution times for different scenarios, in CPU seconds. SC = "search control formulae were used".

6 Implementation and experiments

PTLplan has been implemented in Allegro CommonLisp, and uses a breath first search method involving detection of previously visited situations. The performance of PTLplan has been tested on some scenarios encountered in the literature. All tests were performed on a 300 MHz Pentium II. Table 3 presents the results of the experiments. Note that the use of search control formulae has a considerable impact, in particular in the three last scenarios. We also include some results documented for some other planners [Majercik and Littman, 1999; Onder and Pollack, 1999; Guéré and Alami, 1999] in order to give some rough indication about the relative merits of PTLplan.[3]

The *tiger scenario* was solved for success probabilities $\sigma = 0.5, 0.85, 0.939$. Only the marginally helpful control formula in example 6 was used. The *coffee robot scenario* [Boutilier and Poole, 1996; Onder and Pollack, 1999] involves a robot that is to go to a cafe in order to buy coffee with cream and sugar, and then bring the coffee back to the office. If it is raining (50% chance) the robot has to bring an umbrella along, otherwise not. A conjunction of 6 search control formulae were used in this scenario. The solution plan had two branches; the longest one had 8 steps. In the *asking coffee robot scenario* [Majercik and Littman, 1999], the robot does not have to bother about cream and sugar, but has to ask the user whether he wants coffee (50% chance). The solution had 4 branches. The *cold-room scenario* is a more complex, possibilistic scenario from [Guéré and Alami, 1999] involving a poorly illuminated table with two test tubes. A robot

[3]Unfortunately, the lack of agreed-upon, standard scenarios for conditional probabilistic/possibilistic planning makes systematic comparisons difficult. Also note that differences in hardware, small variations in how the scenarios were encoded etc, could account for some of the differences.

should move the red tube to a second table in a cold-room. A conjunction of 8 search control formulae were used in this scenario, and one of them is shown in example 5. The solution plan involved going to a third well-illuminated table to ensure that the correct test tube was picked up. It had two branches; the longest one had 14 steps.

7 Conclusions and future work

In this paper, we have presented work on the planner PTLplan. It is a progressive planner which utilizes strategic knowledge to reduce its search space. We have described PTLplan's fairly rich representation: actions with context-dependent and uncertain effects and observations, and plans with conditional branches. The semantics of this representation is based on associating different possible situations with degrees of possibility or probability. We have also described the planning algorithm and how we utilize a temporal logic to encode search control knowledge that can be used to prune unpromising branches in the search tree. PTLplan is based on TLplan [Bacchus and Kabanza, 2000] and ETLplan [Karlsson, 2001]. The novel contribution of PTLplan is the introduction of uncertainty into the context of progressive planning with strategic knowledge.

In this paper, we have also given some brief but quite promising performance data on PTLplan, indicating that it compares favorably to other probabilistic/possibilistic conditional planners. Yet, more remains to be done, both in order to improve the performance of PTLplan, and to evalute it more systematically, in particular on larger scenarios. Finally, we are now working on integrating PTLplan with a robotic system capable of action and perception [Coradeschi and Saffiotti, 2001].

Acknowledgements

This work was funded by the Swedish KK foundation. Many thanks to Silvia Coradeschi and Alessandro Saffiotti for helpful suggestions, and to Stephen Majercik for answering questions about ZANDER.

References

[AAAI99, 1999] *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, Florida, 1999. AAAI Press, Menlo Park, California.

[Bacchus and Kabanza, 1996] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In M. Ghallab and A. Milani, editors, *New Directions in Planning*, pages 141–153. IOS Press, Amsterdam, 1996.

[Bacchus and Kabanza, 2000] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.

[Blum and Furst, 1995] A. Blum and M.L. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, 1995. Morgan Kaufmann.

[Blythe and Veloso, 1997] J. Blythe and M. Veloso. Analogical replay for efficient conditional planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Providence, Rhode Island, 1997. AAAI Press, Menlo Park, California.

[Boutilier and Poole, 1996] C. Boutilier and D. Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, 1996. AAAI Press, Menlo Park, California.

[Coradeschi and Saffiotti, 2001] S. Coradeschi and A. Saffiotti. Perceptual anchoring of symbols for action. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, 2001. Morgan Kaufmann.

[Draper *et al.*, 1994] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*. AAAI Press, Menlo Park, Calif., 1994.

[Dubois and Prade, 1988] D. Dubois and H. Prade. *Possibility theory — an approach to computerized processing of uncertainty*. Plenum Press, 1988.

[Emerson, 1990] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 16, pages 997–1072. MIT Press, 1990.

[Guéré and Alami, 1999] E. Guéré and R. Alami. A possibilistic planner that deals with nondeterminism and contingency. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Stockholm, 1999. Morgan Kaufmann.

[Kaebling *et al.*, 1998] L.P. Kaebling, M.L. Littman, and A.R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998.

[Karlsson, 2001] Lars Karlsson. Conditional progressive planning: a preliminary report. In Brian Mayoh, John Perram, and Henrik Hautop Lund, editors, *Proceedings of the Scandinavian Conference on Artificial Intelligence 2001*, 2001. To appear.

[Majercik and Littman, 1999] Stephen M. Majercik and Michael L. Littman. Contingent planning under uncertainty via stochastic satisfiability. In AAAI99 [1999], pages 549–556.

[Onder and Pollack, 1999] N. Onder and M.E. Pollack. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In AAAI99 [1999], pages 577–584.

[Weld *et al.*, 1998] D.S. Weld, C.R. Anderson, and D.E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, Wisconsin, 1998. AAAI Press, Menlo Park, California.

PLANNING

DOMAIN ANALYSIS FOR PLANNING

One action is enough to plan

Emmanuel Guéré and **Rachid Alami**
LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4 - France
e-mail : {Emmanuel.Guere, Rachid.Alami}@laas.fr

Abstract

We describe a new practical domain independent task planner, called ShaPer, specially designed to deal efficiently with large problems.

ShaPer performs in two steps. In the first step, executed *off-line* for a given *domain subclass*[1], ShaPer explores and builds a compact representation of the state space called the *shape* graph. The main contribution of ShaPer is its ability to "resist" to combinatorial explosion thanks to the manipulation of sets of similar state descriptions called *shapes*. The *shape* graph is then used by ShaPer to answer very efficiently to planning requests.

A first version of the planner has been implemented. It has been tested on several well known benchmark domains. The results are very promising when compared with the most efficient planners from AIPS-2000 competition.

1 Introduction

In the stream of research that aims to speed up practical task planners, we propose a new approach to efficiently deal with large problems. Even though task planners have made very substantial progress over the last years, they are still limited in their use. Indeed, they are sometimes overwhelmed by very simple or even trivial problems. Our motivation stems also from the fact that there are domains which heavily influence the "structure" of the task state space; learning such a "structure" will certainly help in building efficiently a solution for a problem of the *domain* $subclass$[1]. Our aim is to develop a domain independent planner that will exhibit and learn the "structure" of a given *domain subclass*. The main difficulty (and the key contribution) in this framework is to face the combinatorial problem of the task space exploration.

In order to solve larger problems, all planners in the literature try to prune the state space. To do this, there are several ways: a first solution consists in having a "good metrics" of the domain in order to guide the search (heuristics). The heuristics can be automatically computed (domain independent planning - e.g. [Bonet and Geffner, 1999; Hoffmann, 2000]) or given by the user (domain dependent planning - e.g. [Doherty and Kvarnstrom, 1999]).

A second way consists in restricting the search to a superset of accessible states from the initial state; such restriction allows to reduce the search space during the backward planning process (e.g. the graph expansion of GraphPlan [Blum and Furst, 1997]).

A third way involves the analysis of the domain "structure" like invariant (e.g. [Gerevini and Schubert, 1998]), type detection, problem decomposition or problem symmetries (e.g. [Fox and Long, 1998; 1999]).

We would like is to develop a new, complete and domain independent method to efficiently solve large problems. To do so, we both restrict the search space through a set of accessible states and deal with a large class of domain symmetries (represented by states which have the same *shape*[2]).

For instance, the ferry problem overwhelms classical planners because of the very high number of possible applicable actions (which may create many redundant states) even though moving one car or another has the same result for the goal. However, it appears clearly that the state space is highly redundant: there is only $2n + 1$ distinguishable *shapes* for n cars. Dealing with such *shapes* will certainly increase the planner capabilities by reducing the search space in a drastic manner.

ShaPer[3] is a new planner which is able to detect all different *shapes* of the state space for a given *domain subclass*. This method is composed of two steps: first the planner builds a *shape* graph of the *domain subclass*. This step is performed *off-line* and only once for a given *domain subclass*, *e.g.* ShaPer builds the *shape* graph for the *subclass* of ferry domain with 50 cars. Then, ShaPer is able to solve *on-line* any planning problem in this class by connecting two *shape* graphs. As we will see, the connection makes use of only **one action**; this ensures the efficiency of the solution extraction step.

The next section explains how to build such *shape* graph. Section 3 presents an expansion algorithm that guarantee

[1] In this paper, we call a *domain subclass* the set of planning problems defined by a set of operators and a set of objects. For example, the 10-blocks-world is a *subclass* of the blocks-world domain. The 20-blocks is another *subclass* of the same domain.

[2] The *shape* of a state, defined more precisely in the next section, can be viewed as a partially instantiated state pattern.

[3] *Sha*pe based *P*lann*er*

ShaPer completeness. We then describe the solution extraction process (section 4). A first version of the planner has been implemented and has produced very promising results. All sections are illustrated by examples obtained by running such an implementation. The last section presents and discusses a number of tests which are compared with some of the best current planners (from AIPS-2000 competition).

2 The Shape Graph: $\mathcal{G}$

A planning problem is usually expressed as a triple $(\mathcal{O}, \mathcal{I}, \mathcal{J})$ where $\mathcal{O}$ corresponds to the set of instantiated actions, $\mathcal{I}$ the initial state and $\mathcal{J}$ the goal. In the STRIPS [Fikes and Nilsson, 1971] formalism, an action $o \in \mathcal{O}$ is described by its *preconditions* P_o ($P_o \subseteq S$ means that o is applicable to the state S), its *Addlist* A_o and its *Dellist* D_o. Applying the operator o from state S results in the new state $o(S) \equiv (S - D_o) \cup A_o$. A state is an instantiated predicate set according to the closed-world assumption.

The purpose of this section is to present the *shape* graph construction process (only extract "relevant" states). This state space exploration is performed *off-line* and once only for each *domain subclass* from a valid state S_{begin}.

Indeed, if we restrict ourselves to the STRIPS framework, there is *a priori* no mean to check if a state is valid or not, except by applying a valid sequence of action to a valid one (S_{begin} - user-defined).

This is the reason why the *shape* graph is an accessibility graph $\mathcal{G}$. Now, the main difficulty comes from the combinatorial explosion of states and applicable actions; this is why we build a graph that only contains "relevant" states.

2.1 Relevant states

The relevance of a state is defined according to the state description of the current graph $\mathcal{G}$. A state S is relevant (i.e. it "augments" $\mathcal{G}$ with new information) iff there exists no $g \in \mathcal{G}$ such that g is a substitution σ of S: $\sigma(g) = S$ (i.e. the state g can be instantiated by a variable permutation σ). Consider for instance the two blocks-world states g and S:

with $g = \{Clear(A), On(A,B), OnTable(B), Clear(C), OnTable(C)\}$ and $S = \{Clear(A), OnTable(B), Clear(B), OnTable(C), On(A,C)\}$.

The state g where A is substituted to A, B to C and C to B is equal to S ($\sigma = \{A/A, C/B, B/C\}$). In this case, g and S are said to have the same *shape*[4].

When S has the same *shape* as g, we can conclude that all *shapes* accessible from g are also accessible from S: if P is a sequence of action applicable to g, then $\sigma(P)$ is applicable to S and $(\sigma(P))(S) = \sigma(P(g))$. Developing the state S is then not informative: S does not lead to a new *shape*.

[4]In order to prevent possible misunderstandings due to the previous figure, let us emphasize on the fact that a *shape* does not characterize a tower, but more generally the structure of a planning problem (i.e. there exists a substitution between S and g).

Build_Graph($\mathcal{O}$,S_{begin})
  $To\_Dev \leftarrow \{S_{begin}\}$
  $\mathcal{G} \leftarrow \{S_{begin}\}$
  While $To\_Dev \neq \phi$ **do**
    $s \leftarrow pop(To\_Dev)$
    For each $o \in \mathcal{O}$ such that $P_o \subseteq s$ **do**
      If $\exists g \in \mathcal{G}$ and $\exists \sigma$ such that $o(s) = \sigma(g)$ **Then**
        If $o(s) = g$ **Then**
          Add the edge (s, g) to $\mathcal{G}$
        Else
          Mark s in $\mathcal{G}$ with σ and g
      Else
        Add the vertex $o(s)$ and the edge $(s, o(s))$ to $\mathcal{G}$
        $To\_Dev \leftarrow To\_Dev \cup \{o(s)\}$

Table 1: Build the *shape* graph $\mathcal{G}$.

2.2 Building the shape graph $\mathcal{G}$

In order to reduce the graph size, we only develop relevant states. The algorithm, presented in table 1, is similar to a breadth-first search algorithm. Relevant states are sequentially developed by applying all possible actions o. The algorithm detects links to other substitutions as well as cycles in a given substitution (when $o(s) = g$); in this case, a new edge is added.

In the case of a non-identical substitution σ, we keep in s the name of the corresponding *shape* in $\mathcal{G}$ and σ. Owing to the relation $o(s) = \sigma(g)$, the accessibility from s to $o(s)$ can be, if necessary, quickly retrieved. Indeed, several substitutions may produce the same result because of the commutativity of the logical and, *i.e.* $\sigma(g) = \sigma'(g)$ does not imply $\sigma = \sigma'$.

2.3 An example

The graph construction algorithm is illustrated by figure 1 with an example from the gripper domain. The goal is to move 3 balls from table T_1 to table T_2. To do this, the robot is able to pick and place a ball and to move from a table to the other. The robot has two arms. The graph expansion begins with the valid state S_1 (all three balls are on T_1 and the robot is near T_1). Four actions are applicable to S_1: $Pick(X, T_1)$, $Pick(Y, T_1)$, $Pick(Z, T_1)$ and $move(T_1, T_2)$; as shown on figure 1 $Pick(X, T_1)(S_1) = S_2$ and $move(T_1, T_2)(S_1) = S_3$. We can note that $Pick(Y, T_1)(S_1)$ is not added to the graph, since there exists a substitution $\sigma = \{Y/X, X/Y, Z/Z, T_1/T_1, T_2/T_2, g_l/g_l, g_r/g_r\}$[5] such that $\sigma(S_2) = Pick(Y, T_1)(S_1)$. Similarly, there exists a substitution between $Pick(Z, T_1)(S_1)$ and S_2; we note it on the figure by a dashed line to S_2. Then S_2 is developed in S_4 and S_5 and so on ...

The graph $\mathcal{G}$ finally contains nine vertices which model all possible *shapes* accessible from the valid state S_1.

[5]Note that in this example, even through the substitution only permutes balls X, Y and Z, the substitution σ must contain g_l (gripper_left), g_r (gripper_right) and the two tables T_1 and T_2 because they are variables too. See for instance the state S_9 which has the same *shape* as the state S_8 when the robot takes a ball.

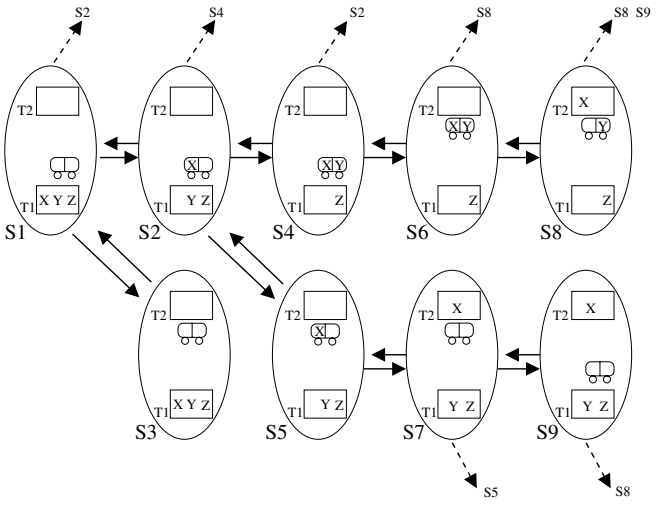


Figure 1: Build the *shape* graph $\mathcal{G}$.

2.4 $\mathcal{G}$ and the state space

This method makes it possible to drastically reduce the size of the state space when the domain contains many (functionally similar) "objects" (variables with possible substitutions). Indeed, in the blocks-world and in the gripper domain, the state space grows exponentially comparing to the *shape* graph size as shown table 2.

As mentioned previously, ShaPer needs a valid state, S_{begin}, to build the graph $\mathcal{G}$. Although $\mathcal{G}$ contains all *shapes* accessible from S_{begin}, in the case of a disjoint state space, ShaPer is unable to construct graphs from the other connected components of state space (their states are not accessible). Therefore, the user must give one state per connexity; otherwise, the *shape* of S_{init} (the initial state of a planning problem) might not be present in $\mathcal{G}$, which will constraint the planner to generate a complementary *shape* graph from S_{init} during the *on-line* process.

2.5 A first step through solution extraction

To perform efficient *on-line* solution extraction, ShaPer takes advantage of the *shape* graph $\mathcal{G}$ built *off-line*. Three steps are necessary to find a plan: first, generate the graph $\mathcal{G}_{init}$ (resp.

problem	state space size	$\mathcal{G}$ size
blocks-world		
3 blocks	13	3
4 blocks	73	5
5 blocks	501	7
6 blocks	4051	11
7 blocks	41413	15
gripper		
3 balls	88	9
4 balls	256	12
5 balls	704	15
6 balls	1856	18

Table 2: Growth of the state space comparing to the $\mathcal{G}$ size.

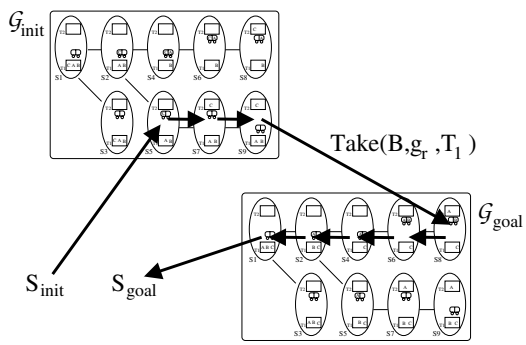


Figure 2: Extract a solution by connecting $\mathcal{G}_{init}$ and $\mathcal{G}_{goal}$.

$\mathcal{G}_{goal}$) associated to the initial state S_{init} (resp. S_{goal}). Then ShaPer searches for **one** action $o \in \mathcal{O}$ that connects $\mathcal{G}_{init}$ with $\mathcal{G}_{goal}$ *via* s_i and s_g ($s_i \in \mathcal{G}_{init}$,$s_g \in \mathcal{G}_{goal}$ and $s_g = o(s_i)$). Finally it is enough to find a path from S_{init} to s_i in $\mathcal{G}_{init}$ and from s_g to S_{goal} in $\mathcal{G}_{goal}$.

To better understand this intuitive algorithm, let us explain it through the 3-balls-gripper example. There are three balls, two tables and two grippers; initially balls A and B are on table T_1 and the ball C is in the left gripper near table T_2. This state has the same *shape* as S_5 of the figure 1 where $\{C/X, A/Y, B/Z\}$. The goal is to obtain the three balls on T_2. To do so, we instantiate the *shape* graph $\mathcal{G}$ with the initial state substitution (resp. goal) and obtain the graph $\mathcal{G}_{init}$ (resp. $\mathcal{G}_{goal}$). Then the algorithm looks for one action which connects a state of $\mathcal{G}_{init}$ to a state of $\mathcal{G}_{goal}$, as illustrated by figure 2 with the action $Take(B, g_r, T_1)$.

3 $\mathcal{G}$ and completeness

The method, for solution extraction (proposed in the previous section) is generally very efficient, however in specific case, it is not always possible to connect the two graphs with a single action. To obtain a solution, the planner may need to connect several substitutions of the graph $\mathcal{G}$; but looking for such transitions may be as costly as planning "from scratch".

Fortunately, we can compute *off-line* a *meta-graph* $\mathcal{H}$ which contains all necessary substitutions of $\mathcal{G}$ to solve any instance of the problem (from the learned domain) in only one action (between $\mathcal{H}_{init}$ and $\mathcal{G}_{goal}$).

3.1 $\mathcal{H}$: a meta-graph of $\mathcal{G}$

In order to ensure the completeness for our method, we propose an expansion algorithm of the graph $\mathcal{G}$ to $\mathcal{H}$. The main idea is the following: if all substitutions accessible from S_{begin} are accessible from $\mathcal{H}$ too, then there exists an action $o \in \mathcal{O}$ which connects $\mathcal{H}_{init}$ to $\mathcal{G}_{goal}$.

To perform such expansion, we begin with $\mathcal{H} = \mathcal{G}$. $\mathcal{H}$ must contain all graph $\mathcal{G}$ that allow to connect graph $\mathcal{G}'$ such that $\mathcal{G}'$ is not directly connected to $\mathcal{H}$. For each $o(h)$ (with $h \in \mathcal{H}$ and $o(h) \notin \mathcal{H}$), generate the graph $\mathcal{G}_{o(h)}$. For each $o'(g)$ (with $g \in \mathcal{G}_{o(h)}$ and $o'(g) \notin \mathcal{G}_{o(h)}$), check if $\mathcal{G}_{o'(g)}$ is directly accessible from $\mathcal{H}$; if it is not the case, then add $\mathcal{G}_{o(h)}$ to $\mathcal{H}$. This algorithm stops when $\mathcal{H}$ becomes invariant. In other words, $\mathcal{H}$ corresponds to the "transitive closure" of $\mathcal{G}$ in terms of substitutions.

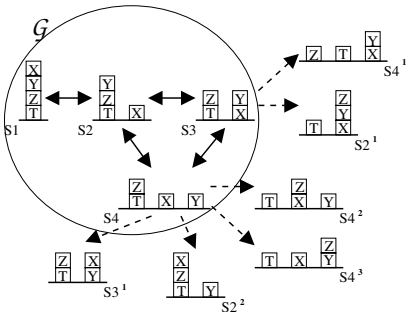


Figure 3: The graph $\mathcal{G}$ for 4-blocks-world domain restricted to 3 stacks.

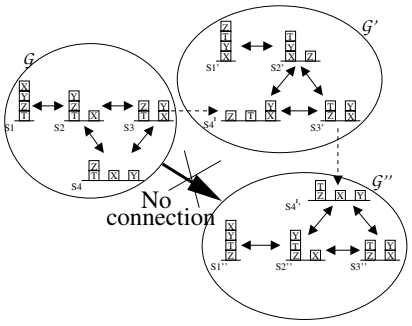


Figure 4: Expanding the graph $\mathcal{G}$ to ensure completeness

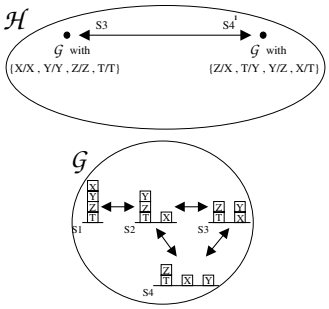


Figure 5: $\mathcal{H}$ models the state space accessible with **one** action from $\mathcal{G}$

3.2 An expansion example

To better understand this expansion, we illustrate it on the 4-blocks-world domain restricted to three stacks. Figure 3 presents the *shape* graph $\mathcal{G}$ and its substitutions (e.g. S'_4 is a substitution of the state S_4). Intuitively, to reverse blocks Z and T in this domain, it is necessary to use two substitutions of S_4; so find a solution with only one action may be impossible only by using $\mathcal{G}$.

To expand $\mathcal{G}$ presented in figure 3, ShaPer must examine states S_4^1, S_2^1, S_4^2, S_4^3, S_2^2 and S_3^1 (states which leave $\mathcal{G}$). Developing $\mathcal{G}'$, the *shape* graph from S_4^1, allows to generate $\mathcal{G}''$ *via* $S_4^{1\prime}$ (see figure 4). Note that there does not exist any action to connect a state of $\mathcal{G}$ to a state of $\mathcal{G}''$; this means that $\mathcal{G}'$ allows to access to new substitutions. Consequently, it is added to $\mathcal{H}$. The graph presented in figure 5 corresponds to the invariant of $\mathcal{H}$ after having examined all the other states.

Naturally, all the graphs included in $\mathcal{H}$ differ only by a substitution. Therefore, $\mathcal{H}$ can be expressed by only using substitutions and $\mathcal{G}$, *e.g.* in figure 5, $\mathcal{H}$ is described by two substitutions on $\mathcal{G}$.

4 Solution extraction

$\mathcal{G}$ and $\mathcal{H}$ being computed *off-line*, ShaPer performs *on-line* plan extraction for any possible problem from the learned domain by searching for **one** action.

Extract_Solution($\mathcal{O}$,$\mathcal{G}$,$\mathcal{H}$,S_{init},S_{goal})
  Find σ and $g \in \mathcal{G}$ such that $S_{init} = \sigma(g)$
  $\mathcal{H}_{init} = \sigma(\mathcal{H})$ /* Instantiate $\mathcal{H}$ with σ */
  $\mathcal{G}_{goal} = \theta(\mathcal{G})$ with $S_{goal} = \theta(g')$
  $cost \leftarrow 0$
  While not examine all states of $\mathcal{H}$ **do**
    For each substitutions τ of $\mathcal{H}$ **do**
      For each triple (s, g'', μ) of $\tau(\mathcal{G})$ **do**
      /* $\exists o \in \mathcal{O}.o(s) = \mu(g'')$ μ a substitution */
      /* and $(s, g'') \in \mathcal{G}^2$ */
        If $C(S_{init} \rightarrow s) + 1 + C(o(s) \rightarrow S_{goal}) = cost$
          Then If $o(s) = \theta(g'')$ **Then**
            return the plan: Plan($S_{init} \rightarrow s$), o,
                Plan($o(s) \rightarrow S_{goal}$)
    $cost \leftarrow cost + 1$
  return: No solution

Table 3: Algorithm for solution extraction

4.1 An efficient algorithm

Table 3 presents the three-steps algorithm used to extract a plan from an initial state S_{init} to S_{goal}. As the graph $\mathcal{G}$ is computed *off-line*, it is also possible to compute the best path for any couple of vertices in $\mathcal{G}$. Let $Plan(g_1, g_2)$ be the optimal plan from g_1 to g_2 in $\mathcal{G}$ and $C(g_1, g_2)$ the number of actions of this plan. Considering the cost C, it is possible to find the best solution using only two *shape* graphs and one action. Indeed, the algorithm examines iteratively all possible connections, ordered by an increasing cost (the cost is defined by $C(S_{init}, s) + 1 + C(o(s), S_{goal})$). Such procedure makes it possible to obtain non-optimal, because of the graph learning, but good solutions. This is illustrated by the number of plan steps in the results presented in table 4.

Figure 6 shows an example of a solution extraction in 4-blocks-world domain restricted to three stacks. First, ShaPer instantiates the graph $\mathcal{H}$ with the initial state ($\sigma_{init} = \{D/X, B/Y, A/Z, C/T\}$) obtaining $\mathcal{H}_{init}$, and $\mathcal{G}$ with the goal ($\sigma_{goal} = \{D/X, C/Y, B/Z, A/T\}$) obtaining $\mathcal{G}_{goal}$. Then, in order to find a connection between $\mathcal{H}_{init}$ and $\mathcal{G}_{goal}$, the algorithm examines (following an increasing cost) the states $S_4^1 \ldots S_3^1$ and $S_4^{1\prime} \ldots S_3^{1\prime}$ (see figure 3) and finds $S_4{}' \rightarrow S_4^{3\prime}$ as first connection. Then, to build a valid

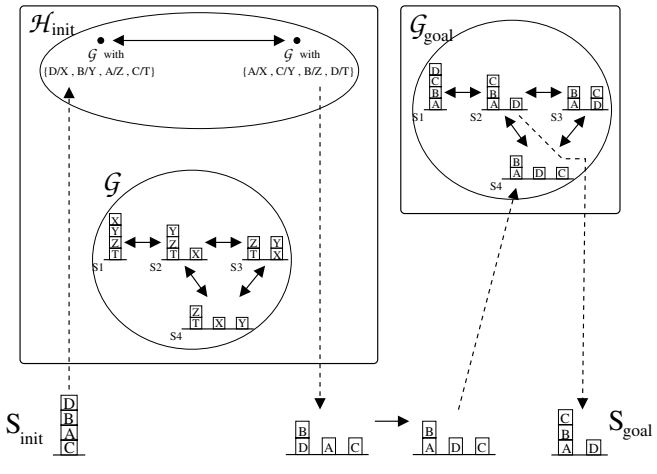


Figure 6: A solution extraction example in 4-blocks-world domain restricted to three stacks

plan, it is enough to find a plan in $\mathcal{H}_{init}$ and a plan in $\mathcal{G}_{goal}$. Thus, the plan extracted by ShaPer is the following: [*MoveToTable(D,B), Move(B,A,D), MoveToTable(A,C)*], *Move(B,D,A)*, [*MoveFromTable(C,B)*].

4.2 Bounds for Computation Time and plan length

Having the *shape* graph computed *off-line* leads to an interesting property: the solution extraction algorithm is time bounded and the plans have a known maximal length.

Indeed, the algorithm presented in table 3 examines iteratively a set of triples (s, g'', μ) of $\tau(\mathcal{G})$. The test consists in checking equality between two states. As $\mathcal{H}$ is computed *off-line*, we know in advance how many triples there are, *e.g.* for 15 blocks there are only 7186 triples. Consequently, we can compute an upper limit for the *on-line* solution extraction computation time.

In the same way, we can compute an upper limit n for the longest path P in $\mathcal{G}$. The length of the longest plan (maximum number of plan steps) will then be bounded by $2n + 1$.

The results, presented in table 4, include a *max* column for number-of-steps and computation time bounds.

4.3 Completeness

This subsection present the main ideas to prove the completeness of ShaPer in informal way.

For a substitution σ, $\sigma(A \cup B) = \sigma(A) \cup \sigma(B)$ and $\sigma(A \cap B) = \sigma(A) \cap \sigma(B)$. So we can demonstrate the following theorem: $\sigma(o(s)) = (\sigma(o))(\sigma(s))$ with o being an action applicable to the state s[6].

The algorithm to build $\mathcal{G}$ ensures that for any state s in $\mathcal{G}$ and any action o such that $o(s) \notin \mathcal{G}$, there exists a substitution σ of a state g of $\mathcal{G}$ with $o(s) = \sigma(g)$. Then it is possible to demonstrate that $\mathcal{G}$ contains all accessible *shapes*. Given $s \in \mathcal{G}$ and $e = o(s) \notin \mathcal{G}$. Suppose that e allows to access to the state $s' = o'(e)$ which has a new *shape*. From $\mathcal{G}$ definition, $\exists \sigma \wedge g \in \mathcal{G}.\sigma(g) = e$. The action $\sigma^{-1}(o')$ is applicable to g; if $\sigma^{-1}(o')(g) \notin \mathcal{G}$ then $\exists \sigma'(g') = \sigma^{-1}(o')(g)$ such that $\sigma(\sigma^{-1}(o')(g)) = o'(\sigma(g)) = o'(e)$ and $\sigma \circ \sigma'(g') = o'(e)$ so g' has the same *shape* as e.

Similarly, $\mathcal{H}$ satisfies the following property: for any state h in $\mathcal{H}$ and g in $\mathcal{G}'$ ($\mathcal{G}'$ is connected to $\mathcal{H}$ by one action through the substitution σ), all accessible substitutions from $\mathcal{G}'$ are also accessible from $\mathcal{H}$. $\mathcal{H}$ is the transitive closure of $\mathcal{G}$ in terms of substitutions. It is also possible to recursively prove that ShaPer finds a plan if there exists one. Given $s \in \mathcal{H}$ and a plan $P = a_n \circ \ldots \circ a_1$ such that $goal = P(s)$. If $a_1(s) \notin \mathcal{H}$ then there exists $\sigma(h) = a_1(s)$ with $a_1(s) \in \mathcal{G}'$ and $h \in \mathcal{H}$. Given i and the state $e = a_{i-1} \circ \ldots \circ a_1(s)$ and $e' = a_i(e)$ with $e \in \mathcal{G}'$ and $e' \notin \mathcal{G}'$. So $e' \in \mathcal{G}''$ (connected to $\mathcal{G}'$ by the state $g' \in \mathcal{G}'$ and a substitution θ: $\theta(g') = e'$). Owing to $\mathcal{H}$'s property, there exists a state h' of $\mathcal{H}$ and a substitution τ such that $\tau(g') = e'$. That means ShaPer is able to find a plan between s and e' (connect $\mathcal{H}$ to $\mathcal{G}''$ with one action); apply recursively this reasoning[7] on $a_n \circ \ldots \circ a_i$, demonstrates the completeness.

[6] Indeed $\sigma(o(s)) = \sigma((s - D_o) \cup A_o) = (\sigma(s) - \sigma(D_o)) \cup \sigma(A_o) = (\sigma(o))(\sigma(s))$

[7] If the plan P uses a third *shape* graph $\mathcal{G}^3$ (one-action-connected to $\mathcal{G}''$ which is also one-action-connected to $\mathcal{H}$) , the previous reasoning allows to prove that there exists a connection with one action between $\mathcal{H}$ and $\mathcal{G}^3$.

5 Results

In this section, we compare our planner with the most efficient planners from AIPS-2000 competition. FF [Hoffmann, 2000] and HSP [Bonet and Geffner, 1999] use heuristics based on a relaxed problem (plan without *Dellist*). IPP [Koehler *et al.*, 1997] and STAN [Fox and Long, 1999] are derived from GraphPlan [Blum and Furst, 1997]. In addition, they use an *on-line* mechanism that allows to deal with some symmetries; that is reason why it is interesting to compare them with ShaPer which is able to deal with a larger class of symmetries.

In this comparison, we do not mention TalPlan [Doherty and Kvarnstrom, 1999] because it is a domain dependent planner; the comparison can then be only made with ShaPer solution extraction step. In such a case, TalPlan exhibits clearly better results than ShaPer: the made-by-hand heuristics is still the best.

All running time presented in table 4 are measured on a Sparc Ultra 5 with 128Mb. '-' means that the planner does not find any solution in 3600sec. and '*' means that the *shape* graph has already been computed for a previous problem (same *domain subclass*).

The ferry and the gripper domains are interesting because they overwhelm the majority of the planners by the number of their applicable actions. Despite their ability to treat state symmetries, IPP and STAN can not solve large problems in both ferry and gripper domain. Ferry and gripper domains are easy for ShaPer because of their low number of *shapes* (resp. $2n + 1$ for n cars and $3n$ for n balls). Thanks to their good heuristics, HSP and FF solve easily all these problems (however, note that HSP solves gripper problems with an inefficient number of steps).

Blocks-world domains give surprising results for HSP and FF. HSP solves '-3' and '-2' problems more easily than '-1' problems. For FF, the situated is opposite. Problems labeled '-i' correspond to i-stacks problems: a goal for a '-1' problem is to put the first block under the stack (without changing the order of the other blocks); for '-2' and '-3' problems, the goal is to build one "interleaved" stack composed of all the blocks from the initial stacks (e.g. move stacks $s_1, s_2 \ldots$ and $p_1, p_2 \ldots$ into $s_1, p_1, s_2, p_2 \ldots$).

With several problems in the same *domain subclass*, *e.g.* all problems with 15 blocks, we better understand the importance of the *off-line* process (computed more efficiently than some HSP or FF solution): the graph building step is performed only once. The small number of *shapes* allows to have a very short computation time upper limit.

In conclusion, ShaPer solves all problems in less than 1 second; in addition it builds the graph and solves all problems faster than IPP and STAN (and often HSP) and extracts all problems faster than FF with a near-optimal number of plan steps.

6 Conclusion

We have proposed an original approach to task planning that allows to deal efficiently with large problems. It has been

	IPP-v4		STAN-v3		HSP-v2		FF-v2.2		ShaPer					
									off-line		*on-line*		*max*	
problem	step	time	step	time	step	time	step	time	nodes	time	step	time	step	time
ferry														
10 cars	39	3.00	39	7.2	39	0.08	39	0.04	21	0.02	39	0.01	41	0.01
20 cars	-	-	-	-	79	0.20	79	0.07	41	0.21	79	0.01	81	0.01
50 cars	-	-	-	-	199	5.50	199	0.20	101	4.86	199	0.07	201	0.08
gripper														
10 balls	29	56.9	29	202	37	0.10	29	0.03	30	0.07	29	0.01	55	0.01
20 balls	-	-	-	-	77	0.70	59	0.08	60	0.55	59	0.01	115	0.01
50 balls	-	-	-	-	197	18.80	149	0.30	150	14.04	149	0.15	295	0.27
blocks-world														
9-1 blocks	16	3.71	16	18.4	16	1.32	16	0.06	30	0.15	16	0.01	19	0.02
9-2 blocks	15	3.70	18	3.6	15	0.60	15	0.06	*	*	18	0.01	*	*
9-3 blocks	17	2.47	19	4.7	14	0.64	26	2.21	*	*	17	0.01	*	*
12-1 blocks	-	-	-	-	22	26.33	22	0.13	77	1.58	22	0.01	29	0.03
12-2 blocks	-	-	-	-	21	3.63	21	0.12	*	*	25	0.01	*	*
12-3 blocks	-	-	-	-	20	3.54	29	21.27	*	*	25	0.01	*	*
15-1 blocks	-	-	-	-	28	657.6	28	0.29	176	16.73	28	0.06	39	0.13
15-2 blocks	-	-	-	-	27	22.26	27	0.27	*	*	29	0.07	*	*
15-3 blocks	-	-	-	-	23	13.43	-	-	*	*	33	0.09	*	*
20-1 blocks	-	-	-	-	-	-	38	0.85	627	173.79	38	0.51	55	0.90
20-2 blocks	-	-	-	-	37	165.6	37	0.72	*	*	37	0.46	*	*
20-3 blocks	-	-	-	-	-	-	-	-	*	*	38	0.77	*	*

Table 4: Running time and quality (in number of Plan action) for some of the current best planners (see the AIPS-2000 competition); Comparing to ShaPer including its *off-line* process.

implemented in a domain independent task planner, called ShaPer. It performs in two steps. The first step is performed *off-line* and only once for a given *domain subclass*. It allows ShaPer to "capture the structure" of the state space and to store it in a data structure called the *shape* Graph. The main contribution here is the ability of ShaPer to build a very compact description when compared to the size of the complete state space. The *shape* graph is then used *on-line* by ShaPer to answer very efficiently to planning requests.

ShaPer exhibits several interesting properties: 1) it is complete, 2) It is possible to determine, after the *shape* Graph construction, the upper limit for the *on-line* solution extraction computation time as well as length of the longest plan, 3) it produces "good" (near-optimal) solutions.

Our future work will be devoted to a state decomposition method. Indeed, we would like to decompose a state into several disjoint parts in order to minimize interferences in action applicability. This should allow to generate sub-graphs and to deal with *sub-shapes* resulting in an even more compact state space description.

This would help to re-use *shape* graph created for a given *domain subclass* (*e.g.* 10 blocks) in order to generate *shapes* for "bigger" *subclasses* (*e.g.* 15 blocks) Besides, we hope to be able to exploit the structure of the *shape* graph for conformant and contingent planning as in [Guéré and Alami, 1999].

References

[Blum and Furst, 1997] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, pages 281–300, 1997.

[Bonet and Geffner, 1999] B. Bonet and H. Geffner. Planning as heuristic search: New results. *5th European Conference on Planning (ECP'99)*, 1999.

[Doherty and Kvarnstrom, 1999] P. Doherty and J. Kvarnstrom. Talplanner: An empirical investigation of temporal logic-based forward chaining planner. In *Proc. 6th Int. Workshop on Temporal Representation and Reasoning*, 1999.

[Fikes and Nilsson, 1971] R.E. Fikes and N.L. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Fox and Long, 1998] M. Fox and D. Long. The automatic inference of state invariants in tim. *AI Research (JAIR)*, 9:367–421, 1998.

[Fox and Long, 1999] M. Fox and D. Long. The detection and exploration of symmetry in planning problems. *Proc. 16th Inter. Joint Conf. on Artificial Intelligence (IJCAI'99)*, 1999.

[Gerevini and Schubert, 1998] A. Gerevini and L.K. Schubert. Inferring state constraints for domain-independent planning. In *Proc. 15th Nat. Conf. AI.(AAAI'98)*, 1998.

[Guéré and Alami, 1999] E. Guéré and R. Alami. A possibilistic planner that deals with non-determinism and contingency. *Proc. 16th Inter. Joint Conf. on Artificial Intelligence (IJCAI'99)*, 1999.

[Hoffmann, 2000] J Hoffmann. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Proc. 12th Int. Symposium on Methodologies for Intelligent Systems*, 2000.

[Koehler *et al.*, 1997] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an adl subset. In *4th European Conference on Planning (ECP'97)*, 1997.

Hybrid STAN: Identifying and Managing Combinatorial Optimisation Sub-problems in Planning

Maria Fox and Derek Long
Department of Computer Science,
University of Durham, UK
maria.fox@dur.ac.uk d.p.long@dur.ac.uk

Abstract

It is well-known that planning is hard but it is less well-known how to approach the hard parts of a problem instance effectively. Using static domain analysis techniques we can identify and abstract certain combinatorial sub-problems from a planning instance, and deploy specialised technology to solve these sub-problems in a way that is integrated with the broader planning activities. We have developed a hybrid planning system (STAN4) which brings together alternative planning strategies and specialised algorithms and selects between them according to the structure of the planning domain. STAN4 participated successfully in the AIPS-2000 planning competition. We describe how sub-problem abstraction is done, with particular reference to route-planning abstraction, and present some of the competition data to demonstrate the potential power of the hybrid approach.

1 Introduction

The knowledge-sparse, or domain-independent, planning community is often criticised for its obsession with toy problems and the inability of its technology to scale to address realistic problems. Planners using weak heuristics, which attempt to guide search using general principles and without recourse to domain knowledge, cannot compete, in a given domain, against a planner tailored to perform well in that domain.

On the other hand, tailoring a planner to a particular domain requires considerable effort on the part of a domain expert. This effort is generally not reusable because a different domain requires a whole new body of expertise to be captured and it is not clear what (if any) general principles can be extracted from any single such effort to facilitate the next. The philosophy underlying our work on domain analysis is that knowledge-sparse planning can only be proposed as realistic general planning technology if it is supplemented by sophisticated domain analyses capable both of assisting a user in the development of correct domain descriptions and of identifying structure in a planning domain that can be effectively exploited to combat search.

In this paper we describe a way of decomposing planning problems to identify instances of NP-hard sub-problems, such as Travelling Salesman, that are most effectively solved by purpose-built technology. Knowledge-sparse, general, planning is unintelligent because it uses the same methods to solve all problems, whether they genuinely require planning or are in fact instances of well-known problems that are themselves the topic of substantive research. A more powerful approach is to allow such sub-problems to be abstracted out of the planning problem and solved using specialised technology. The different problem-solving strategies must then be integrated so that they can cooperate in the solution of the original problem.

We have been experimenting with using the automatic domain analysis techniques of TIM [Fox and Long, 1998] to recognise and isolate certain combinatorial sub-problems and to propose a way in which their solution, by specialised algorithms, might be integrated with a knowledge-sparse planner.

The work described in this paper has been successfully implemented in version 4 of the STAN system (STAN4) and has proved very promising. STAN4 competed in the AIPS-2000 planning competition where it excelled in problems involving route-planning sub-problems and certain resource allocation problems involving a restricted form of resource. The data sets from the competition are discussed in Section 7. In STAN4, TIM selects between a forward and backward planning strategy depending on characteristic features of the domain. The forward planning component, FORPLAN, is integrated with simplified specialist solvers for certain simple route-planning and resource allocation sub-problems. In Section 3 we describe the components of the hybrid architecture of STAN4 and explain the integration of these components.

2 Recognising Generic Behaviours

TIM can identify a collection of *generic types* within a domain. Generic types [Long and Fox, 2000] are collections of types, characterised by specific kinds of behaviours, examples of which appear in many different planning domains. For example, domains often feature *transportation* behaviours since they often involve the movement of self-propelled objects between locations. TIM can identify *mobile* objects, even when they occur implicitly, the operations by which they move and the maps of connected locations on which they move. The analysis automatically determines whether

the maps are static (for example, road networks) or dynamic (for example, corridors with lockable doors). The recognition of transportation features within a domain suggests the likelihood of route-planning sub-problems arising in problem instances.

TIM also recognises certain kinds of *resources* which restrict the use of particular actions in a domain. Finite renewable resources, which can be consumed and released in units, are encoded in STRIPS using discrete state changes to model the stages of their consumption and release (for example, in the Freecell domain). STAN4 recognised and exploited this generic behaviour in the Freecell domain in the AIPS-2000 competition (see Figure 5) but we are not, yet, exploiting such resources in a robust way.

TIM takes as input a standard, unannotated, STRIPS or ADL description of a domain and problem. Integration of specialised technology with the search strategy of a planner is most straightforward to achieve in a heuristic forward-search-based planner, so we have implemented a forward planner, FORPLAN, using a simple best-first search strategy. It uses a heuristic evaluation function, based on solving the relaxed planning problem, similar to the approach taken by HSP [Bonet *et al.*, 1997] and Hoffmann's FF [Hoffmann, 2000]. Like FF, FORPLAN uses a relaxed version of GraphPlan to compute the relaxed plan estimate. The difference between FORPLAN and FF is that the relaxed plan is constructed for the *abstracted* planning problem – that is, the part of the planning problem that remains when operators and preconditions relating to the identified sub-problem have been removed. This gives us only part of the heuristic estimate. The heuristic estimate is then improved by estimating the cost of solving the removed sub-problem. This two-part process can result in much better estimates than those produced by other forward-search-based planners. FORPLAN does not, at present, exploit any general heuristics (such as Hoffmann's *helpful actions*) to inform its distance estimates. Since the repertoire of sub-problems that TIM can recognize is currently very limited, FORPLAN has only been integrated with sub-solvers for certain forms of route planning and discrete resource handling. Without an appropriate sub-solver to exploit, FORPLAN easily becomes lost in the search space and therefore is ineffective as a general planner at present.

3 The Hybrid Planner Architecture

Because FORPLAN is not yet effective as a general planner we cannot rely on it for solving problems that do not have route planning or resource handling sub-problems. Our intention, in the development of STAN4, has been to demonstrate an effective means of abstracting sub-problems from planning instances and of integrating special-purpose sub-solvers within a general planning framework. In order to be able to report results for problems that do not have the sub-problems currently in TIM's repertoire we retained STAN version 3 as a default planning strategy to use in domains not featuring these sub-problems. The presence of STAN3 means that TIM fails safe when it fails to identify a key sub-problem. At the moment this happens quite often because we are working with some simplifying assumptions, described in section 6, but we are gradually increasing the range of sub-problems that can be handled.

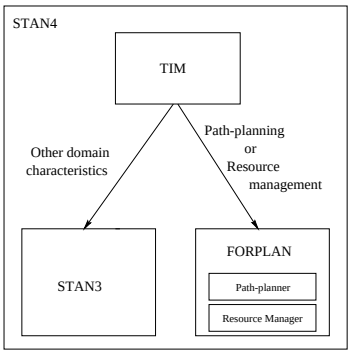


Figure 1: The architecture of the STAN4 hybrid system.

TIM operates as an interface to STAN4, selecting between its components according to the structure of the domain. A high-level view of STAN4 is presented in Figure 1.

In this paper we describe the processes by which route-planning sub-problems, once identified by analysis of a domain description, are abstracted and their solution, by specialised algorithms, integrated with FORPLAN. The processes by which resource-allocation sub-problems are handled are similar, but we do not describe them in detail here.

TIM first analyses the domain and problem instance to identify whether mobile objects are present in the domain, and if so whether a *decomposable* transportation sub-problem can be found. A transportation sub-problem is decomposable if the form of mobility present in the domain is constrained in certain predefined ways (discussed in section 4. TIM identifies the mobile objects and the *locatedness* predicates (also referred to as the *atrel*s) they use (for example: *at*, *inroom*, *by*, etc). The locatedness predicate is important for identifying actions that rely on, or bring about, changes in the locations of mobile objects. If an appropriate form of mobile is found TIM invokes FORPLAN together with the route-planning sub-solver.

The data presented in Section 7 shows the performance obtained in the AIPS-2000 planning competition on domains involving route planning and resource handling sub-problems. These domains were: Logistics and the STRIPS version of the elevator domain (route-planning) and Freecell (resource-allocation).

4 Sub-problem Recognition

Having found that there are mobile objects in the domain TIM determines whether the problem of planning their movement between locations can be safely delegated to a sub-system. If the shortest distance to be travelled by an object moving from one location to another can be ascertained by looking at the map that the object moves on, then path-planning for that object can be devolved to a shortest path algorithm. If not (if the object can temporarily vacate the map en-route between two points) then a shortest path algorithm cannot be guaranteed to find the best path. The ability to vacate the map suggests that the mobile object is able to use two or more maps, each

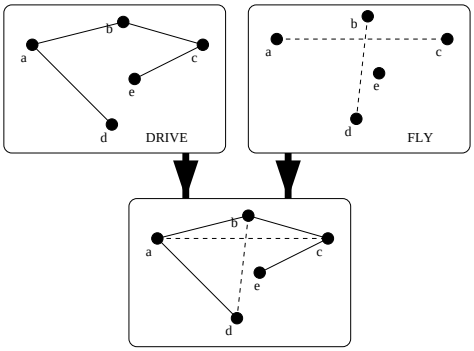


Figure 2: The Result of Amalgamating Two Maps.

by exploiting a different "move" operator. By entering a second map and moving between its locations, the object may be able to reappear on its first map at a location different from the one it left, and this *flying* behaviour might give access to shorter paths than are visible on the first map alone. To enable the problem of movement planning to be delegated to a sub-solver it is necessary to equip the sub-solver with the map that the mobile can traverse.

In the case where several maps can be used it is necessary to amalgamate the maps and to label the edges with the movement actions that the mobile uses to traverse them, and the locatedness predicate that the mobile uses with each action. The bulldozer domain (this can be found in the PDDL release [McDermott, 1998]) is an example of a domain in which an amalgamated map is required to enable route-planning to be delegated. The bulldozer uses two maps – the map of roads, which it traverses by *driving*, and the map of bridges, which it traverses by *crossing*. Both maps give access to the same set of locations.

A simple example arises when a mobile can both fly and drive between locations in the same collection. Figure 2 shows a situation in which the shortest route between a and e (assuming that all edges have the same cost) is to fly from a to c and drive from c to e – a route that is available only in the amalgamated map. To complicate matters, if there are additional actions that must be performed to enable certain edges to be traversed (for example, in order to fly the wings must be bolted on) then the shortest path in the amalgamated map might be to *drive* from a to b, from b to c and then from c to e, despite the fact that this path contains more edges in the amalgamated map.

To perform the amalgamation of n maps $G_i = \langle V, E_i \rangle$ for $i = 1 \ldots n$, we construct a single graph structure $G = \langle V, \bigcup_{i=1\ldots n} E_i \rangle$. We label the edges in G each with the "move" action that can be used to traverse it and the locatedness predicate relevant to that movement action. This allows route plans to be constructed, using the appropriate actions and predicates, by a dedicated sub-solver. The means by which route plans are constructed are described in section 5.

If the object must always re-enter the first map at the same location as the one it left when it entered the second map, then the shortest path between two points *is* guaranteed to be visible on the first map. This restricted form of flying, which we call *hovering*, does not require the amalgamation of different maps. This case is not distinguished from the case in which a single map of locations is traversed using a single "move" operator.

A problem arises when the map (or amalgamated map) that a mobile object traverses is *dynamic* – that is, able to be changed by the actions of the planning agent. The grid world presents such an example, because new routes become available as the robot obtains keys to open doors in the grid. The route planning sub-part of grid planning problems are closely integrated with the key-collecting goals of the robot, so not sufficiently decomposable to be delegated. STAN4 does not attempt to abstract the route planning problem for this domain.

5 Sub-problem Abstraction

To achieve the abstraction of route-planning, once TIM has identified an appropriate mobile type, STAN4 associates with each mobile object a data structure which records the current location of the object. It also identifies each operator, other than the move operation of the mobile itself, the preconditions of which require a mobile of this type to be located at a particular location in order for the action to be executed. Once found, these preconditions are removed from the operators in which they appear, but each operator is then equipped with an additional data value identifying where the mobile must be in order to satisfy the abstracted precondition. In other words, the precondition is transformed from a standard proposition into a specialised representation with equivalent meaning, but allowing specialised treatment. This specialised representation (which we call a *mobile-tag*) provides the means of communication between the planner and a specialised sub-solver. The way the tag is used is described in section 6. Figure 3 shows the result of abstracting the $atrel$ from the $load$ operation in the Logistics domain.

All move operations for the mobiles are then eliminated from the domain altogether. This results in an abstracted version of the domain containing the components of the original planning problem that the planner will be required to solve. The problem is solved by the planner in this abstracted form.

6 Integration

Integration between FORPLAN and the route planning subsolver is required in three places: in determining a heuristic estimate of distance between the current state and the goal state; in constructing a route to be followed in moving a mobile between two locations in the plan; in generating a plan format for reporting the routes to be traversed by the mobiles in the plan. We explain each of these stages in turn.

FORPLAN solves the abstracted problem using a heuristic estimate of the value of a state based on the length of the relaxed plan between that state and the goal. The heuristic estimate is calculated by first constructing a relaxed plan with the abstracted operators, and calculating its length, and then adding to it an estimate for the lengths of the routes that would have to be traversed by any mobiles it uses. For the lengths of these routes to be estimated it is necessary for the abstracted relaxed plan to record the commitments it makes on mobile

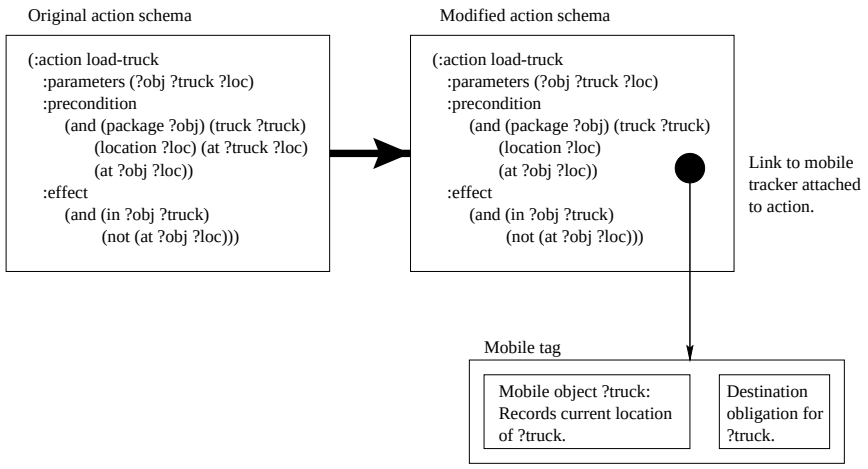


Figure 3: The Sub-problem Abstraction Process

objects to visit locations at which abstracted actions will be performed. For example, in Logistics, the abstracted plan consists just of *load* and *unload* actions, but their successful execution commits mobile objects (trucks and planes) to be in place. The abstracted plan therefore maintains an array of *commitments* of mobiles to visit certain locations, associated with the layers of the relaxed plan graph, and the sub-solver estimates the lengths of the routes that available mobiles must traverse in order to meet these commitments.

The cost of traversing the routes that a plan entails is too expensive to compute with accuracy. The arrays of commitments show which locations each mobile is required to visit to satisfy the requirements of the plan, with some ordering constraints implied by the layering of the relaxed plan graph and indicating the dependencies between the activities the mobile will be involved in at each location (loading must be carried out before unloading and so on). To calculate a shortest path that visits all these locations and respects these orderings is a variation on a Travelling Salesman problem, with multiple travellers and additional constraints. This problem is hard and cannot be solved repeatedly as part of the heuristic evaluation of a state. Instead, we produce an estimate of the cost by assuming that each mobile can visit each location in turn from the closest of the locations it has previously visited in the plan, respecting ordering constraints on the visits. Although this is an unsophisticated approach to tackling the Travelling Salesman problem, its integration with the planning process is a proof of concept, demonstrating the possibility of integrating more specialised technology. Despite its lack of sophistication it gives a better estimate of the cost of a state than a pure relaxed plan estimate, since relaxed plan estimates neglect the fact that a mobile cannot be in two places at the same time (the relaxed plan ignores delete conditions and it is these which express the fact that a mobile cannot be at two places at once).

Integration between the planner and the route-planner is required again when actions are selected for addition to the plan. Once an action is selected it is checked to determine whether it contains an abstracted locatedness precondition. If so, a path is proposed to move the mobile from its current location to the required destination (recorded within the mobile tag associated with action). We use the shortest path between the current location of the mobile (which is always known in a forward search) and the required location recorded in the mobile tag. At present this path is precomputed by TIM using Floyd's shortest paths algorithm [Floyd, 1962], on the (possibly amalgamated) map inferred from the initial state. This approach works well for static maps, where the shortest paths remain fixed, and in situations in which the movement consumes no additional resources. If the mobile does use resources during its movement, it might be that the shortest path is not the best, but instead a longer path which consumes fewer resources is to be preferred.

Finally, it is necessary to integrate the efforts of the planner and the route-planner to produce output in the form of a plan sequence. Once a route has been planned between the appropriate locations, STAN4 generates instantiations of the necessary move operators to produce a plan sequence corresponding to standard format for STRIPS plans. Below we present some of the preliminary results obtained using domains from the STRIPS subset of the AIPS2000 competition data set. In this collection of domains, Logistics and the MICONIC-10 lift domain both contain a path-planning sub-problem which TIM was able to identify and extract. Even using just our simple path-planning strategy we were able to obtain a significant performance advantage from exploiting path-planning abstraction.

7 Experimental Results

The data sets presented here were compiled by Fahiem Bacchus during the AIPS-2000 competition, held in Breckenridge, Colorado. In the graphs, the thick line plots the results of STAN4. Graphs showing time performance are log-scaled.

The graphs show how STAN4 performed on problems from the STRIPS data set involving either route-planning or resource allocation. The planners used for comparison are FF [Hoffmann, 2000], HSP-2 [Bonet and Geffner, 1997],

TALplanner [Doherty and Kvarnstrom, 1999], SHOP [Nau *et al.*, 1999] and, occasionally, GRT [Refanidis and Vlahavas, 1999]. The problems used were Logistics, Freecell and the STRIPS version of the Miconic-10 elevator domain. Only the Logistics and Freecell data is shown, because of space restrictions.

The competition comprised a fully-automated track and a hand-coded track in which planners were allowed to use hand-tailored domain knowledge. In the results presented here, STAN4, FF, GRT and HSP-2 are all fully-automated, whilst TALplanner and SHOP use hand-coded control knowledge.

STAN4 participated in the fully-automated track on the STRIPS problems. All planners able to handle the STRIPS version of PDDL competed in the STRIPS problems, including planners in the hand-coded track. However, despite the advantage of being supplied with hand-coded control knowledge, these planners did not consistently out-perform the fully automated planners. For example, STAN4 and FF were both faster than TALplanner and SHOP on the first Logistics data set (not shown) and produced at least as high quality plans.

From Figure 4 it can be observed that STAN4 took slightly longer than FF on the larger Logistics problems, but produced slightly better quality plans than any other planner, including those in the hand-coded track. As was emphasised earlier, the improvement in plan quality over FF derives from the fact that STAN4 uses a more informative heuristic than FF. STAN4 is using route-abstraction in this domain and achieves a small but consistent improvement in plan quality as a result.

The Freecell domain, Figure 5, was introduced specially for the competition and is a STRIPS formalisation of a solitaire card game released under Windows. Freecell has a resource-allocation sub-problem, because the free cells are a restricted, renewable and critical resource. To estimate how far a state is from the goal it is necessary to take into account the cost of ensuring that sufficient free cells are made available to meet the requirements of the abstracted relaxed plan. Our purpose-built technology for calculating this cost ensures that the consumption of resources does not exceed availability of those resources. If a plan entails over-consumption then the cost of sufficient release actions to redress the balance is added in to the estimate of its value. We have not yet succeeded in obtaining a robust way of accurately estimating these costs, and the performance of STAN4 is somewhat inconsistent as can be seen from the graph. Despite being fastest in all of the problems that it could solve, STAN4 missed several problems and was unable to solve any of the larger instances. Its plan quality was generally good, except for some anomalously long plans. More work is needed to adequately estimate the cost of distributing resources efficiently throughout a plan.

The propositional elevator domain used in the competition reveals one of the weaknesses of the nearest neighbour heuristic, demonstating that it is not a good general purpose approach for route-planning. In this domain STAN4 produces slightly poorer quality plans at the the top end, than either FF or GRT (data not shown). This is because the nearest neighbour heuristic favours visiting all of the pick-up locations before any of the drop-off locations (the simplest way

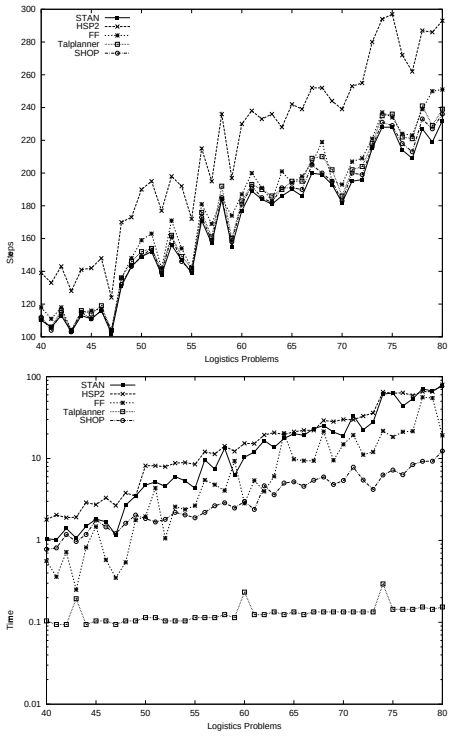


Figure 4: Quality of plans for, and time consumed to solve, Logistics problems 40-80

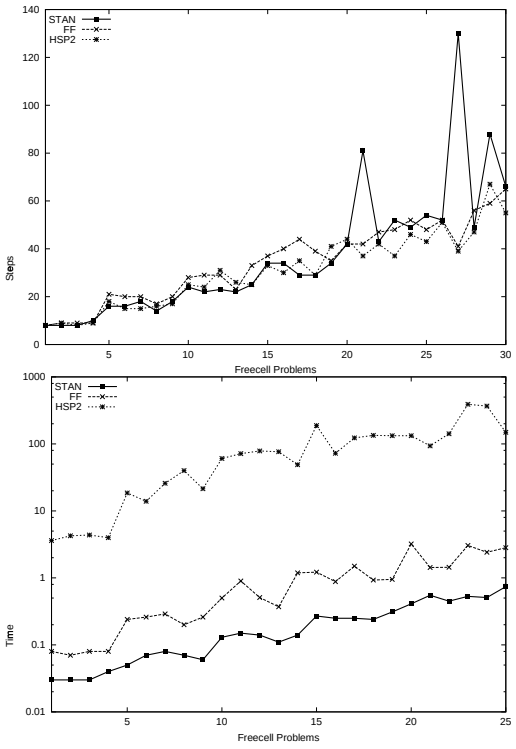


Figure 5: Quality of plans for, and time consumed to solve, Freecell problems.

of respecting the ordering constraints in the plan). In fact, a subtler approach would be to allow the drop-off locations to be inter-mingled with the pick-up ones, provided that a drop-off location is only selected next when the necessary people are on board. The nearest-neighbour heuristic tends to work less well whenever there are many objects to be transported (and many locations to be visited), and few carriers, as well as additional constraints (derived from the need to collect objects before delivering them) as in the elevator domain. The heuristic results in greater accuracy in Logistics because there are (typically) few packages to be transported by any one carrier. However, the nearest-neighbour heuristic was only ever intended to demonstrate that it is possible to integrate purpose-built machinery into the heuristic estimate, allowing the incurred cost of solving an abstracted problem to be taken into account in measuring the goodness of a state. We are currently investigating more sophisticated special-purpose algorithms.

The data presented here shows that FORPLAN can rival the best available planning technology in domains featuring the sub-problems that can be identified by TIM. The fact that FORPLAN has no general search control mechanisms, and obtains its performance in these domains entirely by exploitation of appropriate sub-solvers, gives a clear indication of the potential value of automatic sub-problem abstraction within a forward planning framework. Although FORPLAN is far from effective as a general planner, the exploitation of sub-problem abstraction makes a range of hard problems manageable and the generated solutions efficient.

8 Further Work

Although these foundations have produced promising results the framework we have used to achieve integration is somewhat unsophisticated and inflexible. TIM currently only recognises certain specific forms of mobile and very restricted forms of resource. As TIM fails safe when appropriate forms are not recognised this does not affect the completeness of STAN4. It does mean that STAN4 is often unable to exploit domain structure effectively and we are working on extending its repertoire.

STAN4 can only integrate with one specialised sub-solver, even when there are two or more combinatorial sub-problems in a domain. At present STAN4 emphasises route-planning abstraction because we have made most progress in solving route-planning sub-problems effectively. An important development is to enable integration with more than one sub-solver. This will involve finding a way to communicate constraints between multiple sub-solvers and the planner.

Our "specialised technology" is currently very simplistic. An important refinement is to enable proper integration between the planner and the best available technology for solving combinatorial sub-problems where these arise. Our handling of resources in STAN4 is very restricted. We are working on the recognition of makespan subproblems, which are instances of Multi-processor Scheduling, and their treatment using approximation algorithms for scheduling.

9 Conclusions

We have experimented, using STAN4, with the design of a hybrid planning system in which the choice of problem-solving strategy is made automatically following static analysis of the domain. Our current goals are to improve the integration between FORPLAN and the specialised solvers, allowing a more sophisticated profile of sub-problems to be managed, and to explore what advantages might be gained from integrating other planning strategies into the hybrid.

The key idea underlying our hybrid approach is that planning is not appropriate technology for solving all problems, and that resorting to generic search, or switching between a number of timed strategies, is not an effective way to address such problems. Instead we are interested in building up a collection of purpose-built strategies for combatting some of the most commonly occurring combinatorial sub-problems and making these available, together with techniques for recognising where these problems arise in planning domains. The decision about how to approach a given planning problem can then be made automatically, in a principled way, by deciding how to view the problem and deploying the most effective technology to solve it.

References

[Bonet and Geffner, 1997] B. Bonet and H. Geffner. Planning as heuristic search: new results. In *Proc. ECP*, 1997.

[Bonet *et al.*, 1997] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *AAAI*, 1997.

[Doherty and Kvarnstrom, 1999] P. Doherty and J. Kvarnstrom. Talplanner: An empirical inverstigation of a temporal logic-based forward chaining planner. In *Proceedings of 6th International Workshop on Temporal Representation and Reasoning*, 1999.

[Floyd, 1962] R. W. Floyd. Algorithm 97: shortest path. *CACM*, 5(6), 1962.

[Fox and Long, 1998] M. Fox and D. Long. The automatic inference of state invariants in TIM. *JAIR*, 9, 1998.

[Hoffmann, 2000] J. Hoffmann. A heuristic for domain-independent planning and its use in an enforced hill-climbing algorithm. Technical report, Albert-Ludwigs University, Freiburg, Germany, 2000.

[Long and Fox, 2000] D. Long and M. Fox. Automatic synthesis and use of generic types in planning. In *AIPS*, 2000.

[McDermott, 1998] D. McDermott. PDDL – the planning domain definition language. Technical report, Yale University, http://www.cs.yale.edu/users/mcdermott.html, 1998.

[Nau *et al.*, 1999] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. SHOP: Simple hierarchical orederd planner. In *Proc. IJCAI*, 1999.

[Refanidis and Vlahavas, 1999] I. Refanidis and I. Vlahavas. GRT: A domain independent heuristic for STRIPS worlds based on greedy regression tables. In *Proc. ECP*, 1999.

PLANNING

SEARCH HEURISTICS IN PLANNING

Local Search Topology in Planning Benchmarks: An Empirical Analysis

Jörg Hoffmann
Institute for Computer Science
Albert Ludwigs University
Georges-Köhler-Allee, Geb. 52
79110 Freiburg, Germany

Abstract

Many state-of-the-art heuristic planners derive their heuristic function by relaxing the planning task at hand, where the relaxation is to assume that all delete lists are empty. Looking at a collection of planning benchmarks, we measure topological properties of state spaces with respect to that relaxation. The results suggest that, given the heuristic based on the relaxation, many planning benchmarks are simple in structure. This sheds light on the recent success of heuristic planners employing local search.

1 Introduction

In the last two years, planning systems based on the idea of heuristic search have been very successful. At the AIPS-1998 planning systems competition, HSP1 compared well with the other systems [McDermott, 2000], and at the AIPS-2000 competition, out of five awarded fully automatic planners, FF and HSP2 were based on heuristic search, while another two, Mips and STAN, were hybrids that incorporated, amongst other things, heuristic search [Bacchus and Nau, 2001].

Interestingly, four of these five planners use the same base approach for deriving their heuristic functions: they relax the planning task description by ignoring all delete lists, and estimate, to each search state, the length of an optimal relaxed solution to that state. This general idea has first been proposed by Bonet et al. [1997]. The length of an optimal relaxed solution would yield an admissible heuristic. However, as was proven by Bylander [1994], computing the optimal relaxed solution length is still NP-hard. Therefore, Bonet et al. introduced a technique for approximating optimal relaxed solution length, which they use in both versions of HSP [Bonet and Geffner, 2001]. The heuristic engines in FF [Hoffmann, 2000] and Mips [Edelkamp, 2000] use different approximation techniques.

Three of the above planners, HSP1, FF, and Mips, use their heuristic estimates in variations of local search algorithms, where the search space to a task is the state space, i.e., the space of all states that are reachable from the initial state. Now, the behavior of local search depends crucially on the problem structure, i.e., on the topology of the search space. Thus, the success of these heuristic planners on many planning tasks gives rise to the suspicion that those task's state spaces have a simple structure with respect to relaxed goal distances. In this paper, we shed light on that suspicion. Following Frank et al. [1997], we define a number of structural phenomena in search spaces under heuristic evaluation, impacting the performance of local search algorithms. We compute the optimal relaxed solution length to reachable states in small planning tasks, and measure structural properties. Our results suggest that, in fact, the tasks contained in many benchmark planning domains have a simple state space topology, at least when using the optimal relaxed heuristic. To give an example of how this observation carries over to the approximation techniques used by existing heuristic planners, we apply the same technique of data collection to the FF heuristic. As it turns out, the results are similar. Specifically, it follows that FF's search algorithm is a polynomial solving mechanism in a number of planning benchmark domains, under the hypothesis that the larger instances behave similar to the smaller ones.

Section 2 introduces our general approach, Section 3 gives the basic definitions. Sections 4 and 5 define structural phenomena in search spaces under heuristic evaluation, and give empirical data. Section 6 summarizes the results in a taxonomy for planning domains. Section 7 applies the methodology to the FF heuristic. Section 8 concludes and gives an outlook on further research.

2 General Approach

In our experiments, we used solvable planning tasks only, as we are interested in finding out why local search can succeed so quickly on many benchmark tasks. We looked at instances from 20 different STRIPS and ADL benchmark domains. Due to space restrictions, we only present the results for the domains used in the competitions here, as those domains are well known in the planning community.

To obtain data on how planning tasks behave with respect to the relaxation, rather than with respect to any of the approximation techniques used by existing heuristic planners, we consider the optimal relaxed solution length as our heuristic. As determining that optimal length is NP-hard, it can only be computed for small planning instances. We build an explicit state space representation to such instances, and look at the topology in detail. This yields a clear picture of the fun-

damental structural differences between instances from different planning domains. In that context, we state some hypotheses. A piece of future work is to verify those.

In total, the competitions featured 13 STRIPS and ADL domains: *Assembly*, *Blocksworld*, *Freecell*, *Grid*, *Gripper*, *Logistics*, *Miconic-ADL*, *Miconic-SIMPLE*, *Miconic-STRIPS*, *Movie*, *Mprime*, *Mystery*, and *Schedule*. In 11 of these domains, we used random task generation software to produce small instances, at least 100 per domain. In *Gripper*, there is only one instance of each size: n balls to be transported. In *Movie*, every instance of the AIPS-1998 suite was small enough to be looked at in detail.

Sometimes, we depict scaling behavior. As our instances are all quite small anyway, we need, for that purpose, a finer distinction between instances than obvious criteria like the number of objects. We define the difficulty of a task to be the length of an optimal solution plan, and order our instances within any domain by increasing difficulty. Except in the *Movie* domain (where all instances in the AIPS-1998 suite have the same difficulty), larger instances are on average more difficult than smaller ones. The maximal difficulty of any instance we could look at is 21 in the *Gripper* domain, 20 in the *Assembly* and *Logistics* domains, 18 in *Grid*, and 16 in the *Blocksworld*. In *Movie*, all instances have difficulty 7, and in the remaining domains our maximal difficulty ranges from 10 to 14.

3 Basic Definitions

The competition domains contain tasks specified in the STRIPS and ADL languages. In both cases, a planning task $\mathcal{P}$ is specified in terms of a set of objects O, an initial state $\mathcal{I}$, a goal formula $\mathcal{G}$, and a set of operator schemata $\mathcal{O}$. $\mathcal{I}$, $\mathcal{G}$, and $\mathcal{O}$ are based on a collection of predicate symbols. Planning tasks from the same domain share the same sets of predicate symbols and operator schemata. Instantiating the operator schemata with all objects yields the actions A to the task. States are sets of logical atoms, i.e., instantiated predicates. Any action has a precondition, which is a formula that must hold in a state for the action to be applicable. Also, an action has an add- and a delete-list. These are sets of atoms, where each atom has a condition formula attached to it (in STRIPS, these condition formulae are trivially TRUE). If an action is applied, the atoms with satisfied condition in the add list are added to the state, and those with satisfied condition in the delete list are removed from the state. A plan is a sequence of actions that, when successively applied to the initial state, yields a state that satisfies the goal formula.

Ignoring the delete lists simplifies a task only if all formulae are negation free. In STRIPS, this is the case by definition. In general, for a fixed domain, any task can be polynomially transformed to have that property: compute the negation normal form to all formulae (negations only in front of atoms), then introduce for each negated atom $\neg B$ a new atom not-B and make sure it is TRUE in a state iff B is FALSE [Gazen and Knoblock, 1997]. In the following, we assume formulae to be negation free. We will investigate properties of the optimal relaxed heuristic h^+. For any state s in a planning task with actions A and goal condition $\mathcal{G}$, the *relaxed task to* s is the task defined by the same goal condition $\mathcal{G}$, the initial state s, and the action set A', which is identical to A except that all delete lists are empty. Then, $h^+(s)$ is the length of a shortest plan that solves the relaxed task to s, or $h^+(s) = \infty$ if there is no such plan.

We will be looking at the topology of search spaces with heuristic evaluation. The structural properties we will introduce do not depend on the planning framework. We therefore define them in a general manner, embedding planning state spaces as a special case.

Definition 1 *A* search space *is a* 4*-tuple* (S, E, G, s_0)*, where* S *is the set of* states*,* $E \subseteq S \times S$ *are the* state transitions*,* $\emptyset \neq G \subseteq S$ *are the* goal states*, and* $s_0 \in S$ *is the* initial state*.*

Given a planning task, the search space we look at is what is usually referred to as the *state space*. There, s_0 is simply the initial state $\mathcal{I}$ of the task. S is the set of states that are reachable from the initial state by successively applying actions from A, and E contains all pairs (s, s') where one action, executed in s, yields the state s'. G is the set of all states that satisfy the goal condition. Looking only at solvable instances, there is at least one such state.

Definition 2 *Given a search space* (S, E, G, s_0)*. The* goal distance *of* $s \in S$ *is*

$$gd(s) := min\{dist(s, s') \mid s' \in G\}$$

The distance $dist(s, s')$ between any two states is the length of a shortest path from s to s' in the directed graph given by S and E, or $dist(s, s') = \infty$ if there is no such path. Heuristic functions approximate gd.

Definition 3 *Given a search space* (S, E, G, s_0)*. A* heuristic *is a function* $h : S \mapsto N_0 \cup \{\infty\}$*, such that* $h(s) = 0 \Leftrightarrow gd(s) = 0$*.*

We require that a heuristic recognizes goal states, yielding $h(s) = 0$ if and only if $gd(s) = 0$, which is equivalent to $s \in G$. We allow heuristics to return $h(s) = \infty$, as search spaces can contain *dead ends*.

4 Dead Ends

Because state transitions in a search space are, in general, directed, there can be states from which no goal state is reachable.

Definition 4 *Given a search space* (S, E, G, s_0)*. A state* $s \in S$ *is a* dead end*, if* $gd(s) = \infty$*.*

If a local search algorithm runs into a dead end, it is lost. A heuristic function can return $h(s) = \infty$ to indicate that s might be a dead end. Desirably, it does so only on states s that really are dead ends.

Definition 5 *Given a search space* (S, E, G, s_0) *with a heuristic* h*.* h *is* completeness preserving*, if* $h(s) = \infty \Rightarrow gd(s) = \infty$*.*

With a completeness-preserving heuristic, we can safely prune states where $h(s) = \infty$. For planning tasks, if a task can not be solved even when ignoring the delete lists, then the task is unsolvable. Therefore, the h^+ function is completeness preserving. For the rest of the paper, we only consider those states where the heuristic value is less than ∞.

Definition 6 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h. The* relevant part *of the search space is $\{s \in S \mid h(s) < \infty\}$.*

Any search space with heuristic evaluation falls into one of the following four classes, with respect to dead ends.

Definition 7 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h. The search space is*

1. undirected, *if* $\forall (s, s') \in E : (s', s) \in E$
2. harmless, *if* $\exists (s, s') \in E : (s', s) \notin E$, *and* $\forall s \in S : gd(s) < \infty$
3. recognized, *if* $\exists s \in S : gd(s) = \infty$, *and* $\forall s \in S : gd(s) = \infty \Rightarrow h(s) = \infty$
4. unrecognized, *if* $\exists s \in S : gd(s) = \infty \wedge h(s) < \infty$

For each of our planning instances, we verified which of the above classes the state space belonged to. We say that a domain belongs to class i if the state spaces of all our instances belong to a class $j \leq i$, and at least one instance belongs to class i. We found the following.

1. *Blocksworld*, *Gripper*, and *Logistics* have undirected graphs.
2. *Grid*, *Miconic-STRIPS*, *Miconic-SIMPLE*, and *Movie* are directed, but do not have dead ends.
3. In *Assembly* and *Schedule*, all dead ends are recognized.
4. *Freecell*, *Miconic-ADL*, *Mprime*, and *Mystery* contain unrecognized dead ends.

In addition to our empirical analysis, the results from 1. and 2. can be shown analytically. For undirected graphs, such a method is described by Koehler and Hoffmann [2000]. For the domains in class four, it is also interesting to see how many unrecognized dead ends there are. We measure the percentage of such states in the relevant part of the state space, see Figure 1.

Domain	I_0	I_1	I_2	I_3	I_4
Freecell	0.0	1.1	1.2	2.6	3.0
Miconic-ADL	0.0	0.0	2.5	9.7	9.8
Mprime	18.8	29.3	50.0	58.0	69.6
Mystery	19.5	37.9	54.0	66.4	84.6

Figure 1: Percentage of unrecognized dead ends in the relevant part of the state space. Mean values for increasing task difficulty in different domains.

For each single domain in Figure 1, the sequence of columns gives a picture of how the values develop with increasing task difficulty. In each domain, the tasks are divided into five groups. The difficulty of a task in group i lies within interval I_i, where $I_0 \ldots I_4$ divide our range of difficulty in that domain into five parts of same size. Note that the intervals I_i are different for each domain, so the values within a column are not directly comparable.

As Figure 1 shows, the *Mprime* and *Mystery* tasks can contain a lot of unrecognized dead ends, and have the tendency to contain more of such dead ends the more difficult they get. For *Freecell* and *Miconic-ADL*, we can not conclude much more than that there can be unrecognized dead ends. It seems that the percentage grows with task difficulty, and that tasks with high percentage are out of the range of difficulty we could look at.

5 Search Space Topology

For SAT problems, the topology of search spaces with respect to the behavior of local search has been investigated by Frank et al. [1997]. As the basis of their work, Frank et al. formally define a partitioning of the search space into plateaus of different kinds. For our purposes, we extend their definitions to deal with our general notion of search spaces with heuristic evaluation, where edges can be directed.

Definition 8 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h. For $l \in N_0 \cup \{\infty\}$, a* plateau P of level l *is a maximal subset of S for which the induced subgraph in (S, E) is strongly connected, and $h(s) = l$ for each $s \in P$.*

Plateaus are regions that are equivalent under reachability aspects, and look the same from the point of view of the heuristic function. Obviously, each state s lies on exactly one plateau.

Definition 9 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h, and a plateau P. A state $s \in P$ is an* exit of P, *if there is a state $s' \notin P$ such that $(s, s') \in E$ and $h(s') \leq h(s)$. s is an* improving *exit, if, for at least one such s', $h(s') < h(s)$.*

Exits are states from which one can leave a plateau without increasing the value of the heuristic function. In undirected graphs, like are considered by Frank et al. [1997], leaving a plateau implies changing the value of the heuristic function, so all exits are improving there. According to the proportion of exits on a plateau, Frank et al. divide plateaus into four classes: local minima, benches, contours, and global minima. Taking account of directed edges, we have two types of exits, and define the following six different classes.

Definition 10 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h.*

1. *A* recognized dead end *is a plateau P of level $h = \infty$.*
2. *A* local minimum *is a plateau P of level $0 < h < \infty$ that has no exits.*
3. *A* plain *is a plateau P of level $0 < h < \infty$ that has at least one exit, but no improving ones.*
4. *A* bench *is a plateau P of level $0 < h < \infty$ that has at least one improving exit, and at least one state that is not an improving exit.*
5. *A* contour *is a plateau P of level $0 < h < \infty$ that consists entirely of improving exits.*
6. *A* global minimum *is a plateau P of level 0.*

Each plateau belongs to exactly one of the above classes. In our solvable instances, global minima are exactly the plateaus of level 0. With a completeness-preserving heuristic, recognized dead ends are irrelevant, and can be ignored. From local minima, there is no direct way of getting closer to the goal.

From benches, there is. From contours, one can get closer immediately. Plains behave as a kind of entrance to either local minima or benches.

Definition 11 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h. A* flat path *is a path where all states on the path have the same heuristic value. For a plateau P, the* flat region $FR(P)$ from P *is the set of all plateaus P' such that there is a flat path from some $s \in P$ to some $s' \in P'$.*

For a plain P, if there is at least one bench or contour in $FR(P)$, then P behaves similar to a bench, with at least one improving exit being within reach. Otherwise, starting in P, without increasing the value of the heuristic function, one will inevitably end up in a local minimum.

Definition 12 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h. A plain P* leads to benches *if $FR(P)$ contains some bench or contour. Otherwise, P* leads to local minima*.*

Based on the above definitions, and using an explicit search space representation, one can measure all kinds of structural parameters. Due to space restrictions, we only discuss some of the most interesting parameters here.

5.1 Local Minima

First, we are interested in the percentage of states that lie on local minima. Before doing this, we need to take a closer look at the definition of local minima. These are flat regions where all neighbors have higher evaluation. Stepping on to one of these neighbors does not necessarily improve the situation, though: it might be, for example, that the only exits on that neighbor lead back to the local minimum. In general, a local minimum is only the bottom of a valley, where what we really want to know about is the whole valley. Valleys are characterized by the property that one can not reach a goal state without increasing the value of the heuristic function.

Definition 13 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h. A state $s \in S$* has a full exit path*, if there is a path from s to a goal state s' such that the heuristic value of the states on the path decreases monotonically.*

One state on a plateau has a full exit path if and only if all states on that plateau do so. If a plateau has no full exit paths, then it is part of a valley.

Definition 14 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h. A* valley *is a maximal set V of plateaus such that no $P \in V$ has full exit paths, no $P \in V$ is a recognized dead end, and for all $P, P' \in V$, P is strongly connected to P'.*

The existence of valleys is, in any search space, equivalent to the existence of local minima. It turns out that, in 7 of the 13 competition domains, the state spaces of all our instances do not contain any local minima at all.

Hypothesis 1 *Let $\mathcal{P}$ be a planning task from any of the Assembly, Grid, Gripper, Logistics, Miconic-SIMPLE, Miconic-STRIPS, or Movie domains. Then, the state space of $\mathcal{P}$ does not contain any local minima under evaluation with h^+.*

In Figure 2, we show the mean percentage of states on valleys for those domains where we found local minima.

Domain	I_0	I_1	I_2	I_3	I_4
Blocksworld	9.8	28.1	37.5	50.1	55.6
Freecell	0.0	1.1	1.2	2.6	3.0
Miconic-ADL	0.0	0.0	2.5	9.7	9.8
Mprime	18.8	29.9	50.7	58.5	70.7
Mystery	19.5	38.5	55.1	68.0	89.4
Schedule	20.0	25.6	36.5	31.8	35.3

Figure 2: Percentage of states on valleys in the relevant part of the state space. Mean values for increasing task difficulty in different domains.

Any unrecognized dead end state lies in a valley. Therefore, the percentage of valleys is at least as high as the percentage of unrecognized dead ends for the domains shown in Figure 1. In *Freecell* and *Miconic-ADL*, the states on valleys are exactly the unrecognized dead ends in all our examples. In *Mprime* and *Mystery*, there can be more valley states. In *Blocksworld* and *Schedule*, values seem to approach an upper limit on our most difficult tasks. Computing maximum instead of the mean values shown in Figure 2, we found that some of our *Schedule* tasks contain up to 74.1% valley states. In our *Blocksworld* suite, however, the maximum valley percentage is constantly 59.3%, irrespective of difficulty.

5.2 Contours

We also measure the average percentage of states lying on contours that are not part of a valley—regions in the state space that are dominated by such contours are likely to be passed quickly by a local search algorithm. In *Movie*, the percentage is constantly 98.4%. In *Assembly*, *Logistics*, *Miconic-SIMPLE*, *Miconic-STRIPS*, and *Schedule*, between 30% and 60% of the relevant state space lie on such contours in our examples, and there is no clear tendency that the values decrease with task difficulty. In the remaining 7 domains, there is such a tendency. Values are particularly low in the *Blocksworld*, going down to 3.8% in our most difficult tasks.

5.3 Benches

For benches, the percentage of states alone is not a very informative parameter, as any plateau is a bench given it has at least one improving exit. What really matters is, how difficult is it to find such an exit? Possible criteria for this are the size of benches, or the proportion of improving exits. Here, we define another criterion that is—as will be shown in the next section—especially relevant for FF's search algorithm. The criterion is named *maximal exit distance*. We measure that distance for what we call *bench-related* plateaus. These are benches, and plains leading to benches. Recall Definitions 11 and 12.

Definition 15 *Given a search space (S, E, G, s_0) with a completeness-preserving heuristic h. For a state s on a bench-related plateau, the* exit distance $ed(s)$ *is the length of*

a shortest flat path from s to some s' such that there is some s'' with $(s', s'') \in E, h(s'') < h(s')$. The maximal exit distance of a bench-related plateau P is $med(P) := max\{ed(s) \mid s \in P\}$.

The maximal exit distance in a search space is the maximum over the maximal exit distances of all bench-related plateaus, or 0 if there are no bench-related plateaus. It turns out that, in 5 of the competition domains, the maximal exit distance is constant across all our examples.

Hypothesis 2 *To any of the Gripper, Logistics, Miconic-SIMPLE, Miconic-STRIPS, or Movie domains, there is a constant c, such that, for all tasks $\mathcal{P}$ in that domain, the maximal exit distance in the state space of $\mathcal{P}$ is at most c under evaluation with h^+.*

In the listed domains, all our examples have maximal exit distance 1, so the constant $c = 1$ fulfills the hypothesized property there. The crucial point is that, in those 5 domains, there apparently is an upper limit to the maximal exit distance. In contrast to this, computing mean values, we found that the mean maximal exit distance grows with difficulty in our suites from 7 of the remaining 8 domains. In *Mprime*, mean values show a lot of variance, making it hard to draw any conclusions. See Figure 3.

Domain	I_0	I_1	I_2	I_3	I_4
Assembly	0	1.2	1.6	2.4	3.5
Blocksworld	2.8	3.3	3.8	4.9	5.0
Freecell	0	0	1	1	1.5
Grid	2.6	3.2	3.1	3.8	4.3
Miconic-ADL	1	1	1	1.4	2.3
Mprime	1.9	2.6	2.1	2.0	2.2
Mystery	1.1	1.4	1.5	1.2	2.3
Schedule	1	1	1.2	1.9	2.0

Figure 3: Maximal exit distance. Mean values for increasing task difficulty in different domains.

6 A Planning Domain Taxonomy

Our approach divides planning domains into a taxonomy of different classes with respect to the h^+ heuristic, depending on which dead end class they belong to, whether there can be local minima, and whether there is an upper limit to the maximal exit distance. See a schematic overview of our results in Figure 4.

Remember that the existence of unrecognized dead ends implies the existence of valleys, which implies the existence of local minima. The overview in Figure 4 gives an appealing impression of the kind of domains that state-of-the-art heuristic planners work well on: The "simple" domains are in the left bottom corner, while the "demanding" ones are in the top right corner. In fact, in the AIPS-2000 competition, the *Freecell* and *Miconic-ADL* domains constituted much more of a problem to the heuristic planners than, for example, the *Logistics* domain did.

The majority of the competition domains lie on the "simple" left bottom side of our taxonomy. In fact, this phenomenon gets even stronger when looking at other commonly

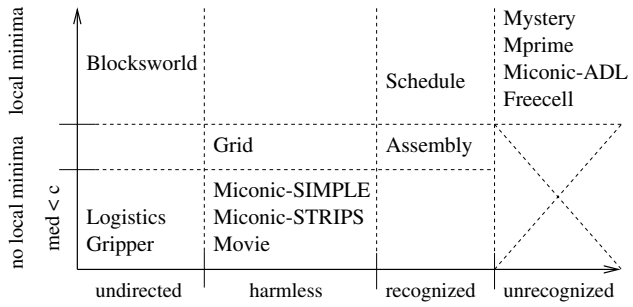


Figure 4: A taxonomy for planning domains, overviewing our results.

used planning benchmark domains: From our 20 domains, 14 do not exhibit any local minima. In 8 of those 14 domains, the mean maximal exit distance does not grow with difficulty. In the *Briefcaseworld*, for example, the distance is apparently bounded by $c = 3$.

For domains without local minima and with bounded maximal exit distance, we can be precise about simplicity. Consider the following algorithm, working on a search space (S, E, G, s_0) with a heuristic h.

$s := s_0$
while $h(s) \neq 0$ **do**
    do breadth first search for s', $h(s') < h(s)$
    $s := s'$
endwhile

This algorithm has been termed Enforced Hill-climbing by Hoffmann [2000], and is used in FF.

Proposition 1 *Let $\mathcal{D}$ be a set of search spaces with heuristics, such that no search space contains a local minimum, and med is an upper limit to the maximal exit distance. Say we have a search space $(S, E, G, s_0) \in \mathcal{D}$, with heuristic h. Let b be the maximal number of outgoing edges of any state. Then, started on (S, E, G, s_0), Enforced Hill-climbing will find a goal state after considering $O(h(s_0) * b^{med+1})$ states.*

Without local minima, each iteration of Enforced Hill-climbing crosses a bench-related region or a contour, so it finds a better state at maximal depth $med + 1$, considering $O(b^{med+1})$ states. Each iteration improves the heuristic value by at least one, so after at most $h(s_0)$ iterations, a goal state is reached.

Reconsider the terminology introduced at the beginning of Section 3. Say we have a planning domain with operator schemata $\mathcal{O}$. Any task specifies, amongst other things, the set of objects O, yielding the action set A. An obvious upper limit to the number of outgoing edges in the task's state space is $|A|$. Furthermore, if the longest add list of any action has size k, then, for non dead end states s, $h^+(s) \leq k * |A|$. This is because with empty delete lists, each atom needs to be added at most once. Finally, $|A|$ and k are polynomial in $|O|$ for fixed $\mathcal{O}$. Thus, considering only the solvable tasks from a domain, applying Proposition 1 gets us the following. If h^+ does not yield any local minima, and produces a constant maximal exit distance, then Enforced Hill-climbing, using h^+, finds a goal state to each task by looking at a number

of states polynomial in $|O|$.[1]

7 Explaining FF's Runtime Behavior

To give an example of how our results under evaluation with h^+ carry over to existing approximation techniques, we ran the same experiments, using the FF heuristic. The results are summarized in Figure 5.

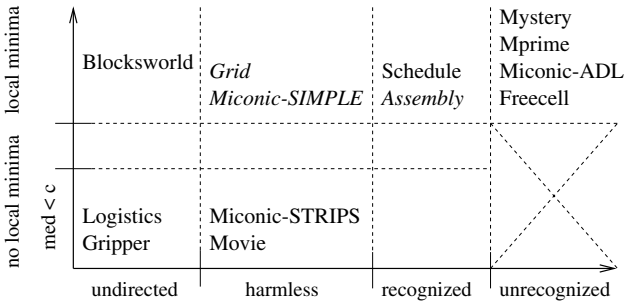


Figure 5: Results overview for the FF heuristic.

Comparing Figure 5 with Figure 4, one sees that there are three domains where local minima arise, when using the FF heuristic instead of h^+. However, the averaged valley percentage is below 6% in *Grid*, below 2% in *Assembly*, and below 0.3% in *Miconic-SIMPLE*. Four of the domains that are simple with h^+ stay simple with the FF heuristic.

As Hoffmann [2000] describes, the FF system uses the Enforced Hill-climbing algorithm as its search method. For STRIPS planning tasks, it is easy to see that, like it is the case for the h^+ heuristic, FF's heuristic estimate to any state is bounded by the number of actions. Thus, if Hypotheses 1 and 2 are true for the STRIPS domains *Gripper*, *Logistics*, *Miconic-STRIPS*, and *Movie*, under evaluation with the FF heuristic, then Enforced Hill-climbing, using that heuristic, solves the tasks in each of these domains by evaluating polynomially many states.

8 Conclusion and Outlook

The intuition that many planning benchmarks are "simple" in some sense is not new to the planning community. What the author personally likes most about the presented work is that it provides a formal notion of what simplicity, in that context, might mean. We give empirical data supporting that many benchmarks are, in fact, simple in that formal sense. The work provides insights into fundamental structural differences between different planning domains, and offers explaining the success of FF—and possibly of other state-of-the-art heuristic planners—as utilizing the simplicity of the benchmarks.

The presented results are preliminary to the effect that observations are made on a collection of comparatively small planning tasks. The stated hypotheses must be verified. For the h^+ function, we are going to prove our hypotheses analytically. For the FF and HSP heuristic functions, we are going to take samples from the state spaces of larger tasks.

Practically, we see the benefits of our results in mainly three areas. Firstly—which is a line of work that we are currently exploring—one can try to recognize simple planning tasks automatically, and thereby predict the runtime behavior of FF or other heuristic planners. Secondly, knowing about the strengths and weaknesses of existing heuristic functions may help in designing better ones. Finally, a better understanding of the structural differences between planning domains may help in designing more challenging benchmarks.

[1] In domains where this holds, deciding plan existence is in NP: To any solvable task, there is a solution plan with at most $h^+(\mathcal{I}) * (med + 1)$ steps, i.e., a plan of polynomial length.

Acknowledgments

The author thanks Bernhard Nebel for discussions on the presented work in all stages of its development. Thanks also go to Malte Helmert for many useful comments on an early version of the paper.

References

[Bacchus and Nau, 2001] Fahiem Bacchus and Dana Nau. The 2000 AI planning systems competition. *The AI Magazine*, 2001. Forthcoming.

[Bonet and Geffner, 2001] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 2001. Forthcoming.

[Bonet *et al.*, 1997] Blai Bonet, Gábor Loerincs, and Héctor Geffner. A robust and fast action selection mechanism for planning. In *Proc. AAAI-97*, pages 714–719. MIT Press, July 1997.

[Bylander, 1994] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, 1994.

[Edelkamp, 2000] S. Edelkamp. Heuristic search planning with BDDs. In *ECAI-Workshop: PuK*, 2000.

[Frank *et al.*, 1997] Jeremy Frank, Peter Cheeseman, and John Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.

[Gazen and Knoblock, 1997] B. Cenk Gazen and Craig Knoblock. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In *Proc. ECP'97*, pages 221–233. Springer-Verlag, September 1997.

[Hoffmann, 2000] Jörg Hoffmann. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Proc. ISMIS-00*, pages 216–227. Springer-Verlag, October 2000.

[Koehler and Hoffmann, 2000] Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 12:338–386, 2000.

[McDermott, 2000] Drew McDermott. The 1998 AI planning systems competition. *The AI Magazine*, 21(2):35–55, 2000.

Reviving Partial Order Planning

XuanLong Nguyen & Subbarao Kambhampati*
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
Email: {xuanlong,rao}@asu.edu

Abstract

This paper challenges the prevailing pessimism about the scalability of partial order planning (POP) algorithms by presenting several novel heuristic control techniques that make them competitive with the state of the art plan synthesis algorithms. Our key insight is that the techniques responsible for the efficiency of the currently successful planners–viz., distance based heuristics, reachability analysis and disjunctive constraint handling–can also be adapted to dramatically improve the efficiency of the POP algorithm. We implement our ideas in a variant of UCPOP called REPOP[1]. Our empirical results show that in addition to dominating UCPOP, REPOP also convincingly outperforms Graphplan in several "parallel" domains. The plans generated by REPOP also tend to be better than those generated by Graphplan and state search planners in terms of execution flexibility.

1 Introduction

Most recent strides in scaling up planning have centered around two dominant themes - heuristic state space planners, exemplified by UNPOP[20], HSP-R[3], and CSP-based planners, exemplified by Graphplan[2] and SATPLAN [14] . This is in stark contrast to planning research up to five years ago, when most of the efforts were focused on scaling up partial order planners[19; 27; 15; 23; 11; 13]. Despite such efforts, the partial order planners continue to be extremely slow and are not competitive with the fastest state search-based and CSP-based planners. Indeed, the recent advances in plan synthesis have generally been (mis)interpreted as establishing the supremacy of state space and CSP-based approaches over POP approaches.

Despite its current scale-up problems, partial order planning remains attractive over state space and CSP-based planning for several reasons. The least commitment inherent in partial order planning makes it one of the more open planning frameworks. This is evidenced by the fact that most existing architectures for integrating planning with execution, information gathering, and scheduling are based on partial order planners. In [25], Smith argues that POP-based frameworks offer a more promising approach for handling domains with durative actions, and temporal and resource constraints as compared to other planning approaches. In fact, most of the known implementations of planning systems capable of handling temporal and durative constraints –including IxTET [6] as well as NASA's RAX [10]–are based on the POP algorithms. Even for simpler planning domains, partial order planners search for and output partially ordered plans that offer a higher degree of execution flexibility. In contrast, none of the known state space planners can find parallel plans efficiently [8], and CSP planners such as Graphplan only generate a very restricted class of parallel plans (see Section 5).

The foregoing motivates the need for improving the efficiency of POP algorithms. We show in this paper that the insights and techniques responsible for the advances in plan synthesis made in the recent years in the context of state-based and CSP-based planners are largely adaptable to POP algorithms. In particular, we present novel methods for adapting distance based heuristics, reachability analysis and disjunctive constraint processing techniques to POP algorithms. Distance-based heuristics are used as the basis for ranking partial plans and as flaw selection methods. The other two techniques are used for efficiently enforcing the consistency of the partial plans–by detecting implicit conflicts and resolving them.

Our methods help scale up POP algorithms dramatically–making them competitive with respect to state space planners, while preserving their flexibility. We present empirical studies showing that REPOP, a version of UCPOP [27] enhanced by our ideas, can perform competitively with other existing approaches in many planning domains. In particular, REPOP appears to scale up much better than Graphplan in the parallel domains we tried. More importantly, the solutions REPOP generates are generally shorter in length, and provide significantly more execution flexibility [25].

The paper is organized as follows. In the next section we will briefly review the basics of the POP algorithm. Section 3 describes how distance based heuristics can be adapted to rank partial plans. Section 4 shows how unsafe links flaws can be generalized and resolved efficiently. Section 5 reports empirical evaluations of the techniques that have been described. Section 6 discusses related work, and Section 7 summarizes the contributions of this work.

2 Background on Partial Order Planning

In this paper we consider the simple STRIPS representation of classical planning problems, in which the initial world state I,

*This research is supported in part by the NSF grant IRI-9801676, AFOSR grant F20602-98-1-0182 and the NASA grants NAG2-1461 and NCC-1225. We thank David Smith, Malik Ghallab, Austin Tate, Dan Weld, Terry Zimmerman, Biplav Srivastava, Minh B. Do, and the IJCAI referees for critical comments on the previous drafts of this paper.

[1] UCPOP [27] → UNPOP [20] → REPOP. REPOP's source code is available from *http://rakaposhi.eas.asu.edu/repop.html.*

goal state G and the set of deterministic actions Ω are given. Each action $a \in \Omega$ has a precondition list and an effect list, denoted respectively as $Prec(a), Eff(a)$. The planning problem involves finding a plan that when executed from the initial state I will achieve the goal G.

A tutorial introduction to POP algorithms can be found in [27]. We will provide a brief review here. Most POP algorithms can be seen as searching in the space of partial plans. A **partial plan** is a five-tuple: $\mathcal{P} = (\mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{OC}, \mathcal{UL})$, where $\mathcal{A} \subseteq \Omega$ is a set of (ground) actions,[2] $\mathcal{O}$ is a set of ordering constraints over $\mathcal{A}$, and $\mathcal{L}$ is a set of causal links over $\mathcal{A}$.[3] A causal link is of the form $a_i \xrightarrow{p} a_j$, and denotes a commitment by the planner that the precondition p of action a_j will be supported by an effect of action a_i. $\mathcal{OC}$ is a set of open conditions, and $\mathcal{UL}$ is a set of unsafe links. An **open condition** is of the form (p, a), where $p \in Prec(a)$ and $a \in \mathcal{A}$, and there is *no* causal link $b \xrightarrow{p} a \in \mathcal{L}$. Loosely speaking, the open conditions are preconditions of actions in the partial plan which have not yet been achieved in the current partial plan. A causal link $a_i \xrightarrow{p} a_j$ is called **unsafe** if there exists an action $a_k \in \mathcal{A}$ such that (i) $\neg p \in Eff(a_k)$ *and* (ii) $\mathcal{O} \cup \{a_i \prec a_k \prec a_j\}$ is consistent. In such a case, a_k is also said to **threaten** the causal link $a_i \xrightarrow{p} a_j$. Open conditions and unsafe links are also called **flaws** in the partial plan. Therefore a solution plan can be seen as a partial plan with no flaws (i.e., $\mathcal{OC} = \emptyset$ and $\mathcal{UL} = \emptyset$).

The POP algorithm starts with a null partial plan $\mathcal{P}$ and keeps refining it until a solution plan is found. The **null partial plan** contains two dummy actions $a_0 \prec a_\infty$ where the preconditions of a_∞ correspond to the top level goals of the problem, and the effects of a_0 correspond to the conditions in the initial state. The null plan has no causal links or unsafe link flaws, but has open condition flaws corresponding to the preconditions of a_∞ (top level goals).

A **refinement** step involves selecting a flaw in the partial plan $\mathcal{P}$, and *resolving* it, resulting in a new partial plan. When the flaw chosen is an open condition (p, a), an action b needs to be selected that achieves p. b can be a new action in Ω, or an action that is already in $\mathcal{A}$. The sets $\mathcal{OC}$, $\mathcal{O}$, $\mathcal{L}$ and $\mathcal{UL}$ also need to be updated with respect to b. Secondly, when the flaw chosen is an unsafe link $a_i \xrightarrow{p} a_j$ that is threatened by action a_k, it can be repaired by either **promotion**, i.e adding ordering constraint $a_k \prec a_i$ into $\mathcal{O}$, or **demotion**, i.e adding $a_j \prec a_k$ into $\mathcal{O}$.

The efficiency of POP algorithms depends critically on the way partial plans are selected from the search queue, and the strategies used to select and resolve the flaws. In Section 3 we present several distance-based heuristics for ranking partial plans in the search queue. Section 4 introduces the disjunctive constraint representation for efficiently handling unsafe link flaws, and reachability analysis for generalizing the notion of unsafe links to include implicit conflicts in the plan.

3 Heuristics for ranking partial plans

In choosing a plan from the search queue for further refinement, we are naturally interested in plans that are likely to lead to a solution with a minimum number of refinements (flaw resolutions). As we handle the unsafe links in a significantly different way than standard UCPOP (see Section 4), the only remaining category of flaws to be resolved are open condition flaws. Consequently one way of ranking plans in the search queue is to estimate the minimum number of new actions needed to resolve all the open condition flaws.

Definition 1 (h*) *Given a partial plan $\mathcal{P}$, let $h^*(\mathcal{P})$ denote the minimum number of new actions that need to be added to $\mathcal{P}$ to make it a solution plan.*

$h^*(\mathcal{P})$ can be seen as the number of actions that, when executed from the initial state I in some order, will achieve the set of subgoals $S = \{p | (p, a) \in \mathcal{OC}\}$. In this sense, this is similar to estimating the number of actions needed to achieve a state from the given initial state in state search planners [3; 21], but for two significant differences: (i) the propositions in S are not necessarily in the same world state and (ii) the set of actions that achieve S cannot conflict with the set of actions and causal links already present in $\mathcal{P}$.

A well-known heuristic for estimating h^* involves simply counting the number of open conditions in the partial plan [13].

Heuristic 1 (Open conditions heuristic) $h_{oc}(\mathcal{P}) = |\mathcal{OC}|$

This estimate is neither admissible nor informed in many domains, because it treats every open condition equally. In particular, it is ineffective when some open conditions require more actions to achieve than others.

We would like to have a closer estimate of h^* function without insisting on admissibility. To do this, we need to take better account of subgoal interactions[21]. Accounting for the negative interactions in estimating h^* can be very tricky, and is complicated by the fact that the subgoals in S may not be in the same state. Thus we will start by ignoring the negative interactions. This has three immediate consequences: (i) the set of unsafe links $\mathcal{UL}$ becomes empty. (ii) the actions needed in achieving a set of subgoals S will have no conflicts with the set of actions $\mathcal{A}$ and and the causal links $\mathcal{L}$ already present in $\mathcal{P}$. and (iii) a subgoal p once achieved from the initial state can never become untrue. Given these consequences, it does not matter much that the subgoals in S are not necessarily present in the same world state, since the minimum number of actions needed for achieving such a set of subgoals in any given temporal ordering is the same as the minimum cost of achieving a state comprising all those subgoals.

The foregoing justifies the adaptation of many heuristic estimators for ranking the goodness of states in state search planners. Most of the early heuristic estimators used in state search not only ignore negative interactions, but also make the stronger assumption of subgoal independence[3; 20]. A few of the recent ones, [21; 9] however account for the positive interactions among subgoals (while still ignoring the negative interactions). It is this latter class of heuristics that we focus on for use in partial order planning. Specifically, to account for the positive interactions, we exploit the ideas for estimating the cost of achieving a set of subgoals S using a serial planning graph.[4]

Specifically, we build a planning graph starting from the initial state I. Let $lev(p)$ be the index of the level in the planning graph that a proposition p first appears, and $lev(S)$ be the index of the first level at which all propositions in S appear. Let p_S be the proposition in S such that $lev(p_S) = \max_{p_i \in S} lev(p_i)$.

[2] Although partial order planners are capable of handling partially instantiated action instances, we restrict our attention to ground action instances.

[3] Strictly speaking $\mathcal{A}$ should be seen as a set of "steps", where each step is mapped to an action instance [15].

[4] We assume that the readers are familiar with the planning graph data structure, which is used in Graphplan algorithm[2].

p_S will *possibly* be the last proposition in S that is achieved during execution. Let a_S be an action in the planning graph that achieves p_S in the level $lev(p_S)$. We can achieve p_S by adding a_S to the plan. Introduction of a_S changes the set of goals to be achieved to $S' = S + Prec(a_S) - Eff(a_S)$. We can express the cost of S in terms of the cost of a_S and S':

$$cost(S) := cost(a_S) + cost(S + Prec(a_S) - Eff(a_S)) \quad (1)$$

where $cost(a_S) = 1$ if $a_S \notin \mathcal{A}$ and 0 otherwise. Since $lev(Prec(a_S))$ is strictly smaller than $lev(p_S)$, recursively applying Equation 1 to its right hand side will eventually express $cost(S)$ in terms of $cost(I)$ (which is zero), and the costs of actions a_S. The process is quite efficient as the number of applications is bounded by $lev(S)$.

Heuristic 2 (Relax heuristic) *$h_{relax}(\mathcal{P}) = cost(S)$, where $S = \{p|(p,a) \in \mathcal{OC}\}$, and $cost(S)$ is computed using the recurrence relation 1.*

Given such a heuristic estimate, plans in the search queue are ranked with the evaluation function: $f(\mathcal{P}) = |\mathcal{A}| + w * h(\mathcal{P})$. The parameter w is used to increase the greediness of the heuristic search and is set to 5 by default.

4 Enforcing consistency of partial plans

The consistency of a partial plan is ensured through the handling of its unsafe links. In this section we describe two ways of improving this phase. The first involves posting disjunctive constraints to resolve unsafe links. The second involves detecting implicit conflicts (unsafe links) using reachability analysis.

4.1 Disjunctive representation of ordering constraints

Normally, an unsafe link $a_i \xrightarrow{p} a_j$ that is in conflict with action a_k is resolved by either promotion or demotion, that is, splitting the current partial plan into two partial plans, one with the constraint $a_k \prec a_i$, and the other with the constraint $a_j \prec a_k$. A problem with this premature splitting is that a single failing plan gets unnecessarily multiplied into many descendant plans poisoning the search queue significantly. A much better idea, first proposed in [16], is to resolve the unsafe link by posting a disjunctive ordering constraint that captures both the promotion and demotion possibilities, and incrementally simplify these constraints by propagation techniques. This way, we can detect many failing plans before they get selected for refinement.

Specifically, an unsafe causal link $a_i \xrightarrow{p} a_j$ that is in conflict with action a_k can be resolved by simply adding a disjunctive ordering constraint $(a_k \prec a_i) \vee (a_j \prec a_k)$ to the plan.

We use the following procedure for simplifying the disjunctive orderings. Whenever an open condition (p, a) is selected and resolved by either adding a new action or reusing an action b in the partial plan, we add a new ordering constraint $b \prec a$ to $\mathcal{O}$, followed by repeated application of the constraint propagation rules below:

- $(a_1 \prec a_2) \in \mathcal{O} \wedge (a_2 \prec a_3) \in \mathcal{O} \Rightarrow \mathcal{O} \leftarrow \mathcal{O} \cup (a_1 \prec a_3)$
- $(a_1 \prec a_2) \in \mathcal{O} \wedge (a_2 \prec a_1) \in \mathcal{O} \Rightarrow$ False
- $(a_1 \prec a_2) \in \mathcal{O} \wedge (a_2 \prec a_1 \vee a_3 \prec a_4) \in \mathcal{O} \Rightarrow$
  $\mathcal{O} \leftarrow \mathcal{O} \cup (a_3 \prec a_4)$
  $\mathcal{O} \leftarrow \mathcal{O} - (a_2 \prec a_1 \vee a_3 \prec a_4)$

The first two propagation rules are already done as part of POP algorithm to ensure the transitive consistency of ordering constraints. The third rule is a unit propagation rule over ordering constraints. This propagation both reduces the disjunction and detects infeasible plans ahead of time. When all the open conditions have already been established and there are still disjunctive constraints left in the plan, the remaining disjunctive constraints are then split into the search space [16].

4.2 Detecting and Resolving implicit conflicts through reachability analysis

Although the unsafe link detection and resolution steps in the POP algorithm are meant to enforce consistency of the partial plan, often times they are too weak to detect implicit inconsistencies. In particular, the procedure assumes that a link $a_i \xrightarrow{p} a_j$ is threatened by an action a only if a has an effect $\neg p$. Often a might have an effect q (or precondition r) such that no legal state can have p and q (or p and r) true together. Detecting and resolving such implicit interactions can be quite helpful in weeding out inconsistent partial plans from the search space.

In order to do implicit conflict detection as described above, we need to have (partial) information about the properties of reachable states. Interestingly, such reachability information has played a significant role in the scale-up of state space planners, motivating the development of procedures for identifying mutex constraints, state invariants and memos etc. [2; 7; 5] (we shall henceforth use the term mutex to denote all these types of reachability information). One simple way of producing reachability information is to expand Graphplan's planning graph structure, armed with mutex propagation procedure [2]. The mutexes present at the level where the graph levels off are state invariants [21].

Exploiting the reachability information to check consistency of partial plans requires identifying the feasibility of the world states that any eventual execution of the partial plan must pass through. Although partial order plans normally do not have explicit state information associated with them, it is nevertheless possible to provide partial characterization of the states their execution must pass through. Specifically, we define the general notion of cutsets as follows:

Definition 2 (Cutsets) *Pre- and post- cutsets, C^- and C^+ of an action a_k in a plan $\mathcal{P}$ are defined as $C^-(a_k) = Prec(a_k) \cup L(a_k)$, and $C^+(a_k) = Eff(a_k) \cup L(a_k)$, where $L(a_k)$ is the set of all conditions p such that there exists a link $a_i \xrightarrow{p} a_j$ where a_i is necessarily before a_k, and a_j is necessarily after a_k*

The pre- and post-cutsets of an action can be seen as partial description of world states that *must* hold before and after the action a_k. If these partial descriptions violate the properties of the reachable states, then clearly the partial plan cannot be refined into an executable solution.

Proposition 1 *If there exists a cutset that contains a mutex, then the partial plan is provably invalid and can be pruned from the search queue.*

While this proposition allows us to detect and prune inconsistent plans, it is often inefficient to wait until the plan becomes inconsistent. Detecting and resolving implicit conflicts is essentially a more active approach that *prevents* a partial plan from becoming inconsistent by this proposition. Specifically, we generalize the notion of unsafe links as follows:

Definition 3 *An action a_k is said to have a* **conflict** *with a causal link $a_i \xrightarrow{p} a_j$ if (i) $\mathcal{O} \cup \{a_i \prec a_k \prec a_j\}$ is consistent and (iii) either $Prec(a_k) \cup \{p\}$ or $Eff(a_k) \cup \{p\}$ contains a mutex. A causal link $a_i \xrightarrow{p} a_j$ is* **unsafe** *if it has a conflict with some action in the partial plan.*

These notions of conflict and unsafe link subsume the original notions of *threat* and *unsafe link* introduced in Section 2, because $\neg p \in Eff(a_k)$ also implies that $Eff(a_k) \cup \{p\}$ is a mutex. Therefore the generalized notion of unsafe links result in detecting a larger number of (implicit) conflicts (unsafe links) present in a partial plan.

Once the implicit conflicts are detected, they are resolved by posting disjunctive orderings as described in the previous subsection. As we shall see later, the combination of disjunctive constraints and detection of implicit conflicts through reachability information leads to quite robust improvements in planning performance.

5 Empirical Evaluation

We have implemented the techniques introduced in this paper on top of UCPOP[27], a popular partial order planning algorithm. We call the resulting planner REPOP. As mentioned in Section 2, both UCPOP and REPOP are given ground action instances, and thus neither of them have to deal with variable binding constraints. Both UCPOP and REPOP use the LIFO as the order in which open condition flaws are selected for resolution. Our empirical studies compare REPOP to UCPOP as well as Graphplan[2] and AltAlt[21], which represent two currently popular approaches (CSP search and state space search) in plan synthesis. All these planners are written in Lisp. In the case of Graphplan, we used the Lisp implementation of the original algorithm, enhanced with EBL and DDB capabilities [17]. AltAlt [22] is a state-of-the-art heuristic regression state search planner, that has been shown to be significantly faster than HSP-R [3]. The empirical studies are conducted on a 500 MHz Pentium-III with 256MB RAM, running Linux. The test suite of problems were taken from several benchmark planning domains from the literature. Some of these, including gripper, rocket world, blocks world and logistics are "parallel" domains which admit solutions with loosely ordered steps, while others, such as grid world and travel world admit only serial solutions.

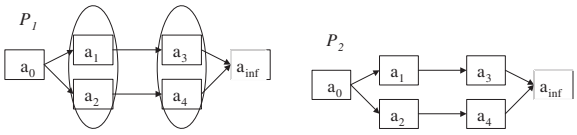
Efficiency of Synthesis: In Table 1, we report the total running times for the REPOP algorithm, including the preprocessing time for computing the mutex constraints (using bilevel planning graph structures [18]). Table 1 shows that REPOP exhibits dramatic improvements from its base planner, UCPOP, in gripper, logistics and rocket domains–all of which are "parallel domains." For instance, REPOP is able to comfortably generate plans with up to 70 actions in logistics and gripper domains, a feat that has hither-to been significantly beyond the reach of partial order planners. More interesting is the comparison between REPOP and the non-partial order planners. In the parallel domains, REPOP manages to outperform Graphplan. Although REPOP still trails state search planners such as AltAlt, these latter planners can only generate serial plans.

Despite the impressive performance of the REPOP over parallel domains, it remains ineffective in "serial" domains including the grid, 8-puzzle and travel world, which admit only totally ordered plan solutions. We suspect that part of the reason for this may be the inability of our heuristics to adequately account for negative interactions. Indeed, we found that the normal open conditions heuristic h_{oc} is better than our relaxed heuristic on these problems. It may also be possible that the least commitment strategies employed by the POP algorithms become a burden in serial domains, since eventually all actions need to be ordered with respect to each other. One silverlining in this matter is that most of the domains where POP algorithms are supposed to offer advantages are likely to be parallel domains from the planner's perspective–either because the actions will have durations (making the serial/parallel distinction moot) or because we want solution output by the planner to offer some degree of scheduling flexibility.

Plan Quality: We also evaluated the quality of plans generated by REPOP, since plan quality is seen as an important issue favoring POP algorithms. To quantify the quality of plans generated, we consider three metrics: (i) the cumulative cost of the actions included in the plan (ii) the minimum time needed for executing the plan and (iii) the scheduling (execution) flexibility of the plan.

For actions with uniform cost, the action cost is equal to the number of actions in the plan. Table 1 shows that REPOP produces plans with lower action cost compared to both Graphplan and AltAlt in all but one problem (*rocket-ext-b*).

We measure the minimum execution time in terms of the *makespan* of the plan, which is loosely defined as the minimum number of time steps needed to execute the plan (taking the possibility of concurrent execution into consideration). Makespan for the plans produced by Graphplan is just the number of steps in the plan, while the makespan for plans produced by AltAlt (and other state space planners) is equal to the number of actions in the plan. For a partially ordered plan P generated by REPOP, the makespan is simply the length of the longest path between a_0 and a_∞. Specifically, $makespan(P) = \max_{a \in P} est(a)$, where $est(a)$ is the earliest start time step for the (instantaneous) action a. To compute est, we can start by initializing est to 0 for all $a \in P$. Next, we repeatedly update them until fixpoint using the following rule: For all $(a_i \prec a_j) \in \mathcal{O}, est(a_j) := \max\{est(a_j), 1 + est(a_i)\}$. Table 1 shows that the solution plans generated by REPOP are highly parallel, since the makespans of these plans are significantly smaller than the total number of actions. Graphplan's solutions have smaller makespans in several problems, but at the expense of having substantially larger number of actions.



(a) A parallel plan generated by Graphplan

(b) A partially ordered plan

Figure 1: Example illustrating the execution flexibility of partially ordered plans over (Graphplan's) parallel plans.

Finally, we measure the execution flexibility of a plan in terms of the number of actions in the plan that do not have any precedence relations among them. The higher this measure, the higher the number of orders in which a plan can be executed ("scheduled"). Figure 1 illustrates a parallel plan P_1 and a partially ordered plan P_2, which are generated by Graphplan and REPOP, respectively. Both plans have 4 actions and a makespan value of 2, but P_2 is noticeably more flexible than

Problem	UCPOP (time)	REPOP			Graphplan			AltAlt	
		Time	#A/ #S	#flex	Time	#A/ #S	#flex	Time	#A
gripper-8	–	1.01	21/ 15	.57	66.82	23/ 15	.69	.43	21
gripper-10	–	2.72	27/ 19	.59	47min	29/ 19	.71	1.15	27
gripper-12	–	6.46	33/ 23	.61	–	–	–	1.78	33
gripper-20	–	81.86	59/ 39	.68	–	–	–	15.42	59
rocket-ext-a	–	8.36	35/ 16	2.46	75.12	40/ 7	7.15	1.02	36
rocket-ext-b	–	8.17	34/ 15	7.29	77.48	30/ 7	4.80	1.29	34
logistics.a	–	3.16	52/ 13	20.54	306.12	80/ 11	6.58	1.59	64
logistics.b	–	2.31	42/ 13	20.0	262.64	79/ 13	5.34	1.18	53
logistics.c	–	22.54	50/ 15	16.92	–	–	–	4.52	70
logistics.d	–	91.53	69/ 33	22.84	–	–	–	20.62	85
bw-large-a(9)	45.78	(5.23) –	(8/ 5) –	(2.75) –	14.67	11/4	2.0	4.12	9
bw-large-b(11)	–	(18.86) –	(11/ 8) –	(3.28) –	122.56	18/ 5	2.67	14.14	11
bw-large-c(15)	–	(137.84) –	(17/ 10) –	(5.06) –	–	–	–	116.34	19
travel1	149.74	(4.32) –	(9/9) –	(0.0) –	0.32	9/ 9	0.0	0.53	9
simple-grid1	56.40	(0.0) –	(6/ 6) –	(0.0) –	0.42	6/ 6	0.0	1.48	6
simple-grid2	–	(2.43) –	(10/ 10) –	(0.0) –	0.95	10/ 10	0.0	1.58	10
simple-grid3	–	–	–	–	3.96	16/ 16	0.0	15.12	16

Table 1: "Time" shows *total* running times in cpu seconds, and includes the time for any required preprocessing. Dashed entries denote problems for which no solution is found in 3 hours or 250MB. Parenthesized entries (for blocks world, travel and grid domains) indicate the performance of REPOP when using h_{oc} heuristic. #A and #S are the action cost and time cost respectively of the solution plans. "flex" is the execution flexibility measure of the plan (see below).

P_1, since P_1 implies ordering constraints such as $a_1 \prec a_4$ and $a_2 \prec a_3$, but P_2 does not. To capture this flexibility, we define, for each action a, $flex(a)$ as the number of actions in the plan that *do not* have any (direct or indirect) ordering constraint with a. $flex(P)$ is defined as the average value of $flex$ over all the actions in the plan. It is easy to see that for a serial plan P, $\forall_{a \in P} flex(a) = 0$, and consequently $flex(P) = 0$. In our example in Figure 1, $flex(a) = 1$ for all a in P_1, and $flex(a) = 2$ for all a in P_2. Thus, $flex(P_1) = 1$ and $flex(P_2) = 2$. It is easy to see that P_2 can be executed in more ways than P_1. Table 1 reports the $flex()$ value for the solution plans. As can be seen, plans generated by REPOP have substantially larger average values of $flex$ than Graphplan in blocks world and logistics, and similar values in gripper. Graphplan produces a more flexible plan in only one problem in the rocket domain.

Problem	UCPOP	+CE	+HP	+HP+CE
gripper-8	*	6557/ 3881	*	1299/ 698
gripper-10	*	11407/ 6642	*	2215/ 1175
gripper-12	*	17628/ 10147	*	3380/ 1776
gripper-20	*	*	*	11097/ 5675
rocket-ext-a	*	*	30110/ 17768	7638/ 4261
rocket-ext-b	*	*	85316/ 51540	28282/ 16324
logistics.a	*	*	411/ 191	847/ 436
logistics.b	*	*	920/ 436	542/ 271
logistics.c	*	*	4939/ 2468	7424/ 4796
logistics.d	*	*	*	16572/ 10512

Table 2: Ablation studies to evaluate the individual effectiveness of the new techniques: heuristic for ranking partial plans (HP) and consistency enforcement (CE). Each entry shows the number of partial plans generated and expanded. Note that REPOP is essentially UCPOP with HP and CE. (*) means no solution found after generating 100,000 nodes.

Before ending the discussion on plan quality, we should mention that it is possible to use post-processing techniques to improve the quality of plans produced by state-space and CSP-based planners. However, such post-processing, in addition to being NP-hard in general [1], does not provide a satisfactory solution for online integration of the planner with other modules such as schedulers and executors [6; 25].

Ablation Studies: We now evaluate the individual effectiveness of each of the acceleration techniques, viz., heuristic functions for ranking partial plans (HP), and consistency enforcement (CE).Table 2 shows the number of partial plans generated and expanded in the search when each of these techniques is added into the original UCPOP. We restrict our focus to the parallel domains where REPOP seems to offer significant advantages.

In the logistics and rocket domains, the use of h_{relax} heuristic accounts for the largest fraction of the improvement from UCPOP. Interestingly, h_{relax} fails to help scale up UCPOP even on very small problems in the gripper domain. We found that the search spends most of the time exploring inconsistent partial plans for failing to realize that a left or right gripper can carry at most one ball. This problem is alleviated by consistency enforcement (CE) techniques through detection and resolution of implicit conflicts (e.g. the conflict between $carry(ball1, left)$ and $carry(ball2, left)$). As a result, REPOP can comfortably solve large gripper problems, such as *gripper-20*.

Among the consistency enforcement techniques, both reachability analysis and disjunctive constraint representation appear to complement each other. For instance, in problem *logistics.d*, if only reachability analysis is used with the heuristic h_{relax}, a solution can be found after generating 255K nodes. When disjunctive representation is also used, the number of generated nodes is reduced by more than 15 times to 16K.

6 Related Work

Several previous research efforts have been aimed at accelerating partial order planners (c.f. [11; 12; 13; 16; 23; 24; 6; 4]). While none of these techniques approach the current level of performance offered by REPOP, many important ideas separately introduced in these previous efforts are either related to or are complementary to our techniques. IxTeT [6] uses distance based heuristic estimates to select among the possible resolutions of a given open condition flaw (although no evaluation of the technique is provided). It is interesting to note that IxTeT's use of distance based heuristics precedes their

independent re-discovery in the context of state-search planners by McDermott [20] and Bonet and Geffner [3]. In [4], Bylander describes the use of a relaxation heuristic based on linear planning for POP; it however seems not to be very effective. The idea of postponing the resolution of unsafe links by posting disjunctive constraints has been pursued by Smith and Peot in [23] as well as by Kambhampati and Yang in [16]. Our work shows that the effectiveness of this idea is enhanced significantly by generalizing the notion of conflicts to include indirect conflicts. The notion of action-proposition mutexes defined in Smith and Weld's work on temporal graphplan [26] is related to our notion of indirect conflicts introduced in Section 4. Finally, there is a significant amount of work on flaw selection strategies (e.g., the order in which open condition flaws are selected to be resolved) [11] that may be fruitfully combined with REPOP. The techniques for recognizing and suspending recursion ("looping") during search may also make a useful addition to REPOP [24].

7 Conclusion and Future Work

The successes in scaling up classical planning using CSP and state space search approaches have generally been (mis)interpreted as a side-swipe on the scalability of partial order planning. Consequently, in the last five years, work on POP paradigm has dwindled down, despite its known flexibility advantages. In this paper we challenged this trend by demonstrating that the very techniques that are responsible for the effectiveness of state search and CSP approaches can also be exploited to improve the efficiency of partial order planners dramatically. By applying the ideas of distance based heuristics, disjunctive representations for planning constraints and reachability analysis, we have achieved an impressive performance for a partial order planner, called REPOP, across a number of "parallel" planning domains. Our empirical studies show that not only does REPOP convincingly outperform Graphplan in parallel domains, the plans generated by REPOP have more execution flexibility. This is very interesting for two reasons. First of all, most of the real-world planning domains tend to have loose ordering among actions. Secondly, the ability for generating loosely ordered plans is very important in hybrid methods that involve on-line integration of planning with scheduling.

There are several avenues for extending this work. To begin with, our partial plan selection heuristics do not take negative interactions into account. This may be one reason for the unsatisfactory performance of REPOP in serial domains. One way to account for the negative interactions, that we are considering currently, involves using the partial state information provided by the pre- and post-cutsets of actions. Our work on AltAlt [22] suggests that the cost of achieving these partial states can be quantified in terms of the level in the planning graph at which the propositions comprising these states are present without any mutex relations. Another idea we are pursuing is to use n-ary state invariants (such as those detected in [5]) to detect and resolve more indirect conflicts in the plan. Finally, a more ambitious extension that we are pursuing involves considering more general versions of POP algorithms–including those that handle partially instantiated actions, as well as actions with conditional effects and durations.

References

[1] C. Backstrom. Computational aspects of reordering plans. *JAIR*. Vol. 9. pp. 99-137.

[2] A. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*. 90(1-2). 1997.

[3] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proc. ECP-99*, 1999.

[4] T. Bylander. A Linear programming heuristic for optimal planning. In *Proc. AAAI-97*, 1997.

[5] M. Fox and D. Long. Automatic inference of state invariants in TIM. *JAIR*. Vol. 9. 1998.

[6] M. Ghallab and H. Laruelle. Representation and control in IxTeT. In *Proc. AIPS-94*, 1994.

[7] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Proc. AAAI-98*, 1998.

[8] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In *Proc. AIPS-2000*, 2000.

[9] J. Hoffman and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. Submitted, 2000.

[10] A. Johnson, P. Morris, N. Muscettola and K. Rajan. Planning in Interplanetary Space: Theory and Practice. In *Proc. AIPS-2000*.

[11] D. Joslin and M. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proc. AAAI-94*.

[12] D. Joslin, M. Pollack. Passive and active decision postponement in plan generation. Proc. 3rd European Conf. on Planning. 1995.

[13] A. Gerevini and L. Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *JAIR*, 5:95-137, 1996.

[14] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proc. AAAI-96*.

[15] S. Kambhampati, C. Knoblock and Q. Yang. Planning as Refinement Search: A unified framework for evaluating design tradeoffs in partial-order planning. In *Artificial Intelligence*, 1995.

[16] S. Kambhampati and X. Yang. On the role of Disjunctive representations and Constraint Propagation in Refinement Planning In *Proc. KR-96*.

[17] S. Kambhampati. Planning Graph as (dynamic) CSP: Exploiting EBL, DDB and other CSP Techniques in Graphplan. *JAIR*. Vol. 12. pp. 1-34. 2000.

[18] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *JAIR*, 10(1-2) 1999.

[19] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. AAAI-91*.

[20] D. McDermott. Using regression graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–160, 1999.

[21] X. Nguyen and S. Kambhampati. Extracting effective and admissible state space heuristics from the planning graph. In *Proc. AAAI-2000*.

[22] X. Nguyen, S. Kambhampati and R. Nigenda. Planning Graph as the Basis for deriving Heuristics for Plan Synthesis by State Space and CSP Search. To appear in *Artificial Intelligence*.

[23] M. Peot and D. Smith. Threat-removal strategies for partial-order planning. In *Proc. AAAI-93*.

[24] D. Smith and M. Peot. Suspending Recursion Causal-link Planning. In *Proc. AIPS-96*.

[25] D. Smith, J. Frank and A. Jonsson. Bridging the gap between planning and scheduling. In *Knowledge Engineering Review*, 15(1):47-83. 2000.

[26] D. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *Proc. IJCAI-99*, 1999.

[27] D. Weld. An introduction to least commitment planning. *AI magazine*, 1994.

PLANNING

PLANNING WITH INCOMPLETE INFORMATION

Heuristic Search + Symbolic Model Checking = Efficient Conformant Planning

Piergiorgio Bertoli[1], **Alessandro Cimatti**[1], **Marco Roveri**[1,2]

[1] ITC-IRST, Via Sommarive 18, 38055 Povo, Trento, Italy
{bertoli,cimatti,roveri}@irst.itc.it
[2] DSI, University of Milano, Via Comelico 39, 20135 Milano, Italy

Abstract

Planning in nondeterministic domains has gained more and more importance. Conformant planning is the problem of finding a sequential plan that guarantees the achievement of a goal regardless of the initial uncertainty and of nondeterministic action effects. In this paper, we present a new and efficient approach to conformant planning. The search paradigm, called heuristic-symbolic search, relies on a tight integration of symbolic techniques, based on the use of Binary Decision Diagrams, and heuristic search, driven by selection functions taking into account the degree of uncertainty. An extensive experimental evaluation of our planner HSCP against the state of the art conformant planners shows that our approach is extremely effective. In terms of search time, HSCP gains up to three orders of magnitude over the breadth-first, symbolic approach of CMBP, and up to five orders of magnitude over the heuristic search of GPT, requiring, at the same time, a much lower amount of memory.

1 Introduction

Planning in nondeterministic domains is being recognized as increasingly important, and much harder than classical planning. In order to find plans that guarantee the achievement of the goal, it is necessary to deal with uncertainty in the initial condition, and with nondeterministic action effects. Several approaches to different planning problems in nondeterministic domains have been recently proposed. In [Pryor and Collins, 1996; Kabanza *et al.*, 1997; Weld *et al.*, 1998; Cimatti *et al.*, 1998; Rintanen, 1999] the problem of producing a conditional plan under the hypothesis of (total or partial) run-time observability has been considered. Conformant planning [Goldman and Boddy, 1996] is the problem of finding a sequential plan that will achieve the goal assuming that no information will be available at run-time. Conformant planning has also a close relation with the problem of finding synchronization sequences in hardware circuits [Kohavi, 1978].

Conformant planning has been recently tackled in different ways. The first efficient approach, based on GRAPHPLAN, was presented in [Smith and Weld, 1998]. Bonet and Geffner [2000] formulate conformant planning as a problem of search in the space of belief states. The approach, implemented within the GPT system, relies on the use of heuristics to drive an A^*-style search algorithm. The state of the art in conformant planning is [Cimatti and Roveri, 2000]. This approach extends symbolic model checking techniques, to compactly represent and efficiently explore the search space. Although the approach inherits from symbolic model checking a blind, breadth-first search style, the CMBP planner shows surprising efficiency, the reported results being superior to the heuristic search of GPT, sometimes by more than two orders of magnitude.

In this paper, we present a new, *heuristic-symbolic* search paradigm, based on the combination of heuristic search with ideas taken from symbolic model checking. We modify and extend the data structures defined in [Cimatti and Roveri, 2000], implementing the new planner HSCP (Heuristic-Symbolic Conformant Planner). Heuristic-symbolic search overcomes the bottleneck of the breadth-first approach while retaining the advantages of symbolic techniques: by means of a simple, domain-independent heuristic, HSCP outperforms CMBP, gaining up to three orders of magnitude in search time, with a much lower memory consumption.

This paper is structured as follows. We first define the problem of conformant planning. Then, we present the combined use of symbolic and heuristic mechanisms for the representation and exploration of the search space. We describe the planning algorithm, and present an experimental analysis comparing HSCP with CMBP and GPT. Finally, we discuss some other related work and draw the conclusions.

2 Conformant Planning

We consider nondeterministic planning domains, where actions can have preconditions, conditional and uncertain effects, and the initial state can be only partly specified.

Definition 1 (Planning Domain) *A Planning Domain is a 4-tuple $\mathcal{D} = (\mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{P}$ is the (finite) set of atomic propositions, $\mathcal{S} \subseteq Pow(\mathcal{P})$ is the set of states, $\mathcal{A}$ is the (finite) set of actions, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation.*

Intuitively, a proposition is in a state if and only if it holds in that state. In the following we assume that a planning domain $\mathcal{D}$ is given. We use s, s' and s'' to denote states of $\mathcal{D}$, and α to denote actions. $\mathcal{R}(s, \alpha, s')$ holds iff when executing the action α in the state s the state s' is a possible outcome. We say that an action α is applicable in s iff there is at least one state s' such that $\mathcal{R}(s, \alpha, s')$ holds. We say that an action

α is deterministic in s iff there is a unique state s' such that $\mathcal{R}(s, \alpha, s')$ holds. An action α has an uncertain outcome in s if there are at least two distinct states s' and s'' such that $\mathcal{R}(s, \alpha, s')$ and $\mathcal{R}(s, \alpha, s'')$ hold.

Plans are sequences of actions, i.e. elements of $\mathcal{A}^*$. We use ϵ for the 0-length plan, π and ρ to denote plans, and $\pi; \rho$ for concatenation. Conformant planning is the problem of finding a plan that, if executed in any initial state in $\mathcal{I} \subseteq \mathcal{S}$, takes the domain into a set of states $\mathcal{G} \subseteq \mathcal{S}$, regardless of nondeterministic action effects. Following [Bonet and Geffner, 2000], we model this problem as search in the space of *belief states*. A belief state is a set of states, intuitively expressing a condition of uncertainty, by collecting together all the states that are equally possible. Conformant planning amounts to searching Pow($\mathcal{S}$), i.e. the powerset of the set of states of the domain. In order to tackle this problem, we need the ability to define the applicability and effect of actions in a belief state, under a condition of uncertainty.

Definition 2 *An action α is applicable in a belief state $\emptyset \neq Bs \subseteq \mathcal{S}$ iff α is applicable in every state in Bs. If α is applicable in Bs, its execution Exec$[\alpha](Bs)$ is the set $\{s' | \mathcal{R}(s, \alpha, s'),$ and $s \in Bs\}$. The execution of a plan π in a belief state, written Exec$[\pi](Bs)$, is defined as follows.*

$$
\begin{array}{lcl}
\textit{Exec}[\epsilon](Bs) & \doteq & Bs \\
\textit{Exec}[\pi](\emptyset) & \doteq & \emptyset \\
\textit{Exec}[\alpha; \pi](Bs) & \doteq & \emptyset, \textit{ if } \alpha \textit{ is not applicable in } Bs \\
\textit{Exec}[\alpha; \pi](Bs) & \doteq & \textit{Exec}[\pi](\textit{Exec}[\alpha](Bs)), \textit{ otherwise}
\end{array}
$$

We say that a plan is applicable in a belief state when its execution is not empty. For a conformant planning problem, solutions are applicable plans, for which all the final states must be goal states.

Definition 3 *Let $\emptyset \neq \mathcal{I} \subseteq \mathcal{S}$ and $\emptyset \neq \mathcal{G} \subseteq \mathcal{S}$. The plan π is a conformant solution to the problem $\langle \mathcal{I} \,.\, \mathcal{G} \rangle$ iff $\emptyset \neq$ Exec$[\pi](\mathcal{I}) \subseteq \mathcal{G}$.*

3 Heuristic-Symbolic Representation

3.1 Representation of Planning Domains

We represent planning domains symbolically, by means of BDDs (Binary Decision Diagrams) [Bryant, 1992], borrowing from the formal verification community the standard machinery used in symbolic model checking [McMillan, 1993]. BDDs are an efficient mechanism for the representation of propositional formulae. A BDD is a binary directed acyclic graph. The terminal nodes are either $True$ or $False$. Each non-terminal node is associated with a boolean variable, and two BDDs, the left and right branches, corresponding to the assignment of the true and false values to the node variable. BDDs enjoy a canonical form, resulting from the imposition of a total order over the variables associated to nonterminal nodes. This allows for equivalence checking in constant time, and for the efficient implementation of boolean transformation (e.g. conjunction, disjunction, negation) and QBF transformations (i.e. universal and existential quantification of boolean variables). Efficient software libraries for the manipulation of BDDs, called BDD packages, are available. For lack of space the reader is referred to [Bryant, 1992] for a thorough description of BDDs.

The symbolic representation of the automaton for a given domain can be efficiently built starting from the domain description language (see for instance [Cimatti *et al.*, 1997]). For a given planning domain, we use two vectors of BDD variables, called current and next state variables, written $\boldsymbol{x}$ and $\boldsymbol{x}'$ respectively, to represent atomic propositions of the domain. A state s can be seen as an assignment to such variables. A set of states $Bs \subseteq \mathcal{S}$ is associated with a unique corresponding BDD, the models of which are the assignments corresponding to the states in Bs. We represent actions by means of a vector of action variables $\boldsymbol{\alpha}$. We write $\boldsymbol{\alpha} = \alpha$ for the BDD in the $\boldsymbol{\alpha}$ variables representing the action α. We assume a mapping between the set-theoretic view of the planning domain and the corresponding BDD-based representation. When clear from the context, we confuse the set-theoretic and symbolic representations. For instance, we use equivalently the $False$ BDD and $\emptyset$. We write $\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}')$ for the BDD representing the transition relation, to stress the dependency on BDD variables. The notions defined in previous section at the set-theoretic level have a direct counterpart in terms of BDD transformations. For instance, the applicability relation $\textsc{Appl}(\boldsymbol{x}, \boldsymbol{\alpha})$ is computed symbolically by the projection operation $\exists \boldsymbol{x}'.\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}')$, the result being a BDD in the current state variables and action variables, whose assignments are the state-action pairs $\langle s \,.\, \alpha \rangle$ such that α is applicable in s. In order to check whether an action α is applicable in a belief state $\emptyset \neq Bs \subseteq \mathcal{S}$, it is possible to check whether $\exists \boldsymbol{\alpha}.\forall \boldsymbol{x}.((Bs(\boldsymbol{x}) \wedge \boldsymbol{\alpha} = \alpha) \rightarrow \textsc{Appl}(\boldsymbol{x}, \boldsymbol{\alpha}))$ is not the $False$ BDD. The execution of α in $Bs(\boldsymbol{x})$ is computed as $\exists \boldsymbol{x}.(Bs(\boldsymbol{x}) \wedge \exists \boldsymbol{\alpha}.(\boldsymbol{\alpha} = \alpha \wedge \mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}')))[\boldsymbol{x}'/\boldsymbol{x}]$, where $[\boldsymbol{x}'/\boldsymbol{x}]$ represents the parallel substitution of the next state variables with the current state variables.

3.2 Representation of the Search Space

The search space for conformant planning is Pow($\mathcal{S}$) (outlined in Figure 1, with belief states represented as circles). The search space can be constructed forward, starting from the initial set of states, or backward, from the goal. In the first case, the arrows outgoing from belief state 1 represent the fact that the labeling action A [B, C, respectively] is applicable to all the states in 1, and can result in any of the states in belief state 2 [3, 4, resp.]. In the backward case, the situation is dual: for instance, belief state -3 is the maximal belief state where the labeling action B is applicable, and the result is contained in the belief state -1. For both search directions, a belief state Bs is directly represented by the corresponding BDD $Bs(\boldsymbol{x})$. Given the canonical form of BDDs, a belief state is simply the pointer to the unique corresponding BDD. In Figure 2, the lower box depicts a possible status of the BDD package. The column on the left shows the variables in the BDD package. The horizontal dashed line separates state variables (below the line) from action variable. A belief state is a subgraph in the state variables. Below the dashed line, a possible configuration for the forward search space of Figure 1 is given. For each belief state, there is a corresponding BDD in the state variables. Solid [dashed, respectively] arcs in a BDD represent the positive [negative, resp.] assignment to the variable in the originating node. For instance, the leftmost BDD Bs1 encodes the formula $(x1 \leftrightarrow \neg x2) \wedge x3 \wedge x4$.

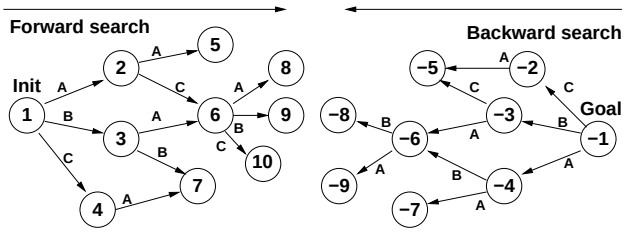


Figure 1: The search space

For the sake of simplicity, only the paths leading to $True$ are shown. The advantage over an enumerative representation of belief states (e.g. the list of the state vectors associated to each state contained in the Bs) is twofold. In terms of memory, there is no direct connection between the cardinality of the belief state and the size of the corresponding BDD. (Consider for instance that, with 10 boolean variables, the set of all the states, of cardinality 1024, is represented by the $True$ BDD, requiring one node.) Furthermore, BDD packages are designed to maximize the sharing between different BDDs, and minimize memory occupation. In terms of efficiency, the set-theoretic operations for the transformation and combination of belief states (e.g. projection, equivalence, inclusion) can be efficiently performed as BDD transformations, with the primitives provided by the BDD package, which make aggressive use of memoizing of previously computed subproblems.

One of the novelties of our approach is based on the integration of the symbolic representation described above with the standard data structures used in heuristic search. In order to allow for heuristic search, a hash table is used to store and retrieve the belief states visited during the search (upper box in Figure 2). Each entry directly points to the BDD representing the belief state, and is annotated with the suitable information (e.g. the plan, the cost factors). For instance, for belief state 6 the stored plan is the sequence B;A, i.e. one of the sequences of actions needed to reach belief state 6 from the initial belief state 1. The hash table is accessed using as key the pointer to the BDDs representing the belief state. The approach relies on the canonical form of BDDs, thanks to which only one BDD representative for equivalent boolean functions is required. The hash table is accessed with a Bs, and if a corresponding entry is present, then it is returned, otherwise a new entry is created. Notice that the memory occupation of the belief state for each entry in the hash table is fixed to a single pointer: the BDD package is responsible to compress the information, as shown in Figure 2. Notice also that this approach allows to reuse a standard BDD package, without interfering with its internal mechanisms (e.g. hashing, memoizing, garbage collection).

The construction of the search space is performed by expanding an individual belief state, resulting in a set of the form $\{\langle Bs_1 \,.\, \alpha_1\rangle, \ldots, \langle Bs_n \,.\, \alpha_n\rangle\}$. For instance, when expanding in forward the initial belief state 1 in Figure 1, the resulting expansion is the set $\{\langle 2 \,.\, A\rangle, \langle 3 \,.\, B\rangle, \langle 4 \,.\, C\rangle\}$, meaning that A, B and C are applicable in 1, and the corresponding execution results are the belief state 2, 3 and 4. Dually, when expanding -6 backward, the result is $\{\langle -8 \,.\, B\rangle, \langle -9 \,.\, A\rangle\}$, meaning that B can be applied in -8 and results in -6. The ability to expand belief states symbolically is one of the keys to the efficient search. The backward step of expansion BWDEXPANDBS, applied to the BDD $Bs(\boldsymbol{x})$, constructs the BDD $\forall \boldsymbol{x}'.(\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}') \rightarrow Bs(\boldsymbol{x})[\boldsymbol{x}/\boldsymbol{x}'])$ $\wedge$ APPL$(\boldsymbol{x}, \boldsymbol{\alpha})$. The result is a BDD in the current state variables $\boldsymbol{x}$ and action variables $\boldsymbol{\alpha}$, representing an association from actions to states $\{\langle s_1^1 \,.\, \alpha_1\rangle, \ldots, \langle s_{n_1}^1 \,.\, \alpha_1\rangle, \ldots, \langle s_1^k \,.\, \alpha_k\rangle, \ldots, \langle s_{n_k}^k \,.\, \alpha_k\rangle\}$, that can be interpreted as an association between actions and belief states, i.e. $\{\langle\{s_1^1, \ldots, s_{n_1}^1\} \,.\, \alpha_1\rangle, \ldots, \langle\{s_1^k, \ldots, s_{n_k}^k\} \,.\, \alpha_k\rangle\}$. The dual forward step FWDEXPANDBS expands $Bs(\boldsymbol{x})$ by computing the executions in Bs of every applicable action, i.e. $(\exists \boldsymbol{x}.(Bs(\boldsymbol{x}) \wedge (\forall \boldsymbol{x}.(Bs(\boldsymbol{x}) \rightarrow$ APPL$(\boldsymbol{x}, \boldsymbol{\alpha})) \wedge \mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}'))))[\boldsymbol{x}'/\boldsymbol{x}]$. The data structure resulting from the expansion is again a BDD in the $\boldsymbol{x}$ and $\boldsymbol{\alpha}$ variables. In Figure 2 the BDD resulting from the forward expansion of belief state 6 is shown: the paths at the levels of action variables represent the different actions A, B and C, and the attached subgraphs represent the corresponding belief states 8, 9 and 10. The link between the symbolic representation of the expansion and the hash-based representation of the search space is the special purpose primitive PRUNEBSEXPANSION. For either search directions, every time a belief state is expanded, the resulting belief states have to be compared with the previously visited belief states. If not present, they must be inserted in the hash of the visited belief states, otherwise eliminated. This analysis is performed by PRUNEBSEXPANSION, that operates directly on the BDD resulting from the expansion of the Bs, and allows to combine the symbolic expansion of belief states with the hashing mechanism. PRUNEBSEXPANSION recursively descends the BDD in the action and state variables, and interprets as a belief state every BDD node having a state variable at its top. Nodes corresponding to previously visited belief states are pruned, while the new ones are inserted in the hash. PRUNEBSEXPANSION assumes that in the BDD package action variables precede state variables, as shown in Figure 2. This does not appear to be a significant limitation in many practical cases. This approach takes care of a very significant source of redundancy: often a belief state can be associated with a large number of equivalent plans (in Figure 1, $A;C$ and $B;A$ are two equivalent action plans associated with 1). The pruning mechanism makes sure that, for each Bs, only one plan is left.

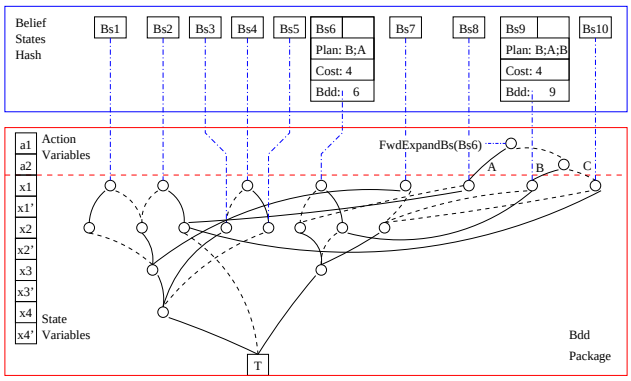


Figure 2: The combined use of the hash and the BDD

4 The Planning Algorithm

In this section we describe the planning algorithm, based on the data structures and primitives described in previous section. Figure 3 depicts the backward search algorithm, that takes in input the problem description in form of the BDDs $\mathcal{I}$ and $\mathcal{G}$, while the transition relation $\mathcal{R}$ is assumed to be globally available. *OpenBsPool* contains the (annotated) belief states which have been reached but still have to be explored, and is initialized to the first belief state of the search, i.e. $\mathcal{G}$, annotated with the empty plan ϵ. BSMARKVISITED inserts $\mathcal{G}$ into the hash table of visited belief states, also updating the cost information. The algorithm loops (lines 3-11) until a solution is found or all the search space has been exhausted. First, an annotated belief state $\langle Bs\ .\ \pi\rangle$ is extracted from the open pool (line 4) by EXTRACT. The belief state is expanded by BWDEXPANDBS, computing the corresponding BDD in the $\boldsymbol{x}$ and $\boldsymbol{\alpha}$ variables. The resulting Bs expansion, stored in *BsExp*, is traversed by PRUNEBSEXPANSION as explained in previous section: for each belief state in the expansion, the hash table of the visited belief states is accessed, and only the belief states that are not present are accumulated in the returned list.

Each of the resulting belief states is compared with the initial set of states $\mathcal{I}$. If $\mathcal{I} \subseteq Bs_i$, then the associated plan $\alpha_i; \pi$ is a solution to the problem, the loop is exited and the plan is returned. Otherwise, the annotated belief state $\langle Bs_i\ .\ \alpha_i; \pi\rangle$ is inserted in *OpenBsPool* and the loop is resumed. If *OpenBsPool* becomes empty and a solution has not been found, then a fix point has been reached, i.e. all the reachable space of belief states has been covered, and the algorithm terminates with failure. The dual algorithm for forward search, not reported here for lack of space, shares the structure of the backward planning algorithm described in Figure 3, but applies the forward expansion steps defined in previous section. The termination of the algorithms is guaranteed by the fact that the set of explored belief sets (stored in the visited hash table) is monotonically increasing. Since the set of accumulated belief states is contained in Pow($\mathcal{S}$), which is finite, a fix point is eventually reached. Furthermore, a solution plan is returned if and only if the problem admits a solution, otherwise a failure is returned.

The length of the constructed plans depends on the effectiveness of the heuristic with respect to the problem being tackled, in particular on the primitives EXTRACT and INSERT. We use a very simple heuristic function. When proceeding backward, it selects the belief state with the highest cardinality. We consider as more promising a belief state with a low degree of knowledge: intuitively, the associated plan requires a low knowledge to lead to the goal. Dually, in forward, we select the belief states with lowest cardinality, that is associated with a plan that yields a guarantee of high knowledge. Our approach departs significantly from the heuristic function used in [Bonet and Geffner, 2000], that are a simple combination of heuristic functions over the states in the belief state. The behavior of GPT on the SQUARE and CUBE variations shows that this approach can lead to dramatic failures, as discussed in [Cimatti and Roveri, 2000]. Although our selection strategy is in general not admissible (in the sense of A*), it seems to be extremely effective, especially in the backward case.

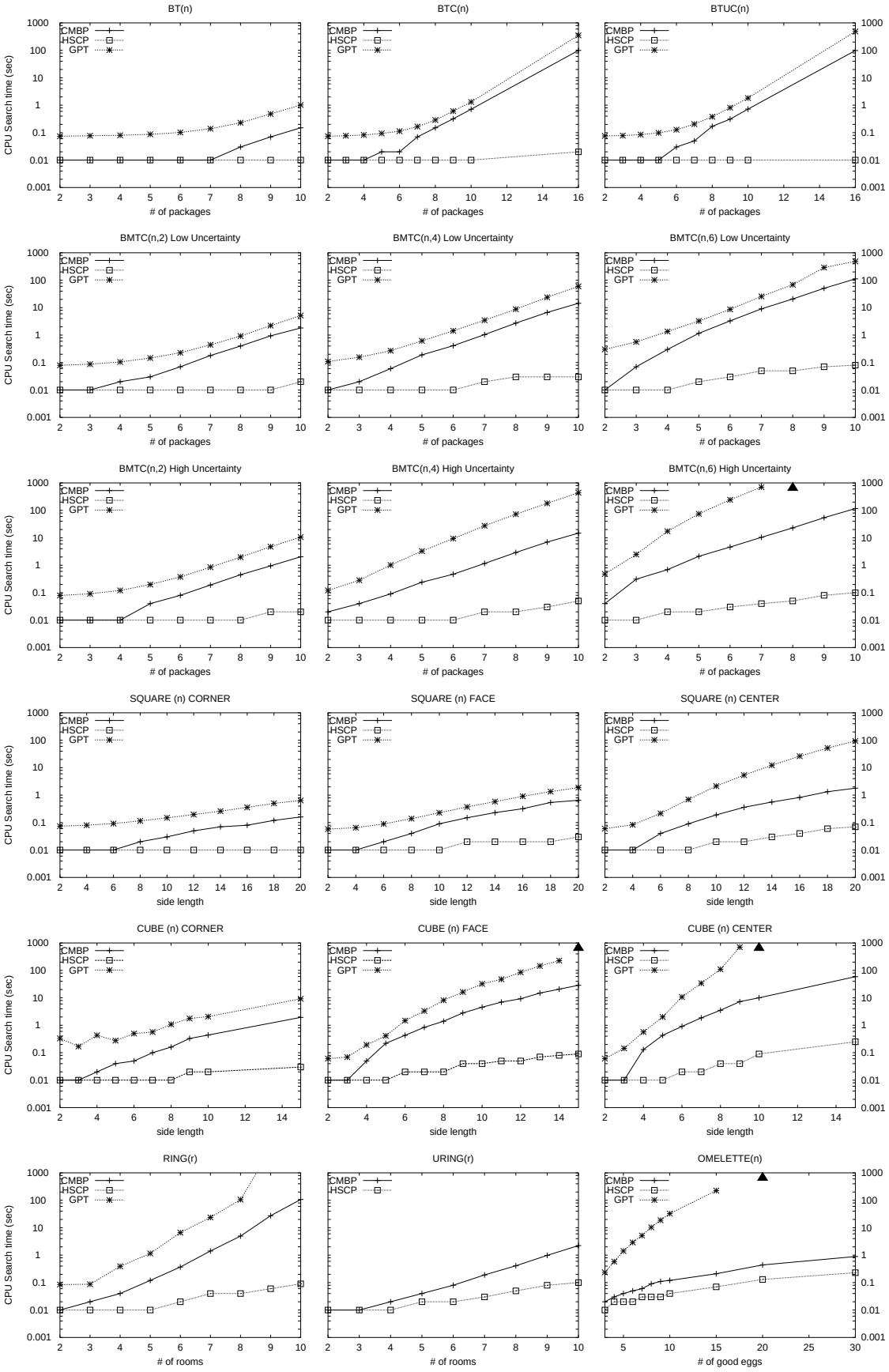
```
    procedure HeurSymBwdConformant(I,G)
0   begin
1     OpenBsPool := {⟨G . ε⟩}; BsMarkVisited(G);
2     Solved := I ⊆ G; Solution := ε;
3     while (OpenBsPool ≠ ∅ ∧ ¬Solved) do
4       ⟨Bs . π⟩ := Extract(OpenBsPool);
5       BsExp := BwdExpandBs(Bs);
6       BsPList := PruneBsExpansion(BsExp);
7       for ⟨Bs_i . α_i⟩ in BsPList do
8         if I ⊆ Bs_i then
9           Solved := True; Solution := α_i; π ; break;
10        else Insert(⟨Bs_i . α_i; π⟩,OpenBsPool) endif;
11    end while
12    if Solved then return Solution;
13    else return Fail;
14  end
```

Figure 3: The backward conformant planning algorithm

5 Experimental Evaluation

We implemented the data structures and the algorithms described in this paper, in the following called HSCP (Heuristic-Symbolic Conformant Planner), on top of CMBP [Cimatti and Roveri, 2000] (see `http://sra.itc.it/tools/mbp/`). We carried out an extensive experimental evaluation of our approach, covering all the test cases presented so far in the literature to evaluate conformant planners. (For lack of space, we refer to [Cimatti and Roveri, 2000] for the description of the problems.) The most relevant systems for the comparison are GPT, for its heuristic search style, and CMBP, for its use of model checking techniques. We do not consider other conformant planners, such as CGP [Smith and Weld, 1998] and QBFPLAN [Rintanen, 1999], that are outperformed by GPT and CMBP, as shown in [Cimatti and Roveri, 2000]. For CMBP and HSCP, the backward search algorithms were run. All the experiments were run on an Pentium II 300MHz with 512Mb of memory running Linux, fixing a memory limit of 128Mb and a CPU time limit of 2 hours. The results of the comparison are depicted in the following. Each plot refers to a problem class. For each problem instance, we report (in seconds on a logarithmic scale) the search time, i.e. the CPU time required to find a solution for the given planning problem, or to discover that the problem admits no solution. (We do not consider the preprocessing time. In the case of GPT, this includes the generation of a C++ file, that is compiled and then linked to the solver program. In the case of CMBP and HSCP, for all the reported experiments, the automaton construction, optimized with respect to the results reported in [Cimatti and Roveri, 2000], requires at most 0.44 secs, in the OMELETTE(30) problem.)

The experimental evaluation shows that the heuristic-symbolic search of HSCP outperforms CMBP (that, in turn, outperforms GPT [Cimatti and Roveri, 2000]). On average, the improvement is of about two orders of magnitude, with a minimum of one order of magnitude in the case of SQUARE and OMELETTE, and a maximum of about three orders of magnitude in the case of BTUC and of BMTC(n,6) with high uncertainty. In the RING(9) problem, GPT reaches the fixed


BT(n)
BTC(n)
BTUC(n)
BMTC(n,2) Low Uncertainty
BMTC(n,4) Low Uncertainty
BMTC(n,6) Low Uncertainty
BMTC(n,2) High Uncertainty
BMTC(n,4) High Uncertainty
BMTC(n,6) High Uncertainty
SQUARE (n) CORNER
SQUARE (n) FACE
SQUARE (n) CENTER
CUBE (n) CORNER
CUBE (n) FACE
CUBE (n) CENTER
RING(r)
URING(r)
OMELETTE(n)
CMBP
HSCP
GPT
CPU Search time (sec)
# of packages
side length
# of rooms
# of good eggs

time limit without finding a solution. The URING problems are not representable in GPT. It is interesting to notice that, while GPT reaches the memory limit in 4 sets of experiments (solid triangles in the plots), CMBP and HSCP complete all the runs within the given limit. This appears to be due to the use of the BDD based representation. In terms of memory, on average, CMBP requires about twice more memory than HSCP: the main reason for this seems to be that CMBP represents plans symbolically, and this requires the introduction of a new set of action variables for each layer of the search. Although CMBP is guaranteed to return plans of minimal length, HSCP returns plans that are surprisingly short. For all the experiments, HSCP returns minimal length plans, with the exception of some of the BMTC problems. Finally, the OMELETTE problem, differently from all the others, admits no conformant solution, and therefore it requires the complete exploration of the state space. We expected CMBP to outperform HSCP, because of its ability to expand a large number of states in one single operation. However, HSCP outperforms CMBP, having to deal with more BDDs, but of smaller size.

6 Related Work and Conclusions

In this paper, we have presented a new, heuristic-symbolic search paradigm, resulting in an extremely effective approach to conformant planning. BDD-based, symbolic techniques are combined with search algorithms that exploit heuristic selection functions taking into account the degree of uncertainty. Our approach gains up to three order of magnitude in search time over the breadth-first symbolic approach of CMBP, and up to five over the heuristic search of GPT, with lower memory requirements.

In the following we describe additional related works. In hardware design, synchronization sequences are used for testing and equivalence checking. The problem of finding a synchronization sequence can be seen as a particular form of conformant planning: a synchronization sequence is a sequence of inputs that takes a circuit from an unknown state into a completely defined one. In [Cimatti *et al.*, 2001], we tackle this problem within the heuristic-symbolic paradigm. A comparison over the standard benchmarks shows that our approach is more than competitive with specialized BDD-based algorithms [Pixley *et al.*, 1992; Rho *et al.*, 1993]. The work in [Finzi *et al.*, 2000] is limited to deterministic actions, and relies on user-defined heuristics to achieve an efficiency comparable with CMBP. [Ferraris and Giunchiglia, 2000] tackles conformant planning in nondeterministic domains described in $\mathcal{C}$ language, that allows for parallel actions. Although the problems tackled by CMBP and HSCP are expressed in the $\mathcal{AR}$ language, that is limited to sequential actions, the planners are not subject to this restriction. For instance, in [Cimatti *et al.*, 2001] HSCP is used to analyze circuits with for parallel inputs (actions) expressed in the SMV language [McMillan, 1993]. The planner of [Ferraris and Giunchiglia, 2000] can not discover when a problem admits no conformant plan, and the reported results are comparable with CGP [Smith and Weld, 1998] (see also [Cimatti and Roveri, 2000] for a detailed discussion). In [Bertoli *et al.*, 2001], the approach presented in this paper is used as a basis for planning under partial observability. As a future work, we will investigate the use of more accurate heuristic functions, that combine a measure of the degree of uncertainty with domain knowledge in the style of GPT. Furthermore, we will extend the approach presented in this paper with the techniques of [Cimatti and Roveri, 2000], that allow to expand symbolically several belief states at a time. We will also generalize our work to deal with extended goals expressed in different forms of temporal logic specifications.

References

[Bertoli *et al.*, 2001] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. IJCAI-01*. AAAI Press, 2001.

[Bonet and Geffner, 2000] B. Bonet and H. Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. AIPS-00*, pages 52–61. AAAI Press, April 2000.

[Bryant, 1992] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Cimatti and Roveri, 2000] A. Cimatti and M. Roveri. Conformant Planning via Symbolic Model Checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.

[Cimatti *et al.*, 1997] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In *Proc. ECP-97*, LNCS 1348, pages 130–142, Toulouse, France, September 1997. Springer-Verlag.

[Cimatti *et al.*, 1998] A. Cimatti, M. Roveri, and P. Traverso. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proc. AIPS-98*, Pittsburgh, USA, June 1998. AAAI Press.

[Cimatti *et al.*, 2001] A. Cimatti, M. Roveri, and P. Bertoli. Searching Powerset Automata by Combining Explicit-State and Symbolic Model Checking. In *Proc. TACAS 2001*, pages 313–327. Springer, April 2001.

[Ferraris and Giunchiglia, 2000] P. Ferraris and E. Giunchiglia. Planning as satisfiability in nondeterministic domains. In *Proc. AAAI'00)*, Austin, Texas, July-August 2000. AAAI Press.

[Finzi *et al.*, 2000] A. Finzi, F. Pirri, and R. Reiter. Open world planning in the situation calculus. In *Proc. AAAI'00)*, Austin, Texas, July-August 2000. AAAI Press.

[Goldman and Boddy, 1996] R.P. Goldman and M.S. Boddy. Expressive Planning and Explicit Knowledge. In *Proc. AIPS-96*, pages 110–117. AAAI Press, 1996.

[Kabanza *et al.*, 1997] F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[Kohavi, 1978] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill Book Company, New York, 1978. ISBN 0-07-035310-7.

[McMillan, 1993] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[Pixley *et al.*, 1992] C. Pixley, S.-W. Jeong, and G. D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proc. of the 29th Design Automation Conference*, IEEE CS Press, 1992.

[Pryor and Collins, 1996] L. Pryor and G. Collins. Planning for Contingency: a Decision Based Approach. *J. of Artificial Intelligence Research*, 4:81–120, 1996.

[Rho *et al.*, 1993] J.-K. Rho, F. Somenzi, and C. Pixley. Minimum length synchronizing sequences of finite state machine. In *Proc. of the 30th ACM/IEEE Design Automation Conference*, pages 463–468, Dallas, TX, June 1993. ACM Press.

[Rintanen, 1999] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intellegence Research*, 10:323–352, 1999.

[Smith and Weld, 1998] D. E. Smith and D. S. Weld. Conformant graphplan. In *Proc. AAAI-98*, pages 889–896, Menlo Park, July 26–30 1998. AAAI Press.

[Weld *et al.*, 1998] D. S. Weld, C. R. Anderson, and D. E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proc. AAAI-98*, pages 897–904, Menlo Park, July 26–30 1998. AAAI Press.

Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking

Piergiorgio Bertoli[1], **Alessandro Cimatti**[1], **Marco Roveri**[1,2], **Paolo Traverso**[1]

[1] ITC-IRST, Via Sommarive 18, 38055 Povo, Trento, Italy
{bertoli,cimatti,roveri,traverso}@irst.itc.it
[2] DSI, University of Milano, Via Comelico 39, 20135 Milano, Italy

Abstract

Planning under partial observability is one of the most significant and challenging planning problems. It has been shown to be hard, both theoretically and experimentally. In this paper, we present a novel approach to the problem of planning under partial observability in non-deterministic domains. We propose an algorithm that searches through a (possibly cyclic) and-or graph induced by the domain. The algorithm generates conditional plans that are guaranteed to achieve the goal despite of the uncertainty in the initial condition, the uncertain effects of actions, and the partial observability of the domain. We implement the algorithm by means of BDD-based, symbolic model checking techniques, in order to tackle in practice the exponential blow up of the search space. We show experimentally that our approach is practical by evaluating the planner with a set of problems taken from the literature and comparing it with other state of the art planners for partially observable domains.

1 Introduction

Research in planning is more and more focusing on the problem of planning in nondeterministic domains and with incomplete information, see for instance [Pryor and Collins, 1996; Kabanza *et al.*, 1997; Weld *et al.*, 1998; Cimatti *et al.*, 1998; Rintanen, 1999a; Bonet and Geffner, 2000]. The search mechanism and the structure of the generated plans depend on how information is assumed to be available at run-time. The approaches in [Kabanza *et al.*, 1997; Cimatti *et al.*, 1998], for instance, construct conditional plans under the assumption of full observability, i.e. the state of the world can be completely observed at run-time. At the other end of the spectrum, conformant planning [Smith and Weld, 1998; Bonet and Geffner, 2000; Cimatti and Roveri, 2000; Bertoli *et al.*, 2001] constructs sequential plans that are guaranteed to solve the problem assuming that no information at all is available at run-time. In this paper we tackle the problem in the middle of the spectrum, i.e. planning under *partial observability*, the general case where only part of the domain information is available at run time. Several approaches have been proposed in the past [Pryor and Collins, 1996; Weld *et al.*, 1998; Bonet and Geffner, 2000]. This problem is however significantly more difficult than the two limit cases of fully observable and conformant planning [Littman *et al.*, 1998]. Compared to planning under full observability, planning under partial observability must deal with uncertainty about the state in which the actions will be executed. This makes the search space no longer the set of states of the domain, but its powerset, i.e. the space of "belief states" [Bonet and Geffner, 2000]. Compared to conformant planning, the structure of the plan is no longer sequential, but tree-shaped, in order to represent a conditional course of actions.

In this paper, we propose a general and efficient approach to conditional planning under partial observability. We make the following contributions. First, we present a formal model of partially observable planning domains, that can represent both observations resulting from the execution of sensing actions [Pryor and Collins, 1996; Weld *et al.*, 1998] and automatic sensing that depends on the current state of the world [Tovey and Koenig, 2000]. Second, we propose a novel planning algorithm that searches through a (possibly cyclic) and-or graph induced by the domain. The algorithm generates conditional, acyclic plans that are guaranteed to achieve the goal despite of the uncertainty in the initial condition, and of the uncertain effects of actions. Third, we implement our approach by means of BDD-based, symbolic model checking techniques, extending the Planning via Model Checking paradigm and the related system MBP [Cimatti *et al.*, 1998].

We experimentally evaluate our approach by analyzing some problems from the distributions of other available planners for partially observable domains and other problems, including the "maze" domains proposed by Koenig [Tovey and Koenig, 2000]. MBP outperforms two state of the art planners for partially observable domains, SGP [Weld *et al.*, 1998] and GPT [Bonet and Geffner, 2000].

The paper is organized as follows. We provide a formal definition of partially observable planning domains and of conditional planning. We then present the planning algorithm, and its implementation in the MBP planner. Then, we report on the experimental evaluation, discuss further related work, and draw some conclusions.

2 Partially Observable Domains

We consider nondeterministic domains under the hypothesis of partial observability, i.e. where a limited amount of information can be acquired at run time.

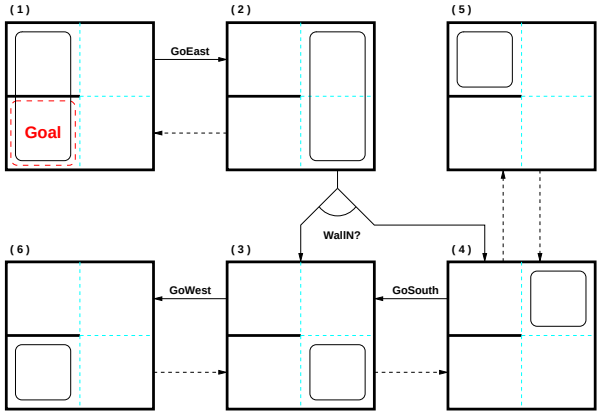


Figure 1: A simple robot navigation domain

Definition 1 *A partially observable planning domain is a tuple* $\langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{O}, \mathcal{X} \rangle$*, where*

- $\mathcal{P}$ *is a finite set of propositions;*
- $\mathcal{S} \subseteq Pow(\mathcal{P})$ *is the set of states;*
- $\mathcal{A}$ *is a finite set of actions.*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ *is the transition relation.*
- $\mathcal{O}$ *is a finite set of observation variables;*
- $\mathcal{X} : \mathcal{O} \rightarrow Pow(\mathcal{S} \times \mathcal{A} \times \{\top, \bot\})$ *is the observation relation.*

Intuitively, a state is a collection of the propositions holding in it. The transition relation describes the effects of action execution. An action a is applicable in a state s iff there exists at least one state s' such that $\mathcal{R}(s, a, s')$. The set $\mathcal{O}$ contains observation variables, whose value can be observed at run-time, during execution. Without loss of generality, we assume that observation variables are boolean. We use o to denote observation variables. We call $\mathcal{X}(o)$, written $\mathcal{X}_o$ in the following, the observation relation of o. Given an action (that has been executed) and the resulting state, $\mathcal{X}_o$ specifies what are the values that can be assumed at run-time by the observation variable o. In a state $s \in \mathcal{S}$ after an action $a \in \mathcal{A}$, an observation variable $o \in \mathcal{O}$ may convey no information: this is specified by stating that both $\mathcal{X}_o(s, a, \top)$ and $\mathcal{X}_o(s, a, \bot)$ hold, i.e. both the true and false values are possible. In this case, we say that o is *undefined* in s after a. If $\mathcal{X}_o(s, a, \top)$ holds and $\mathcal{X}_o(s, a, \bot)$ does not hold, then the value of o in state s after a is true. The dual holds for the false value. In both cases, we say that o is *defined in s after a*. An observation variable is always associated with a value, i.e. for each $s \in \mathcal{S}$ and for each $a \in \mathcal{A}$, at least one of $\mathcal{X}_o(s, a, \top)$ and $\mathcal{X}_o(s, a, \bot)$ holds.

Consider the example of a simple robot navigation domain in Figure 1, containing a 2x2 room with an extra wall. The propositions of the domain are `NW`, `NE`, `SW`, and `SE`, corresponding to the four positions in the room. Exactly one of them holds in each of the four states in $\mathcal{S}$. The robot can move in the four directions (deterministic actions `GoNorth`, `GoSouth`, `GoWest` and `GoEast`), provided that there is not a wall in the direction of motion. The action is not applicable, otherwise. At each time tick, the information of walls proximity in each direction is available to the robot (observation variables `WallN`, `WallS`, `WallW` and `WallE`). For instance, we have that for any action a of the domain $\mathcal{X}_{\texttt{WallE}}(\texttt{NW}, a, \bot)$ and $\mathcal{X}_{\texttt{WallW}}(\texttt{NW}, a, \top)$. In this case, every observation variable is defined in every state and after the execution of any action of the domain. We call *action-independent* the observation variables that provide useful information automatically, independently of the previous execution of an action. We require an action-independent observation variable o to satisfy, for any $a_1, a_2 \in \mathcal{A}$ and any $s \in \mathcal{S}$, the following condition: $(\mathcal{X}_o(s, a_1, \top) \wedge \mathcal{X}_o(s, a_2, \top)) \vee (\mathcal{X}_o(s, a_1, \bot) \wedge \mathcal{X}_o(s, a_2, \bot))$ In the following, we write $\mathcal{X}_o \subseteq \mathcal{S} \times \{\top, \bot\}$ when o is action-independent. In a different formulation of the domain, an action (e.g. `ObsWallE`) could be required in order to acquire the value of a corresponding variable (e.g. `WallE`). In this case, `WallE` would be defined in any state after the action `ObsWallE`, and undefined otherwise. Such observation variables are modeled as *action-dependent*. Action-independent observation variables model "automatic sensing" [Tovey and Koenig, 2000], i.e. information that can always be acquired, as usual in embedded controllers, where a signal from the environment is sampled and acquired at a fixed rate, latched and internally available. Action-dependent observations are used in most observation-based approaches to planning (e.g. [Weld *et al.*, 1998]), where the value of a variable can be observed as the explicit effect of an action, like `ObsWallW`.

3 Conditional Plans

In partially observable domains, plans need to branch on conditions on the value of observable variables. A plan for a domain $\mathcal{D}$ is either the empty plan ϵ, an action $a \in \mathcal{A}$, the concatenation $\pi_1; \pi_2$ of two plans π_1 and π_2, or the conditional plan $o \ ? \ \pi_1 \ : \ \pi_2$ (read "if o then π_1 else π_2"), with $o \in \mathcal{O}$. For the example in Figure 1, a plan that moves the robot from the uncertain initial condition `NW` or `SW`, to state `SW` is `GoEast` ; (`WallN` ? `GoSouth` : ϵ) ; `GoWest`. Figure 1 depicts the corresponding execution. The action `GoEast` is executed first. Notice indeed that in the initial condition all the observation variables would have the same value for both `NW` and `SW`: therefore the states are indistinguishable, and it is pointless to observe. After the robot moves east, it is guaranteed to be either in `NE` or `SE`. The plan then branches on the value of `WallN`. This allows the planner to distinguish between state `NE` and `SE`: if the robot is in `NE`, then it moves south, otherwise it does nothing. At this point the robot is guaranteed to be in `SE`, and can finally move west. This plan is guaranteed to reach `SW` from any of the initial states, either with three actions (if the initial state is `NW`), or with two actions (if the initial state is `SW`).

We define $\mathcal{X}_o[\top, a] \doteq \{s \in \mathcal{S} : \mathcal{X}_o(s, a, \top)\}$ as the set of states in $\mathcal{S}$ where o is true, and $\mathcal{X}_o[\bot, a] \doteq \{s \in \mathcal{S} : \mathcal{X}_o(s, a, \bot)\}$ as the set of states in $\mathcal{S}$ where o is false. If o is undefined in a state s after a, then $s \in \mathcal{X}_o[\top, a] \cap \mathcal{X}_o[\bot, a]$. In the case of action-independent observations, we have $\mathcal{X}_o[\top] \doteq \{s \in \mathcal{S} : \mathcal{X}_o(s, \top)\}$, and $\mathcal{X}_o[\bot] \doteq \{s \in \mathcal{S} : \mathcal{X}_o(s, \bot)\}$. Under partial observability, plans have to work on sets of states whose elements cannot be distinguished, i.e., on "belief states". We say that an action a is applicable to a non empty belief state Bs iff a is

applicable in all states of Bs. We now define plan execution in the case of action-independent observations.

Definition 2 *Let $\emptyset \neq Bs \subseteq \mathcal{S}$. The execution of a plan in a set of states is defined as follows:*

1. *$Exec[\pi](\emptyset) \doteq \emptyset$;*
2. *$Exec[a](Bs) \doteq \{s' | \mathcal{R}(s, a, s'),$ with $s \in Bs\}$, if a is applicable in Bs;*
3. *$Exec[a](Bs) \doteq \emptyset$, if a is not applicable in Bs;*
4. *$Exec[\epsilon](Bs) \doteq Bs$;*
5. *$Exec[\pi_1; \pi_2](Bs) \doteq Exec[\pi_2](Exec[\pi_1](Bs))$;*
6. *$Exec[o\ ?\ \pi_1 : \pi_2](Bs) \doteq Exec[\pi_1](Bs \cap \mathcal{X}_o[\top]) \cup Exec[\pi_2](Bs \cap \mathcal{X}_o[\bot])$, if (a) if $Bs \cap \mathcal{X}_o[\top] \neq \emptyset$, then $Exec[\pi_1](Bs \cap \mathcal{X}_o[\top]) \neq \emptyset$ (b) if $Bs \cap \mathcal{X}_o[\bot] \neq \emptyset$, then $Exec[\pi_2](Bs \cap \mathcal{X}_o[\bot]) \neq \emptyset$*
7. *$Exec[o\ ?\ \pi_1 : \pi_2](Bs) \doteq \emptyset$ otherwise.*

We say that a plan π is applicable in $Bs \neq \emptyset$ iff $Exec[\pi](Bs) \neq \emptyset$. If the plan is applicable, then its execution is the set of all states that can be reached after the execution of the plan. For conditional plans, we collapse into a single set the execution of the two branches (item 6). The conditions *(a)* and *(b)* guarantee that both branches are executable. Definition 2 can be extended to the case of action-dependent observations by replacing $\mathcal{X}_o[\top]$ with $\mathcal{X}_o[\top, a]$ and $\mathcal{X}_o[\bot]$ with $\mathcal{X}_o[\bot, a]$, where a is the last action executed in the plan. Notice that, at starting time, action-dependent observation variables must be undefined since no action has been previously executed. For lack of space, we omit the explicit formal definition. We formalize the notion of planning problem under partial observability as follows.

Definition 3 (Planning Problem and Solution) *A planning problem is defined as a 3-tuple $\langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$, where $\mathcal{D}$ is a planning domain, $\emptyset \neq \mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, and $\emptyset \neq \mathcal{G} \subseteq \mathcal{S}$ is the set of goal states. The plan π is a solution to the problem $\langle \mathcal{I}, \mathcal{G}, \mathcal{D} \rangle$ iff $\emptyset \neq Exec[\pi](\mathcal{I}) \subseteq \mathcal{G}$.*

4 The Planning Algorithm

When planning under partial observability, the search space can be seen as an and-or graph over belief states, recursively constructed from the initial belief state, expanding each encountered belief state by every possible combination of applicable actions and observations. Consider the example in Figure 1, where $\mathcal{I}$ is $\{NW, SW\}$ and $\mathcal{G}$ is $\{SW\}$. For instance, belief state 1 expands in the or-node [(2)], representing the effect of action `GoEast`. Other actions are not applicable, and observation conveys no information. Belief state 2 expands in the or-node [(1) (3,4)], representing the effect of action `GoWest`, resulting in belief state 1, and the effect of observing `WallN`, resulting in the belief states 3 and 4. In order to find a solution for 2, it is either possible to find a solution for 1, or a solution for *both* 3 and 4. Non-applicable actions are discarded.

We plan under partial observability by exploring the and-or search space described above. The algorithm is basically a postorder traversal of the search space, proceeding forward from the initial belief state, and ruling out cyclic plans. The state of the search is stored by associating a mark to each encountered belief state. Possible marks are NONE, PROCESSING, SOLVED and VISITED, associated to a belief state Bs to distinguish between the following situations. NONE: Bs has not been previously encountered. SOLVED: a plan has been already found for Bs. PROCESSING: Bs is being processed (i.e. it is currently on the stack). VISITED: Bs has been previously processed, but the search has failed, and currently Bs is not on the stack. If a belief state marked PROCESSING is found, then the search has bumped into a cycle, and shall therefore fail. A VISITED Bs may deserve further expansion (and be therefore marked PROCESSING again). The primitives for recognizing and setting the mark of a belief state are ISBS⟨MARK⟩ and MARKBS⟨MARK⟩.

In order to avoid visiting over and over portions of the search space, we also store previous failures, associating with a belief state the set of belief states that were marked PROCESSING and blocked the search because of cycle detection. We store under which hypothesis did a search attempt fail, with the MEMOIZEFAILURE primitive. Before retrying to process a visited belief state Bs, the data base of failures is accessed with RETRIEVEFAILURE to check if any of the previous failures applies to the current situation, i.e. it is as-

```
procedure ORSRCH(OrNode)
1  res := ⟨Failure . Nil⟩;
2  while(OrNode ≠ Nil ∧ ISFAILURE(res))
3     res := MERGERESULTS(ANDSRCH(first(OrNode)),res);
4     OrNode := rest(OrNode);
5  return res;

procedure ANDSRCH(AndNode)
1  res := Success;
2  while(AndNode ≠ Nil ∧¬ISFAILURE(res))
3     res := BSSRCH(first(AndNode));
4     AndNode := rest(AndNode);
5  return res;

procedure BSSRCH(Bs)
1  if (ISBSPROCESSING(Bs))
2     return ⟨Failure . (Bs)⟩;
3  else if (ISBSSOLVED(Bs))
4     return Success;
5  else if(ISBSNONE(Bs) ∧ Bs ⊆ G)
6     MARKBSSOLVED(Bs);
7     return Success;
8  else
9     PrevFailure := RETRIEVEFAILURE(Bs);
10    if (ISFAILURE(PrevFailure))
11       return PrevFailure;
12    else
13       MARKBSPROCESSING(Bs);
14       res := ORSRCH(BSEXPAND(Bs));
15       if (ISFAILURE(res))
16          MARKBSVISITED(Bs);
17          MEMOIZEFAILURE(Bs,REMOVEBS(res,Bs));
18          return REMOVEBS(res,Bs);
19       else
20          MARKBSSOLVED(Bs);
21          return Success;
```

Figure 2: The planning algorithm

	Proc.	Vis.	Failures	Solved	Plan
1	1				
2	12				
3	123				
4	1234				
5	12345				
6	1234	5	5(4)		
7	123	54	5(4),4(3)		
8	1236	54	5(4),4(3)		
9	123	54	5(4),4(3)	6	$\pi_6 \doteq \epsilon$
10	12	54	5(4),4(3)	63	$\pi_3 \doteq \texttt{GoW}; \pi_6$
11	124	54	5(4),4(3)	63	
12	12	5	5(4)	634	$\pi_4 \doteq \texttt{GoS}; \pi_3$
13	1		5(4)	6342	$\pi_2 \doteq \texttt{WallN}\ ?\ \pi_4 : \pi_3$
14			5(4)	63421	$\pi_1 \doteq \texttt{GoE}; \pi_2$

Figure 3: The algorithm solves the example

sociated to Bs and all of the belief states contained in it are currently being processed. In this case, Bs is not processed, and the retrieved failure is returned.

The planning algorithm is presented in Figure 2. The algorithm is built on 3 recursive subroutines, each returning either *Success*, to signal that the search completed successfully, or a pair $\langle$*Failure . reason*$\rangle$, to signal that the search has failed because the belief states in *reason* are on the stack. (For lack of space, the plan construction steps carried out in case of search success are not reported here but only outlined in the case of the example.) ORSRCH processes an or-node, i.e. a list of and-nodes. And-nodes are repeatedly extracted from the or-node and used as input to ANDSRCH. The results are processed by MERGERESULTS, that constructs the return value by accumulating the set of the failure reasons of the different and searches. If a success is found, then it becomes the return value. Therefore, the search proceeds until a success is found, or the or-node is completely explored, in which case a failure is returned. ANDSRCH processes an and-node, trying to find a solution for each of the contained belief states. It selects the most promising belief state in the and-node, and uses it as input to BSSRCH. As soon as a failed search is detected, a failure-reason pair is propagated. If a success is received for each belief state, then a success is returned. BSSRCH processes a single belief state. It first checks if Bs is a loop back, in which case a failure (due to the Bs itself) is constructed and returned. In lines 3-4, the case of success is handled. In lines 5-7, a node that is encountered for the first time is checked against the goal. Then (lines 8-11), RETRIEVEFAILURE is called to check if Bs can be pruned based on a previous failure. Otherwise, Bs is put on the stack (line 13) and expanded by BSEXPAND, that constructs the corresponding or-node. This is provided in input to ORSRCH. If the result is a failure, then Bs is removed from the stack. The failure is stored disregarding Bs itself (primitive REMOVEBS) since when the search started it was not on the stack. The planner is invoked as BSSRCH($\mathcal{I}$).

Figure 3 depicts the data structures built by the algorithm while solving the example problem of Figure 1. For each step, we report the belief states marked PROCESSING and marked VISITED, the stored failures, the belief states marked SOLVED and the associated plan. Failure 5(4), introduced at step 6, means that the search started on belief state 5 failed because of a loop back on 4 that was PROCESSING. The last column associates the plan π_i to each belief state i becoming SOLVED. Belief state 6 is a subset of $\mathcal{G}$ and is thus associated with the empty plan. π_3 is the concatenation of the action GoW, that leads from 3 to 6, with π_6. The case for π_4 is similar. The conditional plan π_2 is constructed by ANDSRCH: the observation variable WallN associated with the and-node (3,4) being manipulated is the test of the plan, while the branches are the plans π_3 and π_4 associated to the successful belief states 3 and 4. Notice that belief state 4 is processed again at step 11, after the failure due to the loop-back on 3 at step 7. The storage of previous failures can speed up the search substantially, e.g., if from NW it were possible to enter in a different "branch" of the navigation domain that cannot lead to the solution.

5 The Planner

We integrated the algorithm described above in MBP [Cimatti *et al.*, 1998], a planner for nondeterministic domains based on Binary Decision Diagrams (BDD) [Bryant, 1992] and symbolic model checking techniques [McMillan, 1993]. MBP allows for conditional planning under full observability [Cimatti *et al.*, 1998], also considering temporally extended goals [Pistore and Traverso, 2001]. For this work, we extended MBP in two main directions. First, we developed a BDD-based implementation for the observation relation $\mathcal{X}$. Second, we implemented the search algorithm described in previous section. We rely on the machinery of [Bertoli *et al.*, 2001], where conformant planning is tackled as deterministic (rather than and-or) search in the space of belief states. Each visited belief state is represented by a unique BDD, while a hashing structure is used to efficiently implement the marking mechanism described in previous section. The expansion of a belief state Bs (primitive BSEXPAND) can be described in logical terms, and is implemented by means of BDD-based transformations. We first apply the symbolic expansion used in the case of conformant planning that computes the belief states corresponding to the execution from Bs of all the possible actions. Then, for each of the generated belief states Bs_i, we take into account the effect of observations by generating, for each o, the and-nodes of the form $(Bs_i \cap \mathcal{X}_o[\top], Bs_i \cap \mathcal{X}_o[\bot])$. However, in order to dominate the complexity of the application of the full combination of observations, we apply the following automatic, domain-independent simplifications. First, we analyze the domain to discover if it is possible to consider only one observation at a time without losing completeness. When this is not possible, we apply observations "set-wise" to Bs, i.e. if we consider o_i and o_j, the corresponding splits are both applied, resulting in the and-node $(Bs_i \cap \mathcal{X}_{o_i}[\bot] \cap \mathcal{X}_{o_j}[\bot], Bs_i \cap \mathcal{X}_{o_i}[\bot] \cap \mathcal{X}_{o_j}[\top], Bs_i \cap \mathcal{X}_{o_i}[\top] \cap \mathcal{X}_{o_j}[\bot], Bs_i \cap \mathcal{X}_{o_i}[\top] \cap \mathcal{X}_{o_j}[\top])$. In general, plans may be produced where the same observations are needlessly carried out on all branches, or useless actions precede observations. A special purpose procedure postprocesses the solution, getting rid of these sources of redundancy. Finally, the and-or search is driven by a simple selection heuristic that orders the or-node by delaying the expansion of the belief states where no observation has effect.

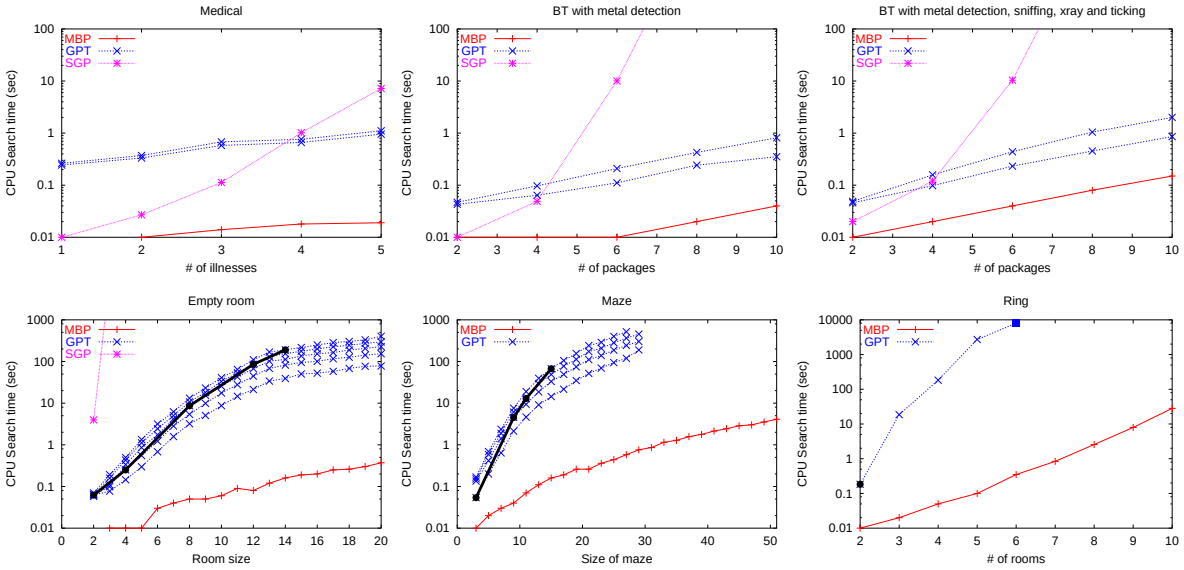


Table 1: Experimental results

6 Experimental Evaluation

We experimentally evaluated our approach against SGP [Weld *et al.*, 1998] and GPT [Bonet and Geffner, 2000]. (Other similar systems, e.g. CASSANDRA [Pryor and Collins, 1996], that are outperformed by SGP, as shown in [Weld *et al.*, 1998], are not considered.) SGP is based on GRAPHPLAN. It produces acyclic conditional plans, but it is unable to deal with non-deterministic action effects, i.e. uncertainty is limited to the initial condition. A planning graph is built for each initial state that can be distinguished by observation. GPT models planning domains as Markov Decision Processes, and, based on the probability distributions, produces a policy associating actions to belief states. The search is based on the repeated generation of learning/control trials, starting from randomly selected states in the initial belief state. The policy tends to improve as the number of trials grows. GPT cannot guarantee that the returned policies are acyclic.

We considered several test domains. The ones from the distributions of SGP turned out to be trivial, and therefore we report only the results for Medical and BT (see [Weld *et al.*, 1998] for a description). The Maze domains [Tovey and Koenig, 2000] are similar to the explanatory example in Figure 1, where a certain goal position has to be reached from a completely unknown initial position. The Empty room problem is basically a maze without internal walls. The Ring domain [Cimatti and Roveri, 2000] is a ring-shaped navigation domain, where each room has a window that the robot can observe and open/close. The goal is to have all windows closed, while the initial situation is unknown. We did not consider some of the domains from the GPT distribution. Some of these are meaningful only in a probabilistic setting, or admit only cyclic solutions (e.g. the Omelette domain).

The experiments were run on a Pentium II 300MHz with 512Mb of memory running Linux, fixing a memory limit of 500Mb and timeout to 1 hour CPU (unless otherwise specified). The results of the comparison are depicted in Table 1. Each plot refers to a problem class. We report on a logarithmic scale the search time (in seconds). The performance of SGP tends to degrade quite rapidly. The instances of BT with 8 packages reached the time limit. In the case of empty room, SGP was unable to solve the 3x3 version of the problem in 12 hours of CPU time. For GPT we report the average performance (over 10 runs) on increasing numbers of trial runs. In order to ensure a fair comparison, we would have liked to report the performance of GPT on the minimum number of trials needed for convergence. Unfortunately, detecting whether GPT has converged to a solution is not evident from the output. A necessary condition for convergence appears to be the existence of a successful trial for each possible initial state: therefore, the cardinality of the initial state (called n in the following) is a lower bound for convergence. The reported results correspond to increasing multiples of n (the computation time grows accordingly). As one increases the number of trials, the probability of GPT converging to a solution increases. For simpler problems like Medical and BT, GPT converges after a small number of trial runs. However, in the more complex problems, the number of initial states increases, as well as the required number of trials. This implies a growth of the computational resources needed, as clear from the results. For the Maze tests, the parser was unable to deal with problems larger than 29. For the Ring(6) domain, GPT fails to compute the heuristic function within the time limit. We tried to run it without heuristics, but it exhausted the available memory after 3 hours of CPU time. A very important point is that the difficulty of the problem slows down convergence, due to increasing possibility of failed and/or repeated trials upon certain initial states. The thick line with bullets, crossing the results of GPT, indicates up to which problem size GPT reached convergence at least once. For instance, in the empty room domain, GPT did not find a solution with $4 \cdot n$ for room size larger than 14 in any of the attempted 10 runs.

Similarly for the mazes with size larger that 15. MBP tackles the analyzed problems quite well. Search in the belief space avoids the explosion following from the enumeration of initial states, while the use of symbolic data structures limits memory requirements. The produced plans are of reasonable length, with the exception is the ring domain, where the selection function is not effective: combined with the depth-first search of the algorithm, this results in extremely intricated plans. Further research is needed to tackle this problem.

7 Related Work and Conclusions

In this paper we have presented a novel approach to conditional planning under partial observability. The approach is based on a model of observation that encompasses automatic sensing [Tovey and Koenig, 2000] and action-based sensing [Cassandra *et al.*, 1994; Weld *et al.*, 1998; Bonet and Geffner, 2000]. See also [Goldman and Boddy, 1996] for a similar model of observation. The planning algorithm is based on the exploration of a (possibly cyclic) and-or graph induced by the domain. It is different from heuristic search algorithms like AO*, that are based on the assumption that and-or search graphs are acyclic. Given the exhaustive style of the exploration, the algorithm can decide whether the problem admits an acyclic solution, i.e. a plan guaranteed to reach the goal in a finite number of steps. The algorithm is efficiently implemented in the MBP planner by means of BDD-based symbolic model checking techniques. We show that MBP outperforms the SGP and GPT planners. Another interesting system is QBFPLAN [Rintanen, 1999a], that extends the SAT-based approach to planning to the case of nondeterministic domains. The planning problem is reduced to a QBF satisfiability problem, that is then given in input to an efficient solver [Rintanen, 1999b]. QBFPLAN relies on a symbolic representation, but the approach seems to be limited to plans with a bounded execution length. The search space is significantly reduced by providing the branching structure of the plan as an input to the planner.

The problem of planning under partial observability has been deeply investigated in the framework of Partially Observable MDP (see, e.g., [Cassandra *et al.*, 1994; Hansen and Zilberstein, 1998; Poupart and Boutilier, 2000]). GPT follows this approach. Methods that interleave planning and execution [Koenig and Simmons, 1998; Genesereth and Nourbakhsh, 1993] can be considered alternative (and orthogonal) approaches to the problem of planning off-line with large state spaces. However, these methods cannot guarantee to find a solution, unless assumptions are made about the domain. For instance, [Koenig and Simmons, 1998] assumes "safely explorable domains" without cycles. [Genesereth and Nourbakhsh, 1993] describes an off-line planning algorithm based on a breadth-first search on an and-or graph. The paper shows that the version of the algorithm that interleaves planning and execution is more efficient than the off-line version, both theoretically and experimentally.

Future research objectives are the extension of the partially observable approach presented in this paper to strong cyclic solutions [Cimatti *et al.*, 1998] and for temporally extended goals [Kabanza *et al.*, 1997]. We will also investigate the use of heuristic search techniques, and the extension to the case of planning with non-deterministic/noisy sensing.

References

[Bertoli *et al.*, 2001] P.G. Bertoli, M. Roveri, and A. Cimatti. Heuristic Search + Symbolic Model Checking = Efficient Conformant Planning Proc. of IJCAI-2001.

[Bonet and Geffner, 2000] B. Bonet and H. Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. Proc. of AIPS-2000.

[Bryant, 1992] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318.

[Cassandra *et al.*, 1994] A. Cassandra, L. Kaelbling, and M. Littman. Acting optimally in partially observable stochastic domains. Proc. of AAAI-94.

[Cimatti and Roveri, 2000] A. Cimatti and M. Roveri. Conformant Planning via Symbolic Model Checking. *JAIR*, 13:305–338, 2000.

[Cimatti *et al.*, 1998] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. Proc. of AAAI-98.

[Genesereth and Nourbakhsh, 1993] M. Genesereth and I. Nourbakhsh. Time-saving tips for problem solving with incomplete information. Proc. of AAAI-93.

[Goldman and Boddy, 1996] R.P. Goldman and M.S. Boddy. Expressive Planning and Explicit Knowledge. Proc. of AIPS-96.

[Hansen and Zilberstein, 1998] E. A. Hansen and S. Zilberstein. Heuristic search in cyclic and-or graphs. Proc. of AAAI-98.

[Kabanza *et al.*, 1997] F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[Koenig and Simmons, 1998] S. Koenig and R. Simmons. Solving robot navigation problems with initial pose uncertainty using real-time heuristic search. Proc. of AIPS-1998.

[Littman *et al.*, 1998] Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. The computational complexity of probabilistic planning. *JAIR*, 9:1–36.

[McMillan, 1993] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[Pistore and Traverso, 2001] M. Pistore and P. Traverso. Planning as Model Checking for Extended Goals in Non-deterministic Domains. Proc. of IJCAI-2001.

[Poupart and Boutilier, 2000] P. Poupart and C. Boutilier. Value-directed belief state approximation for POMDPs. Proc. of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI-2000).

[Pryor and Collins, 1996] L. Pryor and G. Collins. Planning for Contingency: a Decision Based Approach. *JAIR*, 4:81–120.

[Rintanen, 1999a] J. Rintanen. Constructing conditional plans by a theorem-prover. *JAIR*, 10:323–352.

[Rintanen, 1999b] J. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. Proc. of IJCAI-99.

[Smith and Weld, 1998] David E. Smith and Daniel S. Weld. Conformant graphplan. Proc. of AAAI-98.

[Tovey and Koenig, 2000] C. Tovey and S. Koenig. Gridworlds as testbeds for planning with incomplete information. Proc. of AAAI-2000.

[Weld *et al.*, 1998] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending graphplan to handle uncertainty and sensing actions. Proc. of AAAI-98.

Planning as Model Checking for Extended Goals in Non-deterministic Domains

Marco Pistore and **Paolo Traverso**
ITC-IRST, Via Sommarive 18, 38050 Povo, Trento, Italy
{pistore,traverso}@irst.itc.it

Abstract

Recent research has addressed the problem of planning in non-deterministic domains. Classical planning has also been extended to the case of goals that can express temporal properties. However, the combination of these two aspects is not trivial. In non-deterministic domains, goals should take into account the fact that a plan may result in many possible different executions and that some requirements can be enforced on all the possible executions, while others may be enforced only on some executions. In this paper we address this problem. We define a planning algorithm that generates automatically plans for extended goals in non-deterministic domains. We also provide preliminary experimental results based on an implementation of the planning algorithm that uses symbolic model checking techniques.

1 Introduction

Most real world planning domains are intrinsically "non-deterministic". This is the case, for instance, of several robotics, control, and space application domains. Most often, applications in non-deterministic domains require planners to deal with goals that are more general than sets of final desired states. The planner needs to generate plans that satisfy conditions on their whole execution paths, i.e., on the sequences of states resulting from execution. E.g., in a robotic application, we may need to specify that a mobile robot should "move to a given room while avoiding certain areas all along the path".

When dealing with non-deterministic domains, the task of extending the notion of goal leads to a main key issue, related to the fact that the execution of a given plan may non-deterministically result in more than one sequence of states. Consider the previous example in the robotics context. On the one hand, we would like to require a plan that guarantees to reach the room and also guarantees that dangerous areas are avoided. On the other hand, in several realistic domains, no plan might satisfy this strong requirement. We might therefore accept plans that satisfy weaker requirements, e.g., we might accept that the robot has a possibility of reaching the room without being guaranteed to do so, but it is however guaranteed to avoid dangerous areas. Alternatively, we may require a plan that guarantees that the robot reaches the desired location, just trying, if possible, to avoid certain areas, e.g., areas that are too crowded.

In this paper, we define and implement a planning algorithm that generates automatically plans for extended goals in non-deterministic domains. Extended goals are CTL formulas [Emerson, 1990]. They can express temporal conditions that take into account the fact that an action may non-deterministically result in different outcomes. Hence extended goals allow us to distinguish between temporal requirements on "all the possible executions" and on "some executions" of a plan.

The plans built by the algorithm are strictly more expressive than plans that simply map states to actions to be executed, like universal plans [Schoppers, 1987], memory-less policies [Bonet and Geffner, 2000], and state-action tables [Cimatti *et al.*, 1998; Daniele *et al.*, 1999]. Beyond expressing conditional and iterative behaviors, the generated plans can execute different actions is a state, depending on the previous execution history. This expressiveness is required to deal with extended goals.

We have implemented the planning algorithm inside MBP [Cimatti *et al.*, 1998]. MBP uses symbolic techniques based on BDDs [Burch *et al.*, 1992] that provide the ability to represent compactly and explore efficiently large state spaces. In the paper we present preliminary experimental results that show that the proposed algorithm works in practice.

This paper is structured as follows. We first define non-deterministic planning domains and extended goals. We then define the structure of plans that can achieve extended goals. Finally, we describe the planning algorithm, describe its implementation, and show some experimental results.

2 Planning Domains

A *(non-deterministic) planning domain* can be described in terms of *(basic) propositions*, which may assume different values in different *states*, of *actions* and of a *transition relation* describing how an action leads from one state to possibly many different states.

Definition 1 *A* planning domain D *is a tuple* $(\mathcal{B}, Q, A, \rightarrow)$, *where* $\mathcal{B}$ *is the finite set of (basic) propositions,* $Q \subseteq 2^{\mathcal{B}}$ *is the set of states,* A *is the finite set of actions, and* $\rightarrow \subseteq Q \times A \times Q$ *is the transition relation. We write* $q \xrightarrow{a} q'$ *for* $(q, a, q') \in \rightarrow$.

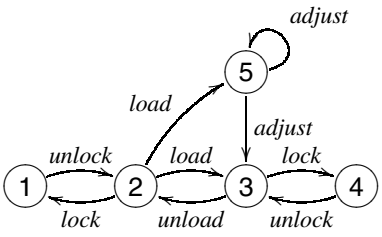


Figure 1: A simple non-deterministic domain

We require that the transition relation $\rightarrow$ is total, i.e., for every $q \in Q$ there is some $a \in A$ and $q' \in Q$ such that $q \xrightarrow{a} q'$. We denote with $\mathrm{Act}(q)$ the set of the actions that can be performed in state q: $\mathrm{Act}(q) = \{a : \exists q'.\, q \xrightarrow{a} q'\}$. We denote with $\mathrm{Exec}(q, a)$ the set of the states that can be reached from q performing action $a \in \mathrm{Act}(q)$: $\mathrm{Exec}(q, a) = \{q' : q \xrightarrow{a} q'\}$.

In Figure 1 we depict a simple planning domain, where an item can be loaded/unloaded to/from a container which can be locked/unlocked. Actions *load* and *adjust* are non-deterministic. *load* can either succeed, and lead to state 3 where the item is loaded correctly and the container can be locked, or it may fail, and lead to state 5, where the item needs to be adjusted to a correct position in order to lock the container. Action *adjust* may in its turn fail, and leave the item positioned incorrectly. For all states in the domain, we assume to have an action *wait* (not represented in the figure) that leaves the state unchanged. The basic propositions are loaded, locked and misplaced. loaded holds in states 3 and 4, locked in states 1 and 4, misplaced in state 5.

All the work presented in this paper is independent of the language for describing planning domains. However many of these languages (e.g., ADL-like languages as PDDL) are not able to represent non-deterministic domains and should be extended allowing for disjunctive effects of the actions. For instance, the action *load* might be described with an extension of PDDL as follows.

```
:action load
:precondition
  (and (not loaded) (not locked) (not misplaced))
:effect (OR (loaded) (misplaced))
```

3 Extended Goals

Extended goals are expressed with CTL formulas.

Definition 2 *Let $\mathcal{B}$ be the set of basic propositions of a domain D and let $b \in \mathcal{B}$. The syntax of an* (extended) goal *g for D is the following:*

$$g ::= \top \mid \bot \mid b \mid \neg b \mid g \wedge g \mid g \vee g \mid \mathrm{AX}\, g \mid \mathrm{EX}\, g \mid \mathrm{A}(g\, U\, g) \mid \mathrm{E}(g\, U\, g) \mid \mathrm{A}(g\, W\, g) \mid \mathrm{E}(g\, W\, g)$$

"X", "U", and "W" are the "next time", "(strong) until", and "weak until" temporal operators, respectively. "A" and "E" are the universal and existential path quantifiers, where a path is an infinite sequence of states. They allow us to specify requirements that take into account non-determinism. Intuitively, the formula $\mathrm{AX}\, g$ ($\mathrm{EX}\, g$) means that g holds in every (in some) immediate successor of the current state. $\mathrm{A}(g_1\, \mathrm{U}\, g_2)$ ($\mathrm{E}(g_1\, \mathrm{U}\, g_2)$) means that for every path (for some path) there exists an initial prefix of the path such that g_2 holds at the last state of the prefix and g_1 holds at all the other states along the prefix. The formula $\mathrm{A}(g_1\, \mathrm{W}\, g_2)$ ($\mathrm{E}(g_1\, \mathrm{W}\, g_2)$) is similar to $\mathrm{A}(g_1\, \mathrm{U}\, g_2)$ ($\mathrm{E}(g_1\, \mathrm{U}\, g_2)$) but allows for paths where g_1 holds in all the states and g_2 never holds. Formulas $\mathrm{AF}\, g$ and $\mathrm{EF}\, g$ (where the temporal operator "F" stands for "future" or "eventually") are abbreviations of $\mathrm{A}(\top\, \mathrm{U}\, g)$ and $\mathrm{E}(\top\, \mathrm{U}\, g)$, respectively. $\mathrm{AG}\, g$ and $\mathrm{EG}\, g$ (where "G" stands for "globally" or "always") are abbreviations of $\mathrm{A}(g\, \mathrm{W}\, \bot)$ and $\mathrm{E}(g\, \mathrm{W}\, \bot)$, respectively. A remark is in order: even if $\neg$ is allowed only in front of basic propositions, it is easy to define $\neg g$ for a generic CTL formula g, by "pushing down" the negations: for instance $\neg \mathrm{AX}\, g \equiv \mathrm{EX}\, \neg g$ and $\neg \mathrm{A}(g_1\, \mathrm{W}\, g_2) \equiv \mathrm{E}(\neg g_2\, \mathrm{U}(\neg g_1 \wedge \neg g_2))$.

Goals as CTL formulas allow us to specify different interesting requirements on plans. Let us consider first some examples of *reachability goals*. $\mathrm{AF}\, g$ ("reach g") states that a condition should be guaranteed to be reached by the plan, in spite of non-determinism. $\mathrm{EF}\, g$ ("try to reach g") states that a condition might possibly be reached, i.e., there exists at least one execution that achieves the goal. As an example, in Figure 1, the strong requirement (locked $\wedge$ $\neg$loaded) $\rightarrow$ AF (locked $\wedge$ loaded) cannot be satisfied, while the weaker requirement (locked $\wedge$ $\neg$loaded) $\rightarrow$ EF (locked $\wedge$ loaded) can be satisfied by unlocking the container, loading the item and then (if possible) locking the container. A reasonable reachability requirement that is stronger than $\mathrm{EF}\, g$ is $\mathrm{A}(\mathrm{EF}\, g\, \mathrm{W}\, g)$: it allows for those execution loops that have always a possibility of terminating, and when they do, the goal g is guaranteed to be achieved. In Figure 1, the goal (locked$\wedge$ $\neg$loaded) $\rightarrow$ A(EF (locked$\wedge$loaded) W(locked$\wedge$loaded)) can be satisfied by a plan that unlocks, loads, and, if the outcome is state 3, locks again, while if the item is misplaced (state 5) repeatedly tries to adjust the position of the item until (hopefully) state 3 is reached, and finally locks the container.

We can distinguish among different kinds of *maintainability goals*, e.g., $\mathrm{AG}\, g$ ("maintain g"), $\mathrm{AG}\, \neg g$ ("avoid g"), $\mathrm{EG}\, g$ ("try to maintain g"), and $\mathrm{EG}\, \neg g$ ("try to avoid g"). For instance, a robot should never harm people and should always avoid dangerous areas. Weaker requirements might be needed for less critical properties, like the fact that the robot should try to avoid to run out of battery.

We can *compose reachability and maintainability goals*. $\mathrm{AF\,AG}\, g$ states that a plan should guarantee that all executions reach eventually a set of states where g can be maintained. For instance, an air-conditioner controller is required to reach eventually a state such that the temperature can then be maintained in a given range. Alternatively, if you consider the case in which a pump might fail to turn on when it is selected, you might require that "there exists a possibility" to reach the condition to maintain the temperature in a desired range ($\mathrm{EF\,AG}\, g$). As a further example, the goal $\mathrm{AG\,EF}\, g$ intuitively means "maintain the possibility of reaching g".

Reachability – preserving goals make use of the "until operators" ($\mathrm{A}(g_1\, \mathrm{U}\, g_2)$ and $\mathrm{E}(g_1\, \mathrm{U}\, g_2)$) to express reachability goals while some property must be preserved. For instance, an air-conditioner might be required to reach a desired tem-

perature while leaving at least n of its m pumps off.

As a last example in the domain of Figure 1, consider the goal "from state 2, where the container is unlocked and empty, lock the container first, and then maintain the possibility of reaching a state where the item is loaded and the container is locked (state 4)". It can be formalized as $(\neg\mathsf{locked} \wedge \neg\mathsf{loaded} \wedge \neg\mathsf{misplaced}) \rightarrow (\mathrm{AF}\,(\mathsf{locked} \wedge \mathrm{AG}\,\mathrm{EF}\,(\mathsf{locked} \wedge \mathsf{loaded})))$. In the rest of the paper, we call this example of goal "*lock-then-load goal*".

Notice that in all examples above, the ability of composing formulas with universal and existential path quantifiers is essential. Logics that do not provide this ability, like LTL [Emerson, 1990], cannot express these kinds of goals[1].

4 Plans for Extended Goals

A plan describes the actions that have to be performed in a given state of the world. In order to satisfy extended goals, actions that have to be executed may also depend on the "internal state" of the executor, which can take into account, e.g., previous execution steps. Consider again the "lock-then-load goal" for domain in Figure 1. The plan, starting from state 2, has first to lead to state 1, and then to state 4. In state 2, the first time we have to execute action *lock*, while we have to *load* the item the second time. In general, a plan can be defined in terms of an *action function* that, given a state and an *execution context* encoding the internal state of the executor, specifies the action to be executed, and in terms of a *context function* that, depending on the action outcome, specifies the next execution context.

Definition 3 *A* plan *for a domain D is a tuple $\langle C, c_0, act, ctxt\rangle$, where:*

- *C is a set of (execution) contexts,*
- *$c_0 \in C$ is the initial context,*
- *$act : Q \times C \rightharpoonup A$ is the action function,*
- *$ctxt : Q \times C \times Q \rightharpoonup C$ is the context function.*

If we are in state q and in execution context c, then $act(q, c)$ returns the action to be executed by the plan, while $ctxt(q, c, q')$ associates to each reached state q' the new execution context. Functions *act* and *ctxt* may be partial, since some state-context pairs are never reached in the execution of the plan. An example of a plan that satisfies the lock-then-load goal is shown in Figure 2. Notice that the context changes from c_0 to c_1 when the execution reaches state 1. This allows the plan to execute different actions in state 2.

In the rest of the paper we consider only plans that are *executable and complete*. We say that plan π is *executable* if, whenever $act(q, c) = a$ and $ctxt(q, c, q') = c'$, then $q \xrightarrow{a} q'$. We say that π is *complete* if, whenever $act(q, c) = a$ and $q \xrightarrow{a} q'$, then there is some context c' such that $ctxt(q, c, q') = c'$ and $act(q', c')$ is defined. Intuitively, a complete plan always specifies how to proceed for all the possible outcomes of any action in the plan.

[1] In general, CTL and LTL have incomparable expressive power (see [Emerson, 1990] for a comparison). We focus on CTL since it provides the ability of expressing goals that take into account nondeterminism.

$act(\mathsf{2}, c_0) = lock$	$ctxt(\mathsf{2}, c_0, \mathsf{1}) = c_1$
$act(\mathsf{1}, c_1) = unlock$	$ctxt(\mathsf{1}, c_1, \mathsf{2}) = c_1$
$act(\mathsf{2}, c_1) = load$	$ctxt(\mathsf{2}, c_1, \mathsf{3}) = c_1$
	$ctxt(\mathsf{2}, c_1, \mathsf{5}) = c_1$
$act(\mathsf{5}, c_1) = adjust$	$ctxt(\mathsf{5}, c_1, \mathsf{5}) = c_1$
	$ctxt(\mathsf{5}, c_1, \mathsf{3}) = c_1$
$act(\mathsf{3}, c_1) = lock$	$ctxt(\mathsf{3}, c_1, \mathsf{4}) = c_1$
$act(\mathsf{4}, c_1) = wait$	$ctxt(\mathsf{4}, c_1, \mathsf{4}) = c_1$

Figure 2: An example of plan

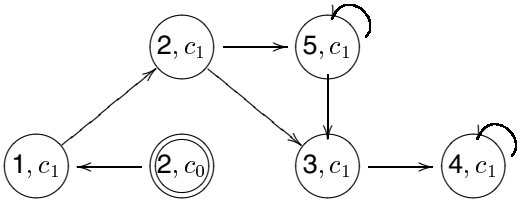


Figure 3: An example of execution structure

The execution of a plan results in a change in the current state and in the current context. It can therefore be described in terms of transitions between pairs state-context. Formally, given a domain D and a plan π, a transition of plan π in D is a tuple $(q, c) \xrightarrow{a} (q', c')$ such that $q \xrightarrow{a} q'$, $a = act(q, c)$, and $c' = ctxt(q, c, q')$. A *run* of plan π from state q_0 is an infinite sequence $(q_0, c_0) \xrightarrow{a_0} (q_1, c_1) \xrightarrow{a_1} (q_2, c_2) \xrightarrow{a_2} (q_3, c_3) \cdots$ where $(q_i, c_i) \xrightarrow{a_i} (q_{i+1}, c_{i+1})$ are transitions. Given a plan, we may have an infinite number of runs due to the nondeterminism of the domain. This is the case of the plan in Figure 2, since the execution can loop non-deterministically over the pair state-context $(\mathsf{5}, c_1)$. We provide a finite presentation of the set of all possible runs of a plan with an *execution structure*, i.e, a Kripke Structure [Emerson, 1990] whose set of states is the set of state-context pairs, and whose transition relation corresponds to the transitions of the runs.

Definition 4 *The* execution structure *of plan π in a domain D from state q_0 is the structure $K = \langle S, R, L\rangle$, where:*

- *$S = \{(q, c) : act(q, c)$ is defined$\}$,*
- *$((q, c), (q', c')) \in R$ if $(q, c) \xrightarrow{a} (q', c')$ for some a,*
- *$L(q, c) = \{b : b \in q\}$*

As an example, the execution structure of the plan in Figure 2 is depicted in Figure 3.

We define when a goal g is true in (q, c), written $K, (q, c) \models g$ by using the standard semantics for CTL formulas over the Kripke Structure K. The complete formal definition can be found in, e.g., [Emerson, 1990]. In order to make the paper self contained, we present here some cases. Propositional formulas are treated in the usual way. $K, (q, c) \models \mathrm{AX}\, g$ iff for every path $(q, c)_0(q, c)_1(q, c)_2 \cdots$, with $(q, c) = (q, c)_0$, we have $K, (q, c)_1 \models g$. $K, (q, c) \models \mathrm{A}(g_1\, \mathrm{U}\, g_2)$ iff for every path $(q, c)_0(q, c)_1(q, c)_2 \cdots$, with $(q, c) = (q, c)_0$, there exists $i \geq 0$ such that $K, (q, c)_i \models g_2$ and, for all $0 \leq j < i$, $K, (q, c)_j \models g_1$. The definition is similar in the case of existential path quantifiers. We can now define the notion of plan that satisfies a given goal.

Definition 5 *Let D be a planning domain and g be a goal for D. Let π be a plan for D and K be the corresponding execution structure. Plan π satisfies goal g from initial state q_0, written $\pi, q_0 \models g$, if $K, (q_0, c_0) \models g$. Plan π satisfies goal g from the set of initial states Q_0 if $\pi, q_0 \models g$ for each $q_0 \in Q_0$.*

For instance, the plan in Figure 2 satisfies the lock-then-load goal from state 2.

5 Planning Algorithm

The planning algorithm searches through the domain by trying to satisfy a goal g in a state q. Goal g defines conditions on the current state and on the next states to be reached. Intuitively, if g must hold in q, then some conditions must be projected to the next states. The algorithm extracts the information on the conditions on the next states by "progressing" the goal g. For instance, if g is $\mathrm{EF}\, g'$, then either g' holds in q or $\mathrm{EF}\, g'$ must still hold in some next state, i.e., $\mathrm{EX}\,\mathrm{EF}\, g'$ must hold in q. One of the basic building blocks of the algorithm is the function *progr* that rewrites a goal by progressing it to next states. *progr* is defined by induction on the structure of goals.

- $\mathit{progr}(q, \top) = \top$ and $\mathit{progr}(q, \bot) = \bot$;
- $\mathit{progr}(q, b) =$ if $b \in q$ then $\top$ else $\bot$;
- $\mathit{progr}(q, \neg b) =$ if $b \in q$ then $\bot$ else $\top$;
- $\mathit{progr}(q, g_1 \wedge g_2) = \mathit{progr}(q, g_1) \wedge \mathit{progr}(q, g_2)$;
- $\mathit{progr}(q, g_1 \vee g_2) = \mathit{progr}(q, g_1) \vee \mathit{progr}(q, g_2)$;
- $\mathit{progr}(q, \mathrm{AX}\, g) = \mathrm{AX}\, g$ and $\mathit{progr}(q, \mathrm{EX}\, g) = \mathrm{EX}\, g$;
- $\mathit{progr}(q, \mathrm{A}(g \,\mathrm{U}\, g')) = (\mathit{progr}(q, g) \wedge \mathrm{AX}\,\mathrm{A}(g \,\mathrm{U}\, g')) \vee \mathit{progr}(q, g')$ and $\mathit{progr}(q, \mathrm{E}(g \,\mathrm{U}\, g')) = (\mathit{progr}(q, g) \wedge \mathrm{EX}\,\mathrm{E}(g \,\mathrm{U}\, g')) \vee \mathit{progr}(q, g')$;
- $\mathit{progr}(q, \mathrm{A}(g \,\mathrm{W}\, g')) = (\mathit{progr}(q, g) \wedge \mathrm{AX}\,\mathrm{A}(g \,\mathrm{W}\, g')) \vee \mathit{progr}(q, g')$ and $\mathit{progr}(q, \mathrm{E}(g \,\mathrm{W}\, g')) = (\mathit{progr}(q, g) \wedge \mathrm{EX}\,\mathrm{E}(g \,\mathrm{W}\, g')) \vee \mathit{progr}(q, g')$.

The formula $\mathit{progr}(q, g)$ can be written in a normal form. We write it as a disjunction of two kinds of conjuncts, those of the form $\mathrm{AX}\, f$ and those of the form $\mathrm{EX}\, h$, since we need to distinguish between formulas that must hold in all the next states and those that must hold in some of the next states:

$$\mathit{progr}(q, g) = \bigvee_{i \in I} \Big(\bigwedge_{f \in A_i} \mathrm{AX}\, f \wedge \bigwedge_{h \in E_i} \mathrm{EX}\, h \Big)$$

where $f \in A_i$ ($h \in E_i$) if $\mathrm{AX}\, f$ ($\mathrm{EX}\, h$) belongs to the i-th disjunct of $\mathit{progr}(q, g)$. We have $|I|$ different disjuncts that correspond to alternative evolutions of the domain, i.e., to alternative plans we can search for. In the following, we represent $\mathit{progr}(q, g)$ as a set of pairs, each pair containing the A_i and the E_i parts of a disjunct:

$$\mathit{progr}(q, g) = \{(A_i, E_i) \mid i \in I\}$$

with $\mathit{progr}(q, \top) = \{(\emptyset, \emptyset)\}$ and $\mathit{progr}(q, \bot) = \emptyset$.

Given a disjunct (A, E) of $\mathit{progr}(q, g)$, we can define a function that assigns goals to be satisfied to the next states. We denote with $\mathit{assign\text{-}progr}((A, E), Q)$ the set of all the possible assignments $i : Q \to 2^{A \cup E}$ such that each universally quantified goal is assigned to all the next states (i.e., if $f \in A$ then $f \in i(q)$ for all $q \in Q$) and each existentially quantified goal is assigned to one of the next states (i.e., if $h \in E$ and $h \notin A$ then $f \in i(q)$ for one particular $q \in Q$). Consider the following example in the domain of Figure 1. Let g be $\mathrm{AF}\,\mathsf{locked} \wedge \mathrm{EX}\,\mathsf{misplaced} \wedge \mathrm{EX}\,\mathsf{loaded}$ and let the current state q be 2. We have that $\mathrm{AX}\,\mathrm{AF}\,\mathsf{locked} \wedge \mathrm{EX}\,\mathsf{misplaced} \wedge \mathrm{EX}\,\mathsf{loaded} \in \mathit{progr}(2, g)$. If we consider action *load*, the next states are 3 and 5. Then $\mathrm{AF}\,\mathsf{locked}$ must hold in 3 *and* in 5, while misplaced and loaded must hold in 3 *or* in 5. We have therefore four possible state-formulas assignments $i_1, \ldots, i_4$ to be explored (in the following we write f_1 for $\mathrm{AF}\,\mathsf{locked}$, h_1 for misplaced, and h_2 for loaded):

$$\begin{array}{ll} i_1(3) = f_1 \wedge h_1 \wedge h_2 & i_1(5) = f_1 \\ i_2(3) = f_1 \wedge h_1 & i_2(5) = f_1 \wedge h_2 \\ i_3(3) = f_1 \wedge h_2 & i_3(5) = f_1 \wedge h_1 \\ i_4(3) = f_1 & i_4(5) = f_1 \wedge h_1 \wedge h_2 \end{array}$$

In this simple example, it is easy to see that the only assignment that may lead to a successful plan is i_3.

Given the two basic building blocks *progr* and *assign-progr*, we can now describe the planning algorithm *build-plan* that, given a goal g_0 and an initial state q_0, returns either a plan or a failure.[2] The algorithm is reported in Figure 4. It performs a depth-first forward search: starting from the initial state, it picks up an action, progresses the goal to successor states, and iterates until either the goal is satisfied or the search path leads to a failure. The algorithm uses as the "contexts" of the plan the list of the active goals that are considered at the different stages of the exploration. More precisely, a context is a list $c = [g_1, \ldots, g_n]$, where the g_i are the active goals, as computed by functions *progr* and *assig-progr*, and the order of the list represents the *age* of these goals: the goals that are active since more steps come first in the list.

The main function of the algorithm is function *build-plan-aux*$(q, c, \mathit{pl}, \mathit{open})$, that builds the plan for context c from state q. If a plan is found, then it is returned by the function. Otherwise, $\bot$ is returned. Argument *pl* is the plan built so far by the algorithm. Initially, the argument passed to *build-plan-aux* is $\mathit{pl} = \langle C, c_0, \mathit{act}, \mathit{ctxt} \rangle = \langle \emptyset, g_0, \emptyset, \emptyset \rangle$. Argument *open* is the list of the pairs state-context of the currently open problems: if $(q, c) \in$ *open* then we are currently trying to build a plan for context c in state q. Whenever function *build-plan-aux* is called with a pair state-context already in *open*, then we have a loop of states in which the same sub-goal has to be enforced. In this case, function *is-good-loop*$((q, c), \mathit{open})$ is called that checks whether the loop is valid or not. If the loop is good, plan *pl* is returned, otherwise function *build-plan-aux* fails.

Function *is-good-loop* computes the set *loop-goals* of the goals that are active during the whole loop: iteratively, it considers all the pairs (q', c') that appear in *open* up to the next occurrence of the current pair (q, c), and it intersects *loop-goals* with the set **setof**(c') of the goals in list c'. Then, function *is-good-loop* checks whether there is some strong until goal among the *loop-goals*. If this is a case, then the loop is bad: the semantics of CTL requires that all the strong until goals are eventually fulfilled, so these goals should not

[2] It is easy to extend the algorithm to the case of more than one initial state.

```
function build-plan(q_0, g_0) : Plan
  return build-plan-aux(q_0, [g_0], ⟨∅, g_0, ∅, ∅⟩, ∅)

function build-plan-aux(q, c, pl, open) : Plan
  if (q, c) ∈ open then
    if is-good-loop((q, c), open) then return pl
    else return ⊥
  if defined pl.act[q, c] then return pl
  foreach a ∈ Act(p) do
    foreach (A, E) ∈ progr(q, c) do
      foreach i ∈ assign-progr((A, E), Exec(q, a)) do
        pl' := pl
        pl'.C := pl'.C ∪ {c}
        pl'.act[q, c] := a
        open' := conc((q, c'), open)
        foreach q' ∈ Exec(q, a) do
          c' := order-goals(i[q'], c)
          pl'.ctxt[q, c, q'] := c'
          pl' := build-plan-aux(q', c', pl', open')
          if pl' = ⊥ then next i
        return pl'
  return ⊥

function is-good-loop((q, c), open) : boolean
  loop-goals := setof(c)
  while (q, c) ≠ head(open) do
    (q', c') := head(open)
    loop-goals := loop-goals ∩ setof(c')
    open := tail(open)
  if ∃g ∈ loop-goals : g = A(_ U _) or g = E(_ U _) then
    return false
  else
    return true
```

Figure 4: The planning algorithm.

stay active during a whole loop. In fact, this is the difference between strong and weak until goals: executions where some weak until goal is continuously active and never fulfilled are acceptable, while the strong untils should be eventually fulfilled if they become active.

If the pair (q, c) is not in *open* but it is in the plan *pl* (i.e., (q, c) is in the range of function *act* and hence condition "**defined** $pl.act[q, c]$" is true), then a plan for the pair has already been found in another branch of the search, and we return immediately with a success. If the pair state-context is neither in *open* nor in the plan, then the algorithm considers in turn all the executable actions a from state q, all the different possible progresses (A, E) returned by function *progr*, and all the possible assignments i of (A, E) to $\mathrm{Exec}(q, a)$. Function *build-plan-aux* is called recursively for each destination state in $q' \in \mathrm{Exec}(q, a)$. The new context is computed by function *order-goals*$(i[q'], c)$: this function returns a list of the goals in $i[q']$ that are ordered by their "age": namely those goals that are old (they appear in $i[q']$ and also in c) appear first, in the same order as in c, and those that are new (they appear in $i[q']$ but not in c) appear at the end of the list, in any order. Also, in the recursive call, argument *pl* is updated to take into account the fact that action a has been selected from state q in context g. Moreover, the new list of open problems is updated to **conc**$((q, c), open)$, namely the pair (q, c) is added in front of argument *open*.

Any recursive call of *build-plan-aux* updates the current plan *pl'*. If all these recursive calls are successful, then the final value of plan *pl'* is returned. If any of the recursive calls returns $\bot$, then the next combination of assign decomposition, progress component and action is tried. If all these combinations fail, then no plan is found and $\bot$ is returned.

As an example, call *build-plan*(2, lock-then-load) is successful and returns the plan in Figure 2 where c_0 and c_1 are goals $\mathrm{AF}\,(\mathsf{locked} \wedge \mathrm{AG}\,\mathrm{EF}\,(\mathsf{locked} \wedge \mathsf{loaded}))$ and $\mathrm{AG}\,\mathrm{EF}\,(\mathsf{locked} \wedge \mathsf{loaded})$, respectively.

The algorithm always terminates, and it is correct and complete: given a state q of a domain D and a goal g for D, if *build-plan*$(q, g) = \pi$ then $\pi, q \models g$, and if *build-plan*$(q, g) = \bot$ then there is no plan π such that $\pi, q \models g$.

6 Symbolic Implementation and Experimental Results

We have implemented the planning algorithm and have performed some experimental evaluations. Though very preliminary, the experiments define some basic test cases for planning for CTL goals in non-deterministic domains, show that the approach is effective in practice with cases of significant complexity, and settle the basis for future comparisons.

We have implemented the algorithm inside MBP ([Cimatti *et al.*, 1998]). MBP uses symbolic techniques based on BDDs [Burch *et al.*, 1992] to overcome the problems of the explicit-state planning algorithm due to the huge size of realistic domains, and in particular of non-deterministic domains.

In order to provide a BDD-based implementation, the explicit algorithm presented in the previous section has to be revisited, taking into account the fact that BDDs work effectively on sets of states rather than on single states. For lack of space, we can not describe the symbolic BDD-based algorithm in details: further information on this algorithm, as well as on the test cases, can be found at the MBP home page `http://sra.itc.it/tools/mbp`.

One of the very few examples of planning for extended goals in non-deterministic domains that we have found in the literature is the "robot-moving-objects" search problem, presented in [Kabanza *et al.*, 1997] to test the SIMPLAN planner for some LTL-like goals. The domain consists of a set of rooms connected by doors, of a set of objects in the rooms, and of a robot that can grasp the objects and carry them to different rooms. The non-determinism in the domain is due to the fact that (some of) the doors are defective and can close without an explicit action of the robot.

We have performed experiments with different extended goals. For lack of space we report only the results for the goal of moving the objects into given rooms and keeping them there (experiment 1 in [Kabanza *et al.*, 1997]). In our framework, this goal is of the form $\mathrm{AF}\,\mathrm{AG}\,g$. The problem is parametrized in the number of the possible objects to be moved and in the number of defective doors. The time required to build the plan is reported in Figure 5 (all tests were

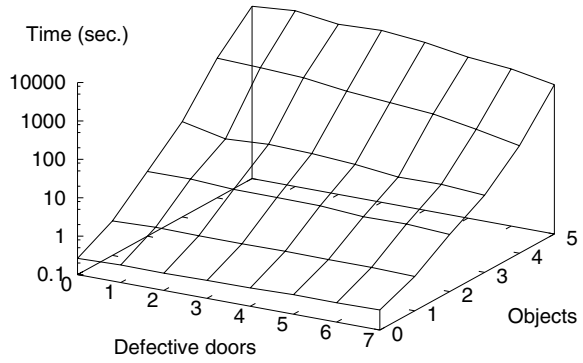


Figure 5: Experimental results

performed on a Pentium II 300 MHz with 512 Mb RAM of memory running Linux). The time scale is logarithmic and shows that the required time grows exponentially in the number of objects (this corresponds to an exponential growth in the size of the domain). Due to the usage of symbolic techniques, instead, the performance is not influenced by the non-determinism. We remark that in the case of 5 objects the domain is quite complex: it has more than 10^8 states.

The results reported in [Kabanza *et al.*, 1997] show a complementary behavior: SIMPLAN scales well with respect to the number of objects, but the explicit state search suffers significantly in the case of non-deterministic domains. We remark, however, that the efficient behavior of SIMPLAN in the case of deterministic domains depends on the enforced domain-dependent search control strategies, that are able to cut the largest part of the search graph. Our experiments show that MBP outperforms SIMPLAN, if the latter is executed without control strategies. This result is not surprising: symbolic techniques have shown to be dramatically more efficient that explicit techniques in the case of huge search space.

7 Conclusions and Related Work

In this paper we have presented an approach to automatic planning in non-deterministic domains where the goals are expressed as CTL formulas. We have implemented the algorithm by using symbolic model checking techniques, which open up the possibility to deal with large state space.

Some future objectives are the following. The symbolic implementation is still a rather naive transcription of the explicit algorithm presented in the paper: further work in needed to develop a symbolic algorithm that fully exploits the potentiality of BDDs and of the symbolic exploration of huge state spaces. Moreover, we plan to perform an extensive test with different kinds of extended goals on a set of realistic domains, to show that the approach is indeed practical. Finally, in this paper we focus on the case of full observability. An extension of the work to the case of planning for extended goals under partial observability is one of the main objectives for future research.

The problem of planning for CTL goals has never been solved before. The starting point of the work presented in this paper is the framework of "Planning via Symbolic Model Checking" (see, e.g., [Cimatti *et al.*, 1998; Daniele *et al.*, 1999; Bertoli *et al.*, 2001]). None of the previous works in this framework deals with temporally extended goals. The issue of "temporally extended goals" is certainly not new. However, most of the works in this direction restrict to deterministic domains, see for instance [de Giacomo and Vardi, 1999; Bacchus and Kabanza, 1998]. A work that considers extended goals in non-deterministic domains is described in [Kabanza *et al.*, 1997]: see Section 6 for a comparison.

Extended goals make the planning problem close to that of automatic synthesis of controllers (see, e.g., [Asarin *et al.*, 1995; Kupferman and Vardi, 1997]). However, most of the work in this area focuses on the theoretical foundations, without providing practical implementations. Moreover, it is based on rather different technical assumptions on actions and on the interaction with the environment.

References

[Asarin *et al.*, 1995] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid System II*, LNCS 999, 1995.

[Bacchus and Kabanza, 1998] F. Bacchus and F. Kabanza. Planning for Temporally Extended Goals. *Annals of Mathematics and Artificial Intelligence*, 22:5–27, 1998.

[Bertoli *et al.*, 2001] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. of IJCAI'01*, 2001.

[Bonet and Geffner, 2000] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc. of AIPS 2000*, 2000.

[Burch *et al.*, 1992] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[Cimatti *et al.*, 1998] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proc. of AAAI'98*, 1998.

[Daniele *et al.*, 1999] M. Daniele, P. Traverso, and M. Y. Vardi. Strong Cyclic Planning Revisited. In *Proc. of ECP'99*, 1999.

[de Giacomo and Vardi, 1999] G. de Giacomo and M.Y. Vardi. Automata-theoretic approach to planning with temporally extended goals. In *Proc. of ECP'99*, 1999.

[Emerson, 1990] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 16. Elsevier, 1990.

[Kabanza *et al.*, 1997] F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[Kupferman and Vardi, 1997] O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *Proc. of 2nd International Conference on Temporal Logic*, 1997.

[Schoppers, 1987] M. J. Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proc. of IJCAI'87*, 1987.

PLANNING

PLANNING WITH TEMPORAL UNCERTAINTY

Executing Reactive, Model-based Programs through Graph-based Temporal Planning

Phil Kim and **Brian C. Williams**
MIT Rm. 37-381
77 Massachusetts Ave.
Cambridge, MA 02139 USA
{kim,williams}@mit.edu

Mark Abramson
Draper Lab
555 Technology Square, MS3F
Cambridge, MA 02139 USA
mabramson@draper.com

Abstract

In the future, webs of unmanned air and space vehicles will act together to robustly perform elaborate missions in uncertain environments. We coordinate these systems by introducing a *reactive model-based programming language (RMPL)* that combines within a single unified representation the flexibility of embedded programming and reactive execution languages, and the deliberative reasoning power of temporal planners. The KIRK planning system takes as input a problem expressed as a RMPL program, and compiles it into a *temporal plan network (TPN)*, similar to those used by temporal planners, but extended for symbolic constraints and decisions. This intermediate representation clarifies the relation between temporal planning and causal-link planning, and permits a single task model to be used for planning and execution. Such a unified model has been described as a holy grail for autonomous agents by the designers of the Remote Agent[Muscettola *et al.*, 1998b].

1 Model-based Programming

The recent spread of advanced processing to embedded systems has created vehicles that execute complex missions with increasing levels of autonomy, in space, on land and in the air. These vehicles must respond to uncertain and often unforgiving environments, both with a fast response time and with a high assurance of first time success. The future looks to the creation of *cooperative robotic networks*. For example, a heterogenous collection of vehicles, such as planes, helicopters and boats, might work in concert to perform a search and rescue during a hurricane or similar natural disaster. In addition, giant space telescopes are being deployed that are composed of satellites carrying the telescope's different optical components. These satellites act in concert to image planets around other stars, or unusual weather events on earth.

The creation of robotic networks cannot be supported by the current programming practice alone. Recent mission failures, such as the Mars Climate Orbiter and Polar Landers, highlight the challenge of creating highly capable vehicles within realistic budget limits. Due to cost constraints, spacecraft flight software teams often do not have time to think through all the plausible situations that might arise, encode the appropriate responses within their software and then validate that software with high assurance. To break through this barrier we need to invent a new programming paradigm.

In this paper we advocate the creation of *embedded, model-based programming languages*. First, programmers should retain control for the overall success of a mission, by programming game plans and contingencies that in the programmer's experience will ensure a high degree of success. The programmer should be able to program these game plans using features of the best embedded programming languages available. For example, reactive synchronous languages[Halbwachs, 1993], like Esterel, Lustre and Signal, offer a rich set of constructs for interacting with sensors and actuators, for creating complex behaviors involving concurrency and preemption, and for modularizing these behaviors using all the standard encapsulation mechanisms. Model-based programming extends this style of reactive language with a minimal set of constructs neccessary to perform flexible mission coordination, while hiding its reasoning capabilities under the hood of the language's interpreter or compiler.

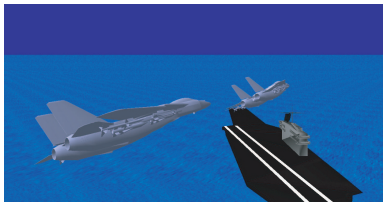
Second, we argue that model-based programming languages should focus on elevating the programmer's thinking, by automating the process of reasoning about low-level system interactions. Many recent space mission failures, such as Mars Climate Orbiter and Mars Polar Lander, can be isolated to difficulties in reasoning through low-level system interactions. On the other hand, this limited form of reasoning and book keeping is the hallmark of computational methods. The interpreter or compiler of a model-based program reasons through these interactions using composable models of the system being controlled. We are developing a language, called the *Reactive Model-Based Programming Language (RMPL)*, that supports four types of reasoning about system interactions: reasoning about contingencies, scheduling, inferring a system's hidden state and controlling that state. This paper develops RMPL in the context of contingencies and scheduling, while [Williams *et al.*, 2001], shows how RMPL is used to infer hidden state.

RMPL offers a middle ground between execution languages, like RAPS [Firby, 1995], and highly flexible, operator-based temporal planners,like HSTS [Muscettola *et al.*, 1998a]. RAPS offers the exception handling and concurrency mechanisms of embedded languages, while adding goal

monitoring, nondeterministic choice and metric constraints. However, RAPS makes its decisions reactively, without addressing concerns of schedulability and threat resolution, and hence can fall into a failure state. RMPL incorporates the forward looking planning and scheduling abilities of modern temporal planners, but can severely restrict the space of plans considered to possible threads of execution through the RMPL program. This speeds response and mitigates risk.

The paper begins by introducing a subset of RMPL that includes constructs from traditional reactive programming plus constructs for specifying contingencies and scheduling constraints. Second, we describe how *Kirk*, an RMPL-based planner/executive, compiles RMPL programs into *temporal plan networks (TPN)*, which compactly represent all possible threads of execution of an RMPL program, and all resource constraints and conflicts between concurrent activities. Third, we present Kirk's online planning algorithm for RMPL that "looks" by using network search algorithms to find threads of execution through the TPN that are temporally consistent. The result is a partially ordered temporal plan. Kirk then "leaps" by executing the plan using plan execution methods[Tsamardinos *et al.*, 1998] developed for Remote Agent[Muscettola *et al.*, 1998b]. Finally, we discuss Kirk's application to a simulated search and rescue mission.

2 Example: Cooperative Search and Rescue



As part of a search and rescue mission, consider an activity called Enroute, in which a group of vehicles fly together from a rendezvous point to the target search area. In this activity, the group selects one of two paths for traveling to the target area, flies together along the path through a series of waypoints to the target position, and then transmits a message to the forward air controller to indicate their arrival, while waiting until the group receives authorization to engage the target search area.

The two paths available for travel to the target area are each only available for a predetermined window of time, which is important to consider when selecting one of these paths. In addition, the timing of the Enroute activity is bound by externally imposed requirements, for example, the search and rescue mission must complete in 25-30 minutes, with 20% to 30% of the time allotted to the Enroute activity.

Codifying the Enroute activity requires most standard features of embedded languages. There are both sequential and concurrent threads of activities, such as going to a series of way points, and sending a message to the forward air controller (FAC), while concurrently awaiting authorization. There are maintenance conditions and synchronizations. For example, the air corridor needs to be maintained safe during flight, and synchronization occurs with the FAC.

In addition to constructs found in traditional embedded languages, we need constructs for expressing timing requirements and alternative choices or contingencies, in this example to use one of two corridors. These constructs are common to robotic execution languages[Firby, 1995]. However, they are only used reactively. Kirk must reason forward through the RMPL program's execution, identifying a course of action that is consistent.

3 RMPL Constructs

To summarize, RMPL needs to include constructs for expressing concurrency, maintaining conditions, synchronization, metric constraints and contingencies. The relevant RMPL constructs are as follows. We use lower case letters, like c, to denote activities or conditions, and upper case letters, like A and B, to denote well-formed RMPL expressions:

a. Invokes primitive activity a, starting at the current time. This is the basic construct for initiating activities.

c. Asserts that condition c is true at the current time, where c is a literal. This is the basic construct for asserting conditions.

if c **thennext** A. Starts executing A if condition c is currently satisfied, where c is a literal. This is the basic construct for expressing conditional branches and asserting preconditions.

do A **maintaining** c. Executes A, and ensures throughout A that c occurs. This is the basic construct for introducing maintenance conditions and protections.

A, B. Concurrently executes A and B. It is the basic construct for forking processes.

$A; B$. Consecutively executes A and then B. It is the basic construct for sequential processes.

$A[l, u]$. Constrains the duration of program A to be at least l and at most u. This is the basic construct for expressing timing requirements.

choose $\{A, B\}$. Reduces non-deterministically to program A or B. This is the basic construct for expressing multiple strategies and contingencies.

Note that together, c and **if** c **thennext** A provide the basic constructs for synchronization, by specifying required and asserted conditions. A, B and $A; B$ provide the neccessary constructs for building complex concurrent threads.

The "do maintaining" construct offers a building block for creating complex preemption and exception handling mechanisms. Note that to fully exploit these mechanisms Kirk would need to perform conditional planning. The algorithms presented in this paper only address unconditional planning. With this restriction "do maintaining" acts as a maintenance condition that Kirk must prove holds at planning time.

Using these constructs we express the Enroute activity as follows:

```
Group-Enroute()[l,u] = {
   choose {
      do {
         Group-Fly-Path(PATH1_1,PATH1_2,
            PATH1_3,TAI_POS)[l*90%,u*90%];

      } maintaining PATH1_OK,
      do {
         Group-Fly-Path(PATH2_1,PATH2_2,
            PATH2_3,TAI_POS)[l*90%,u*90%];
      } maintaining PATH2_OK
```

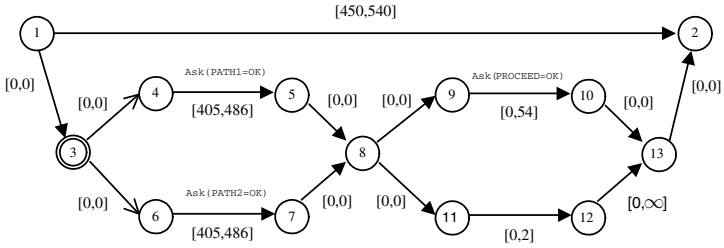
```
    };
    {
       Group-Transmit(FAC,ARRIVED_TAI)[0,2],
       do {
          Group-Wait(TAI_HOLD1,TAI_HOLD2)
             [0,u*10%]
       } watching PROCEED_OK
    }
}
```

The *choose* expression models the two options for flight paths. 90% of the total time of the overall maneuver is allocated to this group flight. Each flight has a maintenance condition that the flight path is okay. Arrival is transmitted to the forward air controller, and receipt of a message to proceed is concurrently monitored.

4 Temporal Plan Networks

Executing an RMPL program involves choosing a set of threads of execution (*Plans*), checking to ensure that the execution is consistent and schedulable, and then scheduling events on the fly. It is essential that we generate these plans quickly. This suggests compiling RMPL programs to a plan graph, along the lines of Graphplan or Satplan [Weld, 1999], and then searching the precompiled graph. However, it is also important for the plan to have the temporal flexibility offered by a partially ordered, temporal plan. Least commitment leaves slack to adapt to execution uncertainties and to recover from faults. This partial committment is expressed in temporal planning through a *Simple Temporal Network (STN)*[Dechter *et al.*, 1991]. Hence, a key observation of our approach is that to build in temporal flexibility we should build our graph-based plan representation, called a *Temporal Plan Network (TPN)*, as a generalization of an STN.

The TPN corresponding to the above Enroute program is shown below. Activity name labels are omitted to keep the figure clear, but the node pairs 4,5 and 6,7 represent the two Group-Fly-Path activities, and node pairs 9,10 and 11,12 correspond to the Group-Wait and Group-Transmit activities, respectively. Node 3 is a decision node that represents a choice between two methods for flying to the search area. The TPN represents the consequences of the constraint that the mission last between 25 and 30 minutes. It also models the decision between the two paths to the target area, and it models the restrictions that each of the paths can only be used if they are available.



A TPN encodes all feasible executions of an activity. It does this by augmenting an STN with two types of constraints: temporal constraints restrict the behavior of an activity by bounding the duration of an activity, time between activities, or more generally the temporal distance between two events. Symbolic constraints restrict the behavior of an activity by expressing the assertion or requirement of certain conditions by activities that all valid executions must satisfy.

For example, consider some of the possible executions of the Enroute activity. One possible execution is that the group flies along path one (pair 4,5) to the target area in 420 time units (seconds in this case), transmits an arrival message to the forward air controller (11,12) for one second, and concurrently waits (9,10) for another 40 seconds to receive authorization to proceed. Another possible execution is that the group selects the second path, flies to the target area in 500 seconds, takes 2 seconds to transmit the arrival message, and is authorized to proceed immediately. If it were the case that path one was available from the time at which the Enroute activity started to at least the time that the group arrived at the target area, then the first execution is valid. This is because it satisfies both the temporal constraints on the Enroute activity, and the requirement that path one is available for the duration of the flight along it. The planning algorithm presented in the next section performs the identification of consistent activity executions.

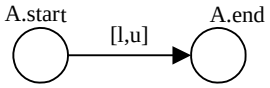
A Temporal Planning Network is a Simple Temporal Network, augmented with *symbolic constraints* and *decision nodes*. These additions are sufficent to capture all RMPL constructs given earlier. Like a simple temporal network, the nodes of a TPN represent temporal events, and the arcs represent temporal relations that constrain the temporal distance between events. An arc of a TPN may be labeled with a symbolic constraint Tell(c) or Ask(c), as well as a duration. A Tell(c) label on an arc (i,j) asserts that the condition represented by c is true over the interval between the temporal events modeled by the nodes i and j. Similarly, an Ask(c) label on an arc (i,j) requires that the condition represented by c is true over the interval represented by this arc. For example, in the Enroute TPN, the Ask(PATH1=OK) label on the arc (4,5) represents the requirement for path one to be available for the interval of time corresponding to the interval of time between the temporal event modeled by node 4 and node 5. These Ask-type symbolic constraints allow for the encoding of conditions in the network.

Decision nodes are used to explicitly introduce choices in activity execution that the planner must make. For example, in the Enroute activity there are two choices of paths for the group to use for flying to the target area, path one and path two. The activity model captures the two choices as out-arcs of node 3 of the enroute TPN. This decision node is designated by a double outline and dashed out-arcs. All other nodes in the Enroute TPN are non-decision nodes.

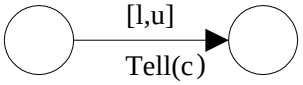
5 Compiling RMPL to TPN

Given a well formed RMPL expression, we compile it to a TPN by mapping each RMPL primitive to a TPN as defined below. RMPL sub-expressions, denoted by upper case letters, are recursively mapped to equivalent TPN:

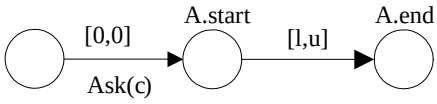
$A[l,u]$. Invoke activity A between l and u time units.



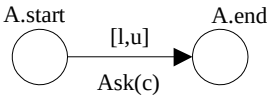
$c[l, u]$. Assert that condition c is true now until $[l, u]$.



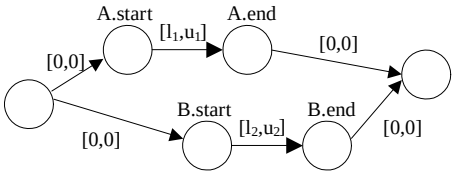
if c **thennext** $A[l, u]$. Execute A for $[l, u]$, if condition c is currently satisfied.



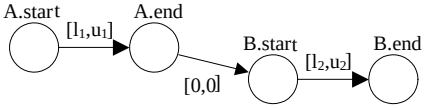
do $A[l, u]$ **maintaining** c. Execute A for $[l, u]$, and ensure throughout A that c occurs.



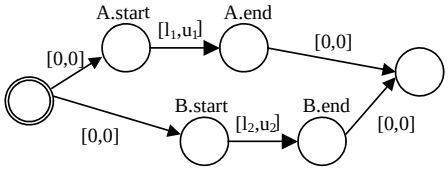
$A[l_1, u_1], B[l_2, u_2]$. Concurrently execute A for $[l_1, u_1]$ and B for $[l_2, u_2]$.



$A[l_1, u_1]; B[l_2, u_2]$. Execute A for $[l_1, u_1]$, then B for $[l_2, u_2]$.



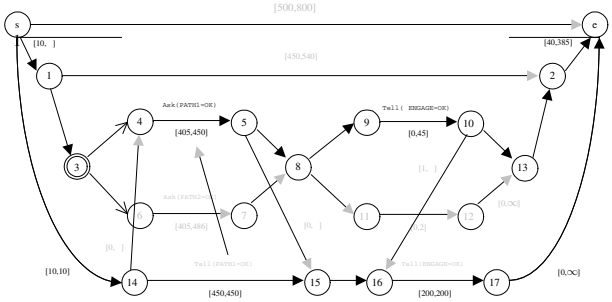
choose $\{A[l_1, u_1], B[l_2, u_2]\}$. Reduces to $A[l_1, u_1]$ or $B[l_2, u_2]$, non-deterministically.



6 Planning using TPNs

After compiling an RMPL program into a TPN, Kirk's planner uses the TPN to search for an execution that is both complete and consistent. The execution corresponds to an unconditional, temporal plan. A plan is complete if choices have been made for each relevant decision point, it contains only primitive-level activities, and all activities labeled Ask(c) have been linked to a Tell(c). A plan is consistent if it does not violate any of its temporal constraints or symbolic constraints. The resulting plan is then executed using the plan runner described in [Tsamardinos *et al.*, 1998].

The input to Kirk's planner is a TPN describing an activity scenario. A scenario consists of the TPN for the top-level activity invoked and any constraints on its invocation. The following TPN invokes Enroute (nodes 1-13). In a parallel thread it constrains the time ranges over which path one is available (nodes 14-15) and over which the vehicles may perform search (nodes 16-17).



The output of the planner consists of a set of paths through the input network from the start-node to the end-node of the top-level activity. In the example the paths s-1-3-4-5-8-9-10-13-2-e and s-14-15-16-17-e define a consistent execution. The first path defines the execution of the group of vehicles, and the second path defines the "execution" of the rest of the world in terms of the assertion or requirement of relevant conditions over the duration of the scenario. The portion of the TPN not selected for execution is shown in gray.

Planning involves two interleaved phases. The first phase resembles a network search that discovers the sub-network ,that constitute a feasible plan, while incrementally checking for temporal consistency. The second phase is analogous to the repair step of a causal link planner, in which threats are detected and resolved, and open conditions are closed[Weld, 1994].

6.1 Phase One: Select Plan Execution

The first phase selects a set of paths from the start-node to the end-node of the top-level activity. The planner handles this execution selection problem as a variant of a network search[Ahuja *et al.*, 1993] rooted at the start-node of the TPN encoding of the top-level activity.

Searching the Network

Recall that each node of a TPN is either a decision node or a non-decision node. If a plan includes a non-decision node with multiple out-arcs, then all of these arcs and their tail nodes must be included in the plan. If a plan includes a decision node with multiple out-arcs, then the arcs represent alternate choices, and the planning algorithm selects exactly one to be included in the plan.

Network search completes only when all paths reach the end-node of the top-level activity, and the subnetwork of the TPN, defined by these paths, is temporally consistent. This corresponds to testing consistency of an STN[Dechter *et al.*, 1991], as discussed in the next section.

The first phase of planning is summarized by the *Modified Network Search algorithm*, shown below. The set A, is the set of active nodes, which are those nodes whose paths have not yet been fully extended. The sets SN and SA are the sets of selected nodes and selected arcs, respectively:

```
1 Modified-Network-Search( N )
2    A = { start-node of N };
3    SN = { start-node of N };
4    SA = { };
5    While ( A is not empty )
```

```
6       Node = Select and remove a member of A;
7       If ( Node is a decision-node )
8          Arc = Select any unmarked out-arc of Node and
9          Mark Arc and
10          Add Arc to SA;
11          If ( tail of Arc is not in SN  )
12             Add tail of Arc to A and SN;
13          End-If
14       Else
15          For each Arc that is an out-arc of Node
16             Add Arc to SA;
17             If ( tail of Arc is not in SN  )
18                Add tail of Arc to A and SN;
19             End-If
20          End-For
21       End-If
22
23       If ( Cycle-Induced(SN, SA) )
24          If ( Not(Temporally-Consistent(SN, SA)) )
25             Backtrack(SN, SA, A);
26          End-If
27       End-If
28    End-While
29 End-Function
```

The algorithm extends an active node at each iteration. Decision nodes are treated by extending the path along one out arc (lines 8-13), while non-decision nodes are treated by branching the path and extending along all out arcs (lines 15-20). At the end of each iteration of the main While-loop, the modified network search tests for temporal consistency (lines 24-26). If the test fails, then the search calls Backtrack(..) in line 25, which reverts SN, SA, and A to their states before the most recent decision that has unmarked choices remaining, and selects a different out-arc. While for simplicity this explanation uses chronological backtracking, a wealth of more efficient search algorithms can be applied.

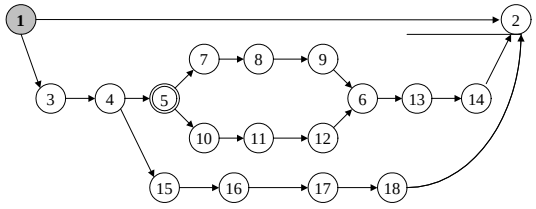
Note that it is not necessary to check temporal consistency after every iteration of the While-loop, since as long as no cycles are induced in the network, there is no way for a temporal inconsistency to be induced. Determining whether a cycle has been created can be done for each arc that is selected by checking whether the arc's tail node has already been selected. Since this can be done in constant time, it is significantly more efficient in practice than testing temporal consistency after every iteration, although it doesn't impact worst case complexity.

Also note that the algorithm stops extending a path when it encounters a node that is already in SN. The fact that this node is already in SN implies that two concurrent threads of execution have merged.

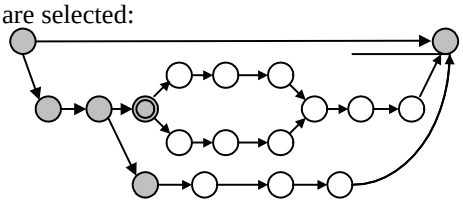
Finally, after the modified network search completes, the selected nodes and arcs define a set of paths from the start-node to the end-node of the top activity.

Example:Searching the Enroute Network

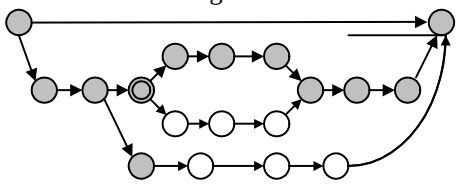
To illustrate the modified network search, we return to the Enroute input network, where node 1 is the start-node and node 2 is the end-node:



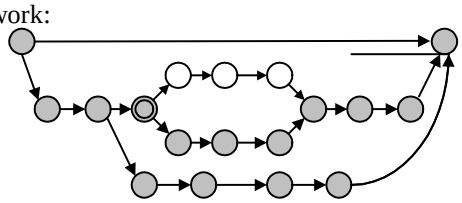
Initially, node 1 is selected, which is indicated by its darker shade, and it is active. In the first iteration, Kirk chooses node 1 from the set of active nodes, and since node 1 is not a decision node, it selects all out-arcs and adds their tails to the selected and active set. This continues until both node 5 and node 15 are selected:



At this point, the modified network search chooses node 5 from the active set. Since node 5 is a decision node, the algorithm must choose either arc (5,7) or arc (5,10). It selects arc (5,7) and continues extending until it reaches the following:



Note that arc (14,2) is selected, forming the cycle, 1-3-4-5-7-8-9-6-13-14-2-1, so the algorithm checks for temporal consistency. In this example, this selected sub-network is temporally inconsistent, so the algorithm backtracks to the most recent decision with open options, which is Node 5. Out-arc (5,10) has not yet been tried, so it is selected and the path extend to the end-node. Finally a path through arc (15,16) is found to the end-node, resulting in the temporally consistent sub-network:



Checking Temporal Consistency

To check temporal consistency we note that any subnet of a Plan Network, minus its symbolic constraint labels, forms a Simple Temporal Network. Hence temporal consistency can be checked using standard methods for Simple Temporal Networks [Dechter *et al.*, 1991]. Recall that an STN is consistent if and only if its encoding as a distance graph contains no negative cycles [Dechter *et al.*, 1991]. There exist several well known algorithms for detecting negative cycles in polynomial time. The Bellman-Ford algorithm [Cormen *et al.*, 1990] can be used to check for negative cycle in $O(nm)$ time, where m and n are the number of arcs and nodes in the distance graph, respectively. This algorithm only needs to maintain one distance label at each node, which takes only $O(n)$ space. A variant of this algorithm is used by HSTS [Muscettola *et al.*, 1998a] for fast inconsistency detection.

The algorithm we use in the Kirk planner is a variant of the generic label-correcting single-source shortest-path algorithm [Ahuja *et al.*, 1993], which takes $O(nm)$ worst-case asymptotic running time, but performs faster in many situations. This algorithm also requires only $O(n)$ space. Space

precludes a more detailed development.

6.2 Phase Two: Threats and Open Conditions

Symbolic constraints– Ask(c) and Tell(c) – are handled analogous to threats and open conditions in causal link planning[Weld, 1994]. Two symbolic constraints conflict if one is either asserting (by using Tell) or requesting (by using Ask) that a condition is true, and the second is asserting or requesting that the same condition is false. For example, Tell(Not(c)) and Ask(c) conflict. An open condition in a TPN appears as Ask constraints, which represent the need for some condition to be true over the interval of time represented by the arc labeled with the Ask constraint.

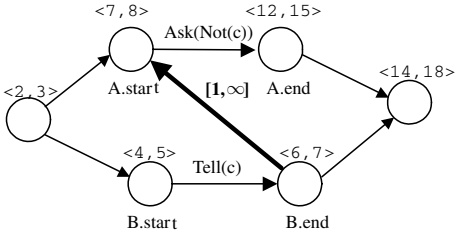
Resolving Threats

To detect threats the planner computes the feasible time bounds for each temporal event (node) in the network, and then uses these bounds to identify potentially overlapping intervals that are labeled with inconsistent constraints. These bounds can be computed by solving an all-pairs shortest-path problem over the distance graph of the partially completed plan. Kirk uses the Floyd-Warshall algorithm for computing all-pairs shortest paths. We are currently evaluating Johnson's algorithm which runs in $O(n^2 \log(n)+mn)$, or $O(n^2 \log(n))$ if $m = O(n)$.

Once these feasible time ranges are determined, the planner detects which arcs may overlap in time. If there are two arcs that may overlap and that are labeled with conflicting symbolic constraints, then they are resolved by ordering the intervals, if possible.

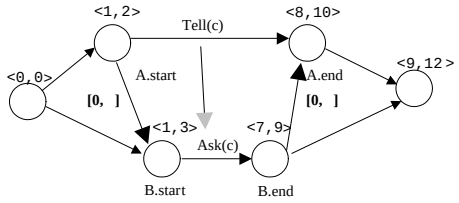
These interval pairs need to be identified efficiently. Kirk maintains an interval set data structure for each proposition p that keeps track of all intervals that assert or require p or its negation. In order to identify threats, the planner need only check each interval set for threats. This takes $O(si^2)$ asymptotic running time, where i is the maximum cardinality over all interval sets, and performs much better in practice because the interval sets typically have few elements. More sophisticated indexing schemes may improve performance, such as interval tree structures [Cormen *et al.*, 1990].

A threat is resolved by introducing temporal constraints. Each threat consists of two arcs that represent intervals of time that may overlap. To resolve threats we introduce a constraint that forces an ordering between the two activites, similar to promotion and demotion in classical planning[Weld, 1994]:



Closing Open Conditions

An open condition is represented by an arc labeled with an Ask constraint, which represents the request for a condition to be satisfied over the interval of time represented by the arc. If this interval of time is contained by another interval over which the condition is asserted by a Tell constraint, then the open condition is satisfied (i.e., closed), and a causal link is drawn from the Tell to the Ask. Open conditions are detected simply by scanning through all activites and checking any Ask constraints. Finding potentially overlapping intervals is performed using the same method described above for detecting threats. Once a Tell is found that can satisfy an open condition, temporal constraints are added so that the duration of the open condition is contained within the Tell. This method of closing open asks is also closely related to the way that the HSTS planner satisfies compatibilities [Muscettola *et al.*, 1998a]:



7 Implementation and Discussion

Kirk's compiler generates TPN specification files, and is written in Lisp. Kirk's planner, written in C++, generates a plan from the TPN and checks consistency. Kirk's executive, based on the remote agent plan runner [Tsamardinos *et al.*, 1998], takes the resulting partially ordered temporal plan and executes it on the multi-air vehicle simulator. The following table summarizes Kirk's performance on nominal plans for several activities within the search and rescue scenario. The fully expanded TPN generated from the Group-Search-and-Rescue activity included 273 nodes. The testing platform was an IBM Aptiva E6U with an Intel 400Mhz Pentium II processor and 128MB of RAM, running Redhat Linux version 6.1:

Top Activity	Nodes	Activities	Plan Time
Follow(..)	4	1	4 ms
Group-Rescue(..)	27	8	235 ms
Group-Enroute()	112	19	16 s
Group-SR-Mission()	273	47	404 s

"Top Activity" refers to the top-level activity that was being planned. "Nodes" is the size of the expanded TPN after planning. Usually, about half of these were included in the final plan, with the rest corresponding to unselected executions. "Activities" indicates the number of primitive activities included in the final plan. Finally, the "Plan Time" gives the time that it took for Kirk to generate a plan corresponding to each of these activities.

Kirk offers two sources for efficiency. First, typically an RMPL program significantly constrains the space of possible plans considered, in the spirit of hierarchical task network planners [Erol *et al.*, 1994]. Second, the use of TPNs reduces online planning to graph search. In the example Kirk does well with no search guidance up to about 100 nodes. At this point the time becomes dominated by the time required to compute feasible time bounds for events. This is due to the use of Bellman-Ford and chronological search. We are exploring a reimplementation based on Johnson's algorithm and a more sophisticated search strategy.

The primary contribution of this paper is the Reactive Model-based Programming Language and the Temporal Plan Network representation. The algorithms presented here only begin to explore RMPL/TPN-based planning. The following are some example directions for further research.

This paper focuses on the use of TPNs as a synthesis of causal link planning[Weld, 1994], temporal planning [Muscettola, 1994] and hierarchical task network planning[Erol *et al.*, 1994]. Can methods from graph-based planning[Blum and Furst, 1997; Weld, 1999; Smith and Weld, 1999], particularly mutual exclusion relationships, be effectively employed within a TPN? An important element of practical temporal planners in the space domain, such as HSTS[Muscettola, 1994] and IxTeT[Laborie and Ghallab, 1995], is the ability to plan with depletable resources. Can RMPL and TPNs be similarly extended? How can RMPL and TPNs be extended to support decision theoretic planning and agile maneuver planning, common to robotic vehicles?

RMPL offers an expressive embedded programming language, by inheriting most of its primitive combinators from the Timed Concurrent Constraint Language (TCC) [Saraswat *et al.*, 1996]. For example, as with TCC, these primitives allow a rich set of operators to be derived for preemption and exception handling, similar to those found in embedded languages like Esterel[Berry and Gonthier, 1992]. However, the algorithm presented here performs unconditional planning, and hence only considers the case where exceptions can be prevented. RMPL's ability to express exception handling mechanisms can best be exploited through the development of conditional planning algorithms.

Finally, RMPL allows the programmer to constrain the family of possible behaviors that the planner considers when controlling an embedded system. It is important that this family of behaviors be safe. Embedded languages like Esterel[Berry and Gonthier, 1992], Lustre[Halbwachs *et al.*,] and Signal[Guernic *et al.*,] offer a clean semantics, and offer support for direct machine verification of safety and liveness properties. The verification of RMPL programs would be similar, but requires methods, such as timed automata verification, that support metric constraints and non-determinism.

Acknowledgments

We would like to thank Michael Hofbaur, Tony Abad and the anonymous reviewers for their invaluable insights. This research is supported in part by the Office of Naval Research under contract N00014-99-1-1080 and by the DARPA MOBIES program under contract F33615-00-C-1702.

References

[Ahuja *et al.*, 1993] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[Berry and Gonthier, 1992] G. Berry and G. Gonthier. The *esterel* programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87 – 152, November 1992.

[Blum and Furst, 1997] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[Cormen *et al.*, 1990] T. Cormen, C Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Camb., MA, 1990.

[Dechter *et al.*, 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *AIJ*, 49:61–95, 1991.

[Erol *et al.*, 1994] K. Erol, J. Hendler, and D. Nau. Htn planning: Complexity and expressivity. In *Proceedings of AAAI-94*, pages 1123–1128, 1994.

[Firby, 1995] R. James Firby. The RAP language manual. Technical Report AAP-6, Univ. Chicago, March 1995.

[Guernic *et al.*,] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with *signal*. pages 1321–1336.

[Halbwachs *et al.*,] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language *lustre*. pages 1305–1320.

[Halbwachs, 1993] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic, 1993.

[Laborie and Ghallab, 1995] P. Laborie and M. Ghallab. Planning with sharable resource constraints. In *Proceedings of IJCAI-95*, 1995.

[Muscettola *et al.*, 1998a] N. Muscettola, P. Morris, B. Pell, and B. Smith. Issues in temporal reasoning for autonomous control systems. In *Autonomous Agents*, 1998.

[Muscettola *et al.*, 1998b] N. Muscettola, P. Nayak, B. Pell, and B. C. Williams. The new millennium remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2):5–48, 1998.

[Muscettola, 1994] N. Muscettola. HSTS: Integrating planning and scheduling. In Mark Fox and Monte Zweben, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.

[Saraswat *et al.*, 1996] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *J Symb Comp*, 22(5-6):475–520, 1996.

[Smith and Weld, 1999] D. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *Proceedings of IJCAI-99*, 1999.

[Tsamardinos *et al.*, 1998] I. Tsamardinos, N. Muscettola, and P. Morris. Fast transformation of temporal plans for efficient execution. In *Proceedings of AAAI-98*, 1998.

[Weld, 1994] D. Weld. An introduction to least commitment planning. In *AI Magazine*, 1994.

[Weld, 1999] D. Weld. Recent advances in ai planning. In *AI Magazine*, 1999.

[Williams *et al.*, 2001] B. C. Williams, S. Chung, and V. Gupta. Mode estimation of model-based programs: Monitoring systems with complex behavior. In *Proceedings of IJCAI-01*, 2001.

Dynamic Control Of Plans With Temporal Uncertainty

Paul Morris
Nicola Muscettola
NASA Ames Research Center
Moffett Field, CA 94035, U.S.A.
{pmorris,mus}@ptolemy.arc.nasa.gov

Thierry Vidal
LGP/ENIT
47, av d'Azereix - BP 1629
F-65016 Tarbes cedex - FRANCE
thierry@enit.fr

Abstract

Certain planning systems that deal with quantitative time constraints have used an underlying Simple Temporal Problem solver to ensure temporal consistency of plans. However, many applications involve processes of uncertain duration whose timing cannot be controlled by the execution agent. These cases require more complex notions of temporal feasibility. In previous work, various "controllability" properties such as Weak, Strong, and Dynamic Controllability have been defined. The most interesting and useful Controllability property, the Dynamic one, has ironically proved to be the most difficult to analyze. In this paper, we resolve the complexity issue for Dynamic Controllability. Unexpectedly, the problem turns out to be tractable. We also show how to efficiently execute networks whose status has been verified.

1 Introduction

Simple Temporal Networks [Dechter *et al.*, 1991] have proved useful in planning and scheduling applications that involve quantitative time constraints (e.g. [Laborie and Ghallab, 1995; Muscettola *et al.*, 1998b]) because they allow fast checking of temporal consistency. However this formalism does not adequately address an important aspect of real execution domains: the time of occurrence of some events may not be under the complete control of the execution agent. For example, when a spacecraft commands an instrument or interrogates a sensor, a varying amount of time may intervene before the operation is completed. In cases like this, the execution agent does not have freedom to select the precise time delay between events in accord with the timing of previously executed events. Instead, the value is selected by Nature independently of the agent's choices. This can lead to constraint violations during execution even if the Simple Temporal Network appeared consistent at plan generation time.

The problem of constraint satisfaction for temporal networks with uncertainty was first addressed formally in [Vidal and Ghallab, 1996; Vidal and Fargier, 1999]. In this setting, the question of temporal feasibility goes beyond mere consistency to encompass issues of "controllability." Essentially, a network is controllable if there is a strategy for executing the timepoints under the agent's control that satisfies all requirements, in all situations involving the uncontrolled timepoints. The previous work has identified three primary levels of controllability. In *Strong Controllability*, there is a static control strategy that is guaranteed to work in all cases. In *Weak Controllability*, for all situations there is a "clairvoyant" strategy that works if all uncertain durations are known when the network is executed. The most interesting controllability property from a practical point of view is *Dynamic Controllability*, where it is assumed that each uncertain duration becomes known (is observed) after it has finished, and the property requires a successful strategy that depends only on the past outcomes.

In previous work, algorithms have been presented for checking Strong and Weak Controllability, and Strong Controllability has been shown to be tractable, while Weak Controllability is co-NP-complete [Vidal and Fargier, 1999; Morris and Muscettola, 1999]. However, Dynamic Controllability has proved difficult to analyze, primarily because of a time asymmetry where a control decision may depend on the past but not on the future. In this paper we present efficient constraint propagation methods for checking Dynamic Controllability. These explicitly add constraints that are implicit in the Dynamic Controllability property. With these additional constraints, Dynamic Controllability checking reduces to a form of consistency checking that turns out to be polynomial. The derived constraints are also used to guide an effective execution strategy.

2 Background

We review the definitions of Simple Temporal Network [Dechter *et al.*, 1991], and Simple Temporal Network with Uncertainty [Vidal and Fargier, 1999].

A Simple Temporal Network (STN) is a graph in which the edges are labelled with upper and lower numerical bounds. The nodes in the graph represent temporal events or *timepoints*, while the edges correspond to constraints on the durations between the events. Formally, an STN may be described as a 4-tuple $< N, E, l, u >$ where N is a set of nodes, E is a set of edges, and $l : E \rightarrow \mathbb{R} \cup \{-\infty\}$ and $u : E \rightarrow \mathbb{R} \cup \{+\infty\}$ are functions mapping the edges into extended Real Numbers, that are the lower and upper bounds of the interval of possible durations. Each STN is associated with a *distance graph* [Dechter *et al.*, 1991] derived from the

upper and lower bound constraints. An STN is consistent if and only if the distance graph does not contain a negative cycle, and this can be determined by a single-source shortest path propagation such as in the Bellman-Ford algorithm [Cormen *et al.*, 1990]. To avoid confusion with edges in the distance graph, we will refer to edges in the STN as *links*.

A Simple Temporal Network With Uncertainty (STNU) is similar to an STN except the links are divided into two classes, *contingent links* and *requirement links*. Contingent links may be thought of as representing causal processes of uncertain duration; their finish timepoints, called *contingent timepoints*, are controlled by Nature, subject to the limits imposed by the bounds on the contingent links. All other timepoints, called *executable timepoints*, are controlled by the agent, whose goal is to satisfy the bounds on the requirement links. We assume the durations of contingent links vary independently, so a control procedure must consider every combination of such durations.

Thus, an STNU is a 5-tuple $< N, E, l, u, C >$, where N, E, l, u are as in a STN, and C is a subset of the edges: the contingent links, the others being requirement links. We assume $0 < l(e) < u(e) < \infty$ for each contingent link e.[1]

An STNU may be regarded as an STN by ignoring the distinction between contingent links and requirement links. This allows us to apply STN terminology and concepts, such as AllPairs shortest-path calculations, to STNUs.

In addition, choosing one of the allowed durations for each contingent link may be thought of as reducing the STNU to an ordinary STN. Thus, an STNU determines a family of STNs, as in the following definition.

Suppose $\Gamma = < N, E, l, u, C >$ is an STNU. A *projection* [Vidal and Ghallab, 1996] of Γ is a Simple Temporal Network derived from Γ where each requirement link is replaced by an identical STN link, and each contingent link e is replaced by an STN link with equal upper and lower bounds $[b, b]$ for some b such that $l(e) \leq b \leq u(e)$.

Given a fixed STNU $< N, E, l, u, C >$, a *schedule* T is a mapping

$$T : N \rightarrow \mathbb{R}$$

where $T(x)$, written T_x here, is called the *time* of time-point x. A schedule is *consistent* if it satisfies all the link constraints. From a schedule, we can determine the durations of all contingent links that finish prior to a timepoint x. (This may be viewed as a partial mapping from C to $\mathbb{R}$.) We call this the *prehistory* of x with respect to T, denoted by $T_{\prec x}$.

Then an *execution strategy* S is a mapping

$$S : \mathcal{P} \rightarrow \mathcal{T}$$

where $\mathcal{P}$ is the set of projections and $\mathcal{T}$ is the set of schedules. An execution strategy S is *viable* if $S(p)$ is consistent (w.r.t. p) for each projection p.

We are now ready to define the various types of controllability, essentially following [Vidal, 2000].

An STNU is *Weakly Controllable* if there is a viable execution strategy. This is equivalent to saying that every projection is consistent.

[1] If $l(e) = u(e)$, there is no uncertainty and we may as well replace e by a requirement link.

An STNU is *Strongly Controllable* if there is a viable execution strategy S such that

$$[S(p1)]_x = [S(p2)]_x$$

for each executable timepoint x and projections $p1$ and $p2$. Thus, a Strong execution strategy assigns a fixed time to each executable timepoint irrespective of the outcomes of the contingent links.

An STNU is *Dynamically Controllable* if there is a viable execution strategy S such that

$$[S(p1)]_{\prec x} = [S(p2)]_{\prec x} \Rightarrow [S(p1)]_x = [S(p2)]_x$$

for each executable timepoint x and projections $p1$ and $p2$. Thus, a Dynamic execution strategy assigns a time to each executable timepoint that may depend on the outcomes of contingent links in the past, but not on those in the future (or present). This corresponds to requiring that only information available from observation may be used in determining the schedule. We will use *dynamic strategy* in the following for a (viable) Dynamic execution strategy.

Networks where two contingent links have the same finishing point are clearly not Dynamically Controllable. Because of this, and for certain technical reasons (following [Morris and Muscettola, 2000]), we will exclude such networks in the remainder of this paper.

It is easy to see from the definitions that Strong Controllability implies Dynamic Controllability, which in turn implies Weak Controllability. Strong Controllability is known to be tractable and Weak Controllability is known to be co-NP-complete. In this paper, we investigate the status of Dynamic Controllability. Note that a naïve algorithm for checking this property is hyperexponential since it requires searching for an execution strategy that is both dynamic and viable, while a method described in [Vidal, 2000] requires worst case exponential space.

The following terminology will be useful in the subsequent discussion. A contingent link is *squeezed* if the other constraints (including the other contingent links) imply a strictly tighter lower bound or upper bound for the link. An STNU is *pseudo-controllable* if it is consistent and none of the contingent links are squeezed.

If a network is pseudo-controllable then all the edges arising from contingent links are shortest paths. Thus, the contingent links survive unchanged in the AllPairs shortest-path graph (abbreviated as the AllPairs graph). Note that pseudo-controllability can be determined in polynomial time by computing the AllPairs graph.

It is easy to see that every Weakly Controllable network is pseudo-controllable since a squeezed contingent link would imply a projection that is not consistent. However, the converse is not true in general.

Even for a STNU that was originally pseudo-controllable, it is possible for a contingent link to be squeezed during execution (which may be viewed as augmenting the network with additional constraints). In this paper, we will make use of results from [Morris and Muscettola, 2000]. These guarantee that a contingent link cannot be squeezed during execution under certain circumstances. Essentially, upper bounds can only be squeezed by propagations through links with

non-negative upper bounds, and lower bounds can only be squeezed by propagations through links with positive lower bounds. Even in these cases, squeezing cannot occur if the relevant bound is *dominated* by that of the contingent link, which essentially means the bound at issue is redundant. If the dominance relations are such that no contingent link can be squeezed, then the network is *safe*. A safe network can be executed like an ordinary STN, and thus is Dynamically Controllable.

3 Triangular Reductions

A starting point for resolving the issue of Dynamic Controllability is to consider *triangular* STNU networks, i.e., networks involving three timepoints and including a contingent link, as shown in figure 1. Here AC is a contingent link with bounds $[x, y]$, while AB and BC are requirement links with bounds $[p, q]$ and $[u, v]$ respectively. This notation for contingent and requirement links will be used in subsequent diagrams. The contingent link AC is called the *focus* of the triangle. We will also assume that the triangular networks we consider are pseudo-controllable and have been placed in AllPairs form, so every edge is a shortest path. It follows that $[u, v] \subseteq [x - q, y - p]$, which implies $[p, q] \supseteq [y - v, x - u]$.

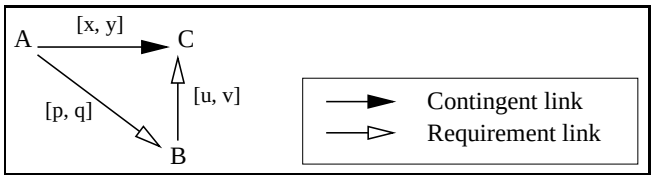


Figure 1: Triangular Network

We will derive a number of results concerning additional tightenings or *reductions* of the bounds that must be obeyed by any schedule resulting from a dynamic strategy (i.e., any $S(p)$ for any projection p, in the notation of the previous section). These will vary according to cases involving the signs of the $[u, v]$ bounds.

1. First suppose that $v < 0$. We call this the *Follow* case, since the lower bound of CB (i.e., BC reversed) is $-v$ and hence B follows C. Then the network is Dynamically Controllable since C has already been observed at the time B is executed. In fact, it may be executed like an ordinary STN since any propagation will go from C to B and not vice versa. Thus, the network is safe and no tightening is needed.

2. Next consider the case where $u \geq 0$. We call this the *Precede* case, since B occurs before or simultaneously with C. Then no information about C is available to B. In this case, we claim that AB can be tightened to $[y - v, x - u]$. Suppose there is a projection p that a dynamic strategy maps to a schedule T with $T_B - T_A < y - v$. Since C is not in $T_{\prec B}$ or $T_{\prec A}$, T_B and T_A cannot depend on AC. Therefore T_A and T_B are unchanged if the projection is mutated to a projection p' where AC equals y. But then we have BC $= T_C - T_B = (T_C - T_A) - (T_B - T_A) > y - (y - v) = v$, so the BC constraint will be violated. Thus, $T_B - T_A \geq y - v$. A similar argument shows $T_B - T_A \leq x - u$. After the tightening of AB to $[y - v, x - u]$ (or equivalently BA to $[u - x, v - y]$), the BC bounds are dominated (redundant) since $[u - x, v - y] + [x, y] = [u, v]$. Thus, the network is safe provided it is still pseudo-controllable.

3. The most interesting case occurs when $u < 0$ and $v \geq 0$, which we call the *Unordered* case, since B may or may not follow C. However, suppose B does not follow C and $T_B - T_A < y - v$. As in the previous case, there is then a projection where the BC constraint is violated. We conclude that, for a dynamic strategy, B cannot be executed at any time before $y - v$ after A if C has not already occurred. This is a conditional constraint on AB, depending on the time of occurrence of C. It may also be viewed as a ternary constraint on A,B, and C, which we call a *wait* since B must wait until either C occurs or the wait expires at $y - v$ after A.

First, there is one subcase for which the conditional constraint turns out to be unconditional, which is when $y - v \leq x$. Then C cannot occur before the wait expires, so we can simply raise the lower bound of AB to $y - v$. We will call this the *unconditional Unordered* reduction.

In the truly conditional subcase where $x < y - v$, an obvious idea is to branch on the conditional and consider separately two possibilities. First if it turns out that AC $< y - v$ (in which case C occurs first and B follows), the network is safe if pseudo-controllable as in the *Follow* case. Otherwise if AC $\geq y - v$ then AB $\geq y - v$ also, which gives BA an upper bound of $v - y$. Thus, the BC upper bound of v is dominated (redundant). Since the lower bound u is negative, the network is safe if pseudo-controllable [Morris and Muscettola, 2000]. Observe that in either case B occurs later than x after A, so without branching we can raise the lower bound of AB to x. We will call this the *general Unordered* reduction.

We see above that assuming a dynamic strategy may lead to a tightening of the constraint bounds. If the tightening produces a violation of pseudo-controllability, then the original network was not Dynamically Controllable. On the other hand, if the network remains pseudo-controllable after the tightening (in the general Unordered case we must verify this for both possibilities), then the triangular network is safe and thus Dynamically Controllable [Morris and Muscettola, 2000]. Thus, the tightenings give a procedure for determining Dynamic Controllability of triangular networks.

4 Local vs Global Dynamic Controllability

To test a general STNU network for Dynamic Controllability, we can construct the AllPairs graph, which may be regarded as a combination of triangular subnetworks. Triangles that involve a contingent link may be viewed as instances of figure 1. If a triangle contains two contingent links,[2] then we consider it twice, with each contingent link in turn playing the role of focus, and the other being treated as a requirement link. Any tightening propagates to neighbour triangles until quiescence of the network is reached. The only problem arises with Unordered cases: if we branch on the conditionals as discussed in the previous section, we end up with a combinatorial search, which we prefer to avoid. Instead we use

[2] Triangles with three contingent links cannot occur, since we have excluded coincident finishing points.

only the two non-branching Unordered reductions discussed earlier, so the resulting iterative algorithm is deterministic and polynomial. (But the network is then not necessarily safe.)

This propagation algorithm with no search may be viewed as a *local* Dynamic Controllability checking procedure. Since it applies to triangles, this is similar to a path-consistency algorithm in a classical constraint network such as a STN. Hence, we call this local property *3-Dynamic Controllability* and call the resulting algorithm 3DC. As with any local filtering algorithm, the process is sound: if it fails, then at least one triangle is not Dynamically Controllable and therefore the whole network is not.

However, it is incomplete as shown by the example in figure 2. We invite the reader to verify that the triangles are all quiescent under the deterministic reductions considered above; therefore the network is stable under 3DC.

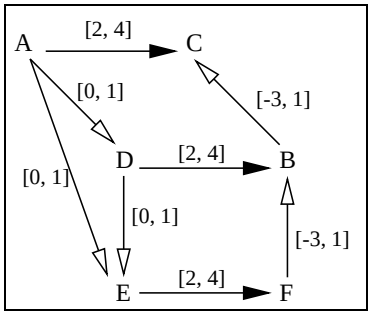


Figure 2: Quiescent non-DC Network

Now consider the subnetwork ACDB. It is not difficult to see that a dynamic strategy requires AD = 1. Similarly, DE must be 1. But that causes a violation of the AE link. Hence the network is not Dynamically Controllable.[3] This example also shows that 3DC does not compute the *minimal* network, i.e., the network in which values not belonging to any dynamic strategy have been removed (for instance here AD would be tightened to [1,1]). A checking algorithm should ideally produce this minimality property, which is desirable for execution purposes. Nevertheless, 3DC is an efficient technique to rule out a wide variety of networks.

5 Regression of Waits

The incompleteness of 3DC might suggest we should reconsider a combinatorial search. However, we have not exhausted the possibilities of obtaining deterministic reductions from the Unordered cases. If the ternary constraint corresponding to the Unordered wait is used directly, then no branching is necessary. Moreover, this ternary constraint can be treated somewhat like a binary constraint. Suppose we have a wait condition that requires B to wait for C until time t after A. We will indicate that by placing a $<\mathrm{C}, t>$ annotation on the AB link. Note that if it is impossible for C to occur before t (for example if the lower bound of AC is greater than t), then the $<\mathrm{C}, t>$ wait becomes a true lower bound of t on AB. This corresponds to the unconditional Unordered reduction discussed earlier.

Now consider figure 2 again. The triangle ABC is an Unordered case, so AB receives a $<\mathrm{C}, 3>$ wait. This is not unconditional since the lower bound of AC is 2. Now consider triangle ADB with this new label on AB. Suppose C has not occurred yet and D is executed before 1 time unit after A. In the projection where DB equals 2, B will then occur before 3 time units after A. If C still has not occurred by then, the wait on AB will be violated. In other words, the wait on AB can be *regressed* through DB to obtain a derived wait on AD, still relative to C: $<\mathrm{C}, 1>$. This, happily, is an unconditional wait since C cannot occur before time 2, which produces a lower bound of 1 on AD and leads to a resolution of the example. One can notice as well that we achieve the hoped-for minimal network. That leads us to the following result.

Lemma 1 (Regression) *Suppose a link AB has a wait* $<\mathrm{C}, t>$ *, where* t *is less than or equal to the upper bound of AC. Then (in a schedule resulting from a dynamic strategy):*

(i) If there is any link DB (including AB itself) with upper bound w*, then we can deduce a wait* $<\mathrm{C}, t-w>$ *on AD.*

(ii) If $t \geq 0$ *and if there is a contingent link DB with lower bound* z*, where* $B \neq C$*, then we can deduce a wait* $<\mathrm{C}, t-z>$ *on AD.*

Proof: Consider (i) first. Suppose D occurs before $t-w$ after A and C has not occurred yet. From the upper bound w on DB, it follows that B must occur before $w+t-w=t$. But this violates the wait on AB in the projection where C occurs at its upper bound (which is $\geq t$). We conclude that D cannot occur before $t-w$ after A unless C has already occurred.

Now consider (ii). If $t \geq 0$, then B must be later than A. Suppose D occurs before $t-z$ after A and C has not occurred yet. Then neither A nor D can depend on the outcomes of AC or DB. Thus, we can consider a mutated projection where DB finishes at z and AC finishes at its upper bound. This leads to a violation of the AB wait. □

Note that (i) and (ii) are both applicable to contingent links but (ii) gives a more restrictive (longer) wait.

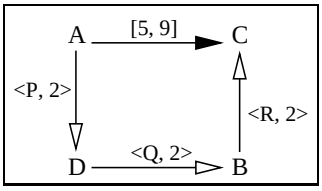


Figure 3: Regression Example

Iterated regression amounts to a new type of propagation, where waits are spread to other links. The propagated waits can be examined for any unordered reductions, which place additional ordinary constraints throughout the network. For example, consider figure 3. Intuitively, we can see this is not Dynamically Controllable because the waits in the worst case will cause an incursion on the AC lower bound (assuming the upper bounds of the AP,DQ,BR contingent links are all at least 2). First we can regress the $<\mathrm{R}, 2>$ wait through

[3] Note that this example is Weakly Controllable, as can be seen by considering the worst case projection for each cycle.

```
procedure DynamicallyControllable? (network W)
1. Compute the All-Pairs graph for W.
   If W is not pseudo-controllable then return false.
2. Select any triangle such that v is non-negative.
   Introduce any tightenings required by the Precede case
   and any waits required by the Unordered case.
3. Do all possible regressions of waits, while converting
   unconditional waits to lower bounds.  Also introduce
   lower bounds as provided by the general reduction.
4. If steps 2 and 3 do not produce any new (or tighter)
   constraints, then return true, otherwise go to 1.
```

Figure 4: DC Checking Algorithm

AC, which gives a wait of $<\mathrm{R}, -3>$ on BA. This gives rise to (unconditional case) a lower bound of -3 on BA, which is equivalent to an upper bound of $+3$ on AB. Now we can regress the $<\mathrm{Q}, 2>$ wait on DB through AB, which gives a $<\mathrm{Q}, -1>$ on DA, giving rise to a $+1$ upper bound on AD. Finally, we regress the $<\mathrm{P}, 2>$ wait on AD through AD *itself*, which gives a $<\mathrm{P}, 1>$ wait on AA. Now the general reduction ensures a positive lower bound on AA, which is a direct inconsistency. Thus, we have reduced the lack of Dynamic Controllability to a violation of consistency.

6 Dynamic Checking and Execution

We are now ready to introduce the algorithm for determining Dynamic Controllability, summarized in figure 4. It is just an enhancement of 3DC with wait regressions and hence is still a local algorithm, but now we can show it is complete.

We prove completeness by presenting a dynamic execution algorithm and showing that it is viable if the DC checking algorithm reports success. For simplicity, we will assume the execution takes place in the AllPairs graph of the tightened network, although performance could be improved by transforming it to a minimum dispatchable graph as in [Muscettola *et al.*, 1998a]. The execution is essentially the same as for an ordinary STN except for adding a requirement to respect the waits. For this purpose, we only consider waits $<\mathrm{C}, t>$ where t satisfies $l(C) < t \leq u(C)$. Note that waits with $t \leq l(C)$ are converted to lower bounds, while waits with $t > u(C)$ are equivalent to those with $t = u(C)$. (Since $l(C) > 0$ by definition, the waits enforced by the algorithm are all positive.)

The execution algorithm is shown in figure 5. We assume there is some start timepoint that is constrained to be before every other timepoint. (If necessary, one can be added.) In step 2, a timepoint is *live* if the current time is within the timepoint's bounds. It is *enabled* if all timepoints required to be executed before it (by links with positive lower bounds) have already been executed [Muscettola *et al.*, 1998a].

It is clear that this algorithm provides a strategy where the decisions depend only on the past. The issue is whether any constraints are violated. Properties of STNs guarantee that they can be executed incrementally [Muscettola *et al.*, 1998a]. Therefore, only the special features introduced for STNUs need be considered. The following are the possible ways in which the execution might fail.

- A deadlock might occur where a wait lasts forever.
- A wait might be forcibly aborted.

```
procedure Execute (network W)
  0. Perform initial propagation from the start timepoint.
  1. Immediately execute any executable timepoints
     that have reached their upper bounds.
  2. Arbitrarily pick an executable timepoint TP that
     is live and enabled and not yet executed, and whose
     waits, if any, have all been satisfied.
  3. Execute TP.  Halt if network execution is complete.
     Otherwise, propagate the effect of the execution.
  4. Advance current time, propagating the effect of any
     contingent timepoints that occur, until an
     executable timepoint becomes eligible for
     execution under 1 or 2.
  5. Go to 1.
```

Figure 5: DC Network Execution

- A propagation might squeeze a contingent link.

An example of a potential deadlock is when AC and DB are contingent links with a $<\mathrm{C}, t1>$ wait on AD and a $<\mathrm{B}, t2>$ wait on DA. More generally, a deadlock requires a cycle of links, each of which is labelled with a wait or a positive lower bound. However, the waits $<\mathrm{C}, t>$ enforced by the execution algorithm satisfy $l(C) < t \leq u(C)$ (see above). These imply a positive lower bound of $l(C)$ by the general reduction. Thus, we would have a cycle where each link has a positive lower bound. This corresponds to an inconsistency in the network that would be detected by step 1 of the DC checking algorithm. The other possibilities are considered in the following lemmas.

Lemma 2 *Suppose a network has successfully passed the DC checking algorithm. Then the first failure that occurs during the DC execution cannot be an aborted wait.*

Proof: Suppose the first failure is an aborted wait, and the earliest time this occurs involves a wait $<\mathrm{C}, t>$ on a link AB. As pointed out above, this wait must be positive, so the link AB will have a positive lower bound. First we note that B obviously cannot be the start timepoint.

There are now two cases to consider. In the first case, the wait is aborted because step 1 required an immediate execution of B. Consider the timepoint D (possibly the start) whose execution initiated the propagation that produced the upper bound of B. Note the regression of $<\mathrm{C}, t>$ through DB produces a wait of $<\mathrm{C}, t - u(\mathrm{DB})>$ on AD. If $t - u(\mathrm{DB}) \leq l(\mathrm{AC})$, the checking algorithm places it as an unconditional lower bound on AD. Otherwise, $<\mathrm{C}, t - u(\mathrm{DB})>$ is an earlier wait that is enforced by the execution algorithm. In either case, D does not occur until $t - u(\mathrm{DB})$ after A. Suppose b and d are the upper bounds of B and D, respectively, and a is the time of execution of A. Then $(d - a) \geq (t - u(\mathrm{DB}))$. Since $b = d + u(\mathrm{DB})$, it follows that $(b - a) \geq t$. This contradicts the assumption that the wait was terminated.

The second case involves the possibility that B is a contingent timepoint. (Thus, the execution is not controlled by the agent). Suppose EB is a contingent link with bounds $[x, y]$. Again we can regress the wait through EB getting $<\mathrm{C}, t-x>$ on AE. Since E is earlier than B, the latter wait must be satisfied. Thus, the duration of AE is greater than $t - x$. Since x is the minimum duration of EB, it follows that AB is greater than $t - x + x = t$, i.e., the wait is satisfied after all. □

Lemma 3 *Suppose a network has successfully passed the DC checking algorithm. Then the first failure that occurs during DC execution cannot be a squeezing of a contingent link.*

Proof: Suppose the earliest failure is the squeezing of a contingent link AC that has bounds $[x, y]$. This must occur during a propagation that either raises the lower bound of AC or lowers the upper bound. However, the triangular reductions ensure that AC dominates adjacent links with finishing point C, except for the case of links BC with negative lower bound u and non-negative upper-bound v such that $y - v > x$ (the conditional Unordered case). This means the only possibility for a squeezing is an upper-bound propagation along some such BC. However, the existence of such a BC would cause the checking algorithm to place a $<\mathrm{C}, y - v>$ wait on AB. If C occurs before B then there is no propagation from B to C. Otherwise, the enforcement of the wait by the execution algorithm ensures that B is not executed before $y - v$ after A. Thus, the upper bound propagated along BC will be $T_B + v \geq (T_A + y - v) + v = T_A + y$, so AC is not squeezed. □

Theorem 1 *Dynamic Controllability can be determined in deterministic polynomial time.*

Proof: Lemmas 2 and 3 demonstrate that the execution algorithm successfully executes networks that are verified by the checking algorithm. Thus, the Dynamic Controllability checking algorithm is complete. As noted earlier, the algorithm is also sound since the added constraints were derived from the assumption of Dynamic Controllability.

The individual tightenings are clearly polynomial, and convergence is assured because the domains of the constraints are strictly reduced by the tightenings. The only issue is how long the convergence takes. A crude upper bound can be obtained by assuming a fixed level of precision δ and finite bounds (say between $\pm\beta$) on all links. If there are η links, then after at most $2\eta\beta/\delta$ reductions, some domain would become empty. This bound grows polynomially with the size of the problem. □

It is worth pointing out that the execution algorithm presented here preserves maximum flexibility, since the additional tightenings and waits were all required by Dynamic Controllability. (In contrast to the approach, for example, of adding *waypoints* [Morris and Muscettola, 1999], which surrenders some flexibility.) Another interesting point is that the execution algorithm allows the selection of any execution time within prescribed limits, without impairing the success of the dynamic strategy. Therefore the incremental application of the DC propagation ensures that the values remaining in the domains are consistent with the dynamic strategy. In other words, the DC checking algorithm produces the minimal network in the sense described earlier.

7 Conclusions

Dynamic Controllability is polynomial! That is certainly the main contribution of this paper, since this property, needed in many real-world applications such as planning and scheduling, was expected to be much harder.

Moreover, the proposed method is directly applicable to the STNU (as opposed to a previous technique that needed a translation into a finite-state automaton model [Vidal, 2000]), and is inspired by classical constraint satisfaction techniques. We have presented a local Dynamic Controllability algorithm based on triangle reductions, and have shown that non-binary constraints that were inherent in the problem give rise to binary constraints through a regression process. We have also proven this local controllability algorithm is complete with respect to Dynamic Controllability of the global network.

We believe our contribution will be valuable in the design of new constraint programming packages that handle temporal uncertainty and will help pave the way to effective real-time execution systems that incorporate such uncertainties.

References

[Cormen *et al.*, 1990] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT press, Cambridge, MA, 1990.

[Dechter *et al.*, 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, May 1991.

[Laborie and Ghallab, 1995] P. Laborie and M. Ghallab. Planning with sharable constraints. In *Proceedings of the 14th International Joint Conference on A.I. (IJCAI-95)*, Montreal (Canada), 1995.

[Morris and Muscettola, 1999] P. Morris and N. Muscettola. Managing temporal uncertainty through waypoint controllability. In *Proc. of Sixteenth Int. Joint Conf. on Artificial Intelligence (IJCAI-99)*, 1999.

[Morris and Muscettola, 2000] P. Morris and N. Muscettola. Execution of temporal plans with uncertainty. In *Proc. of Seventeenth Nat. Conf. on Artificial Intelligence (AAAI-00)*, 2000.

[Muscettola *et al.*, 1998a] N. Muscettola, P. Morris, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *Proc. of Sixth Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, 1998.

[Muscettola *et al.*, 1998b] N. Muscettola, P.P. Nayak, B. Pell, and B.C. Williams. Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.

[Vidal and Fargier, 1999] T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental & Theoretical Artificial Intelligence*, 11:23–45, 1999.

[Vidal and Ghallab, 1996] T. Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc. of 12th European Conference on Artificial Intelligence (ECAI-96)*, pages 48–52, 1996.

[Vidal, 2000] T. Vidal. Controllability characterization and checking in contingent temporal constraint networks. In *Proc. of Seventh Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'2000)*, 2000.

PLANNING

COMPLEXITY OF PLANNING

Complexity of Probabilistic Planning under Average Rewards

Jussi Rintanen
Albert-Ludwigs-Universität Freiburg, Institut für Informatik
Georges-Köhler-Allee, 79110 Freiburg im Breisgau
Germany

Abstract

A general and expressive model of sequential decision making under uncertainty is provided by the Markov decision processes (MDPs) framework. Complex applications with very large state spaces are best modelled implicitly (instead of explicitly by enumerating the state space), for example as precondition-effect operators, the representation used in AI planning. This kind of representations are very powerful, and they make the construction of policies/plans computationally very complex. In many applications, average rewards over unit time is the relevant rationality criterion, as opposed to the more widely used discounted reward criterion, and for providing a solid basis for the development of efficient planning algorithms, the computational complexity of the decision problems related to average rewards has to be analyzed. We investigate the complexity of the policy/plan existence problem for MDPs under the average reward criterion, with MDPs represented in terms of conditional probabilistic precondition-effect operators. We consider policies with and without memory, and with different degrees of sensing/observability. The unrestricted policy existence problem for the partially observable cases was earlier known to be undecidable. The results place the remaining computational problems to the complexity classes EXP and NEXP (deterministic and nondeterministic exponential time.)

1 Introduction

Markov decision processes (MDPs) formalize decision making in controlling a nondeterministic transition system so that given utility criteria are satisfied. An MDP consists of a set of states, a set of actions, transition probabilities between the states for every action, and rewards/costs associated with the states and actions. A policy determines for every state which action is to be taken. Policies are valued according to the rewards obtained or costs incurred.

Applications for the kind of planning problems addressed by this work include agent-based systems, including Internet agents and autonomous robots, that have to repeatedly perform actions over an extended period of time in the presence of uncertainty about the environment, and the actions have to – in order to produce the desired results – follow a high-level strategy, expressed as a plan.

Classical deterministic AI planning is the problem of finding a path between the initial state and a goal state. For explicit representations of state spaces as graphs this problem is solvable in polynomial time, and for implicit representations of state spaces in terms of state variables and precondition-effect operators, which sometimes allows an exponentially more concise representation of the problem instances, the path existence problem is PSPACE-complete [Bylander, 1994]. This result is closely related to the PSPACE-completeness of the existence of paths in graphs represented as circuits [Papadimitriou and Yannakakis, 1986; Lozano and Balcázar, 1990]. Similarly, the complexity of most other graph problems increases when a compact graph representation is used [Galperin and Widgerson, 1983; Papadimitriou and Yannakakis, 1986; Lozano and Balcázar, 1990; Balcázar, 1996; Feigenbaum *et al.*, 1999].

MDPs and POMDPs can be viewed as an extension of the graph-based deterministic planning framework with probabilities: an action determines a successor state only with a certain probability. The objective is to visit valuable states with a high probability. A policy (a plan) determines which actions are chosen given the current state (or a set of possible current states, possibly together with some information on the possible predecessor states.) For explicitly represented MDPs, policy evaluation under average rewards reduces to the solution of sets of linear equations. Sets of linear equations can be solved in polynomial time. Similarly, policies for many types of explicitly represented MDPs can be constructed in polynomial time by linear programming. Papadimitriou and Tsitsiklis [1987] have shown that policy existence for explicitly represented MDPs is P-complete. Madani et al. [1999] have shown the undecidability of policy existence for UMDPs and POMDPs with all main rationality criteria.

Like in classical AI planning, MDPs/POMDPs can be concisely represented in terms of state variables and precondition-effect operators. The important question in this setting is, what is the impact of concise representations on the complexity of these problems. In related work [Mundhenk *et al.*, 2000; Littman, 1997; Littman *et al.*, 1998], this

question has been investigated in the context of finite horizons. Not surprisingly, there is in general an exponential increase in problem complexity, for example from deterministic polynomial time to deterministic exponential time. The undecidability results for explicitly represented UMDPs and POMDPs directly implies the undecidability of the respective decision problems with concise representations.

In the present work we investigate the complexity of the policy existence problems for MDPs and POMDPs under expected average rewards over an infinite horizon. For many practically interesting problems from AI – for example autonomous robots, Internet agents, and so on – the number of actions taken is high over a period time and lengths of sequences of actions are unbounded. Therefore there is typically no reasonable interpretation for discounts nor reasonable upper bounds on the horizon length, and average reward is the most relevant criterion. A main reason for the restriction to bounded horizons and discounted rewards in earlier work is that the structure of the algorithms in these cases is considerably simpler, because considerations on MDP structural properties, like recurrence and periodicity, can be avoided. Also, for many applications of MDPs that represent phenomena over extended periods of times (years and decades), for example in economics, the discounts simply represent the unimportance of events in the distant future, for example transcending the lifetimes of the decision makers. Boutilier and Puterman [1995] have advocated the use of average-reward criteria in AI.

The structure of the paper is as follows. Section 2 describes the planning problems addressed by the paper, and Section 3 introduces the required complexity-theoretic concepts. In Section 4 we present the results on the complexity of testing the existence of policies for MDPs under average reward criteria, and Section 5 concludes the paper.

2 Probabilistic Planning with Average Rewards

The computational problem we consider is the existence of policies for MDPs (fully observable), UMDPs (unobservable) and POMDPs (partially observable, generalizing both MDPs and UMDPs) that are represented concisely; that is, states are represented as valuations on state variables and transition probabilities are given as operators that affect the state variables. The policies we consider may have an arbitrary size, but we also briefly discuss complexity reduction obtained by restricting to polynomial size policies.

As pointed out in Example 2.1, the average reward of a policy sometimes cannot unambiguously be stated as a single real number. The computational problem that we consider is the following. Is the expected average reward greater than (or equal to) some constant c. This amounts to identifying the recurrent classes determined by the policy, and then taking a weighted average of the rewards according to the probabilities with which the classes are reached.

2.1 Definition of MDPs

MDPs can be represented explicitly as a set of states and a transition relation that assigns a probability to transitions between states under all actions. We restrict to finite MDPs and formally define them as follows.

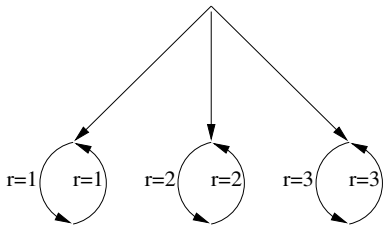


Figure 1: A multichain MDP

Definition 1 *A (partially observable) Markov decision process is a tuple $\langle S, A, T, I, R, B\rangle$ where S is a set of states, A is a set of actions, $T : A \times S \times S \to \mathcal{R}$ gives the transition probability between states (the transition probabilities from a given state must sum to 1.0) for every action, $I \in S$ is the initial state, $R : S \times A \to \mathcal{R}$ associates a reward for applying an action in a given state, and $B \subseteq 2^S$ is a partition of S to classes of states that cannot be distinguished.*

Policies map the current and past observations to actions.

Definition 2 *A policy $P : (2^S)+ \to A$ is a mapping from a sequence of observations to an action. A stationary policy $P : 2^S \to A$ is a mapping from the current observation to an action.*

For UMDPs the observation is always the same (S), for MDPs the observations are singleton sets of states (they determine the current state uniquely), and for POMDPs the observations are members of a partition of S to sets of states that are indistinguishable from each other (the limiting cases are S and singletons $\{s_i\}$ for $s_i \in S$: POMDPs are a generalization of both UMDPs and MDPs.)

The expected average reward of a policy is the limit

$$\lim_{N\to\infty} \frac{1}{N} \sum_{t\geq 0}^{N} \sum_{a\in A, s\in S} r_{a,s} p_{a,s,t}$$

where $r_{a,s}$ is the reward of taking action a in state s and $p_{a,s,t}$ is the probability of taking action a at time point t in state s. There are policies for which the limit does not exist [Puterman, 1994, Example 8.1.1], but when the policy execution has only a finite number of internal states (like stationary policies have), the limit always exists.

The recurrent classes of a POMDP under a given policy are sets of states that will always stay reachable from each other with probability 1.

Example 2.1 Consider a policy that induces the structure shown in Figure 1 on a POMDP. The three recurrent classes each consist of two states. The initial state does not belong to any of the recurrent classes. The state reached by the first transition determines the average reward, which will be 1, 2 or 3, depending on the recurrent class. ■

2.2 Concise Representation of MDPs

An exponentially more concise representation of MDPs is based on state variables. Each state is an assignment of truth-values to the state variables, and transitions between states are expressed as changes in the values of the state variables.

In AI planning, problems are represented by so-called STRIPS operators that are pairs of sets of literals, the *precondition* and the *effects*. For probabilistic planning, this can be extended to probabilistic STRIPS operators (PSOs) (see [Boutilier *et al.*, 1999] for references and a discussion of PSOs and other concise representations of transition systems with probabilities.) In this paper, we further extend PSOs to what we call extended PSOs (EPSOs). An EPSO can represent an exponential number of PSOs, and we use them because they are closely related to operators with conditional effects commonly used in AI planning. Apart from generating the state space of a POMDP, the operators can conveniently be taken to be the actions of the POMDP.

Definition 3 (Extended probabilistic STRIPS operators) *An* extended probabilistic STRIPS operator *is a pair $\langle C, E\rangle$, where C is a Boolean circuit and E consists of pairs $\langle c, f\rangle$, where c is a Boolean circuit and f is a set of pairs $\langle p, e\rangle$, where $p \in]0..1]$ is a real number and e is a set of literals such that for every f the sum of the probabilities p is 1.0.*

For all $\langle c_1, f_1\rangle \in E$ and $\langle c_2, f_2\rangle \in E$, if e_1 contradicts e_2 for some $\langle p_1, e_1\rangle \in f_1$ and $\langle p_2, e_2\rangle \in f_2$, then c_1 must contradict c_2.

This definition generalizes PSOs by not requiring that the cs of members $\langle c, F\rangle$ of E are logically disjoint and their disjunction is a tautology. Hence in EPSOs the effects may take place independently of each other. Some of the hardness proofs given later would be more complicated – assuming that they are possible – if we had to restrict to PSOs.

The application of an EPSO is defined iff the precondition C is true in the current state. Then the following takes place for every $\langle c, f\rangle \in E$. If c is true, one of the $\langle p, e\rangle \in f$ is chosen, each with probability p, and literals e are changed to true.

Example 2.2 Let

$$\begin{aligned} o = \langle \top, \{ \quad & \langle p_1, \{\langle 1.0, \{\neg p_1\}\rangle\}\rangle, \\ & \langle \neg p_1, \{\langle 1.0, \{p_1\}\rangle\}\rangle, \\ & \ldots, \\ & \langle p_n, \{\langle 1.0, \{\neg p_n\}\rangle\}\rangle, \\ & \langle \neg p_n, \{\langle 1.0, \{p_n\}\rangle\}\rangle\}\rangle. \end{aligned}$$

Now o is an EPSO but not a PSO because the antecedents $p_1, \neg p_1, p_2, \neg p_2, \ldots$ are not logically disjoint. A set of PSOs corresponding to o has cardinality exponential on n. ■

Rewards are associated with actions and states. When an action is taken in an appropriate state, a reward is obtained. For every action, the set of states that yields a given reward is represented by a Boolean circuit.

Definition 4 (Concise POMDP) A concise POMDP *over a set P of state variables is a tuple $\langle I, O, r, B\rangle$ where I is an initial state (assignment $P \to \{\top, \bot\}$), O is a set of EPSOs representing the actions, and $r : O \to C \times \mathcal{R}$ associates a Boolean circuit and a real-valued reward with every action, and $B \subseteq P$ is the set of observable state variables.*

Having a set of variables observable – instead of arbitrary circuits/formulae – is not a restriction. Assume that the values of a circuit are observable (but the individual input gates are not.) We could make every EPSO evaluate the value of this circuit and set the value of an observable variable accordingly.

Definition 5 (Concise MDP) A concise MDP *is a concise POMDP with $B = P$.*

Definition 6 (Concise UMDP) A concise UMDP *is a concise POMDP with $B = \emptyset$.*

2.3 Concise Representation of Policies

We consider history/time-dependent and stationary policies, and do not make a distinction between history and time-dependent ones. Traditionally explicit (or flat) representations of policies have been considered in research on MDPs/POMDPs: each state or belief state is explicitly associated with an action. In our setting, in which the number of states can be very high, also policies have to be represented concisely. Like with concise representations of POMDPs, there is no direct connection between the size of a concisely represented policy and the number of states of the POMDP.

A concise policy could, in the most general case, be a program in a Turing-equivalent programming language. This would, however, make many questions concerning policies undecidable. Therefore less powerful representations of policies have to be used. A concise policy determines the current action based on the current observation and the past history. We divide this to two subtasks: keeping track of the history (maintaining the internal state of the execution of the policy), and mapping the current observation and the internal state of the execution of the policy to an action. The computation needed in applying one operator is essentially a state transition of a concisely represented finite automaton.

A sensible restriction would be that computation of the action to be taken and the new internal state of the policy execution is polynomial time. An obvious choice is the use of Boolean circuits, because the circuit value problem is P-complete (one of the hardest problems in P.) Work on algorithms for concise POMDPs and AI planning have not used this general a policy representation, but for our purposes this seems like a well-founded choice. Related definitions of policies as finite-state controllers have been proposed earlier [Hansen, 1998; Meuleau *et al.*, 1999; Lusena *et al.*, 1999].

Definition 7 (Concise policy) A concise policy *for a concise POMDP $M = \langle I, O, r, B\rangle$ is a tuple $\langle T, C, v\rangle$ where T is a Boolean circuit with $|B| + p$ input gates and p output gates, C is a Boolean circuit with $|B| + p$ input gates and $\lceil \log_2 |O| \rceil$ output gates, and v is a mapping from $\{1, \ldots, p\}$ to $\{\bot, \top\}$.*

The circuit T encodes the change in execution state in terms of the preceding state and the observable state variables

	stationary	history-dependent
UMDP	PSPACE-hard, in EXP (L8,9)	undecidable
MDP	EXP (T11)	EXP (C12)
POMDP	NEXP (T13)	undecidable

Table 1: Complexity of policy existence, with references to the lemmata, theorems, and corollaries.

B. The circuit C encodes the action to be taken, and v gives the initial state of the execution. The integer p is the number of bits for the representation of the internal state of the execution. When $p = 0$ we have a stationary policy.

The complexity results do not deeply rely on the exact formal definition of policies. An important property of the definition is that one step of policy execution can be performed in polynomial time.

3 Complexity Classes

The complexity class P consists of decision problems that are solvable in polynomial time by a deterministic Turing machine. NP is the class of decision problems that are solvable in polynomial time by a nondeterministic Turing machine. $C_1^{C_2}$ denotes the class of problems that is defined like the class C_1 except that Turing machines with an oracle for a problem in C_2 are used instead of ordinary Turing machines. Turing machines with an oracle for a problem B may perform tests for membership in B for free. A problem L is *C-hard* if all problems in the class C are polynomial time *many-one reducible* to it; that is, for all problems $L' \in C$ there is a function $f_{L'}$ computable in polynomial time on the size of its input and $f_{L'}(x) \in L$ if and only if $x \in L'$. A problem is *C-complete* if it belongs to the class C and is C-hard.

PSPACE is the class of decision problems solvable in deterministic polynomial space. EXP is the class of decision problems solvable in deterministic exponential time ($O(2^{p(n)})$ where $p(n)$ is a polynomial.) NEXP is the class of decision problems solvable in nondeterministic exponential time. A more detailed description of the complexity classes can be found in standard textbooks on complexity theory, for example by Balcazár et al. [1995].

4 Complexity Results

Table 1 summarizes the complexity of determining the existence of stationary and history-dependent policies for UMDPs, MDPs and POMDPs. In the average rewards case the existence of history-dependent and stationary policies for MDPs coincide. The undecidability of UMDP and POMDP policy existence with history-dependent policies of unrestricted size was shown by Madani et al. [1999]. The result is based on the emptiness problem of probabilistic finite automata [Paz, 1971; Condon and Lipton, 1989] that is closely related to the unobservable plan existence problem.

The results do not completely determine the complexity of the UMDP stationary policy existence problem, but as the stationary UMDP policies repeatedly apply one single operator, the problem does not seem to have the power of EXP. It is also not trivial to show membership in PSPACE.

The rest of the paper formally states the results summarized in Table 1 and gives their proof outlines.

Lemma 8 *Existence of a policy with average reward $r \geq c$ for UMDPs, MDPs and POMDPs with only one action is PSPACE-hard.*

Proof: It is straightforward to reduce any decision problem in PSPACE to the problem. This is by constructing a concise UMDP/MDP/POMDP with only one action that simulates a polynomial-space deterministic Turing machine for the problem in question.

There are state variables for representing the input, the working tape, and the state of the Turing machine. The EPSO that represents the only action is constructed to follow the state transitions of the Turing machine. The size of the EPSO is polynomial on the size of the input. When the machine accepts, it is restarted. A reward $r \geq c$ is obtained as long as the machine has not rejected. If the machine rejects, all future rewards will be 0. Therefore, if the Turing machine accepts the average reward is r, and otherwise it is 0. □

There are two straightforward complexity upper bounds respectively for polynomial size and stationary policies. Polynomial size policies can maintain at most an exponential number of different representations of the past history, and hence an explicit representation of the product of the POMDP and the possible histories has only exponential size, just like the POMDP state space alone. Stationary policies, on the other hand, do not maintain a history at all, and they therefore encode at most an exponential number of different decision situations, one for each (observable) state of a (PO)MDP. For the unrestricted size partially observable non-stationary case there is no similar exponential upper bound, and the problem is not decidable.

Lemma 9 *Let c be a real number. Testing the existence of a poly-size MDP/UMDP/POMDP policy with average reward $r \geq c$ is in EXP.*

Proof: This computation has complexity $\mathrm{NP}^{EXP} = EXP$, that corresponds to guessing a polynomial size policy (NP) followed by the evaluation of the policy by an EXP oracle. Policy evaluation proceeds as follows. Produce the explicit representation of the product of the POMDP and the state space of the policy. They respectively have sizes $2^{p_1(x)}$ and $2^{p_2(x)}$ for some polynomials $p_1(x)$ and $p_2(x)$. The product, which is a Markov chain and represents the states the POMDP and the policy execution can be in, is of exponential size $2^{p_1(x)+p_2(x)}$.

From the explicit representation of the state space one can identify the recurrent classes in polynomial time, for example by Tarjan's algorithm for strongly connected components. The probabilities of reaching the recurrent classes can be computed in polynomial time on the size of the explicit representation of the state space. The steady state probabilities associated with the states in the recurrent classes can be determined in polynomial time by solving a set of linear equations [Nelson, 1995]. The average rewards can be obtained in polynomial time by summing the products of the probability and reward associated with each state. Hence all the computation

is polynomial time on the explicit representation of the problem, and therefore exponential time on the size of the concise POMDP representation, and the problem is in EXP. □

Lemma 10 *Let c be a real number. Testing the existence of a stationary MDP/UMDP/POMDP policy with average reward $r \geq c$ is in NEXP. The policy evaluation problem in this case is in EXP.*

Proof: First a stationary policy (potentially of exponential size as every state may be assigned a different action) is guessed, which is NEXP computation.

The rest of the proof is like in Lemma 9: the number of states that have to be considered is exponential, and evaluating the value of the policy is EXP computation. Hence the whole computation is in NEXP. □

Theorem 11 *Let c be a real number. Testing the existence of an arbitrary stationary policy with average reward $r \geq c$ for a MDP is EXP-complete.*

Proof: EXP-hardness is by reduction from testing the existence of winning strategies of the perfect-information (fully observable) game G_4 [Stockmeyer and Chandra, 1979]. This game was used by Littman [1997] for showing that finite-horizon planning with sequential effect trees is EXP-hard.

G_4 is a game in which two players take turns in changing the truth-values of variables occurring in a DNF formula. Each player can change his own variables only. Who first makes the formula true has won. For $2n$ variables the game is formalized by n EPSOs, each of which reverses the truth-value of one variable (if it is the turn of player 1) or reverses the truth-value of a randomly chosen variable (if it is the turn of player 2.) Reward 1 is normally obtained, but if the DNF formula evaluates to true after player 2 has made his move, all subsequent rewards will be 0. This will eventually take place if the policy does not represent a winning strategy for player 1, and the average reward will hence be 0. Therefore, the existence of a winning strategy for player 1 coincides with the existence of a policy with average reward 1.

EXP membership is by producing the explicit exponential size representation of the MDP, and then using standard solution techniques based on linear programming [Puterman, 1994]. Linear programming is polynomial time. □

Corollary 12 *Let c be a real number. Testing the existence of an arbitrary history-dependent policy with average reward $r \geq c$ for a MDP is EXP-complete.*

Proof: For fully observable MDPs and policies of unrestricted size, the existence of arbitrary policies with a certain value coincides with the existence of stationary policies with the same value. □

Theorem 13 *Let c be a real number. Testing the existence of an arbitrary stationary policy with average reward $r \geq c$ for a POMDP is NEXP-complete.*

Proof: Membership in NEXP is by Lemma 10. For NEXP-hardness we reduce the NEXP-complete succinct 3SAT [Papadimitriou and Yannakakis, 1986] to concise POMDPs. The reduction is similar to the reduction from the NP-complete 3SAT in [Mundhenk *et al.*, 2000, Theorem 4.13]. Their Theorem 4.25 claims a reduction of succinct 3SAT to stationary policies of POMDPs represented as circuits.

The reduction works as follows. The POMDP randomly chooses one of the clauses and makes the proposition of its first literal observable (the state variables representing the proposition together with two auxiliary variables are the only observable state variables). The stationary policy observes the proposition and assigns it a truth-value. If the literal became true, evaluation proceeds with another clause, and otherwise with the next literal in the clause. Because the policy is stationary, the same truth-value will be selected for the variable irrespective of the polarity of the literal and the clause. If none of the literals in the clause is true, the reward which had been 1 so far will on all subsequent time points be 0.

The succinct 3SAT problem is represented as circuits $\mathcal{C}$ that map a clause number and a literal location (0, 1, 2) to the literal occurring in the clause in the given position. The POMDP uses the following EPSOs the application order of which has been forced to the given order by means of auxiliary state variables. The first EPSO selects a clause by assigning truth-values to state variables representing the clause number. The second EPSO copies the number of the proposition in the current literal (first, second or third literal of the clause) to observable variables, The third and fourth EPSO respectively select the truth-value true and false (this is the only place where the policy has a choice.) The fifth EPSO checks whether the truth-value matches, and if it does not and the literal was the last one, the reward is turned to 0. If it does, the execution continues from the first EPSO, and otherwise, the literal was not the last one and execution continues from the second EPSO and the next literal. □

5 Conclusions

We have analyzed the complexity of probabilistic planning with average rewards, and placed the most important decidable decision problems in the complexity classes EXP and NEXP. Earlier it had been shown that without full observability the most general policy existence problems are not decidable. These results are not very surprising because the problems generalize computational problems that were already known to be very complex (PSPACE-hard), like plan existence in classical deterministic AI planning. Also, these problems are closely related to several finite-horizon problems that were earlier shown EXP-complete and NEXP-complete [Mundhenk *et al.*, 2000]. The results are helpful in devising algorithms for average-reward planning as well as in identifying further restrictions that allow more efficient planning. As shown by Lemma 9, polynomial policy size brings the complexity down to EXP, also in the otherwise undecidable cases. There are likely to be useful structural restrictions on POMDPs that could bring down the complexity further. Restricted but useful problems in PSPACE would be of high interest.

References

[Balcázar *et al.*, 1995] J. L. Balcázar, I. D´az, and J. Gabarró. *Structural Complexity I*. Springer-Verlag, Berlin, 1995.

[Balcázar, 1996] José L. Balcázar. The complexity of searching implicit graphs. *Artificial Intelligence*, 86(1):171–188, 1996.

[Boutilier and Puterman, 1995] Craig Boutilier and Martin L. Puterman. Process-oriented planning and average-reward optimality. In C. S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1096–1103. Morgan Kaufmann Publishers, 1995.

[Boutilier *et al.*, 1999] Craig Boutilier, Thomas Dean, and Steve Hanks. Planning under uncertainty: structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[Bylander, 1994] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[Condon and Lipton, 1989] Anne Condon and Richard J. Lipton. On the complexity of space bounded interactive proofs (extended abstract). In *30th Annual Symposium on Foundations of Computer Science*, pages 462–467, 1989.

[Feigenbaum *et al.*, 1999] Joan Feigenbaum, Sampath Kannan, Moshe Y. Vardi, and Mahesh Viswanathan. Complexity of problems on graphs represented as OBDDs. *Chicago Journal of Theoretical Computer Science*, 5(5), 1999.

[Galperin and Widgerson, 1983] Hana Galperin and Avi Widgerson. Succinct representations of graphs. *Information and Control*, 56:183–198, 1983. See [Lozano, 1988] for a correction.

[Hansen, 1998] Eric A. Hansen. Solving POMDPs by searching in policy space. In Gregory F. Cooper and Seraf´n Moral, editors, *Proceedings of the 1998 Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 211–219. Morgan Kaufmann Publishers, 1998.

[Littman *et al.*, 1998] M. L. Littman, J. Goldsmith, and M. Mundhenk. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research*, 9:1–36, 1998.

[Littman, 1997] Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97) and 9th Innovative Applications of Artificial Intelligence Conference (IAAI-97)*, pages 748–754, Menlo Park, July 1997. AAAI Press.

[Lozano and Balcázar, 1990] Antonio Lozano and José L. Balcázar. The complexity of graph problems for succinctly represented graphs. In Manfred Nagl, editor, *Graph-Theoretic Concepts in Computer Science, 15th International Workshop, WG'89*, number 411 in Lecture Notes in Computer Science, pages 277–286, Castle Rolduc, The Netherlands, 1990. Springer-Verlag.

[Lozano, 1988] Antonio Lozano. NP-hardness of succinct representations of graphs. *Bulletin of the European Association for Theoretical Computer Science*, 35:158–163, June 1988.

[Lusena *et al.*, 1999] Christopher Lusena, Tong Li, Shelia Sittinger, Chris Wells, and Judy Goldsmith. My brain is full: When more memory helps. In Kathryn B. Laskey and Henri Prade, editors, *Uncertainty in Artificial Intelligence, Proceedings of the Fifteenth Conference (UAI-99)*, pages 374–381. Morgan Kaufmann Publishers, 1999.

[Madani *et al.*, 1999] Omid Madani, Steve Hanks, and Anne Condon. On the decidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99) and the Eleventh Conference on Innovative Applications of Artificial Intelligence (IAAI-99)*, pages 541–548. AAAI Press, 1999.

[Meuleau *et al.*, 1999] Nicolas Meuleau, Kee-Eung Kim, Leslie Pack Kaelbling, and Anthony R. Cassandra. Solving POMDPs by searching the space of finite policies. In Kathryn B. Laskey and Henri Prade, editors, *Uncertainty in Artificial Intelligence, Proceedings of the Fifteenth Conference (UAI-99)*, pages 417–426. Morgan Kaufmann Publishers, 1999.

[Mundhenk *et al.*, 2000] Martin Mundhenk, Judy Goldsmith, Christopher Lusena, and Eric Allender. Complexity of finite-horizon Markov decision process problems. *Journal of the ACM*, 47(4):681–720, July 2000.

[Nelson, 1995] Randolph Nelson. *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling*. Springer-Verlag, 1995.

[Papadimitriou and Tsitsiklis, 1987] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, August 1987.

[Papadimitriou and Yannakakis, 1986] Christos H. Papadimitriou and Mihalis Yannakakis. A note on succinct representations of graphs. *Information and Control*, 71:181–185, 1986.

[Paz, 1971] Azaria Paz. *Introduction to Probabilistic Automata*. Academic Press, 1971.

[Puterman, 1994] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 1994.

[Stockmeyer and Chandra, 1979] Larry J. Stockmeyer and Ashok K. Chandra. Provably difficult combinatorial games. *SIAM Journal on Computing*, 8(2):151–174, 1979.

Computational Complexity of Planning with Temporal Goals

Chitta Baral
CS&E, Arizona State University
Tempe, AZ 85287-5406, USA
chitta@asu.edu

Vladik Kreinovich
University of Texas at El Paso
El Paso, TX 79968, USA
vladik@cs.utep.edu

Raúl A. Trejo
ITESM Campus Edo. México
Atizapan, México 52926
rtrejo@campus.cem.itesm.mx

Abstract

In the last decade, there has been several studies on the computational complexity of planning. These studies normally assume that the goal of planning is to make a certain fluent true after the sequence of actions. In many real-life planning problems, the goal is represented in a much more complicated temporal form: e.g., in addition to having a desired fluent true at the end, we may want to keep certain fluents true at all times. In this paper, we study the complexity of planning for such temporal goals. We show that for goals expressible in Linear Temporal Logic, planning has the same complexity as for non-temporal goals: it is **NP**-complete; and for goals expressible in a more general Branching Temporal Logic, planning is **PSPACE**-complete.

1 Introduction

In the presence of complete information about the initial situation, a plan – in the sense of classical planning – is a sequence of actions that takes the agent from the initial situation to the state which satisfies a given goal. Traditionally, a goal is described as a fluent which must be true after all the actions. For such goals, the computational complexity of finding a plan has been well-studied [Bylander, 1994; Erol *et al.*, 1995; Liberatore, 1997; Baral *et al.*, 2000]. In the most natural formulation, the problem of finding polynomial length plans is **NP**-complete (for exact definitions of standard complexity terms such as **NP**-completeness, see, e.g., [Papadimitriou, 1994; Baral *et al.*, 2000]).

In many real-life planning problems, the goal is represented in a much more complicated temporal form: e.g., in addition to having a desired fluent true at the end, we may want to keep certain fluents true at all times; for example, we may want to make sure that certain safety constraints are satisfied at all times. *In this paper, we study the complexity of planning for such temporal goals.* There exist two formalisms for describing temporal goals: *Linear Temporal Logic* (LTL) [Bacchus and Kabanza, 1998] in which we are allowed to refer to the *actual* past and future events, and *Branching Temporal Logic* CTL [Niyogi and Sarkar, 2000] in which we are also allowed to refer to events from the *possible* future. In this paper, we will describe the computational complexity of planning in both logics. To the best of our knowledge this has not been done before.

Our complexity analysis will be based on the action description language $\mathcal{A}$ proposed in [Gelfond and Lifschitz, 1993]. The language $\mathcal{A}$ and its variants have made it easier to understand the fundamentals (such as inertia, ramification, qualification, concurrency, sensing, etc.) involved in reasoning about actions and their effects on a world, and we would like to stick to that simplicity principle here. To stick to the main point we consider the simplest action description, and do not consider features such as executability conditions. We now start with a brief description of the language $\mathcal{A}$.

1.1 The language $\mathcal{A}$: brief reminder

In the language $\mathcal{A}$, we start with a finite list of properties (fluents) $f_1, \ldots, f_n$ which describe possible properties of a state. A *state* is then defined as a finite set of fluents, e.g., $\{\}$ or $\{f_1, f_3\}$. We are assuming that we have complete knowledge about the initial state: e.g., $\{f_1, f_3\}$ means that in the initial state, properties f_1 and f_3 are true, while all the other properties $f_2, f_4, \ldots$ are false. The properties of the initial state are described by formulas of the type

$$\mathsf{initially}\ f,$$

where f is a *fluent literal*, i.e., either a fluent f_i or its negation $\neg f_i$.

To describe possible changes of states, we need a finite set of *actions*. In the language $\mathcal{A}$, the effect of each action a can be described by formulas of the type

$$a\ \mathsf{causes}\ f\ \mathsf{if}\ f_1, \ldots, f_m,$$

where $f, f_1, \ldots, f_m$ are fluent literals. A reasonably straightforward semantics describes how the state changes after an action:

- If, before the execution of an action a, fluent literals $f_1, \ldots, f_m$ were true, and the domain description contains a rule "a causes f if $f_1, \ldots, f_m$", then this rule is *activated*, and after the execution of the action a, f becomes true.
- If for some fluent f_i, no activated rule enables us to conclude that f_i is true or false, this means that the execution of action a does not change the truth of this fluent; therefore, f_i is true in the resulting state if and only if it was true in the old state.

Formally, a *domain description* D is a finite set of *value propositions* of the type "initially f" (which describe the initial state), and a finite set of *effect propositions* of the type "a causes f if $f_1, \ldots, f_m$" (which describe results of actions). The *initial state* s_0 consists of all the fluents f_i for which the corresponding value proposition "initially f_i" is contained in the domain description. (Here we are assuming that we have complete information about the initial situation.) We say that a fluent f_i *holds* in s if $f_i \in s$; otherwise, we say that $\neg f_i$ holds in s. The *transition function* $Res_D(a, s)$ which describes the effect of an action a on a state s is defined as follows:

- we say that an effect proposition "a causes f if $f_1, \ldots, f_m$" is *activated* in a state s if all m fluent literals $f_1, \ldots, f_m$ hold in s;
- we define $V_D^+(a, s)$ as the set of all fluents f_i for which a rule "a causes f_i if $f_1, \ldots, f_m$" is activated in s;
- similarly, we define $V_D^-(a, s)$ as the set of all fluents f_i for which a rule "a causes $\neg f_i$ if $f_1, \ldots, f_m$" is activated in s;
- if $V_D^+(a, s) \cap V_D^-(a, s) \neq \emptyset$, we say that the result of the action a is *undefined*;
- if the result of the action a is *defined* in a state s (i.e., if $V_D^+(a, s) \cap V_D^-(a, s) = \emptyset$), we define $Res_D(a, s) = (s \cup V_D^+(a, s)) \setminus V_D^-(a, s)$.

A *plan* p is defined as a sequence of actions $[a_1, \ldots, a_m]$. The *result* $Res_D(p, s)$ of applying a plan p to the initial state s_0 is defined as

$$Res_D(a_m, Res_D(a_{m-1}, \ldots, Res_D(a_1, s_0)\ldots)).$$

The *planning problem* is: given a domain D and a desired property, find a plan for which the resulting trajectory s_0, $s_1 \stackrel{\text{def}}{=} Res_D(a_1, s_0)$, $s_2 \stackrel{\text{def}}{=} Res_D(a_2, s_1)$, etc., satisfies the desired property. In particular, if the goal is to make a certain fluent f true, then the planning problem consists of finding a plan which leads to the state in which f is true.

In addition to the planning problem, it is useful to consider the *plan checking* problem: given a domain, a desired property, and a candidate plan, check whether this action plan satisfies the desired property. It is known that in the presence of complete information about the initial situation, for fluent goals, plan checking is a *tractable* problem – i.e., there exists a polynomial-time algorithm for checking whether a given plan satisfies the given fluent goal [Bylander, 1994; Erol *et al.*, 1995; Liberatore, 1997; Baral *et al.*, 2000].

1.2 Temporal extensions: motivations

Let us give two examples of planning problems explaining why temporal extensions are desirable:

1) If we are planning a flight of an automatic spy miniplane, then the goal is not only to reach the target point (which can be described by the fluent r), but also to avoid detection; more formally, a fluent d ("detected") must remain false all the time.

2) When planning a movement of a robot, we may want to require that not only the robot achieve its goal but also that, whenever the robot strays from the desired trajectory, it should always be possible to bring the robot back to this trajectory (i.e., make the fluent $on\_trajectory$ true) by a single corrective action.

1.3 Linear temporal logic: brief reminder

In Linear Temporal Logic (LTL), in addition to the truth values of a fluent at the current moment of time, we can also refer to its truth values in the past and in the future. For this, LTL has several *operators*. Different authors use different notations for these operators. Since we will also analyze planning in branching time logic described in [Niyogi and Sarkar, 2000] as an extension of LTL, we will use notations from [Niyogi and Sarkar, 2000] for LTL operators.

LTL has four basic *future* operators:

- X (neXt time in the future): $\mathsf{X}p$ is true at a moment time t if p is true at the moment $t + 1$;
- G (Going to be always true): $\mathsf{G}p$ is true at the moment t is p is true at all moments of time $s > t$.
- F (sometime in the Future): $\mathsf{F}p$ is true at the moment t if p is true at some moment $s > t$;
- U (Until): $p\mathsf{U}q$ is true at the moment t if p is true at this moment of time and at all the future moments of time until q becomes true.

Similarly, LTL has four basic *past* operators:

- P (Previously): $\mathsf{P}p$ is true at a moment time t if p is true at the moment $t - 1$;
- H (Has always been): $\mathsf{H}p$ is true at the moment t if p was true at all moments of time $s < t$;
- O (Once or sometime in the past): $\mathsf{O}p$ is true at the moment t if p was true at some moment $s < t$;
- S (Since): $p\mathsf{S}q$ is true at the moment t if p is true at this moment of time and at all the past moments of time since the last time when q was true.

We can combine several such operators: e.g., $\mathsf{X}^3p \stackrel{\text{def}}{=} \mathsf{XXX}p$ is true at a moment t if p is true at the moment $t + 3$.

In general, an *LTL-goal* is a goal which is obtained from fluents by using LTL operators and propositional connectives & ("and"), $\vee$ ("or"), and $\neg$ ("not").

For example, the objective from our first planning problem can be easily formulated as the following LTL-goal: $S \stackrel{\text{def}}{=} r \,\&\, \neg d \,\&\, \mathsf{H}(\neg d)$.

Comment. Some versions of LTL have additional operators, e.g., we may have *interval* operators in which moments of time $s > t$ or $s < t$ are restricted to a given interval [Bacchus and Kabanza, 1998; Niyogi and Sarkar, 2000].

1.4 Branching temporal logic: brief reminder

In the Branching Temporal Logic CTL [Emerson, 1990; Niyogi and Sarkar, 2000], in addition to LTL operations, we have two additional operators E and A which describe possible futures:

- $\mathsf{E}p$ (Exists path) is true at the state s at the time t if there exists a possible evolution of this state for which p is true at this same moment t.

- $\mathsf{A}p$ (All paths) is true at the state s at the time t if for all possible evolutions of this state, p is true at this same moment t.

For example, the requirement that, no matter what action we apply to the state s, a fluent f will always stay true, can be described as $\mathsf{A}(\mathsf{X}f)$.

Similarly, we can describe in this language the *fast maintainability* requirement from our second planning problem: no matter what action we apply to the state s, if a fluent f stops being true after this action, we can always make the property f true by applying appropriate correcting action.

In CTL, this fast maintainability requirement can be formulated as follows:

- Once we have already reached the next state s', the possibility to get f back by applying a single correcting action means that either f is already true, or there is a path in which f will be true at the next moment of time ($\mathsf{X}f$), i.e., that $f \vee \mathsf{E}(\mathsf{X}f)$.
- So, the fast maintainability requirement means that every possible immediate future state s' satisfies the above property $f \vee \mathsf{E}(\mathsf{X}f)$, i.e., in CTL notations, that

$$\mathsf{A}(\mathsf{X}(f \vee \mathsf{E}(\mathsf{X}f))). \quad (1)$$

Comment. The description of more general temporal logics can be found in [Gabbay et al., 1994].

2 Results

2.1 What kind of planning problems we are interested in

Informally speaking, we are interested in the following problem:

- *given* a domain description (i.e., the description of the initial state and of possible consequences of different actions) and a goal (i.e., a fluent which we want to be true),
- *determine* whether it is possible to achieve this goal (i.e., whether there exists a plan which achieves this goal).

We are interested in analyzing the *computational complexity* of the planning problem, i.e., analyzing the computation time which is necessary to solve this problem.

Ideally, we want to find cases in which the planning problem can be solved by a *tractable* algorithm, i.e., by an algorithm $\mathcal{U}$ whose computational time $t_{\mathcal{U}}(w)$ on each input w is bounded by a polynomial $p(|w|)$ of the length $|w|$ of the input w: $t_{\mathcal{U}}(x) \le p(|w|)$ (this length can be measured bit-wise or symbol-wise). Problems which can be solved by such *polynomial-time* algorithms are called problems from the class **P** (where **P** stands for *p*olynomial-time). If we cannot find a polynomial-time algorithm, then at least we would like to have an algorithm which is as close to the class of tractable algorithms as possible.

Since we are operating in a time-bounded environment, we should worry not only about the time for *computing* the plan, but we should also worry about the time that it takes to actually *implement* the plan. If a (sequential) action plan consists of a sequence of 2^{2^n} actions, then this plan is not tractable. It is therefore reasonable to restrict ourselves to *tractable* plans, i.e., to plans u whose duration $T(u)$ is bounded by a polynomial $p(|w|)$ of the input w.

With this tractability in mind, we can now formulate the above planning problem in precise terms:

- *given:* a polynomial $p(n) \ge n$, a domain description D (i.e., the description of the initial state and of possible consequences of different actions) and a goal statement S (i.e., a statement which we want to be true),
- *determine* whether it is possible to tractably achieve this goal, i.e., whether there exists a tractable-duration plan u (with $T(u) \le p(|D| + |S|)$) which achieves this goal.

We are interested in analyzing the *computational complexity* of this planning problem.

2.2 Complexity of the planning problem with goals expressible in Linear Temporal Logic

Theorem 1. *For goals expressible in Linear Temporal Logic* (LTL)*, the planning problem is* **NP***-complete.*

Comments.

- Since the planning problem is **NP**-complete even for simple (non-temporal) goals [Bylander, 1994; Erol *et al.*, 1995; Liberatore, 1997; Baral *et al.*, 2000], this result means that allowing temporal goals from LTL does not increase the computational complexity of planning.
- This result is in good accordance with the fact that the decidability problem for linear temporal logic is also **NP**-complete [Emerson, 1990].
- For readers' convenience, all the proofs are placed in the special Proofs section.

The proof of Theorem 1 is based on the following result:

Theorem 2. *For goals expressible in Linear Temporal Logic* (LTL)*, the plan checking problem is tractable.*

We give the proofs of Theorems 1 and 2 for the version of Linear Temporal Logic which only uses eight basic temporal operators. However, as one can easily see from the proofs, these result remains true if we allow more sophisticated temporal operators, e.g., interval operators of the type $F_{[t,s]}$ meaning that the fluent is true in some future moment of time from the interval $[t, s]$ [Bacchus and Kabanza, 1998; Niyogi and Sarkar, 2000].

2.3 Complexity of the planning problem with goals expressible in Branching Temporal Logic

Theorem 3. *For goals expressible in Branching Temporal Logic* CTL*, the planning problem is* **PSPACE***-complete.*

Comment. This result is in good accordance with the fact that the decidability problem for most branching temporal logics is also **PSPACE**-complete [Gabbay et al., 1994].

For the Branching Temporal Logic, not only planning, but even plan checking is difficult:

Theorem 4. *For goals expressible in Branching Temporal Logic* CTL*, the plan checking problem is* **PSPACE***-complete.*

Theorems 3 and 4 mean that allowing temporal goals from CTL can drastically increase the computational complexity of planning. These results, however, are not that negative because most safety and maintainability-type conditions can be expressed in a simple subclass of CTL. Namely, in the maintainability conditions like the one above, we do not need to consider all possible trajectories, it is sufficient to consider trajectories which differ from the actual one by no more than one (or, in general, by no more than k) states. In this case, the planning problem becomes much simpler:

Definition 1. *Let $k > 0$ be a positive integer. We say that an expression in* CTL *is k-limited if this expression remains true when in all operators* E *and* A*, we only allow possible trajectories differ from the actual trajectory in no more than k moments of time.*

For example, the above maintainability statement (1) means that all possible 1-deviations from the actual trajectory are maintainable and therefore, this statement is 1-limited.

Theorem 5. *Let $k > 0$ be an integer. For k-limited goals expressible in Branching Temporal Logic* CTL*, the planning problem is* **NP***-complete.*

Theorem 6. *Let $k > 0$ be an integer. For k-limited goals expressible in Branching Temporal Logic* CTL*, the plan checking problem is tractable.*

2.4 Conclusions

Our first conclusion is that if, instead of traditional goals which only refer to the state of the system at the last moment of time, we allow goals which explicitly mention the actual past and actual future states, the planning problem does not become much more complex: it stays on the same level of complexity hierarchy.

Our second conclusion is that if we allow goals which refer to *potential* future, the planning problem can become drastically more complicated. Thus, we should be very cautious about such more general goals.

3 Proofs

Proof of Theorems 1 and 2. We already know that the planning problem is **NP**-complete even for the simplest possible case of LTL-goals: namely, for goals which are represented simply by fluents [Bylander, 1994; Erol *et al.*, 1995; Liberatore, 1997; Baral *et al.*, 2000]. Therefore, to prove that the general problem of planning under LTL-goals is **NP**-complete, it is sufficient to prove that this general problem belongs to the class **NP**.

Indeed, it is known [Papadimitriou, 1994] that a problem belongs to the class **NP** if the corresponding formula $F(w)$ can be represented as $\exists u P(u,w)$, where $P(u,w)$ is a tractable property, and the quantifier runs over words of tractable length (i.e., of length limited by some given polynomial of the length of the input).

For a given planning situation w, checking whether a successful plan exists or not means checking the validity of the formula $\exists u\, P(u,w)$, where $P(u,w)$ stands for "the plan u succeeds for the situation w". According to the above definition of the class **NP**, to prove that the planning problem belongs to the class **NP**, it is sufficient to prove the following two statements:

- the quantifier runs only over words u of tractable length, and
- the property $P(u,w)$ can be checked in polynomial time.

The first statement immediately follows from the fact that in this paper, we are considering only plans of polynomial (tractable) duration, i.e., sequential plans u whose length $|u|$ is bounded by a polynomial of the length $|w|$ of the input w: $|u| \le p(|w|)$, where $p(n)$ is a given polynomial. So, the quantifier runs over words of tractable length.

Let us now prove the second statement – that plan checking can be done in polynomial time. (This statement constitutes Theorem 2.) Once we have a plan u of tractable length, we can check its successfulness in a situation w as follows:

- we know the initial state s_0;
- take the first action from the action plan u and apply it to the state s_0; as a result, we get the state s_1;
- take the second action from the action plan u and apply it to the state s_1; as a result, we get the state s_2; etc.

At the end, we get the values of all the fluents at all moments of time. On each step of this construction, the application of an action to a state requires linear time; in total, there are polynomial number of steps in this construction. Therefore, computing the values of all the fluents at all moments of time indeed requires polynomial time.

Let us now take the desired goal statement S and *parse* it, i.e., describe, step by step, how we get from fluents to this goal statement S.

> For example, for the above spy-plane goal statement $S \equiv r \,\&\, \neg d \,\&\, \mathsf{H}(\neg d)$, parsing leads to the following sequence of intermediate statements: $S_1 := \neg d$, $S_2 := r \,\&\, S_1$, $S_3 := \mathsf{H}S_1$, and finally, $S \equiv S_4 := S_2 \,\&\, S_3$.

The number of the resulting intermediate statements cannot exceed the length of the goal statement; thus, this number is bounded by the length $|S|$ of the goal statement.

Based on the values of all the fluents at all moments of time, we can now sequentially compute the values of all these intermediate statements S_i at all moments of time:

- When a new statement is obtained from one or two previous ones by a logical connective (e.g., in the above example, as $S_4 := S_2 \,\&\, S_3$), then, to compute the value of the new statement at all T moments of time, we need T logical operations.
- Let us now consider the case when a new statement is obtained from one or two previously computed ones by using one temporal operations: e.g., in the above example, as $S_3 := \mathsf{H}S_1$). Then, to compute the truth value of S_3 at each moment of time, we may need to go over all other moments of time. So, to compute S_i for each moment of time t, we need $\le T$ steps. Hence, to compute the truth value of S_i for *all* T moments of time, we need $\le T^2$ steps.

In both cases, for each of $\leq |S|$ intermediate statements, we need $\leq T^2$ computations. Thus, to compute the truth value of the desired goal statement, we need $\leq T^2 \cdot |S|$ computational steps. Since we look for plans for which $T \leq p(|D|+|S|)$ for some polynomial $p(n)$, we thus need a polynomial number of steps to check whether the given plan satisfies the given goal.

So, we can check the success of a plan in polynomial time, and thus, the planning problem indeed belongs to the class **NP**. The theorems are proven.

Proof of Theorem 3. This proof follows the same logic as proofs of **PSPACE**-completeness of other planning problems; see, e.g., [Littman, 1997] and [Baral *et al.*, 2000].

By definition, the class **PSPACE** is formed by problems which can be solved by a polynomial-*space* algorithm. It is known (see, e.g., [Papadimitriou, 1994]) that this class can be equivalently reformulated as a class of problems for which the checked formula $P(w)$ can be represented as $\forall u_1 \exists u_2 \ldots P(u_1, u_2, \ldots, u_k, w)$, where the number of quantifiers k is bounded by a polynomial of the length of the input, $P(u_1, \ldots, u_k, w)$ is a tractable property, and all k quantifiers run over words of tractable length (i.e., of length limited by some given polynomial of the length of the input). In view of this result, it is easy to see that for CTL-goals, the planning problem belongs to the class **PSPACE**. Indeed, all the operators of CTL can be described by quantifiers over words of tractable length, namely, either over paths (for operators A and E) or over moments of time (for LTL operators). A plan is also a word of tractable length. Thus, the existence of a plan which satisfies a given CTL-goal can be described by a tractable sequence of quantifiers running over words of tractable length. Thus, for CTL-goals, the planning problem does belong to **PSPACE**.

To prove that the planning problem is **PSPACE**-complete, we will show that we can reduce, to the planning problem, a problem known to be **PSPACE**-complete: namely, the problem of checking, for a given propositional formula F with the variables $x_1, \ldots, x_m, x_{m+1}, \ldots, x_n$, the validity of the formula $\mathcal{F}$ of the type $\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots F$. This reduction will be done as follows. Consider the planning problem with two actions a^+ and a^-, and $2n+1$ fluents $x_1, \ldots, x_n$, $t_0, t_1, \ldots, t_n$. These actions and fluents have the following meaning:

- the meaning of t_i is that we are at moment of time i;
- the action a^+, when applied at moment t_{i-1}, makes i-th variable x_i true;
- the action a^-, when applied at moment t_{i-1}, makes i-th variable x_i false.

The corresponding initial conditions are:

- $\mathsf{initially}\ \neg x_i$ (for all i);
- $\mathsf{initially}\ t_0$; $\mathsf{initially}\ \neg t_i$ (for all $i > 0$).

The effect of actions if described by the following rules (effect propositions):

- for $i = 1, 2, \ldots, n$, the rules

$$a^+ \ \mathsf{causes}\ x_i \ \mathsf{if}\ t_{i-1}; \quad a^- \ \mathsf{causes}\ \neg x_i \ \mathsf{if}\ t_{i-1};$$

  describe how we assign values to the variables x_i;

- for $i = 1, 2, \ldots, n$, the rules

$$a^+ \ \mathsf{causes}\ t_i \ \mathsf{if}\ t_{i-1}; \quad a^- \ \mathsf{causes}\ t_i \ \mathsf{if}\ t_{i-1};$$
$$a^+ \ \mathsf{causes}\ \neg t_{i-1} \ \mathsf{if}\ t_{i-1}; \quad a^- \ \mathsf{causes}\ \neg t_{i-1} \ \mathsf{if}\ t_{i-1};$$

  describe the update of the time fluents t_i.

The corresponding goal is designed as follows:

- first, we replace in the above quantified propositional formula $\mathcal{F}$, each existential quantifier $\exists x_i$ by EX, each universal quantifier $\forall x_i$ by AX; let us denote the result of this replacement by F';
- then, we add $\,\&\, t_0$ to the resulting formula F';
- finally, we add P^n in front of the whole thing – creating $\mathsf{P}^n(F' \,\&\, t_0)$.

For example, for a formula $\exists x_1 \forall x_2 F$, this construction leads to the following goal: $\mathsf{P}^2(\mathsf{EX}(\mathsf{AX}(F)) \,\&\, t_0)$. This reduction leads to a linear increase in length, so this reduction is polynomial-time.

To complete the proof, we must show that this is a "valid" reduction, i.e., that the resulting planning problem is solvable if and only if the original quantified propositional formula is true.

To prove this equivalence, let us first remark that, by definition of the operator P ("previous"), the goal formula $\mathsf{P}^n(F' \,\&\, t_0)$ is true at a moment t if and only if the formula $F' \,\&\, t_0$ holds at a moment $t-n$. Since, due to our rules, t_0 is only true at the starting moment of time $t = 0$, the goal can only be true if $t = n$. Thus, to check whether we can achieve the goal, it is sufficient to check whether we can achieve it at the moment n, i.e., after a sequence of n actions. In this case, the validity of the goal is equivalent to validity of the formula F' at the initial moment $t = 0$.

Let us now show that the validity of the formula F' at the moment $t = 0$ is indeed equivalent to the validity of the above quantified propositional formula. We will prove this equivalence by induction over the total number of variables n.

Induction base: For $n = 0$, we have no variables x_i at all, so F is either identically true or identically false. In this case, F' simply coincides with F, so they are, of course, equivalent.

Induction step: Let us assume that we have proven the desired equivalence for all quantified propositional formulas with $n-1$ variables; let us prove it for quantified propositional formulas with n variables.

Indeed, let a quantified propositional formula $\mathcal{F}$ of the above type be given. There are two possibilities for the first variable x_1 of this formula:

- it may be under the existential quantifier $\exists x_1$; or
- it may be under the universal quantifier $\forall x_1$.

1°. In the first case, the formula $\mathcal{F}$ has the form $\exists x_1 \mathcal{G}$, where for each x_1, $\mathcal{G}$ is a quantified propositional formula with $n-1$ variables $x_2, \ldots, x_n$. According to our construction, the CTL formula F' has the form $\mathsf{E}(\mathsf{X}G')$, where G' is the result of applying this same construction to the formula $\mathcal{G}$.

To show that F' is indeed equivalent to $\mathcal{F}$, we will first show that F' implies $\mathcal{F}$, and then that $\mathcal{F}$ implies F'.

1.1°. Let us first show that F' implies $\mathcal{F}$.

Indeed, by definition of the operator E, if the formula $F' \equiv \mathsf{E}(\mathsf{X}G')$ holds at the moment $t = 0$ this means that there exists a path for which, at moment $t = 0$, the formula $\mathsf{X}G'$ is true.

By definition of the operator X ("next"), the fact that the formula $\mathsf{X}G'$ is true at the moment $t = 0$ means that the formula G' is true at the next moment of time $t = 1$.

By the time $t = 1$, we have applied exactly one action which made x_1 either true or false, after which the value of this variable x_1 does not change. Let us select the value x_1 as "true" or "false" depending on which value was selected along this path.

The moment t_1 can be viewed as a starting point for the planning problem corresponding to the remaining formula $\mathcal{G}$. By induction assumption, the validity of G' at this new starting moment is equivalent to the validity of the quantified propositional formula $\mathcal{G}$. Thus, the formula $\mathcal{G}$ is true for this particular x_1, hence the original formula $\mathcal{F} \equiv \exists x_1 \mathcal{G}$ is also true. So, F' indeed implies $\mathcal{F}$.

1.2°. Let us now show that $\mathcal{F}$ implies F'.

Indeed, if $\mathcal{F} \equiv \exists x_1 \mathcal{G}$ is true, this means that there exists a value x_1 for which $\mathcal{G}$ is true. By the induction assumption, this means that for this same x_1, the goal formula G' is also true at the new starting moment $t = 1$. Thus, for any path which starts with selecting this x_1, the formula $\mathsf{X}G'$ is true at the previous moment $t = 0$. Since this formula is true for *some* path, by definition of the operator E, it means that the formula $\mathsf{E}(\mathsf{X}G')$ is true at the moment $t = 0$, and this formula is exactly F'.

Thus, $\mathcal{F}$ does imply F', and hence $\mathcal{F}$ and F' are equivalent.

2°. The second case, when x_1 is under the universal quantifier $\forall x_1$, can be handled similarly.

The induction step is proven, and thus, by induction, the equivalence holds for all n.

Thus, the reduction is valid, and the corresponding planning problem is indeed **PSPACE**-complete. Q.E.D.

Proof of Theorem 4. Similarly to the proof of Theorem 3, we can show that plan checking belongs to the class **PSPACE**, so all we need to prove is the desired reduction. From the proof of Theorem 3, one can see that the exact same reduction will work here as well, because in this reduction, the equivalence between $\mathcal{F}$ and F' did not depend on any action plan at all. The equivalence used in the proof of Theorem 3 is based on the analysis of *possible* trajectories and does not use the actual trajectory at all.

Thus, we can pick any action plan (e.g., a sequence consisting of n actions a^+), and the desired equivalence will still hold. Q.E.D.

Proof of Theorems 5 and 6. For trajectories of duration T, with A possible actions at each moment of time, there are no more than $T \cdot A$ possible trajectories differing in one state, no more than $(T \cdot A)^2$ trajectories differing in two states, etc. In general, whatever number k we fix, there is only a polynomial number ($\leq (T \cdot A)^k$) of possible trajectories which differ from the actual one in $\leq k$ places.

Therefore, for fixed k, we can explicitly describe the new operators E and A by enumerating all such possible trajectories. Thus, similarly to the proof of Theorems 1 and 2, we can conclude that for k-planning, plan checking is tractable and the corresponding planning problem is **NP**-complete.

Acknowledgments

This work was supported by NASA (grant NCC5-209 and a Research on Intelligent Systems grant), by the AFOSR grant F49620-00-1-0365, by the grant W-00016 from the U.S.-Czech Science and Technology Joint Fund, and by the NSF grants IRI 9501577, 0070463, and 9710940 Mexico/Conacyt.

The authors are thankful to the anonymous referees for valuable comments.

References

[Bacchus and Kabanza, 1998] Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22:5–27, 1998.

[Baral *et al.*, 2000] Chitta Baral, Raúl Trejo, and Vladik Kreinovich. Computational complexity of planning and approximate planning in the presence of incompleteness. *Artificial Intelligence*, 122:241–267, 2000.

[Bylander, 1994] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:161–204, 1994.

[Emerson, 1990] E. Allen Emerson. Temporal and modal logics. In: Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, Vol. B, pages 995–1072, MIT Press, Cambridge, Massachusetts, 1990.

[Erol *et al.*, 1995] Kuthulan Erol, Dana S. Nau, and V.S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88, 1995.

[Gabbay et al., 1994] Dov M. Gabbay, Ian Hodkinson, and Mark Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*. Oxford University Press, New York, 1994.

[Gelfond and Lifschitz, 1993] Michael Gelfond and Vladimir Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.

[Liberatore, 1997] Paolo Liberatore. The complexity of the language $\mathcal{A}$. *Electronic Transactions on Artificial Intelligence*, 1:13–28 (http://www.ep.liu.se/ej/etai/1997/02), 1997.

[Littman, 1997] Michael L. Littman. Probabilistic propositional planning: representations and complexity. In *AAAI 97*, pages 748–754, 1997.

[Niyogi and Sarkar, 2000] Rajdeep Niyogi and Sudeshna Sarkar. Logical specification of goals. In *International Conference on Information Technology ICIT'2000*, Bhubaneswar, India, December 21-23, 2000. Tata McGraw-Hill.

[Papadimitriou, 1994] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

A Simplifier for Propositional Formulas with Many Binary Clauses

Ronen I. Brafman
Department of Computer Science
Ben-Gurion University
Beer-Sheva, Israel
brafman@cs.bgu.ac.il

Abstract

Deciding whether a propositional formula in conjunctive normal form is satisfiable (SAT) is an NP-complete problem. The problem becomes linear when the formula contains binary clauses only. Interestingly, the reduction to SAT of a number of well-known and important problems – such as classical AI planning and automatic test pattern generation for circuits – yields formulas containing many binary clauses. In this paper we introduce and experiment with 2-SIMPLIFY, a formula simplifier targeted at such problems. 2-SIMPLIFY constructs the implication graph corresponding to the binary clauses in the formula and uses this graph to deduce new unit literals. The deduced literals are used to simplify the formula and update the graph, and so on, until stabilization. Finally, we use the graph to construct an equivalent, simpler set of binary clauses. Experimental evaluation of this simplifier on a number of bench-mark formulas produced by encoding AI planning problems prove 2-SIMPLIFY to be fast and effective.

1 Introduction

Propositional satisfiability (SAT) is the problem of deciding whether a propositional formula in conjunctive normal form (CNF) is satisfiable. SAT was the first problem shown to be NP-complete [Cook, 1971] and has important practical applications. In the last decade we have witnessed great progress in SAT solution methods, first with the introduction of efficient stochastic local search algorithms [Selman *et al.*, 1992; 1994], and more recently with a number of efficient systematic solvers, such as REL-SAT [Bayardo and Schrag, 1997] and SATZ [Li and Anbulagan, 1997] and their randomized versions [Gomes *et al.*, 1998].

Among AI researchers, interest in SAT solution algorithms has increased farther since Kautz and Selman showed that some classical planning problems can be solved more quickly when they are reduced to SAT problems [Kautz and Selman, 1996]. Kautz and Selman's *planning as satisfiability* approach is based on generic SAT technology, and, aside from the translation process itself, makes no use of properties specific to planning problems. However, SAT-encoded planning problems have an important syntactic property: they contain a large fraction of binary clauses. Interestingly, this property is found in another important domain – automatic test-pattern generation for circuits [Larrabee, 1992].

Unlike the general SAT problem, 2-SAT, the problem of deciding whether a propositional formula containing binary clauses *only* has a satisfying assignment is (constructively) solvable in linear time. One would hope that this property would make it easier to solve SAT instances containing a large fraction of binary clauses. In this article we describe the 2-SIMPLIFY preprocessor, a simplifier that is geared to such formulas. Like other simplifiers (e.g., Crawford's COMPACT), this algorithm takes a propositional formula in CNF as input and outputs a new propositional formula. Naturally, for this process to be worthwhile, the new formula should be easier to solve, and the overall time required for simplification and solution of the simplified formula should be less than the time required to solve the original SAT instance. Experiments on a number of bench-mark formulas derived from planning problems show that 2-SIMPLIFY meets this criteria.

2-SIMPLIFY efficiently implements and combines well-known 2-SAT techniques, a limited form of hyper-resolution, and novel use of transitive reduction to reduce formula size. The basic idea is as follows: each clause of the form $p \vee q$ is equivalent to two implications: $\neg p \rightarrow q$ and $\neg q \rightarrow p$. We use this property to construct a graph (known as the *implication graph* [Aspvall *et al.*, 1979]) from the set of binary clauses. This graph contains a node for each literal in the language and a directed edge from a literal l to another literal l' if the (disjunction equivalent to the) implication $l \rightarrow l'$ appears among the set of binary clauses. After constructing this graph, we compute its transitive closure and check each literal to see whether its negation appears among its descendants. If $\neg l$ is a descendant of l, we can immediately conclude that $\neg l$ is a consequence of the original formula. Once we know that l holds, we can simplify the original formula. The simplified formula may contain new binary clauses, which are immediately added to the graph. 2-SIMPLIFY utilizes these and other ideas to quickly derive unit literals from the original formula and produce the simplest equivalent formula as its output.

In the next section we discuss the background of this work in more detail. In Section 3 we present the simplification algorithm used by 2-SIMPLIFY. In Section 4 we present some experimental results, and we conclude in Section 5.

2 Background

We start with some SAT background and then we briefly explain the *planning as satisfiability* approach, which motivated this work.

2.1 The SAT Problem

The SAT problem is defined as follows: given a propositional formula in conjunctive normal form (CNF) output YES if the formula is satisfiable and NO otherwise. In practice, a positive answer is accompanied by some satisfying assignment.

There are two classes of SAT algorithms: stochastic and systematic. Stochastic methods, such as G-SAT[Selman *et al.*, 1992] and WALKSAT[Selman *et al.*, 1994], perform stochastic local search in the space of truth assignment. Often, they can find solutions quickly, but they cannot identify an unsatisfiable instance. Their performance is extremely sensitive to the choice of heuristic and various other parameters. Systematic methods systematically search the space of truth assignments. Thus, they can identify unsatisfiable instances. Modern systematic algorithms are quite fast and stable thanks to improved branch choice heuristics and backtracking techniques. In addition, systematic solvers can be improved by introducing some randomization into their search procedure, e.g., their choice of branch variable. See [Gomes *et al.*, 1998] for more information on this topic.

Often, a formula simplifier is applied before the SAT solver. Simplifiers use specialized, efficient deductive methods to reduce the original formula into a simpler formula which is typically easier to solve. The best known simplification method is unit propagation. When one of the clauses in the formula contains a single literal, it must be assigned the value *true* in any satisfying assignment. For example, if a formula contains the clause $\{\neg p\}$ then $\neg p$ must be *true* in any satisfying assignment, i.e., p must be *false*. Once we deduce this fact, we can use it to simplify other clauses: clauses that contain $\neg p$ can be removed since their satisfaction is guaranteed when p is *false*, and the literal p can be removed from any clause containing it (e.g., $\{p, \neg s\}$ will be transformed into $\{\neg s\}$) because it is equivalent to *false*. As we just saw, the simplification process can yield additional unit clauses, which are used to produce additional simplifications. If during the simplification process an empty clause is discovered (e.g., if we assigned s the value *true* and there is a unit clause $\{\neg s\}$) we can conclude that the formula is unsatisfiable.

There are a number of additional simplification methods, such as failed unit clause and failed binary clause, where a unit or binary clause are added to the current formula and we attempt to show (e.g., using unit propagation) that the resulting formula is inconsistent. In that case, the negation of the added clause is implied by the original formula, and we update the truth assignment accordingly. For example, if our original formula becomes inconsistent once we add the clause $\{p, q\}$, we know that both p and q must be assigned *false*.

2.2 2-SAT

2-SAT is a subclass of SAT in which clauses contain no more than two literals. While SAT is NP-complete [Cook, 1971], 2-SAT can be solved in linear time. The key step in solving 2-SAT problems is the construction of the *implication graph* [Aspvall *et al.*, 1979]). The nodes of the implication graph correspond to the literals in the formula. The graph contains an edge between the literal l and the literal l' if the clause $\{\neg l, l'\}$ appears in the formula. That is, edges in the graph correspond to implications (since $\{\neg l, l'\}$ is equivalent to $l \rightarrow l'$). Since implication is transitive, we have that $l_1 \rightarrow l_2$ is implied by the formula whenever there is a path in the graph between node l_1 and node l_2. In particular, if we have a path between l_1 and $\neg l_1$, we know that l_1 cannot hold. Therefore, $\neg l_1$ is implied by the formula. If, in addition, we have a path from $\neg l_1$ to l_1, then neither l_1 nor $\neg l_1$ can hold, and so the formula is unsatisfiable. Finally, we know that in every satisfying truth assignment, if l is assigned *true* then any literal implied by l, i.e., any descendant of l in the graph, must be assigned *true* as well.

Instance	% Binary Clauses
log-dir.a	49%
log-dir.b	55%
log-dir.c	55%
log.d	80%
log-gp.a	98%
log-gp.b	98%
log-gp.c	98%
log-un.a	98%
log-un.b	98%
log-un.c	99%
bw-dir.a	70%
bw-dir.b	71%
bw-dir.c	74%
bw-dir.d	78%

Figure 1: Percentage of Binary Clauses in SAT-Encoded Planning Problems

2.3 Planning As Satisfiability

The planning problem is defined as follows: given a description of an initial state, a goal state, and a set of operators (=Actions) for changing the state of the world, find a sequence of operators that, when applied in the initial state, yield the goal state. An important development in planning algorithms was Kautz and Selman's *planning as satisfiability* approach [Kautz and Selman, 1996]. Kautz and Selman showed that by reducing planning problems to satisfiability problems, we can often solve them more quickly than by using standard planning algorithms. Planning problems can be encoded as satisfiability problems in a number of ways (e.g., see [Ernst *et al.*, 1997] for a description and analysis of some of these methods).

As [Brafman, 1999] points outs, encoded planning problems contain a large number of binary clauses. In Figure 1 we show this for a number of instances of SAT-encoded planning problems.[1] This is no accidental phenomenon. Close inspection of the types of constraints expressed within encoded planning problems makes it apparent that many classes of these constraints generate binary formulas. For example,

[1] Available at ftp://ftp.research.att.com/dist/ai/logistics/tar.Z and satplan.data.tar.Z

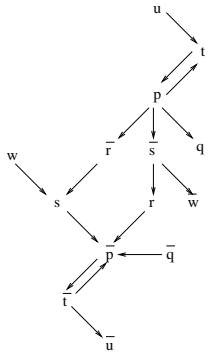


Figure 2: The Implication Graph

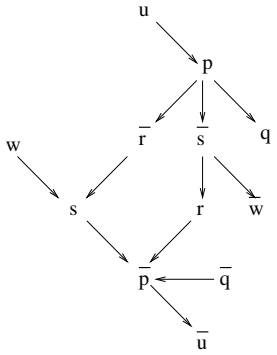


Figure 3: Removing Strongly Connected Components

the constraint that if an action is executed at some time point then all its preconditions must hold prior, produces binary clauses. Similarly, the constraint asserting that if an action is executed at some point then all its effects must hold after the execution yields binary clauses as well. In the encoding used by the BLACKBOX planner [Kautz and Selman, 1999] mutual exclusion constraints (on actions and on state variables) play a prominent role. These constraints are expressed using binary clauses as well.

Interestingly, it turns out that the SAT encodings of other important problems exhibit the same large percentage of binary clauses. These include test-pattern generation for circuits [Larrabee, 1992] and bounded model checking [Shtrichman, 2000].

3 The 2-SIMPLIFY PreProcessor

We now explain the algorithm implemented by the 2-SIMPLIFY preprocessor using the following formula:

$$\{\neg p, q\}, \{\neg p, \neg r\}, \{r, s\}, \{\neg w, s\}, \{\neg p, t\}, \{\neg t, p\}, \{\neg u, t\}$$
$$\{\neg p, \neg s\}\{\neg p, s, q\}, \{\neg q, \neg s, p\}, \{u, \neg w, q\}, \{\neg q, \neg r, s\}$$
$$\{s, v, \neg m\}$$

(1) *Construct Implication Graph.* A graph containing all literals in the language is constructed with directed edges from l to l' if $\{\neg l, l'\}$ is a binary clause. Figure 2 shows the implication graph for the formula above.

(2) *Collapse Strongly Connected Components.* A subgraph in which there is a path between every pair of nodes is called a strongly connected component (SCC). When a path from node l to node l' exists, we know that $l \rightarrow l'$ is a consequence of our formula. Therefore, all nodes within an SCC imply each other, and they must all be assigned the same value.

Once we discover an SCC we replace it by a single node. The children of this node are the children of the nodes in the SCC, and the parents of this node are the parents of the nodes in this SCC. In addition, all literals in the SCC must be replaced by this new node within all non-binary clauses.

In our example, the nodes t and p, and the nodes $\neg t$ and $\neg p$, form strongly connected components. We choose p to represent the first SCC and we choose $\neg p$ to represent the second SCC. The reduced graph is shown in Figure 3.

(3) *Generate Transitive Closure.* We generate the transitive closure. Now, we know that if l' is a child of l then $l \rightarrow l'$ is implied by the original formula. We can deduce l if either:

1. for some proposition p, both p and $\neg p$ are children of $\neg l$.
2. l is a child of $\neg l$.

Once we deduce l, we can perform unit propagation: We know that all children of l are true, and we can reduce the current formula by applying unit propagation. If the reduced formula contains new binary clauses, we add the appropriate edges to the graph and update the transitive closure.

In Figure 4 we can see the effect of this step. First, we compute the transitive closure of the current graph, shown in Figure 4A. In this graph, we see that u has $\neg u$ as a descendant and that p has $\neg p$ as a descendant. Therefore, we conclude that p and u must be assigned the value *false*. We can remove nodes that correspond to assigned propositions (i.e., $p, u, \neg p, \neg u$ in our case). The resulting graph is shown in Figure 4B. Next, we perform unit propagation, and our initial ternary clauses: $\{\neg p, s, q\}, \{\neg q, \neg s, p\}, \{u, \neg w, q\}, \{\neg q, \neg r, s\}$ are reduced to $\{\neg q, \neg s\}, \{\neg w, q\}, \{\neg q, \neg r, s\}$. The first clause was remove because it is satisfied, and a (false) literal was removed from the next two clauses. Since we have new binary clauses, we can update the graph, as shown in Figure 4C, making sure it is transitively closed. In the resulting graph, $\neg w$ is a child of w, and we can deduce that $w =$ *false*. The reduced graph is shown in Figure 4D.

(4) *Derive Shared Implications.* Let $\{l_1, \ldots, l_k\}$ be some non-binary clause in the formula. Let L_i be the set of literals implied by l_i for $i = 1, \ldots, k$. Let $L = L_1 \cap \cdots \cap L_k$. All literals in L are consequences of our formula, and we can use them to perform unit propagation.

Consider the clause $\{\neg q, \neg r, s\}$, the sets of literals implied by each of the literals in this clause are $(\neg q), (\neg r, s, \neg q), (s, \neg q)$, respectively. Their intersection contains $\neg q$. Hence, we can deduce that q is *false*.

(5) *Compute Transitive Reduction.* The transitive reduction of a graph G is a graph G' with the same nodes as G but with a minimal set of edges such that a path between l and l' exists in G iff a path between l and l' exists in G'.

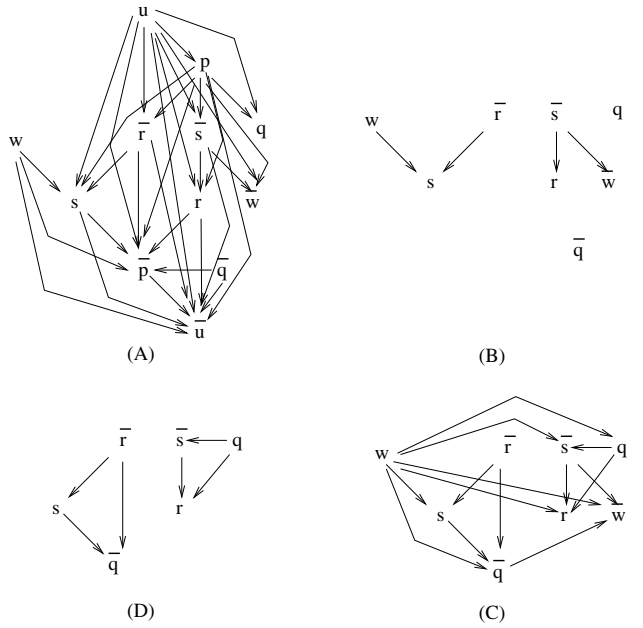


Figure 4: (A) Initial Transitive Closure. (B) Removal of Assigned Nodes. (C) Update with New Binary Clauses. (D) Removal of Assigned Nodes.

(6) *Output Simplified Formula.* We output a formula whose clauses consist of the non-binary clauses remaining after all simplifications were performed and all binary clauses corresponding to edges in the transitive reduction of the graph. Given the assignments deduced so far and the mappings between elements of strongly connected components, the simplified formula is equivalent to the original formula. For example, the output for our original formula will be: $\{r, s\}, \{s, v, \neg m\}$ together with the partial assignment $u =$ *false*, $p =$ *false*, $w =$ *false*, $q =$ *false*.

Step (4) is a novel implementation of an old technique (hyper-resolution [Robinson, 1965]) and step (5) is new. Both have important impact on 2-SIMPLIFY's performance. The *Derive Shared Literals* step enhances the ability of 2-SIMPLIFY to derive unit literals. In some cases, it can derive hundreds of new unit literals quickly. In fact, 2-SIMPLIFY uses a more sophisticated version of this procedure: if no shared unit literals exist, we attempt to derive new binary clauses by intersecting the implications of all literals but one. These binary clauses are then added to the implication graph. The *Compute Transitive Reduction* step leads to a minimal sufficient set of binary clauses, leading to smaller and simpler formulas. We have found this reduction to have an important positive influence on systematic solvers.

4 Experimental Evaluation

We evaluated the 2-SIMPLIFY preprocessor on a set of benchmark instances of encoded planning problems which were used to test the REL-SAT solver. 2-SIMPLIFY is written in C++ and all experiments described here were conducted on a DELL Latitude CPi notebook with a Pentium II-400 processor with 64MB RAM running LINUX. All time measurements refer to CPU time.

Instance	Simp. Time	Assigned/Total
log-dir.a	0.02	152/828
log-dir.b	0.02	152/843
log-dir.c	0.04	186/1141
log.d	2.4	753/4713
log-gp.a	0.19	148/1782
log-gp.b	0.31	169/2069
log-gp.c	0.52	191/2809
log-un.a	0.13	160/1415
log-un.b	0.21	161/1729
log-un.c	0.37	179/2353
bw-dir.a	0.08	173/459
bw-dir.b	0.2	351/1087
bw-dir.c	0.96	824/3016
bw-dir.d	3.86	1658/6325

Figure 5: Running time and deduction power of 2-SIMPLIFY

Instance	SATZ	2-Simplify	SATZ on 2S	Total
log-dir.a	**108.03**	0.02	0.51	**0.53**
log-dir.b	**0.4**	0.02	0.38	**0.4**
log-dir.c	**3.23**	0.04	1.11	**1.15**
log.d	**1636.7**	2.4	1.41	**3.81**
log-gp.a	**0.95**	0.19	0.17	**0.36**
log-gp.b	**2.82**	0.31	0.6	**0.91**
log-gp.c	**4.08**	0.52	0.8	**1.32**
log-un.a	**0.5**	0.13	0.09	**0.22**
log-un.b	**UA**	0.21	UA	**UA**
log-un.c	**UA**	0.37	UA	**UA**
bw-dir.a	**0.2**	0.08	0.08	**0.16**
bw-dir.b	**0.7**	0.2	0.32	**0.52**
bw-dir.c	**3.27**	0.96	1.79	**2.75**
bw-dir.d	**1061.83**	3.86	300.72	**304.58**

Figure 6: Solution times for SATZ and 2-SIMPLIFY+SATZ.

First, we examined 2-SIMPLIFY's ability to deduce unit literals. In Figure 5 we show 2-SIMPLIFY's running time on each of the instances and the number of variables it was able to assign. Note that in addition to deducing unit literals, 2-SIMPLIFY also supplies additional important information in the form of equivalent literals.

To assess the utility of 2-SIMPLIFY we generated simplified formulas for each of the instances and compared the solution time of the original formulas with the combined simplification and solution times for the simplified formulas. We performed this comparison using two systematic solvers: SATZ and REL-SAT. The results for SATZ are shown in Figure 6, where we show the simplification time again, in order to give a sense of its magnitude in comparison to solution time. We see that in *all* instances 2-SIMPLIFY proves itself useful, reducing the overall solution time. 2-SIMPLIFY is particularly useful on the instances on which SATZ takes longest.

The results for REL-SAT are shown in Figure 7. We note that in many cases, REL-SAT's performance on the simplified formulas can be improved by disabling its own preprocessor.

Instance	REL-SAT	2-SIMPLIFY	2S+R
log-dir.a	**0.46**	0.02	**0.41**
log-dir.b	**0.47**	0.02	**0.45**
log-dir.c	**1.04**	0.04	**0.8**
log.d	**9.64**	2.4	**9.47**
log-gp.a	**1.43**	0.19	**0.86**
log-gp.b	**2.22**	0.31	**1.56**
log-gp.c	**3.35**	0.52	**2.93**
log-un.a	**0.57**	0.13	**0.33**
log-un.b	**0.57**	0.13	**0.33**
log-un.b	**0.57**	0.13	**0.33**
log-un.b	**12.21**	0.21	**6.74**
log-un.c	**23.14**	0.37	**12.09**
bw-dir.a	**0.25**	0.08	**0.18**
bw-dir.b	**1.58**	0.2	**1.1**
bw-dir.c	**29.05**	0.96	**18.6**
bw-dir.d	**>1040**	3.86	**528.73**

Figure 7: Solution times for REL-SAT, 2-SIMPLIFY, and 2-SIMPLIFY+REL-SAT.

(We refer the reader to the full paper where these experiments appear.) We see that 2-SIMPLIFY+REL-SAT is always faster than REL-SAT alone when we use REL-SAT's preprocessor. The improvement is especially significant in the hardest instances. We note that the REL-SAT figures represent average running times (because REL-SAT has a stochastic element).[2]

We run another sequence of experiments to compare 2-SIMPLIFY with Crawford's COMPACT simplifier. First, we examined the performance of various COMPACT options on the test problems and found that the best performance is obtained almost always using either no flags or using the psl flags.[3] To see whether 2-SIMPLIFY improves on COMPACT's performance we compare the sum of simplification time and solution time for combinations of COMPACT with and without 2-SIMPLIFY and with either SATZ or REL-SAT.

In Figure 8 we see the results for SATZ. The columns correspond to the combined running time of SATZ and the simplification algorithms on each of the instances. The first column is SATZ applied to COMPACT simplified formulas, while the second is SATZ applied to 2-SIMPLIFY+COMPACT. The third and forth columns are similar, except that we used the psl options in COMPACT. We see that 2-SIMPLIFY leads to reduced running times in all cases except two (log-dir.a and log.d on compact with the psl options).

In Figure 9 we show the corresponding results for REL-SAT (with its preprocessor). Here we see that 2-SIMPLIFY improves the overall performance on *all* problems, and in some cases significantly so.

Finally, we examined 2-SIMPLIFY's influence on the performance of WALKSAT, a stochastic solver. As noted, stochastic solvers require tuning, and we tried to find the best parameters in each case. As Figure 10 shows, the results are mixed. On the *log* instances, we get 2-6 fold improvement, except for the log.d. In this instance, the simplified formula is solved faster, but simplification time is larger than solution time. However, on the *bw-dir* instances we get significant reduction in performance. If 2-SIMPLIFY ignores the transitive-reduction step (i.e., we maintain many binary clauses), we get somewhat different results (shown in the last column). These mixed result are not surprising as stochastic methods are know to be quite sensitive to the form of the formula.

$Instance$	C+S	2S+C+S	psl + S	psl + 2S+ S
bw-dir.a	0.29	0.25	0.16	0.16
bw-dir.b	0.84	0.67	1.0	0.89
bw-dir.c	3.39	3.32	6.30	4.69
bw-dir.d	686.66	302.47	123.33	77.31
log-dir.a	1.17	0.77	2.29	13.35
log-dir.b	0.52	0.53	0.64	0.41
log-dir.c	1.32	3.94	398.33	3.23
log.d	454.76	4.20	4.06	5.39
log-gp.a	1.02	0.48	5.82	1.60
log-gp.b	1.59	0.76	10.24	2.25
log-gp.c	3.66	1.13	22.82	13.82
log-un.a	0.62	0.32	3.18	0.88
log-un.b	-	-	12.70	3.62
log-un.c	-	-	104.48	21.00

Figure 8: Simplification with COMPACT and 2-SIMPLIFY, Solution with SATZ.

$Instance$	C+R	2S+C+R	psl + R	2S+ psl + R
bw-dir.a	0.38	0.16	0.16	0.16
bw-dir.b	1.55	1.24	1.47	1.38
bw-dir.c	25.93	18,47	28.33	17.64
bw-dir.d	-	608.27	-	510.59
log-dir.a	0.45	0.52	0.56	0.38
log-dir.b	0.80	0.59	0.71	0.49
log-dir.c	1.43	1.13	1.21	0.79
log.d	10.96	9.33	9.32	8.63
log-gp.a	1.62	0.98	6.56	1.80
log-gp.b	2.40	1.54	12.16	3.01
log-gp.c	4.46	3.23	27.25	5.75
log-un.a	0.82	0.44	3.35	0.85
log-un.b	7.64	6.81	9.80	3.95
log-un.c	18.92	13.09	27.48	12.98

Figure 9: Simplification with COMPACT and 2-SIMPLIFY, Solution with REL-SAT.

5 Conclusion and Related Work

SAT instances with many binary clauses arise naturally in a number of important applications. The abundance of binary clauses in such problems can be exploited using 2-SAT solution methods and other specialized inference algorithms. Here, we presented 2-SIMPLIFY, a principled and efficient simplification algorithm that uses the transitive closure of the

[2] In the case of bw-dir.d, REL-SAT timed out on the original problem in some of the iterations and we provided a lower bound on its average running time.

[3] With no flags, COMPACT does unit resolution, removes satisfied clauses, and renames variables to be contiguous. With psl, in addition to the above, COMPACT eliminates pure literals, resolves away variables with a single occurrence, and for each literal checks whether its addition leads to contradiction.

Instance	WALKSAT	2S+WALKSAT	2S-TR+WALKSAT
bw-dir.a	0.06	0.20	0.16
bw-dir.b	4.13	79.55	7.97
bw-dir.c	39.26	—	17.79
bw-dir.d	94.47	—	160.61
log-dir.a	0.25	0.17	0.13
log-dir.b	0.45	0.0.26	0.17
log-dir.c	1.01	0.37	0.25
log.d	1.15	2.76	10.35
log-gp.a	8.20	1.20	4.24
log-gp.b	7.19	3.23	29.36
log-gp.c	20.34	7.95	72.26

Figure 10: Solution times using WALKSAT.

implication graph together with a novel implementation of hyper-resolution (i.e, the *derive shared literals* step) and *transitive reduction* to obtain a smaller equivalent formula. This leads to an approach that is faster, more powerful, and more efficient the ad-hoc resolution of binary clauses used in [Brafman, 1999]. Our experiments on a set of encoded planning problems show that 2-SIMPLIFY is beneficial in conjunction with systematic solution algorithms: in virtually all tested experiments shorter solution times were obtained. In conjunction with a stochastic solver, the results were mixed, and the utility depends on the problem instance.

We are not the first to utilize binary resolution in this area. Larrabee used the implication graph to devise a SAT algorithm in the context of test-pattern generation [Larrabee, 1992]. Larrabee systematically generates satisfying assignments consistent with the implication graph. Any assignment that satisfies the non-binary clauses is a satisfying assignment for the whole formula. This method exploits the binary portion of the formula, but it does not utilize the power of contemporary variable ordering and search techniques.

2CL [Van Gelder and Tsuji, 1996] is a solver based on the Davis-Putnam-Logemann-Loveland algorithm [Davis *et al.*, 1962]. At each branch point, 2CL constructs the transitive-closure of the current implication graph and uses it to choose the next branching variable. Thus, 2CL is a dynamic extension of a key aspect of 2-SIMPLIFY. We have yet to experiment with 2CL. However, our initial attempt to produce a dynamic version of 2-SIMPLIFY along similar lines were not competitive with SATZ for two reasons: Maintaining an updated transitive closure is costly in terms of time and memory, and SATZ seems to obtain better information through its use of unit propagation, and faster. At this point, it seems that the techniques that 2-SIMPLIFY utilizes (i.e., implication graph analysis, restricted hyper-resolution, and transitive reduction) provide a basis for a good simplifier, but not necessarily a good systematic solver.

Acknowledgments: I am grateful to Yefim Dinitz for his help and advice on graph algorithms and for important comments on previous versions of this paper. This work was supported in part by the Paul Ivanier Center for Robotics and Production Management.

References

[Aspvall *et al.*, 1979] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[Bayardo and Schrag, 1997] R. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. AAAI-97*, pages 203–208, 1997.

[Brafman, 1999] R. I. Brafman. Reachability, relevance, resolution, and the planning as satisfiability approach. In *IJCAI'99*, pages 976–981, 1999.

[Cook, 1971] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*. ACM, 1971.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communication of the ACM*, 5(7):394–397, July 1962.

[Ernst *et al.*, 1997] M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1997.

[Gomes *et al.*, 1998] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of 15th Nat. Conf. AI*, pages 431–437, 1998.

[Kautz and Selman, 1996] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. of the 13th National Conference on AI (AAAI'96)*, pages 1194–1201, 1996.

[Kautz and Selman, 1999] H. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *Proc. 16th Intl. Joint Conf. on AI (IJCAI'99)*, pages 318–325, 1999.

[Larrabee, 1992] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, pages 4–15, January 1992.

[Li and Anbulagan, 1997] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. IJCAI-97*, 1997.

[Robinson, 1965] J. A. Robinson. Automatic deduction with hyper-resolution. *Int. J. of Com. Math*, 1:227–234, 1965.

[Selman *et al.*, 1992] B. Selman, H. J. Levesque, and D. Mitchell. Gsat: A new method for solving hard satisfiability problems. In *Proc. of the 10th National Conf. on AI (AAAI '92)*, pages 440–446, 1992.

[Selman *et al.*, 1994] Bart Selman, Henry A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. Nat. Conf. on AI*, pages 337–343, 1994.

[Shtrichman, 2000] O. Shtrichman. Tuning sat checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification 2000*, 2000.

[Van Gelder and Tsuji, 1996] A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. American Mathematical Society, 1996.