Faster Conflict Generation for Dynamic Controllability

Nikhil Bhargava, Tiago Vaquero, Brian Williams
Massachusetts Institute of Technology
{nkb, tvaquero, williams}@mit.edu

Abstract

In this paper, we focus on speeding up the temporal plan relaxation problem for dynamically controllable systems. We take a look at the current best-known algorithm for determining dynamic controllability and augment it to efficiently generate conflicts when the network is deemed uncontrollable. Our work preserves the $O(n^3)$ runtime of the best available dynamic controllability checker and improves on the previous best runtime of $O(n^4)$ for extracting dynamic controllability conflicts. We then turn our attention to temporal plan relaxation tasks and show how we can leverage our work on conflicts and the structure of the network to efficiently make incremental updates intended to restore dynamic controllability by relaxing constraints. Our new algorithm, RELAXIDC, has the same asymptotic runtime as previous algorithms but sees dramatic empirical improvements over the course of repeated dynamic controllability checks.

1 Introduction

In temporal planning, an agent specifies a set of temporal constraints between events and determines whether those constraints can all be satisfied. When the exact durations between events cannot be specified ahead of time, the agent is interested in knowing if a valid execution strategy exists for this set of constraints, or if the problem is dynamically controllable.

In temporal plan relaxation tasks, the original plan specification is overconstrained, and the agent must find a way to relax certain constraints so that the plan becomes dynamically controllable. For constraints whose durations can be chosen by the agent, this means expanding the range of allowed durations. For constraints whose durations cannot be specified, an agent can relax the constraint by shrinking its bounds and accepting some small risk that the outputted plan is invalid [Fang *et al.*, 2014]. In this paper, we focus on improving the problem of computing temporal plan relaxations. We provide two main contributions.

First, we more efficiently reduce the available search space for plan relaxation. Temporal plan relaxation at its core is a generate-and-test search problem. When a proposed plan is shown to be uncontrollable, the next step is to modify some constraints in hopes of finding one that remedies the underlying issue. Conflict-directed search is a generalized technique for efficiently pruning the remaining search space, but to work it needs a way to extract conflicts from an infeasible problem that explains why the problem was infeasible [Williams and Ragno, 2007]. To that end, we augment the best known $O(n^3)$ dynamic controllability checking algorithm [Morris, 2014] to output conflicts to be used in conflict-directed search. This represents an improvement over the $O(n^4)$ runtime for the best available algorithm for generating conflicts for dynamic controllability [Yu *et al.*, 2014].

Second, we make successive dynamic controllability checks faster by leveraging structural similarities. In the search for an acceptable relaxation of a temporal plan, we have to re-check dynamic controllability for every generated candidate until we find a plan that is controllable. A plan that has undergone the relaxation of a single constraint is still nearly identical to the original. To take advantage of this, we developed an incremental dynamic controllability checker, RELAXIDC, for use in temporal relaxation problems. While the incremental approach does not improve on the asymptotic worst-case runtime of $O(n^3)$, empirically it runs much faster than the non-incremental algorithm.

Together these developments represent significant progress towards improving and scaling temporal plan relaxation problems. The natural follow-up question is whether these improvements can be extended to other algorithms that focus on searching the temporal plan space by tightening constraints, such as FastIDC [Stedl and Williams, 2005; Shah *et al.*, 2007], EfficientIDC [Nilsson *et al.*, 2013], and EIDC2 [Nilsson *et al.*, 2014]. While we believe the answer is yes, we leave this as a separate question to consider in future work.

2 Background

The input to a dynamic controllability checker is a Simple Temporal Network with Uncertainty (STNU), first defined by [Vidal, 1999]. An STNU is a graph composed of nodes, which correspond to timepoints, and links which correspond to temporal constraints between those timepoints. To model events with uncertain duration, we distinguish between the different types of temporal constraints induced on the network. *Requirement constraints* are constraints where the exact duration of the constraint can be picked by the agent to be anything within the specified bounds. *Contingent constraints* are constraints whose true value cannot be assigned ahead of time. We say that the links derived from these constraints are requirement and contingent links, respectively.

For convenience, we will also assume that all STNUs are in *normal form*, i.e. all contingent links have a lower bound of zero. For any contingent link $A \rightarrow B$ with lower bound $l \neq 0$ and upper bound u, it's easy to see that we can create an equivalent STNU by introducing a new timepoint B', such that there is a requirement link $A \rightarrow B'$ with lower and upper bounds l and a contingent link $B' \rightarrow B$ with lower bound 0 and upper bound $u - l$.

Informally, we say an STNU is *dynamically controllable* if it is possible to just-in-time assign values to timepoints given knowledge about assignments to all timepoints that happened in the past. Intuitively if an STNU is dynamically controllable, an agent could improvise as it goes, making plans based on the actual durations of observed contingent constraints and ensuring that all constraints are satisfied.

Given an STNU, we often find it easier to analyze its controllability by first transforming it into a *labeled distance graph* [Morris, 2006]. Each timepoint of the STNU corresponds to a node in the labeled distance graph, and for each link $A \rightarrow B$ with lower bound l and upper bound u, we add an edge $A \rightarrow B$ with weight u and an edge $B \rightarrow A$ with weight $-l$. Finally, for each contingent link we additionally add an edge $A \rightarrow B$ with lower-case label b and weight l and an edge $B \rightarrow A$ with upper-case label B and weight $-u$. Because STNUs are in normal form, contingent links have a lower bound of 0, and we can safely assume that all lower-case edges have weight 0.

An unlabeled edge in a labeled distance graph corresponds to the maximum time that can elapse between the corresponding timepoints (with a negative weight implying that the end node of an edge must occur before the start node). An upper-case edge denotes the maximum time that would elapse between two events if the associated contingent link were to take on the value of its upper bound, and a lower-case edge denotes the maximum time that would elapse between two events if the associated contingent link were to take on the value of its lower bound. Morris [2006] introduces a series of reductions that can be applied to a labeled distance graph that preserve these relations and extends them to derive new constraints beyond those originally specified in the STNU. These were later updated in [Morris, 2014] and are reproduced below:

- *Upper case reduction*: With edges $A \xrightarrow{x} B \xrightarrow{C:y} D$, produce edge $A \xrightarrow{C:(x+y)} D$.
- *Lower case reduction*: With edges $A \xrightarrow{b:0} B \xrightarrow{y} C$, if $y < 0$, produce edge $A \xrightarrow{y} C$.
- *Cross case reduction*: With edges $A \xrightarrow{b:0} B \xrightarrow{C:y} D$, if $y < 0, B \neq C$, produce $A \xrightarrow{C:y} D$.
- *No-case reduction*: With edges $A \xrightarrow{x} B \xrightarrow{y} C$, produce $A \xrightarrow{x+y} C$.
- *Label removal*: With edge $A \xrightarrow{B:x} C$, if $x \geq 0$, produce $A \xrightarrow{x} C$.

More detail justifying these reductions can be found in [Morris and Muscettola, 2005; Morris, 2006]. The repeated application of these reductions and a search for inconsistencies form the basis for checking dynamic controllability.

3 Extracting Conflicts

When specifying a set of temporal constraints, the agent often has some leeway with which it can relax constraints that are overly constrictive. If a system is found not to be dynamically controllable, it's useful to know which specific constraints caused the system to be uncontrollable. Here we introduce the notion of a *conflict*, which serves as a certificate explaining why a problem is infeasible.

```
Input: A labeled distance graph, $G = \langle V, E \rangle$
Output: Whether the STNU derived from the distance graph
        is dynamically controllable and if not, a set of
        conflicts
Initialization:
1  $negNodes \leftarrow$ the set of all vertices with incoming
   negative edges;
2  $novel \leftarrow []$; list of newly added edges;
3  $preds \leftarrow \{\}$; mapping of function call to predecessor
   list;
DCCHECKER:
4  for $v \in negNodes$ do
5      $cycleFree?, edges \leftarrow$DCDIJKSTRA$(G, v, preds,$
         $novel, [v], negNodes)$;
6      if $!cycleFree?$ then
7          return $false$, EXTRACTCONFLICTS$(edges,$
             $novel, preds)$
8  return $true, \emptyset$
```
Algorithm 1: Dynamic Controllability algorithm that reports conflicts

To make the output of our dynamic controllability algorithm more useful, we introduce a variant of the dynamic controllability detection algorithm (Algorithm 1) that is capable of reporting conflicts. The best algorithm known for computing dynamic controllability takes time $O(n^3)$ and works by trying to find a semi-reducible negative cycle [Morris, 2014]. A *semi-reducible negative cycle* is a cycle in a labeled distance graph with negative weight, where all lower-case edges can be eliminated through a series of reductions; an STNU is dynamically controllable iff its labeled distance graph does not have a semi-reducible negative cycle [Morris, 2006]. Accordingly in the event of failure, our algorithm derives the conflict by extracting the set of edges that compose a semi-reducible negative cycle and suggesting modifications that would eliminate it.

Briefly, the original algorithm works by walking the graph in reverse (traversing edges in the opposite direction) and attempting to show that every walk starting from a negative weight edge that follows a semi-reducible path eventually takes on a non-negative weight. It walks the graph efficiently by emulating Dijkstra's algorithm until it finds a set

```
Input: Labeled distance graph, $G = \langle V, E \rangle$, start node $s$, list
       of predecessor edges $preds$, list of new edges $novel$,
       $callStack$, and negative nodes $negNodes$
Output: Whether the current walk is cycle-free, and the
        edges composing a semi-reducible negative cycle
Initialization:
$Q \leftarrow PriorityQueue()$;
$labelDist \leftarrow []$; shortest distances for labeled path;
$unlabelDist \leftarrow []$; shortest distances for unlabeled path;
$labelDist[s] \leftarrow \langle 0, \emptyset \rangle$;
$unlabelDist[s] \leftarrow \langle 0, \emptyset \rangle$;
for $e \in s.incomingEdges()$ where $e.weight < 0$ do
    $Q.add(\langle e.from, e.label \rangle, e.weight)$;
    $(e.label == \emptyset\ ?\ unlabelDist : labelDist)[e.from]$
      $\leftarrow \langle e.weight, e \rangle$
DCDijkstra:
if $s \in callStack[1 : end]$ then
    return $false, \emptyset, s$
$preds[s] \leftarrow \langle labelDist, unlabelDist \rangle$;
while $Q.size() > 0$ do
    $weight, label, v \leftarrow Q.pop()$;
    if $weight \geq 0$ then
        $G.add(\langle v, s, weight \rangle)$;
        $novel.add(\langle v, s, weight \rangle)$;
        $continue$;
    if $v \in negNodes$ then
        $newStack \leftarrow [v].concat(callStack)$;
        $result, edges, end \leftarrow$ DCDIJKSTRA($G$, $v$,
          $preds$, $novel$, $newStack$, $negNodes$);
        if $!result$ then
            if $end \neq \emptyset$ then
                $edges.add($EXTRACTEDGEPATH($s$, $v$,
                  $labelDist$, $unlabelDist$));
            if $end == s$ then
                $end \leftarrow \emptyset$
            return $false, edges, end$
    for $e \in v.incomingEdges()$ do
        if $e.weight \geq 0$ and $(!e.isLowerCase()$ or
          $e.label \neq label)$ then
            $w \leftarrow e.weight + weight$;
            $distArray \leftarrow label \neq \emptyset\ ?$
              $labelDist : unlabelDist$;
            if $distArray[e.from] == \emptyset$ or
              $w < distArray[e.from][0]$ then
                $distArray[e.from] \leftarrow \langle w, e \rangle$;
                $Q.addOrDecKey(\langle e.from, label \rangle, w)$;
$negNodes.remove(s)$;
return $true, \emptyset, \emptyset$
```

Algorithm 2: Function DCDIJKSTRA

of shortest paths to other nodes that are positive. Because Dijkstra's algorithm cannot handle negative weights besides those connected to the start node, whenever a negative edge is discovered, the algorithm recursively calls itself and begins a new walk from the node connected to the newly discovered negative edge. A function call that completes successfully will add edges corresponding to the discovered positive semi-reducible paths, so after the recursive call we can safely ignore any negative edges and continue the walk. If the recursive calls form a cycle, then we've found a negative semi-reducible cycle.

In order to return a semi-reducible negative cycle, we make a few modifications to the subroutine DCDIJKSTRA (Algorithm 2). In particular, while we incrementally build up our set of shortest paths, we also record the set of backwards edges (for both labeled and unlabeled paths) that compose those shortest paths. When we detect a cycle of recursive calls (line 8 of Algorithm 2), we know we have a semi-reducible negative cycle and can unwind the call stack, recording the edges that compose the shortest path. We later translate those edges back to the original STNU constraints using EXTRACT-CONFLICTS (line 7 of Algorithm 1).

The remaining question is how to handle edges that were newly added as a byproduct of running the algorithm. To do so, we use the maintained list of new edges, *novel*. When an edge is determined to have not originated from the STNU, we can use our list of recorded predecessor edges for a previous shortest path calculation of DCDIJKSTRA to recursively expand that edge until we end with a path of edges that all map to STNU constraints.

After generating the edges of the semi-reducible negative cycle, we need an encoding that suggests how to eliminate it. The first way to eliminate a semi-reducible negative cycle is by increasing edge weights to make the cycle non-negative. For a cycle C, this can be modeled by $\sum_{e \in C} w_e \geq 0$. The second way to eliminate a semi-reducible negative cycle is to adjust some constraints so some lower-case edges cannot be eliminated by a series of reductions. The only way to eliminate a lower-case edge is to follow it with a series of edges whose combined weights are negative. Thus, if we take the shortest path P that follows a lower-case edge and has combined negative weight, we can model a new constraint in the conflict by $\sum_{e \in P} w_e \geq 0$. If any one of these newly created constraints is satisfied by a relaxation of the initial constraints, then we say our conflict is resolved.

Introducing conflict generation doesn't damage the $O(n^3)$ runtime guarantee. The maintenance of the new data structures occurs a constant time overhead. The EXTRACT-EDGEPATH function adds at most $O(n)$ edges per call and is called at most $O(n)$ times throughout the stack unwinding; this limits the additional work to $O(n^2)$, which is dominated by the normal algorithm runtime. This represents a strong improvement over the previous best runtime for generating dynamic controllability conflicts, which took $O(n^4)$ time.

4 Faster Incremental Updates

With a newly reported conflict explaining why an STNU is uncontrollable, a planner will want to try several different relaxations to generate an acceptable, controllable network. This might be because resolving one conflict reveals another inconsistency in the network or because an end-user may want to try several different relaxations in search for the most preferable one.

Despite the fact that when modifying a single constraint most of the graph remains the same, existing algorithms fail to leverage those structural similarities when trying to determine if the change yields a controllable network. To speed up subsequent controllability checks, we introduce a set of data structures, used by our new algorithm RELAXIDC (Algorithm 4), that efficiently encode relationships on the labeled distance graph and allow subsequent runs of the algorithm to skip over portions of the STNU that are known to remain the same. While these changes don't improve the asymptotic worst-case runtime of checking dynamic controllability, they do yield empirically faster runtimes.

4.1 Search

To illustrate the importance of RELAXIDC, we provide a simple conflict-directed search algorithm (RELAXSEARCH, Algorithm 3) that iteratively relaxes an over-constrainted problem in search of a feasible solution that resembles the original set of inputs. The goal of RELAXSEARCH is to take an input STNU and find some set of relaxations to constraints that ensure that the final STNU is dynamically controllable.

```
Input: $G$, the labeled distance graph
Initialization:
1  $negNodes \leftarrow$ the set of all vertices with incoming
   negative edges;
2  $novel \leftarrow []$; list of newly added edges;
3  $preds \leftarrow \{\}$; mapping of function call to predecessor
   list;
4  $dag \leftarrow$ an empty graph;
5  $relaxations \leftarrow []$;
RELAXSEARCH:
6  $controllable, conflict \leftarrow$ DCCHECKER($G$, $negNodes$,
     $novel$, $preds$);
7  while !$controllable$ do
8      $relaxedEdge, weight \leftarrow$ RELAX($G, conflict$);
9      $relaxations.push(\langle relaxedEdge, weight\rangle)$;
10     $controllable, conflict \leftarrow$ RELAXIDC($G$,
         $\langle relaxedEdge, weight\rangle$, $negNodes$, $novel$,
         $preds$)
11 return $relaxations$
```

Algorithm 3: RELAXSEARCH - an algorithm that relaxes constraints in an STNU until a dynamically controllable solution is found.

The algorithm makes an initial call to DCCHECKER to check for controllability and initialize the relevant data structures. If the output is not controllable, it uses the conflicts to generate relaxed edges (line 8) before re-executing the controllability check. Our choice of relaxation strategy does not matter so long as we resolve the generated conflict. In this paper, our relaxation strategy involves taking a random edge from the conflict and modifying its weight so the conflict's semi-reducible cycle becomes positive, but we could have just as easily used a cost function or some additional set of constraints to refine our conflict resolution strategy.

It's worth noting that the decision to use RELAXIDC instead of DCCHECKER at line 10 has no impact on the search algorithm's correctness. However, as we'll show later, this decision is what allows us to speed up our search dramatically.

4.2 Auxiliary Data Structures

An important observation is that in a dynamically controllable STNU, the recursive calls to DCDIJKSTRA represent an acyclic dependency tree. Only when the recursive calls form a cycle do we report that the network is uncontrollable. As we traverse the graph, we can build an explicit dependency directed acyclic graph (DAG), where we draw a link from a parent call to a child call only after the child call returns successfully. For convenience, we represent a function call by the associated starting node parameter.

This DAG will allow us to avoid superfluous computation in subsequent dynamic controllability checks when we're only interested in relaxing the value of a single constraint. We perform a topological sort on the nodes of the DAG requiring children to come before their parents to get the order in which we should re-process the STNU.

```
Input: $G$, the labeled distance graph; $\langle e, w\rangle$, the edge to be
       modified and its new weight; $preds$, a map from
       recursive call to predecessor list; $novel$, list of edges
       that were newly added by DCCHECKER; $negNodes$,
       list of nodes that still have negative incoming edges;
       $dag$, DAG of recursive calls
Initialization:
1  $sortedNodes \leftarrow$ TOPOLOGICALSORT($dag$);
2  $changed \leftarrow new\ Set()$;
RELAXIDC:
3  for $v \in sortedNodes$ do
4      if $e == preds[e.labeled?][e.start]$ or
         $dag.children(v) \cap changed \neq \emptyset$ then
5          remove novel edges that end at $v$ from $novel$ and
             $G$;
6          DCDIJKSTRA($G$, $v$, $preds$, $novel$, $[v]$,
             $negNodes$);
7          if outputted edges changed then
8              $changed.add(v)$;
9  return DCCHECKER($G$, $negNodes$, $novel$, $preds$)
```

Algorithm 4: RELAXIDC - an algorithm for incrementally relaxing constraints to find a dynamically controllable STNU.

For any given call to DCDIJKSTRA for a particular starting node, we start by looking at the predecessor edges that were generated by the shortest path computation. If the edge associated with the constraint whose value is being relaxed is in that starting node's predecessor list, we remove the edges that had been added by the previous run of the recursive call and re-run DCDIJKSTRA to add in the updated edges. We can check if the edge is in the predecessor list in constant time because the predecessor lists are indexed by vertex. We just need to check the bucket corresponding to the modified edge's start node.

In the case where the edge is not in the starting node's predecessor list, we see if any of its child dependencies were re-run. If some of its child dependencies were re-run and generated different edges, we also re-run DCDIJKSTRA.

Once all nodes in the DAG have been processed, we resume the normal non-incremental algorithm from any node

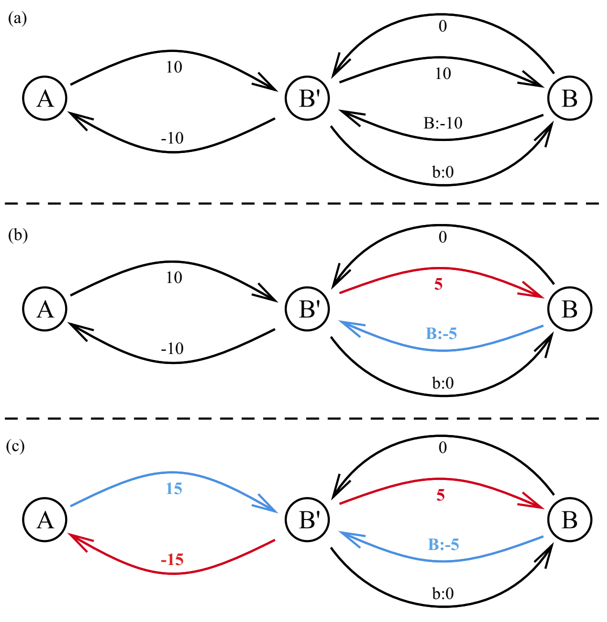

Figure 1: (a) labeled distance graph for a contingent link with bounds [10, 20]; (b) labeled distance graph after relaxing upper bound to 15; (c) labeled distance graph after instead relaxing lower bound to 15; edges in blue are values that increased after relaxation, edges in red decrease in value after relaxation

that has an incoming negative edge.

4.3 Correctness

Since the original algorithm for determining dynamic controllability is agnostic to the order in which initial calls to DCDIJKSTRA are made, to prove correctness we only need to show that the skipped calls would not have a different result if we were to re-run them.

For the relaxation of a controllable constraint, we know that the corresponding edge weights will only increase since we either increase the upper bound u or decrease the lower bound l and have edge weights u and $-l$ in the labeled distance graph. It's clear that an increased edge weight can only affect a shortest path calculation if it were used in a shortest path. Because we keep track of both the edges of a shortest path using the predecessor list, we know the skipped calls will not care about this edge.

However, it is possible for the relaxation of a contingent constraint to cause an edge weight to decrease. But even in these cases, our algorithm correctly chooses whether to re-run DCDIJKSTRA.

When relaxing the upper bound of a contingent constraint (Figure 1b), an upper-case labeled edge's value increases while an unlabeled edge's value decreases. When an edge value decreases, it seems like we have to re-run all shortest-path calculations because, in theory, every path across the full graph could choose to use the shortened edge. However, when we relax a contingent upper-bound, the edge whose value decreases is never used in a shortest path calculation.

Assume for the sake of contradiction that the shortened edge was part of a shortest path. For any such path, we could instead replace that edge with the lower-case edge of weight 0 and have a shorter path. We can always do this unless the path that was being constructed to that point used the corresponding upper-case edge. If the lowest cost upper-case path has a value less than the original upper-case edge weight, we know that there is a negative cycle somewhere else in the graph because the original STNU only has one upper-case edge per label. If the lowest-cost upper-case path equals the value of the original upper-case edge, we have no interest in taking the edge whose value just decreased because that would yield a zero-length cycle. Thus, no path would use that edge.

Now, we can turn our attention to relaxing the lower bound of a contingent constraint (Figure 1c). We know the two edges whose values increase don't affect correctness, and by the previous argument, the positive weight unlabeled edge whose value decreased won't impact our shortest path calculation. We now consider the negative unlabeled edge that decreased in value.

We know that this edge can only be directly used in one of the DCDIJKSTRA calls since every call ignores negative edges except for the ones that are immediately incoming. Any remaining propagation is handled for us by the topological sort. Thus, for any set of relaxed constraints, our incremental preprocessing guarantees that we correctly update all edges.

5 Empirical Results

To understand the performance of our algorithm, we randomly generated a series of STNUs to model the parallel deployment of autonomous underwater vehicles (AUV). Each of the AUVs had to navigate to a series of different sites and conduct experiments along the way; the travel durations were assumed to be contingent links while the AUVs were given the agency to control how long the experiments would take. The AUVs were all given a global deadline by which time they had to complete all tasks. These constraints were all flexible and could be modified by the search algorithm if a valid plan could not be found. For all trials we assumed 70 AUVs, which each had 70 tasks to complete.

In our trials, we intentionally overconstrained the STNUs to make them uncontrollable. This helped us understand how our algorithm performed under a true plan relaxation task. Over the course of 50 trials, we took our randomly generated STNU and ran it through the dynamic controllability checking algorithm. Based on the returned conflict, we relaxed a constraint to eliminate the conflict. We repeated these iterations 10 times each for the incremental and non-incremental dynamic controllability algorithm.

From Figure 2, we see that the time it takes for successive runs of the incremental algorithm tend to decrease over time whereas successive runs of the non-incremental algorithm increase. This matches quite closely to our expectations – the non-incremental algorithm operates by terminating as soon as it finds a semi-reducible negative cycle. As we continue to modify the graph to reduce the number of such cycles, the algorithm has to explore more and more of the graph, before eventually exploring the whole graph and determining that it is controllable.

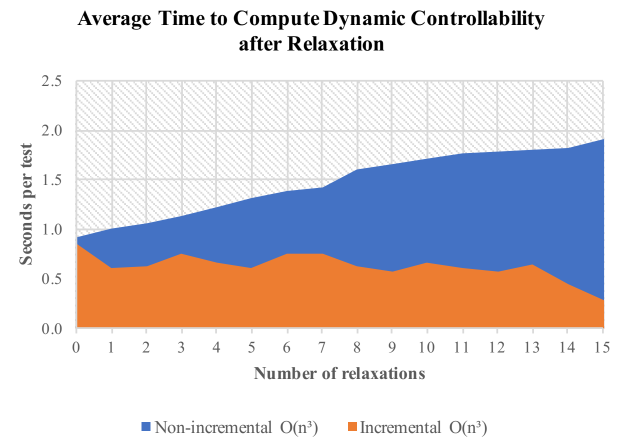

Figure 2: The average runtime of each algorithm after performing relaxations to the original uncontrollable STNU over 50 trials.

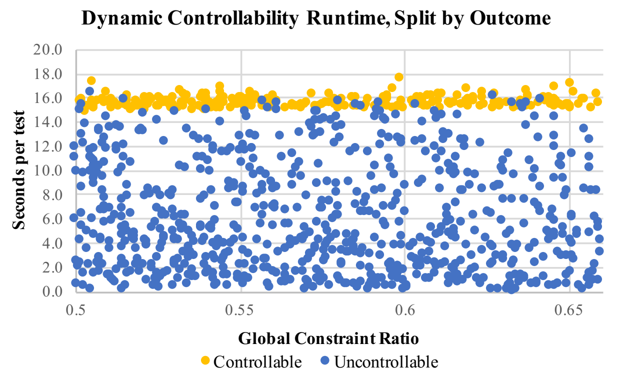

Figure 3: The phase transition of runtimes for uncontrollable and controllable STNUs.

More generally, the problem of detecting dynamic controllability seems to undergo a phase transition that resembles that of other constraint satisfaction problems [Prosser, 1996]. Obviously uncontrollable problems can be detected quite quickly while the closer a graph is to controllable, the more time it takes to determine its controllability (Figure 3).

Our algorithm's advantage in minimizing redundant computation is an improvement specifically because it is able to navigate the phase transition between uncontrollable and controllable states. Over the course of the 15 relaxations, the incremental algorithm approach was in aggregate 2.33 times faster than the non-incremental algorithm.

6 Conclusion

In this paper, we focus on speeding up temporal plan relaxation tasks for overconstrained problems by improving the process of extracting conflicts from a dynamic controllability checker and providing an incremental algorithm for checking the controllability of relaxed STNUs. With our conflict extraction methods, we can drop the runtime of conflict extraction for dynamically controllable systems to $O(n^3)$ from the previous best $O(n^4)$. With this in hand, we take the logical next step for consumers of these conflicts, designing and implementing RELAXIDC, an incremental dynamic controllability checker, that significantly improves the empirical runtime of iterative modifications to temporal constraints.

Acknowledgements

The authors would like to thank Ben Sklaroff, Yuening Zhang, and the conference reviewers for their comments and feedback on this work. This research was funded in part by the Toyota Research Institute under grant number LP-C000765-SR.

References

[Fang *et al.*, 2014] Cheng Fang, Peng Yu, and Brian C Williams. Chance-constrained probabilistic simple temporal problems. 2014.

[Morris and Muscettola, 2005] Paul H Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *AAAI*, pages 1193–1198, 2005.

[Morris, 2006] Paul Morris. A structural characterization of temporal dynamic controllability. In *International Conference on Principles and Practice of Constraint Programming*, pages 375–389. Springer, 2006.

[Morris, 2014] Paul Morris. Dynamic controllability and dispatchability relationships. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 464–479. Springer, 2014.

[Nilsson *et al.*, 2013] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Incremental dynamic controllability revisited. In *ICAPS*, 2013.

[Nilsson *et al.*, 2014] Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Incremental dynamic controllability in cubic worst-case time. In *Temporal Representation and Reasoning (TIME), 2014 21st International Symposium on*, pages 17–26. IEEE, 2014.

[Prosser, 1996] Patrick Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):81–109, 1996.

[Shah *et al.*, 2007] Julie A Shah, John Stedl, Brian C Williams, and Paul Robertson. A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In *ICAPS*, pages 296–303, 2007.

[Stedl and Williams, 2005] John Stedl and Brian C Williams. A fast incremental dynamic controllability algorithm. In *Proceedings of the ICAPS Workshop on Plan Execution: A Reality Check*, pages 69–75, 2005.

[Vidal, 1999] Thierry Vidal. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental & Theoretical Artificial Intelligence*, 11(1):23–45, 1999.

[Williams and Ragno, 2007] Brian C Williams and Robert J Ragno. Conflict-directed a* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, 2007.

[Yu *et al.*, 2014] Peng Yu, Cheng Fang, and Brian C Williams. Resolving uncontrollable conditional temporal problems using continuous relaxations. In *ICAPS*, 2014.