Online Bridged Pruning for Real-Time Search with Arbitrary Lookaheads

Carlos Hernández Ulloa[†], **Adi Botea**[‡], **Jorge A. Baier**[*,§], **Vadim Bulitko**[⋆]
[§]Chilean Center for Semantic Web Research
[‡]IBM Research, Ireland
[*]Pontificia Universidad Católica de Chile
[†]Universidad Andrés Bello, Chile
[⋆]University of Alberta, Canada

Abstract

Real-time search algorithms are relevant to time-sensitive decision-making domains such as video games and robotics. In such settings, the agent is required to decide on each action under a constant time bound, regardless of the search space size. Despite recent progress, poor-quality solutions can be produced mainly due to state re-visitation. Different techniques have been developed to reduce such a re-visitation, with state pruning showing promise. In this paper, we propose a novel pruning approach applicable to the wide class of real-time search algorithms. Given a local search space of arbitrary size, our technique aggressively prunes away all states in its interior, possibly adding new edges to maintain the connectivity of the search space frontier. An experimental evaluation shows that our pruning often improves the performance of a base real-time search algorithm by over an order of magnitude. This allows our implemented system to outperform state-of-the-art real-time search algorithms used in the evaluation.

1 Introduction

Real-Time Heuristic Search (RTHS) [Korf, 1990] comprises a family of algorithms for solving deterministic search problems in time-constrained scenarios, where agents must act before a complete solution can be computed. It has applications in robotics and video games. To solve a search problem, RTHS algorithms (e.g., LSS-LRTA* [Koenig and Sun, 2009]) interleave planning and plan execution. On each iteration, most such algorithms build a bounded local-search space, select an action for execution and use learning mechanisms to update their heuristic function h. The learning updates make h more informed, eventually guaranteeing that the goal state will be reached.

In practical applications, such as video-game pathfinding, heuristic functions have local depressions and simple RTHS algorithms revisit some of the states many times [Shimbo and Ishida, 2003], resulting a phenomenon known as "scrubbing". This hampers the use of such algorithms in real-world applications [Bulitko *et al.*, 2011]. While scrubbing is unavoidable in simple RTHS algorithms [Sturtevant and Bulitko, 2016], more complex techniques can mitigate or even eliminate scrubbing. These include actively avoiding heuristic depressions [Hernández and Baier, 2012], escaping depressions faster by accelerating the heuristic growth rate [Rivera *et al.*, 2015; Bulitko and Sampley, 2016], and pruning states from the search space [Sharon *et al.*, 2013].

We present a new approach to mitigating scrubbing and improving the resulting RTHS solution quality based on pruning. Our technique, PALMA (Pruning with Arbitrary Lookaheads and Macro Actions), is based on a simple idea: if in a local search the search frontier (the $Open$ list) is a *connected* set of states, all states expanded in this search (the $Closed$ list) can safely be pruned from the search space. If the frontier ($Open$ list) is *disconnected* we still prune all states in $Closed$ from the search space, but add new edges to ensure that states on the frontier remain reachable from each other.

A limited implementation of this idea was evaluated by Sharon *et al.* [2013] who detected and pruned at most a single *expendable* state per move. PALMA is more general and can prune much larger sets of states expanded during the search. In fact, by pruning *all* expanded states per move while adding new edges to preserve goal reachability, PALMA guarantees reaching the goal even without heuristic learning. Finally, PALMA can operate on top of any local-search-space-building RTHS algorithm. Similarly to macro-edges used in previous route planning and pathfinding approaches, such as contraction hierarchies [Geisberger *et al.*, 2008] for example, our new edges are shortcuts that allow avoiding intermediate nodes during a search.

We implement PALMA on top of LSS-LRTA* [Koenig and Sun, 2009] and call the resulting algorithm PALMA(LSS-LRTA*). An empirical evaluation is performed on standard pathfinding benchmarks [Sturtevant, 2012]. We observe significant performance gains for a range of lookahead values. For lower lookahead values, the improvements on top of LSS-LRTA* can exceed one order of magnitude. Even with deeper lookahead, the improvements remain significant. Overall, PALMA(LSS-LRTA*) outperforms high-performance RTHS algorithms used in our evaluation.

We formulate the real-time heuristic search problem in Section 2 and review related work in Section 3. We then discuss the idea of connectivity-preserving pruning in Section 4 and propose our novel method of implementing it in Section 5. This is followed by a theoretical analysis in Section 6 and an empirical evaluation in Section 7. A short discussion and future work directions conclude the paper.

2 Problem Formulation

We use a standard formulation of the real-time heuristic search which we adopt from Bulitko [2016]. A *search problem* is a tuple (S, E, c, s_0, s_g, h), where S is a finite set of *states* (also called *nodes*), and $E \subset S \times S$ is a finite set of *edges* between them. S and E jointly define the search graph G which we assume to be undirected (i.e., $(s, s') \in E$ iff $(s', s) \in E$). A cost function c assigns a positive finite cost to each edge in E. Two states s_a and s_b are *immediate neighbors* iff there is an edge between them: $(s_a, s_b) \in E$; we denote the set of immediate neighbors of a state s by $N(s)$. A *path* π in graph $G = (S, E)$ is a sequence of states $(s_0, s_1, \dots, s_n)$ such that for all $i \in \{0, \dots, n-1\}$, $(s_i, s_{i+1}) \in E$. If there exists a path starting with s and ending with t we say that t *is reachable from s in G*, denoted by $s \rightsquigarrow_G t$. Given a graph (S, E) and a subset of nodes S' in S, *the graph induced by S'* is $G' = (S', E')$, where $E' = \{(u, v) \in E \mid u, v \in S'\}$. We assume that the search graph (S, E) is connected (i.e., any two vertices have a path between them) which makes it safely explorable. We also assume that the search graph is stationary (e.g., the edge weights do not change during search).

At all times $t \in \{0, 1, \dots\}$ the agent occupies a single state $s_t \in S$, called the *current state*. The state s_0 is the start state and is given as a part of the problem. The agent can change its current state, that is, move to any immediately neighboring state in $N(s)$. The traversal incurs a travel *cost* of $c(s_t, s_{t+1})$. The agent solves the search problem at the earliest time it arrives at the goal state. The *solution* is a path $P = (s_0, \dots, s_g)$. The cumulative cost of all edges in a solution is called the *solution cost*. The cost of the shortest possible path between states $s_a, s_b \in S$ is denoted by $h^*(s_a, s_b)$. We abbreviate $h^*(s, s_g)$ as $h^*(s)$. The *suboptimality* of the agent on a problem is the ratio of the solution cost the agent incurred to the cost of the shortest possible solution. Lower suboptimality values are preferred.

The agent has access to a heuristic $h : S \to [0, \infty)$. The initial heuristic is a part of the problem specification and is meant to give the agent an estimate of the remaining cost to go. The search agent can modify the heuristic at will. We say that a search agent is *real time* iff the computation time between the agent's moves is upper-bounded by a constant independent of the number of states in the search space. A search algorithm is *correct* if its computed solutions (paths to goal) are valid, and *complete* if it outputs a solution to any search problem as defined above. With our pruning approach we aim to reduce the solution suboptimality and the total planning time while keeping the algorithm correct and complete.

For presentational clarity we introduce our pruning strategy on top of a commonly used LRTA*-style real-time search framework (Algorithm 1).

A search agent in this framework begins in the start state s_0. It then executes a fixed loop until it reaches the goal s_g (line 3). At each iteration[*] of the loop, the agent expands[†] some states around its current state s_t (line 4). The expanded states comprise the $Closed$ list. All states generated but not expanded constitute the $Open$ list. Then the agent updates heuristic values of states in $Closed$ (line 6). Finally, it moves from s_t to s_{t+1}, which is the most promising state in $Open$ (line 7). The cycle then repeats until the goal state is reached. The goal state is never expanded. Thus, it can belong only to the $Open$ list and is never removed from it.

[*]Search performed within an iteration is called a *search episode*.
[†]A state is expanded if its immediate neighbors are generated.

Algorithm 1: Real-time Heuristic Search Framework

```
input : search problem $(S, E, c, s_0, s_g, h)$
output: solution path $(s_0, s_1, \dots, s_g)$
1 $t \leftarrow 0$
2 $h_t \leftarrow h$
3 while $s_t \neq s_g$ do
4     form local search space $Closed \cup Open$
5     set $\pi$ to the shortest path connecting $s_t$ and the most
        promising state in $Open$
6     update heuristic values in $Closed$
7     move the agent through every state in path $\pi$
8     $t \leftarrow t + 1$
```

3 Related Work

Pruning approaches mitigate the scrubbing problem of real-time heuristic search by permanently removing some states from the search graph. The graph is then reduced in size, decreasing the number of state revisits. The key questions are: how to efficiently identify the states to prune, how to preserve algorithm completeness and what impact on solution suboptimality the pruning will have.

Pochter *et al.* [2011] used graph automorphism to detect symmetries and reduce the search graph in planning. In real-time heuristic search, Sturtevant *et al.* [2010] identified two special types of states: dead-ends and redundant states. Informally, if a state is a dead-end then it cannot belong to an optimal path. Two or more states are redundant when only one of them is needed to preserve solution optimality. To identify such states, their algorithm, RIBS, learned the optimal g costs from the start state to a given state. Sturtevant and Bulitko [2011] adopted the pruning technique of RIBS and combined its g learning with the more traditional h learning. This has led to a stronger pruning performance.

Pruning techniques in RIBS and f-LRTA* rely on learning g and thus are not applicable to LRTA*-style algorithms, which learn h-values only. Sharon *et al.* [2013] introduced the notion of a locally expendable state as a state whose immediate neighbors are connected through themselves. Their algorithm then pruned away the agent's current state if it is found to be expendable. Note that their pruning technique was applied only to the local search space of a single state (i.e., lookahead of 1) and, in grid pathfinding, could create a checkerboard-like pattern of pruned states. Such a pattern

By definition, the lookahead value is the the number of states expanded in one episode. With the lookahead of 1 only the current state is expanded.

would then preclude pruning other states as they become non-expendable. Thus they pruned only the states whose heuristic was updated by the agent which, in the usual grid pathfinding, happens only around obstacles [Sturtevant, 2016].

Our pruning strategy is a generalization of their idea to arbitrarily large local search spaces and their frontiers. Furthermore, when the frontier of the local search space is disconnected, we still prune away the local search space and restore connectivity by adding new edges.

Contraction hierarchies (CHs) [Geisberger *et al.*, 2008] introduce shortcut edges that can prune away intermediate nodes from a search. CHs have been used for optimal route planning on road maps and, more recently, for pathfinding on gridmaps [Sturtevant *et al.*, 2015]. They are built in a preprocessing step. In fact, using macro-edges in gridmap pathfinding, as a way to avoid searching in a given area, is a popular idea in the literature (e.g., [Botea *et al.*, 2004; Uras *et al.*, 2013]). In contrast, our technique applies to real-time search, and it performs search-space contractions online.

4 Bridged Pruning

The main idea of our approach is to prune the search space of nodes that do not need to be considered as part of the search space in subsequent episodes. Pruning is performed after each search episode, right after the movement has concluded (i.e., immediately after line 7 in Algorithm 1).

Definition 1 (Pruning)**.** Given a graph $G = (S, E)$ and a set of states $T \subset S$, the result of *pruning T from G* is the graph induced from G by $S \setminus T$, denoted by G_{-T}.

Definition 2 (Frontier)**.** Given a graph $G = (S, E)$ and a set of states $T \subset S$, the *frontier of T*, denoted by $\partial_G T$ (or simply by ∂T if G is clear from the context), is the set $\{s \mid (t, s) \in E, t \in T, s \notin T\}$.

In undirected connected graphs, any set of nodes is connected. This property, however, may not hold true as a result of pruning states of T. Indeed, the frontier ∂T may become disconnected (this can be observed in Figure 2, described later, where ∂T are the gray dotted cells with three numbers each). A property that guarantees that the goal is reachable after pruning is that the frontier of the pruned area, ∂T, remains connected. Hence, we define the notion of *bridged pruning*, which accounts for the fact that, after generating G_{-T}, extra edges can be added to G_{-T} to connect (bridge) states in ∂T.

Definition 3 (Bridged Pruning)**.** Given a graph $G = (S, E)$ and a set of states $T \subset S$, the graph $G_{\ominus T}$ is the result of a *bridged pruning of T from G* iff $G_{\ominus T}$ is obtained from G_{-T} by adding zero or more new edges between states in $\partial_G T$ such that $\partial_G T$ induces a connected subgraph of $G_{\ominus T}$.

Lemma 1 (Preserving Reachability)**.** Let $G_{\ominus T}$ be a bridged pruning of T from a graph G. Furthermore let s and t be such that $s \rightsquigarrow_G t$. If s and t have not been pruned (i.e., are not in T) then $s \rightsquigarrow_{G_{\ominus T}} t$.

Proof. Let π be a path in G from s to t (depicted as the shorter trajectory in Figure 1). If π does not contain a state from T then π is a path between s and t in $G_{\ominus T}$ and therefore $s \rightsquigarrow_{G_{\ominus T}} t$. Otherwise, π is of the form $\pi = (n_1, \ldots, n_m)$, where $n_1 = s$ and $n_m = t$. Since $n_1, n_m \notin T$ there is a non-empty subpath $(n_i, n_{i+1}, \ldots, n_j)$ of states in T such that n_{i-1} and n_{j+1} are both in ∂T. As ∂T is connected in $G_{\ominus T}$, there must be another path between n_{i-1} and n_{j+1} in $G_{\ominus T}$ (depicted as the dashed detour in the figure). Let σ denote its states from the second to the second-to-last. Since σ lies entirely in $G_{\ominus T}$, it has no states in T. Therefore the path $\pi' = (n_1, \ldots, n_{i-1}, \sigma, n_{j+1}, \ldots, n_m)$ contains at least one fewer state from T than π did. This process can be repeated until the constructed path has no states from T at all. $\square$

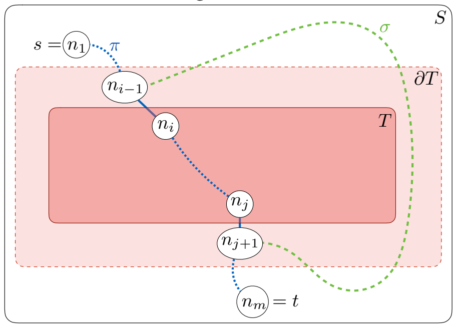

Figure 1: To the proof of Lemma 1.

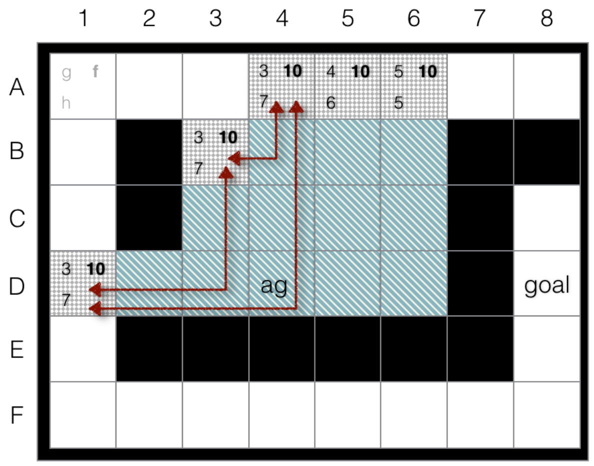

Figure 2: PALMA illustrated on a 4-connected grid.

The important implication of Lemma 1 is that bridged pruning keeps the goal reachable from the agent's current state as long as neither the agent's current state nor the goal are pruned. We use this in the next section to show that our bridged-pruning system preserves the search completeness.

5 The PALMA Pruning Method

Above we have discussed bridged pruning, which ensures reachability of the goal state. We have not discussed yet how a bridged pruning can be carried out in practice. In this section we present PALMA, our implementation of bridged pruning. While PALMA can be used in any undirected graph, we will use grid pathfinding to illustrate it.

The four-connected grid in Figure 2 will serve as a running example to describe PALMA. Black cells are obstacles, whereas all other cells form the traversable area. Cell $D8$ is

Algorithm 2: PALMA

```
input : a set of states $Closed$, a search weighted graph
        $G = (S, E, c)$
output: $G_{\ominus Closed}$
1 $S' \leftarrow S \setminus Closed$
2 $E' \leftarrow E \cap (S' \times S')$
3 $E_{\text{new}}, c_{\text{new}} \leftarrow$ PALMACONNECT($\partial Closed, G$)
4 $E' \leftarrow E' \cup E_{\text{new}}$
5 $c' \leftarrow c \cup c_{\text{new}}$
6 return $G' = (S', E', c')$
```

Algorithm 3: PALMACONNECT

```
input : a set of states, $Closed$, a weighted graph $G$
output: a set of new weighted edges between states in $Closed$
1 $E_{\text{new}} \leftarrow \emptyset$
2 $c_{\text{new}} \leftarrow \emptyset$
3 $G_B \leftarrow$ subgraph of $G$ induced by $\partial Closed$
4 $G_L \leftarrow$ subgraph of $G$ induced by $Closed \cup \partial Closed$
5 if $G_B$ is not connected then
6     $\{G_1, \ldots, G_r\} \leftarrow$ connected components of $G_B$
7     for each pair $(i, j)$ such that $1 \leq i < j \leq r$ do
8         select a state $s_i$ in $G_i$ and a state $s_j$ in $G_j$
9         $E_{\text{new}} \leftarrow E_{\text{new}} \cup \{(s_i, s_j), (s_j, s_i)\}$
10        $c' \leftarrow$ cost of a shortest path between $s_i$ and $s_j$ in $G_L$
11        $c_{\text{new}} \leftarrow c_{\text{new}} \cup \{((s_i, s_j), c'), ((s_j, s_i), c')\}$
12 return $E_{\text{new}}, c_{\text{new}}$
```

the goal. $D4$ is the agent location at the *beginning* of the iteration. The figure shows a local search space formed in line 4 of Algorithm 1. In this example we used an A*-shaped local search. Dotted cells comprise the $Open$ list. They are labeled with h, g and $f = g + h$ (clockwise from the bottom left). Diagonally striped cells form the $Closed$ list. After the search, the agent follows a path to a lowest f-value state in the $Open$. In our example, all states in the $Open$ have $f = 10$ so any of them can be the state to go to, depending on the tie-breaking schema employed.

After line 7 of Algorithm 1, we invoke our bridged pruning procedure PALMA. Algorithm 2 gives an overview of PALMA. Its input is the current search graph and the $Closed$ list which contains a set of states expanded in the current search episode. In the first two lines, $G_{-Closed}$ is computed. Then additional edges E_{new} are computed by calling PALMACONNECT in line 3. That function returns weighted edges that reconnect the frontier $\partial Closed$ if $\partial Closed$ becomes disconnected and returns an empty set otherwise. The function PALMA then returns $G_{\ominus Closed}$, the result of the bridged pruning of $Closed$ from G (line 6). The current search graph is replaced with the graph returned by PALMA.

We will now walk through PALMACONNECT, presented as Algorithm 3. PALMACONNECT adds new edges to make sure that the frontier is connected. Specifically, in line 5 we check whether the graph G_B, the graph induced by $\partial Closed$ (line 3), is connected. If so there is no need to add any new edges because there already exists a path between any two states of $\partial Closed$ that visits only states in $\partial Closed$. If G_B is not connected, however, we compute an edge between two states that lie in two different disconnected components of G_B in line 9. The cost assigned to an action connecting two states u and v is the cost of a shortest path between u and v in the search graph before pruning away the states in $Closed$ at the current episode (lines 10 and 11).

Algorithm 3 specifies neither how to select states in the connected components in line 8 nor how to compute the cost of new edges in line 10. In this paper we used the following implementation. For each of the r connected components $G_i; i \in \{1, \ldots, r\}$ we select the state $s_i = \arg\min\{g(s) | s \in G_i\}$. To compute the costs of the new edges $(s_i, s_j); i, j \in \{1, \ldots, r\}$, we run $r - 1$ Dijkstra searches starting them in $s_1, s_2, \ldots, s_{r-1}$ respectively. A Dijkstra search is restricted to G_L, is started in s_i and is stopped as soon as optimal costs from s_i to all states $s_j, j > i$ are computed. This process ensures that a path will be discovered for every pair of selected states while bounding the search effort.[‡]

To illustrate the process of adding new edges, we revisit the example in Figure 2. Recall that the dotted cells with three numbers each compose the $Open$ list. The $Open$ list has three connected components: $\{D1\}$, $\{B3\}$ and $\{A4, A5, A6\}$. Two of the three connected components have one state each ($\{D1\}$ and $\{B3\}$). For the three-state component $\{A4, A5, A6\}$ the state selected for connection is $A4$. We then compute shortest paths between $D1$ and $B3$; $B3$ and $A4$; and $D1$ and $A4$. For each such path we add a single new weighted edge (e.g., $(D1, B3)$ will have a cost of 4). New edges are shown as bi-directional arrows in the figure. All diagonally striped states (the $Closed$ list) are then pruned out from the search graph G.

Note that the modifications to the search graph made by bridged pruning exist solely in the agent's representation of the world. The actual graph traversed by the agent remains unchanged throughout the search. So when PALMA prunes a state, it simply marks it as such in the agent's representation of the world. When PALMA adds a new edge, it actually remembers a sequence of states in the original graph the agent would have to travel in the world to traverse the new edge. In our running example, traversing the new edge between states $B3$ and $D1$ would make the agent visit the states $C3$, $D3$, and $D2$, marked as "pruned" from the search graph.

6 Theoretical Analysis

We will now discuss the completeness of RTHS algorithms equipped with PALMA. Let PALMA(A) be an application of PALMA to a real-time heuristic algorithm A which itself conforms to the framework presented in Algorithm 1.

Lemma 2. PALMA(A) runs at most $n - 1$ search episodes.

Proof. After each search episode, at least one state is pruned from the search graph by PALMA. Unless the goal is reached earlier, after $n - 2$ episodes the remaining search graph will contain at most 2 states. Thus, the goal state will be generated

[‡]We evaluated two other ways of selecting a representative state s_i from each connected component G_i: (i) minimizing $f(s_i)$ instead of $g(s_i)$ and (ii) minimizing $c'(s_i, s_j)$ across all i, j. Both proved inferior in the experiments we ran.

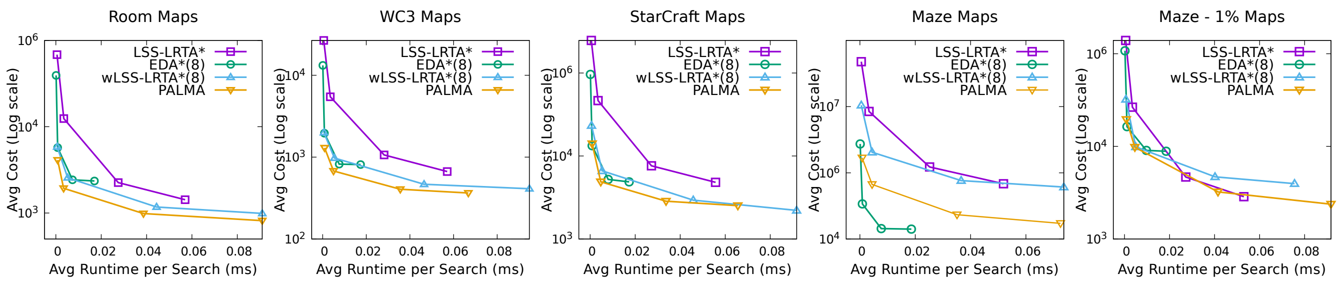

Figure 3: Solution cost versus average runtime per search episode. Note the logarithmic scale on the vertical axis.

within the next search episode which will lead to the agent's reaching the goal on the next episode. □

In real-time heuristic search, heuristic updates are often used to ensure completeness. Without such updates, algorithms such as LRTA* can degenerate to simple hill-climbing and get forever stuck in a local heuristic depression. Remarkably, the bridged pruning implemented in PALMA guarantees completeness on its own, without any heuristic learning.

Lemma 3. When using PALMA(A), the goal is always reachable from the agent's current state.

Proof. The agent's current state is never pruned as the agent moves to $Open$ before pruning the $Closed$ list. The goal is never pruned as it can never be in $Closed$. It follows from Lemma 1 that the goal is reachable from every state in the bridge-pruned search space, including the agent's state. □

Theorem 1. For any RTHS algorithm A that fits the framework in Algorithm 1, PALMA(A) is correct (i.e., terminates only at the goal state) and complete (i.e., reaches the goal state) *regardless* of any learning (or lack thereof) of the heuristic function.

Proof. After each search episode, the goal remains reachable from the agent's position, according to Lemma 3 and therefore PALMA(A) terminates only when A terminates. A terminates only upon reaching the goal (line 3 in Algorithm 1). Thus PALMA(A) is correct. PALMA(A) is complete as it terminates after at most $n-1$ search episodes by Lemma 2. □

For simplicity, our analysis focused on solvable problems. With minor modifications to Algorithm 3, on a finite graph PALMA(A) can detect that no solution exists (i.e., the goal is not reachable from the agent's initial state). We omit the proof due to space limitations. Being able to detect that a solution does not exist is a benefit PALMA brings to LRTA*-style algorithms that cannot do so by themselves.

7 Empirical Evaluation

PALMA is applicable to any real-time search algorithm that conforms to Algorithm 1. For our evaluation, we chose to apply it on top of LSS-LRTA* [Koenig and Sun, 2009], which is a high-performance commonly used real-time search algorithm generalizing the classic LRTA* algorithm [Korf, 1990]. To assess the resulting algorithm performance relative to the state of the art we compare it to wLSS-LRTA* [Hernández and Baier, 2012] and EDA* [Sharon *et al.*, 2014]. The former is a representative of the depression-avoiding family of algorithms while the latter is an IDA*-inspired algorithm that does not fit into the framework of Algorithm 1 but was shown to have good performance with respect to depression-avoiding algorithms [Sharon *et al.*, 2014]. For both algorithms we use the best parameters as reported in the original publications ($w = 8$ for wLSS-LRTA* and $C = 8$ for EDA*).

We implemented all algorithms in C over a common code base. For LSS-LRTA* and its variants, we use a binary heap for $Open$ in which ties are broken in favor of larger g-values.

We took maps from the commonly used MovingAI pathfinding repository [Sturtevant, 2012]. We used all five 1024×1024 *StarCraft* maps, 30 maze maps of size 512×512 (corridor widths of $4, 8, 16$), all 512×512 maps *World of Warcraft III* (WC3) and all room maps of size 512×512. Finally, as a fifth benchmark, we consider the same maze maps, but with 1% of their obstacles removed. For each map we generated 100 random solvable problems. Each map is modeled as an undirected eight-connected grid, with costs set to 1 for cardinal moves and $\sqrt{2}$ for diagonal moves. The octile distance is used as a heuristic. Experiments are run on a 2.60GHz Intel Core i7 machine under Linux, as a single-threaded process.

By varying the lookahead value (i.e., number of expanded states per episode) LSS $(1, 10, 100$ and $200)$ we can trade solution quality with the time per search episode as shown in Figure 3. When the time spent per each search is relatively uniform between episodes, as is the case with all of these algorithms, these plots provide a fair comparison between algorithms. They show PALMA(LSS-LRTA*) is the best-performing algorithm in four out of five benchmarks. In the remaining domain (mazes), for all but the smallest lookahead value, EDA* computes shorter solutions in a shorter time.

Table 1 shows the average solution cost, the average per-problem runtime, and the average number of search episodes. Best performance per lookahead value is shown in bold. Given a combination c of a lookahead value and a map type (i.e., a block in the table), we say that that algorithm *dominates* the others for combination c if it has both the lowest solution cost and the lowest runtime. PALMA(LSS-LRTA*) is the dominant algorithm in 15 out of 20 combinations (blocks in the table). EDA* dominates in three out of 20 combinations, all corresponding to mazes. In two out of the 20 combinations no algorithm is dominant yet PALMA(LSS-LRTA*) has the shortest solution in one of these two combinations.

Comparing PALMA(LSS-LRTA*) to its base algorithm

Table 1: Empirical results showing the influence of lookahead value (*LSS*) on average solution cost (*Cost*), average per-problem running time in milliseconds (*Time*) and average per-problem number of search episodes (*Epis*).

		Room Maps			**WC3 Maps**			**SC Maps**			**Mazes**			**Mazes - 1% obst.**		
	LSS	*Cost*	*Time*	*Epis.*	*Cost*	*Time*	*Epis.*	*Cost*	*Time*	*Epis.*	*Cost*	*Time*	*Epis.*	*Cost*	*Time*	*Epis.*
LSS-LRTA*	1	68,478	31.25	63,574	26,624	12.50	24,723	246,763	116.98	229,262	4,774,362	2,103.83	4,470,766	139,137	63.17	128,661
EDA*(8)	1	39,330	4.55	35,905	13,212	1.52	11,725	96,103	11.47	85,986	273,001	**33.22**	257,987	107,600	15.38	99,268
wLSS-LRTA(8)	1	5,843	3.03	5,386	1,995	1.10	1,808	23,262	12.09	20,598	1,049,026	466.29	891,429	32,154	16.36	28,494
PALMA	1	**4,113**	**2.66**	**3,142**	**1,299**	**0.92**	**973**	**14,048**	**8.34**	**9,292**	**168,440**	70.73	**86,666**	**19,488**	**11.08**	**13,104**
LSS-LRTA*	10	12,432	18.90	5,439	5,459	8.32	2,352	46,477	72.02	20,468	846,310	1,211.58	365,178	26,496	38.72	11,169
EDA*(8)	10	5,728	2.86	3,591	1,960	0.97	1,173	13,302	7.44	8,599	**33,925**	**22.46**	25,799	16,330	10.66	9,927
wLSS-LRTA(8)	10	2,565	2.99	571	974	1.12	207	6,632	7.74	1,447	204,548	187.95	42,401	9,883	10.93	2,200
PALMA	10	**1,936**	**2.02**	**424**	**678**	**0.68**	**135**	**4,882**	**4.85**	**1,034**	66,810	40.93	**9,339**	**9,775**	**8.56**	**1,848**
LSS-LRTA*	100	2,251	7.66	279	1,064	3.52	125	7,624	26.90	989	122,731	375.08	14,903	4,662	14.17	520
EDA*(8)	100	2,424	2.63	360	822	0.89	118	5,224	6.85	860	**14,351**	**19.84**	2,580	9,041	9.59	993
wLSS-LRTA(8)	100	1,174	3.35	75	466	1.21	26	2,939	8.75	191	76,335	174.37	4,769	4,687	11.66	290
PALMA	100	**987**	**2.34**	**58**	**405**	**0.71**	**20**	**2,853**	**5.82**	**173**	23,273	30.52	**870**	**3,223**	**6.37**	**153**
LSS-LRTA*	200	1,427	6.52	114	667	2.78	49	4,816	23.10	415	68,297	277.87	5,356	2,865	10.64	201
EDA*(8)	200	2,351	3.25	188	812	1.09	63	4,881	7.73	448	**14,110**	**24.43**	1,316	8,860	9.19	497
wLSS-LRTA(8)	200	989	4.04	44	411	1.47	16	**2,214**	9.05	99	60,897	188.86	2,564	3,959	12.35	164
PALMA	200	**816**	**2.96**	**32**	**365**	**0.79**	**12**	2,522	**6.43**	**98**	17,210	29.58	**408**	**2,375**	**6.92**	**75**

Table 2: Lookahead depth (*LSS*), the average number of connected components in the $Open$ list (r) as well as the average and maximum numbers of state successors. Each number is an average over 100 instances.

	LSS	*r*	*Avg # Succ*	*Max # Succ*
Room maps	1	1.41	4.9	8.3
	10	1.67	5.9	8.3
	100	3.21	6.9	9.7
	200	4.45	7.1	11.4
WC3 maps	1	1.19	5.8	8.1
	10	1.16	6.6	8.1
	100	1.04	7.3	8.1
	200	1.07	7.4	8.1
SC maps	1	1.36	5.1	8.9
	10	1.40	6.1	8.5
	100	1.57	7.1	8.5
	200	1.76	7.3	8.7
Maze maps	1	1.57	4.3	9.6
	10	1.71	5.5	10.2
	100	2.23	6.7	9.4
	200	2.62	6.8	9.9
Maze-1% maps	1	1.51	4.5	8.7
	10	1.66	5.6	9.2
	100	2.51	6.7	9.7
	200	3.22	6.9	10.5

LSS-LRTA*, Table 1 shows that PALMA(LSS-LRTA*) dominates LSS-LRTA* in all 20 combinations. PALMA's improvements can reach a factor of 20 in terms of both solution cost and runtime. Overall PALMA(LSS-LRTA*) shows significant performance improvements over state-of-the-art algorithms used in our evaluation. PALMA(LSS-LRTA*) consistently requires the smallest number of search episodes.

PALMA dynamically removes and adds edges to the search graph, varying the number of successors that a node can have and therefore its expansion time. Furthermore, the time effort required to add new edges depends on the size of the $Open$ list (to check if the list is connected) and the number r of connected components of the $Open$ list (as we run $r - 1$ Dijkstra searches to add edges between connected components). Table 2 shows summary statistics relevant to these computations. First it shows the average number of connected components (r) which turns out to be small, ranging from 1.04 to 4.45. In 13 out of the 20 combinations, the average r is below 2 implying that often the $Open$ list is already connected and no Dijkstra searches are needed at all.

As PALMA adds new edges, we inspected the branching factor of the pruned graph (i.e., the number of successors for a state). Table 2 reports the average and the maximum number of state successors. Without new edges, an eight-connected grid map has at most 8 successors for any state. PALMA does not increase it substantially, with the maximum values ranging from 8.1 to 11.4. In 11 out of the 20 combinations, the maximum number of successors does not exceed 9.

8 Conclusion

Real-time heuristic search algorithms often produce low-quality solutions due to state re-visitation. Pruning states from the graph has shown promise in previous work but was limited to the basic case of removing the current state.

We presented a principled approach to pruning arbitrarily large sets of states while preserving the underlying algorithm's correctness and completeness. Our pruning can be implemented on top of most real-time search algorithms. Its aggressive pruning guarantees completeness on its own, without relying on heuristic learning. An empirical evaluation against state-of-the-art algorithms shows significant benefits.

Future work will investigate how pruning from a problem instance can be re-used in solving another problem instance. We also plan to perform an evaluation on scenarios with partial terrain knowledge.

Acknowledgements

Carlos Hernández and Jorge Baier were partly funded by Fondecyt grants 1150328 and 1161526.

References

[Botea *et al.*, 2004] A. Botea, M. Müller, and J. Schaeffer. Near Optimal Hierarchical Path-Finding. *Journal of Game Development*, 1(1):7–28, 2004.

[Bulitko and Sampley, 2016] Vadim Bulitko and Alexander Sampley. Weighted lateral learning in real-time heuristic search. In *Proceedings of the Symposium on Combinatorial Search*, 2016.

[Bulitko *et al.*, 2011] Vadim Bulitko, Yngvi Björnsson, Nathan Sturtevant, and Ramon Lawrence. *Real-time Heuristic Search for Game Pathfinding*. Applied Research in Artificial Intelligence for Computer Games. Springer, 2011.

[Bulitko, 2016] Vadim Bulitko. Evolving real-time heuristic search algorithms. In *Proceedings of the Fifteenth International Conference on the Synthesis and Simulation of Living Systems*, 2016.

[Geisberger *et al.*, 2008] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, pages 319–333, 2008.

[Hernández and Baier, 2012] Carlos Hernández and Jorge A. Baier. Avoiding and escaping depressions in real-time heuristic search. *Journal of Artificial Intelligence Research*, 43:523–570, 2012.

[Koenig and Sun, 2009] Sven Koenig and Xiaoxun Sun. Comparing real-time and incremental heuristic search for real-time situated agents. *Journal of Autonomous Agents and Multi-Agent Systems*, 18(3):313–341, 2009.

[Korf, 1990] R.E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2–3):189–211, 1990.

[Pochter *et al.*, 2011] Nir Pochter, Aviv Zohar, and Jeffrey S Rosenschein. Exploiting problem symmetries in state-based planners. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2011.

[Rivera *et al.*, 2015] Nicolas Rivera, Jorge A. Baier, and Carlos Hernández. Incorporating weights into real-time heuristic search. *Artificial Intelligence*, 225:1–23, 2015.

[Sharon *et al.*, 2013] Guni Sharon, Nathan Sturtevant, and Ariel Felner. Online detection of dead states in real-time agent-centered search. In *Sixth Annual Symposium on Combinatorial Search (SoCS)*, pages 167–174, 2013.

[Sharon *et al.*, 2014] Guni Sharon, Ariel Felner, and Nathan R. Sturtevant. Exponential deepening A* for real-time agent-centered search. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 871–877, 2014.

[Shimbo and Ishida, 2003] Masashi Shimbo and Toru Ishida. Controlling the learning process of real-time heuristic search. *Artificial Intelligence*, 146(1):1–41, 2003.

[Sturtevant and Bulitko, 2011] Nathan R. Sturtevant and Vadim Bulitko. Learning where you are going and from whence you came: H- and G-cost learning in real-time heuristic search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 365–370, 2011.

[Sturtevant and Bulitko, 2016] Nathan Sturtevant and Vadim Bulitko. Scrubbing during learning in real-time heuristic search. *Journal of Artificial Intelligence Research*, 2016.

[Sturtevant *et al.*, 2010] N. R. Sturtevant, V. Bulitko, and Y. Björnsson. On learning in agent-centered search. In *Proceedings of the Conference on Autonomous Agents and Multiagent Systems*, pages 333–340, 2010.

[Sturtevant *et al.*, 2015] Nathan R. Sturtevant, Jason M. Traish, James R. Tulip, Tansel Uras, Sven Koenig, Ben Strasser, Adi Botea, Daniel Harabor, and Steve Rabin. The Grid-Based Path Planning Competition: 2014 Entries and Results. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SoCS*, 2015.

[Sturtevant, 2012] N. R. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.

[Sturtevant, 2016] Nathan Sturtevant. Personal communication. 2016.

[Uras *et al.*, 2013] Tansel Uras, Sven Koenig, and Carlos Hernández. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS*, 2013.