Journal of Artificial Intelligence Research 32 (2008) 419 - 452 Submitted 11/07; published 06/08

Dynamic Control in Real-Time Heuristic Search

Vadim Bulitko BULITKO@UALBERTA.CA
Department of Computing Science, University of Alberta
Edmonton, Alberta, T6G 2E8, CANADA
Mitja Luštrek MITJA.LUSTREK@IJS.SI
Department of Intelligent Systems, Jožef Stefan Institute
Jamova 39, 1000 Ljubljana, SLOVENIA
Jonathan Schaeffer JONATHAN@CS.UALBERTA.CA
Department of Computing Science, University of Alberta
Edmonton, Alberta, T6G 2E8, CANADA
Yngvi Björnsson YNGVI@RU.IS
School of Computer Science, Reykjavik University
Kringlan 1, IS-103 Reykjavik, ICELAND
Sverrir Sigmundarson SVERRIR.SIGMUNDARSON@LANDSBANKI.IS
Landsbanki London Branch, Beaufort House,
15 St Botolph Street, London EC3A 7QR, GREAT BRITAIN

Abstract

Real-time heuristic search is a challenging type of agent-centered search because the agent's planning time per action is bounded by a constant independent of problem size. A common problem that imposes such restrictions is pathfinding in modern computer games where a large number of units must plan their paths simultaneously over large maps. Common search algorithms (e.g., A*, IDA*, D*, ARA*, AD*) are inherently not real-time and may lose completeness when a constant bound is imposed on per-action planning time. Real-time search algorithms retain completeness but frequently produce unacceptably suboptimal solutions. In this paper, we extend classic and modern real-time search algorithms with an automated mechanism for dynamic depth and subgoal selection. The new algorithms remain real-time and complete. On large computer game maps, they find paths within 7% of optimal while on average expanding roughly a single state per action. This is nearly a three-fold improvement in suboptimality over the existing state-of-the-art algorithms and, at the same time, a 15-fold improvement in the amount of planning per action.

1. Introduction

In this paper we study the problem of *agent-centered real-time heuristic search* (Koenig, 2001). The distinctive property of such search is that an agent must repeatedly plan and execute actions within a constant time interval that is independent of the size of the problem being solved. This restriction severely limits the range of applicable heuristic search algorithms. For instance, static search algorithms such as A* (Hart, Nilsson, & Raphael, 1968) and IDA* (Korf, 1985), re-planning algorithms such as D* (Stenz, 1995), anytime algorithms such as ARA* (Likhachev, Gordon, & Thrun, 2004) and anytime re-planning algorithms such as AD* (Likhachev, Ferguson, Gordon, Stentz, & Thrun, 2005) cannot guarantee a constant bound on planning time per action. LRTA* can, but with potentially low solution quality due to the need to fill in heuristic depressions (Korf, 1990; Ishida, 1992).

As a motivating example, consider an autonomous surveillance aircraft in the context of disaster response (Kitano, Tadokoro, Noda, Matsubara, Takahashi, Shinjou, & Shimada, 1999). While surveying a disaster site, locating victims, and assessing damage, the aircraft can be ordered to fly to a particular location. Radio interference may make remote control unreliable thereby requiring a certain degree of autonomy from the aircraft by using AI. This task presents two challenges. First, due to flight dynamics, the AI must control the aircraft in real time, producing a minimum number of actions per second. Second, the aircraft needs to reach the target location quickly due to a limited fuel supply and the need to find and rescue potential victims promptly.

We study a simplified version of this problem which captures the two AI challenges while abstracting away from robot-specific details. Specifically, in line with most work in real-time heuristic search (e.g., Furcy & Koenig, 2000; Shimbo & Ishida, 2003; Koenig, 2004; Botea, Müller, & Schaeffer, 2004; Hernández & Meseguer, 2005a, 2005b; Likhachev & Koenig, 2005; Sigmundarson & Björnsson, 2006; Koenig & Likhachev, 2006) we consider an agent on a finite search graph with the task of traveling a path from its current state to a given goal state. Within this context we measure the amount of planning the agent conducts per action and the length of the path traveled between the start and the goal locations. These two measures are antagonistic as reducing the amount of planning per action leads to suboptimal actions and results in longer paths. Conversely, shorter paths require better actions that can be obtained by larger planning effort per action.

We use navigation in grid world maps derived from computer games as a testbed. In such games, an agent can be tasked to go to any location on the map from its current location. Examples include real-time strategy games (e.g., Blizzard, 2002), first-person shooters (e.g., id Software, 1993), and role-playing games (e.g., BioWare Corp., 1998). Size and complexity of game maps as well as the number of simultaneously moving units on such maps continues to increase with every new generation of games. Nevertheless, each game unit or agent must react quickly to the user's command regardless of the map's size and complexity. Consequently, game companies impose a time-per-action limit on their pathfinding algorithms. For instance, Bioware Corp., a major game company that we collaborate with, sets the limit to 1-3 ms for all units computing their paths at the same time.

Search algorithms that produce an entire solution before the agent takes its first action (e.g., A* of Hart et al., 1968) lead to increasing action delays as map size increases. Numerous optimizations have been suggested to remedy these problems and decrease the delays (for a recent example deployed in a forthcoming computer game refer to Sturtevant, 2007). Real-time search addresses the problem in a fundamentally different way. Instead of computing a complete, possibly abstract, solution before the first action is to be taken, real-time search algorithms compute (or plan) only a few first actions for the agent to take. This is usually done by conducting a lookahead search of fixed depth (also known as "search horizon", "search depth" or "lookahead depth") around the agent's current state and using a heuristic (i.e., an estimate of the remaining travel cost) to select the next few actions. The actions are then taken and the planning-execution cycle repeats (e.g., Korf, 1990). Since the goal state is not reached by most such local searches, the agent runs the risks of heading into a dead end or, more generally, selecting suboptimal actions. To address this problem, real-time heuristic search algorithms update (or learn) their heuristic function with experience. Most existing algorithms do a constant amount of planning (i.e., lookahead search) per action. As a result, they tend to waste CPU cycles when the heuristic function is fairly accurate and, conversely, do not plan enough when the heuristic function is particularly inaccurate. Additionally, they compute heuristic with respect to a distant global goal state which can put unrealistic requirements on heuristic accuracy as we demonstrate in this paper.

In this paper we address both problems by making the following three contributions. First, we propose two ways for selecting lookahead search depth dynamically, on a per action basis. Second, we propose a way for selecting intermediate subgoals on a per action basis. Third, we apply these extensions to the classic LRTA* (Korf, 1990) and the state-of-the-art real-time PR LRTS (Bulitko, Sturtevant, Lu, & Yau, 2007) and demonstrate the improvements in performance. The resulting algorithms are the new state of the art in real-time search. To illustrate, on large computer game maps the new algorithms find paths within 7% of the optimal while expanding only a single state for any action. For comparison, the previous state-of-the-art, PR LRTS, is 15 times slower per action while finding paths that are between two and three times more suboptimal. Furthermore, the dynamically controlled LRTA* and PR LRTS are one to two orders of magnitude faster per action than A*, weighted A* and the state-of-the-art Partial Refinement A* (PRA*) (Sturtevant & Buro, 2005). Finally, unlike A* and its modern extensions used in games, the new algorithms are provably real-time and do not slow down as maps become larger.

The rest of the paper is organized as follows. In Section 2 we formulate the problem of real-time heuristic search and show how the core LRTA* algorithm can be extended with dynamic lookahead and subgoal selection. Section 3 analyzes related research. Section 4 provides intuition for dynamic control in search. In Section 5 we describe two approaches to dynamic lookahead selection: one based on induction of decision-tree classifiers (Section 5.1) and one based on precomputing a depth table using state abstraction (Section 5.2). In Section 6 we present an approach to selecting subgoals dynamically. Section 7 evaluates the efficiency of these extensions in the domain of pathfinding. We conclude with a discussion of applicability of the new approach to general planning.

This paper extends our conference publication (Bulitko, Björnsson, Luštrek, Schaeffer, & Sigmundarson, 2007) with a new set of features for the decision tree approach, a new way of selecting subgoals, an additional real-time heuristic search algorithm (PR LRTA*) extended with dynamic control, numerous additional experiments and a more detailed presentation.

2. Problem Formulation

We define a heuristic search problem as a directed graph containing a finite set of states and weighted edges, with a single state designated as the *goal state*. At every time step, a search agent has a single *current state*, vertex in the search graph, and takes an action by traversing an out-edge of the current state. Each edge has a positive cost associated with it. The total cost of edges traversed by an agent from its start state until it arrives at the goal state is called the *solution cost*. We require algorithms to be *complete* and produce a path from start to goal in a finite amount of time if such a path exists. In order to guarantee completeness for real-time heuristic search we make the assumption of safe explorability of our search problems. Namely, all costs are finite and the goal state is reachable from any state that the agent can possibly reach from its start state.

Formally, all algorithms discussed in this paper are applicable to any such heuristic search problem. To keep the presentation focused and intuitive as well as to afford a large-scale empirical evaluation, we will use a particular type of heuristic search problems, pathfinding in grid worlds, for the rest of the paper. However, we will discuss applicability of the new methods we suggest to other heuristic search problems in Section 5.3 and to general planning problems in Section 9.

In computer-game map settings, states are vacant square grid cells. Each cell is connected to four cardinally (i.e., west, north, east, south) and four diagonally neighboring cells. Outbound edges of a vertex are moves available in the corresponding cell and in the rest of the paper we will use the terms *action* and *move* interchangeably. The edge costs are 1 for cardinal moves and $\sqrt{2}$ for diagonal moves. An agent plans its next action by considering states in a local search space surrounding its current position. A *heuristic function* (or simply *heuristic*) estimates the (remaining) travel cost between a state and the goal. It is used by the agent to rank available actions and select the most promising one. In this paper we consider only admissible heuristic functions which do not overestimate the actual remaining cost to the goal. An agent can modify its heuristic function in any state to avoid getting stuck in local minima of the heuristic function, as well as to improve its action selection with experience.

The defining property of real-time heuristic search is that the amount of planning the agent does per action has an upper bound that does not depend on the problem size. We enforce this property by setting a *real-time cut-off* on the amount of planning for any action. Any algorithm that exceeds such a cut-off is discarded. Fast planning is preferred as it guarantees the agent's quick reaction to a new goal specification or to changes in the environment. We measure mean *planning time* per action in terms of CPU time as well as a machine-independent measure – the *number of states expanded* during planning. A state is called expanded if all of its successor states are considered/generated in search. The second performance measure of our study is *sub-optimality* defined as the ratio of the solution cost found by the agent to the minimum solution cost. Ratios close to one indicate near-optimal solutions.

The core of most real-time heuristic search algorithms is an algorithm called Learning Real-Time A* (LRTA*) (Korf, 1990). It is shown in Figure 1 and operates as follows. As long as the goal state $s_{\text{global goal}}$ is not reached, the algorithm interleaves planning and execution in lines 4 through 7. In our generalized version we added a new step at line 3 for selecting a search depth d and goal s_{goal} individually at each execution step (the original algorithm uses fixed d and $s_{\text{global goal}}$ for all planning searches). In line 4, a d-ply breadth-first search with duplicate detection is used to find frontier states precisely d actions away from the current state s. For each frontier state $\hat{s}$, its value is the sum of the cost of a shortest path from s to $\hat{s}$, denoted by $g(s, \hat{s})$, and the estimated cost of a shortest path from $\hat{s}$ to s_{goal} (i.e., the heuristic value $h(\hat{s}, s_{\text{goal}})$). We use the standard path-max technique (Mero, 1984) to deal with possible inconsistencies in the heuristic function when computing $g + h$ values. As a result, $g + h$ values never decrease along any branch of such a lookahead tree. The state that minimizes the sum is identified as s_{frontier} in line 5. The heuristic value of the current state s is updated in line 6 (we keep separate heuristic tables for the different goals). Finally, we take one step towards the most promising frontier state s_{frontier} in line 7.

3. Related Research

Most algorithms in single-agent real-time heuristic search use fixed search depth, with a few notable exceptions. Russell and Wefald (1991) proposed to estimate the utility of expanding a state and use it to control lookahead search on-line. To do so one needs to estimate how likely an additional search is to change an action's estimated value. Inaccuracies in such estimates and the overhead of meta-level control led to "reasonable but unexciting" benefits in combinatorial puzzle and pathfinding. An additional problem is the relatively low branching factor of combinatorial puzzles which makes it difficult to eliminate parts of search space early on. The same problem is likely to occur in grid-

```
LRTA*(s_start, s_global goal)
s ← s_start
while s ≠ s_global goal do
  select search depth d and goal s_goal
  expand successor states up to d actions away, generating a frontier
  find a frontier state s_frontier with the lowest g(s, s_frontier) + h(s_frontier, s_goal)
  update h(s, s_goal) to g(s, s_frontier) + h(s_frontier, s_goal)
  change s one step towards s_frontier
end while
```

Figure 1: LRTA* algorithm with dynamic control.

based pathfinding. Finally, their method adds substantial implementation complexity and requires non-trivial changes to the underlying search algorithm. In contrast, our approach to search depth selection can be easily interfaced with any real-time search algorithm with a search depth parameter without modifying the existing code.

Ishida (1992) observed that LRTA*-style algorithms tend to get trapped in local minima of their heuristic function, termed "heuristic depressions". The proposed remedy was to switch to a limited A* search when a heuristic depression is detected and then use the results of the A* search to correct the depression at once. This is different from our approach in two ways: first, we do not need a mechanism to decide when to switch between real-time and A* search and thus avoid the need to hand-tune control parameters of Ishida's control module. Instead, we employ an automated approach to decide on search horizon depth for every action. Additionally, we do not spend extra time filling in all heuristic values within the heuristic depression by A* estimates.

Bulitko (2003a) showed that optimal search depth selection can be highly beneficial in real-time heuristic search. He linked the benefits to avoiding the so-called lookahead pathologies where deeper lookahead leads to worse moves but did not suggest any practical way of selecting lookahead depth dynamically. Such a way was proposed in 2004 via the use of a generalized definition of heuristic depressions (Bulitko, 2004). The proposed algorithm extends the search horizon incrementally until the search finds a way out of the depression. After that all actions leading to the found frontier state are executed. A cap on the search horizon depth is set by the user. The idea of pre-computing a depth table of heuristic values for real-time pathfinding was first suggested by Luštrek and Bulitko (2006). This paper extends their work as follows: (i) we introduce intermediate goals, (ii) we propose an alternative approach that does not require map-specific pre-computation and (iii) we extend and evaluate a state-of-the-art algorithm in addition to the classic LRTA*.

There is a long tradition of search control in two-player search. High-performance game-playing programs for games like chess and checkers rely extensively on search to decide on which actions to take. The search is performed under strict real-time constraints where programs have typically only minutes or seconds for deliberating on the next action. Instead of using a fixed-depth lookahead strategy the programs employ sophisticated search control mechanisms for maximizing the quality of their action decisions within the given time constraints. The search control techniques can be coarsely divided into three main categories: move ordering, search extensions/reductions, and time allotment. One of the earlier works on dynamic move ordering is the history heuristic technique (Schaeffer, 1989), and more recent attempts include work on training neural networks (Kocsis, 2003). There exist a large variety of techniques for adjusting the search horizon for different branches within the game tree; interesting continuations are explored more deeply while less promising ones are terminated prematurely. Whereas most of the early techniques were static, the research focus has shifted towards more dynamic control as well using machine-learning approaches for automatic parameterization (Buro, 2000; Björnsson & Marsland, 2003). To the best of our knowledge, none of these techniques have been applied to single-agent real-time search.

4. Intuition for Dynamic Search Control

It has been observed in the literature that common heuristic functions are not uniformly inaccurate (Pearl, 1984). Namely, they tend to be more accurate closer to the goal state and less accurate farther away. The intuition for this fact is as follows: heuristic functions usually ignore certain constraints of the search space. For instance, the Manhattan distance heuristic in a sliding tile puzzle would be perfectly accurate if the tiles could pass through each other. Likewise, Euclidian distance on a map ignores obstacles. The closer a state is to a goal the fewer constraints a heuristic function is likely to ignore and, as a result, the more accurate (i.e., closer to the optimal solution cost) the heuristic is likely to be.

This intuition motivates adaptive search control in real-time heuristic search. First, when heuristic values are inaccurate, the agent should conduct a deeper lookahead search to compensate for the inaccuracies and maintain the quality of its actions. Deeper lookaheads have been generally found beneficial in real-time heuristic search (Korf, 1990), though lookahead pathologies (i.e., detrimental effects of deeper lookaheads on action quality) have been observed as well (Bulitko, Li, Greiner, & Levner, 2003; Bulitko, 2003b; Luštrek, 2005; Luštrek & Bulitko, 2006). As an illustration, consider Figure 2. Every state on the map is shaded according to the minimum lookahead depth that an LRTA* agent should use to select an optimal action. Darker shades correspond to deeper lookahead depths. Notice that many areas are bright white, indicating that the shallowest lookahead of depth one will be sufficient. We use this intuition for our first control mechanism: dynamic selection of lookahead depth in Section 5.

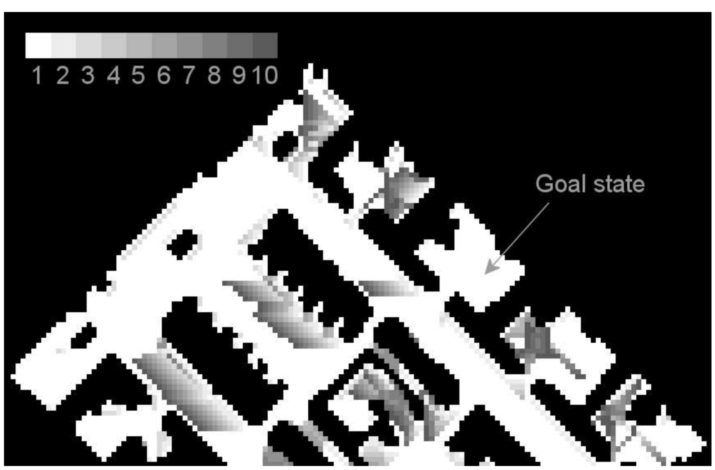

Figure 2: A partial grid world map from a computer game "Baldur's Gate" (BioWare Corp., 1998). Shades of grey indicate optimal search depth values with white representing one ply. Completely black cells are impassable obstacles (e.g., walls).

Dynamic search depth selection helps eliminate wasted computation by switching to shallower lookahead when the heuristic function is fairly accurate. Unfortunately, it does not help when the heuristic function is grossly inaccurate. Instead, it calls for very deep lookahead in order to select an optimal action. Such a deep search tremendously increases planning time and, sometimes, leads to violating a real-time cut-off on planning time per move. To address this issue, in Section 6 we propose our second control mechanism: dynamic selection of subgoals. The idea is straightforward: if being far from the goal leads to grossly inaccurate heuristic values, let us move the goal closer to the agent, thereby improving heuristic accuracy. We do this by computing the heuristic function with respect to an intermediate, and thus nearby, goal as opposed to a distant global goal — the final destination of an agent. Since an intermediate goal is closer than the global goal, the heuristic values of states around an agent will likely be more accurate and thus the search depth picked by our first control mechanism is likely to be shallower. Once the agent gets to an intermediate goal, a next intermediate goal is selected so that the agent makes progress towards its actual global goal.

5. Dynamic Search Depth Selection

First, we define *optimal search depth* as follows. For each $(s, s_{\text{global goal}})$ state pair, a true optimal action $a^*(s, s_{\text{global goal}})$ is to take an edge that lies on an optimal path from s to $s_{\text{global goal}}$ (there can be more than one optimal action). Once $a^*(s, s_{\text{global goal}})$ is known, we can run a series of progressively deeper LRTA* searches from state s. The shallowest search depth that yields $a^*(s, s_{\text{global goal}})$ is the optimal search depth $d^*(s, s_{\text{global goal}})$. Not only may such search depth forfeit LRTA*'s real-time property but it is also impractical to compute. Thus, in the following subsections we present two different practical approaches to *approximating* optimal search depth. Each of them equips LRTA* with a dynamic search depth selection (i.e., realizing the first part of line 3 in Figure 1). The first approach uses a decision-tree classifier to select the search depth based on features of the agent's current state and its recent history. The second approach uses a pre-computed depth database based on an automatically built state abstraction.

5.1 Decision-Tree Classifier Approach

An effective classifier needs input features that are not only useful for predicting the optimal search depth, but are also efficiently computable by the agent in real time. The features we use for our classifier were selected as a compromise between these two considerations, as well as for being domain independent. The features were calculated based on properties of states an agent has recently visited, as well as features gathered by a shallow pre-search from an agent's current state. Example features are: the distance from the state the agent was in n steps ago, estimate of the distance to agent's goal, the number of states visited during the pre-search phase that have updated heuristics. In Appendix A all the features are listed and the rationale behind them is explained.

The classifier predicts the optimal search depth for the current state. The optimal depth is the shallowest search depth that returns an optimal action. For training the classifier we must thus label our training states with optimal search depths. However, to avoid pre-computing optimal actions, we make the simplifying assumption that a deeper search always yields a better action. Consequently, in the training phase the agent first conducts a lookahead search to a pre-defined maximum depth, $d_{\max}$, to derive the "optimal" action (under our assumption). The choice of the maximum depth is domain dependent and would typically be set as the largest depth that still guarantees the search to return within the acceptable real-time requirement for the task at hand. Then a series of progressively shallower searches are performed to determine the shallowest search depth, d^*_{DT}, that still returns the "optimal" action. During this process, if at any given depth an action is returned that differs from the optimal action, the progression is stopped. This enforces all depths from d^*_{DT} to d_{max} to agree on the best action. This is important for improving the overall robustness of classification, as the classifier must generalize over a large set of states. The depth d^*_{DT} is set as the class label for the vector of features describing the current state.

Once we have a classifier for choosing the lookahead depth, LRTA* can be augmented with it (line 3 in Figure 1). The overhead of using the classifier consists of the time required for collecting the features and running them through the classifier. Its overhead is negligible as the classifier itself can be implemented as a handful of nested conditional statements. Collecting the features takes somewhat more time but, with a careful implementation, such overhead can be made negligible as well. Indeed, the four history-based features are all efficiently computed in small constant time, and by keeping the lookahead depth of the pre-search small (e.g., one or two) the overhead of collecting the pre-search features is usually dwarfed by the time the planning phase (i.e., the lookahead search) takes. The process of gathering training data and building the classifier is carried out off-line and its time overhead is thus of a lesser concern.

5.2 Pattern Database Approach

A naïve approach would be to precompute the optimal depth d^* for each (s, s_{goal}) state pair. There are two problems with this approach. First, $d^*(s, s_{\text{goal}})$ is not *a priori* upper-bounded independently of the map size, thereby forfeiting LRTA*'s real-time property. Second, pre-computing $d^*(s, s_{\text{goal}})$ or $a^*(s, s_{\text{goal}})$ for all pairs of (s, s_{goal}) states on, for instance, a 512×512 cell computer game map has prohibitive time and space complexity. We solve the first problem by capping $d^*(s, s_{\text{goal}})$ at a fixed constant $c \geq 1$ (henceforth called *cap*). We solve the second problem by using an automatically built abstraction of the original search space. The entire map is partitioned into regions (or abstract states) and a single search depth value is pre-computed for each pair of *abstract* states. During run-time a single search depth value is shared by all children of the abstract state pair (Figure 3). The search depth values are stored in a table which we will refer to as *pattern database* or PDB for short. In the past, pattern databases have been used to store approximate heuristic values (Culberson & Schaeffer, 1998) and important board features (Schaeffer, 2000). Our work appears to be the first use of pattern databases to store search depth values.

Computing search depths for abstract states speeds up pre-computation and reduces memory overhead (both important considerations for commercial computer games). In this paper we use previously published clique abstraction (Sturtevant & Buro, 2005). It preserves the overall topology of a map but requires storing the abstraction links explicitly.[1] The clique abstraction works by finding fully connected subgraphs (i.e., the cliques) of the original graph and abstracting all states within such a clique into a single abstract state. Two abstract states are connected by an abstract action if and only if there is a single original action that leads from a state in the first clique to a state in the single clique (Figure 4). The costs of the abstract actions are computed as Euclidean distances between average coordinates of all states in the cliques.

In typical grid world computer-game maps, a single application of clique abstraction reduces the number of states by a factor of two to four. On average, at the abstraction level of five (i.e., after five applications of the abstraction procedure), each region contains about one hundred original

1. An alternative is to use the regular rectangular tiles (e.g., Botea et al., 2004).

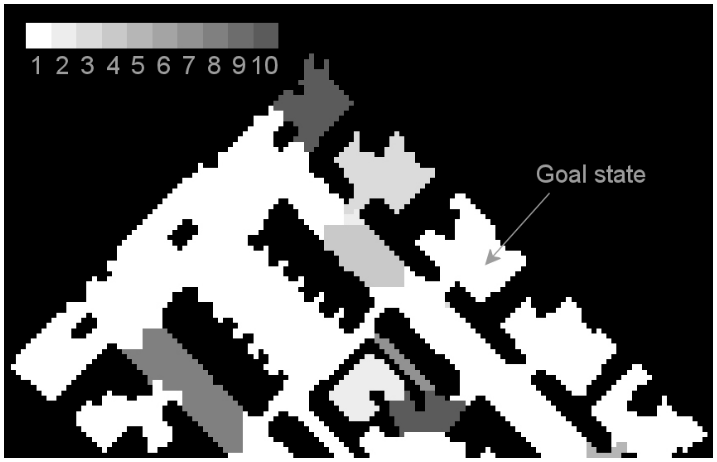

Figure 3: A single optimal lookahead depth value shared among all children of an abstract state. This is a memory-efficient approximation to the true per-ground-state values in Figure 2.

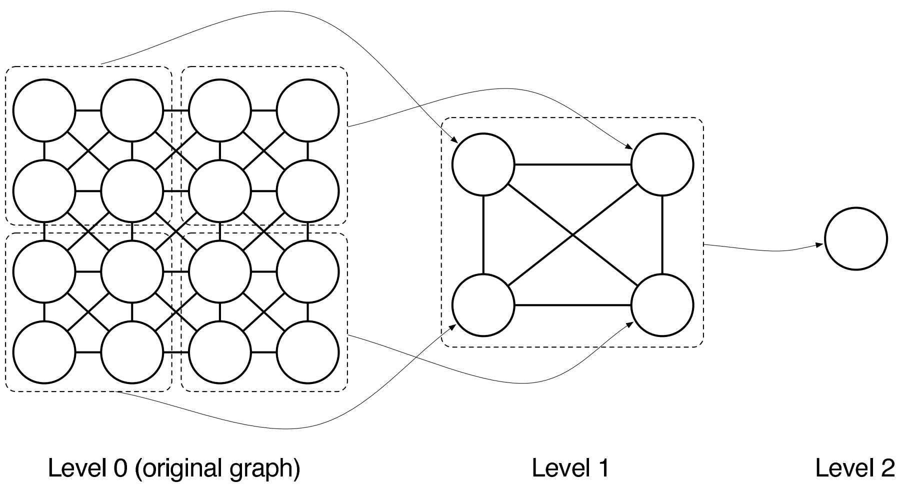

Figure 4: Two iterations of the clique abstraction procedure produce two abstract levels from the ground-level search graph.

(or ground-level) states. Thus, a single search depth value is shared among about ten thousand state pairs. As a result, five-level clique abstraction yields a four orders of magnitude reduction in memory and about two orders of magnitude reduction in pre-computation time (as analyzed later). On the downside, higher levels of abstraction effectively make the search depth selection less and less dynamic as the same depth value is shared among progressively more states. The abstraction level for a pattern database is a control parameter that trades pre-computation time and pattern database size for on-line performance of the algorithm that uses such a database.

Two alternatives to storing the optimal search depth are to store an optimal action or the optimal heuristic value. The combination of abstraction and real-time search precludes both of them. Indeed, sharing an optimal action computed for a single ground-level representative of an abstract region among all states in the region may cause the agent to run into a wall (Figure 5, left). Likewise, sharing a single heuristic value among all states in a region leaves the agent without a sense of

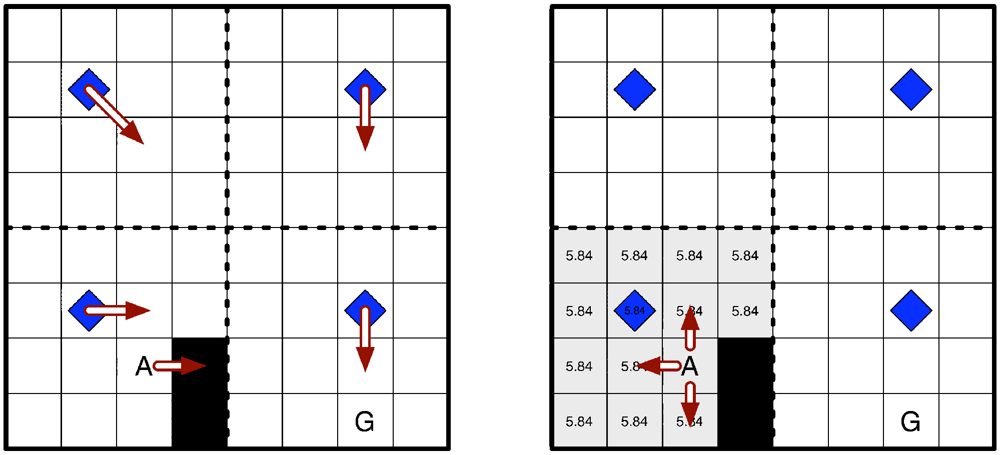

Figure 5: Goal is shown as G, agent as A. Abstract states are the four tiles separated by dashed lines. Diamonds indicate representative states for each tile. **Left:** Optimal actions are shown for each representative of an abstract tile; applying the optimal action of the agent's tile in the agent's current location leads into a wall. **Right:** Optimal heuristic value (h^*) for lower left tile's representative state (5.84) is shared among all states of the tile. As a result, the agent has no preference among the three legal actions shown.

direction as all states in its vicinity would look equally close to the goal (Figure 5, right). This is in contrast to sharing a heuristic value among all states within an abstract state (known as "pattern") when using optimal non-real-time search algorithms such as A* or IDA* (Culberson & Schaeffer, 1996). In the case of real-time search, agents using either alternative are not guaranteed to reach a goal, let alone minimize travel. On the contrary, sharing the search depth among any number of ground-level states is safe because LRTA* is complete for any search depth.

We compute a single depth table per map off-line (Figure 6). In line 1 the state space is abstracted ℓ times. Lines 2 through 7 iterate through all pairs of abstract states. For each pair (s', s'_{goal}), representative ground-level states s and s_{goal} (i.e., ground-level states closest to centroids of the regions) are picked and the optimal search depth value d^* is calculated for them. To do this, Dijkstra's algorithm (Dijkstra, 1959) is run over the ground-level search space (V, E) to compute the true minimal distances from any state to s_{goal}. Once the distances are known for all successors of s, an optimal action $a^*(s, s_{\text{goal}})$ can be computed greedily. Then the optimal search depth $d^*(s, s_{\text{goal}})$ is computed as previously described and capped at c (line 5). The resulting value is stored for the pair of abstract states (s', s'_{goal}) in line 6. Figures 2 and 3 show optimal search depth values for a single goal state on a grid world game map with and without abstraction respectively.

During run-time, an LRTA* agent going from state s to state s_{goal} takes its search depth from the depth table value for the pair (s', s'_{goal}), where s' and s'_{goal} are images of s and s_{goal} under an ℓ-level abstraction. The additional run-time complexity is minimal as s', s'_{goal}, $d(s', s'_{\text{goal}})$ can be computed with a small constant-time overhead on each action.

In building such a pattern database Dijkstra's algorithm is run V_ℓ times[2] on the graph (V, E) – a time complexity of $O(V_\ell(V \log V + E))$ on sparse graphs (i.e., $E = O(V)$). The optimal search depth is computed V_ℓ^2 times. Each time, there are at most c LRTA* invocations with the total

2. For brevity, we use V and E to mean both sets of vertices/edges and their sizes (i.e., $|V|$ and $|E|$).

BuildPatternDatabase(V, E, c, ℓ)

apply an abstraction procedure ℓ times to (V, E) to compute abstract space $S_\ell = (V_\ell, E_\ell)$
for each pair of states $(s', s'_{\text{goal}}) \in V_\ell \times V_\ell$ **do**
  select $s \in V$ as a representative of $s' \in V_\ell$
  select $s_{\text{goal}} \in V$ as a representative of $s'_{\text{goal}} \in V_\ell$
  compute c-capped optimal search depth value d^* for state s with respect to goal s_{goal}
  store capped d^* for pair (s', s'_{goal})
end for

Figure 6: Pattern database construction.

complexity of $O(b^c)$ where b is the maximum degree of V. Thus, the overall time complexity is $O(V_\ell(V \log V + E + V_\ell\, b^c))$. The space complexity is lower because we store optimal search depth values only for all pairs of abstract states: $O(V_\ell^2)$. Table 1 lists the bounds for sparse graphs.

Table 1: Reduction in complexity due to state abstraction.

	no abstraction	**ℓ-level abstraction**	**reduction**
time	$O(V^2 \log V)$	$O(V_\ell V \log V)$	V/V_ℓ
space	$O(V^2)$	$O(V_\ell^2)$	$(V/V_\ell)^2$

5.3 Discussion of the Two Approaches

Selecting the search depth with a pattern database has two advantages. First, the search depth values stored for each pair of abstract states are optimal for their non-abstract representatives, unless either the value was capped or the states in the local search space have been visited before and their heuristic values have been modified. This (conditional) optimality is in contrast to the classifier approach where no optimal actions are ever computed as deeper searches are merely assumed to lead to a better action. The assumption does not always hold – a phenomenon known as lookahead pathology, found in abstract graphs (Bulitko et al., 2003) as well as in grid-based pathfinding (Luštrek & Bulitko, 2006). The second advantage is that we do not need features of the current state, recent history and pre-search. The search depth is retrieved from the depth table simply on the basis of the current state's identifier, such as its coordinates.

The decision-tree classifier approach has two advantages over the depth table approach. First, the classifier training does not need to happen in the same search space that the agent operates in. As long as the training maps used to collect the features and build the decision tree are representative of run-time maps, this approach can run on never-before-seen maps (e.g., user-created maps in a computer game). Second, there is a much smaller memory overhead with this method as the classifier is specified procedurally and no pattern database needs to be loaded into memory.

Note that both approaches assume that there is a structure to the heuristic search problem at hand. Namely, the pattern database approach shares a single search depth value across a region of states. This works most effectively if the states in the region are indeed such that the same lookahead depth is the best for all of them. Our abstraction mechanism forms regions on the basis of the search graph structure, with no regard for search depth. As the empirical study will show, clique abstraction seems to be the right choice for pathfinding. However, the choice of the best abstraction technique for a general heuristic search problem is an open question.

Similarly, the decision-tree approach assumes that states that share similar feature values will also share the best search depth value. It appears to hold to a large extent in our pathfinding domain but feature selection for arbitrary heuristic search problems is an open question as well.

6. Dynamic Goal Selection

The two methods just described allow the agent to select an individual search depth for each state. However, as in the original LRTA*, the heuristic is still computed with respect to the global goal s_{goal}. To illustrate: in Figure 7, the map is partitioned into eight abstract states (in this case, 4×4 square tiles) whose representative states are shown as diamonds (1–8). An optimal path between the agent (A) and the goal (G) is shown as well. A straight-line distance heuristic will ignore the wall between the agent and the goal and will lead the agent in a south-western direction. An LRTA* search of depth 11 or higher is needed to produce an optimal action (such as $\uparrow$). Thus, for any cap value below 11, the agent will be left with a suboptimal action and will spend a long time above the horizontal wall raising heuristic values. Spending large amounts of time in corners and other heuristic depressions is the primary weakness of real-time heuristic search agents and, in this example, is not remedied by dynamic search depth selection due to the cap.

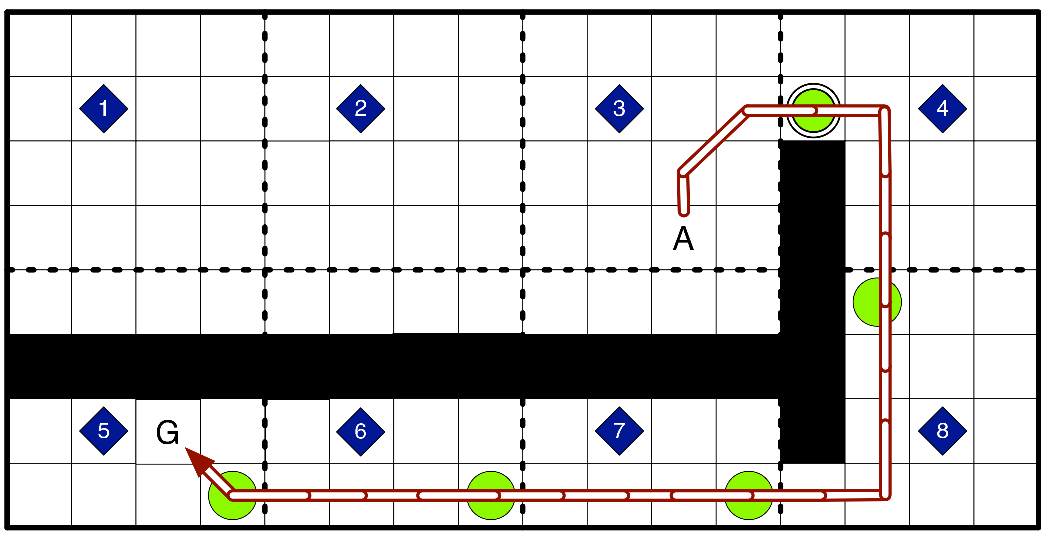

Figure 7: Goal is shown as G, agent as A. Abstract states are the eight tiles separated by dashed lines. Diamonds indicate ground-level representative for each tile. An optimal path is shown. Entry points of the path into abstract states are marked with circles.

5a	compute $s_{\text{intermediate goal}}$ goal for (s, s_{goal})
5b	compute capped optimal search depth value d^* for s with respect to $s_{\text{intermediate goal}}$
6	store $(d^*, s_{\text{intermediate goal}})$ for pair (s', s'_{goal})

Figure 8: Switching s_{goal} to $s_{\text{intermediate goal}}$; replaces lines 5–6 of Figure 6.

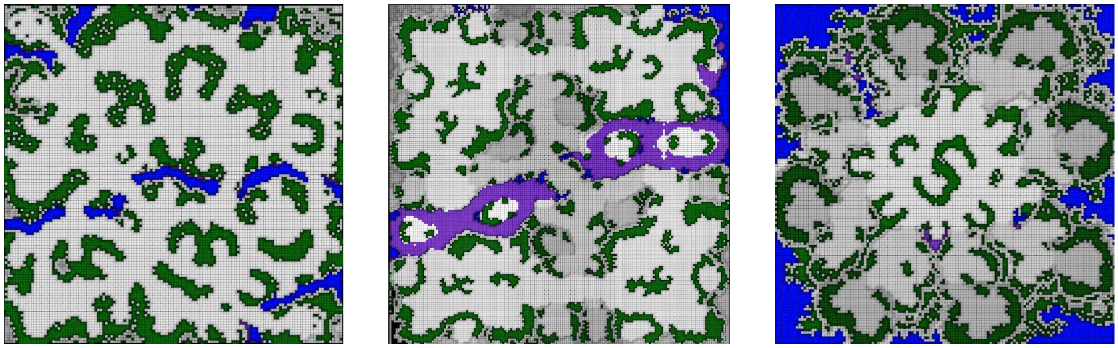

Figure 9: The three maps used in our experiments.

To address this issue, we switch to *intermediate* goals in our pattern-database construction as well as on-line LRTA* operation. In the example in Figure 7 we now compute the heuristic around A with respect to an intermediate goal marked with a double-border circle on the map. Consequently, an eleven times shallower search depth is needed for an optimal action towards the next abstract state (right-most upper tile). Our approach replaces lines 5 - 6 in Figure 6 with those in Figure 8. In line 5a, we compute an intermediate goal $s_{\text{intermediate goal}}$ as the ground-level state where an optimal path from s to s_{goal} enters the next abstract state. These entry points are marked with circles in Figure 7. We compared entry states to centroids of abstract states as intermediate goals (Bulitko et al., 2007) and found the former superior in terms of algorithm's performance. Note that an optimal path is easily available off-line after we run the Dijkstra's algorithm (Section 5.2).

Once an intermediate goal is computed, line 5b computes a capped optimal search depth for s with respect to the intermediate goal $s_{\text{intermediate goal}}$. The depth computation is done as described in Section 5.2. The search depth and the intermediate goal are then added to the pattern database in line 6. At run-time, the agent executes LRTA* with the stored search depth *and* computes the heuristic h with respect to the stored goal (i.e., s_{goal} is set to $s_{\text{intermediate goal}}$ in line 3 of Figure 1). In other words, both search depth and agent's goal are selected dynamically, per action.

This approach works because heuristic functions used in practice tend to become more accurate for states closer to the goal state. Therefore, switching from a distant global goal to a nearby intermediate goal makes the heuristics around the current state s more accurate and leads to a shallower search depth necessary to achieve an optimal action. As a result, not only does the algorithm run more quickly with the shallower search per move but also the search depth cap is reached less frequently and therefore most search depth values actually result in optimal moves.

7. Empirical Evaluation

This section presents results of an empirical evaluation of algorithms with dynamic control of search depth and goals against classic and state-of-the-art published algorithms. All algorithms avoid re-expanding states during planning for each move via a transposition table. We report sub-optimality in the solution found and the average amount of computation per action, expressed in the number of states expanded. We believe that all algorithms can be implemented in such a way that a single expanded state takes the same amount of time. This was not the case in our testbed as some code was more optimized than other. For that reason and to avoid clutter, we report CPU times only in Section 7.7. We used a fixed tie-breaking scheme for all real-time algorithms.

We use grid world maps from a computer game as our testbed. Game maps provide a realistic and challenging environment for real-time search and have been seen in a number of recent publications (e.g., Nash, Daniel, & Felner, 2007; Hernández & Meseguer, 2007). The original maps were sized 161×161 to 193×193 cells (Figure 9). In line with Sturtevant and Buro (2005) and Sturtevant and Jansen (2007), we also experimented with the maps upscaled up to 512×512 – closer in size to maps used in modern computer games. Note that while all three maps depicted in the figure are outdoor-type maps, we also ran preliminary experiments in indoor-type game maps (e.g., the one shown in Figure 2). The trends were similar and we decided to focus on the larger outdoor maps.

There were 100 search problems defined on each of the three original size maps. The start and goal locations were chosen randomly, although constrained such that optimal solution paths cost between 90 and 100 in order to generate more difficult instances. The upscaled maps had the 300 problems upscaled as well. Each data point in the plots below is an average of 300 problems (3 maps $\times 100$ runs each). A different legend entry is used for each algorithm, and multiple points with the same legend entry represent alternative parameter instantiation of the same algorithm. The heuristic function used is octile distance – a natural extension of the Manhattan distance for maps with diagonal actions. To enforce the real-time constraint we disqualified all parameter settings that caused an algorithm to expand more than 1000 states for any move on any problem. Such points were excluded from the empirical evaluation. Maps were known *a priori* off-line in order to build decision-tree classifiers and pattern databases.

We use the following notation to identify all algorithms and their variants: **AlgorithmName (X, Y)** where **X** and **Y** are defined as follows. **X** denotes search depth control: **F** for fixed search depth, **DT** for search depth selected dynamically with a decision tree, **ORACLE** for search depth selected with a decision-tree oracle (see the next section for more details) and **PDB** for search depth selected dynamically with pattern databases. **Y** denotes goal state selection: **G** when the heuristic is computed with respect to a single global goal, **PDB** when the heuristic is computed with respect to an intermediate goal with pattern databases. For instance, the classic LRTA* is **LRTA* (F, G)**.

Our empirical evaluation is organized into eight parts as follows. Section 7.1 describes six algorithms that compute their heuristic with respect to a global goal and discusses their performance. Section 7.2 describes five algorithms that use intermediate goals. Section 7.3 compares global and intermediate goals. Section 7.4 studies the effects of path-refinement with and without dynamic control. Secton 7.5 pits the new algorithms against state-of-the-art real-time and non-real-time algorithms. We then provide an algorithm selection guide for different time limits on planning per move in Section 7.6. Finally, Section 7.7 considers the issue of amortizing off-line pattern-database build time over on-line pathfinding.

7.1 Algorithms with Global Goals

In this subsection we describe the following algorithms that compute their heuristic with respect to a single global goal (i.e., do not use intermediate goals):

1. **LRTA* (F, G)** is Learning Real-Time A* (Korf, 1990). For each action it conducts a breadth-first search of fixed depth d around the agent's current state. Then the first move towards the best depth d state is taken and the heuristic of the agent's previous state is updated using Korf's mini-min rule.[3] We used $d \in \{4, 5, \ldots, 20\}$.

3. Instead of using LRTA* we could have used RTA*. Our experiments showed that in grid pathfinding there is no significant performance difference between the two for a search depth beyond one. Indeed for deeper searches the

2. **LRTA* (DT, G)** is LRTA* in which the search depth d is dynamically controlled by a decision tree as described in Section 5.1. We used the following parameters: $d_{\max} \in \{5, 10, 15, 20\}$ and a history trace of length $n = 60$. For building the decision-tree classifier in WEKA (Witten & Frank, 2005) the *pruning factor* was set to 0.05 and the *minimum number of data items per leaf* to 100 for the original size maps and 25 for the upscaled ones. As opposed to learning a tailor-made classifier for each game map, a single common decision-tree classifier was built based on data collected from all the maps (using 10-fold cross-validation). This was done to demonstrate the ability of the classifier to generalize across maps.

3. **LRTA* (ORACLE, G)** is LRTA* in which the search depth is dynamically controlled by an oracle. Such an oracle always selects the best search depth to produce a move given by LRTA* (F, G) with a fixed lookahead depth $d_{\max}$ (Bulitko et al., 2007). In other words, the oracle acts as a perfect decision-tree and thus sets an upper bound on LRTA* (DT, G) performance. The oracle was run for $d_{\max} \in \{5, 10, 15, 20\}$, and only on the original size maps as it proved prohibitively expensive to compute it for upscaled maps. Note that this is not a practical real-time algorithm and is used only as a reference point in our experiments.

4. **LRTA* (PDB, G)** is LRTA* in which the search depth d is dynamically controlled by a pattern database as described in Section 5.2. For original size maps, we used an abstraction level $\ell \in \{0, 1, \ldots, 5\}$ and a depth cap $c \in \{10, 20, 30, 40, 50, 1000\}$. For upscaled maps, we used an abstraction level $\ell \in \{3, 4, \ldots, 7\}$ and a depth cap $c \in \{20, 30, 40, 50, 80, 3000\}$. Considering the size of our maps, a cap value of 1000 or 3000 means virtually capless search.

5. **K LRTA* (F, G)** is a variant of LRTA* proposed by Koenig (2004). Unlike the original LRTA*, it uses A*-shaped lookahead search space and updates heuristic values for all states within it using Dijkstra's algorithm.[4] The number of states that K LRTA* expands per move took on these values: $\{10, 20, 30, 40, 100, 250, 500, 1000\}$.

6. **P LRTA* (F, G)** is Prioritized LRTA* – a variant of LRTA* proposed by Rayner, Davison, Bulitko, Anderson, and Lu (2007). It uses a lookahead of depth 1 for all moves. However, for every state whose heuristic value is updated, all its neighbors are put onto an update queue, sorted by the magnitude of the update. Thus, the algorithm propagates heuristic function updates in the space in the fashion of Prioritized Sweeping (Moore & Atkeson, 1993). The control parameter (queue size) was set to $\{10, 20, 30, 40, 100, 250, 500, 1000\}$ for original size maps and $\{10, 20, 30, 40, 100, 250\}$ for upscaled maps.

In Figure 10 we evaluate the performance of the new dynamic depth selection algorithms on the original size maps. We see that both the decision-tree and the pattern-database approach do improve significantly upon the LRTA* algorithm, expanding two to three times fewer states for generating solutions of comparable quality. Furthermore, they perform on par with current state-of-the-art real-time search algorithms without abstraction, as can seen when compared with K LRTA* (F, G). The solutions generated are of acceptable quality for our domain (e.g., 50% suboptimal), even when expanding only 100 states per action. Also of interest is that the decision-tree approach performs

likelihood of having multiple actions with equally low $g+h$ cost is very high, reducing the distinction between RTA* and LRTA*. By using LRTA* we can have agents learn over repeated trials.

4. We also experimented with A*-shaped lookahead in our new algorithms and found it inferior to breadth-first lookahead for deeper searches.

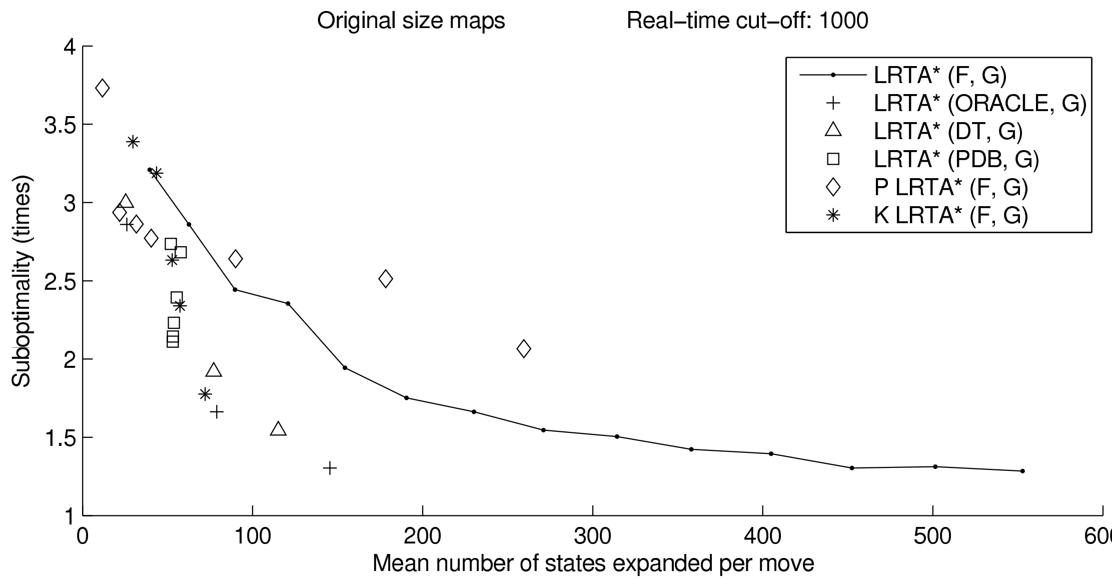

Figure 10: Global-goal algorithms on original size maps.

quite close to its theoretical best case, as seen when compared to LRTA* (ORACLE, G). This shows that the features we use, although seemingly simplistic, do a good job at predicting the most appropriate search depth.

We ran similar sets of experiments on the upscaled maps. However, none of the global goal algorithms generated solutions of acceptable quality given the real-time cut-off (the solutions were between 300 and 1700% suboptimal). The experimental results for the upscaled maps are provided in Appendix B. This shows the inherent limitations of global goal approaches; in large search spaces they cannot compete on equal footing with abstraction-based methods. This brings us to the intermediate goal selection methods.

7.2 Algorithms with Intermediate Goals

In this section we describe the algorithms that use intermediate goals during search. To the best of our knowledge, there is only one previously published *real-time* heuristic search algorithm that does so. Thus, we compare it to the new algorithms proposed in this paper. Given that intermediate goals increase the performance of all algorithms significantly, we present results only on the more challenging upscaled maps. The full roster of algorithms used in this section is as follows:

1. **PR LRTA* (F, G)** is Path Refinement Learning Real-Time Search (Bulitko et al., 2007). The algorithm has two components: it runs LRTA* with a fixed search depth d and a global goal in an abstract space (abstraction level ℓ in a clique abstraction hierarchy) and refines the first move using a corridor-constrained A* running on the original ground-level map.[5] Constraining A* to a small set of states, collectively called a *corridor* by Sturtevant and Buro

5. The algorithm was actually called PR LRTS (Bulitko et al., 2007). Based on findings by Luštrek and Bulitko (2006), we modified it to refine only a single abstract action in order to reduce its susceptibility to lookahead pathologies. This modification is equivalent to substituting the LRTS component with LRTA*. Hence, in the rest of the paper, we call it PR LRTA*.

(2005) or *tunnel* by Furcy (2006), speeds it up and makes it real-time if the corridor size is independent of map size (Bulitko, Sturtevant, & Kazakevich, 2005). While the heuristic is computed in the abstract space with respect to a fixed global goal, the A* component computes a path from the current state to an intermediate goal. This qualifies PR LRTA* to enter this section of empirical evaluation. The control parameters are as follows: abstraction level $\ell \in \{3, 4, \ldots, 7\}$, LRTA* lookahead depth $d \in \{1, 3, 5, 10, 15\}$ and LRTA* heuristic weight $\gamma \in \{0.2, 0.4, 0.6, 1.0\}$ (γ is imposed on g in line 5 of Figure 1).

2. **LRTA* (F, PDB)** is LRTA* with fixed search depth that uses a pattern database only to select intermediate goals. The control parameters are as follows: abstraction level $\ell \in \{3, 4, \ldots, 7\}$ and search depth $d \in \{1, 2, \ldots, 9, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30\}$.

3. **LRTA* (PDB, PDB)** is LRTA* generalized with dynamic search depth and intermediate goal selection with pattern databases as presented in Sections 5.2 and 6. The control parameters are as follows: abstraction level $\ell \in \{3, 4, \ldots, 7\}$ and lookahead cap $c \in \{20, 30, 40, 50, 80, 3000\}$.

4. **PR LRTA* (PDB, G)** is PR LRTA* whose LRTA* component is equipped with dynamic search depth but uses a global (abstract) goal with respect to which it computes its abstract heuristic. The pattern database for the search depth is constructed for the same abstraction level ℓ that the LRTA* component runs on, making the component as optimal as the lookahead cap allows. We used abstraction level $\ell \in \{3, 4, \ldots, 7\}$ and lookahead cap $c \in \{5, 10, 15, 20, 1000\}$. We also ran a version of PR LRTA* (PDB, G) where the pattern database is constructed at abstraction level ℓ_2 above the level ℓ where LRTA* operates (Table 2). We used $(\ell, \ell_2) \in \{(1, 3), (2, 4), (3, 5), (4, 6), (5, 7), (1, 4), (2, 6), (3, 7), (4, 8), (5, 9)\}$.

5. **PR LRTA* (PDB, PDB)** is the same as the two-database version of PR LRTA* (PDB, G) except it uses the second database for goal selection as well as depth selection. We used $(\ell, \ell_2) \in \{(1, 3), (2, 4), (3, 5), (4, 6), (5, 7), (1, 4), (2, 6), (3, 7), (4, 8), (5, 9)\}$ (Table 2).

Table 2: PR LRTA* (PDB, G and PDB) uses LRTA* at abstraction level ℓ to define a corridor within which it refines the path using A*. Dynamic depth (and goal) selection is performed either at abstraction level ℓ or $\ell_2 > \ell$.

Abstraction level	Single abstraction PR LRTA*(PDB,G)	Dual abstraction PR LRTA*(PDB,{G,PDB})
ℓ_2	-	dynamic depth (and goal) selection
ℓ	abstract-level LRTA* dynamic depth selection	abstract-level LRTA*
0	corridor-constrained ground-level A*	corridor-constrained ground-level A*

The pattern database for the algorithms presented above stores a depth value and an intermediate ground-level goal for each pair of abstract states. We present performance results for algorithms with intermediate goals in Sections 7.3–7.6 and then analyze the complexity of pattern database computation and its effects on performance in Section 7.7.

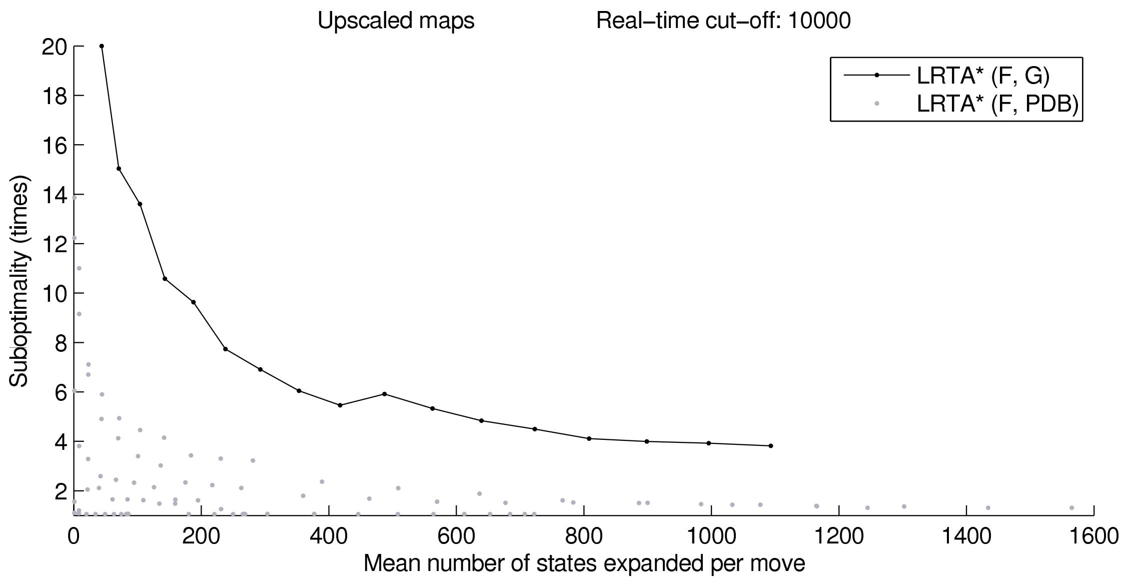

Figure 11: Effects of intermediate goals: LRTA* (F, G) versus LRTA* (F, PDB).

7.3 Global versus Intermediate Goals

Sections 7.1 and 7.2 presented algorithms with global and intermediate goals respectively. In this section we compare algorithms across the two groups. To include LRTA* (PDB, G), we increased the real-time cut-off from 1000 to 10000 for all graphs in this section. We start with the baseline LRTA* with fixed lookahead. The effects of adding intermediate goal selection are dramatic: LRTA* with intermediate goals (F, PDB) finds five times better solutions while being three orders of magnitude faster than LRTA* with global goals (F, G) (see Figure 11). We believe that this is a result of the octile distance heuristic being substantially more accurate around a goal. Consequently, LRTA* (F, PDB) is benefiting from a much better heuristic function.

In the second experiment, we equip both versions with dynamic search depth control and compare LRTA* (PDB, G) with LRTA* (PDB, PDB) in Figure 12. The performance gap is now less dramatic: while the planning speed-up is still around three orders of magnitude, the suboptimality advantage went down from five to two times. Again, note that we had to increase the real-time cut-off by an order of magnitude to get more points in the plot.

Finally, we evaluate what is more beneficial: dynamic depth control or dynamic goal control by comparing the baseline LRTA* (F, G) with LRTA* (PDB, G) and LRTA* (F, PDB) in Figure 13. It is clear that dynamic goal selection is a much stronger addition to the baseline LRTA* than dynamic search depth selection. Dynamic depth selection sometimes actually performs worse than fixed depth, as evidenced by the data points above the LRTA* (F, G) line. This happens primarily with high abstraction levels and small caps. When the optimal lookahead depth is computed at a high abstraction level, the same depth value is shared among many ground-level states. The selected depth value can be beneficial near the entry point into the abstract state, but if the abstract state is too large, the depth is likely to become inappropriate for ground-level states further away. For example, if the optimal depth at the entry point is 1, it can be worse than a moderate fixed depth

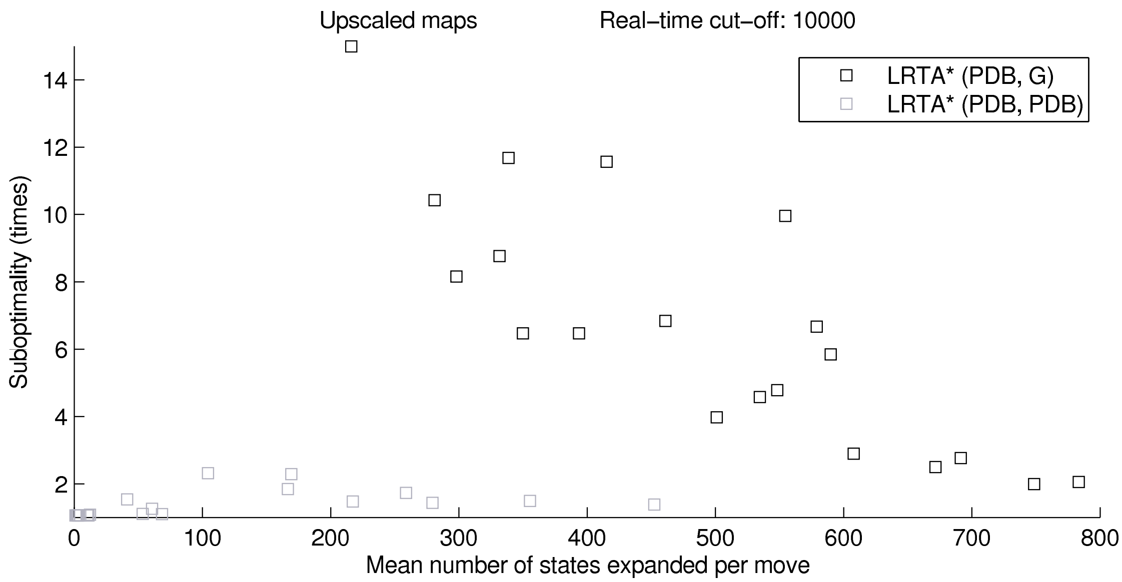

Figure 12: Effects of intermediate goals: LRTA* (PDB, G) versus LRTA* (PDB, PDB).

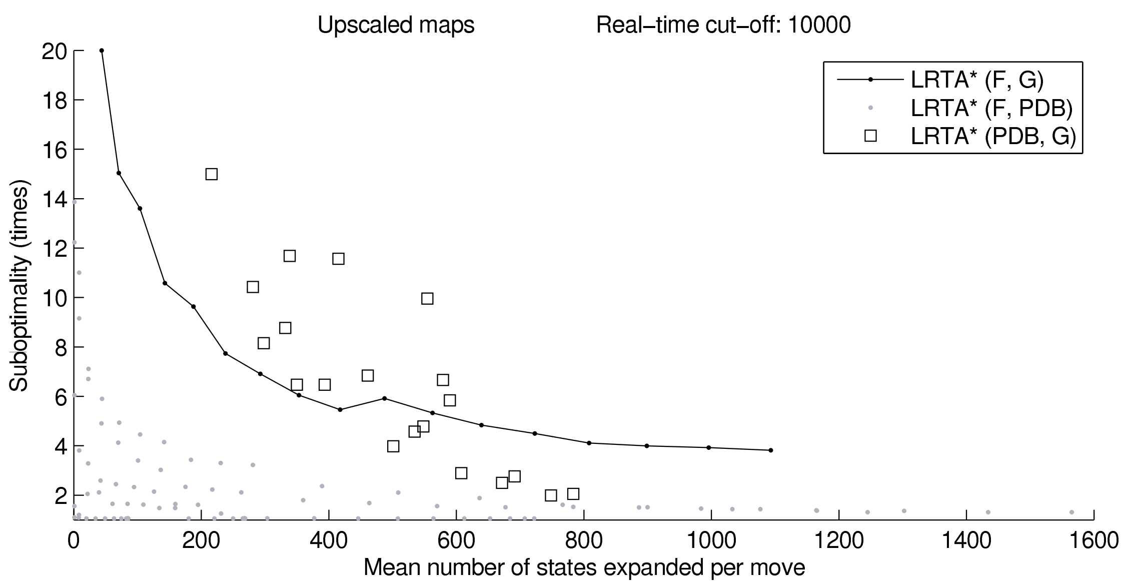

Figure 13: Dynamic search depth control versus dynamic goal control.

in ground-level states far from the entry point. Small caps compound the problem by sometimes preventing the selection of the optimal depth even at the entry point.

While not shown in the plot, running both (i.e., LRTA* (PDB, PDB)) leads to only marginal further improvements. This is because the best parameterizations of LRTA* (F, PDB) already expands only a single state per move virtually at all times. Consequently, the only benefit of adding dynamic depth control is a slight improvement in suboptimality — more on this in the next section.

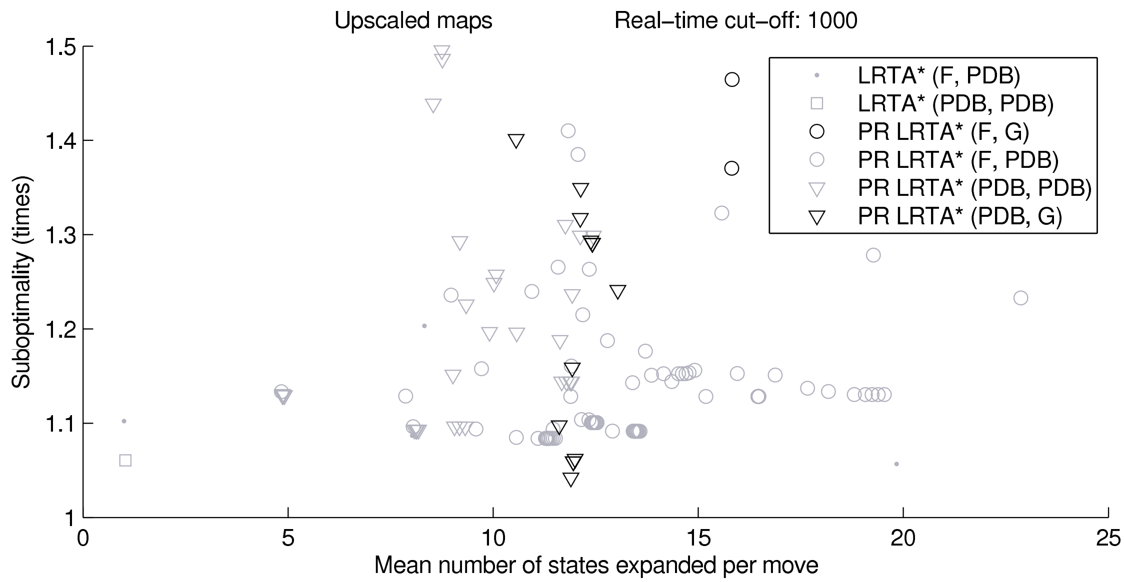

Figure 14: Effects of path refinement: LRTA* versus PR LRTA*.

7.4 Effects of Path Refinement

Path-refinement algorithms (denoted by the 'PR' prefix) run learning real-time search (LRTA*) in an abstract space and refine the path by running A* at the ground level. Non-PR algorithm do not run A* at all as their real-time search happens in the ground-level space. We examine the effects of path-refinement by comparing LRTA* and PR LRTA*. Note that even the statically controlled baseline PR LRTA* (F, G) uses intermediate goals in refining its abstract actions. We match it by using dynamic intermediate goal selection in LRTA*. Thus, we compare four versions of PR LRTA*: (F, G), (PDB, G), (F, PDB) and (PDB, PDB) to two versions of LRTA*: (F, PDB) and (PDB, PDB). The results are found in Figure 14. For the sake of clarity, we show the high performance area by capping the number of states expanded per move at 25 and suboptimality at 1.5.

The best parameterizations of LRTA* find near-optimal solutions while expanding just one state per move at virtually all times. This is astonishing performance because one state expansion per move corresponds to search depth of one and is the fastest possible operation of any algorithm in our framework. Thus, LRTA* (F, PDB) and LRTA* (PDB, PDB) are virtually unbeatable in terms of planning time. On the other hand, PR LRTA* incurs planning overhead due to its path-refinement component (i.e., running a corridor-constrained A*). As a result, PR LRTA* also finds nearly-optimal solutions but incurs at least five times higher planning cost per move. Dynamic control in PR LRTA* results in moderate performance gains.

7.5 Comparison to the Existing State of the Art

Traditionally, computer games have used A* for pathfinding needs (Stout, 2000). As map size and the number of simultaneously planning agents increase, game developers find even highly optimized implementations of A* insufficient. As a result, variants of A* that use state abstraction have been used (Sturtevant, 2007). Another way of speeding up A* is to introduce a weight in computing travel cost through a state. If this is done as $f = \gamma g + h, \gamma \geq 0$ then values of γ below 1 make the agent more greedy (more weight is put on h) which usually leads to fewer states expanded at the price of suboptimal solutions. In this section, we compare the new algorithms to weighted A* (Korf, 1993) and state-of-the-art Partial Refinement A* (PRA*) (Sturtevant & Buro, 2005). Note that neither algorithm is real-time and, thus, the planning times per move are map-size specific. That is, with larger maps, A*'s and PRA*'s planning times per move will increase as these algorithms compute a complete (abstract) path between start and goal states before they take the first move. For instance, for the maps we used PRA* expands 3454 states on its most expensive move. Weighted A* with $\gamma = \frac{1}{5}$ expands 40734 states and the classic A* expands 88138 states on their worst moves. Thus, to include these two algorithms in our comparison we had to effectively remove the real-time cut-off.

The results are found in Table 3. Dynamically controlled LRTA* is one to two orders of magnitude faster in average planning time per move. It produces shorter paths than the existing state-of-the-art real-time algorithm (PR LRTA*) and the fastest weighted A* we tried. The original A* is provably optimal in solution quality and PRA* is nearly optimal. We argue that with hundreds of units simultaneously planning their paths in a computer game, LRTA* (PDB, PDB)'s low planning time per move and real-time guarantees are worth its 6.1% path-length suboptimality (e.g., 106 screen pixels versus the optimal 100 screen pixels).

Table 3: Comparison of high-performance algorithms, best values are in bold. Standard errors are reported after $\pm$.

Algorithm, parameters	**Planning per move**	**Suboptimality (times)**
PR LRTA* (F, G), $\ell = 4, d = 5, \gamma = 1.0$	15.06 ± 0.0722	1.161 ± 0.0177
LRTA* (PDB, PDB), $\ell = 3, c = 3000$	$\mathbf{1.032} \pm 0.0054$	1.061 ± 0.0027
A*	119.8 ± 3.5203	$\mathbf{1} \pm 0.00$
weighted A*, $f = \frac{1}{5}g + h$	24.86 ± 1.4404	1.146 ± 0.0072
PRA*	10.83 ± 0.0829	1.001 ± 0.0003

7.6 Best Solution Quality Under a Time Limit

In this section we identify the algorithms that deliver the best solution quality under a time limit. Specifically, we impose a hard limit on planning time per move, expressed in the number of states expanded. Any algorithm that exceeds the limit on even a single move made in any of the 300 problems on upscaled maps is excluded from consideration. Among the remaining algorithms, we select the one with the highest solution quality (i.e., the lowest suboptimality). The results are found in Table 4. All algorithms expand at least one state per move for some move, leaving the first row empty. LRTA* (F, PDB) $d = 1, \ell = 3$ is the best choice when the time limit is between one and eight states expanded per move. As the limit rises, more expensive but more optimal algorithms become affordable. Note that all the best choices are dynamically controlled algorithms until the time limit rises to 3454 states. At this point, non-real-time PRA* takes over ending the domain of real-time algorithms. Such cross-over point is specific to problem and map sizes. With larger problems/maps, PRA*'s maximum planning time per move will necessarily increase, making it the best choice only for progressively higher planning-time-per-move limits.

Table 4: Best solution quality under a strict limit on planning time per move. Planning time is in the states expanded per move. For the sake of readability, suboptimality is shown as percentage (e.g., $1.102267 = 10.2267\%$).

Planning time limit	Algorithm, parameters	Suboptimality (%)
0	-	-
$[1, 8]$	LRTA* (F, PDB) $d = 1, \ell = 3$	10.2267%
$[9, 24]$	LRTA* (F, PDB) $d = 2, \ell = 3$	8.6692%
$[25, 48]$	LRTA* (F, PDB) $d = 3, \ell = 3$	5.6793%
$[49, 120]$	LRTA* (F, PDB) $d = 4, \ell = 3$	5.6765%
$[121, 728]$	LRTA* (F, PDB) $d = 6, \ell = 3$	5.6688%
$[729, 753]$	LRTA* (F, PDB) $d = 14, \ell = 4$	5.6258%
$[754, 1223]$	PR LRTA* (PDB, G) $c = 15, \gamma = 1.0, \ell = 3$	4.2074%
$[1224, 1934]$	PR LRTA* (PDB, G) $c = 20, \gamma = 1.0, \ell = 3$	3.6907%
$[1935, 3453]$	PR LRTA* (PDB, G) $c = 1000, \gamma = 1.0, \ell = 3$	3.5358%
$[3454, 88137]$	PRA*	0.1302%
$[88138, \infty)$	A*	0%

7.7 Amortization of Pattern-database Build Time

Our pattern-database approach invests time into computing a PDB for each map. In this section we study the amortization of this off-line investment over multiple problem instances. PDB build times on a 3 GHz Pentium CPU are listed in Table 5 for a *single* map. Consider algorithm LRTA* (PDB, PDB) with a cap $c = 20$ and with pattern databases built at level $\ell = 3$. On average, it has solution suboptimality of 1.058 while expanding 1.536 states per move in 31.065 microseconds. Its closest statically controlled competitor is PR LRTA* (F, G) with $\ell = 4, d = 15, \gamma = 0.6$ which has suboptimality of 1.059 while expanding an average of 28.63 states per move in 131.128 microseconds. Thus, LRTA* (PDB, PDB) is about 100 microseconds faster on each move. Consequently, 4.7×10^8 moves are necessary to recoup the off-line PDB build time of 13 hours. With each move taking about 31 microseconds, LRTA* will have a lower total run-time after the first four hours of pathfinding. We computed such recoup times for all parameterizations of LRTA* (PDB, PDB) whose closest statically controlled competitor was slower per move. The results are found in Table 6 and demonstrate that LRTA* (PDB, PDB) recoups the PDB build time in the first 1.4 to 27 hours of its pathfinding time. Note that the numbers are highly implementation and domain-specific. In particular, our code for building PDBs leaves substantial room for optimization. For the completeness sake, we report detailed times in Appendix C.

8. Discussion of Empirical Results

In this section we recap the trends we have observed in the previous sections. Dynamic selection of lookahead with either the decision-tree or the PDB approach helps reduce planning time per move as well as solution suboptimality (Section 7.1). As a result, LRTA* becomes competitive with such modern algorithms as Koenig's LRTA*. However, all real-time search algorithms with global goals do not scale well to large maps.

Table 5: Pattern database for an average 512×512 map, computed for intermediate goals. Database size is listed as the number of abstract state pairs. Suboptimality and planning per move are listed for a representative algorithm: LRTA* (PDB, PDB) with a cap $c = 20$.

Abstraction level	Size	Time	Planning per move	Suboptimality (times)
0	1.1×10^{10}	est. 2 years	-	-
1	7.4×10^{8}	est. 1.5 months	-	-
2	5.9×10^{7}	est. 4 days	-	-
3	6.1×10^{6}	13 hours	1.5	1.058
4	8.6×10^{5}	3 hours	3.2	1.059
5	1.5×10^{5}	1 hour	41.3	1.535
6	3.1×10^{4}	24 minutes	104.4	2.315
7	6.4×10^{3}	10 minutes	169.3	2.284

Table 6: Amortization of PDB build times. For each dynamically controlled LRTA*, we list the statically controlled PR LRTA* that is the closest in terms of solution suboptimality.

LRTA* (PDB, PDB)	PR LRTA* (F, G)	Amortization moves	Amortization run-time
$c = 20, \ell = 3$	$\ell = 4, d = 15, \gamma = 0.6$	4.7×10^{8}	4 hours
$c = 20, \ell = 4$	$\ell = 4, d = 15, \gamma = 0.6$	1.2×10^{8}	1.4 hours
$c = 30, \ell = 3$	$\ell = 4, d = 15, \gamma = 0.6$	5.1×10^{8}	5.1 hours
$c = 40, \ell = 3$	$\ell = 4, d = 15, \gamma = 0.6$	5.3×10^{8}	6 hours
$c = 40, \ell = 4$	$\ell = 4, d = 15, \gamma = 0.4$	3.4×10^{8}	9.3 hours
$c = 50, \ell = 3$	$\ell = 4, d = 15, \gamma = 0.6$	6.2×10^{8}	9 hours
$c = 50, \ell = 4$	$\ell = 4, d = 15, \gamma = 0.6$	6.7×10^{8}	21.1 hours
$c = 80, \ell = 3$	$\ell = 4, d = 15, \gamma = 0.6$	1.1×10^{9}	27 hours

Adding intermediate goals brings even the classic LRTA* on par with the previous state-of-the-art real-time search algorithm PR LRTA* and is a much stronger addition than dynamic lookahead depth selection (Section 7.3). Using both dynamic lookahead depth and subgoals brings further improvements. As Section 7.5 details, LRTA* equipped with both dynamic lookahead depth and subgoal selection expands barely over a state per move and has less than 7% solution suboptimality. While it is not better than previous state-of-the-art algorithms PR LRTA*, PRA* and A* in *both* solution quality and planning time per move, we believe that the trade-offs it makes are appealing in practice. To aid practitioners further, we provide an algorithm selection guide in Section 7.6 which makes it clear that LRTA* with dynamic subgoal selection are the best algorithms when the time per move is severely limited. The speed advantage they deliver over the state-of-the-art PR LRTA* algorithm allows them to recoup the PDB build time in several hours of pathfinding.

9. Current Limitations and Future Work

This project opens several interesting avenues for future research. In particular, it would be worthwhile to investigate performance of the algorithms in this paper in dynamic environments (e.g., a bridge gets destroyed in a real-time strategy game or the goal moves away from the agent).

Another area of future research is application of the proposed algorithms to general planning. Heuristic search has been a successful approach in planning with such planners as ASP (Bonet, Loerincs, & Geffner, 1997), the HSP-family (Bonet & Geffner, 2001), FF (Hoffmann, 2000), SHERPA (Koenig, Furcy, & Bauer, 2002) and LDFS (Bonet & Geffner, 2006). In line with recent planning work (Likhachev & Koenig, 2005) and Bonet and Geffner (2006), we did not evaluate proposed algorithms for general STRIPS-style planning problem. Nevertheless, we do believe that our new real-time heuristic search algorithms may also offer benefits to a wider range of planning problems. Indeed, the core heuristic search algorithm extended in this paper (LRTA*) was previously applied to general planning (Bonet et al., 1997). The extensions we introduced may have a beneficial effect in a similar way to how the B-LRTA* improved the performance of ASP planner. Subgoal selection has been long studied in planning and is a central part of our intermediate-goal depth-table approach. Decision trees for search depth selection are induced from sample trajectories through the space and appear scalable to general planning problems. The only part of our approach that requires solving numerous ground-level problems optimally is pre-computation of optimal search depth in the PDB approach. We conjecture that the approach will still be effective if, instead of computing the optimal search depth based on an optimal action a^*, one were to solve a relaxed planning problem and use the resulting action in place of a^*. The idea of deriving heuristic guidance from solving relaxed problems is quite common to both planning and the heuristic search community.

10. Conclusions

Real-time pathfinding is a non-trivial problem where algorithms must trade solution quality for the amount of planning per move. These two measures are antagonistic and thus we are interested in Pareto optimal algorithms which are not outperformed in both measures by any other algorithms. The classic LRTA* provides a smooth trade-off curve, parameterized by the lookahead depth. Since its introduction in 1990, a variety of extensions have been proposed. The most recent extension, PR LRTS (Bulitko et al., 2005) was the first application of automatic state abstraction in real-time search. In a large-scale empirical study with pathfinding on game maps, PR LRTS outperformed many other algorithms with respect to several antagonistic measures (Bulitko et al., 2007).

In this paper we also employ automatic state abstraction but instead of using it for path-refinement, we pre-compute pattern databases and use them to select the amount of planning and intermediate goals dynamically, per move. Several mechanisms for such dynamic control are proposed and can be used with virtually any existing real-time search algorithm. As a demonstration, we equip both the classic LRTA* and the state-of-the-art PR LRTS with our dynamic control. The resulting improvements are substantial. For instance, LRTA* equipped with PDB-based control for lookahead and intermediate goal selection significantly outperforms the existing state of the art (PR LRTS) simultaneously in planning per move and solution quality. Furthermore, on average it expands only a little more than one state per move which is the minimum amount of planning for an LRTA*-based algorithm.

The new algorithms compare favorably to A* and its state-of-the-art extension, PRA*, which are presently popular industrial choices for pathfinding in computer games (Stout, 2000; Sturtevant, 2007). First, per-move planning time of our algorithms is provably unaffected by any increase in map size. Second, we are two orders of magnitude faster than A* and one order of magnitude faster than PRA* in planning time per move. These improvements come at the price of about 7%

suboptimality, likely to be unnoticed by a computer game player in most scenarios. Thus it appears that not only the new algorithms redefine the state of the art in the real-time search arena but also that they are well-suited for industrial applications.

Acknowledgments

Sverrir Sigmundarson was at the School of Computer Science, Reykjavik University during this project. We appreciate consultation by Robert C. Holte and detailed feedback from the anonymous reviewers. This research was supported by grants from the National Science and Engineering Research Council of Canada (NSERC); Alberta's Informatics Circle of Research Excellence (iCORE); Slovenian Ministry of Higher Education, Science and Technology; Icelandic Centre for Research (RANNÍS); and by a Marie Curie Fellowship of the European Community programme *Structuring the ERA* under contract number MIRG-CT-2005-017284. Special thanks to Nathan Sturtevant for his development and support of HOG.

Appendix A. Decision-Tree Features

We devised two different categories of classifier features: the first consists of features based on the agent's recent history, whereas the second contains features sampled by a shallow pre-search from the agent's current state. Thus, collectively, the features in the two categories make predictions based on both the agent's recent history as well as its current situation.

The first category has the four features listed in Table 7. These features are computed at each execution step. Some of them are aggregated over the most recent states the agent was in, which is done in an incremental fashion for an improved performance. The parameter n is set by the user and controls how long a history to aggregate over. We use the notation s_{-1} to refer to the state the agent was in one step ago, s_{-2} for the state two steps ago, etc.; the agent thus aggregates over states s_{-1}, ..., s_{-n}. Feature f_1 provides a rough estimate of the location of the agent relative to the goal. The distance to the goal state can affect the required lookahead depth, for example because heuristics closer to the goal are usually more accurate. This feature makes it possible for the classifier to make decisions based on that if deemed necessary. Features f_2 (known as *mobility*) and f_3 provide a measure of how much progress the agent has made towards reaching the goal in the past few steps. Frequent state revisits may indicate a heuristic depression and a deeper search is usually beneficial in such situations (Ishida, 1992). Feature f_4 is a measure of inaccuracies and inconsistencies in the heuristic around the agent; again, many heuristic updates may warrant a deeper search.

The features in the second category are listed in Table 8. They are also computed at each execution step. Before the planning phase starts, a shallow lookahead pre-search is performed to gather information about the nearby part of the search space. The types of features in this category can be coarsely divided into features that (i) compute the fraction of states on the pre-search lookahead frontier that satisfy some property, (ii) compare the action chosen by the pre-search to previous actions (either of the previous state or taken the last time the current state was visited), and (iii) check heuristic estimates of the immediate successors of the current state. Feature f_5 is a rough measure of the density of obstacles in the agent's vicinity: the more obstacles there are, the more beneficial a deeper search might be. Feature f_6 is an indicator of the "difficulty" of traversing the local area. If the proportion is high, many states have been updated, possibly suggesting a heuristic depression. As for feature f_7, if a pre-search selects the same action again this might indicate that

Table 7: History based classifier features.

Feature	Description
f_1	The initial heuristic estimate of the distance from the current state to the goal: $h_{\text{octile}}(s, s_{\text{global goal}})$.
f_2	The heuristic estimate of the distance between the current state and the state the agent was in n steps ago: $h(s, s_{-n})$.
f_3	The number of distinct states the agent visited in the last n steps: $\lvert\{s_{-1}, s_{-2}, ..., s_{-n}\}\rvert$.
f_4	The total volume of heuristic updates over the last n steps: $\sum_{i=1}^{n} h_{\text{after update}}(s_{-i}) - h_{\text{before update}}(s_{-i})$ (line 6 in Figure 1).

Table 8: Pre-search based classifier features.

Feature	Description
f_5	The ratio of the actual number of states on the pre-search frontier to the expected number of states if there were no obstacles on the map.
f_6	The fraction of frontier states with an updated heuristic value.
f_7	A boolean feature telling whether the action chosen by the pre-search is the same as the action chosen by the planning phase the last time this state was visited. If this is the first time the state is visited this feature is false.
f_8	A boolean feature telling whether the direction suggested by the pre-search is the same as the direction the agent took the previous step.
f_9	The ratio between the current state's heuristic and the best successor state suggested by the pre-search: $h(s, s_{\text{goal}})/h(\tilde{s}, s_{\text{goal}})$.
f_{10}	A boolean feature telling whether the best action proposed by the pre-search phase would lead to a successor state with an updated heuristic value.
f_{11}	A boolean feature telling whether the heuristic value of the current state is larger than the heuristic value of the best immediate successor found by the pre-search.

the heuristic values in this part of the search space are already mutually consistent and thus only a shallow lookahead is needed; the same applies to feature f_8. Features f_9 to f_{11} compare the current state to the successor state suggested by the pre-search.

Appendix B. Experiments on Upscaled Maps Using Global Goals

Empirical results of running the global-goal algorithms on upscaled maps are shown in Figure 15. The LRTA* (DT, G) shows a significant improvement over LRTA* (F, G), making it comparable in quality to the existing state-of-the-art algorithms: on par with P LRTA* (F, G) and slightly better than K LRTA* (F, G) when allowed to expand over 200 states per move. It is also worth noting that LRTA* (PDB, G) is no longer competitive with the other algorithms and, in fact, does not make the real-time cut-off of 1000 states for any of its parameters combinations (and thus is not shown in the plot). The reason lies with the fact that the problems are simply too difficult for LRTA* to find an optimal move with a small lookahead depth. For instance, with abstraction level $\ell = 3$ and cap

$c = 80$, LRTA* (PDB, G) has suboptimality of 1.36. Unfortunately, its lookahead depth hits the cap in 11% of all visited states. As a result, the algorithm expands an *average* of 1214 states per move which disqualifies it under a cut-off of 1000.

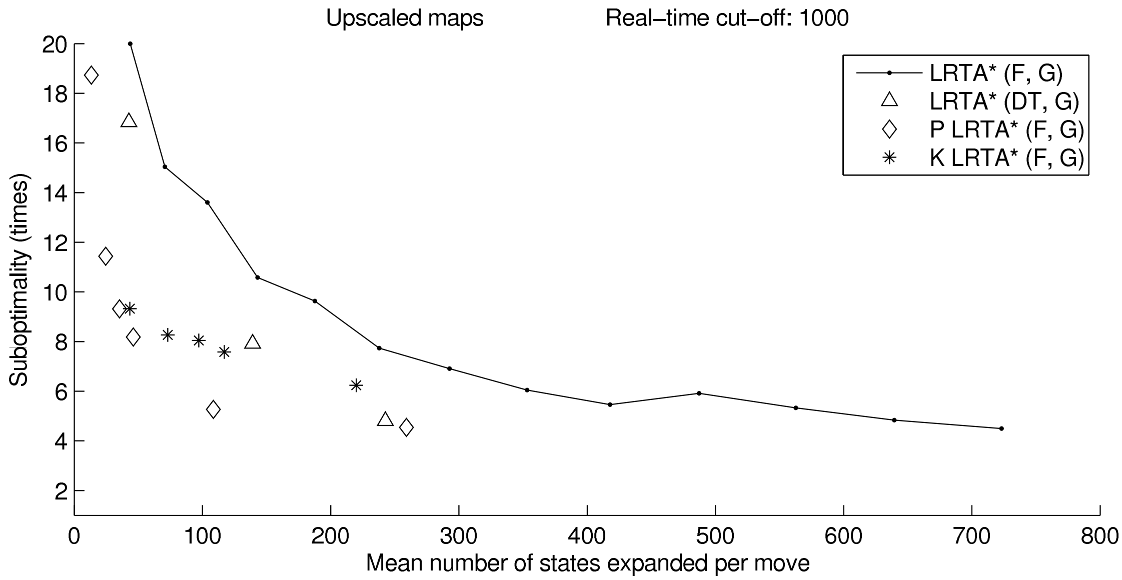

Figure 15: Performance of global-goal algorithms on upscaled maps.

Looking collectively at the small and upscaled up map results, LRTA* (DT, G) demonstrates excellent performance among the global goal algorithms as it is both robust with respect to map upscaling and one of the more efficient ones (the only comparable algorithm is K LRTA* (F, G)). However, within the provided 1000 states cut-off limit, none of the real-time global-goal algorithms returned solutions that would be considered of an acceptable quality in pathfinding. Indeed, even the best solutions found are approximately four times worse than the optimal.

Appendix C. Pattern Database Build Times

In order to operate LRTA* and PR LRTA* that use both lookahead depth and intermediate goals controlled dynamically, we build pattern databases. Each pattern database is built off-line and contains a single entry for each pair of abstract states. There are three types of entries: (i) intermediate goal which is a ground-level entry state in the next abstract state; (ii) capped optimal lookahead depth with respect to the intermediate goal and (iii) optimal lookahead depth with respect to the global goal. When running algorithms with capped lookaheads (i.e., $c < 1000$) we need two databases per map: one containing intermediate goals and one containing capped optimal lookahead depths. When running effectively uncapped algorithms (i.e., $c = 1000$ or $c = 3000$) we also need a third database with lookahead depths for global goals (see Appendix D for further discussion). Tables 5 and 9–12 report build times and LRTA* (PDB, PDB) performance when capped (i.e., when we have to build only two pattern databases). Tables 13 and 14 report build times and performance with effectively no cap (i.e., when we have built all three pattern databases).

Finally, in the interest of speeding up experiments we did not in fact compute pattern databases for *all* pairs of abstract states. Instead, we took advantage of prior benchmark problem availability

Table 9: Pattern databases for an average 512×512 map, computed for intermediate goals. Database size is listed as the number of abstract state pairs. Suboptimality and planning per move are listed for LRTA* (PDB, PDB) with a cap $c = 30$.

Abstraction level	Size	Time	Planning per move	Suboptimality (times)
0	1.1×10^{10}	est. 2 years	-	-
1	7.4×10^{8}	est. 1.5 months	-	-
2	5.9×10^{7}	est. 4 days	-	-
3	6.1×10^{6}	13.4 hours	2.1	1.058
4	8.6×10^{5}	3.1 hours	12.3	1.083
5	1.5×10^{5}	1.1 hours	60.7	1.260
6	3.1×10^{4}	24 minutes	166.6	1.843
7	6.4×10^{3}	11 minutes	258.8	1.729

Table 10: Pattern databases for an average 512×512 map, computed for intermediate goals. Database size is listed as the number of abstract state pairs. Suboptimality and planning per move are listed for LRTA* (PDB, PDB) with a cap $c = 40$.

Abstraction level	Size	Time	Planning per move	Suboptimality (times)
0	1.1×10^{10}	est. 2 years	-	-
1	7.4×10^{8}	est. 1.5 months	-	-
2	5.9×10^{7}	est. 4 days	-	-
3	6.1×10^{6}	13.1 hours	2.7	1.058
4	8.6×10^{5}	3.1 hours	10.2	1.060
5	1.5×10^{5}	1.0 hours	53.4	1.102
6	3.1×10^{4}	24 minutes	217.3	1.474
7	6.4×10^{3}	10 minutes	355.4	1.490

and computed PDBs only for abstract goal states that come into play in the problems that our agents were to solve. Thus, the times in the tables are our estimates for all possible pairs.

Appendix D. Intermediate Goals and Loops

As shown by Korf in his original paper, LRTA* is complete for any lookahead depth when its heuristic is taken with respect to a single global goal. Such completeness guarantee is lost when one uses intermediate goals (i.e., for LRTA* (F, PDB), LRTA* (PDB, PDB) as well as the PR LRTA* counter-parts). Indeed, while in an abstract tile A, the dynamic goal control module will guide the agent towards an entry state in tile B. However, on its way, the agent may stumble in different abstract tile C. As soon as it happens, the dynamic control module may select an entry state in tile A as its new intermediate goal. The unsuspecting agent heads back to A and everything repeats.

To combat such loops we equipped all algorithms that use intermediate goals with a state reentrance detector. Namely, as soon as an agent re-visits a ground-level state, the dynamic control switches from an intermediate goal to the global goal. Additionally, a new lookahead depth is selected. Ideally, such lookahead depth should be the optimal depth with respect to the global goal,

Table 11: Pattern databases for an average 512×512 map, computed for intermediate goals. Database size is listed as the number of abstract state pairs. Suboptimality and planning per move are listed for LRTA* (PDB, PDB) with a cap $c = 50$.

Abstraction level	**Size**	**Time**	**Planning per move**	**Suboptimality (times)**
0	1.1×10^{10}	est. 2 years	-	-
1	7.4×10^{8}	est. 1.5 months	-	-
2	5.9×10^{7}	est. 4 days	-	-
3	6.1×10^{6}	13.6 hours	3.5	1.058
4	8.6×10^{5}	3.1 hours	11.1	1.059
5	1.5×10^{5}	1.0 hours	68.5	1.098
6	3.1×10^{4}	24 minutes	279.4	1.432
7	6.4×10^{3}	11 minutes	452.3	1.386

Table 12: Pattern databases for an average 512×512 map, computed for intermediate goals. Database size is listed as the number of abstract state pairs. Suboptimality and planning per move are listed for LRTA* (PDB, PDB) with a cap $c = 80$.

Abstraction level	**Size**	**Time**	**Planning per move**	**Suboptimality (times)**
0	1.1×10^{10}	est. 2 years	-	-
1	7.4×10^{8}	est. 1.5 months	-	-
2	5.9×10^{7}	est. 4 days	-	-
3	6.1×10^{6}	13.5 hours	6.6	1.058
4	8.6×10^{5}	3.2 hours	22.9	1.059
5	1.5×10^{5}	1.0 hours	109.7	1.087
6	3.1×10^{4}	25 minutes	523.3	1.411
7	6.4×10^{3}	10 minutes	811.5	1.301

capped at c. Unfortunately, computing optimal lookahead depths for global goals is quite expensive off-line (Tables 13 and 14). Given that loops occur fairly infrequently, we do not normally compute optimal lookahead depths for global goals. Instead, when a state re-visit is detected, we switch to global goals and simply set the lookahead to cap c. Doing so saves off-line PDB computation time but sometimes causes the agent to conduct a deeper search (c plies) than really necessary.[6]

An alternative solution to be investigated in future research is to progressively increase lookahead on-line when re-visits are detected (i.e., every time a re-visit occurs, lookahead depth at that state is increased by a certain number of plies).

6. The only exception to this practice were the cases of $c = 1000$ and $c = 3000$ where setting lookahead depth d to c would have immediately disqualified the algorithm, provided a reasonable real-time cut-off. Consequently, for these two cap values, we did invest a large amount of time and computed effectively uncapped optimal lookahead depth with respect to global goals.

Table 13: Pattern databases for an average 512×512 map, computed for *global* goals. Database size is listed as the number of abstract state pairs. Suboptimality and planning per move are listed for LRTA* (PDB, PDB) with a cap $c = 3000$.

Abstraction level	**Size**	**Time**	**Planning per move**	**Suboptimality (times)**
0	1.1×10^{10}	est. 350 years	-	-
1	7.4×10^{8}	est. 25 years	-	-
2	5.9×10^{7}	est. 2 years	-	-
3	6.1×10^{6}	73 days	1.0	1.061
4	8.6×10^{5}	10.3 days	4.8	1.062
5	1.5×10^{5}	1.7 days	27.9	1.133
6	3.1×10^{4}	9.8 hours	86.7	3.626
7	6.4×10^{3}	2.5 hours	174.1	3.474

Table 14: Pattern databases for an average 512×512 map, computed for *global* goals. Database size is listed as the number of abstract state pairs. Suboptimality and planning per move are listed for LRTA* (PDB, G) with a cap $c = 20$.

Abstraction level	**Size**	**Time**	**Planning per move**	**Suboptimality (times)**
0	1.1×10^{10}	est. 12 years	-	-
1	7.4×10^{8}	est. 6 months	-	-
2	5.9×10^{7}	est. 13 days	-	-
3	6.1×10^{6}	38.0 hours	349.9	6.468
4	8.6×10^{5}	7.5 hours	331.6	8.766
5	1.5×10^{5}	2.3 hours	281.0	10.425
6	3.1×10^{4}	52 minutes	298.1	8.155
7	6.4×10^{3}	21 minutes	216.1	14.989

References

BioWare Corp. (1998). Baldur's Gate., Published by Interplay, http://www.bioware.com/bgate/, November 30, 1998.

Björnsson, Y., & Marsland, T. A. (2003). Learning extension parameters in game-tree search. *Inf. Sci*, *154*(3–4), 95–118.

Blizzard (2002). Warcraft 3: Reign of chaos. http://www.blizzard.com/war3.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1–2), 5–33.

Bonet, B., & Geffner, H. (2006). Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 142–151, Cumbria, UK.

Bonet, B., Loerincs, G., & Geffner, H. (1997). A fast and robust action selection mechanism for planning.. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 714–719, Providence, Rhode Island. AAAI Press / MIT Press.

Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of Game Development*, *1(1)*, 7–28.

Bulitko, V. (2003a). Lookahead pathologies and meta-level control in real-time heuristic search. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pp. 13–16, Porto, Portugal.

Bulitko, V. (2003b). Lookahead pathologies and meta-level control in real-time heuristic search. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pp. 13–16.

Bulitko, V. (2004). Learning for adaptive real-time search. Tech. rep. http://arxiv.org/abs/cs.AI/0407016, Computer Science Research Repository (CoRR).

Bulitko, V., Björnsson, Y., Luštrek, M., Schaeffer, J., & Sigmundarson, S. (2007). Dynamic Control in Path-Planning with Real-Time Heuristic Search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 49–56, Providence, RI.

Bulitko, V., Li, L., Greiner, R., & Levner, I. (2003). Lookahead pathologies for single agent search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1531–1533, Acapulco, Mexico.

Bulitko, V., Sturtevant, N., & Kazakevich, M. (2005). Speeding up learning in real-time search via automatic state abstraction. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 1349–1354, Pittsburgh, Pennsylvania.

Bulitko, V., Sturtevant, N., Lu, J., & Yau, T. (2007). Graph Abstraction in Real-time Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)*, *30*, 51–100.

Buro, M. (2000). Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. In van den Herik, H. J., & Iida, H. (Eds.), *Games in AI Research*, pp. 77–96. U. Maastricht.

Culberson, J., & Schaeffer, J. (1996). Searching with pattern databases. In *CSCI (Canadian AI Conference)*, Advances in Artificial Intelligence, pp. 402–416. Springer-Verlag.

Culberson, J., & Schaeffer, J. (1998). Pattern Databases. *Computational Intelligence*, *14*(3), 318–334.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs.. *Numerische Mathematik*, *1*, 269–271.

Furcy, D. (2006). ITSA*: Iterative tunneling search with A*. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, Boston, Massachusetts.

Furcy, D., & Koenig, S. (2000). Speeding up the convergence of real-time search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 891–897.

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Hernández, C., & Meseguer, P. (2005a). Improving convergence of LRTA*(k). In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Workshop on Planning and Learning in A Priori Unknown or Dynamic Domains*, Edinburgh, UK.

Hernández, C., & Meseguer, P. (2005b). LRTA*(k). In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, Edinburgh, UK.

Hernández, C., & Meseguer, P. (2007). Improving real-time heuristic search on initially unknown maps. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), Workshop on Planning in Games*, p. 8, Providence, Rhode Island.

Hoffmann, J. (2000). A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, pp. 216–227.

id Software (1993). Doom., Published by id Software, http://en.wikipedia.org/ wiki/Doom, December 10, 1993.

Ishida, T. (1992). Moving target search with intelligence. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 525–532.

Kitano, H., Tadokoro, S., Noda, I., Matsubara, H., Takahashi, T., Shinjou, A., & Shimada, S. (1999). Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *Man, Systems, and Cybernetics*, pp. 739–743.

Kocsis, L. (2003). *Learning Search Decisions*. Ph.D. thesis, University of Maastricht.

Koenig, S. (2004). A comparison of fast search methods for real-time situated agents. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 864–871.

Koenig, S. (2001). Agent-centered search. *AI Magazine*, *22*(4), 109–132.

Koenig, S., Furcy, D., & Bauer, C. (2002). Heuristic search-based replanning. In *Proceedings of the Int. Conference on Artificial Intelligence Planning and Scheduling*, pp. 294–301.

Koenig, S., & Likhachev, M. (2006). Real-time adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 281–288.

Korf, R. (1985). Depth-first iterative deepening : An optimal admissible tree search. *Artificial Intelligence*, *27*(3), 97–109.

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

Korf, R. (1993). Linear-space best-first search. *Artificial Intelligence*, *62*, 41–78.

Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., & Thrun, S. (2005). Anytime dynamic A*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Likhachev, M., Gordon, G. J., & Thrun, S. (2004). ARA*: Anytime A* with provable bounds on sub-optimality. In Thrun, S., Saul, L., & Schölkopf, B. (Eds.), *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA.

Likhachev, M., & Koenig, S. (2005). A generalized framework for lifelong planning A*. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 99–108.

Luštrek, M. (2005). Pathology in single-agent search. In *Proceedings of Information Society Conference*, pp. 345–348, Ljubljana, Slovenia.

Luštrek, M., & Bulitko, V. (2006). Lookahead pathology in real-time path-finding. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search*, pp. 108–114, Boston, Massachusetts.

Mero, L. (1984). A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, *23*, 13–27.

Moore, A., & Atkeson, C. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, *13*, 103–130.

Nash, A., Daniel, K., & Felner, S. K. A. (2007). Theta*: Any-angle path planning on grids. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 1177–1183.

Pearl, J. (1984). *Heuristics*. Addison-Wesley.

Rayner, D. C., Davison, K., Bulitko, V., Anderson, K., & Lu, J. (2007). Real-time heuristic search with a priority queue. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2372–2377, Hyderabad, India.

Russell, S., & Wefald, E. (1991). *Do the Right Thing: Studies in Limited Rationality*. MIT Press.

Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *PAMI-11*(1), 1203–1212.

Schaeffer, J. (2000). Search ideas in Chinook. In van den Herik, H. J., & Iida, H. (Eds.), *Games in AI Research*, pp. 19–30. U. Maastricht.

Shimbo, M., & Ishida, T. (2003). Controlling the learning process of real-time heuristic search. *Artificial Intelligence*, *146*(1), 1–41.

Sigmundarson, S., & Björnsson, Y. (2006). Value Back-Propagation vs. Backtracking in Real-Time Search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search*, pp. 136–141, Boston, Massachusetts, USA.

Stenz, A. (1995). The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1652–1659.

Stout, B. (2000). The basics of A* for path planning. In *Game Programming Gems*. Charles River Media.

Sturtevant, N. (2007). Memory-efficient abstractions for pathfinding. In *Proceedings of the third conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 31–36, Stanford, California.

Sturtevant, N., & Buro, M. (2005). Partial pathfinding using map abstraction and refinement. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 1392–1397.

Sturtevant, N., & Jansen, R. (2007). An analysis of map-based abstraction and refinement. In *Proceedings of the 7th International Symposium on Abstraction, Reformulation and Approximation*, Whistler, British Columbia.

Witten, I. H., & Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques* (2nd edition). Morgan Kaufmann, San Fransisco.