Dynamic Control in Path-Planning with Real-Time Heuristic Search

Vadim Bulitko[1], Yngvi Björnsson[2], Mitja Luštrek[3], Jonathan Schaeffer[1], Sverrir Sigmundarson[2]

[1] Department of Computing Science
University of Alberta
Edmonton, Alberta
T6G 2E8, CANADA
{bulitko, jonathan}@cs.ualberta.ca

[2] Department of Computer Science
Reykjavik University
Kringlan 1
IS-103 Reykjavik, ICELAND
{yngvi, sverrirs01}@ru.is

[3] Department of Intelligent Systems
Jožef Stefan Institute
Jamova 39
1000 Ljubljana, SLOVENIA
mitja.lustrek@ijs.si

Abstract

Real-time heuristic search methods, such as LRTA*, are used by situated agents in applications that require the amount of planning per action to be constant-bounded regardless of the problem size. LRTA* interleaves planning and execution, with a fixed search depth being used to achieve progress towards a fixed goal. Here we generalize the algorithm to allow for a dynamically changing search depth and a dynamically changing (sub-)goal. Evaluation in path-planning on video-game maps shows that the new algorithm significantly outperforms fixed-depth, fixed-goal LRTA*. The new algorithm can achieve the same quality solutions as LRTA*, but with nine times less computation, or use the same amount of computation, but produce four times better quality solutions. These extensions make real-time heuristic search a practical choice for path-planning in computer video-games.

Introduction

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 this paper we study the problem of real-time planning where 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* 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 *et al.* 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 *et al.* 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 some of the peripheral details. Specifically, in line with Likhachev & Koenig (2005) we consider an agent on a finite search graph with the task of planning 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 found between the start and the goal locations. These two measures are antagonistic insomuch 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 video games as a testbed. In such games, an agent can be tasked to go to any location on the map from its current location. The 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 path-planning algorithms. Bioware Corp., a major game company we collaborate with, sets the limit to 1-3 ms.

LRTA* is the best-known algorithm that addresses this problem. In an agent's current state, LRTA* uses a breadth-first search of a user-specified fixed depth (henceforth *search depth*, also known as lookahead depth or search horizon in the literature). A heuristic function computed with respect to the global goal state is then used to select the next action.

The main contribution of this paper is a new generalized variant of LRTA* that can dynamically alter both its search depth and its target (sub-)goal in an automated fashion. Empirical evaluation on video-game maps shows that the new algorithm outperforms fixed-depth, fixed-goal LRTA*. It can achieve the same quality solutions as LRTA* with nine times less computation per move. Alternatively, with the same amount of computation, it finds four times better solutions. As a result, the generalized LRTA* becomes a practical choice for computer video-games.

The rest of the paper is organized as follows. We first for-

mulate the problem of real-time heuristic search and show how the core LRTA* algorithm can be extended with dynamic lookahead and subgoal selection. We then describe two approaches to dynamic lookahead selection: one based on induction of decision-tree classifiers and one based on precomputing pattern-databases. We then present the idea of selecting subgoals dynamically and evaluate the efficiency of these extensions in the domain of path-planning. We review related research and conclude with a discussion of applicability of the new approach to general planning.

Problem Formulation

We define a heuristic search problem as a directed graph containing a finite set of states and 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*.

In video-game map settings, states are defined as vacant square grid cells. Each cell is connected to adjacent vacant cells. In our empirical testbed, cells have up to eight neighbors: four in the cardinal and four in the diagonal directions. The costs are $1 / \sqrt{2}$ for cardinal / diagonal actions.

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 cumulative cost between a state and the goal and is used by the agent to rank available actions and select the most promising one. In this paper we consider only admissible heuristic functions that 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.

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. In applications, low bounds are preferred as they guarantee the agent's fast reaction to a new goal specification or to changes in the environment. We will be measuring mean *planning time* per action in terms of CPU time as well as a machine-independent measure – the *number of states (or nodes) expanded* during planning. A state is called expanded if its successor states are considered. 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.

In this paper we use a generalized version of the core of most real-time heuristic search algorithms: an algorithm called Learning Real-Time A* (LRTA*) (Korf 1990). It is shown in Figure 1. As long as the goal state s_{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_g individually at each execution step (the original algorithm uses the user-specified parameters d and s_{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'

General LRTA*$(s_{\text{start}}, s_{\text{goal}}, d)$

1 $s \leftarrow s_{\text{start}}$; $s_g \leftarrow s_{\text{goal}}$; $d' \leftarrow d$
2 **while** $s \neq s_{\text{goal}}$ **do**
3   select search depth d' and goal s_g
4   expand successor states up to d' actions away
5   find state s' with the lowest $g(s, s') + h(s', s_g)$
6   update $h(s, s_g)$ to $g(s, s') + h(s', s_g)$
7   move s one step towards s'
8 **end while**

Figure 1: General LRTA* algorithm.

actions away from the current state s. We use the standard path-max technique to deal with possible inconsistencies in the heuristic function when computing $g + h$-values. For each frontier state $\hat{s}$, its value is a 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_g (i.e., the heuristic value $h(\hat{s}, s_g)$). The state that minimizes the sum is identified as s' in line 5. The heuristic value of the current state s is updated in line 6 (the generalized version keeps separate heuristic tables for the different goals). Finally, we take one step towards the most promising frontier state s' in line 7.

Dynamic Selection of Search Depth

We will now present two different approaches to equipping 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 pattern database.

Decision-Tree Classifier Approach

LRTA* can be augmented with a classifier for choosing the search depth at each execution step. The agent feeds the recent search history and observed dynamic properties of the search space into the classifier. The classifier returns a search depth to use for the current state.

An effective classifier needs input features that are not only useful for predicting the ideal search depth, but are also efficiently computable by the agent in real-time. The following features were selected as a compromise between those two considerations: (1) initial heuristic estimate, $h(s, s_{\text{goal}})$, of the distance from the current state to the goal; (2) heuristic estimate of the distance between the current state and the state the agent was in n steps ago (n is a user-set parameter); (3) total number of revisits to states previously encountered in the last n steps; and (4) total number of heuristic updates in the last n steps. Feature 1 provides a rough estimate of the location of the agent relative to the goal. Features 2 and 3 measure the agent's mobility in the search space. Frequent state revisits may indicate a heuristic depression; a deeper search is usually beneficial in such situations (Ishida 1992). Feature 4 is a measure of inaccuracies and inconsistencies in the heuristic around the agent; again, frequent heuristic updates may warrant a deeper search. When calling the classifier for the current state, features (3) and (4) are based on statistic collected from the n previous execution steps of the

agent, i.e., (3) looks at the states the agent was in during the last n steps, and counts the number of reoccurring states.

The classifier predicts the optimal search depth for the current state. To avoid pre-computing true optimal actions, we assume that deeper search usually yields a better action. In the training phase, the agent first conducts a d_{max}-ply search to derive the "optimal" action. Then a series of progressively shallower searches are performed to determine the shallowest search depth, d_s, 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_s to d_{max} to agree on the best action. This is important for improving the overall robustness of the classification, as the classifier must generalize over a large set of states. The depth d_s is set as the class label for the vector of features describing the current state

Once a decision-tree classifier is built, the overhead of using it is negligible. The real-time response is ensured by fixing the maximum length that a decision-tree branch can grow to, as well as the length of the recent history from which we collect input features. This maximum is independent of the problem (map) size. Indeed, the four input features for the classifier are all efficiently computed in small constant time, and the classifier itself is only a handful of shallowly nested conditional statements. Thus, the execution time is dominated by LRTA*'s lookahead search. The process of gathering training data and building the classifier is carried out off-line. Although the classifier appears simplistic, with minimal knowledge about the domain, as shown later it performs surprisingly well.

Pattern Database Approach

We start with a naïve approach as follows. For each (s_{start}, s_{goal}) state pair, the true optimal action $a^*(s_{start}, s_{goal})$ is to take an edge that lies on an optimal path from s_{start} to s_{goal}. Once $a^*(s_{start}, s_{goal})$ is known, we can run a series of progressively deeper searches from state s_{start}. The shallowest search depth that yields $a^*(s_{start}, s_{goal})$ is the optimal search depth $d^*(s_{start}, s_{goal})$.

There are two problems with the naïve approach. First, $d^*(s_{start}, s_{goal})$ is not *a priori* upper-bounded, thereby forfeiting LRTA*'s real-time property. Second, pre-computing $d^*(s_{start}, s_{goal})$ or $a^*(s_{start}, s_{goal})$ for all pairs of (s_{start}, s_{goal}) states on even a 512×512 cell video-game map has prohibitive time and space complexity. We solve the first problem by capping $d^*(s_{start}, s_{goal})$ 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 current and goal states. The resulting data are a *pattern database* (Culberson & Schaeffer 1998).

This approach speeds up pre-computation and reduces memory overhead (both important considerations for commercial video games). To illustrate, in typical grid world video-game maps, a single application of clique abstraction (Sturtevant & Buro 2005) reduces the number of states by a factor of 2 to 3. At the abstraction level of five, each re-

Figure 2: Goal is shown as G, agent as A. Diamonds denote representative states for each tile. **Left:** Optimal actions are shown for each representative of an abstract tile; applying the action of the agent's tile in the agent's current location leads into a wall. **Right:** The dashed line denotes the optimal path.

gion contains about one hundred 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 in the section).

Two alternatives to storing the optimal search depth are to store an optimal action or the optimal heuristic value. The use of abstraction excludes 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 2, left). Likewise, sharing a single heuristic value among all states in a region leaves the agent without a sense of direction as all states in its vicinity would look equally close to goal. Such agents are not guaranteed to reach a goal, let alone minimize travel. In contrast, sharing the search depth among any number of ground-level states is safe as LRTA* is complete for any search depth. Additionally, optimal search depth is more robust with respect to map changes (e.g., a bridge being destroyed in a video-game).

We compute a single pattern database per map off-line (Figure 3). In line 1 the state space is abstracted ℓ times. In this paper we use clique abstraction that merges fully connected states. This respects topology of a map but requires storing the abstraction links explicitly. An alternative is to use regular rectangular tiles of Botea, Müller, & Schaeffer (2004). Lines 2 through 7 iterate through all pairs of abstract states. For each pair (s'_{start}, s'_{goal}), representative ground-level states s_{start} and s_{goal} (e.g., centroids of the regions) are picked and the optimal search depth value d is calculated for them. To do this, Dijkstra's algorithmis run over the ground-level search space (V, E) to compute true minimal distances from any state to s_{goal}. Once the distances are known for all successors of s_{start}, an optimal action $a^*(s_{start}, s_{goal})$ can be computed trivially. Then the optimal search depth $d^*(s_{start}, s_{goal})$ is computed as described above and capped at c (line 5). The resulting value d is stored for the pair of abstract states (s'_{start}, s'_{goal}) in line 6.

During run-time, an LRTA* agent located in state s_{start} and with the goal state s_{goal} sets the search depth as the pattern database value for pair (s'_{start}, s'_{goal}), where s'_{start} and s'_{goal} are images of s_{start} and s_{goal} under an ℓ-level abstraction. The run-time complexity is minimal as $s'_{start}, s'_{goal}, d(s'_{start}, s'_{goal})$

BuildPatternDatabase(V, E, c, ℓ)

1.   apply an abstraction procedure ℓ times to the original space (V, E) to compute abstract space $S_\ell = (V_\ell, E_\ell)$
2.   **for** each pair of states $(s'_{\text{start}}, s'_{\text{goal}}) \in V_\ell$ **do**
3.   select a representative of s'_{start}: $s_{\text{start}} \in V$
4.   select a representative of s'_{goal}: $s_{\text{goal}} \in V$
5.   compute optimal search depth value d for state s_{start} with respect to goal s_{goal}; cap d at c
6.   store d for pair $(s'_{\text{start}}, s'_{\text{goal}})$
7.   **end for**

Figure 3: Pattern-database construction.

can be computed with a minimal constant-time overhead on each action. We will now analyze the time and space complexity of building a pattern database.

Dijkstra's algorithm is run V_ℓ times on the graph (V, E) – a time complexity of $O(V_\ell(V \log V + E))$ on sparse graphs. The optimal search depth is computed V_ℓ^2 times. Each time, there are at most c LRTA* invocations with the total complexity of $O(b^c)$ where b is the maximum degree of any vertex in (V, E). 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, simplified for sparse graphs (i.e., $E = O(V)$).

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	$O(V^2)$	$O(V_\ell^2)$	$(V/V_\ell)^2$

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 deeper searches are assumed to lead to the better action. The assumption does not always hold – a phenomenon known as lookahead pathology (Bulitko *et al.* 2003). The second advantage is that we do not need features of the current state and recent history. The search depth is looked up on the basis of the current state's identifier, such as its coordinates.

The decision-tree classifier approach has two advantages over the pattern-database approach. First, the classifier training does not need happen on the same search space the agent operates on. As long as the training maps used to collect the features and build the decision tree are representative of runtime 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.

Dynamic Selection of Goal

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}. Going back to Figure 2, consider the grid world example to the right. The map is partitioned into four abstract states whose representative states are shown as diamonds (1-4). A straight line distance heuristic will ignore the wall between the agent (A) and the goal (G) and will lead the agent towards the wall. A very deep search is needed in LRTA* to produce the optimal action. Thus, for any realistic cap value, the agent will be left with the suboptimal $\downarrow$ action and will spend a long time in its corner raising the heuristic values. Getting stuck in corners and other heuristic depressions is the primary weakness of real-time heuristic search agents which, in this example, is not addressed by dynamic search depth selection (due to the cap).

A solution is to switch to an *intermediate* goal when the heuristic with respect to the global goal is grossly inaccurate and, as a result, the optimal search depth is too high. In the example, a much shallower search depth is needed for an optimal action towards the next abstract state (marked with diamond 2). The approach is implemented by replacing lines 5–6 in Figure 3 with those in Figure 4.

5.   compute optimal search depth value d for s_{start} to s_{goal}
5a.   **if** $d > c$ **then**
5b.   re-compute d for s_g, cap at c
5c.   store (d, s_g) for pair $(s'_{\text{start}}, s'_{\text{goal}})$
6.   **else**
6a.   store (d, s_{goal}) for pair $(s'_{\text{start}}, s'_{\text{goal}})$

Figure 4: Switching from global goal s_{goal} to intermediate goal s_g. Replaces lines 5–6 of Figure 3.

As long as the cap is not reached, the new algorithm works as described earlier. When the cap is reached in line 5a, the global goal s_{goal} is replaced with an intermediate goal s_g in line 5b. The intermediate goal s_g is the ground-level representative of the next abstract state that an optimal path from s_{start} to s_{goal} will enter. In the right map in Figure 2, the optimal path (a thick dash line) will enter the upper right quadrant. Its representative, the diamond 2, would thus be selected as s_g. It will then be stored in line 5c, together with the corresponding search depth.

The capping is still necessary (line 5b) to guarantee a constant bound on planning time per action. However, switching to a closer goal (i.e., from s_{goal} to s_g) works because it usually improves heuristic accuracy (heuristic functions used in practice are usually more accurate closer to goal). Consequently, a shallower search depth is needed to get the optimal action out of LRTA* and is less likely to be capped. This intuition is supported empirically: on 512×512 maps, pattern database LRTA* ($\ell = 5$) reaches the cap of 30 about 10% of the time with global goal, and never with local goals.

At run-time, the agent executes LRTA* with the stored search depth *and* computes the heuristic h with respect to the stored goal (be it s_{goal} or s_g) in line 3 of Figure 1. In
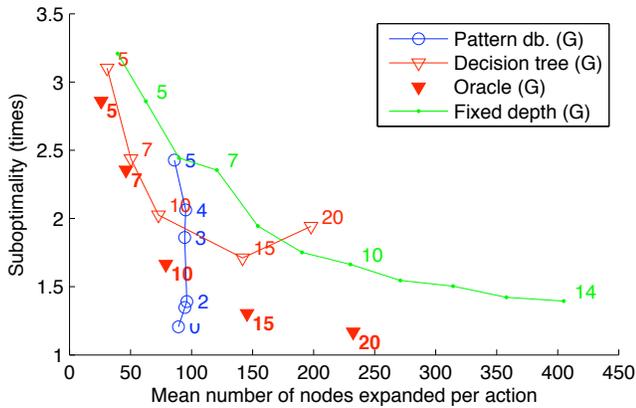
Figure 5: Margins for improvement over fixed-depth LRTA*.

Figure 6: Effects of dynamic goal selection.

other words, in addition to selecting the search depth per state, the goal is also selected dynamically, per action.

Finally, one can imagine *always* using intermediate goals (until the agent enters the abstract state containing s_{goal}). This gives us three approaches to selecting goal states in LRTA*: always global goal (G); intermediate goal if the global goal requires excessive search depth (G+I); always intermediate goal unless the agent is in the goal region (I).

Empirical Evaluation

This section presents results of an empirical evaluation of LRTA* agents with dynamic control of search depth and goal selection. All algorithms except Koenig's LRTA* use breadth-first search for planning and avoid re-expanding states 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 and actual timings on a 3 GHz Pentium IV computer.

We use grid world maps from a popular real-time strategy game as our testbed. Such maps provide a realistic and challenging environment for real-time search (Sigmundarson & Björnsson 2006). The agents were first tested on three different maps (sized 161×161 to 193×193 cells), performing 100 searches on each map. The heuristic function used is octile distance – a natural extension of the Manhattan distance for maps with diagonal actions. The start and goal locations were chosen randomly, although constrained such that optimal solution paths cost between 90 and 100. Each data point in the plots below is an average of 300 problems (3 maps $\times 100$ runs each). All algorithms were implemented in the HOG framework (Sturtevant 2005), and use a shared code-base for algorithmic independent features. This has the benefit of minimizing performance differences caused by different implementation details (e.g. all algorithms are using the same tie-breaking mechanism for node selection).

For building the classifiers, we used the J48 decision tree algorithm in the WEKA library (Witten & Frank 2005). The training features were collected from a history trace of $n = 20$ steps. The game maps were used for training and 10-fold cross-validation was used to avoid over-fitting the data. The *pruning factor* and *minimum number of data items per leaf* parameters of the decision tree algorithm were set
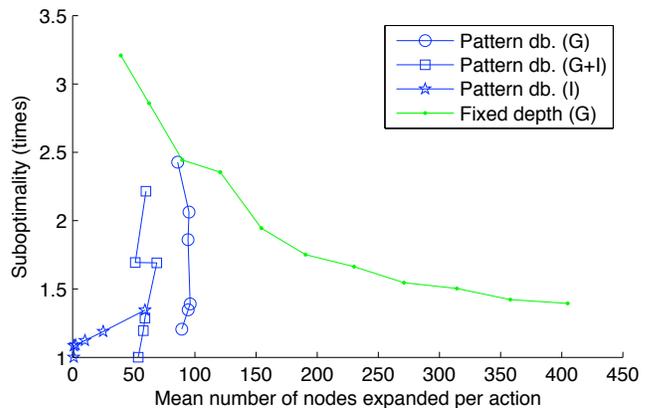
to 0.05 and 200, respectively. For the reference, at depth $d_{\text{max}} = 20$ there were approximately 140 thousand training samples recorded on the problems. Collecting the data and building the classifier took about an hour. Of these two tasks, collecting the training data is by far the more time consuming — building the decision trees using the WEKA library takes only a matter of minutes. On the other hand, the decision-tree building process requires more memory. For example, during the large map experiments described in a later subsection, we collected several hundred of thousand training samples (taking several hours to collect) and this required and order of 1GB of memory for WEKA to process. As this calculation is done offline, we are not overly concerned with efficiency. As described earlier, once the decision-tree classifiers have been build they have negligible time and memory footprint.

Margins for Improvement. The result of running the different LRTA* agents is shown in Figure 5. Search depth for all versions of LRTA* was capped at 20. The x-axis represents the amount of work done in terms of the mean number of states (nodes) expanded per action, whereas the y-axis shows the quality of the solutions found (as a multiple of the length of the optimal path). The standard LRTA* provides a baseline trade-off curve. We say that a generalized version of LRTA* outperforms the baseline for some values of control parameters if it is better along both axes (i.e., appears below a segment of the baseline curve on the graph).

The topmost line in the figure shows the performance of LRTA* using fixed search depths in the range $[4, 14]$. The classifier approach, shown as the triangle-labeled line, outperforms the baseline for all but the largest depth (20). As seen in the figure, the standard LRTA* must expand as many as twice the number of states to achieve a comparable solution quality. The solid triangles show the scope for improvement if we had a perfect classifier (oracle) returning the "optimal" search depth for each state (as defined for the classifier approach). For the shallower searches, the classifier performs close to the corresponding oracle, but as the search depth increases, the gap between them widens. This is consistent with classification accuracy: the ratio of correctly classified instances dropped as the depth increased (from 72% at depth 5 to 44% at depth 20). More descrip-
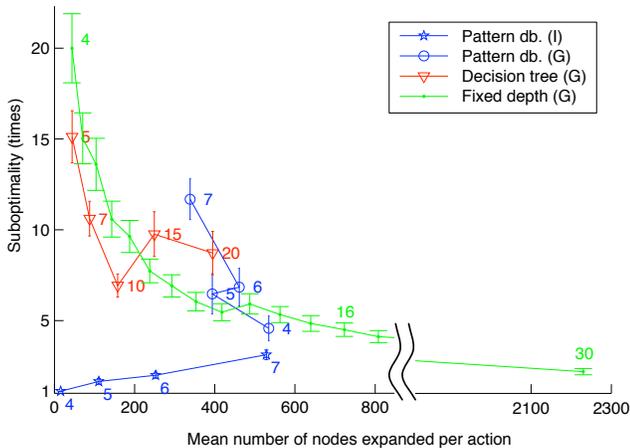
Figure 7: Benefits of dynamic search control over fixed-depth LRTA*. The pattern-database curve is labeled with the level of abstraction used. The classifier approach curve is labeled with $d_{\max}$.

tive features and longer history traces are likely needed.

The circle markers show the performance of agents using global-goal pattern databases with abstraction levels of 1 through 5. All abstraction levels outperform the baseline, and for level 3 and less, also the classifier approach. For example, a 2-level abstraction expands on average four times fewer states than the fixed strategy that achieves comparable solution quality. The 0 point is produced by pre-computing the optimal search depth for all ground-level state pairs. The resulting paths are still somewhat suboptimal because of the imposed cap of 20.

Effects of Intermediate Goals. Figure 6 shows the benefits of using intermediate goals. The two rightmost curves plot the performance of fixed strategies and the global goal pattern database (same as in the previous figure), whereas the two leftmost curves show the effect of adding intermediate goals at various levels of abstraction (the lowest point on each curve represents level 0, no abstraction, and the top point is for level 5). The performance improvement of using intermediate goals is substantial, even at higher abstraction levels. For instance, a pattern database of abstraction level 5 with intermediate goals performs similarly to a larger level-2 database with mixed global/intermediate goals. Both are expanding an order of magnitude fewer states than a fixed depth LRTA* of a similar sub-optimality.

Scaling Up. Recent releases of real-time strategy games use even larger maps than we used in the above experiments. To find out how well our approaches scale to such large maps, we scaled the maps to 512×512 cells (the start and goal locations moved further apart accordingly). The results are shown in Figure 7 (the error bars show the standard error). We used the same settings as in the previous experiments except to adjust to the larger map size, we increased the cap from 20 to 30 and levels of abstraction to 4 through 7.

The 7 to 10 fold increase in the map size led to less optimal solutions for the same search depths. The classifier of depth 10 or less still outperforms the fixed-depth LRTA*. Likewise, global goal pattern-database approach

outperforms the fixed strategies for abstraction level of 4. The intermediate goals version is less affected by map scaling due to the locality of its goals. It still shows a strong superiority over all other approaches; even at as high abstraction levels as 6, it expands an order of magnitude fewer states than a fixed strategy counterpart of a similar sub-optimality.

The number of states expanded per action is a machine-independent measure. It is also fair for comparing our different approaches because the dynamic selection schemes impose negligible time overhead at run-time. In terms of the actual time scale, a search expanding 150 - 200 states per move, which is a typical average for our approaches, takes under a millisecond.

Pattern Database Construction. The pattern database approach with intermediate goals exhibits the best performance tradeoff in terms of sub-optimality versus computation per action. This is achieved through building a pattern database containing a search depth d and intermediate goal s_g for all pairs of abstract states. Table 2 shows the trade-offs involved. For each level of abstraction $1 - 7$ and the case of no abstraction (0), we list the size of the pattern database (in the number of (d, s_g) entries stored), construction time, and the performance of the resulting LRTA* (sub-optimality and states expanded per action). Abstraction levels 0 to 2 are prohibitively expensive and their results are estimated.

Table 2: Pattern databases for a 512×512 map.

Abs. level	Size	Time	Subopt.	Planning
0	1.1×10^{10}	est. 1 year	-	-
1	7.4×10^8	est. 1 month	-	-
2	5.9×10^7	est. 4 days	-	-
3	6.1×10^6	19 hours	1.09	5
4	8.6×10^5	5 hours	1.11	16
5	1.5×10^5	2 hours	1.65	111
6	3.1×10^4	50 min	1.99	253
7	6.4×10^3	20 min	3.12	529

Finally, instead of using LRTA* as our base algorithm, we could have used RTA*. Experiments showed that there is no significant performance difference between the two for a search depth beyond one. Indeed for deeper searches the 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 keep open the possibility of the agents learning over repeated trials.

Comparison to State-of-the-Art Algorithms. In the pervious subsections we investigated the performance gains dynamic control offers in real-time search. This was best tested in LRTA* – the most fundamental real-time search algorithm, and a core of most modern algorithms.

Here we compare our dynamic-control LRTA* against the current state-of-the-art. Figure 8 also includes plots for three recent real-time algorithms, Koenig's LRTA* (Koenig 2004), PR LRTS (Bulitko, Sturtevant, & Kazakevich 2005), and Prioritized LRTA* (Rayner *et al.* 2007), for various parameter settings. In deciding on the most appropriate parameter setting for each algorithm, we imposed a constraint that
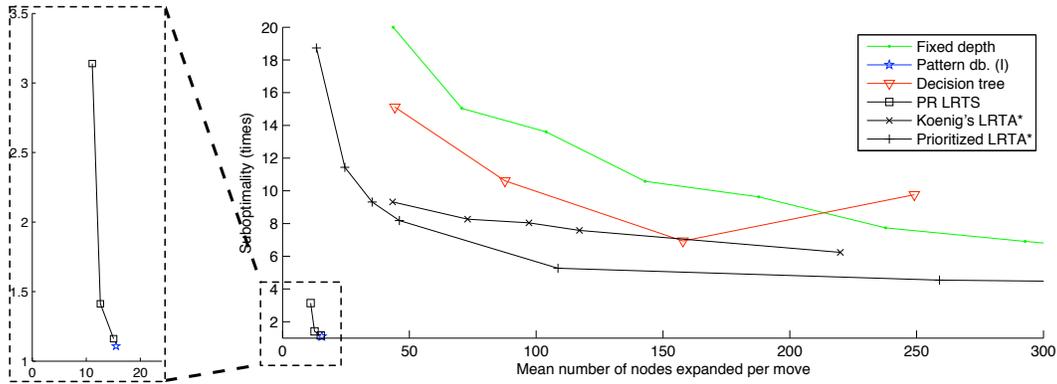
Figure 8: Dynamic-control LRTA* versus state-of-the-art algorithms. The zoomed-in part highlights the abstraction-based methods, PR LRTS and intermediate-goal PDB; both methods use abstraction level 4; PR LRTS uses γ=1.0 and depths 1, 3, and 5 (from top to bottom).

the *worst-case* number of nodes expanded per move does not exceed 1000 nodes – approximately the number of nodes that an optimized implementation of these algorithms can expand in the amount of time available for planning each action in video games. All parameter settings that violate this constraint were excluded from the figure.

Of the surviving algorithms, two clearly dominate the others: our new pattern-database approach with intermediate goals and PR LRTS (both use state abstraction). At their best parameter settings, PR LRTS with the abstraction level of 4, depth 5, and γ=1.0, and the pattern-database approach with the abstraction level of 4 yield comparable performance and neither algorithm dominates the other. PR LRTS searches slightly fewer nodes but the pattern-database approach returns slightly better solutions, as shown in the zoomed-in part in the figure. Thus, the pattern-database approach is a new competitive addition to the family of state-of-the-art real-time search algorithms. Furthermore, as PR LRTS runs LRTA* in an abstract state space, it can be equipped with our dynamic control scheme and is likely to achieve even better performance; this is the subject of ongoing work.

Previous Research

Most algorithms in single-agent real-time heuristic search use fixed search depth, with a few notable exceptions. Russell & Wefald (1991) proposed to estimate the value of search off-line/on-line. They estimated 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 small benefits in combinatorial puzzles.

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. A generalized definition of heuristic depressions was used by Bulitko (2004) who argued for extending 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. In our ap-

proach, we execute only a single action toward the frontier and do not use backtracking — two feature that have been shown to improve robustness of real-time search (Sigmundarson & Björnsson 2006; Luštrek & Bulitko 2006). Additionally, in the pattern-database approach we switch to intermediate goals when the agent discovers itself in a deep heuristic depression. The idea of pre-computing a pattern database of heuristic values for real-time path-planning was recently suggested by Luštrek & Bulitko (2006). This paper extends their work in several directions: (i) we introduce the idea of intermediate goals, (ii) we propose an alternative approach that does not require map-specific pre-computation, and (iii) we demonstrate superiority over fixed-ply search on large-scale computer game maps.

There is a long tradition for search control in two-player search. The problem of semi-dynamically allotting time to each action decision in two-player search is somewhat analogous to the depth selection investigated here.

Applicability to General Planning

So far we have evaluated our approach empirically only on path-planning. However, it may also offer benefits to a wider range of planning problems. The core heuristic search algorithm extended in this paper (LRTA*) was previously applied to general planning (Bonet, Loerincs, & Geffner 1997). The extensions we introduced may have a beneficial effect in a similar way to how the B-LRTA* extensions improved the performance of ASP planner. Subgoal selection has been long studied in planning and is a central part of our intermediate-goal pattern-database 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 is pre-computation of optimal search depth in the pattern databases 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^*. Deriving heuristic guidance from solving relaxed problems is common to both planning and the heuristic search community.

Conclusions and Future Work

Since Korf's seminal work on LRTA*, most real-time search algorithms use a fixed search depth that is manually tuned for an application. Furthermore, the heuristic function is usually computed with respect to the agent's global goal. In this paper we demonstrated that selecting both the search depth and the agent's goal dynamically for each action has substantial benefits, and results in performance improvements that are on a par with of what is offered by advanced state-of-the-art real-time algorithms. To illustrate, on maps with quarter of a million states, pattern-database LRTA* is an order of magnitude faster than the standard version for the same solution quality. Alternatively, for the same mean amount of computation per action, dynamic control finds four times shorter solutions. This is accomplished by pre-computing a pattern database of about 62 thousand values in 50 minutes. Our current research focus is on incorporating the techniques introduced here into other state-of-the-art methods (e.g. PR LRTS), and to run more thorough set of experiments to isolate better the individual effects of dynamic depth vs. goal selection. Both play an important role as our experiments show. We also plan to experiment with these algorithms in environments with dynamic obstacles.

This project opens several interesting avenues for future research. In particular, we defined a space of algorithms of the following three dimensions: search depth (fixed versus dynamic), goal (global versus global and intermediate versus intermediate), selection mechanism (pattern database versus machine-learned classifier). Only four out of the resulting twelve combinations were explored in this paper. It would be of interest to explore the others.

Acknowledgments

References

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

Bonet, B., and 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 *ICAPS'06*, 142–151.

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

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

Bulitko, V.; Li, L.; Greiner, R.; and Levner, I. 2003. Lookahead pathologies for single agent search. In *IJCAI*, 1531–1533.

Bulitko, V.; Sturtevant, N.; and Kazakevich, M. 2005. Speeding up learning in real-time search via automatic state abstraction. In *AAAI*, 1349 – 1354.

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

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

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)*, 216–227.

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

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

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

Koenig, S. 2004. A comparison of fast search methods for real-time situated agents. In *Proceedings of Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 864 – 871.

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. *AIJ* 42(2-3):189–211.

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

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and 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.; and Thrun, S. 2004. ARA*: Anytime A* with provable bounds on sub-optimality. In Thrun, S.; Saul, L.; and Schölkopf, B., eds., *Advances in Neural Information Processing Systems 16*. Cambridge, MA: MIT Press.

Luštrek, M., and Bulitko, V. 2006. Lookahead pathology in real-time path-finding. In *AAAI WS on Learning For Search*, 108–114.

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

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

Sigmundarson, S., and Björnsson, Y. 2006. Value back-propagation versus backtracking in real-time heuristic search. In *AAAI Workshop on Learning For Search*, 136–141.

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

Sturtevant, N., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *AAAI*, 1392–1397.

Sturtevant, N. 2005. Hog - hierarchical open graph. http://www.cs.ualberta.ca/˜nathanst/hog/.

Witten, I., and Frank, E. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2nd edition.