Speeding Up Learning in Real-time Search via Automatic State Abstraction*

Vadim Bulitko and **Nathan Sturtevant** and **Maryia Kazakevich**
Department of Computing Science, University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
{bulitko|nathanst|maryia}@cs.ualberta.ca

Abstract

Situated agents which use learning real-time search are well poised to address challenges of real-time path-finding in robotic and computer game applications. They interleave a local lookahead search with movement execution, explore an initially unknown map, and converge to better paths over repeated experiences. In this paper, we first investigate how three known extensions of the most popular learning real-time search algorithm (LRTA*) influence its performance in a path-finding domain. Then, we combine automatic state abstraction with learning real-time search. Our scheme of dynamically building a state abstraction allows us to generalize updates to the heuristic function, thereby speeding up learning. The novel algorithm converges up to 80 times faster than LRTA* with only one fifth of the response time of A*.

Introduction

In this paper, we consider a simultaneous planning and learning problem. More specifically, we require an agent to navigate on an initially unknown map under real-time constraints. As an example, consider a robot driving to work every morning. Imagine the robot to be a newcomer to the town. The first route the robot finds may not be optimal because the traffic jams, road conditions, and other factors are initially unknown. With a passage of time, the robot continues to learn and eventually converges to a nearly optimal commute. Note that planning and learning happen while the robot is driving and therefore are subject to time constraints.

Present-day mobile robots are often plagued by localization problems and power limitations, but simulation counterparts already allow researchers to focus on the planning and learning problem. For instance, the RoboCup Rescue simulation (Kitano *et al.* 1999) requires real-time planning and learning with multiple agents mapping out unknown terrain.

Similarly, many current-generation real-time strategy games employ *a priori* known maps. Full knowledge of the maps enables complete search methods such as A*. Prior availability of the maps allows path-finding engines to pre-compute data (e.g., visibility maps) to speed up on-line navigation. Neither technique will be applicable in forthcoming generations of commercial and academic games (Buro 2002) which will require the agent to cope with the initially unknown maps via exploration and learning during the game.

To compound the problem, the dynamic A* (D*) (Stenz 1995) and D* Lite (Koenig & Likhachev 2002), frequently used in robotics, work well when the robot's movements are slow with respect to its planning speed. In real-time strategy games, however, the AI engine can be responsible for hundreds to thousands of agents traversing the map simultaneously and the planning cost becomes a major factor. We thus discuss the following three questions.

First, how planning time per move and, particularly the first-move delay, can be minimized so that each agent moves smoothly and responds to user requests nearly instantly. Second, given the local nature of the agent's reasoning and the initially unknown terrain, how the agent can learn a better global path. Third, how learning can be accelerated so that only a few repeated path-finding experiences are needed before converging to a near-optimal path.

In the rest of the paper, we first make the problem settings concrete and derive specific performance metrics based on the questions above. Then we discuss the challenges that incremental heuristic search faces when applied to real-time path-finding. As an alternative, we will review a family of learning real-time search algorithms which are well poised for use by situated agents. Starting with the most popular real-time search algorithm, LRTA*, we make our initial contribution by evaluating three known complementary extensions in the context of real-time path-finding. The resulting algorithm, LRTS, exhibits a 46-fold speed-up in the travel until convergence while having one sixth of the first-move delay of an A* agent. Despite the improvements, the learning and search still happen on a large ground-level map. Thus, all states are considered distinct and no generalization is used in learning. We then make the primary contribution by introducing an effective mechanism for building and repairing a hierarchical abstraction of the map. This allows us to constrain the search space, reduce the amount of learning required for convergence, and generalize learning in each individual state onto neighboring states. The novel algorithm, PR-LRTS, is then empirically evaluated.

Problem Formulation

In this paper, we focus on a particular real-time path-finding task. Specifically, we will assume that the agent is tasked to

*We appreciate funding from NSERC, iCORE, and AICML.

travel from the start state (x_s, y_s) to the goal state (x_g, y_g). The coordinates are on a two-dimensional rectangular grid. In each state, up to eight moves are available leading to the eight immediate neighbors. Each straight move (i.e., north, south, west, east) has the *travel cost* of 1 while each diagonal move has the travel cost of $\sqrt{2}$. Each state on the map can be passable or occupied by a wall. In the latter case, the agent is unable to move into it. Initially, the map in its entirety is unknown to the agent. In each state (x, y) the agent can see the status (occupied/free) of the neighborhood of the *visibility radius* v: $\{(x', y') \mid |x' - x| \leq v \ \& \ |y' - y| \leq v\}$. The agent can choose to remember the observed parts of the map and use that information in subsequent planning.

A *trial* is defined as a finite sequence of moves the agent takes to travel from the start to the goal state. Once the goal state is reached, the agent is reset to the start state and the next trial begins. A *convergence run* is defined as the first sequence of trials such that the agent does not learn or explore anything new on the subsequent trials.

Each problem instance is fully specified by the map and start and goal coordinates. We then run the agent until convergence and measure the cumulative travel cost of all moves (*convergence travel*), the average delay before the first move (*first-move lag*), and the length of the path found on the final trial (*final solution length*). The last measure is used to compute the *amount of suboptimality* defined as percentage of the length excess.

Incremental Search

Classical A* search is inapplicable due to an initially unknown map. Specifically, it is impossible for the agent to plan its path through state (x, y) unless it is either positioned within the visibility radius of the state or has visited this state on a prior trial.

A simple solution to this problem is to generate the initial path under the assumption that the unknown areas of the map contain no occupied states (the free space assumption (Koenig, Tovey, & Y. 2003)). With the *octile distance*[1] as the heuristic, the initial path is close to the straight line since the map is assumed empty. The agent follows the existing path until it runs into an occupied state. During the travel, it updates the explored portion of the map in its memory. Once the current path is blocked, A* is invoked again to generate a new complete path from the current position to the goal. The process repeats until the agent arrives at the goal. It is then reset to the start state and a new trial begins. The convergence run ends when no new states are seen.

To increase efficiency, several methods of re-using information over subsequent planning episodes have been suggested. The two popular versions are D* (Stenz 1995) and D* Lite (Koenig & Likhachev 2002). Unfortunately, these enhancements do not reduce the first-move lag time. Specifically, after the agent is given the destination coordinates, it has to conduct an A* search from its position to the destination before it can move. Even on small maps, this delay can be substantial. Consider, for instance, a map from BioWare's game "Baldur's Gate" shown in Figure 1. Before an A*-controlled agent can make its first move, a complete path from start to goal state has to be generated. This is in contrast to LRTA* (Korf 1990), which only performs a small local search to select the first move. As a result, several orders of magnitude more agents can calculate and make their first move in the time it takes one A* agent.

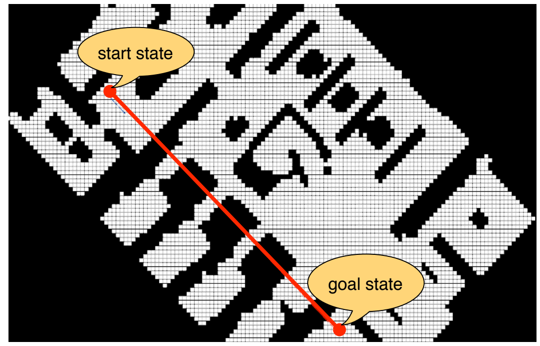

Figure 1: A sample map from a BioWare's game.

A thorough comparison between D* Lite and an extended version of LRTA* is found in (Koenig 2004). It investigates the conditions under which real-time search outperform incremental search. Since our paper focuses on real-time search and uses incremental search only as a reference point and because D*/D* Lite does not reduce the first-move lag on the final trial, we use the simpler incremental A* in our experiments.

Real-time Search

Real-time search was pioneered by (Korf 1990) with the presentation of RTA* and LRTA* algorithms. Unlike A*, which can freely traverse its open list, each RTA*/LRTA* search assumes the agent to be in a single current state that can be changed only by taking moves and, thereby, incurring travel cost. From its state, the agent conducts a full-width fixed-depth local forward search (called lookahead) and, similarly to minimax game-playing agents, uses its heuristic h to evaluate the frontier states. It then takes the first move towards the most promising frontier state (i.e., the state with the lowest $g + h$ value where g is the cost of traveling from the current state to the frontier state) and repeats the cycle. The initial heuristic is set to the octile distance. On every move, the heuristic value of the current state is increased to the $g + h$ value of the most promising state.[2] As discussed in (Barto, Bradtke, & Singh 1995), this operation is analogous to the "backup" step used in value iteration reinforcement learning agents with the learning rate $\alpha = 1$ and no discounting. LRTA* will refine an initial admissible heuristic to the perfect heuristic along a shortest path. This constitutes a convergence run. The updates to the heuristic also guarantee that LRTA* will not get trapped in infinite cycles. We now make the first contribution of this paper by

[1]Octile distance is a natural adaptation of Euclidian distance to the case of the eight discrete moves and can be computed in a closed form.

[2]As (Shimbo & Ishida 2003), we do not decrement h of any state. Convergence to optimal paths is still possible as the initial heuristic is admissible but the convergence is accelerated.

Table 1: **Top:** Effects of the lookahead depth d on deliberation time per unit of distance and average travel per trial in LRTA*. **Middle:** Effects of the optimality weight γ on suboptimality of the final solution and total travel in LRTA* ($d = 1$). **Bottom:** Effects of learning quota T on amount of first trial and total travel.

d	Deliberation per move (ms)	Travel per trial
1	0.0087	661.5
3	0.0215	241.8
5	0.0360	193.3
7	0.0514	114.9
9	0.0715	105.8

γ	Suboptimality	Convergence travel
0.1	6.19%	9,300
0.3	4.92%	8,751
0.5	2.41%	9,435
0.7	1.23%	13,862
0.9	0.20%	25,507
1.0	0.00%	31,336

T	First trial travel	Convergence travel
0	434	457
10	413	487
50	398	592
1,000	390	810
5,000	235	935

evaluating the effects of three known complementary extensions in the context of real-time path-finding.

First, increasing lookahead depth increases the amount of deliberation per move but, on average, causes the agent to take better moves, thereby finding shorter paths. This effect is demonstrated in Table 1 with averages of 50 convergence runs over 10 different maps. Hence, the lookahead depth can be selected dynamically depending on the amount of CPU time available per move and the ratio between the planning and moving speeds (Koenig 2004).

Second, the distance from the current state to the state on the frontier (the g-function) can be weighted by the $\gamma \in (0, 1]$. This allows us to trade-off the quality of the final solution and the convergence travel. This extension of LRTA* is equivalent to scaling the initial heuristic by the constant factor of $1+\varepsilon = 1/\gamma$ (Shimbo & Ishida 2003). Bulitko (2004) proved that γ-weighted LRTA* will converge to a solution no worse than $1/\gamma$ of optimal. In practice, much better paths are found (Table 1). A similar effect is observed in weighted A*: increasing the weight of h (i.e., decreasing the relative weight of g) dramatically reduces the number of states generated, at the cost of longer solutions (Korf 1993).

Third, backtracking within LRTA* was first proposed in (Shue & Zamani 1993). Their SLA* algorithm used the lookahead of one and the same update rule as LRTA*. However, upon updating (i.e., increasing) the heuristic value in a state, the agent moved (i.e., backtracked) to its previous state. Backtracking increases travel on the first trial but reduces the convergence travel (Table 1). Note that backtracking does not need to happen after *every* update to the heuristic function. SLA*T, introduced in (Shue, Li, & Zamani 2001), backtracks only after the cumulative amount of updates to the heuristic function made on a trial exceeds the learning quota (T). We will use an adjusted implementation

```
LRTS(d, γ, T)
1   initialize: h ← h_0, s ← s_start, u ← 0
2   while s ≠ s_goal do
3     expand children i moves away, i = 1 . . . d
4        on level i, find state s_i with the lowest f = γ · g + h
5     update h(s) ← max_{1≤i≤d} f(s_i)
6     increase amount of learning u by |Δh|
7     if u ≤ T then
8       execute d moves to get to s_d
9     else
10      execute d moves to backtrack to previous s, set u = T
11    end if
12  end while
```

Figure 2: LRTS algorithm unifies LRTA*, ε-LRTA*, and SLA*T.

of this idea which enables us to bound the length of the path found on the *first* trial by $(h^*(s_{\text{start}}) + T)/\gamma$ where $h^*(s_{\text{start}})$ is the actual shortest distance between the start and goal.

An algorithm combining all three extensions (lookahead d, optimality weight γ, and backtracking control T) operates as follows. In the current state s, it conducts a lookahead search of depth d (line 3 in Figure 2). At each ply, it finds the most promising state (line 4). Assuming that the initial heuristic h_0 is admissible, we can safely increase $h(s)$ to the maximum among the f-values of promising states for all plies (line 5). If the total learning amount u exceeds the learning quota T, the agent backtracks to the previous state (lines 7, 10). Otherwise, it executes d moves forward towards the most promising frontier state (line 8). In the rest of the paper, we will refer to this combination of three extensions as LRTS (learning real-time search).

LRTS with domain-tuned parameters converges two orders of magnitude faster than LRTA* while finding paths within 3% of optimal. At the same time, LRTS is about five times faster on the first move than incremental A* as shown in Table 2. Despite the improvements, LRTS takes hundreds of moves before convergence is achieved, even on smaller maps with only a few thousand states.

Novel Method: Path-refinement LRTS

The problem with LRTA* and LRTS described in the previous section stems from the fact that the heuristic is learnt in a tabular form. Each entry in the table corresponds to a single state and no generalization is attempted. Consequently, thousands of heuristic values have to be incrementally computed via individual updates – one per move of the agent. Thus, significant traveling costs are incurred before the heuristic function converges. This is not the way humans and animals appear to learn a map. We do not learn at the micro-level of individual states but rather reason over *areas*

Table 2: Incremental A*, LRTA*, LRTS averaged over 50 runs on 10 maps. The average solution length is 59.5. LRTA* is with the lookahead of 1. LRTS is with $d = 10, \gamma = 0.5, T = 0$. All timings are taken on a dual G5, 2.0GHz with gcc 3.3.

Algorithm	1st move time	Conv. travel	Suboptimality
A*	5.01 ms	186	0.0%
LRTA*	0.02 ms	25,868	0.0%
LRTS	**0.93 ms**	**555**	**2.07%**

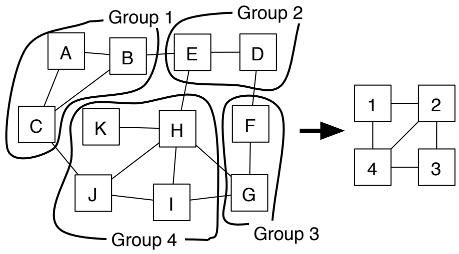

Figure 3: The process of abstracting a graph.

of the map as if they were single entities. Thus, the primary contribution of this paper is extension of learning real-time heuristic search with a state abstraction mechanism.

Building a State Abstraction

State abstraction has been studied extensively in reinforcement learning (Barto & Mahadevan 2003). While our approach is fully automatic, many algorithms, such as MAXQ (Dietterich 1998), rely on manually engineered hierarchical representation of the space.

Automatic state abstraction has precedents in heuristic search and path-finding. For instance, Hierarchical A* (Holte *et al.* 1995) and AltO (Holte *et al.* 1996) used abstraction to speed up classical search algorithms. Our approach to automatically building abstractions from the underlying state representation is similar to Hierarchical A*.

We demonstrate the abstraction procedure on a hand-traceable micro-example in Figure 3. Shown on the left is the original graph of 11 states. In general, we can use a variety of techniques to abstract the map, and we can also process the states in any order. Some methods and orderings may, however, work better in specific domains. In this paper, we look for cliques in the graph.

For this example, we begin with the state labeled A, adding it and its neighbors, B and C, to abstract group 1, because they are fully connected. Their group becomes a single state in the abstract graph. Next we consider state D, adding its neighbor, E, to group 2. We do not add H because it is not connected to D. We continue to state F, adding its neighbor, G, to group 3. States H, I, and J are fully connected, so they become group 4. Because state K can only be reached via state H, we add it to group 4 with H. If all neighbors of a state have already been abstracted, that state will become a single state in the abstract graph. As states are abstracted, we add edges between existing groups. Since there is an edge between B and E, and they are in different groups, we add an edge between groups 1 and 2 in the abstract graph. We proceed similarly for the remaining inter-group edges. The resulting abstracted graph of 4 states is shown in the right portion of the figure.

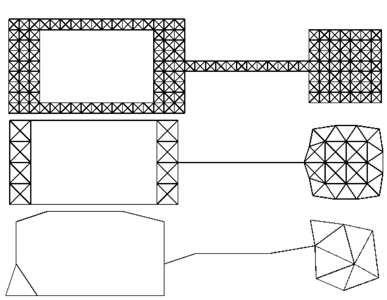

Figure 4: Abstraction levels 0, 1, and 2 of a toy map. The number of states is 206, 57, and 23 correspondingly.

We repeat the process iteratively, building an abstraction hierarchy until there are no edges left in the graph. If the original graph is connected, we will end up with a single state at the *highest* abstraction level, otherwise we will have multiple disconnected states. Assuming a sparse graph of V vertices, the size of all abstractions is at most $O(V)$, because we are reducing the size of each abstraction level by at least a factor of two. The cost of building the abstractions is $O(V)$. Figure 4 shows a micro example.

Because the graph is sparse, we represent it with a list of states and edges as opposed to an adjacency matrix. When abstracting an entire map, we first build its connectivity graph and then abstract this graph in two passes. Our abstractions are most uniform if we remove 4-cliques in a first pass, and then abstract the remaining states in a second pass.

Repairing Abstraction During Exploration

A new map is initially unknown to the agent. Under the free space assumption, the unknown areas are assumed empty and connected. As the map is explored, obstacles are found and the initial abstraction hierarchy needs to be repaired to reflect these changes. This is done with four operations: remove-state, remove-edge, add-state, and add-edge. We describe the first two in detail here.

In the abstraction, each edge either abstracts into another edge in the parent graph, or becomes internal to a state in the parent graph. Thus, each abstract edge must maintain a count of how many edges it is abstracting from the lower level. When remove-edge removes an edge, it decrements the count of edges abstracted by the parent edge, and recursively removes the parent if the count falls to zero. If an edge is abstracted into a state in the parent graph, we add that state to a *repair queue* to be handled later. The remove-state operation is similar. It decrements the number of states abstracted by the parent, removing the parent recursively if needed, and then adds the parent state to a repair queue. This operation also removes any edges incident to the state.

When updating larger areas of the map in one pass, using a repair queue allows us to share the cost of the additional steps required to perform further repairs in the graph. Namely, there is no need to completely repair the abstraction if we know we are going to make other changes. The repair queue is sorted by abstraction level in the graph to ensure that repairs do not conflict.

In a graph with n states, the remove-state and remove-edge operations can, in the worst case, take $O(\log n)$ time. However, their time is directly linked to how many states are affected by the operation. If there is one edge that cuts the entire graph, then removing it will take $O(\log n)$ time. However, in practice, most removal operations have a local influence and take time $O(1)$. Handling states in the repair queue is an $O(\log n)$ time operation in the worst case, but again, we only pay this cost when we are making changes that affect the connectivity of the entire map. In practice,

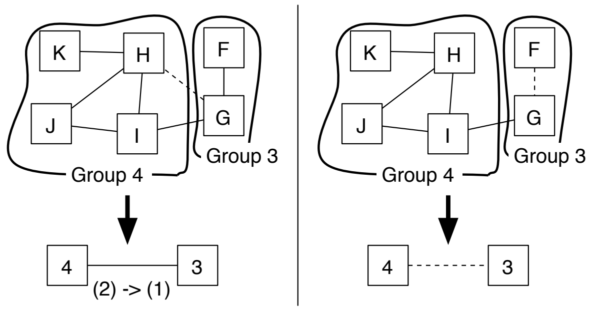

Figure 5: Repairing abstractions.

there will be many states for which we only need to verify their internal connectivity.

Figure 5 illustrates the repair process. Shown on the left is a subgraph of the 11-state graph from Figure 3. When in the process of exploration it is found that state H is not reachable from G, the edge (H,G) will be removed (hence shown with a dashed line). Thus, the abstraction hierarchy needs to be repaired. The corresponding abstracted edge (4,3) represents two edges: (G,H) and (G,I). When (G,H) is removed, the edge count of (4,3) is decremented from 2 to 1.

Suppose it is subsequently discovered that edge (F,G) is also blocked. This edge is internal to the states abstracted by group 3 and so we add group 3 to the repair queue. When we handle the repair queue, we see that states abstracted by group 3 are no longer connected. Because state G has only a single neighbor, we can merge it into group 4, and leave F as the only state in group 3. When we merge state G into group 4, we also delete the edge between groups 3 and 4 in the abstract graph (right part of Figure 5).

Abstraction in Learning Real-time Search

Given the efficient on-line mechanism for building state abstraction, we propose, implement, and evaluate a new algorithm called PR-LRTS (Path-Refining Learning Real-Time Search). A PR-LRTS agent operates at several levels of abstraction. Each level from 0 (the ground level) to $N \geq 0$ is "populated" with A* or LRTS. At higher abstract levels, the heuristic distance between any two states is Euclidian distance between them, where the location of a state is the average location of the states it abstracts. This heuristic is not admissible with respect to the actual map. Octile distance is used as the heuristic at level 0.

At the beginning of each trial, no path has been constructed at any level. Thus, the algorithm at level N is invoked. It works at the level N and produces the path p_N. In the case of A*, p_N is a complete path from the N-level parent of the current state to the N-level parent of the goal state. In the case of LRTS, p_N is the first d steps towards the abstracted goal state at level N. The process now repeats at level $N-1$ resulting in path p_{N-1}. But, when we repeat the process, we restrict any planning process at level $N-1$ to a *corridor* induced by the abstract path at level N. Formally, the corridor c_{N-1} is the set of all states which are abstracted by states in p_N. To give more leeway for movement and learning, the corridor can also be expanded to include any states abstracted by the k–step neighbors of p_N. In this paper, we choose $k = 1$. While executing p_0, new areas of the map may be seen. The state abstraction hierarchy will be repaired as previously described. This path-refining approach, summarized in Figure 7, benefits path-finding in three ways.

PR LRTS

assign A*/LRTS to abstraction levels $0, \ldots, N$
initialize the heuristic for all LRTS-levels
reset the current state: $s \leftarrow s_{\text{start}}$
reset abstraction level $\ell = 0$
while $s \neq s_{\text{goal}}$ **do**
  if algorithm at level ℓ reached the end of corridor c_ℓ **then**
    if we are at the top level $\ell = N$ **then**
      run algorithm at level N
      generate path p_N and corridor c_{N-1}
      go down abstraction level: $\ell = \ell - 1$
    else
      go up abstraction level: $\ell = \ell + 1$
    end if
  else
    run algorithm at level ℓ within corridor c_ℓ
    generate path p_ℓ and corridor $c_{\ell-1}$
    if $\ell = 0$ **then** execute path p_0
    else continue refinement: $\ell = \ell - 1$
  end if
end while

Figure 7: Path refinement learning real-time search.

First, algorithms running at the higher levels of abstraction reason over a much smaller (abstracted) search space (e.g., Figure 4). Consequently, the number of states expanded by A* is smaller and the execution is faster.

Second, when LRTS learns at a higher abstraction level, it maintains the heuristic at that level. Thus, a single update to the heuristic function effectively influences the agent's behavior on a set of ground-level states. Such a generalization via state abstraction reduces the convergence time.

Third, algorithms operating at lower levels are *restricted* to the corridors c_i. This focuses their operation on more promising areas of the state space and speeds up search (in the case of A*) and convergence (in the case of LRTS).

Empirical Evaluation

We evaluated the benefits of state abstraction in learning real-time heuristic search by running PR-LRTS against the incremental A* search, LRTA*, and LRTS for path-finding on 9 maps from Bioware's "Baulder's Gate" game. The maps ranged in size from 64×60 to 144×148 cells, averaging 3212 passable states. Each state had up to 8 passable neighbors and the agent's visibility radius was set to 10. LRTS and PR-LRTS have been run with a number of parameters and a representative example is found in Table 3. Starting from the top entry in the table: incremental A* shows an impressive convergence travel (only three times longer than the shortest path) but has a substantial first-move lag of 5.01 ms. LRTA* with the lookahead of 1 is about 250 times faster but travels 140 times more before convergence. LRTS($d = 10$, $\gamma = 0.5$, $T = 0$) has less than 20% of A*'s first-move lag and does only 2% of LRTA*'s travel. State abstraction in PR-LRTS (with A* at level 0 and LRTS(5,0.5,0.0) at level 1) reduces the conver-

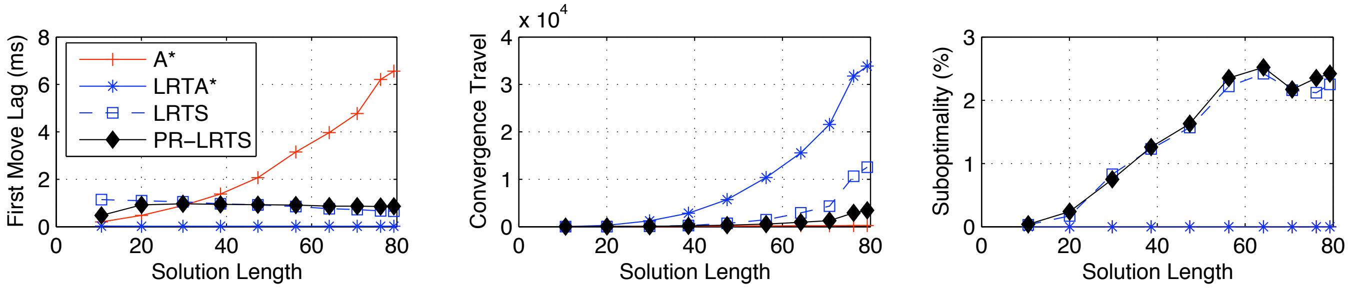

Figure 6: First-move lag, convergence travel, and final solution suboptimality over the optimal solution length.

gence travel by an additional 40% while preserving the lag time of LRTS. Figure 6 plots the performance measures over the optimal solution length. PR-LRTS appears to scale well and its advantages over the other algorithms become more pronounced on more difficult problems.

Conclusions and Future Work

We have considered some of the challenges imposed by real-time path-finding as faced by mobile robots in unknown terrain and characters in computer games. Such situated agents must react quickly to the commands of the user while at the same time exhibiting reasonable behavior. As the first result, combining three complementary extensions of the most popular real-time search algorithm, LRTA*, yielded substantially faster convergence for path-finding tasks. We then introduced state abstraction for learning real-time search. The dynamically built abstraction levels of the map increase performance by: (i) constraining the search space, (ii) reducing the amount of updates made to the heuristic function, thereby accelerating convergence, and (iii) generalizing the results of learning over neighboring states.

Future research will investigate if the savings in memory gained by learning at a higher abstraction level will afford application of PR-LRTS to moving target search. The previously suggested MTS algorithm (Ishida & Korf 1991) requires learning $O(n^2)$ heuristic values which can be prohibitive even for present-day commercial maps. Additionally, we are planning to investigate how the A* component of PR-LRTS compares with the incremental updates to the routing table in Trailblazer search (Chimura & Tokoro 1994) and its hierarchical abstract map sequel (Sasaki, Chimura, & Tokoro 1995). Finally, we will investigate sensitivity of PR-LRTS to the control parameters as well as the different abstraction schemes in path-finding and other domains.

Table 3: Typical results averaged over 50 convergence runs on 10 maps. The average shortest path length is 59.6.

Algorithm	1st move time	Conv. travel	Suboptimality
A*	5.01 ms	186	0.0%
LRTA*	0.02 ms	25,868	0.0%
LRTS	0.93 ms	555	2.07%
PR-LRTS	**0.95 ms**	**345**	**2.19%**

References

Barto, A. G., and Mahadevan, S. 2003. Recent advances in hierarchical reinforcement learning. *DEDS* 13:341 – 379.

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *AIJ* 72(1):81–138.

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

Buro, M. 2002. ORTS: A hack-free RTS game environment. In *Proceedings of Int. Computers and Games Conference*, 12.

Chimura, F., and Tokoro, M. 1994. The Trailblazer search: A new method for searching and capturing moving targets. In *Proceedings of the National Conf. on Artificial Intelligence*, 1347–1352.

Dietterich, T. G. 1998. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, 118–126.

Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1995. Hierarchical A*: Searching abstraction hierarchies efficiently. Technical Report tr-95-18, U. of Ottawa.

Holte, R.; Mkadmi, T.; Zimmer, R. M.; and MacDonald, A. J. 1996. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence* 85(1-2):321–361.

Ishida, T., and Korf, R. 1991. Moving target search. In *Proceedings of the Int. Joint Conf. on Artificial Intelligence*, 204–210.

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 *IEEE Conf. on Man, Systems, and Cybernetics*.

Koenig, S., and Likhachev, M. 2002. D* Lite. In *Proceedings of the National Conference on Artificial Intelligence*, 476–483.

Koenig, S.; Tovey, C.; and Y., S. 2003. Performance bounds for planning in unknown terrain. *Artificial Intelligence* 147:253–279.

Koenig, S. 2004. A comparison of fast search methods for real-time situated agents. In *Proceedings of the 3rd Int. Joint Conf. on Autonomous Agents and Multiagent Systems - vol. 2*, 864 – 871.

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

Korf, R. 1993. Linear-space best-first search. *AIJ* 62:41–78.

Sasaki, T.; Chimura, F.; and Tokoro, M. 1995. The Trailblazer search with a hierarchical abstract map. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 259–265.

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

Shue, L.-Y., and Zamani, R. 1993. An admissible heuristic search algorithm. In *Proceedings of the 7th Int. Symp. on Methodologies for Intel. Systems (ISMIS-93)*, volume 689 of *LNAI*, 69–75.

Shue, L.-Y.; Li, S.-T.; and Zamani, R. 2001. An intelligent heuristic algorithm for project scheduling problems. In *Proceedings of the 32nd Annual Meeting of the Decision Sciences Institute*.

Stenz, A. 1995. The focussed D* algorithm for real-time replanning. In *Proceed. of the Int. Conf. on Artificial Intel.*, 1652–1659.