Journal of Artificial Intelligence Research 54 (2015) 123–158 Submitted 04/15; published 09/15

Achieving Goals Quickly Using Real-time Search: Experimental Results in Video Games

Scott Kiesel SKIESEL AT CS.UNH.EDU
Ethan Burns EABURNS AT CS.UNH.EDU
Wheeler Ruml RUML AT CS.UNH.EDU
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA

Abstract

In real-time domains such as video games, planning happens concurrently with execution and the planning algorithm has a strictly bounded amount of time before it must return the next action for the agent to execute. We explore the use of real-time heuristic search in two benchmark domains inspired by video games. Unlike classic benchmarks such as grid pathfinding and the sliding tile puzzle, these new domains feature exogenous change and directed state space graphs. We consider the setting in which planning and acting are concurrent and we use the natural objective of minimizing goal achievement time. Using both the classic benchmarks and the new domains, we investigate several enhancements to a leading real-time search algorithm, LSS-LRTA*. We show experimentally that 1) it is better to plan after each action or to use a dynamically sized lookahead, 2) A*-based lookahead can cause undesirable actions to be selected, and 3) on-line de-biasing of the heuristic can lead to improved performance. We hope this work encourages future research on applying real-time search in dynamic domains.

1. Introduction

In many applications, it is desirable for an agent to achieve an assigned task as quickly as possible. Consider the common example of navigation in a video game. When a user selects a destination for a character to move to, they expect the character to begin moving immediately and to arrive at its destination as soon as possible. This suggests a planning strategy featuring concurrent planning and execution. The area of real-time heuristic search has been developed to address this problem. Algorithms in this class perform short planning episodes that are limited by a provided real-time bound, finding partial solutions and beginning execution before a complete plan to a goal has been found. While solution quality and search time are traditional heuristic search metrics, real-time heuristic search algorithms are usually compared by the length of the trajectories they execute.

Most recent work in real-time heuristic search has focused on grid pathfinding problems because of their simplicity. While important, grid pathfinding has some characteristics that do not exist in other search problems: the search space is undirected and small enough to easily fit in memory. We explore the use of real-time heuristic search on two additional domains that more closely reflect features of dynamic application domains, such as robotics. One is a platform-based pathfinding domain proposed by Burns, Ruml, and Do (2013b), and the other is a novel domain that we call the *traffic* problem, featuring navigation through a field of moving obstacles. Unlike the traditional grid pathfinding problem used to evaluate real-time search, both of these benchmarks have dynamics and large state spaces that form directed graphs.

In addition to evaluating real-time heuristic search on new domains, we introduce three modifications to LSS-LRTA* (Koenig & Sun, 2008), which is among the state-of-the-art in real-time heuristic search algorithms. First, we show that LSS-LRTA*, which executes multiple actions per planning episode, can be improved by executing only a single action at a time. Second, it has become the standard practice to construct the local search space of a real-time search using a partial A* search. We show that, if care is not taken to compare search nodes correctly, the agent may execute unnecessary actions. Third, we show that applying on-line de-biasing of the heuristic used during search can significantly reduce the overall goal achievement time. Together, these modifications can be easily applied to improve the overall performance of an agent being controlled by a real-time heuristic search algorithm. Videos illustrating our new domains and the discussed algorithms are provided on-line (Kiesel, Burns, & Ruml, 2015b) as well as described in Appendix B.

Instead of comparing techniques based solely on solution length or convergence time, we evaluate our new methods by comparing their goal achievement times—the time from when the problem is issued until the goal is achieved. This metric follows naturally from our benchmark domains and allows us to easily compare real-time search algorithms with off-line planning techniques such as A*. Our results show that A*, which performs optimally with respect to the number of expansions required to produce an optimal solution, can easily be outperformed when one cares about goal achievement time. We hope that this work and its methodology will encourage future research on applying real-time search in dynamic domains.

2. Previous Work

There has been much work in the area of real-time search since it was initially proposed by Korf (1990). In this section we will review those real-time search algorithms most relevant for our study. (Additional related algorithms are reviewed in Section 7.)

2.1 LRTA*

Many real-time search algorithms are considered *agent-centered* because the agent performs a bounded amount of lookahead search rooted at its current state before acting. Since the size of each lookahead search is bounded, the agent can respect its real-time constraints by restricting its lookahead to be completed by the time that the real-time limit has been reached. In his seminal paper, Korf (1990) presents Learning Real-time A* (LRTA*), a complete, agent-centered, real-time search algorithm. To select the next action to perform, LRTA* uses the action costs and an estimate of the cost-to-goal, or heuristic value, for the states resulting from applying each of its current applicable actions; it chooses to execute the action that has the lowest estimated cost-to-goal.

LRTA* estimates the heuristic value for states in two different ways. First, if a state has never been visited before, then it uses a depth-bounded, depth-first lookahead search. The estimated cost of the state is the minimum f value among all leaves of the lookahead search, where f is the cost-so-far (notated as g) plus the estimated cost-to-goal (notated as

LSS-LRTA*(s, *expansion_limit*)
1. until a goal is reached
2. perform *expansion_limit* expansions of best-first search on f from s
3. update heuristic values of nodes in $CLOSED$
4. $s \leftarrow$ state on $OPEN$ with the lowest f
5. start executing path to s
6. $OPEN \leftarrow \{s\}$; clear $CLOSED$

Figure 1: Pseudocode for LSS-LRTA*.

h). The second way that it estimates cost is through learning. Each time LRTA* performs a search, it learns an updated heuristic value for its current state. If this state is encountered again, the learned estimate is used instead of searching again. Korf (1990) proved that as long as the state's heuristic estimate is increased after each move by an amount bounded from below by some ϵ, then the agent will never get into an infinite cycle, and the algorithm will be complete. In the original algorithm, the second best action's heuristic value is used to update the cost estimate of the current state before the agent moves.

2.2 LSS-LRTA*

Local Search Space Learning Real-time A* (LSS-LRTA*, Koenig & Sun, 2008) is currently one of the most popular real-time search algorithms. LSS-LRTA* has two big advantages over the original LRTA*: it has much less variance in lookahead times and it does significantly more learning. LRTA* can have a large variance in its lookahead times because, even with the same depth limit, different searches can expand very different numbers of nodes due to pruning. Instead of using bounded depth-first search beneath each successor state, LSS-LRTA* uses a single A* search rooted at the agent's current state. The A* search can be limited by an exact number of nodes to expand, so there is significantly less variance in lookahead times. The second advantage is that the original LRTA* only learns updated heuristics for states that the agent has visited; LSS-LRTA* learns updated heuristics for every state expanded in each lookahead search. This is accomplished using Dijkstra's algorithm to propagate more accurate heuristic values from the fringe of the lookahead search back to the interior before the agent moves. Koenig and Sun showed that LSS-LRTA* can find much cheaper solutions than LRTA* and that it is even competitive with a state-of-the-art incremental search, D*Lite (Koenig & Likhachev, 2002).

Another major difference between LRTA* and LSS-LRTA* is how the agent moves. In LRTA*, after each lookahead, the agent moves by performing a single action; in LSS-LRTA* the agent moves all the way to the node on the fringe of its current lookahead search with the lowest f value. As a result, the agent performs many fewer lookahead searches before reaching a goal. If one is concerned with minimizing the total number of expansions, this may be advantageous. However, as we will see below, when search and execution are allowed to happen in parallel, the movement method of LSS-LRTA* can actually be detrimental to performance. Pseudocode for LSS-LRTA* is presented in Figure 1.

3. Evaluating Real-time Search Algorithms

Traditionally, real-time heuristic search algorithms have been evaluated using two criteria: convergence time and solution length. Convergence time measures the number of repeated start-to-goal plans that an algorithm must execute before it learns the optimal path between a given start and goal pair. While it is useful for comparing the rate at which different algorithms learn more accurate heuristic values, it does not seem to be as useful in practice; agents rarely need to repeatedly plan between exactly the same start and goal states. Often just one solution is needed, and an algorithm that finds a better first solution is preferred even if it takes a long time to converge.

Solution length is the number of actions executed to achieve the goal. In real-time search, where planning and action execution can happen in parallel, solution length is a good proxy for the amount of time between when a real-time agent is given a problem and when the goal is actually achieved. The downside of simply using the solution length, however, is that it makes comparison with offline techniques unfair. For example, when comparing algorithms solely by the solution length, no technique can perform better than an optimal search like A*. But, in practice, A* may not be the best method to solve the problem. An agent using A* could spend a very long time planning before it finally begins executing an optimal path, but an agent using a real-time algorithm may start executing a long path right away, and consequently it can arrive at the goal first.

3.1 Goal Achievement Time

Recently, Hernández, Baier, Uras, and Koenig (2012) introduced the *game time* model for evaluating real-time heuristic search algorithms. In the game time model, time is divided into uniform intervals. During each interval, an agent has three choices: it can search, it can execute an action, or it can both search and execute in parallel. The objective of the game time model is for the agent to move from the initial state to a goal state using the fewest time intervals. The advantage of the game time model is that it allows for comparisons between real-time algorithms that search and execute in the same time step and off-line algorithms, like A*, that search first and execute only after all search has completed. For our experiments, we compare algorithms directly on their *goal achievement time*. Goal achievement time (GAT) is a slight generalization of the game time model that allows for real-valued times, not just fixed-size discrete time intervals. It is computed as the planning time plus the execution time minus the time spent planning and executing in parallel:

$$\textit{goal achievement time} = \textit{time}_{\text{planning}} + \textit{time}_{\text{executing}} - \textit{time}_{\text{both}}$$

Since some of the benchmark domains used in our experiments have no natural definition of execution time (e.g., in the 15-puzzle, exactly how much time is needed to slide a tile?), we present results using a variety of different execution times. We define execution time as the number of seconds required to execute a unit-cost action. We call this value the *unit action duration*; it effectively converts action costs into units of time. For example, on an 8-way grid pathfinding problem where diagonal edges cost $\sqrt{2}$, we can simply multiply the edge costs by the unit action duration to convert them to seconds of execution. A large unit action duration models an agent that moves slowly and a small unit action duration models an agent that moves quickly, relative to its planning speed.

In the two following sections, we present modifications to the LSS-LRTA* algorithm. The benefit of each modification is evaluated using goal achievement time.

4. Lookahead Commitment

An important step in a real-time search algorithm is selecting how far to move the agent before the next phase of planning begins. As mentioned above, in the original LRTA* algorithm the agent moves a single step, while in LSS-LRTA* the agent moves all of the way to a frontier node in the local search space. Luštrek and Bulitko (2006) reported that solution length increased when switching from a single-step to multi-step policy using the original LRTA* algorithm. It was unclear if this behavior would carry over given the increased learning performed by LSS-LRTA* and the use of a new goal achievement metric.

4.1 Single-step and Dynamic Lookahead

We implemented a standard LSS-LRTA* as well as a version that executes single actions like LRTA*. We also implemented LSS-LRTA* using a dynamic lookahead strategy that executes multiple actions leading from the current state to a selected state on the fringe of the most recent local search. With a dynamic lookahead, the agent selects the amount of lookahead search to perform based on the duration of its currently-executing trajectory. When the agent commits to executing multiple actions, it simply adjusts its lookahead to fill the entire execution time.

Because of the learning step, any algorithm based on LSS-LRTA* cannot simply search until the real-time bound expires — it must leave time for learning. To account for this we use offline training to determine the speed at which the agent searches. With the fixed lookahead algorithms, it is necessary to know the maximum lookahead size that the agent can search during the minimum action execution time. This can be found by simply running the search algorithm with different fixed lookahead settings on a representative set of training instances and recording the per-step search times. In the case of dynamic lookahead, the agent must learn a function mapping durations to lookahead sizes. When the agent commits to a trajectory that requires time t to execute, then it must use this function to find $l(t)$, the maximum lookahead size that the agent can search in time t. Note that, because the data structures used during search often have non-linear-time operations, this function may not be linear. It is possible to create a conservative approximation of $l(t)$ by running an algorithm on a representative set of training instances with a large variety of fixed lookahead sizes. The approximation of $l(t)$ selects the largest lookahead size that always completed within time t.

4.2 Experimental Evaluation

We compare the different techniques on the platform pathfinding benchmark of Burns et al. (2013b). This domain is inspired by popular platform-based video games like Super Mario Bros. The agent must find a path, jumping from platform to platform, through a maze. A screenshot of the domain and an example instance is shown in Figure 2. Videos are also available on-line (Kiesel et al., 2015b) and described in Appendix B.

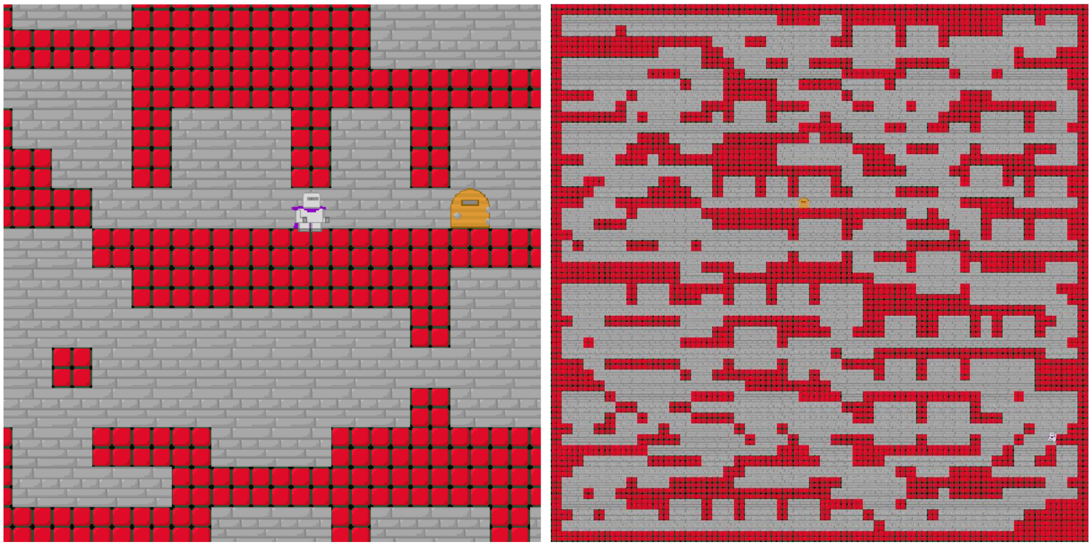


Figure 2: A screenshot of a problem instance in the platform path-finding domain (left), and a zoomed-out image of the entire instance (right). The knight must find a path from its starting location, through a maze, to the door (on the right side in the left image, and just above the center in the right image).

The available actions are different combinations of controller keys that may be pressed during a single iteration of the games main loop: left, right, and jump. Left and right move to the knight in the respective directions (holding both at the same time is never considered by the search domain, as the movements would cancel each other out, leaving the knight in place), and the jump button makes the knight jump, if applicable. The knight can jump to different heights by holding the jump button across multiple actions in a row up to a maximum of 8. The actions are unit cost.

Each state in the state space contains the x, y position of the knight using double precision floating point values, the velocity in the y direction (x velocity is not stored as its determined solely by the left and right actions), the number of remaining actions for which pressing the jump button will add additional height to a jump, and a boolean stating whether or not the knight is currently falling. The knight moves at a speed of 3.25 units per frame in the horizontal direction, it jumps at a speed of 7 units per frame, and to simulate gravity while falling, 0.5 units per frame are added to the knights downward velocity up to a maximum of 12 units per frame.

This benchmark is a natural fit for real-time search algorithms, since the agent must decide on an action to execute before it is forced to move due to gravity. The state space for the platform domain is directed, because while in the air the agent's actions are not reversible. The heuristic is based on visibility navigation (see Burns et al. for details) and it is quite accurate except that it does not account for the player's limited jumping height. The C++ source code is available on GitHub (Kiesel, Burns, & Ruml, 2015a). All

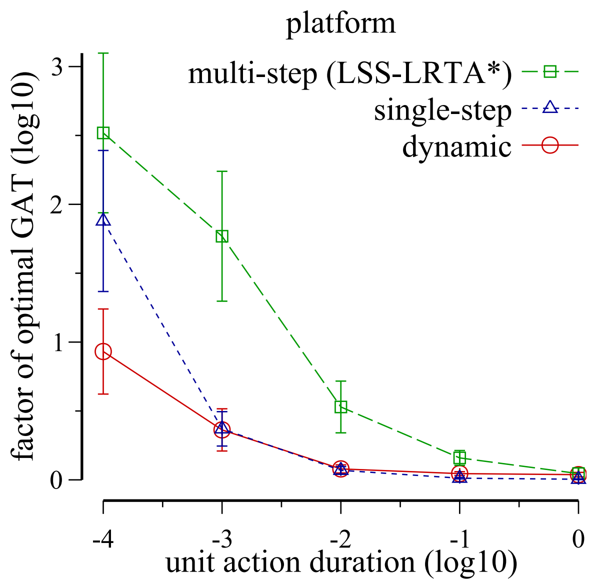


Figure 3: LSS-LRTA*: multi-step, single-step, and dynamic lookahead.

experiments were run on a Core2 duo E8500 3.16 GHz with 8GB RAM running Ubuntu 10.04.

For our experiments we used 25 test instances created using the level generator described by Burns et al. (2013b), where the maze for each instance was unique and had a random start and goal location. We used the offline training techniques described above to learn the amount of time required to perform different amounts of lookahead search. For the offline training, we generated an additional 25 training instances. The lookahead values used were 1, 5, 10, 20, 50, 100, 200, 400, 800, 1000, 1500, 2000, 3000, 4000, 8000 10000, 16000, 32000, 48000, 64000, 128000, 196000, 256000, and 512000. For algorithms that use a fixed-size lookahead, the lookahead value was selected by choosing the largest lookahead size for which the mean step time on the training instances was within a single action duration. If none of the lookahead values were fast enough to fit within a single action time for a given action duration, then no data is reported. Our implementation used the mean step time instead of the maximum step time, as the latter was usually too large due to very rare, slow steps. We attribute these outliers to occasional, unpredictable overhead in system-related subroutine calls such as memory allocation. We suspect that this issue would go away if a true real-time operating system were used, where such operations perform more predictable computations, or if a domain-specific optimized implementation were used. (Unfortunately, developing such optimized implementations would have made it much more difficult to perform thorough scientific comparisons.)

Figure 3 shows a comparison of these different techniques for the LSS-LRTA* algorithm. The y axis shows the goal achievement time as a factor of the optimal goal achievement time for each instance. The optimal goal achievement time is computed as the GAT of an optimal solution with no planning time taken into account. One could imagine an oracle that is able to instantly provide the optimal set of actions to execute. In the plot, the y h

3	3	2	2	1	1	0
g=2 f=5	**g=1 f=4**	**g=2 f=4**	g=3 f=5	g=4 f=5	g=5 f=6	g=6 f=6
g=1 f=4	S	g=1 f=3	**g=2 f=4**	**g=3 f=4**	g=4 f=5	☆
g=2 f=5	**g=1 f=4**	**g=2 f=4**	g=3 f=5	g=4 f=5	g=5 f=6	g=6 f=6

Figure 4: Example of heuristic error and f layers.

axis is shown on a $\log_{10}$ scale. We consider a variety of action durations, these are shown on the x axis, also on a $\log_{10}$ scale. Smaller action durations represent an agent that can move relatively quickly, so spending a lot of time planning to make small decreases in solution cost may not be worth the time. For larger values, the agent moves more slowly, and it may be worth planning more to execute cheaper paths. The unit action duration is used to limit the number of expansions performed in each iteration.

Each point on the plot shows the mean goal achievement time out of the 25 test instances solved by all algorithms given as the factor of the optimal goal achievement time at that action duration. A value of $log_{10}(0) = 1$ indicates that the given algorithm had an optimal time. Error bars show the 95% confidence intervals on the means.

From this plot, it is clear that the multi-step approach (standard LSS-LRTA*) performed worse than both the single-step and the dynamic lookahead variants. This is likely because the multi-step technique commits to too many actions with only a little bit of planning—the same amount of planning that the single-step variant uses to commit to just one action. As the unit action duration was increased to 1 second, the algorithms started to perform similarly. However, single-step and dynamic lookahead still appear to perform slightly better. Note that there were 0.02 seconds per frame in the game from which the platform domain was derived, so values greater than $log_{10}(0.02) \approx -1.7$ represent an agent that moves at an unusually slow pace.

It is also important to note that for some very small unit action durations, one algorithm may perform better than another, while as the unit action duration increases, this relationship inverts. At the very small unit action durations, there is not much time for search to be performed.

5. A*-based Lookahead

In standard LSS-LRTA*, the lookahead search is A*-based, so nodes are expanded in f order. After searching, the agent moves to the node on the open list with the lowest f value. While this may seem intuitively reasonable, we will show why this choice can be problematic, and we will see how it can be remedied.

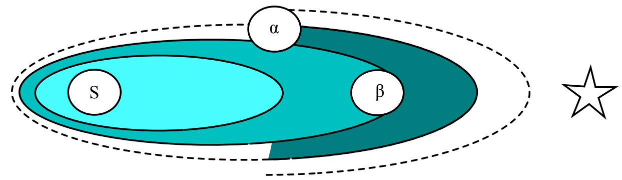


Figure 5: f-layered lookahead.

5.1 Heuristic Error

The crux of the problem is that f-based lookahead doesn't account for heuristic error. The admissible heuristic used to compute f is, by definition, low-biased, so f will typically optimistically underestimate the true solution cost through each node. Because of this heuristic error, not all nodes with the same f value will actually lead toward the goal node. Figure 4 shows an example using a simple grid pathfinding problem. In the figure, the agent is located in the cell labeled 'S' and the goal node is denoted by the star. The admissible h values are the same for each column of the grid; they are listed across the top of the columns. The g and f values are shown in each cell. Cells with $f = 4$ are bold, and the rest are light gray. We can see that nodes with equivalent f values form elliptical rings around the start node. In the heuristic search literature, these are referred to as f *layers*. While some nodes in an f layer are closer to the goal node, there are many nodes in each layer that are not—some nodes in an f layer will even be exactly away from the goal. In this simple problem, the optimal solution is to move the agent right until the goal is reached, however, of the 7 nodes with $f = 4$, only 2 nodes are along this optimal path; the other nodes are not, but they have the same f value because of the heuristic error. If the agent were to move to a random node with $f = 4$, chances are it will not be following the optimal path to the goal.

One way to alleviate this problem is to use a second criterion for breaking ties among nodes with the same f value. A common tie breaker is to favor nodes with lower h values as, according to the heuristic, these nodes will be closer to the goal. We can see, in Figure 4 that among all nodes in the $f = 4$ layer, the one with the lowest h value ($h = 1$) is actually along the optimal path. In LSS-LRTA* this tie breaking is insufficient, because when LSS-LRTA* stops its lookahead, it may not have generated all of the nodes in the largest f layer. If the node with $h = 1$ was not generated, then even with tie breaking, the agent can be led astray.

5.2 Incomplete f Layers

These incomplete f layers cause other problems too. Recall that in LSS-LRTA*, the agent moves to the node at the front of the open list. If low-h tie breaking is used to order the expansions of the local search space, then the best nodes in the first f layer on the open list will actually be expanded first and will not be on the open list when it comes time for the agent to move. Figure 5 shows the problem diagrammatically. As before, the agent is at the node labeled 'S' and the goal is denoted by the star. Each ellipse represents a different f layer, the shaded portions show closed nodes, darker shading denotes nodes with larger

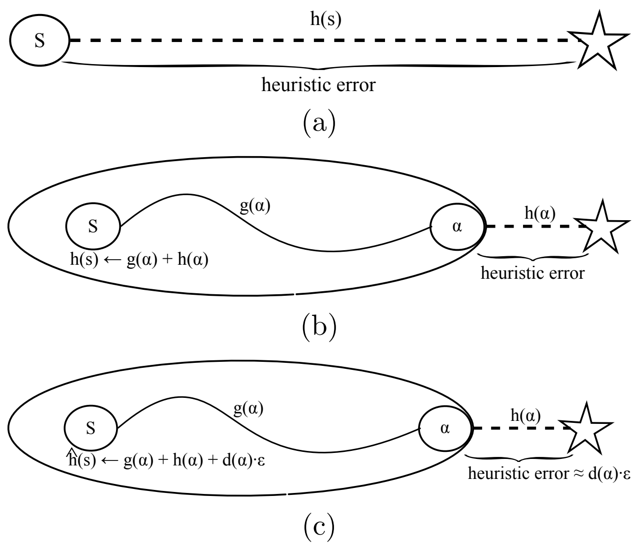


Figure 6: (a) A standard heuristic and its error. (b) An updated heuristic and its error. (c) Using an updated heuristic and accounting for heuristic error.

f values, and the dotted lines surround nodes on the open list. As we can see, the closed nodes with the largest f values cap the tip of the second-largest f layer. This is caused by low-h tie breaking where the first open nodes to be expanded and added to the closed list will be those that have the lowest h. These are the nodes on the portion of the f layer that are nearest to the goal. If the agent moves to the node on its open list with the lowest f value, tie breaking on low h, then it will select node α and will not take the best route toward the goal.

5.3 Improving Lookahead Search

We have demonstrated two problems: 1) because of heuristic error, f layers can contain a large number of nodes, many of which do not lead toward the goal, and 2) even with good tie breaking, LSS-LRTA* may miss the good nodes because it only considers partial f layers when deciding where to move. Next we present two possible solutions to these problems.

The first is quite simple. When choosing where to move, select a node with the lowest h value on the completely expanded f layer with the largest f value, not the next node on open. In Figure 5, this corresponds to the node labeled β. We call this the "complete" technique, as it considers only completely expanded f layers instead of partially expanded, incomplete layers.

The second technique explicitly accounts for heuristic error—it orders both the search and the agent's action selection not on f, but on a less-biased estimate of solution cost. Ideally, we would prefer to use an unbiased (and hence inadmissible) estimate that accounts for and attempts to correct heuristic error. While any inadmissible heuristic could be used, we note that Thayer, Dionne, and Ruml (2011) investigated the use of inadmissible heuristics for offline search and here we adopt their simple heuristic correction technique for real-time search. We call this estimate $\hat{f}$. Like f, $\hat{f}$ attempts to estimate the solution cost through a node in the search space. Unlike f, $\hat{f}$ is not explicitly biased—it is not a lower bound. $\hat{f}$ is computed similarly to f, however, it attempts to correct for the heuristic error by adding in an additional term:

$$\hat{f}(n) = \overbrace{g(n) + h(n)}^{f} + \overbrace{d(n) \cdot \epsilon}^{\text{error}}$$

where ϵ is the average single-step error in the heuristic, and the additional term $d(n) \cdot \epsilon$ corrects the error by adding ϵ back to the cost estimate for each of the $d(n)$ steps estimated to remain from n to the goal. Following Thayer et al. (2011), we make the simplifying assumption that the error in the heuristic is distributed evenly among each of the actions on the path from a node to the goal.

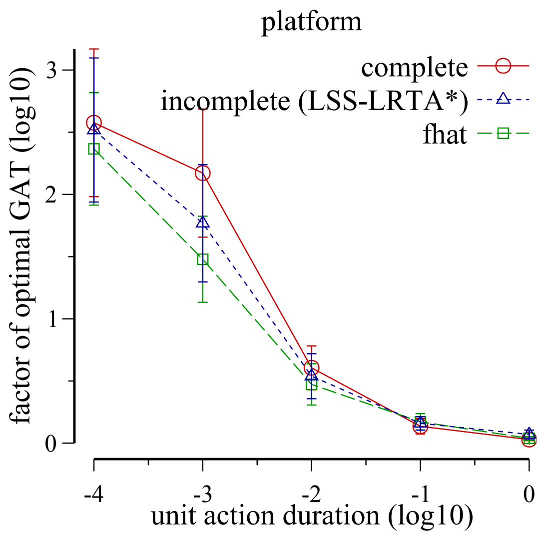
Distance estimates d are readily available for many domains; they tend to be just as easy to compute as heuristic estimates (Thayer & Ruml, 2009). To estimate the single-step heuristic error, we use an average of the difference in the f values between each expanded node and its best child. This difference accounts for the amount of heuristic error due to the single step between a parent node and its child. With a perfect heuristic, one with no error, the f values of a parent node and its best child would be equal—some of the f will simply have moved from h into g:

$$\begin{aligned} f(\mathit{parent}) &= f(\mathit{child}), \text{ in the ideal case, so} \\ h(\mathit{parent}) &= h(\mathit{child}) + c(\mathit{parent}, \mathit{child}), \text{ and} \\ g(\mathit{parent}) &= g(\mathit{child}) - c(\mathit{parent}, \mathit{child}) \end{aligned}$$

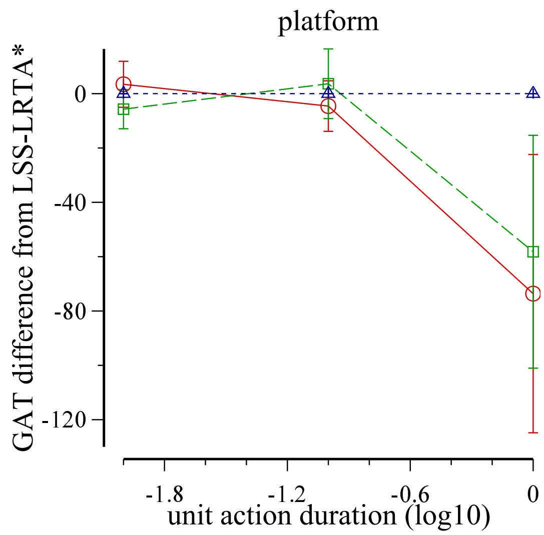
Since g is known exactly, as is the cost of the edge $c(\mathit{parent}, \mathit{child})$, with an imperfect heuristic any difference between $f(\mathit{child})$ and $f(\mathit{parent})$ must be caused by error in the heuristic over this step. Averaging these differences gives us our estimate ϵ.

Adapting this technique to real-time search requires some subtlety. In real-time search algorithms like LSS-LRTA*, the heuristic values of nodes that are expanded during a lookahead search are updated each time the agent moves. Figure 6a schematically depicts the error in the default heuristic value for a node S. Its error is accrued over the distance from S to the goal. After lookahead (Figure 6b), the updated heuristics are more accurate than the originals because they are based on the heuristic values of nodes that are closer to the goal, and thus have less heuristic error. Here, we can see that α is the node on the fringe from which the start state inherits its updated heuristic value. Since $g(\alpha)$, the cost from S to α, is known exactly, the error in the backed up heuristic now comes entirely from the steps between α and the goal. Since α is closer to the goal, the error is less than the error of the original heuristic value for S.

When computing $\hat{f}(S)$ in a real-time search, it is necessary to account for the fact that error in the updated heuristic comes from the node α. To do this, we track a value called derr, the distance over which heuristic error is accrued, for each node, and we use it to compute $\hat{f}$. Initially, for nodes without updated heuristic values $\mathit{derr}(n) = d(n)$. After performing a lookahead search, h is updated from backed-up h values as before. If $h(n)$ receives a backed up value that originated at node α, then we set $\mathit{derr}(n) = d(\alpha)$, since the error in the updated heuristic comes from the instance between the fringe node α an



(a)



(b)

Figure 7: LSS-LRTA*: f-based lookahead and $\hat{f}$-based lookahead.

the goal, not the distance between n and the goal. The updated heuristic, when accounting for heuristic error, is $\hat{h}(s) = g(\alpha) + h(\alpha) + derr(n) \cdot \epsilon$, where $g(\alpha) + h(\alpha)$ is the standard heuristic backup and $derr(n) \cdot \epsilon$ is the error correction (cf Figure 6b, $derr(n) = d(\alpha)$ due to the update). This is demonstrated by Figure 6c. Our new technique uses $\hat{f}$ to order expansions during the lookahead search in LSS-LRTA*, and it moves the agent toward the node on the open list with the lowest $\hat{f}$ value.

Figure 7 shows a comparison of the three node selection techniques: the standard *incomplete* f layer method of LSS-LRTA*, the *complete* f-layer method, and the approach that uses $\hat{f}$ (denoted *fhat*). To better demonstrate the problem with the standard approach, the plot shows results for the multi-step movement model that commits to an entire path from the current state to the fringe of the local search space after each lookahead. The style of the plot in panel (a) is the same as for Figure 3.

In this figure, we can see that *complete* performed worse than the standard LSS-LRTA* algorithm for small action durations where the agent may not have time to expand many nodes and thus ignoring some expansions has a large effect. For longer action durations, however, this performance improves and *complete* becomes the best performer on the right side of the plot where cheaper solutions are preferred. To clarify the improvement on the right side of the plot in Figure 7 (a), we have included Figure 7 (b). This plot has a different y-axis that highlights the improvement by comparing each algorithm directly to the incomplete version of LSS-LRTA*. This indicates that using the completed f layer does lead to fewer extraneous actions and gives cheaper solutions. Using $\hat{f}$ to sort the open list of lookahead searches performs much better than the other two algorithms on the left side of the plot, although it begins to perform slightly worse as the unit action duration is increased. This is likely because the inadmissibility in $\hat{f}$ hinders its ability to find solutions that are as cheap as those found by the complete f layer variant.

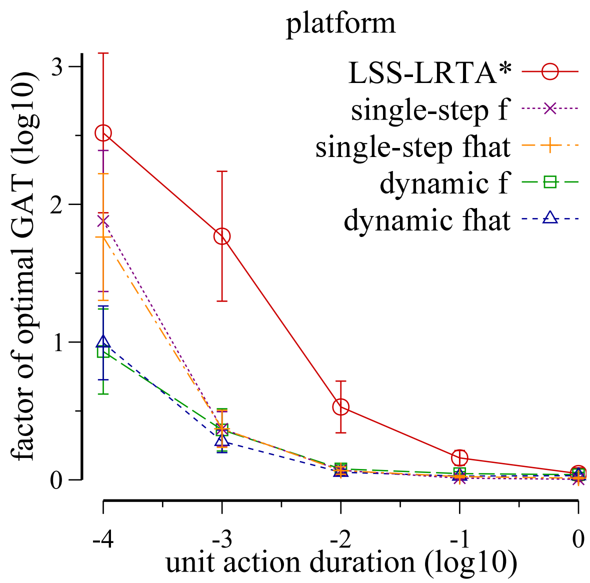


Figure 8: Comparison of the four new real-time techniques.

Dynamic-$\hat{f}$(s, *expansion_limit*)

```
until a goal is reached
    perform expansion_limit expansions of best-first search on f̂ from s
    update heuristic values of nodes in CLOSED
    s ← state on OPEN with the lowest f̂
    start executing path to s
    execution_time ← execution time to reach s
    expansion_limit ← number of expansions possible within execution_time
    OPEN ← {s}; clear CLOSED
```

Figure 9: Pseudocode for LSS-LRTA* with a dynamic lookahead using $\hat{f}$.

Figure 8 shows the results of a comparison between the four combinations of single-step versus dynamic lookahead and f-based versus $\hat{f}$-based node ordering. In this domain, $\hat{f}$ with dynamic lookahead tended to give the best goal achievement times in portions of the plot where all algorithms did not have significant overlap (i.e., everywhere except for the right-half of the plot).

In both Figure 7 and Figure 8, ordering the lookahead search on $\hat{f}$ was the common link to the best performance out of all considered algorithms. Figure 7 demonstrates the initial intuition that heuristic correction can be used in real-time search. Figure 8 builds on this idea by adding in the dynamic look ahead proposed in the previous section to yield the strongest algorithm we have seen so far. We present pseudocode in Figure 9 for this dynamic-$\hat{f}$ method.

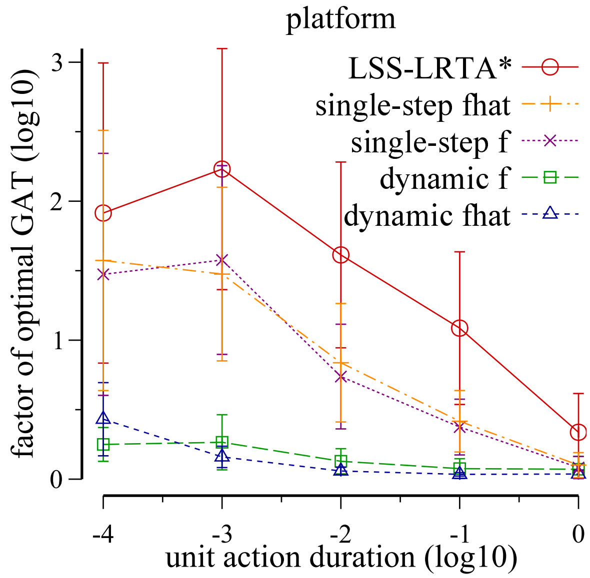


Figure 10: Comparison of the four new real-time techniques with a weak heuristic.

5.3.1 Heuristic Accuracy

It could be argued that the effects of dynamic lookahead and heuristic correction are distorted by such a strong heuristic as a visibility graph. In this short experiment, we replace the visibility graph heuristic in the platform domain with a much weaker euclidean distance heuristic. To solve instances using this heuristic, we had to decrease the overall size of the instances from 50x50 to 25x25.

Even with this decreased instance size, only the dynamic algorithms were able to solve all 25 instances for all unit action durations. The other algorithms, for some unit action durations, solved as few as 13 out of the full 25 instances. Figure 10 shows results for this experiment, each data point represents the mean over only those instances that were solved by all algorithms at that unit action duration (between 13 and 17 instances).

We can see that the ranking of the algorithms remains the same between the algorithms regardless of the weakening in heuristic. We note that at a unit action duration of 0.001 seconds there is a rise in the data. This can be attributed to solving a larger subset of the 25 instances containing more difficult instances, thus increasing the GAT.

5.3.2 CPU Usage

While this paper does not focus on minimizing CPU time and assumes that any time allocated to search may be utilized, it is still instructive to compare CPU usage of these new techniques. As one might imagine, there is a tradeoff between CPU usage and GAT. This inverse relationship can be seen by examining how each algorithm utilizes its search time. Single step policies will execute search during every action until the goal is reached. This behavior can increase the overall demand on the CPU. However, using a single step policy, the goal can be achieved more quickly than using a multi-step policy. Dynamic

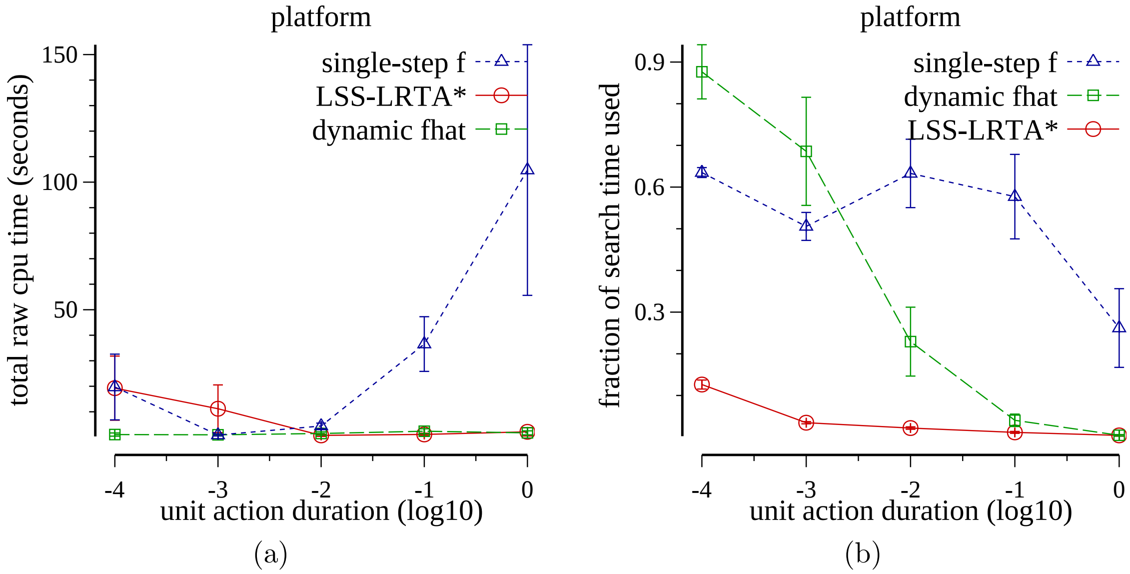


Figure 11: Comparison of LSS-LRTA*, single-step LSS-LRTA* and dynamic $\hat{f}$ in terms of cpu usage.

lookahead will greedily use more search time as it becomes available, but will arrive at the goal more quickly.

In Figure 11, search time is plotted. Figure 11 (a) shows the raw cpu time used by each algorithm in the platform domain. On the y axis is the raw cpu time in seconds and on the x axis is the $\log_{10}$ unit action duration. As can be seen from the plot, dynamic $\hat{f}$ uses a very small amount of planning time. This can be attributed to dynamic $\hat{f}$ finding the goal very quickly. While the single-step policy is able to find the goal more quickly than LSS-LRTA*(see Figure 3), it requires more cpu time to do so as the unit action duration increases.

Figure 11 (b) plots the raw cpu time divided by the goal achievement time. This provides an idea of how active the CPU is during execution for each of the algorithms. It is important to keep in mind that quick goal achievement times will have a small denominator, causing utilization to appear higher than for the same raw cpu time with a longer goal achievement time. The reduced utilization for longer action durations is likely because dynamic $\hat{f}$ is able to find goals quickly using a small number of iterations and the remainder of execution has no overlapping planning time.

5.3.3 Implementation Details

The $\hat{f}$ technique requires more information that standard LSS-LRTA*, so it has slightly greater storage requirements. We should note, however, that during our experiments we did not run into memory issues, so we did not optimize our implementation to reduce memory requirements.

Our implementation uses two different types of nodes: persistent and transient. Persistent nodes form the agent's memory of all states it has encountered during all past lookahead searches: their connectivity, learned heuristic values, and the distance estimates over which their heuristic error is accrued. Transient nodes exist for a single round of lookahead search and h-cost learning; they are akin to traditional search nodes used in, for example, an A* search, however they include information required to order the search on $\hat{f}$.

In each persistent node, we store information about the connectivity of the search graph. This includes the set of predecessors and successors of the node and the costs of the associated edges. We store the predecessors because we do not assume an undirected search graph or that a predecessor function is easily computable. Both the predecessors and successors are computed lazily. Each predecessor is added only when it is first expanded, while the entire set of successors is populated the first time the node is expanded. For successor nodes, we also store the cost of reversing the edge (which matters in the case where edge costs are not symmetric) and the operator used to generate that successor. Persistent nodes also store the learned heuristic estimate and cached values such as the original h and d estimates for the node, which would otherwise need to be computed each time they are needed.

Transient nodes hold additional information needed to perform a best first search ordered on $\hat{f}$. First, each transient node has a pointer to the corresponding persistent node. In addition, each transient node has the g-cost, f-cost, and $\hat{f}$-cost as computed during the single lookahead search for which the node persists. Other information contained in the transient nodes are: a pointer to the best parent during the current lookahead search; the node's index on the open list (which is implemented as an array-based binary heap), needed for updating the node's position in the heap if it is encountered via a better path; and two booleans used during h-cost learning to easily determine if the node has already had its h value updated and to determine if it is on the closed list. For more detail, we refer the reader to the source code which is freely available on GitHub (Kiesel et al., 2015a).

The only differences between the information stored by $\hat{f}$ and our normal LSS-LRTA* implementation is that the latter does not store d estimates in its persistent node set, and it does not store $\hat{f}$ values in its transient nodes. All other information is exactly the same.

5.3.4 Theoretical Evaluation

Here we prove that, under certain conditions, our modifications to LSS-LRTA* retain the completeness property of the algorithm. This is because learning will cause $\hat{f}$ to converge to f^*. To begin, we assume that the heuristic is admissible and the state space is finite.

Proposition 1. *Following the spirit of Korf's completeness proof for LRTA*, we note that, if a search algorithm is incomplete in a finite state space, then there must exist nodes that are visited an infinite number of times.* [1]

Our goal now is to show that such nodes cannot exist.

Lemma 1. *If dynamic $\hat{f}$ searches only within some finite set of nodes D, the h values of the nodes that are in the interior of D will reach a fixed point (remain static and unchanging) at some finite time T_1, for at least as long as the search remains within D.*

1. We note that, contrary to the assumptions of some previous work, the algorithm need not actually enter a loop, as the trajectory may vary in a non-repeating way, just as the digits of π are conjectured to.

Proof.

1. Because h is updated using the same update rule as in LSS-LRTA*, all h values in the state space are non-decreasing via Koenig and Sun's (2008) Theorem 1 regarding LSS-LRTA*.
2. Note that the h values of nodes on the fringe of D will remain static because the h-value learning only updates the heuristic of nodes in the interior of an LSS. Every update during a learning step obeys $h(p) = max(h(p), c(p, bc) + h(bc))$, where bc is the best child of p — the one with the lowest f value. Thus every $h(p)$ value will be the sum of numbers drawn from the set C that contains: all edge costs, the set of all fringe h values, and the set of initial $h(p)$ values. Because there are a finite number of costs in C, any update to $h(p)$ must be larger than the minimum positive difference between any two possible sums of costs drawn from C. Thus the increases are bounded from below by a constant.
3. Similarly to step 1, the h values remain admissible via Theorem 2 of Koenig and Sun regarding LSS-LRTA*, so they cannot rise above the true cost to go.
4. By steps 1, 2, and 3, there must be a time T_1, after which the h values do not change for as long as the search remains within the set D. □

Lemma 2. *If a search visits some finite set of nodes D an infinite number of times, there exists a time T' after which the search visits only nodes in D.*

Proof. Consider the LSSes that are formed in each iteration and the learning step in which parents inherit h values from their child with the lowest f. Considering each of these pairs of nodes, there are two cases: a) those two nodes will be in the same LSS an infinite number of times as the number of search iterations approaches infinity, or b) these nodes will only be in the same LSS a finite number of times. For those pairs in case (b), note that there must exist a time, T', after which they are never in the same LSS, for otherwise the two nodes would have been covered by case (a) instead. □

Lemma 3. *If dynamic $\hat{f}$ searches only within some set D, then the one-step heuristic error ϵ goes to 0 by some time T_2 and stays there for at least as long as the search remains within D.*

Proof.

1. Consider those pairs of nodes in the same LSS after T'. By Lemma 1, there exists some T_1 after which the h values have converged. So after T_1, we know that $h(p) = c(p, bc) + h(bc)$.
2. Note that ϵ is the average, across all internal nodes in an LSS and their associated best children, of the differences between the f values of parents and their best children. By step 1, $f(p) = f(bc)$, so $\epsilon = 0$ holds after some time $T_2 \geq T_1$. □

Lemma 4. *There cannot exist a set of nodes D that dynamic $\hat{f}$ visits infinitely often.*

Proof.

1. For the sake of contradiction, let D be a set of nodes which are part of an LSS an infinite number of times.
2. By Lemma 3, ϵ will become 0 by some time T_2.

3. At time T_2 when $\epsilon = 0, \hat{h} = h, \hat{f} = h$, and so dynamic $\hat{f}$ will behave like LSS-LRTA* (dynamic-sized lookahead makes no difference to LSS-LRTA*'s theoretical properties as they hold without regard to lookahead size).

4. LSS-LRTA* is complete (Koenig & Sun, 2008, Thm. 3), so the search will eventually reach a goal. This contradicts 1, so dynamic $\hat{f}$ will not visit a set of states infinitely often. □

Note that, if dynamic $\hat{f}$ were to escape a potential set D, as described in step 1 of the proof of Lemma 4, but then circulate within another set of nodes D' for an infinite time, then D would have equaled D' instead. Also, if it were to oscillate between two or more sets, then D would be defined as the union of all such sets.

Theorem 1. *In a finite search space with admissible h, dynamic $\hat{f}$ will eventually reach a goal.*

Proof. Proof by contradiction:

1. Assume the search never reaches a goal. Because dynamic $\hat{f}$ goes to any goal in the LSS, this means that a goal is never in the LSS.

2. In a finite space, if the search never sees a goal, then it must visit other states an infinite number of times.

3. By Lemma 4, dynamic $\hat{f}$ will not exhibit such behavior.

4. Thus, using dynamic $\hat{f}$ retains the completeness of LSS-LRTA*. □

6. Comparison With Off-line Techniques

In the previous sections, we explored modifications to the LSS-LRTA* algorithm to improve its ability to achieve goals quickly. LSS-LRTA*'s performance can be improved by applying a heuristic correction and either using a single-step movement policy or using dynamically-sized lookahead searches. In this section we evaluate the performance of these algorithms against standard offline techniques. We have included three extra domains for this final comparison, the 15-puzzle, grid pathfinding, and a novel domain that we call the *traffic* domain. For the 15-puzzle, we used the 94 instances from Korf's 100 instances (Korf, 1985) that our implementation of A* was able to solve using a 6GB memory limit. For grid pathfinding, we ran on the orz100d grid map from the video game Dragon Age: Origins (Sturtevant, 2012). This map, shown in Figure 12, includes a mix of open space and maze-like areas with narrow corridors. We used the 25 start and end locations with the longest optimal path lengths in the scenarios from Sturtevant (2012). For completeness, results for the best performing algorithms on a random selection of 10 additional maps are presented in Appendix A.

Figure 12: Grid path-finding on a video game map.

6.1 The Traffic Domain

The traffic domain is a new domain inspired in part by video games such as Frogger[2]. The goal in the traffic domain is to navigate through a grid to a given goal location while avoiding obstacles, many of which are in motion. Each state includes the x, y location of the agent and the x, y location of each obstacle (In our implementation, the locations of obstacles were not stored in the state; instead, we store each state's current time, and obstacle locations were computed based on their initial location, velocity, and the time). Time is divided into discrete intervals called ticks, and the agent can move in one of the four cardinal directions or remain still during each tick. Obstacles each have both horizontal and vertical velocities that are either -1, 0, or 1 cell per-tick in the respective direction. Obstacle locations are known to the agent at any time in the future. When an obstacle hits the edge of the grid it "bounces" off, reversing its velocity in the direction of the hit. Obstacles simply pass through each other. The search space is directed, because time is ticking forward and, once the obstacles move, the agent cannot perform an action to move the obstacles back to their previous locations. This domain is especially well-suited for real-time techniques as the agent must have an action ready to execute at each tick when the world transitions and the obstacles move.

To eliminate dead end states (real-time algorithms are incomplete in the presence of dead ends), when the result of executing an action by the agent is an intersection with an obstacle, the agent is teleported back to the start state location (at time t=0). This is especially important for offline algorithms that expect to begin execution from the initial

2. This domain is similar to the 2011 ICAPS International Probabilistic Planning Competition domain called 'crossing traffic' which was constructed as an MDP and also a POMDP. Our version of the domain is deterministic and fully observable.

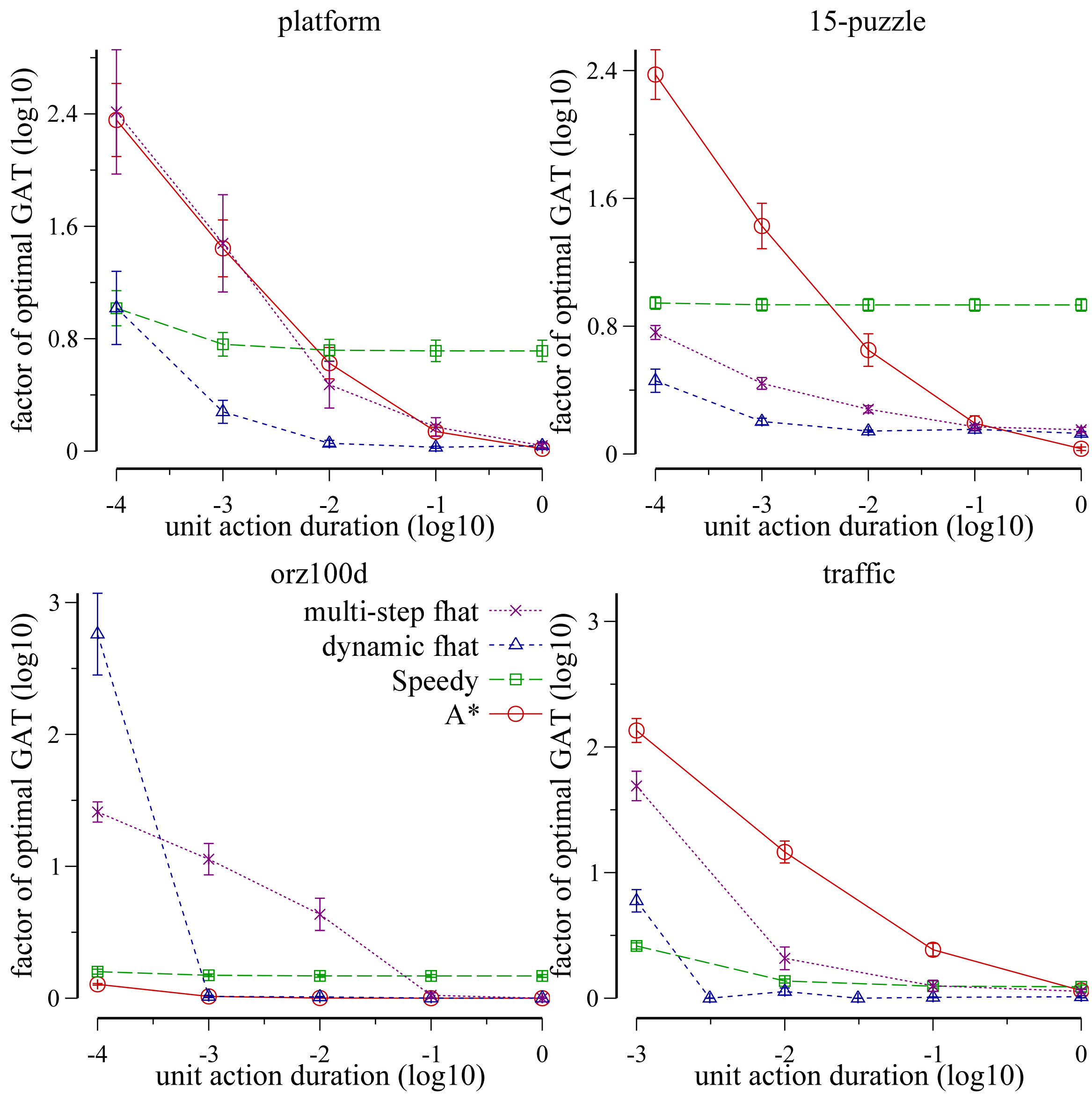


Figure 13: Comparison with off-line techniques.

state despite the passage of time while planning. For our experiments, we generated 25 random solvable instances consisting of 100x100 grids with 5,000 obstacles placed randomly with random velocities. The start location was in the upper-left corner of the grid and the goal location was in the lower-right corner. The average solution length was 211.24 moves. Videos showing the traffic domain are available on-line (Kiesel et al., 2015b) and discussed in Appendix B.

6.2 Results

By allowing planning and execution to take place simultaneously, it should be possible to improve over offline techniques that delay execution until planning is finished. To assess this, we compared the best-performing variants of LSS-LRTA* with A* and an algorithm called Speedy (Thayer & Ruml, 2009). Speedy is a best-first greedy search on $d(n)$, the estimated number of actions remaining to the goal. It tends to find poor plans very quickly, providing an informative contrast with A*. Figure 13 shows the results of this comparison, again with the $\log_{10}$ factor of optimal goal achievement time on the y axis and the unit action duration, on a $\log_{10}$ scale, on the x axis.

$\hat{f}$-based search with a dynamic lookahead gave the best goal achievement times on the platform, 15-puzzle, and traffic domains. Speedy was a strong performer in both the grid pathfinding domain and the traffic domain. This is not surprising as these domains are both based on grid navigation. In the traffic domain Speedy is able to quickly find a collision free path (avoiding additional cost overhead). On grid pathfinding A* actually had the lowest goal achievement times for all unit action durations, and $\hat{f}$ with a dynamic lookahead was about tied with A* for all except the fastest unit action duration. These results on grid pathfinding are consistent with the results presented by Hernández et al. (2012), where their best performer was about as good as A*. (In their study, the best performer was TBA*, which we don't compare against as it does not work for directed graphs.) This is likely because A* can solve these grid pathfinding problems very quickly, thus it has short planning times, and still finds optimal solutions.

Even though A* performs well, it is not applicable when a real-time constraint is present and an action needs to be returned within a bound. A*-based real-time algorithms give similar results with an infinite lookahead, although some would waste time learning.

7. Related Work

There is a large body of work relating to real-time search. In this section we review some of this work and discuss its relation with the techniques that we have presented in the previous sections.

7.1 Pruning Dead States

f-LRTA* (Sturtevant & Bulitko, 2011) is an extension of LSS-LRTA* and RIBS (Sturtevant, Bulitko, & Björnsson, 2010), combining both h-cost learning and g-cost learning. The g-cost learning enables the algorithm to label states as dead-ends or redundant. Determining these types of states using the basic algorithm relies on an underlying undirected graph. This arises from the requirement to compute the cost from successor to parent. In an undirected graph this is simply the reverse operator, but in the case of a directed graph, this would require a call to either a heuristic or a call to an additional search to determine the cost of that edge. In consultation with Sturtevant, we created a small modification to include reverse edge costs where they were easily computable. However, in practice this did not perform well in our directed graph domains. We conclude that further work is needed to adapt the ideas behind f-LRTA* to directed graphs.

Sharon, Sturtevant, and Felner (2013) introduce a technique for pruning dead states in real-time agent-centered search. This work detects two types of dead states: expendable and swamp. These are determined by considering reachability and shortest paths in the local neighborhood of a state. While dead state pruning has been shown to lead to speed-ups in 8-way grid pathfinding, it is only applicable in certain domains. Let us first consider the undirected domains considered in this paper. In the sliding tile puzzle, for example, expendable and swamp states do not exist locally. No neighbors of a state s can reach another neighbor without passing through s when only considering a local neighborhood. Without reachability, no shortest paths can exist, so no locally expendable or swamp states would be pruned. We also note that in a grid navigation problem that only considers movement in the four cardinal directions, no pruning could occur either for the same reason.

In a directed graph, the predecessors of s must also be considered in the local neighborhood. In this case, an expendable state would be a state s whose predecessors can reach all of s's successors without traversing through s. Similarly, a swamp state s would be a state whose predecessors have shortest paths to all successors of s that do not pass through s. In the traffic domain, for example, the state contains time, so all predecessors of a state s with time t, would have time $t-1$ and its successors all have a time of $t+1$. There is no way to traverse from a state $t-1$ to a state with time $t+1$ without traversing through a state with time t. As state s is the only state in the local neighborhood with a time t, no paths can exist between predecessors and successors of s that do not pass through s. No expendable or swamp states would be pruned in this domain.

7.2 Minimizing Search Effort

The GAT model assumes that search can occur during action execution, which is appropriate for situations in which separate processor cores are responsible for planning versus managing execution. When processor resources are scarce and shared among many tasks, one may want to minimize search effort even during execution. Bulitko, Luštrek, Schaeffer, Björnsson, and Sigmundarson discuss methods for dynamically adjusting real-time search lookahead in order to minimize search effort while still selecting good actions. In contrast, as we discussed in section 5.3.2, dynamic $\hat{f}$ attempts to use all available execution time to perform as much search as possible.

7.3 Time-bounded A*

Time-bounded A* (TBA*, Björnsson, Bulitko, & Sturtevant, 2009) is a non-agent-centered real-time search algorithm. Instead of performing a bounded amount of lookahead search from the agent's current state, TBA* maintains a single A* search from the agent's initial starting state to the goal state. During each iteration, a fixed number of expansions are done on this single search and the agent attempts to move toward the most promising node on the search frontier. Since the agent may have already moved away from the initial state during previous iterations, and because A* vacillates between many different paths, the agent's current state may not be along the current best path. If this occurs, the agent backtracks toward the initial state until it is on the current best path. In their experiments, Björnsson et al. showed that, on grid pathfinding benchmarks, TBA* requires fewer iterations to find the same quality paths as other real-time algorithms such as LRTA*.

As mentioned briefly above, Hernández et al. (2012) found that TBA* was the best technique for optimizing goal achievement time on grid pathfinding problems in fully-known grids. They state that its performance was about the same as that of A*. In our experiments, we do not compare with TBA*, as we consider domains that form directed graphs, and TBA* only works on undirected graphs, due to the agent's need to backtrack. We suspect, however, that our dynamic lookahead $\hat{f}$ technique would be quite competitive with TBA*, as it also matched the performance of A* on grid pathfinding problems and it was able to greatly outperform A* on all of the other domains.

7.4 Avoiding Depressions in Real-time Heuristic Search

Real-time search algorithms can become temporarily stuck in a heuristic local minimum for an extended period of search and execution time (Sturtevant & Bulitko, 2014). The agent will typically wander around in a heuristic minimum until it learns that the heuristic in that area is inaccurate and corrects it. This behavior results in long solutions and can be aesthetically undesirable.

daLSS-LRTA* and daRTAA* (Hernández & Baier, 2012) attempt to actively avoid and escape heuristic depressions. Instead of selecting the node with the lowest f value, daLSS-LRTA* selects the node along the frontier whose heuristic value has changed the least. daRTAA* is similar but uses a simpler learning phase borrowed from Real-Time Adaptive A* (Koenig & Likhachev, 2006). RTAA* and daRTAA* update the entire interior of the local search with the f value of the node on the open list with the best f value.

We implemented both daLSS-LRTA* and daRTAA* and compared them to standard LSS-LRTA* as well as the multi-step and dynamic lookahead variants of LSS-LRTA* using $\hat{f}$. The results of the comparison are shown in Figure 14. Our results on the grid pathfinding problem (orz100d) agree with results from Hernández and Baier (2012) and show that using depression avoidance techniques can help improve performance (for example, compare LSS-LRTA* and daLSS-LRTA*). Also, daLSS-LRTA* and daRTAA* outperform the standard multi-step variant of LSS-LRTA* using $\hat{f}$. Dynamic lookahead $\hat{f}$ clearly gives the best performance for all but the fastest unit action duration of 0.0001. In the platform and traffic domains, however, daLSS-LRTA* appears slightly worse than LSS-LRTA*, and in the 15-puzzle depression avoidance appears to have little effect. Overall, we found dynamic $\hat{f}$ to dominate the other techniques.

Similar to Figure 10, we can see a spike at 0.001 unit action duration in the platform domain. This can be attributed to a jump in the number of instances that were solved by all algorithms between 0.0001 and 0.001. The larger set contains more difficult instances and increases the factor of optimal GAT.

It may be interesting future work to combine depression avoidance with dynamic lookahead, especially for grid pathfinding domains.

7.5 Weighted Real-time Heuristic Search

Weighted A* (wA*, Pohl, 1970) is a popular heuristic search algorithm that proceeds like A*, but orders nodes on the open list using $f'(n) = g(n) + w \cdot h(n)$, for some $w \geq 1$. As w increases, the search becomes more greedy and can often find solutions faster than A*. While these solutions may not be optimal, they are guaranteed to be within a factor w of the

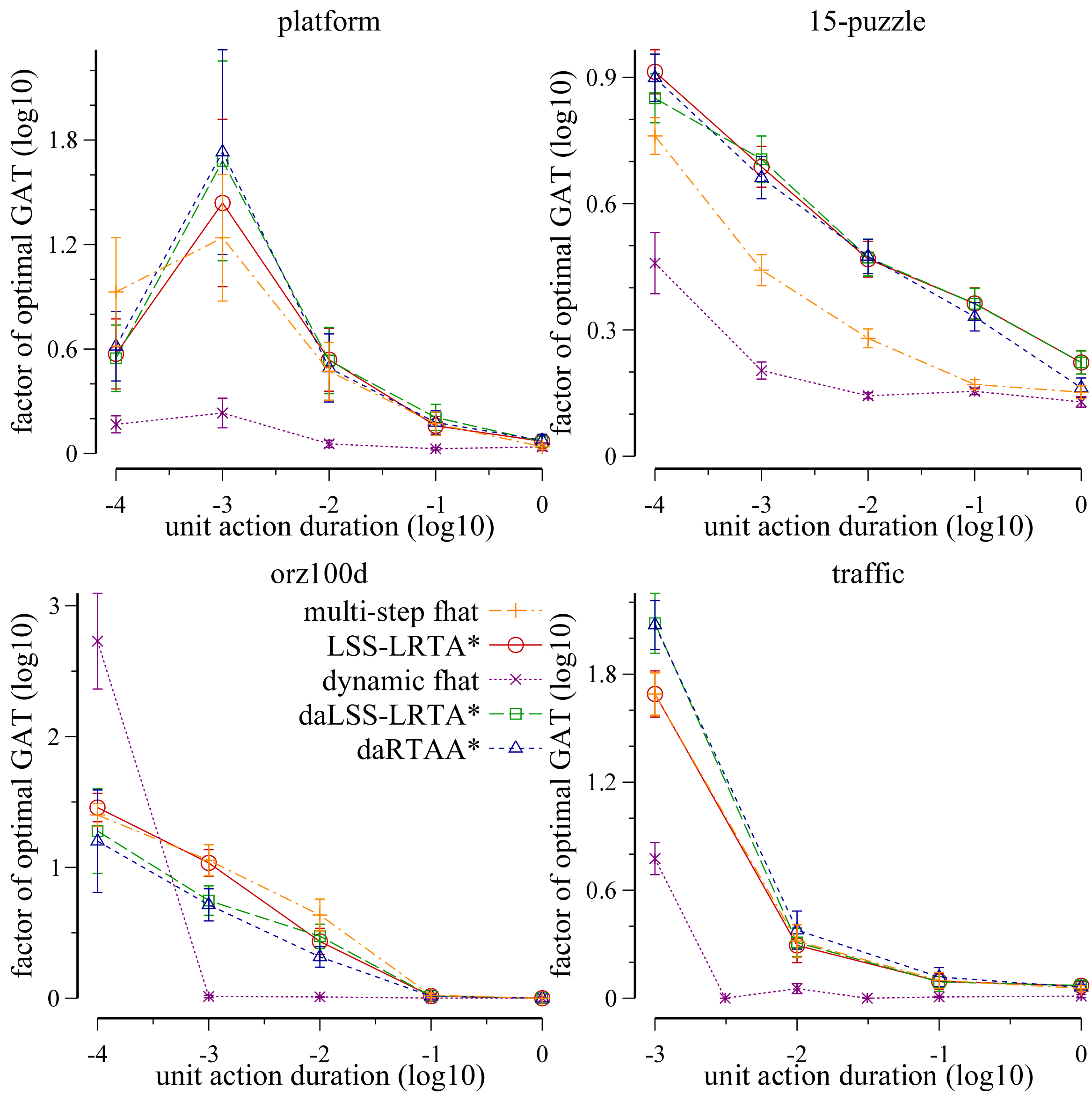


Figure 14: Depression avoidance real-time heuristic search.

optimal solution cost. Rivera, Baier, and Hernández (2012) recently showed a variant of wA* for real-time search called wLSS-LRTA*. One obvious way to implement a real-time variant of wA* would be to simply multiply the heuristic value by $w \geq 1$ during a lookahead search in LSS-LRTA*, however, wLSS-LRTA* does not do this. Instead, wLSS-LRTA* multiplies the edge weights by w during the learning phase of LSS-LRTA*. The update rule becomes: $h(n) \leftarrow min_{m \in open} w \cdot g(n,m) + h(m)$, where $g(n,m)$ is the cost between the node n being updated and a node m from the open list of the lookahead search.

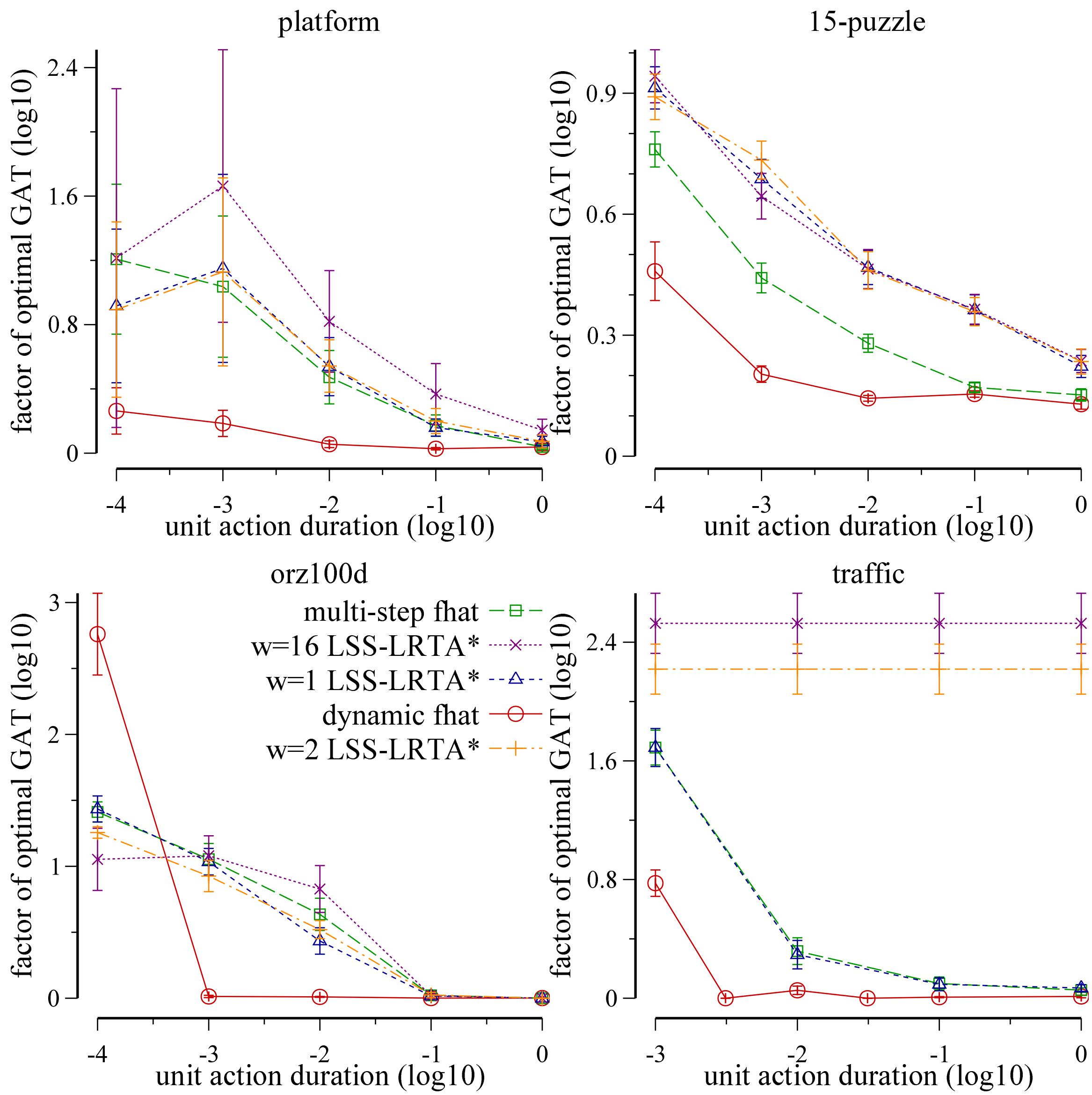


Figure 15: Weighted real-time heuristic search.

Rivera et al. (2012) show that using an increased weight in wLSS-LRTA* can lead to lower-cost solutions. They point out that, because admissible heuristics are lower bounds, inflating the heuristic by a factor w may make the heuristic more accurate. This is the same reasoning behind our $\hat{f}$ technique. The difference is that wLSS-LRTA* uses a weight to inflate the g portion of the updated heuristic whereas the $\hat{f}$ technique adds a correction based on the h portion of the updated heuristic. We would argue that the $\hat{f}$ approach makes more sense because the error causing the heuristic to underestimate does not come from the perfectly-known g portion of the update, but from the estimated h portion.

We implemented wLSS-LRTA* and compared it to the standard multi-step and dynamic lookahead variants of LSS-LRTA* using $\hat{f}$. The results of the comparison are shown in Figure 15. Our results on the grid pathfinding problem (orz100d) tend to agree with those of Rivera et al. (2012): using larger weights for wLSS-LRTA* can increase performance. This trend seems to depend, however, on the unit action duration; it is more noticeable when actions are fast. Also, for grid pathfinding, wLSS-LRTA* outperforms the standard multi-step variant of LSS-LRTA* using $\hat{f}$, but the dynamic lookahead $\hat{f}$ clearly gives the best performance for all but the fastest unit action duration of 0.0001. However, in the platform, 15-puzzle, and traffic domains we found almost the opposite to be true! Dynamic $\hat{f}$ is still the best—it nearly dominates all other techniques. But the fairest comparison is LSS-LRTA* using $\hat{f}$ (with statically sized lookahead) which provides better performance than wLSS-LRTA*, dominating wLSS-LRTA* with a weight greater than 1 on the 15-puzzle and traffic problems, and increasing the weight in wLSS-LRTA* either has no effect or it makes the performance worse. For the traffic domain with weights greater than 1, wLSS-LRTA* was unable to solve any problems with a lookahead of 1, and all greater lookahead values tried (including a lookahead of 2) were too slow to meet the real-time deadline for a unit action duration of 0.0001, thus there is no data point for x=0.0001 for either of these algorithms. Based on these results, we conclude that using the parameter-free $\hat{f}$ technique to explicitly attempt to account for heuristic error is the recommended approach over weighting the edge costs during learning by a user-specified parameter.

7.6 FRIT

Follow and Reconnect with the Ideal Tree, or FRIT (Rivera, Illanes, Baier, & Hernández, 2013), takes another approach to dealing with heuristic minima in real-time search. Rather than applying heuristic learning and updating to escape a local minimum, FRIT instead tries to follow the *ideal tree* of the state space. An ideal tree represents a family of paths that connect states of the search space with the goal state. It also can be thought of as being implicitly represented by a heuristic.

The ideal tree is explored by following the heuristic greedily as if all operators were applicable in all states, until the heuristic suggests an operator that is inapplicable. A simple example is following the Manhattan distance heuristic in a grid pathfinding domain until encountering an obstacle. When an inapplicable operator is suggested, the tree becomes disconnected and the agent must reconnect with the tree. This is done by performing a local search around the agent's current state until a state believed to be in the Ideal Tree is found. The agent then moves to that state and continues on. The resulting behavior in grid pathfinding domains can appear very similar to wall following.

Some modifications are required to make FRIT into a real-time algorithm. The local search to find a state in the Ideal Tree is bounded only by the size of the state space, rather than a time bound or an expansion limit. The authors suggest a few techniques for bounding the local search but in our experiments, we allowed FRIT to be thought of as offline and allowed it as much time as needed when looking to reconnect with the Ideal Tree. We also used breadth first search as the local search algorithm.

We implemented FRIT and compared it against standard LSS-LRTA* and the multi-step and dynamic versions of LSS-LRTA* using $\hat{f}$. The results are shown in Figure 16. We

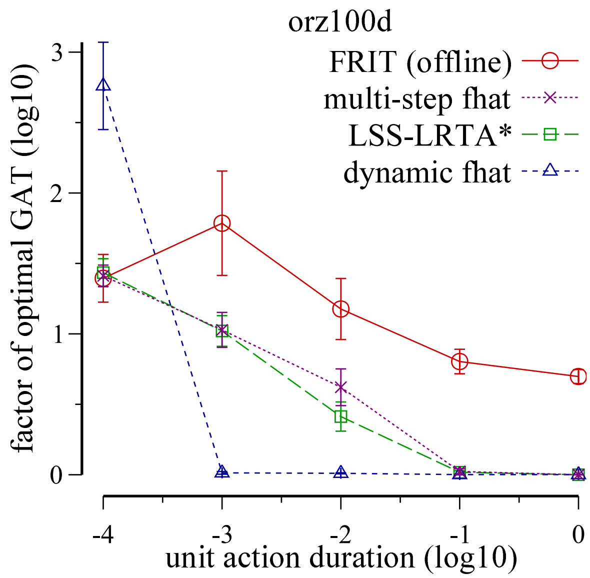


Figure 16: Comparison with offline FRIT using breadth first search.

only present results for the grid pathfinding problem (orz100d). FRIT was only able to solve a few of the easier instances in the platform and traffic domains within the five minute timeout. As for the sliding tile puzzle domain, it was unclear how to adapt this algorithm for this domain. A naive approach results in a branching factor of 44 and poor results.

On grid pathfinding, even by treating FRIT as an offline algorithm and not penalizing it for search time in its final goal achievement time, it performs worse than the three variants of LSS-LRTA* presented. The exception is when the unit action duration is very small, at which point FRIT is competitive with the other algorithms (ignoring its search time).

7.7 FALCONS

Furcy and Koenig (2000) present two modifications to LRTA* to speed up its convergence time. They noticed that by breaking ties in favor of successors with smaller f-values LRTA* would converge more quickly. They also point out that if you also use this tie-breaking criteria to select which successor to move to, convergence occurs even faster. These two modifications yield two new algorithms: Tie Breaking LRTA* (TB-LRA*) and FAst Learning and CONverging Search (FALCONS).

In Figure 17 we compare against the original LRTA*, TB-LRTA* and FALCONS. In all domains these three algorithms perform worse than the newer LSS-LRTA* and modified versions of LSS-LRTA*.

7.8 RTA*

Korf (1990) not only proposed LRTA* in his seminal paper but also another algorithm simply called Real Time A* (RTA*). RTA*, unlike its counterpart LRTA*, is focused on solving the problem of getting from the start state to the goal state only once. LRTA* is

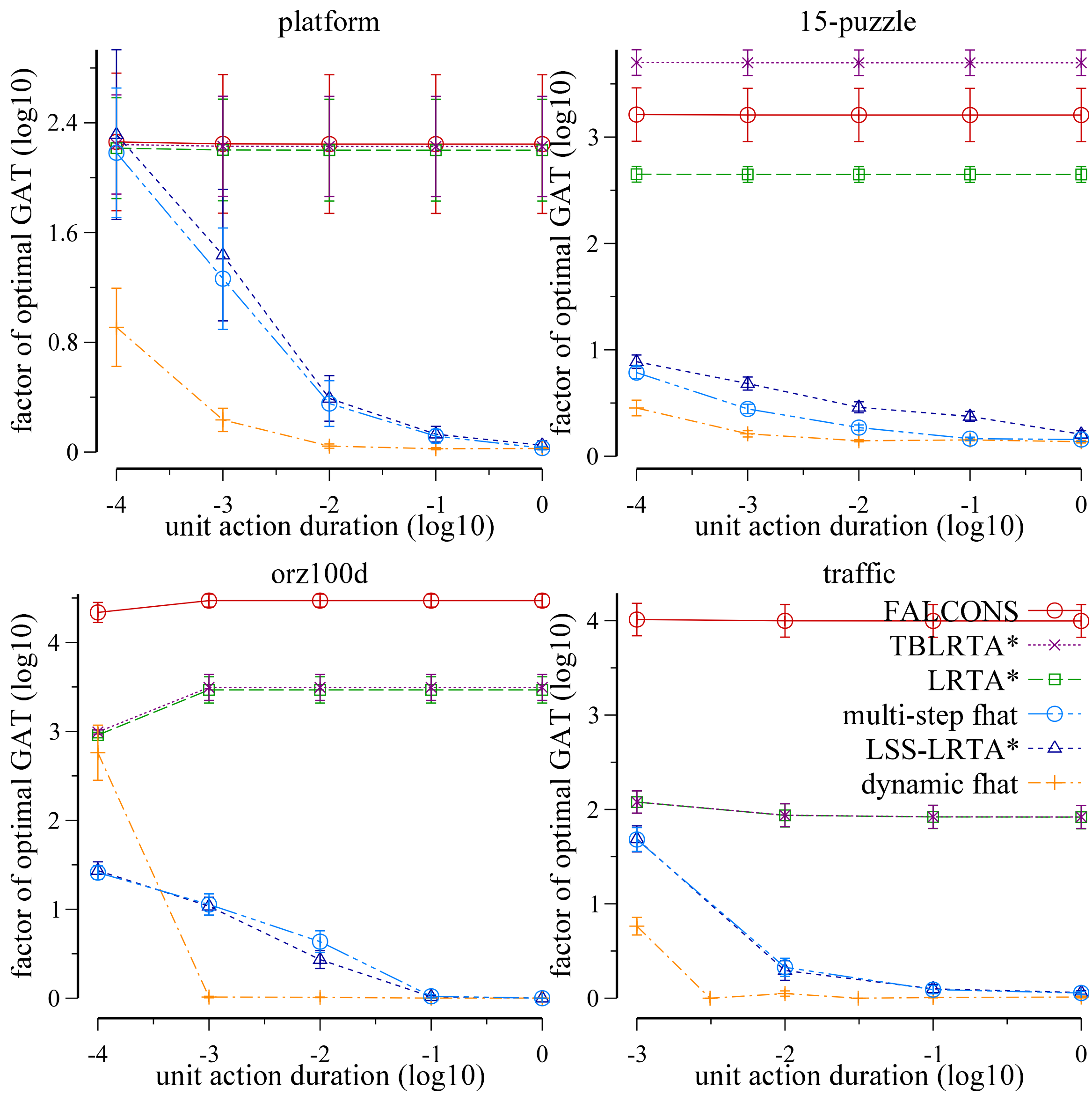


Figure 17: Comparison with LRTA*, TBLRTA* and FALCONS.

proven to converge to optimal heuristic values over successive trials. RTA*'s learning policy does not guarantee a convergence of heuristic values but in practice can find solutions more quickly than LRTA*.

In Figure 18 we compare against RTA*. We include LRTA* in these plots as well to show the tradeoff between convergence and initial goal achievement time RTA* makes. RTA*'s lookahead is based on a bounded depth first search, so its run time is difficult to predict. For these experiments we ran RTA* with lookahead depths of {1, 5, 10, 20, 50, 100, 200, 400, 800, 1000, 1500, 2000, 3000, 4000, 8000, 10000, 16000} and chose the largest depth where

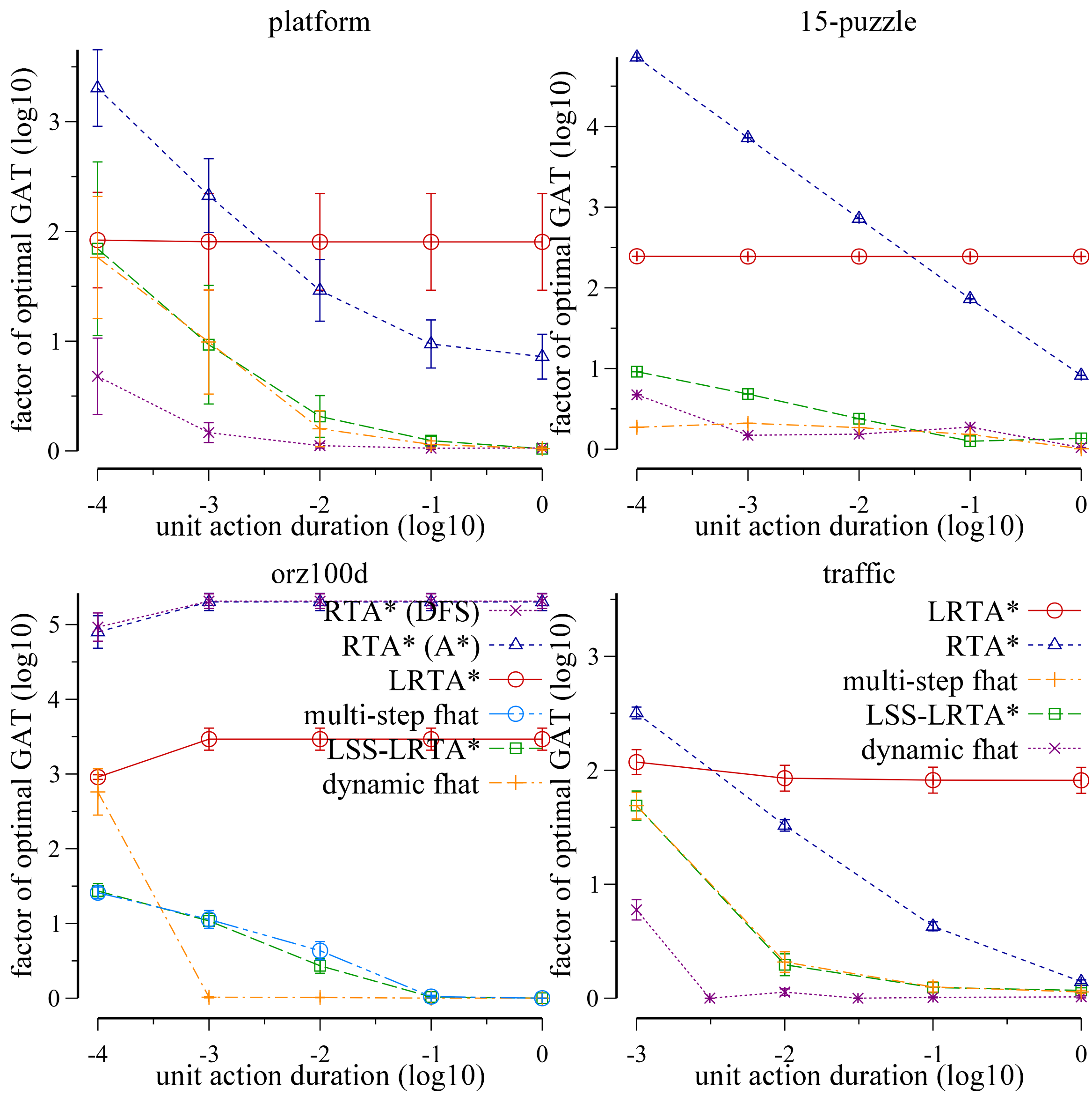


Figure 18: Comparison with RTA*.

instances were solved within the timeout. It is interesting to note that for the 15-puzzle a depth of 1500 was able to be used, while in platform and traffic only a lookahead of 10 could be used. We attribute this to platform and traffic being very graphy domains, while tiles has fewer cycles. The extreme case is grid pathfinding on the orz100d map where only a maximum lookahead of 5 could solve instances within the timeout. For supplementary comparison, we also provide a line for RTA* using an A* lookahead instead of depth first search in the grid pathfinding domain. An expansion limit of 4000 was the largest lookahead size that solved all the instances. In the four plots in Figure 18, we can see that the newer algorithms outperform RTA* in all domains.

7.9 Bugsy

Bugsy (Burns et al., 2013b) is not a real-time search, but it is an off-line algorithm that explicitly attempts to optimize a utility function given as a linear combination of search time and solution cost. If solution cost is specified in units of time, then Bugsy can explicitly attempt to minimize goal achievement time by appropriately weighting search and execution times so that they are given in the same units. As the only off-line algorithm that can optimize our goal achievement time objective, it is interesting to see how Bugsy compares to real-time algorithms. Since it performs a global search, it may be better able to optimize cost, but it is inherently less efficient, as it cannot plan and execute in parallel.

Figure 19 shows the results. Bugsy tended to have the lowest goal achievement times in all domains except for the traffic domain, where the dynamic lookahead $\hat{f}$ method nearly dominated all other approaches. However, in all domains except the 15-puzzle, the advantage of Bugsy was small. We conclude that, if a full solution can be found upfront, then off-line methods like Bugsy can often given the best results. When an agent must respect real-time constraints, however, the dynamic lookahead $\hat{f}$ technique is the algorithm of choice.

It may be possible to create a new algorithm that incorporates the ideas of Bugsy into a real-time search. Bugsy proceeds like A*, but it orders its open list on a utility estimate $u(n) = w_f \cdot f(n) + w_t \cdot \mathit{time}(n)$, where $\mathit{time}(n)$ is an estimate of the time the search will take to reach the best solution beneath node n (for details, see Burns et al., 2013b). The difficulty in incorporating the ideas of Bugsy into real-time search is that Bugsy's utility estimate assumes that none of the planning time, $\mathit{time}(n)$, will occur in parallel with the execution time $f(n)$ (recall that cost is in units of time when optimizing goal achievement time). In real-time search, this is not true. If solution cost is in units of time and all planning happens during execution, then optimizing cost seems appropriate.

8. Conclusion

In this paper we considered real-time search in the context of minimizing goal achievement time when concurrent planning and execution is possible. When optimizing goal achievement time, it is important to consider the tradeoff between searching and executing. We presented three modifications to LSS-LRTA*: 1) taking single steps instead of moving all of the way to the fringe of the lookahead search, 2) use multiple steps, but dynamically increase the lookahead size to match the execution time of each trajectory, and 3) using $\hat{f}$ to correct the bias in the heuristic. We then evaluated these techniques against plain LSS-LRTA*, A*, Speedy, daRTAA*, daLSS-LRTA*, wLSS-LRTA*, FRIT, TBLRTA*, FALCONS, LRTA*, RTA*, and Bugsy on four domains. In addition to the 15-puzzle and grid pathfinding domains, which are classic heuristic search benchmarks, we used two video-game-inspired domains: the platform domain and a new traffic domain.

We showed that committing to single actions at a time can give better performance than using the traditional multiple action approach. Then we demonstrated that using the multiple action technique can be even better than performing single steps if the amount of

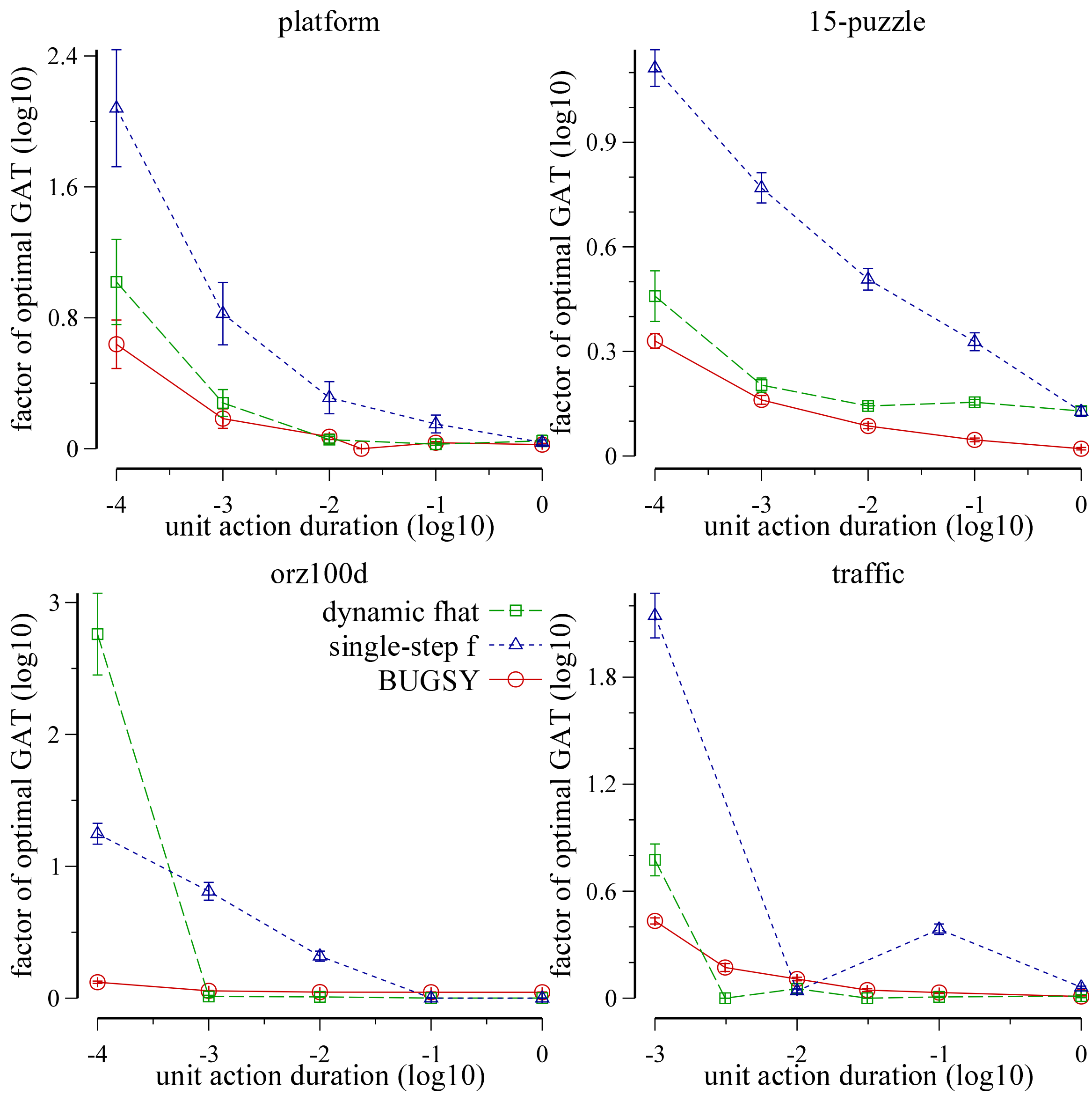


Figure 19: Comparison with Bugsy.

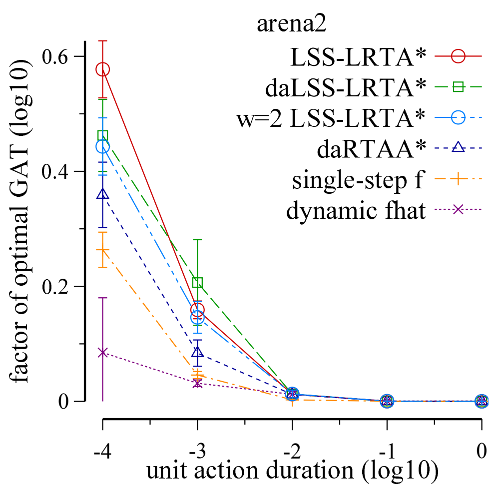
lookahead search is dynamically adjusted to use all of the time available for the execution of the currently-executing multi-step trajectory. We pointed out some possible reasons why using an A*-based lookahead search may lead to poor performance and showed how $\hat{f}$ could be used to fix these issues. Overall, the combination of a dynamically sized lookahead and $\hat{f}$ gave the best performance when compared to previous real-time techniques. We hope that this work will spur further research into applying real-time heuristic search to dynamic domains.

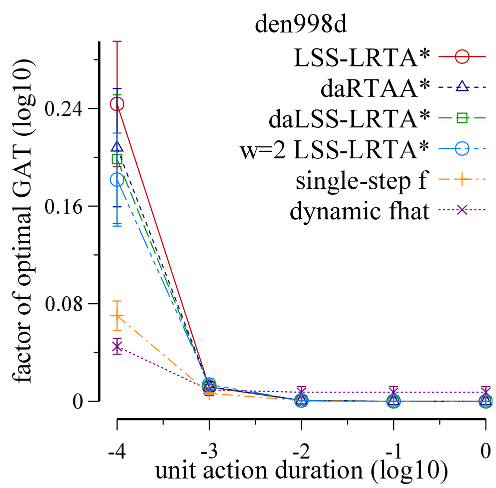
9. Acknowledgments

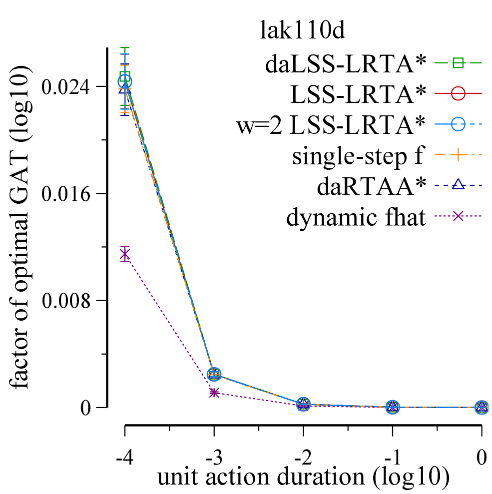
This work was supported in part by the NSF (grants 0812141 and 1150068), the DARPA CSSG program (grant D11AP00242), and the University of New Hampshire Dissertation Year Fellowship. A preliminary version of this work was published by Burns, Kiesel, and Ruml (2013a).

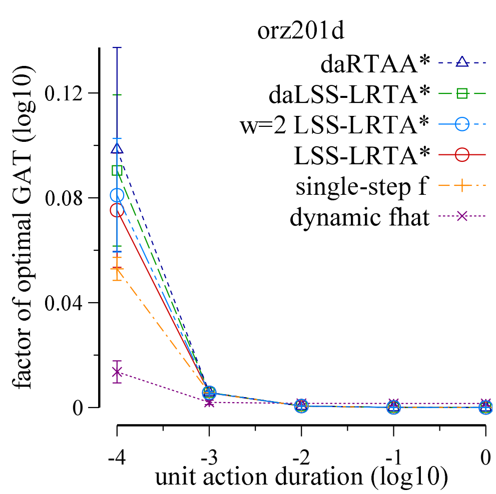
Appendix A. Grid Pathfinding Results on Additional Maps

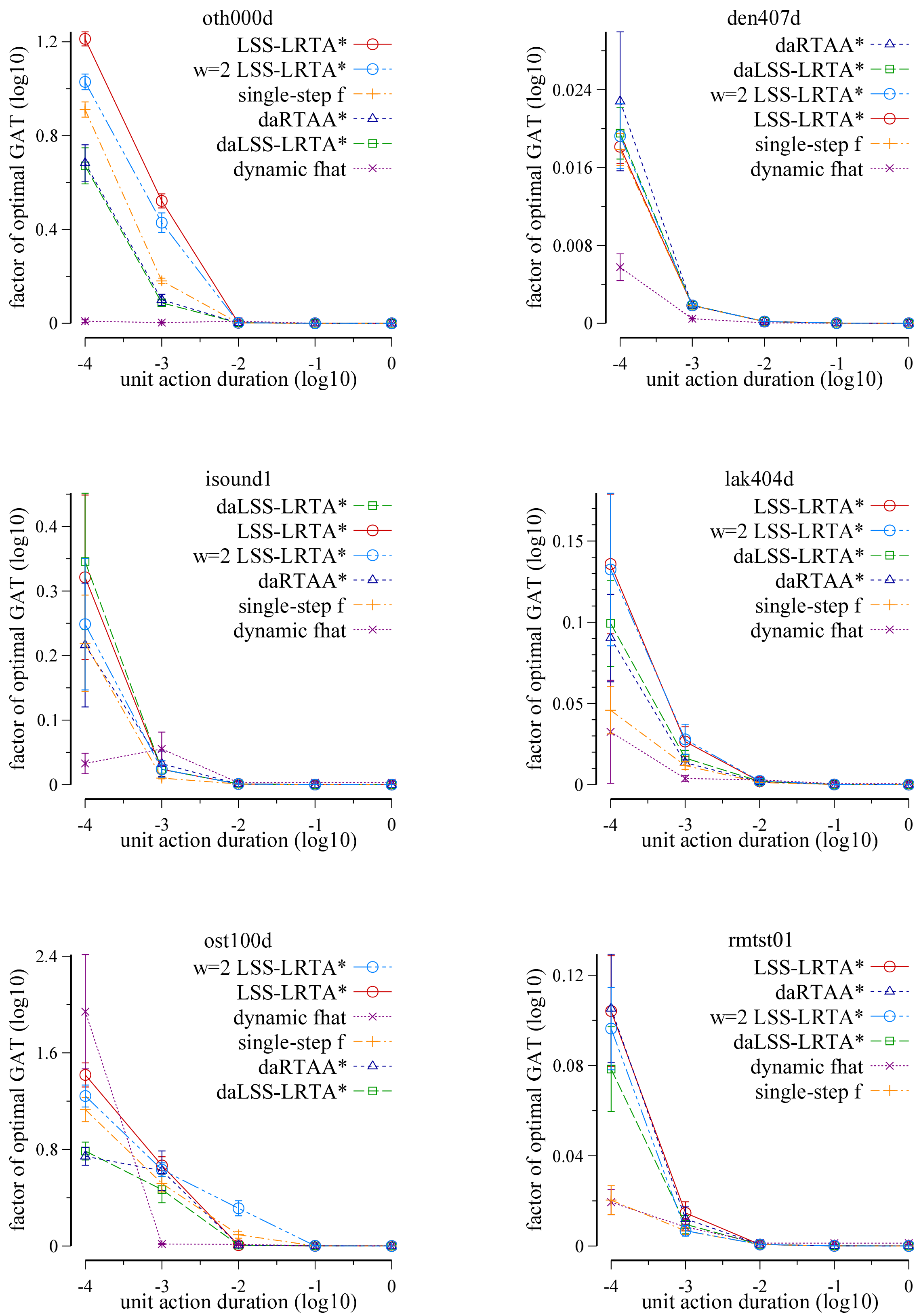
The following are a random sample of 10 maps from Sturtevant's repository with the top performing algorithms plotted. These plots are similar to those included in the paper with the $\log_{10}$ factor of optimal goal achievement time on the y-axis and the $\log_{10}$ unit action duration on the x-axis.










oth000d
LSS-LRTA*
w=2 LSS-LRTA*
single-step f
daRTAA*
daLSS-LRTA*
dynamic fhat
den407d
daRTAA*
daLSS-LRTA*
w=2 LSS-LRTA*
LSS-LRTA*
single-step f
dynamic fhat
isound1
daLSS-LRTA*
LSS-LRTA*
w=2 LSS-LRTA*
daRTAA*
single-step f
dynamic fhat
lak404d
LSS-LRTA*
w=2 LSS-LRTA*
daLSS-LRTA*
daRTAA*
single-step f
dynamic fhat
ost100d
w=2 LSS-LRTA*
LSS-LRTA*
dynamic fhat
single-step f
daRTAA*
daLSS-LRTA*
rmtst01
LSS-LRTA*
daRTAA*
w=2 LSS-LRTA*
daLSS-LRTA*
dynamic fhat
single-step f
factor of optimal GAT (log10)
unit action duration (log10)

Appendix B. Video Descriptions

Here we describe the videos that are available on-line (Kiesel et al., 2015b).

B.1 Platform Videos

The videos numbered 1-10 in the playlist show algorithms solving the Platform domain.

B.1.1 Random Instance

Videos 1-7 provide an example of a random instance of the Platform of the platform domain being solved by various algorithm configurations.

Video 1 is LSS-LRTA* with a 1,000 node lookahead using a multi-step policy. In this video you can see the algorithm get stuck in local heuristic minima and actively trying to update its heuristic estimates for those states in the minimum.

Video 2 is LSS-LRTA* with a 1,000 node lookahead using a single-step policy. In this video you can see the algorithm traverse the smaller sized local minima much more quickly but still become stuck for a short while in the larger local minimum at around 9 seconds.

Video 3 is LSS-LRTA* using a dynamically sized lookahead. Initially, it gets stuck inside of a local minimum, similar to LSS-LRTA* with a static lookahead, but soon is able to escape through learning and increasing lookahead sizes.

Video 4 is LSS-LRTA* using a dynamically sized lookahead and heuristic correction. It is very quickly able to escape the various heuristic minima on the way to the goal.

Video 5 is Speedy. In this video you can see that the algorithm is able to find a solution very quickly as it starts moving almost instantly. The next 2 minutes of the video are spent executing the highly suboptimal solution.

Video 6 is A*. We do not visualize the planning time for complete solution to be found before A* begins moving (roughly 1 minute and 15 seconds, see video 7). Its solution is optimal and very quickly gets the agent from its initial position to its goal.

Video 7 is a comparison of LSS-LRTA* with a 1,000 node lookahead, LSS-LRTA* with a dynamically sized lookahead and heuristic correction, A* and Speedy. In this video, planning time is visualized.

B.1.2 Ladder Instance

Videos 8-10 demonstrate an extreme example of a heuristic minimum. The visibility graph heuristic assumes the agent is able to jump infinitely high, creating a large local minimum that the agent must learn its way out of.

Video 8 shows LSS-LRTA* with a 1,000 node lookahead and a multi-step policy struggling to climb the platform ladder.

Video 9 shows LSS-LRTA* with a 1,000 node lookahead and a single-step lookahead policy quickly climbing up the ladder but struggling at the end.

Video 10 shows an optimal example of how to climb the ladder using A*.

B.2 Traffic Videos

The remaining videos illustrate the traffic domain, a highly dynamic domain with many moving obstacles that the agent must avoid. These videos are provided only as a visualization of the domain that the algorithms are trying to solve.

Video 11 shows an example of an optimal solution found by A* that never collides with an obstacle.

Video 12 shows LSS-LRTA* with a 1,000 node lookahead solving the same problem. In this video around 20 seconds, the agent wanders into a situation where a collision occurs and is transported back to the start state.

References

Björnsson, Y., Bulitko, V., & Sturtevant, N. (2009). TBA*: time-bounded A*. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 431–436.

Bulitko, V., Luštrek, M., Schaeffer, J., Björnsson, Y., & Sigmundarson, S. (2008). Dynamic control in real-time heuristic search. *Journal of Artificial Intelligence Research*, *32*, 419–452.

Burns, E., Kiesel, S., & Ruml, W. (2013a). Experimental real-time heuristic search results in a video game. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SoCS)*.

Burns, E., Ruml, W., & Do, M. B. (2013b). Heuristic search when time matters. *Journal of Artificial Intelligence Research (JAIR)*, *47*, 697–740.

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.

Hernández, C., & Baier, J. (2012). Avoiding and escaping depressions in real-time heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, *43*, 523–570.

Hernández, C., Baier, J., Uras, T., & Koenig, S. (2012). Time-bounded adaptive A*. In *Proceedings of the Eleventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

Kiesel, S., Burns, E., & Ruml, W. (2015a). Research code for heuristic search. https://github.com/eaburns/search. Accessed September 2, 2015.

Kiesel, S., Burns, E., & Ruml, W. (2015b). Videos for 'achieving goals quickly using real-time search'. http://bit.ly/1bW3Ey8. Accessed September 2, 2015.

Koenig, S., & Likhachev, M. (2002). D* lite. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI)*, pp. 476–483.

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

Koenig, S., & Sun, X. (2008). Comparing real-time and incremental heuristic search for real-time situated agents. In *Journal of Autonomous Agents and Multi-Agent Systems*, pp. 18(3):313–341.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97–109.

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

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.

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, *1*, 193–204.

Rivera, N., Illanes, L., Baier, J. A., & Hernández, C. (2013). Reconnecting with the ideal tree: An alternative to heuristic learning in real-time search. In *Proceedings of the Sixth International Symposium on Combinatorial Search (SoCS)*.

Rivera, N., Baier, J. A., & Hernández, C. (2012). Weighted real-time heuristic search. In *Proceedings of the Twelfth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

Sharon, G., Sturtevant, N. R., & Felner, A. (2013). Online detection of dead states in real-time agent-centered search. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search (SoCS)*.

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

Sturtevant, N., & Bulitko, V. (2014). Reaching the goal in real-time heuristic search: Scrubbing behavior is unavoidable. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS)*.

Sturtevant, N. R., & Bulitko, V. (2011). Learning where you are going and from whence you came: h-and g-cost learning in real-time heuristic search. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 365–370.

Sturtevant, N. R., Bulitko, V., & Björnsson, Y. (2010). On learning in agent-centered search. In *Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 333–340. International Foundation for Autonomous Agents and Multiagent Systems.

Thayer, J. T., Dionne, A., & Ruml, W. (2011). Learning inadmissible heuristics during search. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS)*.

Thayer, J. T., & Ruml, W. (2009). Using distance estimates in heuristic search. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS)*.