# Experimental Real-Time Heuristic Search Results in a Video Game

**Ethan Burns** and **Scott Kiesel** and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
*eaburns*, *skiesel*, *ruml* at *cs.unh.edu*

**Abstract**

In real-time domains such as video games, a planning algorithm has a strictly bounded time before it must return the next action for the agent to execute. We introduce a realistic video game benchmark domain that is useful for evaluating real-time heuristic search algorithms. Unlike previous benchmarks such as grid pathfinding and the sliding tile puzzle, this new domain includes dynamics and induces a directed graph. Using both the previous and new domains, we investigate several enhancements to a leading real-time search algorithm, LSS-LRTA*. We show experimentally that 1) it is not difficult to outperform A* when optimizing goal achievement time, 2) it is better to plan after each action than to commit to multiple actions or to use a dynamically sized lookahead, 3) A*-based lookahead can cause undesirable actions to be selected, and 4) on-line de-biasing of the heuristic can lead to improved performance. We hope that this new domain and results will stimulate further research on applying real-time search to dynamic real-time domains.

## Introduction

In many applications it is highly desirable for an agent to achieve an issued task as quickly as possible. Consider the common example of a navigation task in a video game. When a user selects a destination for a character to move to, they expect the character to arrive at its destination as soon as possible, thus suggesting that planning and execution happen in parallel. Real-time heuristic search has been developed to address this problem. Algorithms in this class perform short planning episodes (limited by a provided real-time bound), finding partial solutions and beginning execution before the goal has been found. While solution quality and overall search time are traditional heuristic search metrics, real-time heuristic search algorithms are compared by the length of the trajectories they execute.

In this paper, we introduce a new platformer video game domain where the agent must navigate through a series of platforms by moving and jumping around obstacles to arrive at a specified goal location. Using this new domain, as well as several classic search domains, we show modifications to Local Search Space Learning Real Time A* (LSS-LRTA*), a leading real time search algorithm, that significantly increase its performance with respect to goal achievement time.

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

First, we show that LSS-LRTA* (Koenig and Sun 2009), which executes multiple actions per planning episode, can greatly improve its goal achievement 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, superfluous actions can be selected for execution. Third, we show that applying on-line de-biasing of the heuristic used during search can significantly reduce the overall goal achievement time. Fourth, we present results showing 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.

Together, these modifications can be easily applied and deployed to vastly improve the overall performance of an agent being controlled by a real-time heuristic search algorithm. Videos illustrating our new domain and the discussed algorithms are provided at the following URL: http://bit.ly/YUFx00.

## 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 the real-time search algorithms most relevant for our study.

### LRTA*

Many real-time search algorithms are considered *agent-centered search* algorithms, because the agent performs a bounded amount of *lookahead* search rooted at its current state before acting. By bounding the size of each lookahead search, the agent can respect the real-time constraints and complete each search iteration by the time 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. LRTA* selects the next action to perform by using the edge costs and estimates of cost-to-goal, or heuristic values, for the states resulting from applying each of its current applicable actions. It chooses to execute the action with the lowest sum of edge cost and 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. The second way that it estimates cost is through learning. Each time LRTA* performs an action, it learns an updated heuristic value for its current state. If this state is encountered again, the learned estimate is used instead searching again. Korf proved that as long as the previous state's heuristic estimate is increased after each move by an amount bounded from below by some $\epsilon$ (Russell and Wefald 1991), 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.

## LSS-LRTA*

Local Search Space Learning Real-time A* (LSS-LRTA*, Koenig and Sun 2009) 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 is limited by the number of nodes that it can 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 competitive with a state-of-the-art incremental search, D*Lite (Koenig and 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. This allows the agent to perform many fewer searches before arriving at a goal. 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.

## Evaluating Real-time search Algorithms

Recently, Hernández et al. (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 do both search and execute in parallel. The game time model objective is to mimimize the number of time intervals required for the agent to move from its start state to a goal state. 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 a full solution has been found. Off-line algorithms often find lower-cost paths (for example, A* finds optimal paths), but they do not allow search and execution to take place during the same time interval; they always search for a full solution before any execution begins. Real-time algorithms tend to find higher-cost paths, but they can be more efficient by allowing search and execution to occur in parallel.

## Goal Achievement Time

The game time model is a good fit for domains where all actions are unit cost. However, in practice domains can have varying actions durations, for example video games can support 8-way navigation which allows diagonal movement at the cost of $\sqrt{2}$. In these situations it is more accurate to track performance based on a real-valued metric. Goal achievement time (GAT) is a novel generalization of the game time model discussed in the previous section. The game time model requires that time be divided into discrete, uniform time intervals while goal achievement time requires no such restriction. Agents optimizing goal achievement time, for example, can execute actions at any point in time and can execute actions of varying durations without syncing up to a predefined time interval.

When computing goal achievement time in the case of parallel planning and execution, it is important to differentiate between planning that occurs on its own and planning that occurs in parallel with execution. If one were to simply sum the total planning and execution times, then the time spent planning and executing in parallel would be double counted. All algorithms discussed in this paper, except for A* and Speedy (Thayer and Ruml 2009), perform their planning in parallel with execution except for the planning required to find the very first action. To compute goal achievement time for these algorithms, we simply add this short amount of initial planning time to the execution time. In the case of an offline algorithm such as A*, we add its entire planning time to the time required to execute the solution it finds.

## The Platform Game Benchmark

Unlike previous benchmarks such as grid pathfinding and the sliding tile puzzle, this new platform game benchmark domain includes dynamics and induces a directed graph. This domain is a path-finding domain of our own creation with dynamics based on a 2-dimensional platform-style video game (e.g. Super Mario Bros and Duke Nukem) called `mid` (http://code.google.com/p/mid-game). The left image of Figure 1 shows a screenshot from `mid`. The goal is for the knight to traverse a maze from its initial location, jumping from platform to platform, until it reaches the door (the goal). Short videos of each algorithm solving instances of

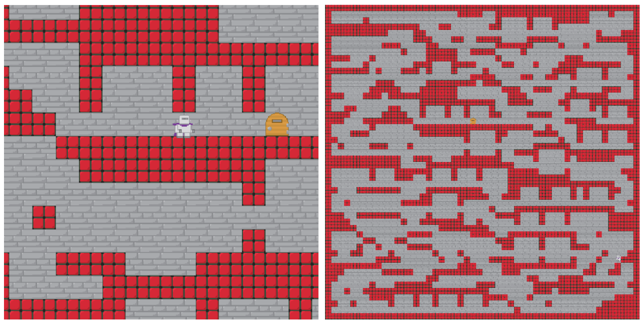


Figure 1: A screenshot of the platform path-finding domain (left), and a zoomed-out image of a single instance (right). The knight must find a path from its starting location, through a maze, to the door (on the right-side in the left image, and just above the center in the right image).

the game are available from the URL provided in the Introduction and an example instance is shown on the right panel in Figure 1. The domain is unit-cost (actions are discretized to the duration of one frame of animation) and has a large state space with a well-informed visibility graph heuristic (Nilsson 1969).

The instances used in our experiments were created using the level generator from `mid`, a special maze generator that builds 2-dimensional platform mazes on a grid of blocks. In our experiments we used $50 \times 50$ grids. Each block is either open or occluded, and to ensure solvability given the constraints imposed by limited jump height, the generator builds the maze by stitching together pieces from a hand-created portfolio. The source code for the level generator is available in the `mid` source repository mentioned above.

The available actions in this domain are different combinations of controller keys (left, right, and jump) that may be pressed during a single iteration of the game's main loop. This allows for the obvious actions such as move left, move right or jump, and also more complicated actions like jump up and left or jump up and right. The average branching factor across the state space is approximately 4. The knight can also jump to different heights by holding the jump button across multiple actions in a row up to a maximum of 8 actions. The actions are unit cost, so the cost of an entire solution is the number of game loop iterations, called frames, required to execute the path. Typically, each frame corresponds to 50ms of game play.

Each state in the state space contains the x and y position of the knight using double-precision floating point values, the velocity in the y direction (x velocity is not stored as it is 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 knight's downward velocity up to a terminal velocity of 12 units per frame.

## Lookahead Commitment

An important step in a real-time search algorithm is selecting how 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 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.

### Single-step and Dynamic Lookahead

In addition to the standard LSS-LRTA* algorithm, we implemented a version that executes single actions like LRTA*, but still applies heuristic learning across the entire local search space. If all of the agent's actions have the same duration, then except for the time spent to find the first action, no time is wasted executing without also planning.

We also implemented LSS-LRTA* using a dynamic lookahead strategy that executes multiple actions. 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 then it simply adjusts its lookahead to fill the entire execution time.

All approaches require offline training to determine the speed at which the agent can perform lookahead search. In the first cases, 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 last case, with a 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 $t$.

### Experimental Evaluation

For our first set of experiments using the platform domain we used 25 test instances. The instances were created using the level generator from `mid` where levels for each instance were 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 25 training platform domain instances. The lookahead values used were 1, 5, 10, 20, 50, 100, 200, 400, 800, 1000,

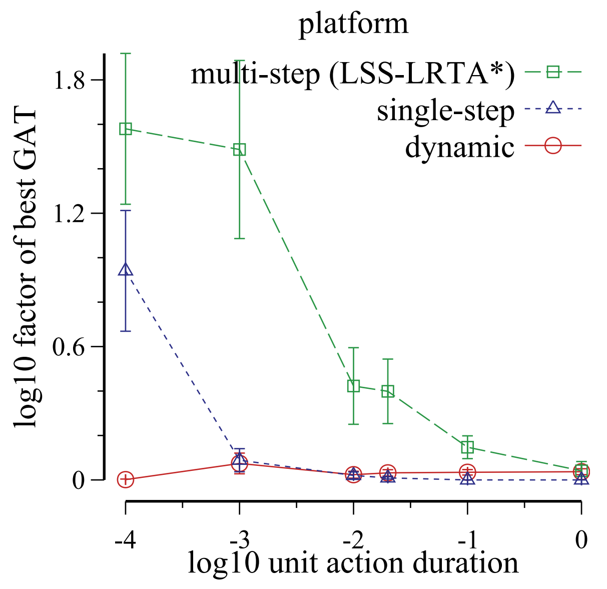
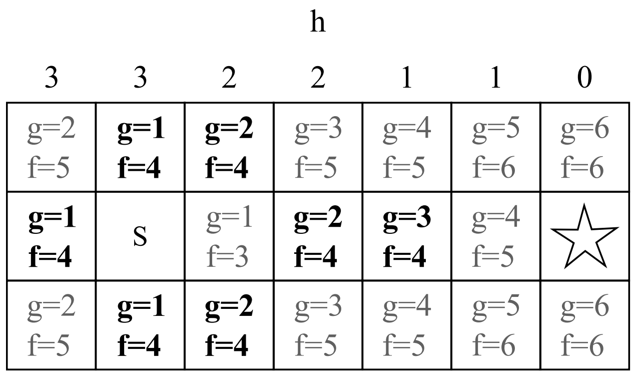


Figure 2: LSS-LRTA*: multi-step, single-step, and dynamic lookahead.



| | | | 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 3: Example of heuristic error and $f$ layers.

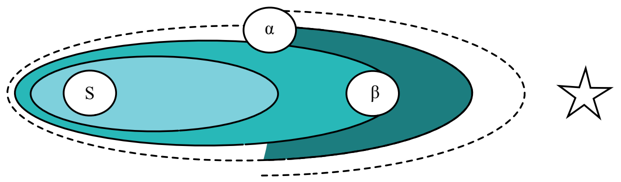


Figure 4: $f$-layered lookahead.

1500, 2000, 3000, 4000, 8000 10000, 16000, 32000, 48000, 64000, 128000, 196000, 256000, 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 later was usually too large due to very rare, slow steps. We attribute these outliers to occasional, unpredictive overhead in things such as memory allocation. We suspect that this issue would go away if our experiments had been run on a real-time operating system where such operations perform more predictive computations.

Figure 2 shows a comparison of these different techniques for the LSS-LRTA* algorithm. The y axis is on a $\log_{10}$ scale and shows the goal achievement time as a factor of the best goal achievement time reported across an instance. We consider a variety of actions durations simulating different possible frame rates or agent capabilities; these are shown on the x axis also on a $\log_{10}$ scale. Smaller action durations represent an agent that can move relatively quickly or a high frame rate, 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. Each point on the plot shows the mean goal achievement time over the 25 test instances, and error bars show the 95% confidence interval.

From this plot, it is clear that the multi-step approach (standard LSS-LRTA*) performs worse than both the single-step and the dynamic lookahead variants in the platform domain. This is likely because the multi-step technique commits to 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. However, as the unit action duration is increased to 1 second (the recommended frame rate for the platform domain is 0.02 seconds) the algorithms begin to perform similarly. However, single-step and dynamic still do appear to perform slightly better.

## 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. We will show why this choice can be problematic, and we will see how it can be remedied.

### Heuristic Error

$f$-based lookahead does not account for heuristic error. The admissible heuristic used to compute $f$ is a low-biased, underestimate of 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 3 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. We call these $f$ *layers*. While some nodes in an $f$ layer are getting closer to the goal node, there are many nodes in each layer that are not. Some nodes in an $f$ layer will be moving exactly away from the goal. In this simple problem, the optimal solution is to move the agent to the 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, and yet 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 3 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 will be led astray.

## 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 4 shows the problem diagramatically. As before, the agent is at the node labelled '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 $f$ 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 consistent with 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 $\alpha$ and will not take the best route toward the goal.

## Improving Lookahead Search

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

The first is quite simple. When choosing where to move, select the lowest $h$ value on the completely expanded $f$ layer with the largest $f$ value, not the next node on open. In Figure 4, this corresponds to the node labeled $\beta$. 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 and orders the search queue and selects actions not based on $f$, but on a less-biased estimate of solution cost.

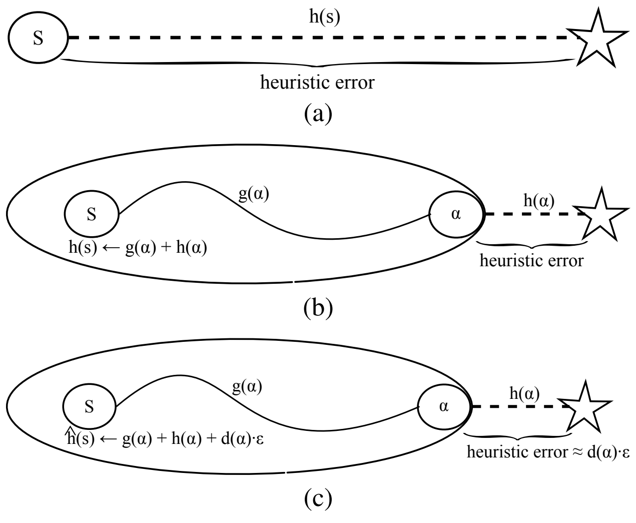


Figure 5: (a) A standard heuristic and its error. (b) An updated heuristic and its error. (c) Using an updated heuristic and accounting for heuristic error.

Instead of using the lower-bound estimate $f$ we use an unbiased estimate that accounts for, and attempts to correct, heuristic error. Thayer, Dionne, and Ruml (2011) did this for offline search and we have adopted this 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 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 $\epsilon$ is the single-step error in the heuristic, and the additional term $d(n) \cdot \epsilon$ corrects the error by adding $\epsilon$ back to the cost estimate for each of the $d(n)$ steps remaining from $n$ to the goal. Following Thayer, Dionne, and Ruml (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, $d$, estimates are readily available for many domains; they tend to be just as easy to compute as heuristic estimates. To estimate $\epsilon$, the single-step heuristic error, we use a global average of the difference between the $f$ values of 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 $g$:

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

Since $g$ is known exactly, as is the cost of the edge $c(parent, child)$, with an imperfect heuristic any difference

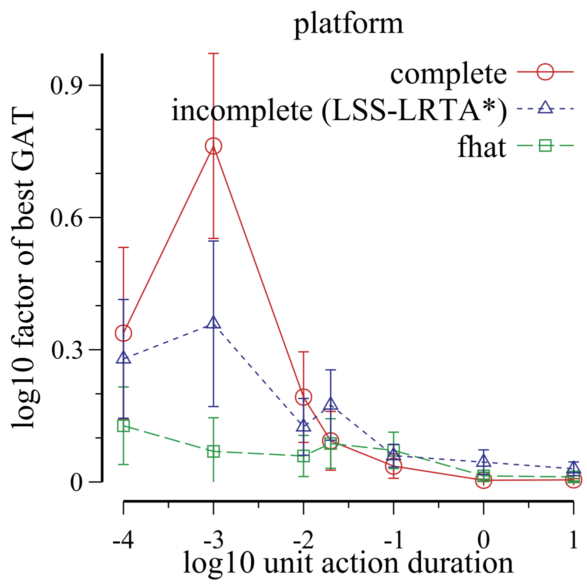


Figure 6: LSS-LRTA*: f-based lookahead and $\hat{f}$-based lookahead.

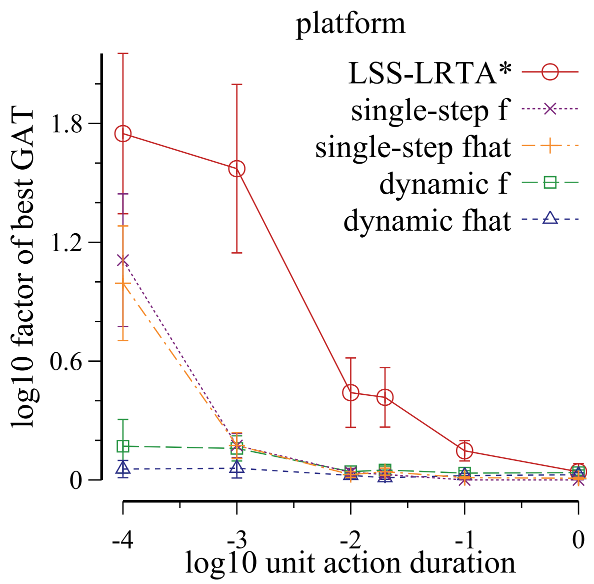


Figure 7: Comparison of the four new real-time techniques.

between $f(child)$ and $f(parent)$ must be caused by error in the heuristic over this step. Averaging these differences gives us our estimate $\epsilon$.

In real-time searches like LSS-LRTA*, the heuristic values of nodes that are expanded during a lookahead search are updated each time the agent moves. Figure 5 (a) shows the default heuristic value for a node $S$. Its error is accrued over the distance from $S$ to the goal. The updated heuristics are more accurate than the originals because they are based on the heuristic values of a node that is closer to the goal, and thus have less heuristic error. This is shown in Figure 5 (b). Here, we can see that $\alpha$ is the node on the fringe from which the start state, $S$, inherits its updated heuristic value. Since $g(\alpha)$, the cost from $S$ to $\alpha$, is known exactly, the error in the backed up heuristic now comes entirely from the steps between $\alpha$ and the goal. Since $\alpha$ is closer to the goal, the error is less than 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 $\alpha$. To do this, we track $d(\alpha)$ for each node with an updated $h$ value, and we use it to compute $\hat{f}$. This is demonstrated by Figure 5 (c). The updated heuristic, when accounting for heuristic error, is $\hat{h}(s) = g(\alpha) + h(\alpha) + d(\alpha) \cdot \epsilon$, where $g(\alpha) + h(\alpha)$ is the standard heuristic backup (cf Figure 5 (b)). Our new technique uses $\hat{f}$ to order expansions during the lookahead search in LSS-LRTA*, and it moves to the next node on the open list with the lowest $\hat{f}$ value.

Figure 6 shows a comparison of the three techniques: the standard *incomplete* $f$-layer method of LSS-LRTA*, the *complete* $f$-layer method, and the approach that uses $\hat{f}$ (*fhat*). To better demonstrate the problem with the standard approach, the plot shows results for the multi-step movement model that commits to a entire path from the current state to the fringe of the local search space after each lookahead. The style of this plot is the same as for Figure 2.

Surprisingly in this figure, we can see that *complete* performed worse than the standard LSS-LRTA* algorithm. However this performance spike subsides and *complete* becomes the dominant performer on the right side of the plot. We attribute this behavior to the underlying motivation for using complete $f$-layers. By only considering complete $f$-layers, the agent will follow the heuristic more closely. However, as we know, the heuristic is not perfectly accurate and cannot be relied on to lead directly to a goal in many cases. Using $\hat{f}$, which attempts to correct the heuristic, to sort the open list of lookahead searches does perform better than the other two algorithms on the left side of the plot but begins to perform slightly worse as the unit action duration is increased.

Figure 7 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 6 and Figure 7, ordering the lookahead search on $\hat{f}$ was the common link to the best performance out of all considered algorithms. Figure 6 demonstrates the initial intuition that heuristic correction can be used and greatly improves performance in real-time search. Figure 7 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.

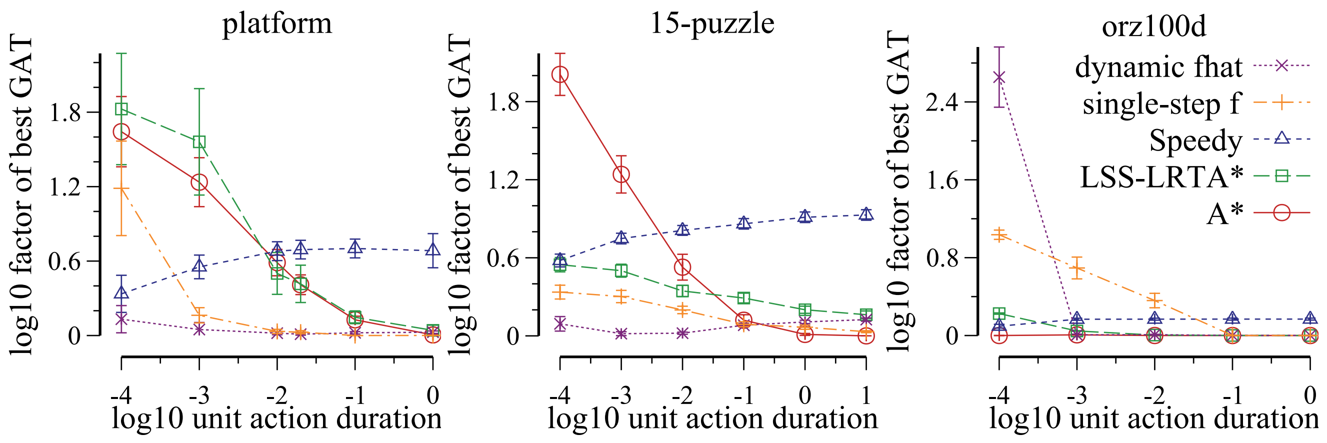


Figure 8: Comparison with off-line techniques.

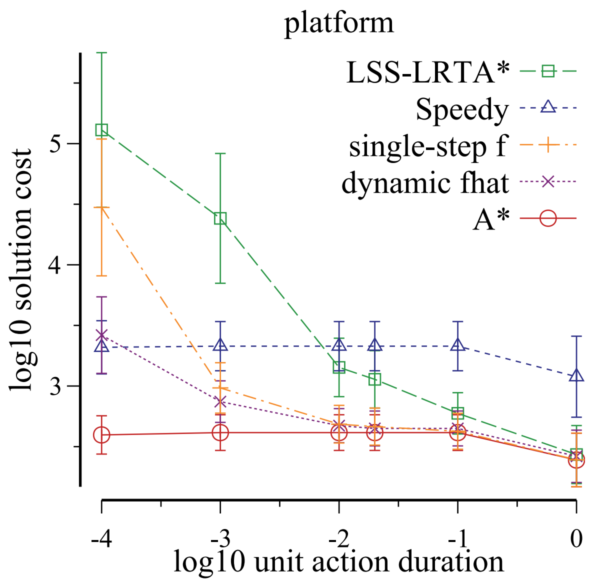


Figure 9: Solution costs for dynamic $\hat{f}$ and single-step $f$.

## Comparison With Off-line Techniques

In the previous sections we have explored modifications to the LSS-LRTA* algorithm. LSS-LRTA*'s performance can be improved by applying a single-step policy, using a dynamically sized lookahead search and applying heuristic value correction. In this section we evaluate the performance of these algorithms against standard offline techniques. We have included two extra domains, 15-puzzle and grid pathfinding, to demonstrate the generality of these approaches in this final comparison. We used the 25 instances that A* was able to solve using a 6GB memory limit from Korf's 100 instances (Korf 1985). For pathfinding, we ran on the orz100d grid map from the video game Dragon Age: Origins (Sturtevant 2012). We used the 25 instances with the longest optimal path length for this domain.

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. Speedy is a best first greedy search on estimated action distance remaining to the goal. Figure 8 shows the results of this comparison with the factor of optimal goal achievement time on the y axis, and Figure 9 shows the same algorithms with the solution costs on the y axis. Optimal goal achievement time is calculated simply as the time required to execute the optimal plan from the start state to a goal state.

In the platform domain plot, dynamic $\hat{f}$ gave the best performance. Its required planning time remained low, as the only time it plans without also executing is during its very first lookahead search used to find first action to execute. Dynamic $\hat{f}$ also performed quite well in the 15-puzzle domain until the unit action duration increased and it was beaten by A*. In the grid pathfinding domain on the orz100d map, A* was clearly the best as these instances were able to be solved quickly.

## Discussion

When optimizing goal achievement time it is important to consider the tradeoff between searching and executing. For very challenging domains such as the platform domain and the 15-puzzle, real-time techniques such as LSS-LRTA* with dynamic $\hat{f}$ provide a very good balance between search and execution. These algorithms are able to search while executing and come up with suboptimal trajectories that can dominate offline algorithms. However, in easier domains such as gridpathfinding on the orz100d map, an offline algorithm that can find optimal solutions very quickly will dominate.

## Conclusion

In this paper we considered real-time search in the context of minimizing goal achievement time. First, we introduced a new platform game benchmark domain that includes dynamics and induces a directed graph. Second, we examined LSS-LRTA*, a popular, state-of-the-art, real-time search algorithm. We saw how it can be improved in challenging domains in three different ways: 1) by performing single-steps instead of multiple actions per-lookahead or by using a dynamic lookahead, 2) by comparing nodes more carefully in the local A*-based lookaheads, and 3) by using $\hat{f}$ to explicitly *unbias* the heuristic estimates, and account for heuristic error. We also showed experimentally that it is not difficult to outperform A* when optimizing goal achievement time in challenging domains or when the unit action duration is relatively small compared to search time.

Given that real-time agent-centered search is inherently suboptimal the use of inadmissible heuristics is a promising area of future research.

## Acknowledgments

This work was supported in part by NSF (grant 0812141) and the DARPA CSSG program (grant D11AP00242).

## References

Hernández, C.; Baier, J.; Uras, T.; and Koenig, S. 2012. Time-bounded adaptive A*. In *Proceedings of the Eleventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-12)*.

Koenig, S., and Likhachev, M. 2002. D* lite. In *Proceedings of the eighteenth national conference on artificial intelligence (AAAI-02)*, 476–483. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Koenig, S., and Sun, X. 2009. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems* 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., and 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*, 108–114.

Nilsson, N. J. 1969. A mobile automaton: an application of artificial intelligence techniques. In *Proceedings of the First International Joint Conference on Artificial intelligence (IJCAI-69)*, 509–520.

Russell, S., and Wefald, E. 1991. *Do the right thing: studies in limited rationality*. The MIT Press.

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

Thayer, J. T., and Ruml, W. 2009. Using distance estimates in heuristic search. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*.

Thayer, J. T.; Dionne, A.; and Ruml, W. 2011. Learning inadmissible heuristics during search. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*.