Auton Agent Multi-Agent Syst (2009) 18:313–341
DOI 10.1007/s10458-008-9061-x

Comparing real-time and incremental heuristic search for real-time situated agents

Sven Koenig · **Xiaoxun Sun**

Published online: 5 September 2008
Springer Science+Business Media, LLC 2008

Abstract Real-time situated agents, such as characters in real-time computer games, often do not know the terrain in advance but automatically observe it within a certain range around themselves. They have to interleave searches with action executions to make the searches tractable when moving autonomously to user-specified coordinates. The searches face real-time requirements since it is important that the agents be responsive to the commands of the users and move smoothly. In this article, we compare two classes of fast heuristic search methods for these navigation tasks that speed up A* searches in different ways, namely real-time heuristic search and incremental heuristic search, to understand their advantages and disadvantages and make recommendations about when each one should be used. We first develop a competitive real-time heuristic search method. LSS-LRTA* is a version of Learning Real-Time A* that uses A* to determine its local search spaces and learns quickly. We analyze the properties of LSS-LRTA* and then compare it experimentally against the state-of-the-art incremental heuristic search method D* Lite on our navigation tasks, for which D* Lite was specifically developed, resulting in the first comparison of real-time and incremental heuristic search in the literature. We characterize when to choose each one of the two heuristic search methods, depending on the search objective and the kind of terrain. Our experimental results show that LSS-LRTA* can outperform D* Lite under the right conditions, namely when there is time pressure or the user-supplied h-values are generally not misleading.

Keywords Time constraints · Agents · Real-time search · Incremental search

S. Koenig (✉) · X. Sun
University of Southern California, Los Angeles, CA 90089-0781, USA
e-mail: skoenig@usc.edu

X. Sun
e-mail: xiaoxuns@usc.edu

1 Introduction

Consider navigation tasks for characters in real-time computer games (such as Baldur's Gate, Total Annihilation, Age of Empires or Dark Reign) as an example of search tasks that real-time situated agents face. Agents in real-time computer games often do not know the terrain in advance but automatically observe it within a certain range around themselves and then remember it for future use. To make the agents easy to control, one needs to give them the capability to understand and execute high-level user commands. For example, the users can click on certain coordinates in known or unknown terrain and the agents then move autonomously to these coordinates.

These navigation tasks are interesting because they are different from traditional search tasks encountered in fields other than autonomous agents. Traditional search tasks have state spaces that are completely known in advance (= off-line search), do not fit into the available computer memory and do not need to be solved in real time. The predominant research issue is how to search larger and larger state spaces. Our navigation tasks, on the other hand, have state spaces that are not completely known in advance (= on-line search), fit into the available computer memory and need to be solved in real time. Our research issue is how to search the state spaces faster and faster.

In particular, our agents operate in terrain whose maps fit into computer memory but they might not know the terrain in advance and the resulting large number of contingencies makes search difficult. Finding shortest trajectories is often intractable since it involves finding large conditional plans. However, the agents need to act in real-time in order to be responsive to the commands of the users and move smoothly. Thus, they need to use search approaches that make search fast by sacrificing the optimality of the resulting trajectories, in our case by interleaving searches with movements (= action executions). As the agents move in the terrain, they observe more of it, which speeds up future searches since it reduces the number of possible contingencies. The resulting trajectories are likely not optimal but this is often outweighed by the computational savings gained.

There are two classes of fast heuristic search methods that are well suited for agents and fit this framework but have never been compared. Thus, it is unclear what their advantages and disadvantages are and when each one should be used. The first class is real-time heuristic search from artificial intelligence, which makes A* search efficient by limiting the lookahead of the A* search and can thus satisfy hard real-time requirements. The second class is incremental heuristic search from robotics, which makes A* search efficient by reusing information from the previous A* search to speed up the current one. Both classes of heuristic search methods are extensions of A* and have similar properties, for example, use heuristic estimates of the goal distances (= h-values) to focus their A* search and eventually follow a shortest trajectory from the start state to the goal state if they are teleported back to the start state whenever they reach the goal state.

Incremental heuristic search methods have been developed specifically for our navigation tasks but we suggest that one can develop real-time heuristic search methods that outperform them under the right conditions. We test this hypothesis as follows: We first develop a competitive real-time heuristic search method. Local Search Space LRTA* (= LSS-LRTA*) is a version of Learning Real-Time A* that uses A* to determine its local search spaces and learns quickly. We analyze its properties and then compare it experimentally against the state-of-the-art incremental heuristic search method D* Lite [21] on our navigation tasks, for which D* Lite was specifically developed, resulting in the first comparison of real-time and incremental heuristic search in the literature. We characterize when to choose each one of the two heuristic search methods, depending on the kind of terrain (which determines how informed the h-values are) and the search objective, for example, minimizing the sum of the search and action-execution time or minimizing the trajectory length subject to the hard real-time requirement that only a certain amount of time is available for each search, where the search time either can or cannot be amortized over the action executions. Our experimental results show that LSS-LRTA* can indeed outperform D* Lite under the right conditions, namely when there is time pressure or the user-supplied h-values are generally not misleading.

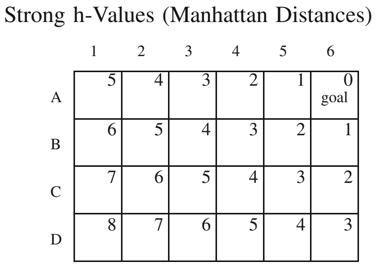
2 Our navigation tasks

We need a test domain to compare real-time and incremental heuristic search methods. For this purpose, we use navigation tasks for which incremental heuristic search methods, such as D* Lite, were specifically developed, namely navigation tasks where a real-time situated agent has to move autonomously to user-specified coordinates in terrain that it does not know in advance. The terrain is discretized into cells that are either blocked or unblocked, a common practice in the context of real-time computer games [2]. We assume for simplicity that the agent can move in the four main compass directions with equal cost and thus operates on undirected four-neighbor grids. The agent does not know in advance which cells are blocked. It always observes which (unblocked) cell it is in, observes the blockage status of its neighboring cells, and can then move to any one of the unblocked neighboring cells. Its task is to move from a given start cell to a given goal cell. Because the agent does not know in advance which cells are blocked, it might have to try out many paths that eventually turn out to be blocked before it finds an unblocked path to the goal cell. Thus, its trajectory length tends to be much longer than if it knew in advance which cells are blocked. As h-value of a cell we use the sum of the absolute difference of its x and y coordinates to the x and y coordinates of the goal cell (= Manhattan distance), as shown in Fig. 1 (left).

Our navigation tasks are just one possible test domain for real-time and incremental heuristic search methods since both classes of heuristic search methods also apply to more complex scenarios, such as state spaces with nonuniform action costs and more complex topologies than grids as well as agents with larger sensor ranges.

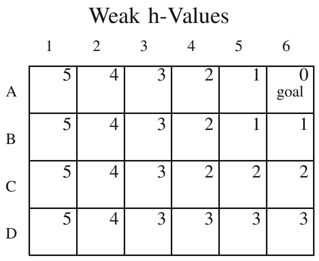
2.1 General approach: freespace assumption

Finding shortest trajectories involves finding large conditional plans since the agent does not know in advance which cells are blocked. However, it needs to act in real-time in order to be



Strong h-Values (Manhattan Distances)

	1	2	3	4	5	6
A	5	4	3	2	1	0 goal
B	6	5	4	3	2	1
C	7	6	5	4	3	2
D	8	7	6	5	4	3



Weak h-Values

	1	2	3	4	5	6
A	5	4	3	2	1	0 goal
B	5	4	3	2	1	1
C	5	4	3	2	2	2
D	5	4	3	3	3	3

Fig. 1 Strong h-values (left) and weak h-values (right) [upper right number in a cell = h-value]

responsive to the commands of the users and move smoothly. Thus, it needs to use a search approach that makes search fast by sacrificing the optimality of the resulting trajectories. It needs to use the same search approach whether it uses real-time or incremental heuristic search methods to make the search methods comparable.

In our case, the agent always uses the following approach: It interleaves searches with movements (= action executions). As it moves in the terrain, it observes more of it, which speeds up future searches since it reduces the number of possible contingencies. The resulting trajectories are likely not optimal but this is often outweighed by the computational savings gained. This approach allows the agent to make search tractable by using the freespace assumption: it always finds a path under the assumption that cells whose blockage status it has not observed yet are unblocked [28] and thus determines a path that starts at its current cell and does not pass through cells that it has observed to be blocked (= a presumed unblocked path). It then moves along that path, which allows it to observe additional blocked cells. If it observes its current path to be blocked, then it determines another presumed unblocked path that starts at its current cell and moves along that path. Thus, the agent interleaves repeated searches with action executions, and the trajectory of the agent can thus be different from the paths found by the searches. The agent cannot inadvertently execute actions that make it impossible for it to reach the goal cell with finite path length from its current cell since our grids are undirected and the agent is therefore able to undo the effects of all actions that it executes.

2.2 Finding complete paths with incremental heuristic search

An agent that uses *incremental heuristic search* interleaves searches with action executions and uses the freespace assumption as follows: It always finds a complete shortest presumed unblocked path from its current cell to the goal cell. If such a path does not exist, then there does not exist an unblocked path from its current cell (or the start cell) to the goal cell either. Since blocked cells cannot become unblocked, there cannot exist a path from its current cell (or the start cell) to the goal cell in the future either. Otherwise, the agent moves along the complete shortest presumed unblocked path until it reaches the goal cell or observes the path to be blocked. If the current cell of the agent is different from the goal cell, then it repeats the process, otherwise it terminates successfully.

Figure 2 shows an example. The solid circle depicts the agent. Black cells depict blocked cells that the agent has already observed, and grey cells depict blocked cells that the agent has not yet observed and thus assumes to be unblocked. The arrows depict the complete shortest presumed unblocked paths.

Some theoretical properties of this navigation approach have been studied in the literature [28,31]. It either moves the agent to the goal cell or correctly reports that this is impossible. The number of action executions of the agent tends not to be minimal (since it makes the simplifying freespace assumption). However, it uses all available information about blocked cells, and the number of action executions of the agent is thus reasonably small. In fact, it either moves the agent to the goal cell or correctly reports that this is impossible after $O(|S| \log |S|)$ action executions on grids with $|S|$ unblocked cells and thus is a reasonable navigation approach for our navigation tasks. The agent eventually follows a shortest trajectory from the start cell to the goal cell if it is teleported back to the start cell whenever it reaches the goal cell (since it makes the optimistic freespace assumption).

The repeated searches can be implemented efficiently with incremental heuristic search methods [25]. Incremental heuristic search methods are extensions of A* [12,13] that reuse

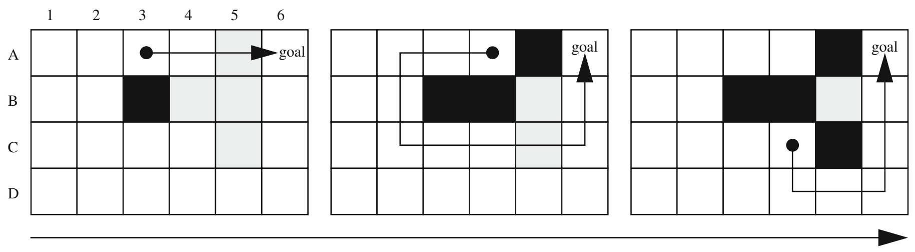


Fig. 2 Following complete shortest presumed unblocked paths [solid circle = agent; arrow = complete shortest presumed unblocked path; black cell = blocked cell that has been observed; grey cell = blocked cell that has not been observed yet and thus is assumed to be unblocked; white cell = unblocked cell]

information from the previous A* search to speed up the current one. Reusing information from the previous A* search is possible because the agent typically discovers only a small number of blocked cells between A* searches, and successive A* searches are thus similar. There are three classes of incremental heuristic search methods. The first class restarts A* at the point where its search deviates from the second one. Examples of incremental heuristic search methods from this class are Incremental A* (an unpublished incremental heuristic search method by Peter Yap) and Fringe Saving A* [38]. The second class updates the h-values from the previous search during the current search to make them more informed. An example of incremental heuristic search methods from this class is Adaptive A* [23], that builds on a principle that was first used in [15]. The third class updates the g-values from the previous search during the current search to correct them when necessary, which can be interpreted as transforming the A* search tree from the previous search into the A* search tree for the current search. Examples of incremental heuristic search methods from this class are Differential A* [41], Focused Dynamic A* (D*) [37], Lifelong Planning A* [24] and its generalization D* Lite [21]. They all find the same paths but have different search times. The existing comparisons among them suggest that the third class is most efficient on our navigation tasks. D* and D* Lite are more sophisticated than Differential A*. D* is widely used in mobile robotics, including Mars rovers and tactical mobile robot prototypes [14,40]. We use D* Lite in our comparison since D* and D* Lite are about equally efficient but D* Lite is easier to understand. It is beyond this article to describe the details of D* Lite but they can be found in [22].

2.3 Finding prefixes of complete paths with real-time search

One problem with always finding a complete shortest presumed unblocked path from the current cell of the agent to the goal cell is that the search time grows with the size of the terrain. Thus, the search time per search is not constant, and this navigation approach thus does not satisfy hard real-time requirements where only a certain amount of time is available for each search, independent of the size of the terrain.

An agent that uses *real-time heuristic search* therefore interleaves searches with action executions and uses the freespace assumption as follows, potentially at the expense of increasing the number of action executions: It always finds the beginning (= prefix) of a complete presumed unblocked path from its current cell to the goal cell. The agent then moves along that path until it reaches the end of the path or observes the path to be blocked. If the current cell of the agent is different from the goal cell, then it repeats the process, otherwise it terminates successfully.

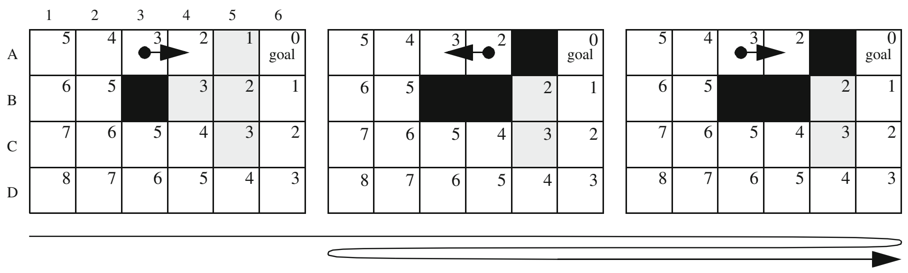


Fig. 3 Cycling forever [solid circle = agent; arrow = prefix of the complete shortest presumed unblocked path; black cell = blocked cell that has been observed; grey cell = blocked cell that has not been observed yet and thus is assumed to be unblocked; white cell = unblocked cell; upper right number in a cell = h-value]

One way of achieving a constant search time per search is to restrict the search to the part of the terrain around the current cell of the agent (= agent-centered search) [18]. Agent-centered search determines the local search space and then finds a presumed unblocked path within it that starts at the current cell of the agent. Real-time heuristic search is an agent-centered search approach that stores an h-value in memory for all cells that it encounters during its searches and uses asynchronous dynamic programming to update the h-values as the search progresses to make them more informed and, this way, prevent the agent from executing actions forever. Figure 3 shows how the agent can execute actions forever if it always moves to the neighboring cell with the smallest h-value and thus greedily performs steepest descent on the h-value surface. Its first two moves repeat indefinitely since it gets stuck in a local minimum of the h-value surface. Updating the h-values avoids this problem. The initial h-values can be zero (= completely uninformed) but one can supply initial h-values that are more informed in order to focus the searches toward the goal cell right away.

Real-time heuristic search methods have been studied more extensively in artificial intelligence than incremental heuristic search methods and been extended in various directions. Learning Real-Time A* (= LRTA*) [29] is probably the most popular real-time heuristic search method. It works in deterministic state spaces, associates h-values with states, selects actions so as to greedily perform steepest descent on the h-value surface and converges to a shortest trajectory from the start state to the goal state if it is teleported back to the start state whenever it reaches the goal state. However, researchers have also studied real-time heuristic search methods that work in nondeterministic state spaces [19] or in probabilistic state spaces [1,3], that associate h-values with the actions [26,27,43], and that select actions in more sophisticated ways than greedily performing steepest descent on the h-value surface to either result in shorter trajectories from the start state to the goal state [27,29] (which gives up convergence to a shortest trajectory from the start state to the goal state if they are teleported back to the start state whenever they reach the goal state) or speed up convergence [7,10,35,36]. A longer overview is given in [17].

Real-time heuristic search methods have been used in artificial intelligence to solve large off-line search tasks, including the 24 puzzle [30] and STRIPS-type planning tasks [4]. Simple versions of real-time heuristic search methods have also been studied in the context of situated agents [1,11,17], including robots [19,39,42] and characters in real-time computer games [6].

3 LSS-LRTA*

We use a version of LRTA* in our comparison since LRTA* follows a shortest trajectory from the start state to the goal state if it is teleported back to the start state whenever it reaches the goal state (since the h-values are admissible), which is compatible with the freespace assumption. Here, we are concerned with two research issues, namely which states should be in the local search spaces and the h-values of which states in the local search spaces should be updated:

- An important research issue is which states should be in the local search spaces. LRTA* is most often used with local search spaces that contain only the current state of the agent but their sizes should really be optimized for the search objective. Although versions of LRTA* with larger local search spaces have been suggested [6,16,33,34], they typically do not satisfy hard real-time requirements since they determine the size of the local search spaces based on other considerations, such as the size of the depression of the h-value surface around the current state or the size of the known terrain. The easiest way of bounding the search time per search is to limit the size of the local search spaces. It makes sense for the local search spaces to be continuous parts of the state space around the current state of the agent since these parts of the state space contain the states that the agent might soon be in and are thus immediately relevant for the agent in its current situation. The original version of LRTA* [29] chooses its local search spaces by performing a breadth-first search with pruning from the current state of the agent toward the goal state up to a given depth. The states expanded by the breath-first search then form the local search space. The original version of LRTA* thus does not choose the local search spaces in a fine-grained way. Furthermore, the local search spaces are disk-shaped and might thus have a suboptimal shape.
- Another important research issue is the h-values of which states in the local search spaces should be updated. The original version of LRTA* updates only the h-value of the current state of the agent. It therefore does not learn quickly [34] and thus is not the heuristic search method that we want to compare against D* Lite.

We therefore introduce Local Search Space LRTA* (= LSS-LRTA*) [20], a version of LRTA* that addresses both of the research issues above. It uses A* to determine its non-disk-shaped local search spaces in a fine-grained way and updates the h-values of all states in the local search spaces to learn quickly. Figure 4 shows the steps of LSS-LRTA*. In Step 1, LSS-LRTA* uses A* to choose its local search spaces. A* searches from the current state of the agent toward the goal state until *lookahead* > 0 states have been expanded or the goal state is about to be expanded. We refer to the external parameter *lookahead* as the lookahead of LSS-LRTA*. The states expanded by A* form the local search space. In Step 2, LSS-LRTA* uses Dijkstra's algorithm [8] to update the h-values of all states in the local search space. Dijkstra's algorithm replaces the h-values of all states in the local search space with the sum

1. Use A* [13, 12] to search from the current state of the agent toward the goal state until *lookahead* > 0 states have been expanded or the goal state is about to be expanded. The states expanded by A* form the local search space.
2. Use Dijkstra's algorithm [8] to replace the h-values of all states in the local search space with the sum of the distance from the state to a state s and the h-value of state s, minimized over all states $s \in S$ that border the local search space.
3. Move the agent along the path found by A* until it reaches the end of the path (and leaves the local search space) or action costs on the path increase.
4. If the current state of the agent is different from the goal state, then go to Step 1, otherwise terminate successfully.

Fig. 4 Overview of LSS-LRTA*

of the distance from the state to a state s and the h-value of state s, minimized over all states $s \in S$ that were generated but not expanded by A* (= that border the local search space). In Step 3, LSS-LRTA* moves the agent along the path found by A* until it reaches the end of the path (and leaves the local search space) or action costs on the path increase. In Step 4, if the current state of the agent is different from the goal state, then LSS-LRTA* repeats the process, otherwise it terminates successfully.

All searches of A* and Dijkstra's algorithm expand all states in the local search space only once and are thus efficient. The idea behind using A* in Step 1 of LSS-LRTA* is to try to reject the current path if additional search time is available. The agent moves in Step 3 from its current state toward the state that A* would have expanded next if it had been allowed to expand one additional state, which could have changed the current path. The idea behind using Dijkstra's algorithm in Step 2 of LSS-LRTA* is to update the h-values of all states in the local search space to make them locally consistent [33] and thus propagate as much information as possible from the states that border the local search space to the states in the local search space. Thus, LSS-LRTA* learns quickly.

We need to show experimentally that LSS-LRTA* with local search spaces chosen with A* can indeed outperform LSS-LRTA* with local search spaces chosen with breath-first search. We also need to show experimentally that LSS-LRTA* can indeed outperform D* Lite under the right conditions.

3.1 Formalization of LSS-LRTA*

We use the following notation to describe search tasks: S is the finite set of states. $s_{\text{start}} \in S$ is the start state (changed during execution to be the current state of the agent), and $s_{\text{goal}} \in S$ is the goal state. $A(s)$ is the finite set of actions that can be executed in state $s \in S$. $c(s, a) > 0$ is the action cost of executing action $a \in A(s)$ in state $s \in S$ (changed during execution), and $succ(s, a) \in S$ is the resulting successor state.

Besides the description of the search task, one also needs to supply the lookahead $lookahead > 0$ of LSS-LRTA* and initial h-values $h(s)$ for all states $s \in S$ (changed during execution). The user-supplied h-values estimate the goal distances and need to satisfy the triangle inequality (= be consistent) [32], that is, satisfy $h(s_{\text{goal}}) = 0$ and $h(s) \leq c(s, a) + h(succ(s, a))$ for all states $s \in S \setminus \{s_{\text{goal}}\}$ and actions $a \in A(s)$. Consistent h-values are guaranteed to not overestimate the goal distances (= are admissible).

For our navigation tasks, the states correspond to the cells. The start state corresponds to the start cell, and the goal state corresponds to the goal cell. The actions in states correspond to the movements from the corresponding cells to their neighboring blocked or unblocked cells. All action costs are one initially. If the agent observes a cell to be blocked, then the action costs of all actions that can be executed in the corresponding state or that result in the corresponding state are set to infinity. Then, every shortest action sequence (= path) from the start state to the goal state in the state space corresponds to a complete shortest presumed unblocked path from the start cell to the goal cell on the grid and vice versa.

Figure 5 gives the pseudo code of LSS-LRTA*. (The minimum over an empty set is infinity on Line 8.) The pseudo code uses the standard data structures of A*, namely an *OPEN* list (which is a priority queue) and a *CLOSED* list (which is a set). It also uses the standard variables, namely g-values $g(s)$, h-values $h(s)$, f-values $g(s) + h(s)$ and tree pointers $tree(s)$ for all states $s \in S$. State $s'_{\text{goal}} \in S$ on Line 32 is the state that A* would have expanded next if it had been allowed to expand one additional state. The path found by A* then is a shortest path from the current state of the agent to state s'_{goal}. Following the tree pointers of A* from

```
 1 procedure Astar()
 2 for each $s \in S$
 3   $g(s) := \infty$;
 4 $g(s_{start}) = 0$;
 5 $OPEN := CLOSED := \emptyset$;
 6 insert $s_{\text{start}}$ into $OPEN$;
 7 $expansions := 0$;
 8 while $g(s_{\text{goal}}) > \min_{s' \in OPEN}(g(s') + h(s'))$ AND $expansions < lookahead$
 9   $expansions := expansions + 1$;
10   delete a state $s$ with the smallest f-value $g(s) + h(s)$ from $OPEN$;
11   $CLOSED := CLOSED \cup \{s\}$;
12   for each $a \in A(s)$
13     if $g(succ(s, a)) > g(s) + c(s, a)$
14       $g(succ(s, a)) := g(s) + c(s, a)$;
15       $tree(succ(s, a)) := s$;
16       if $succ(s, a)$ is not in $OPEN$ then insert it into $OPEN$;
17 procedure Dijkstra()
18 for each $s \in CLOSED$
19   $h(s) := \infty$;
20 while $CLOSED \neq \emptyset$
21   delete a state $s$ with the smallest h-value $h(s)$ from $OPEN$;
22   if $s$ is in $CLOSED$ then delete it from $CLOSED$;
23   for each $s' \in S$ and $a \in A(s')$ with $succ(s', a) = s$
24     if $s' \in CLOSED$ AND $h(s') > c(s', a) + h(s)$
25       $h(s') := c(s', a) + h(s)$;
26       if $s'$ is not in $OPEN$ then insert it into $OPEN$;
27 procedure Main()
28 while $s_{\text{start}} \neq s_{\text{goal}}$
29   Astar();
30   if $OPEN = \emptyset$
31     stop;
32   $s'_{\text{goal}} := \arg\min_{s' \in OPEN}(g(s') + h(s'))$ (assign $s_{\text{goal}}$ if possible);
33   Dijkstra();
34   move the agent along the path from $s_{\text{start}}$ to $s'_{\text{goal}}$ identified by the tree pointers of A*
       until it reaches $s'_{\text{goal}}$ (and leaves the local search space) or action costs on the path increase;
35   set $s_{\text{start}}$ to the current state of the agent (if the current state of the agent has changed);
36   update the action costs (if action costs have increased);
```

Fig. 5 LSS-LRTA*

state s'_{goal} to the current state of the agent identifies such a path in reverse. The pseudo code of LSS-LRTA* initializes the g-values of A* before each execution of A* on Lines 2–3 for ease of description even though many of them might not get used. In reality, LSS-LRTA* initializes a g-value only when it is used for the first time during an A* search. Similarly, the pseudo code of LSS-LRTA* assumes that the h-values are initialized before each execution of LSS-LRTA* even though many of them might not get used. In reality, LSS-LRTA* initializes an h-value only when it is used for the first time during the execution of LSS-LRTA*. It caches all h-values updated by Dijkstra's algorithm that are different from the user-supplied h-values. If it has cached an h-value for a state, then it uses the cached h-value, otherwise it uses the user-supplied h-value instead.

The *CLOSED* list of A* contains the states expanded by A* and thus the states in the local search space. The *OPEN* list of A* contains the states generated but not expanded by A* and thus the states that border the local search space. If the *OPEN* list is empty on Line 30 then there does not exist a path of finite path length from the current state of the agent to the goal state. Since the action costs never decrease, there cannot exist a path of finite path length from the current state of the agent to the goal state in the future, and LSS-LRTA* thus terminates unsuccessfully on Line 31.

Dijkstra's algorithm sets the h-values of all states in the local search space to infinity on Lines 18–19 and then assigns each state in the local search space a new h-value. Dijkstra's algorithm needs to initialize its *OPEN* list with the states that border the local search space

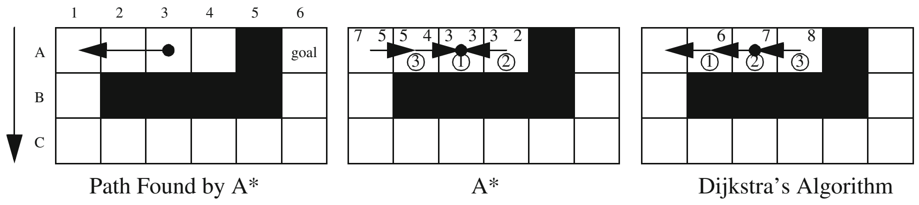


Fig. 6 Interfacing A* and Dijkstra's Algorithm (Example 1) [solid circle = agent; arrow = prefix of the complete shortest presumed unblocked path (left), tree pointer of A* (center) and tree pointer of Dijkstra's algorithm (right); black cell = blocked cell that has been observed; white cell = unblocked cell; upper left number in a cell = f-value; upper right number in a cell = h-value (in parentheses if it is not cached since Dijkstra's algorithm did not change it); circled number = order in which A* (center) or Dijkstra's algorithm (right) updates the cells]

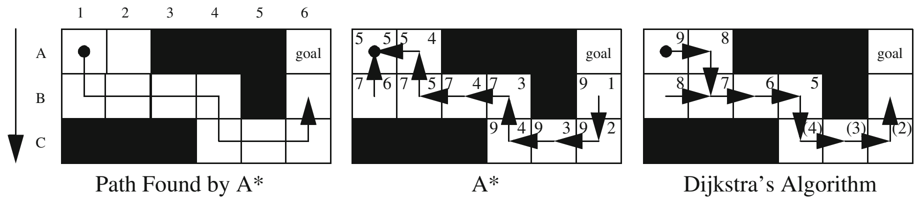


Fig. 7 Interfacing A* and Dijkstra's Algorithm (Example 2) [solid circle = agent; arrow = prefix of the complete shortest presumed unblocked path (left), tree pointer of A* (center) and tree pointer of Dijkstra's algorithm (right); black cell = blocked cell that has been observed; white cell = unblocked cell; upper left number in a cell = f-value; upper right number in a cell = h-value (in parentheses if it is not cached since Dijkstra's algorithm did not change it)]

but these are exactly the states in the *OPEN* list of A*. Thus, Dijkstra's algorithm can reuse the *OPEN* list of A*. However, it needs to change the priorities of the states in the *OPEN* list from their f-values to their h-values. It appears that Dijkstra's algorithm should be able to reuse more information from the previous A* search than the *OPEN* list, such as the order of state expansions or the tree pointers of A*. However, it is unclear how to exploit this information. Figure 6, for instance, gives an example of LSS-LRTA* with lookahead three in known terrain where the order in which Dijkstra's algorithm updates the cells is different from both the order and the reverse of the order in which A* expanded the cells. The circled numbers show this order. Similarly, Fig. 7 gives an example of LSS-LRTA* with lookahead nine in known terrain where the h-value of cell B2 is used to update the h-value of cell B1 during the execution of Dijkstra's algorithm even though these cells are not connected by tree pointers of A*.

3.2 Illustration of LSS-LRTA*

Figure 8 shows the beginning of an example of how LSS-LRTA* operates. The lookahead is three. The left column shows how the agent moves, similar to Fig. 2. The center column shows the results of A*. All cells generated by A* and all cells with cached h-values are labeled with their h-values in the upper right corner. All cells generated by A* are also labeled with their f-value in the upper left corner. The arrows depict the tree pointers of A*. Thus, one arrow leaves each cell generated by A*. This arrow points to the cell expanded by A* whose g-value was used to calculate the g-value of the cell in question during the A* search. The

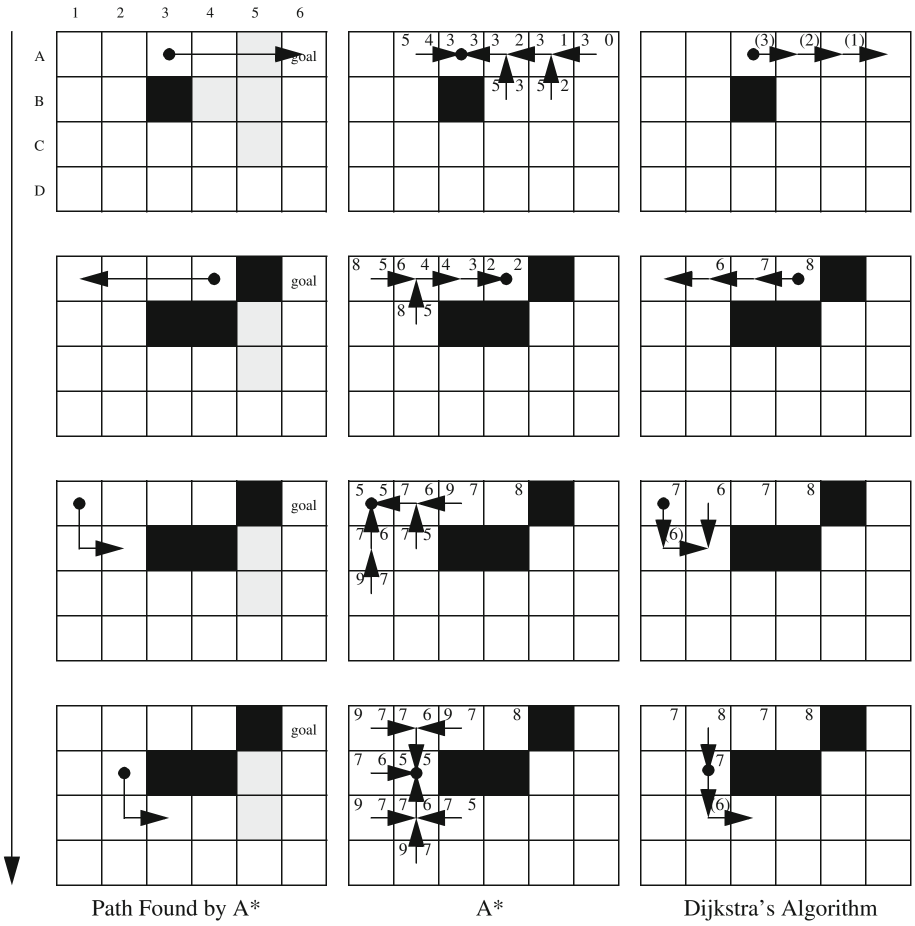


Fig. 8 Example trace of LSS-LRTA* [solid circle = agent; arrow = prefix of the complete shortest presumed unblocked path (left), tree pointer of A* (center) and tree pointer of Dijkstra's algorithm (right); black cell = blocked cell that has been observed; grey cell = blocked cell that has not been observed yet and thus is assumed to be unblocked; white cell = unblocked cell; upper left number in a cell = f-value; upper right number in a cell = h-value (in parentheses if it is not cached since Dijkstra's algorithm did not change it)]

cells expanded by A* form the local search space and get their h-values updated by Dijkstra's algorithm. The right column shows the results of Dijkstra's algorithm. All cells with updated or cached h-values are labeled with them in the upper right corner. The arrows depict the tree pointers of Dijkstra's algorithm. One arrow leaves each cell with an updated h-value. This arrow points to the cell whose h-value was used to update the h-value of the cell in question during the execution of Dijkstra's algorithm. Updated h-values in parentheses are not cached since they are identical to the user-supplied h-values. Only the beginning of the trajectory of the agent is shown. The trajectory is longer than the one from Fig. 2 since all A* searches now use only small local search spaces. In contrast, LSS-LRTA* with an infinite lookahead always finds a complete shortest presumed unblocked path from the current cell of the agent to the goal cell and thus follows the same trajectory as D* Lite, that is shown in Fig. 2.

3.3 Analysis of LSS-LRTA*

We now prove several properties of LSS-LRTA* that hold for all positive lookaheads. These proofs generalize previous proofs in [19] substantially, including to action costs that are not uniformly one and can increase. They are (implicitly) by induction on the number of searches with A* and Dijkstra's algorithm.

We use the following known properties of A* with consistent h-values [32]: First, it expands all states at most once. Second, the g-values of states expanded by A* are equal to the distance from the start state to these states. Following the tree pointers of A* from these states to the start state identifies a shortest path from the start state to them in reverse. The g-value of the goal state after the A* search is equal to the distance from the start state to the goal state, that is, the goal distance of the start state. Following the tree pointers of A* from the goal state to the start state after the A* search identifies a shortest path from the start state to the goal state in reverse. Third, the f-values of the series of expanded states over time are monotonically nondecreasing. Thus, $f(s) \leq f(s_{\text{goal}}) = g(s_{\text{goal}})$ for all states $s \in S$ expanded by A* (states in the *CLOSED* list after termination) and $g(s_{\text{goal}}) = f(s_{\text{goal}}) \leq f(s)$ for all states $s \in S$ generated but not expanded by A* (states in the *OPEN* list after termination). Fourth, an A* search expands no more states than another A* search for the same search task except possibly for some states $s \in S$ with $g(s_{\text{goal}}) = f(s_{\text{goal}}) = f(s)$ if the h-values used by the first A* search are no smaller than the corresponding h-values used by the second A* search for all states (= the former h-values dominate the latter h-values).

We also use that the h-values $h'(s)$ after the execution of Dijkstra's algorithm satisfy the following system of equations for all states $s \in S$ in the local search space:

$$h'(s) = \min_{a \in A(s)} (c(s,a) + h'(succ(s,a))). \tag{1}$$

Theorem 1 *The h-value of the same state is monotonically nondecreasing over time and thus indeed becomes more informed over time.*

Proof by contradiction: The h-values can change only during the execution of Dijkstra's algorithm, and only the h-values in the local search space can change. Let $h(s)$ be the consistent h-values before the execution of Dijkstra's algorithm and $h'(s)$ be the h-values afterwards. Assume that the h-values of one or more states have decreased. Consider a state $s \in S$ with the smallest h-value $h'(s)$ among all states whose h-values have decreased. Thus, state s is in the local search space. The goal state is never in the local search space. Thus, state s is not the goal state. Let $a = \arg\min_{a' \in A(s)}(c(s,a') + h'(succ(s,a')))$. Then, $h'(s) = \min_{a' \in A(s)}(c(s,a') + h'(succ(s,a'))) = c(s,a) + h'(succ(s,a)) > h'(succ(s,a))$. Thus, state $succ(s,a)$ is not a state whose h-value has decreased, implying that $h(succ(s,a)) \leq h'(succ(s,a))$. Furthermore, the h-values $h(s)$ are consistent, implying that $h(s) \leq c(s,a) + h(succ(s,a))$. Thus, $h'(s) = c(s,a) + h'(succ(s,a)) \geq c(s,a) + h(succ(s,a)) \geq h(s)$, which is a contradiction. □

Theorem 2 *The h-values remain consistent and thus also admissible.*

Proof The only reason why the h-values might not remain consistent are the changing h-values and action costs. First, the h-values can change only during the execution of Dijkstra's algorithm. Let $h(s)$ be the consistent h-values before the execution of Dijkstra's algorithm and $h'(s)$ be the h-values afterwards. Consider any state $s \in S$. We distinguish three cases, making use of the consistency of the h-values $h(s)$ and the fact that they cannot decrease according to Theorem 1. Case 1: State s is in the local search space and thus not a goal state. Then, $h'(s) = \min_{a' \in A(s)}(c(s,a') + h'(succ(s,a'))) \leq c(s,a) + h'(succ(s,a))$ for all actions $a \in A(s)$. Case 2: State s is not in the local search space and not the goal state. Then, $h'(s) = h(s) \leq c(s,a) + h(succ(s,a)) \leq c(s,a) + h'(succ(s,a))$ for all actions $a \in A(s)$. Case 3: State s is the goal state. Then, $h'(s) = h(s) = 0$. Thus, the h-values $h'(s)$ are consistent. Second, let $c(s,a)$ be the action costs before any action cost increases, and $c'(s,a)$ be the action costs afterwards. We distinguish two cases, making use of the consistency of the h-values $h'(s)$. Consider any state $s \in S$. Case 1: State s is not the goal state. Then, $h'(s) \leq c(s,a) + h'(succ(s,a)) \leq c'(s,a) + h'(succ(s,a))$ for all actions $a \in A(s)$. Case 2: State s is the goal state. Then, $h'(s) = 0$. Thus, the h-values $h'(s)$ remain consistent. □

Lemma 1 *It holds that $h(s) = c(s,a) + h(succ(s,a))$ for the h-values $h(s)$ at the time when the agent executes action $a \in A(s)$ in state $s \in S$.*

Proof Consider the A* search immediately before the agent executes action $a \in A(s)$ in state $s \in S$. State s was expanded by A* since it is in the local search space. A* eventually terminated. Assume that A* would have expanded state s'_{goal} next if it had been allowed to expand one additional state. Let $g(s)$ be the g-values of A*. Let $h(s)$ be the h-values of A*, that is, before the subsequent execution of Dijkstra's algorithm. Finally, let $h'(s)$ be the h-values afterwards. Consider any path in the local search space from the current state of the agent via some state s' to some state s'' that borders the local search space. Assume that the formula $h'(s) = c(s,a) + h'(succ(s,a))$ is satisfied for all states s on the path (and the actions executed in them), starting at state s' or earlier. Let action a' be the action that is executed in state s'. The f-value of state s'' along this path is $g(s') + h'(s') = g(s') + c(s',a') + h'(succ(s',a')) = g(succ(s',a')) + h'(succ(s',a')) = \cdots = g(s'') + h'(s'') = g(s'') + h(s'') = f(s'')$ by induction. Now consider the path in the local search space that the agent follows from its current state to state s'_{goal}, that borders the local search space. We show that the formula is satisfied for all states on the path. Assume not. Let state s be the last state on the path that does not satisfy the formula and action a be the action that is executed in state s. The f-value of state s'_{goal} along the path in the local search space that the agent follows from its current state to state s'_{goal} is $f(s'_{\text{goal}}) = g(succ(s,a)) + h'(succ(s,a)) = g(s) + c(s,a) + h'(succ(s,a))$. Now consider the path in the local search space from the current state of the agent via state s to some state s' that borders the local search space, where the part of the path from the current state of the agent to state s is the path that the agent follows and the part of the path from state s to state s' satisfies the formula for all states, which is possible by executing an action in each state that satisfies Eq. 1. The f-value of state s' along this path is $f(s') = g(s) + h'(s)$. State s is in the local search space and it thus holds that $h'(s) = \min_{a' \in A(s)}(c(s,a') + h'(succ(s,a')) \leq c(s,a) + h'(succ(s,a))$. Since $h'(s) \neq c(s,a) + h'(succ(s,a))$ per assumption, it must be the case that $h'(s) < c(s,a) + h'(succ(s,a))$ and thus that $g(s) + h'(s) < g(s) + c(s,a) + h'(succ(s,a))$. Thus, the f-value of state s' is smaller than the f-value of state s'_{goal} and A* has to expand state s' before state s'_{goal}, which is a contradiction since state s' borders the local search space per assumption and is thus not expanded by A*. □

Theorem 3 *Let $S' \subseteq S$ be a superset of the states that the agent is going to visit if it stops when it reaches the goal state. Let $h(s)$ be the current h-value of each state $s \in S'$. Let mincost be the smallest current action cost of any action $a \in A(s)$ in any state $s \in S'$. Let $\overline{gd}(s)$ be an upper bound on all future goal distances of state $s \in S'$. Then, the agent reaches the goal state with at most $(\sum_{s' \in S'}(\overline{gd}(s') - h(s')) + h(s_{\text{start}}))/mincost$ action executions from its current state s_{start}.*

Theorem 3 states that the agent reaches the goal state with at most $\sum_{s \in S'} \overline{gd}(s)$ action executions from its current state if the smallest current action cost is one. This value can be infinity and the agent can execute actions forever. However, it is finite and the agent is then guaranteed to reach the goal state from its current state if the future goal distances of all states $s \in S'$ remain bounded from above (by a finite constant). This condition is satisfied for our navigation tasks if there exists a trajectory from the current cell to the goal cell since our grids are undirected and all action costs are either one or infinity. If the condition is satisfied, then the formula predicts that the agent reaches the goal cell with zero action executions from its current cell if it starts in the goal cell. Otherwise, the formula predicts that the agent reaches the goal cell with at most

$$\begin{aligned}
&\left(\sum_{s' \in S'} (\overline{gd}(s') - h(s')) + h(s_{\text{start}})\right) \Big/ mincost \\
&= \sum_{s' \in S'} (\overline{gd}(s') - h(s')) + h(s_{\text{start}}) \\
&\leq \sum_{s' \in S'} \overline{gd}(s') \\
&\leq \sum_{i=0}^{|S|-1} i \\
&= |S|^2/2 - |S|/2
\end{aligned}$$

action executions from its current cell. If the condition is not satisfied, then the agent can execute actions forever.

Proof We consider "the sum of the current h-values of all states in S' except for the current state of the agent" as potential. This potential changes only when the current h-values or the current state of the agent change. First, the h-values can change only during the execution of Dijkstra's algorithm. The potential cannot decrease during the execution of Dijkstra's algorithm since the h-values cannot decrease according to Theorem 1. Second, the potential increases by at least *mincost* with every action execution, for the following reason: Let x be the sum of the current h-values $h(s')$ of all states $s' \in S'$ except for the current state s of the agent before the execution of action $a \in A(s)$ and x' be the sum of the h-values $h(s')$ of all states $s' \in S'$ except for the current state $succ(s, a)$ of the agent afterwards. Let $c(s, a)$ be the current action cost of action $a \in A(s)$ during its execution. Then, $x = \sum_{s' \in S' \setminus \{s\}} h(s')$ and $x' = \sum_{s' \in S' \setminus \{succ(s,a)\}} h(s')$, implying that $x' - x = h(s) - h(succ(s, a)) = c(s, a) \geq mincost$ according to Lemma 1 and the fact that action costs never decrease. Now consider how the potential changes over time. It starts at $\sum_{s' \in S'} h(s') - h(s_{\text{start}})$ for the current state s_{start} of the agent and the current h-values $h(s)$. It is bounded from above by $\sum_{s' \in S'} \overline{gd}(s')$ since the h-values remain admissible according to Theorem 2. It cannot decrease and increases by at least $mincost > 0$ with each action execution. Thus, the agent reaches the goal state with at most $(\sum_{s' \in S'} \overline{gd}(s') - (\sum_{s' \in S'} h(s') - h(s_{\text{start}}))/mincost = (\sum_{s' \in S'} (\overline{gd}(s') - h(s')) + h(s_{\text{start}}))/mincost$ action executions from its current state. □

Theorem 4 *Assume that the agent is teleported back to the start state whenever it reaches the goal state. Let $S' \subseteq S$ be a superset of the states that the agent is going to visit. Then, the number of times that it does not follow a shortest trajectory from the start state to the goal* *state is bounded from above if all action cost increases are bounded from below by a positive constant and the goal distances of all states $s \in S'$ remain bounded from above.*

Theorem 4 holds due to the freespace assumption being optimistic and the h-values of LSS-LRTA* remaining consistent according to Theorem 2, which is due to LSS-LRTA* being based on LRTA* rather than real-time heuristic search methods that do not have this property, such as RTA* [29]. For our navigation tasks, the action costs can only increase from one to infinity. All action cost increases are thus indeed bounded from below by a positive constant.

Proof Assume for now that the agent always reaches the goal state and the action cost increases leave the goal distances $gd(s)$ of all states $s \in S'$ unchanged. We distinguish two cases. Case 1: The agent is in the start state and the h-values of all states that it is going to visit until it reaches the goal state are already equal to their respective goal distances. The h-values of the visited states could only increase according to Theorem 1 which would make them inadmissible, which is impossible according to Theorem 2. Thus, the h-values of the visited states can no longer change. The h-value of the current state of the agent is equal to the goal distance of the start state initially, decreases by the action cost of the executed action with each action execution according to Lemma 1, and is equal to zero when the agent reaches the goal state. Thus, the agent follows a trajectory whose length is equal to the goal distance of the start state, implying that the trajectory is a shortest trajectory from the start state to the goal state. Case 2: The agent is in the start state and the h-value at this point in time (= the initial h-value) of at least one state that the agent is going to visit until it reaches the goal state is not yet equal to its goal distance. Then, the agent executes at least once an action $a \in A(s)$ in a state $s \in S'$ so that the initial h-value of state s is not equal to the goal distance of state s but the initial h-value of state $succ(s, a)$ is equal to the goal distance of state $succ(s, a)$. This property holds because the agent reaches the goal state per assumption and the initial h-value of the goal state is zero and thus equal to its goal distance since the h-values remain consistent according to Theorem 2. We now prove that the h-value of state s is equal to its goal distance by the time the agent reaches the goal state. The h-value of state s is admissible according to Theorem 2 and thus a lower bound on its goal distance. The h-value of state s also satisfies $h(s) = c(s, a) + h(succ(s, a)) = c(s, a) + gd(succ(s, a)) \geq \min_{a' \in A(s)}(c(s, a') + gd(succ(s, a'))) = gd(s)$ according to Lemma 1 at the time the agent executes action a in state s and thus is an upper bound on its goal distance. Thus, the h-value of state s is indeed equal to and thus must have been set to its goal distance by the time the agent executes action a in state s. Once the h-value of state s is equal to its goal distance, it could only increase according to Theorem 1 which would make it inadmissible, which is impossible according to Theorem 2. Thus, the h-value can no longer change. Since the number of states is finite, the number of times that the h-value of a state is set to its goal distance is bounded from above. Thus, the number of times that the agent does not follow a shortest trajectory from the start state to the goal state is bounded from above. The theorem then follows since the number of times that action cost increases do not leave the goal distances of all states $s \in S'$ unchanged is bounded from above (since the action cost increases are bounded from below by a positive constant but the goal distances of all states $s \in S'$ remain bounded from above per assumption) and the number of times that the agent does not follow a shortest trajectory from the start state to the goal state in between these times is also bounded from above (as shown above). If the agent does not reach the goal state, then this is the last time that it does not follow a shortest trajectory from the start state to the goal state. Thus, the number of times that it does not follow a shortest trajectory from the start state to the goal state is even smaller. □

4 Experimental comparison of LSS-LRTA* and D* Lite

The ideas behind real-time and incremental heuristic search could, in principle, be combined by restricting the A* search to the part of the state space around the current state of the agent and reusing information from the previous A* search to speed up the current one. However, incremental heuristic search methods based on D* Lite require the start of the search to remain unchanged. Thus, they search from the goal state toward the current state of the agent, which does not work in conjunction with real-time heuristic search although it has been tried once to combine incremental and real-time heuristic search [9]. The research issue then is when to use real-time heuristic search and when to use incremental heuristic search. Incremental heuristic search has advantages over real-time heuristic search: Since incremental heuristic search finds complete paths, it can easily discover that the goal state cannot be reached with finite path length from the current state of the agent. On the other hand, real-time heuristic search also has advantages over incremental heuristic search: Since real-time heuristic search finds only the beginning of complete paths it can satisfy hard real-time requirements in state spaces of any size by choosing small local search spaces (potentially at the expense of increasing the number of action executions) while the search time per search of incremental heuristic search can increase with the size of the state spaces.

We therefore compare LSS-LRTA*, a representative real-time heuristic search method, against the optimized final version of D* Lite as published in [21], a representative incremental heuristic search method. Their runtimes depend not only on the experimental setup but also on other factors. This point is especially important since the terrain in real-time computer games typically fits into memory and the resulting state spaces are thus small but the searches have to be fast, especially if the number of characters is large. Thus, the scaling behavior of the heuristic search methods is less important than the hardware and implementation details, including the data structures, tie-breaking strategies, and coding tricks used. Indeed, we noticed during our experiments that small coding details can be important. For example, it is beneficial for LSS-LRTA* with small lookaheads to generate the successors of states during its A* searches in random order rather than a fixed one. We do not know of any better method for evaluating heuristic search methods than to implement them as best as possible, publish their runtimes, and let other researchers validate the experimental results with their own and thus potentially slightly different implementations. For example, it is difficult to compare LSS-LRTA* and D* Lite with proxies, such as the number of state expansions, instead of the search time itself since they perform different basic operations. For fairness, we use comparable implementations of LSS-LRTA* and D* Lite. For example, we use standard binary heaps to implement the *OPEN* lists of both heuristic search methods.

We use different kinds of grids as test domain. Figure 9 shows a game map from Baldur's Gate II from BioWare. Such game maps typically have different areas of different kinds, including areas with twisted passages and wide open areas with small obstacles. These kinds of areas have different properties. For example, h-values can be misleading in areas with twisted passages. On the other hand, they are generally not misleading in wide open areas with small obstacles. Agents generally cannot guarantee to follow short trajectories to the goal cells in areas with twisted passages both because the shortest paths to the goal cells are long and because the agents might have to try many paths before they find unblocked paths to the goal cells. On the other hand, agents are generally able to follow short trajectories to the goal cells in wide open areas with small obstacles both because the shortest paths to the goal cells are short and because the agents are generally able to move in the direction of their goal cells by circumnavigating obstacles that are in their way. We expect the kind of area to be important for characterizing when each heuristic search method should be used. We

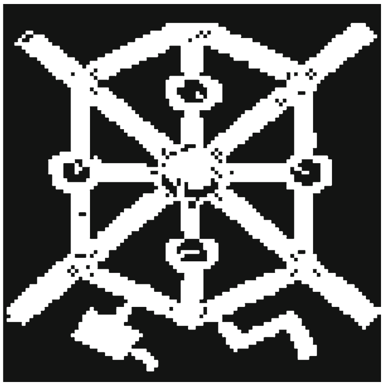


Fig. 9 Baldur's Gate II from BioWare

therefore use two different kinds of grids that are ideal versions of these different kinds of areas, namely mazes, that are similar to areas with twisted passages, and grids with random obstacles, that are similar to wide open areas with small obstacles. In all cases, we average over 1000 (for mazes) and 5000 (for grids with random obstacles) randomly generated undirected four-neighbor grids of size 301×301 with randomly chosen start cells and goal cells with the restriction that there exists a trajectory from the start cell to the goal cell since agents using LSS-LRTA* are then guaranteed to reach the goal cell according to Theorem 3. In the following, we describe our experimental results in detail. We measure all times in microseconds and all distances in number of action executions.

4.1 Mazes

We first use acyclic mazes as test domain whose corridor structure is generated with depth-first search. Figure 10 (left) shows an example of smaller size than used in the experiments. As h-value of a cell we use the sum of the absolute difference of its x and y coordinates to the x and y coordinates of the goal cell (= strong h-values), as shown in Fig. 1 (left). We also use the maximum of the absolute difference of its x and y coordinates to the x and y coordinates of the goal cell (= weak h-values), as shown in Fig. 1 (right). The strong and weak h-values are both consistent. We expect A* with the strong h-values to have smaller search times than A* with the weak h-values since the strong h-values dominate the weak h-values and A* with the strong h-values thus expands no more cells than A* with the weak h-values (but likely expands fewer cells). We also expect LSS-LRTA* or D* Lite with either kind of h-values to find trajectories of comparable lengths since both kinds of h-values are consistent and A* with either kind of h-values thus finds shortest paths. D* Lite is an incremental version of A*. Table 1 shows that D* Lite with the strong h-values also has smaller search times than D* Lite with the weak h-values and finds trajectories of comparable lengths. (The table shows smaller search times and trajectory lengths in bold to make it easier to interpret the data.) The small difference is due to the randomized tie breaking. However, the table also shows that this is not the case for LSS-LRTA*. LSS-LRTA* with the strong h-values tends to have larger search times than LSS-LRTA* with the weak h-values and find longer trajectories, at least for small lookaheads. This effect is due to the relative differences in h-values being much more important for focusing the A* search of LSS-LRTA* than their absolute differences. Because the h-values can be misleading in mazes, it is better if the differences of the h-values

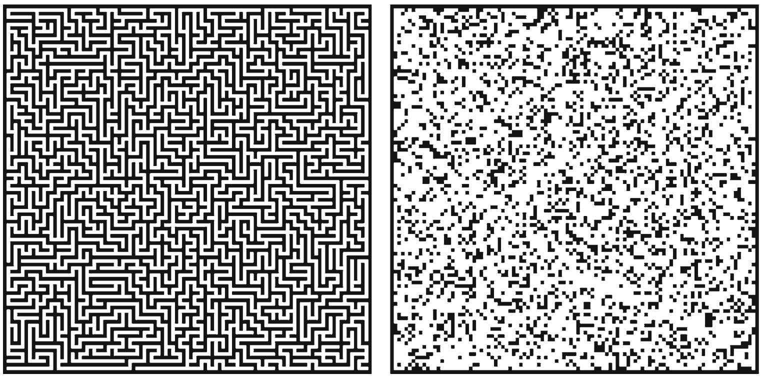


Fig. 10 Maze (left) and grid with random obstacles (right)

Table 1 Different h-values in mazes

Lookahead[a]	Strong h-values		Weak h-values	
	Search time[b]	Trajectory length[c]	Search time[b]	Trajectory length[c]
D Lite*				
–	**357417.38**	21737.53	373560.62	**21140.40**
*LSS-LRTA**				
1	985361.73	1987574.25	**628175.97**	**1259958.00**
3	640567.24	931230.38	**477551.83**	**685570.04**
5	441522.49	594675.73	**366611.13**	**477525.03**
7	395581.98	499083.52	**321784.17**	**382949.65**
9	358532.52	422475.56	**296545.64**	**321547.69**
11	313997.99	337704.16	**277974.35**	**272841.73**
13	302791.98	303562.40	**276238.32**	**252374.66**
15	290252.67	268652.32	**281280.60**	**239072.72**
17	284827.58	243952.42	**277453.57**	**215615.93**
19	**276990.44**	217852.00	280483.72	**199517.41**
21	279855.69	205370.41	**273280.22**	**177142.96**
23	285321.52	196601.90	**283999.33**	**171600.71**
25	**274999.82**	169685.19	283950.17	**155736.31**
27	293554.47	176642.26	**292767.20**	**151277.34**
29	**293262.79**	163418.55	296116.07	**140895.33**
...	...	...	...	...

[a] Cell expansions per search (= lookahead)
[b] Average search time with smaller numbers shown in bold
[c] Average number of action executions (= trajectory length) with smaller numbers shown in bold

are small because LSS-LRTA* can then correct the h-values faster. A similar phenomenon had previously been described for searching the eight-puzzle with LRTA* [19]. We use LSS-LRTA* and D* Lite with those h-values that work best for them. Thus, in mazes, we use LSS-LRTA* with the weak h-values and D* Lite with the strong h-values.

Table 2 shows our experimental results.[1] We mentioned that it is difficult to compare LSS-LRTA* and D* Lite with proxies, such as the number of cell expansions, instead of the

[1] The number of cell expansions can be smaller than the product of the lookahead and the number of searches because, around the goal cell, the number of cell expansions per search is smaller than the lookahead since A* terminates once it is about to expand the goal cell.

Table 2 LSS-LRTA* and D* Lite in mazes

Lookahead[a]	Cell expansions[b]	Searches[c]	Trajectory length[d]	Trajectory length per search[e]	Search time[f]	Search time per search[g]	Search time per action[h]
D Lite*							
–	230893.54	6606.37	21737.53	3.29	357417.38	54.10	16.44
*LSS-LRTA**							
1	1259958.00	1259958.00	1259958.00	1.00	628175.97	0.50	0.50
3	1012633.01	337544.61	685570.04	2.03	477551.83	1.41	0.70
5	765644.80	153129.55	477525.03	3.12	366611.13	2.39	0.77
7	658618.41	94089.35	382949.65	4.07	321784.17	3.42	0.84
9	588810.14	65424.97	321547.69	4.91	296545.64	4.53	0.92
11	531955.23	48361.94	272841.73	5.64	277974.35	5.75	1.02
13	518431.33	39882.58	252374.66	6.33	276238.32	6.93	1.09
15	517913.09	34531.72	239072.72	6.92	281280.60	8.15	1.18
17	495466.48	29150.37	215615.93	7.40	277453.57	9.52	1.29
19	487622.82	25670.60	199517.41	7.77	280483.72	10.93	1.41
21	459565.74	21891.42	177142.96	8.09	273280.22	12.48	1.54
23	470419.04	20461.61	171600.71	8.39	283999.33	13.88	1.66
25	456751.93	18279.86	155736.31	8.52	283950.17	15.53	1.82
27	465707.78	17259.47	151277.34	8.76	292767.20	16.96	1.94
29	460964.20	15907.64	140895.33	8.86	296116.07	18.61	2.10
31	469144.66	15147.32	135554.16	8.95	310131.20	20.47	2.29
33	460947.12	13983.02	125789.79	9.00	304691.69	21.79	2.42
35	474447.93	13571.97	123304.98	9.09	315807.85	23.27	2.56
37	492481.84	13327.82	122274.35	9.17	329287.75	24.71	2.69
39	514415.51	13209.17	122839.82	9.30	344388.42	26.07	2.80
41	512638.46	12523.62	114917.08	9.18	348330.44	27.81	3.03
43	517674.55	12060.69	111242.82	9.22	354049.77	29.36	3.18
45	507532.55	11301.47	100257.67	8.87	354495.33	31.37	3.54
47	532693.30	11358.48	103038.69	9.07	370009.76	32.58	3.59
49	555105.16	11355.05	104059.85	9.16	385354.53	33.94	3.70

[a] Cell expansions per search (= lookahead)
[b] Average number of cell expansions
[c] Average number of searches
[d] Average number of action executions (= trajectory length)
[e] Average number of action executions per search (= trajectory length per search)
[f] Average search time
[g] Average search time per search
[h] Average search time per action execution

search time itself since they perform different basic operations, which forces us to compare their search times directly. However, Fig. 11 shows that the search time of LSS-LRTA* with different lookaheads appears to be roughly proportional to its number of cell expansions, which gives us hope that different hardware and implementation details change the search time of LSS-LRTA* by only a constant factor.

Table 2 shows some interesting trends: First, the trajectory length of LSS-LRTA* decreases as its lookahead increases: more search results in shorter trajectories, which confirms earlier experimental results in different domains [29] although exceptions to this property have also been reported [5]. Second, the search time of LSS-LRTA* first decreases and then increases as its lookahead increases. This is the result of two different effects, namely an increasing search time per search due to the larger lookahead and a decreasing number of searches due to the larger local search spaces, which result in both a larger number of action executions per search and shorter trajectories. Figures 12 and 13 visualize this trade-off between the search

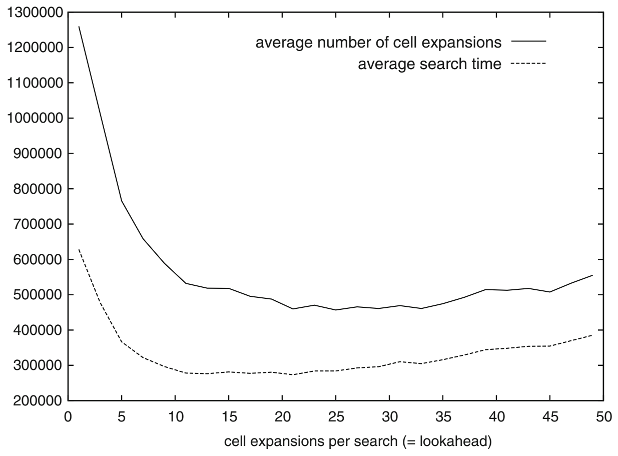


Fig. 11 Search time and number of cell expansions of LSS-LRTA* in mazes

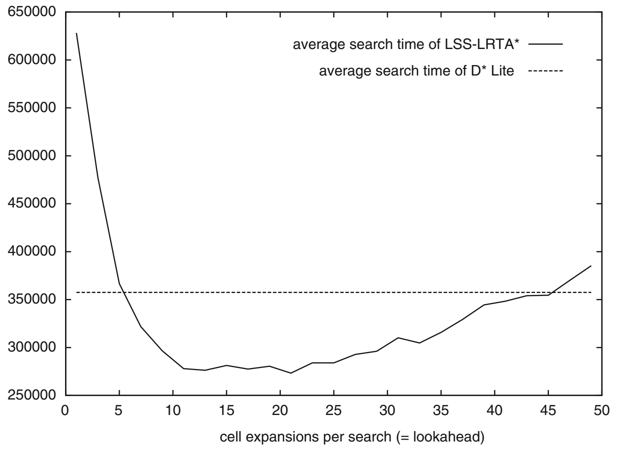


Fig. 12 Search times of LSS-LRTA* and D* Lite in mazes

time and the resulting trajectory length. The search time of LSS-LRTA* is larger than the one of D* Lite for lookaheads larger than 45, and its trajectories are longer than the ones of D* Lite for all tabulated lookaheads. Searching all the way to the goal cell is important for finding short trajectories because the h-values can be misleading in mazes, Thus, the trajectories of D* Lite are short while the trajectories of LSS-LRTA* are short only for large lookaheads. LSS-LRTA* with small lookaheads moves the agent back and forth in local minima of the h-value surface until it has increased the h-values of the cells sufficiently to be able to escape them, which results in long and also less believable trajectories.

In the following, we describe three search objectives and analyze which heuristic search method one should choose for each one. We assume that one cannot overlap searches and action executions and thus has to interleave them.

– We first study what to do if one wants to minimize the sum of the search and action-execution time. If search is fast relative to action execution (a realistic assumption for

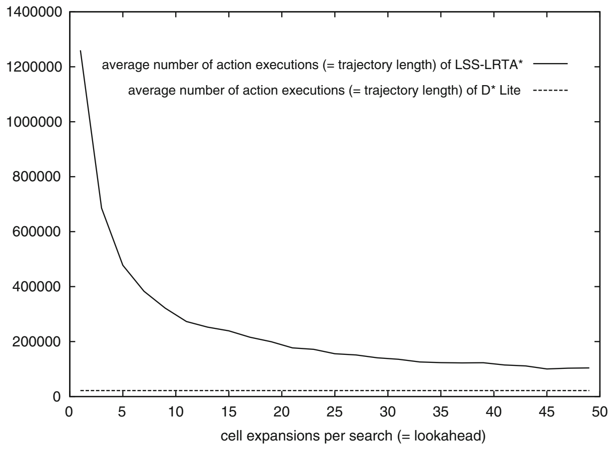


Fig. 13 Trajectory lengths of LSS-LRTA* and D* Lite in mazes

many applications, including real-time computer games), then the sum of the search and action-execution time is determined by the action-execution time, that is, trajectory length (x is large, see below). In this case, a large lookahead is optimal for LSS-LRTA* but then LSS-LRTA* is slower than D* Lite since it does not use information from the previous A* search to speed up the current one. Thus, D* Lite should be used instead of LSS-LRTA*. On the other hand, if action execution is fast relative to search, then the sum of the search and action-execution time is determined by the search time (x is small). In this case, a lookahead of 21 is optimal for LSS-LRTA*, as shown in Fig. 12. To understand the cases between these extremes better, let $x > 0$ be the ratio of the search speed and action-execution speed. The sum of the search and action-execution time is then proportional to: search time + $x \times$ trajectory length. Table 3 is derived from Table 2 and shows the optimal lookahead of LSS-LRTA* as a function of x, with intuitive results that follow from Figs. 12 and 13: Lookaheads that are smaller than the lookahead that minimizes the search time cannot be optimal for LSS-LRTA* since both its search time and its trajectory length decrease as its lookahead increases. The lookahead that minimizes its search time (namely, a lookahead of 21) is optimal for $x = 0$. Its search time increases and its trajectory length decreases as the lookahead increases, starting with the lookahead that minimizes its search time. Thus, its optimal lookahead increases as x increases. If x is larger than $10^{-0.27}$, then D* Lite should be used instead of LSS-LRTA* with the optimal lookahead since LSS-LRTA* finds trajectories of comparable lengths with a much larger search time than D* Lite.

Table 3 Optimal lookaheads for LSS-LRTA* in mazes [x = ratio of search and action-execution speed]

Range of x	Optimal lookahead of LSS-LRTA*
0 to $10^{-0.31}$	21
$10^{-0.30}$ to $10^{-0.16}$	25
$10^{-0.15}$ to $10^{+0.29}$	33
…	…

- We now study what to do if one wants to minimize the trajectory length subject to the hard real-time requirement that only a certain amount of time is available for each search. Thus, there is a time limit on the search time per search. We argue (slightly incorrectly) with averages in the following to make our argument simple. D* Lite has a search time per search of 54.10 and thus cannot be used if the time limit is smaller than this. Thus, LSS-LRTA* with the largest lookahead that fits the time limit should be used instead of D* Lite if the time limit is smaller than 54.10. LSS-LRTA* with a lookahead of up to 75 (not shown in Table 2) has a search time per search that is smaller than 54.10. On the other hand since LSS-LRTA* with a lookahead of even 241 (not shown in Table 2) finds only trajectories of length 44721.30 (compared to 21737.53 for D* Lite) but has a search time per search of 168.19, D* Lite should be used if the time limit is larger than 54.10.
- Finally, we study what to do if one wants to minimize the trajectory length subject to the hard real-time requirement that only a certain amount of time is available for each search but the search time can be amortized over the action executions. Thus, there is a time limit on the search time per action execution. D* Lite has a search time per action execution of 16.44 and thus cannot be used if the time limit is smaller than this. Thus, LSS-LRTA* with the largest lookahead that fits the time limit should be used instead of D* Lite if the time limit is smaller than 16.44. LSS-LRTA* with a lookahead of up to 165 (not shown in Table 2) has a search time per action execution that is smaller than the one of D* Lite. On the other hand since LSS-LRTA* with a lookahead of even 241 finds only trajectories of length 44721.30 (compared to 21737.53 for D* Lite) but has a search time per action execution of 28.61, D* Lite should be used if the time limit is larger than 16.44.

So far, our experimental results are not surprising. LSS-LRTA* with small lookaheads should be used instead of D* Lite if there is not enough time to find complete paths. Otherwise, D* Lite should be used instead of LSS-LRTA* since it uses information from the previous search to speed up the current one. However, there are additional benefits to LSS-LRTA* if the h-values are generally not misleading, as is the case in grids with random obstacles, discussed in the next section.

Table 4 shows experimental results for a version of LSS-LRTA* that generates its local search spaces with breadth-first search (= BFS) rather than A*, which has a smaller search time per search because one can implement a breadth-first search with a first-in first-out queue rather than a priority queue. The table shows that LSS-LRTA* with breadth-first search tends to expand fewer cells than LSS-LRTA* with A* and find shorter trajectories, at least for small lookaheads (different from three). (The table shows smaller numbers of cell expansions and trajectory lengths in bold to make it easier to interpret the data.) This result is due to the fact that the h-values can be misleading in mazes and then result local search spaces of suboptimal shapes.

4.2 Grids with random obstacles

We now use grids with randomly placed blocked cells as test domain. Their obstacle density is 25%. Figure 10 (right) shows an example of smaller size than used in the experiments. Testing LSS-LRTA* and D* Lite on these different kinds of grids is interesting because they have different properties from mazes. They are more difficult than mazes since their branching factor is larger. They are easier than mazes since the shortest paths to the goal cells tend to be shorter, agents are generally able to find such paths by moving in the direction of their goal cells by circumnavigating blocked cells that are in their way, and the h-values are generally

Table 4 Different search spaces in mazes

Lookahead[a]	LSS-LRTA* (with A*)		LSS-LRTA* with BFS	
	Cell expansions[b]	Trajectory length[c]	Cell expansions[b]	Trajectory length[c]
1	1259958.00	1259958.00	**1244573.34**	**1244573.34**
3	**1012633.01**	**685570.04**	2151181.27	1427453.49
5	765644.80	477525.03	**608563.97**	**339733.20**
7	658618.41	382949.65	**470885.88**	**239418.18**
9	588810.14	321547.69	**439573.64**	**204921.68**
11	531955.23	272841.73	**437526.96**	**189936.61**
13	518431.33	252374.66	**410861.79**	**165348.26**
15	517913.09	239072.72	**460207.28**	**177181.12**
17	495466.48	215615.93	**430183.56**	**154345.68**
19	487622.82	199517.41	**453565.17**	**154292.26**
21	459565.74	177142.96	**448383.49**	**144253.88**
23	470419.04	171600.71	**470230.40**	**144736.66**
25	**456751.93**	155736.31	473433.00	**138034.91**
27	**465707.78**	151277.34	483322.87	**135636.58**
29	**460964.20**	140895.33	499253.40	**133028.67**
…	…	…	…	…

[a] Cell expansions per search (= lookahead)
[b] Average number of cell expansions (as an imperfect indicator for the average search time since breadth-first search can expand cells faster than A*) with smaller numbers shown in bold
[c] Average number of action executions (= trajectory length) with smaller numbers shown in bold

Table 5 Different h-values in grids with random obstacles

Lookahead[a]	Strong h-values		Weak h-values	
	Search time[b]	Trajectory length[c]	Search time[b]	Trajectory length[c]
D Lite*				
–	**36825.63**	**308.98**	40737.34	313.78
*LSS-LRTA**				
1	**28279.51**	498.55	28292.81	**363.16**
3	**28380.11**	377.15	28446.67	**363.30**
5	**28435.03**	**337.67**	28568.88	339.47
7	**28536.61**	329.00	28658.00	**327.50**
9	**28617.42**	322.19	28769.02	**318.39**
11	**28698.35**	**315.32**	28877.77	**315.32**
13	**28785.79**	**310.35**	29008.12	315.57
15	**28873.00**	**307.15**	29118.05	314.14
17	**28966.89**	**305.47**	29226.43	311.46
19	**29056.67**	**303.58**	29341.93	311.65
21	**29152.59**	**302.27**	29476.83	311.11
23	**29241.04**	**301.54**	29585.83	310.20
25	**29332.52**	**300.77**	29701.80	309.44
27	**29428.39**	**300.24**	29822.47	309.43
29	**29524.39**	**299.44**	29949.31	309.84
…	…	…	…	…

[a] Cell expansions per search (= lookahead)
[b] Average search time with smaller numbers shown in bold
[c] Average number of action executions (= trajectory length) with smaller numbers shown in bold

not misleading. Table 5 shows that LSS-LRTA* with the weak h-values now tends to have larger search times than LSS-LRTA* with the strong h-values and find longer trajectories, at least for large lookaheads. (The table shows smaller search times and trajectory lengths in bold to make it easier to interpret the data.) This effect is due to the h-values generally not being misleading. Thus, in grids with random obstacles, we use both LSS-LRTA* and D* Lite with the strong h-values. Table 6 shows our experimental results, which are similar to the ones in mazes. The trajectory length of LSS-LRTA* again decreases as its lookahead increases but its search time now increases right away. Figures 14 and 15 visualize these properties of the search time and the trajectory length. The search times per search of both LSS-LRTA* and D* Lite are larger than in mazes due to the larger branching factor. The search time of LSS-LRTA* is now smaller than the one of D* Lite for all tabulated lookaheads although eventually its search time is larger than the one of D* Lite as the lookahead

Table 6 LSS-LRTA* and D* Lite in grids with random obstacles

Lookahead[a]	Cell expansions[b]	Searches[c]	Trajectory length[d]	Trajectory length per search[e]	Search time[f]	Search time per search[g]	Search time per action[h]
D Lite*							
–	11424.90	72.54	308.98	4.26	36825.63	507.65	119.18
*LSS-LRTA**							
1	498.55	498.55	498.55	1.00	28279.51	56.72	56.72
3	622.46	207.83	377.15	1.81	28380.11	136.56	75.25
5	686.46	137.77	337.67	2.45	28435.03	206.39	84.21
7	796.09	114.30	329.00	2.88	28536.61	249.66	86.74
9	902.13	100.92	322.19	3.19	28617.42	283.58	88.82
11	1013.99	92.98	315.32	3.39	28698.35	308.65	91.01
13	1128.42	87.72	310.35	3.54	28785.79	328.14	92.75
15	1238.49	83.63	307.15	3.67	28873.00	345.25	94.00
17	1353.46	80.83	305.47	3.78	28966.89	358.36	94.83
19	1464.02	78.40	303.58	3.87	29056.67	370.61	95.71
21	1578.59	76.68	302.27	3.94	29152.59	380.16	96.45
23	1701.07	75.63	301.54	3.99	29241.04	386.62	96.97
25	1822.36	74.75	300.77	4.02	29332.52	392.40	97.53
27	1953.37	74.38	300.24	4.04	29428.39	395.67	98.02
29	2077.55	73.85	299.44	4.05	29524.39	399.76	98.60
31	2202.06	73.40	299.20	4.08	29615.10	403.48	98.98
33	2323.90	72.99	298.70	4.09	29715.34	407.11	99.48
35	2438.37	72.38	297.90	4.12	29799.11	411.73	100.03
37	2569.27	72.30	298.38	4.13	29904.07	413.58	100.22
39	2683.20	71.80	297.61	4.14	29994.63	417.73	100.78
41	2803.77	71.60	297.65	4.16	30094.02	420.30	101.10
43	2925.34	71.35	297.39	4.17	30193.43	423.15	101.53
45	3053.07	71.36	297.81	4.17	30303.25	424.66	101.75
47	3170.40	71.14	297.48	4.18	30391.28	427.23	102.16
49	3285.16	70.87	297.27	4.19	30484.89	430.14	102.55

[a] Cell expansions per search (= lookahead)
[b] Average number of cell expansions
[c] Average number of searches
[d] Average number of action executions (= trajectory length)
[e] Average number of action executions per search (= trajectory length per search)
[f] Average search time
[g] Average search time per search
[h] Average search time per action execution

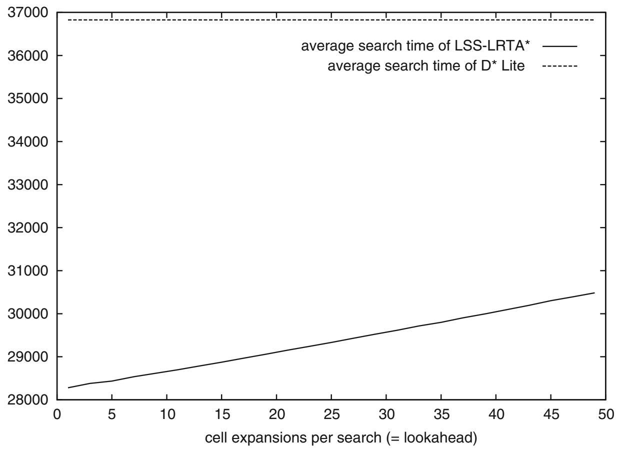


Fig. 14 Search times of LSS-LRTA* and D* Lite in grids with random obstacles

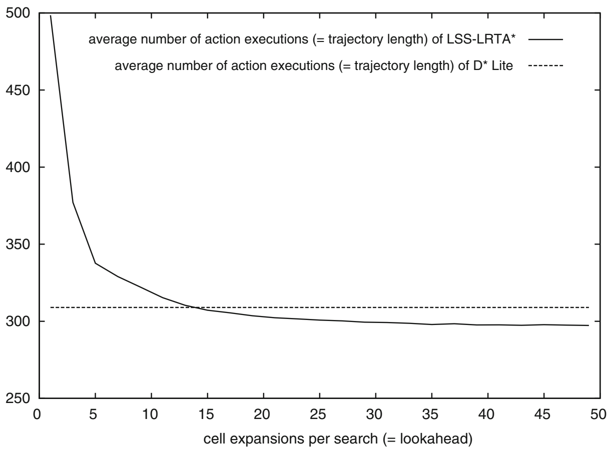


Fig. 15 Trajectory lengths of LSS-LRTA* and D* Lite in grids with random obstacles

increases since it does not use information from the previous search to speed up the current one, and its trajectories are shorter than the ones of D* Lite for all tabulated lookaheads larger than 13. These good experimental results for LSS-LRTA* are due to the fact that the h-values are generally not misleading. One can therefore reduce the search time by greedily performing steepest descent on the h-value surface, with some lookahead to avoid being misled by inaccuracies of the h-values caused by obstacles. This means that it is unnecessary to search all the way to the goal cell. Larger lookaheads then decrease the trajectory length only marginally.

We now study again what to do if one wants to minimize the sum of the search and action-execution time. Table 7 is derived from Table 6 and shows the optimal lookahead of LSS-LRTA* as a function of x, the ratio of the search speed and action-execution speed. A lookahead of one now minimizes the search time of LSS-LRTA*, which is smaller than the

Table 7 Optimal lookaheads for LSS-LRTA* in grids with random obstacles [x = ratio of search and action-execution speed]

Range of x	Optimal lookahead of LSS-LRTA*
0 to $10^{-0.09}$	1
$10^{+0.08}$ to $10^{+0.14}$	3
$10^{+0.15}$ to $10^{+1.06}$	5
$10^{+1.07}$ to $10^{+1.07}$	7
$10^{+1.08}$ to $10^{+1.24}$	11
$10^{+1.25}$ to $10^{+1.43}$	13
$10^{+1.44}$ to $10^{+1.71}$	15
$10^{+1.72}$ to $10^{+1.86}$	19
$10^{+1.87}$ to $10^{+2.07}$	21
$10^{+2.08}$ to $10^{+2.15}$	25
$10^{+2.16}$ to $10^{+2.25}$	29
$10^{+2.26}$ to $10^{+2.82}$	35
$10^{+2.83}$ to $10^{+2.95}$	39
$10^{+2.96}$ to $10^{+3.38}$	43
…	…

lookahead of 21 that minimized the search time of LSS-LRTA* in mazes. For all values of x, LSS-LRTA* with the optimal lookahead has a smaller sum of search and action-execution time than D* Lite, and thus LSS-LRTA* with the optimal lookahead should always be used instead of D* Lite since D* Lite finds longer trajectories with a larger search time than LSS-LRTA*. This is not surprising since both the search time and the resulting trajectories of LSS-LRTA* with lookaheads larger than 13 are shorter than the ones of D* Lite, as shown in Figs. 14 and 15. Thus, even if search is fast relative to action executions (a realistic assumption for many applications, including real-time computer games), LSS-LRTA* should always be used instead of D* Lite. For the same reason, LSS-LRTA* should always be used instead of D* Lite if there is a time limit on the search time per search or action execution.

Our experimental results are surprising. Clearly, LSS-LRTA* with small lookaheads should be used instead of D* Lite if there is not enough time to find complete paths. However, it is surprising that LSS-LRTA* should be used instead of D* Lite in grids with random obstacles regardless of the time limit. We thus achieved the objective behind our design of LSS-LRTA* and demonstrated that real-time heuristic search can outperform incremental heuristic search if the h-values are generally not misleading.

It currently remains unclear why the trajectories of LSS-LRTA* with lookaheads larger than 13 are shorter than the ones of D* Lite. We would have expected them to be short but slightly longer than the ones of D* Lite, in which case there could have been a trade-off between LSS-LRTA* and D* Lite. However, switching from D* Lite to LSS-LRTA* would still have reduced the search time by 20% or more while increasing the trajectory length only slightly.

Table 8 shows experimental results for a version of LSS-LRTA* that generates its local search spaces with breadth-first search rather than A*. The table shows that LSS-LRTA* with A* now tends to expand fewer cells than LSS-LRTA* with breadth-first search and find shorter trajectories since the h-values are generally not misleading. (The table shows smaller numbers of cell expansions and trajectory lengths in bold to make it easier to interpret the data.) Again, we thus achieved the objective behind our design of LSS-LRTA* and demonstrated that real-time heuristic search with local search spaces determined by A* can

Table 8 Different search spaces in grids with random obstacles

Lookahead[a]	LSS-LRTA* (with A*)		LSS-LRTA* with BFS	
	Cell expansions[b]	Trajectory length[c]	Cell expansions[b]	Trajectory length[c]
1	498.55	498.55	**496.82**	**496.82**
3	**622.46**	**377.15**	751.35	382.46
5	**686.46**	**337.67**	883.16	340.95
7	**796.09**	**329.00**	1081.67	331.05
9	**902.13**	322.19	1224.09	**322.09**
11	**1013.99**	**315.32**	1377.21	317.84
13	**1128.42**	**310.35**	1554.31	316.33
15	**1238.49**	**307.15**	1716.99	313.97
17	**1353.46**	**305.47**	1871.26	312.06
19	**1464.02**	**303.58**	2020.39	310.60
21	**1578.59**	**302.27**	2169.39	309.72
23	**1701.07**	**301.54**	2315.88	308.09
25	**1822.36**	**300.77**	2465.16	307.65
27	**1953.37**	**300.24**	2605.35	306.62
29	**2077.55**	**299.44**	2763.52	306.93
…	…	…	…	…

[a] Cell expansions per search (= lookahead)
[b] Average number of cell expansions (as an imperfect indicator for the average search time since breadth-first search can expand cells faster than A*) with smaller numbers shown in bold
[c] Average number of action executions (= trajectory length) with smaller numbers shown in bold

expand fewer cells than real-time heuristic search with local search spaces determined by breadth-first search if the h-values are generally not misleading.

5 Conclusions

In this article, we compared two classes of fast heuristic search methods for real-time situated agents that speed up A* searches in different ways, namely real-time heuristic search and incremental heuristic search, to understand their advantages and disadvantages and make recommendations about when each one should be used. We used navigation tasks as test domain where a real-time situated agent, such as a character in a real-time computer game, has to move autonomously from its current cell to a goal cell without knowing in advance which cells are blocked. We developed a novel version of Learning Real-Time A*. LSS-LRTA* is a real-time heuristic search method that uses A* to determine its local search spaces and learns quickly. We analyzed its properties and then compared it experimentally against the incremental heuristic search method D* Lite [21] on our navigation tasks, for which D* Lite was specifically developed, resulting in the first comparison of real-time and incremental heuristic search in the literature. Table 9 summarizes our conclusions with respect to LSS-LRTA*. We characterized when to choose which one of the two heuristic search methods, depending on the kind of terrain (which determines how informed the h-values are) and the search objective. Our experimental results are surprising since LSS-LRTA* can outperform D* Lite under the right conditions, namely when there is time pressure or the user-supplied h-values are generally not misleading. It is future work to extend our experiments to real-time heuristic search methods that are even less computationally intensive than LSS-LRTA*, especially since our experimental results show that less computationally intensive heuristic

Table 9 Properties of LSS-LRTA* in mazes and grids with random obstacles

Mazes	Grids with random obstacles
The h-values can be misleading, resulting in deep local minima	The h-values are generally not misleading, resulting in shallow local minima
LSS-LRTA* with weak h-values results in smaller search times and trajectory lengths than LSS-LRTA* with strong h-values	LSS-LRTA* with strong h-values results in smaller search times and trajectory lengths than LSS-LRTA* with weak h-values
LSS-LRTA* with small lookaheads results in relatively long trajectories	LSS-LRTA* with small lookaheads results already in relatively short trajectories
LSS-LRTA* that determines the local search spaces with breadth-first search results in fewer cell expansions and shorter trajectories (for small lookaheads) than LSS-LRTA* that determines the local search spaces with A*	LSS-LRTA* that determines the local search spaces with A* results in fewer cell expansions and shorter trajectories than LSS-LRTA* that determines the local search spaces with breadth-first search

search methods can outperform more computationally intensive heuristic search methods if the h-values are generally not misleading.

Acknowledgements We thank Xiaoming Zheng for his suggestions regarding one of the proofs. We thank Vadim Bulitko for making a map from the real-time game Baldur's Gate II available to us. This research has been partly supported by an NSF award to Sven Koenig under contract IIS-0350584. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, companies or the U.S. government.

References

1. Barto, A., Bradtke, S., & Singh, S. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence, 73*(1), 81–138.
2. Bjornsson, M., Enzenberger, M., Holte, R., Schaeffer, J., & Yap, P. (2003). Comparison of different abstractions for pathfinding on maps. In *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 1511–1512).
3. Bonet, B., & Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling* (pp. 52–61).
4. Bonet, B., Loerincs, G., & Geffner, H. (1997). A robust and fast action selection mechanism. In *Proceedings of the National Conference on Artificial Intelligence* (pp. 714–719).
5. Bulitko, V. (2003). Lookahead pathologies and meta-level control in real-time heuristic search. In *Proceedings of the Euromicro Conference on Real-Time Systems* (pp. 13–16).
6. Bulitko, V., Bjornsson, Y., Luvstrek, M., Schaeffer, J., & Sigmundarson, S. (2007). Dynamic control in path-planning with real-time heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling* (pp. 49–56).
7. Bulitko, V., & Lee, G. (2006). Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research, 25*, 119–157.
8. Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik, 1*, 269–271.
9. Edelkamp, S. (1998). Updating shortest paths. In *Proceedings of the European Conference on Artificial Intelligence* (pp. 655–659).
10. Furcy, D., & Koenig, S. (2000). Speeding up the convergence of real-time search. In *Proceedings of the National Conference on Artificial Intelligence* (pp. 891–897).
11. Goldenberg, M., Kovarksy, A., Wu, X., & Schaeffer, J. (2003). Multiple agents moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 1538–1538).
12. Hart, P., Nilsson, N., & Raphael, N. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics, SSC4*(2), 100–107.

13. Hart, P., Nilsson, N., & Raphael, B. (1972). Correction to 'a formal basis for the heuristic determination of minimum cost paths'. *SIGART Newsletter, 37*, 28–29.
14. Hebert, M., McLachlan, R., & Chang, P. (1999). Experiments with driving modes for urban robots. In *Proceedings of the SPIE Mobile Robots*.
15. Holte, R., Mkadmi, T., Zimmer, R., & MacDonald, A. (1996). Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence, 85*(1–2), 321–361.
16. Ishida, T. (1992). Moving target search with intelligence. In *Proceedings of the National Conference on Artificial Intelligence* (pp. 525–532).
17. Ishida, T. (1997). *Real-Time search for learning autonomous agents*. Kluwer Academic Publishers.
18. Koenig, S. (2001). Agent-centered search. *Artificial Intelligence Magazine, 22*(4), 109–131.
19. Koenig, S. (2001). Minimax real-time heuristic search. *Artificial Intelligence, 129*, 165–197.
20. Koenig, S. (2004). A comparison of fast search methods for real-time situated agents. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems* (pp. 864–871).
21. Koenig, S., & Likhachev, M. (2002). D* Lite. In *Proceedings of the National Conference on Artificial Intelligence* (pp. 476–483).
22. Koenig, S., & Likhachev, M. (2005). Fast replanning for navigation in unknown terrain. *Transactions on Robotics, 21*(3), 354–363.
23. Koenig, S., & Likhachev, M. (2006). A new principle for incremental heuristic search: Theoretical results. In *Proceedings of the International Conference on Autonomous Planning and Scheduling* (pp. 402–405).
24. Koenig, S., Likhachev, M., & Furcy, D. (2004). Lifelong Planning A*. *Artificial Intelligence Journal, 155*(1–2), 93–146.
25. Koenig, S., Likhachev, M., Liu, Y., & Furcy, D. (2004). Incremental heuristic search in Artificial Intelligence. *Artificial Intelligence Magazine, 25*(2), 99–112.
26. Koenig, S., & Simmons, R. G. (1996). Easy and hard testbeds for real-time search algorithms. In *Proceedings of the National Conference on Artificial Intelligence* (pp. 279–285).
27. Koenig, S., & Szymanski, B. (1999). Value-update rules for real-time search. In *Proceedings of the National Conference on Artificial Intelligence* (pp. 718–724).
28. Koenig, S., Tovey, C., & Smirnov, Y. (2003). Performance bounds for planning in unknown terrain. *Artificial Intelligence, 147*, 253–279.
29. Korf, R. (1990). Real-time heuristic search. *Artificial Intelligence, 42*(2–3), 189–211.
30. Korf, R. (1993). Linear-space best-first search. *Artificial Intelligence, 62*(1), 41–78.
31. Mudgal, A., Tovey, C., & Koenig, S. (2004). Analysis of greedy robot-navigation methods. In *Proceedings of the Conference on Artificial Intelligence and Mathematics*.
32. Pearl, J. (1985). *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley.
33. Pemberton, J., & Korf, R. (1992). *Making locally optimal decisions on graphs with cycles*. Technical Report 920004. Los Angeles, CA: Computer Science Department, University of California at Los Angeles.
34. Russell, S., & Wefald, E. (1991). *Do the right thing—Studies in limited rationality*. MIT Press.
35. Shue, L., Li, S., & Zamani, R. (2001). An intelligent heuristic algorithm for project scheduling problems. In *Proceedings of the Annual Meeting of the Decision Sciences Institute*.
36. Shue, L., & Zamani, R. (1993). An admissible heuristic search algorithm. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems* (pp. 69–75).
37. Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 1652–1659).
38. Sun, X., & Koenig, S. (2007). The Fringe-Saving A* search algorithm—A feasibility study. In *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 2391–2397).
39. Svennebring, J., & Koenig, S. (2004). Building terrain-covering ant robots. *Autonomous Robots, 16*(3), 313–332.
40. Thayer, S., Digney, B., Diaz, M., Stentz, A., Nabbe, B., & Hebert, M. (2000). Distributed robotic mapping of extreme environments. In *Proceedings of the SPIE: Mobile Robots XV and Telemanipulator and Telepresence Technologies VII* (Vol. 4195, pp. 84–95).
41. Trovato, K. (1990). Differential A*: An adaptive search method illustrated with robot path planning for moving obstacles and goals, and an uncertain environment. *Journal of Pattern Recognition and Artificial Intelligence, 4*(2), 245–268.
42. Wagner, I., Lindenbaum, M., & Bruckstein, A. (1999). Distributed covering by ant-robots using evaporating traces. *IEEE Transactions on Robotics and Automation, 15*(5), 918–933.
43. Yanovski, V., Wagner, I., & Bruckstein, A. (2003). A distributed ant algorithm for efficiently patrolling a network. *Algorithmica, 37*, 165–186.