Incremental Heuristic Search in Games: The Quest for Speed*

Maxim Likhachev
Carnegie Mellon University
Pittsburgh, PA 15213
maxim+@cs.cmu.edu

Sven Koenig
University of Southern California
Los Angeles, CA 90089
skoenig@usc.edu

Abstract

Robot path-planning methods can be used in real-time computer games but then need to run fast to ensure that the game characters move in real time, an issue addressed by incremental heuristic search methods. In this paper, we demonstrate how one can speed up D* Lite, an incremental heuristic search method that implements planning with the freespace assumption to move game characters in initially unknown or partially unknown terrain to given goal coordinates. We speed up D* Lite by implementing its priority queue with buckets rather than a binary heap. This non-trivial change reduces its runtime by a factor of two and effectively doubles the number of game characters that real-time computer games can afford.

Introduction

Consider real-time computer games such as Total Annihilation, Age of Empires or Dark Reign. The game characters initially do not know the terrain ("fog of war") but observe it within a certain range around them and then remember it for future use. To make them easy to control, the player can click on known or unknown terrain, and the game characters then move autonomously to the location that the player clicked on, which is the navigation task we study in this paper in the following setting: The terrain is discretized into cells that are either blocked or unblocked. The game characters can move in the four main compass directions with equal cost and use the sum of the absolute differences of the x and y coordinates of two cells (Manhattan distance) as heuristic estimate of their distance, resulting in consistent heuristics (but our search methods are more general than that and apply also to more complex graph topologies, non-uniform edge costs, and arbitrary consistent heuristics, including the zero heuristics). They do not know initially which cells are blocked. They always know which (unblocked) cells they are in, sense the blockage status of their neighboring cells, and can then move to any one of the unblocked neighboring cells. Their navigation task is to move to a given goal cell, for which they use planning with the freespace assumption:

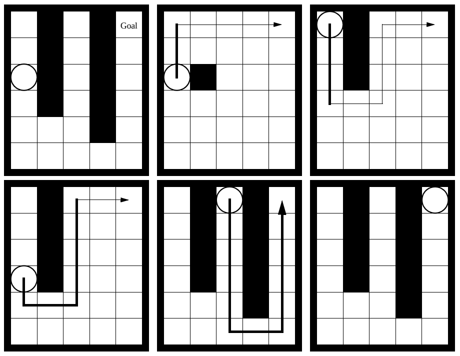

Figure 1: Planning with the Freespace Assumption: The goal cell is in the upper-right corner. The upper-left figure shows the terrain. The circle denotes the game character, and black cells denote blocked cells. The remaining figures show the knowledge that the game character has about the terrain at various points in time. The circle again denotes the game character, black cells denote blocked cells that the game character has already observed, the line denotes the cost-minimal presumed unblocked path from the current cell of the game character to the goal cell, and the thick part of the line denotes the beginning of the path that the game character actually traverses until it reaches the goal cell or observes the path to be blocked (and re-plans).

They plan and follow a cost-minimal presumed unblocked path from their current cell to the goal cell, that is, a cost-minimal path under the assumption that cells with unknown blockage status are unblocked. If they observe their current path to be blocked, they plan and follow another cost-minimal presumed unblocked path from their new current cell to the goal cell. They repeat this process until they either reach the goal cell or no presumed unblocked path from their current cell to the goal cell exists, in which case they cannot reach the goal cell. Figure 1 gives an example.

Planning with the freespace assumption results in paths that are both believable and provably short (Tovey, Greenberg, & Koenig 2003). It needs to run fast since every game character needs to find shortest presumed unblocked paths whenever it observes its current path to be blocked, yet needs to be responsive to the commands of the players and move smoothly. Therefore, it is often implemented with the incremental heuristic search methods D* (Stentz 1995) or its successor D* Lite (Koenig & Likhachev 2005). We demonstrate how one can speed up D* Lite by implementing its

*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 authors 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.

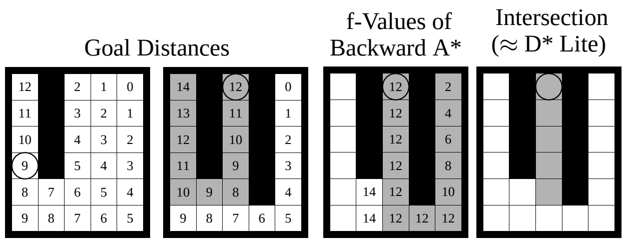

Figure 2: Illustration of D* Lite: The first and second figures in the top row show the goal distances of all cells during two consecutive searches. An incremental uninformed search basically only expands those cells whose goal distances have changed from the preceeding search, as shown in the second figure in grey. Heuristic search with Backward A* expands the cells shown in the third figure. D* Lite then (approximately) expands the cells expanded by both incremental and heuristic search, as shown in the fourth figure.

priority queue with buckets rather than a binary heap, which reduces the runtime of D* Lite by a factor of two.

Overview of D* Lite

D* Lite (Koenig & Likhachev 2005) is a version of Backward A*[1] that re-uses information from previous searches to speed up the current search, basically by transforming the previous A* search tree to the new one. It needs to search backward from the goal cell to the current cell of the game character since it requires the root of the search trees and thus the start of the searches to remain stationary. The original version of D* Lite uses a binary heap as priority queue to break ties in favor of cells with smaller g-values. Recently, an improved version of D* Lite was developed that uses a binary heap as priority queue to break ties in favor of cells with larger g-values (Likhachev & Koenig 2005). It tends to expand fewer cells and has a smaller runtime than the original version of D* Lite. One way of understanding D* Lite is to view it as a combination of incremental and heuristic search, as shown in Figure 2 for one particular search of our example.

D* Lite with a Binary Heap

D* Lite, in its most general form, finds a cost-minimal path from the current vertex of the game character to the goal vertex and moves the game character along it until it reaches the goal vertex or observes that some edge costs have changed, in which case it repeats the process. Figure 3 shows the standard implementation of D* Lite (Koenig & Likhachev 2005), using the following notation: S denotes the finite set of vertices of the graph. $s_{start} \in S$ denotes the current vertex of the game character (initially: its start vertex), and $s_{goal} \in S$ denotes its goal vertex. $Succ(s) \subseteq S$ denotes the set of successors of vertex $s \in S$ in the graph. Similarly, $Pred(s) \subseteq S$ denotes the set of predecessors of vertex $s \in S$ in the graph. $0 < c(s, s') \leq \infty$ denotes the cost of moving from vertex $s \in S$ to vertex $s' \in Succ(s)$. D* Lite maintains

[1]Backward A* is a version of A* that searches from the goal cell to the current cell of the game character, while Forward A* is a version of A* that searches in the opposite direction.

The pseudocode uses the following functions to manage the priority queue: U.TopKey() returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey() returns $[\infty; \infty]$.) U.Pop() deletes the vertex with the smallest priority in priority queue U and returns the vertex. U.Insert(s, k) inserts vertex s into priority queue U with priority k. Finally, U.Remove(s) removes vertex s from priority queue U.

```
procedure CalcKey($s$)
{01} return $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))]$;
procedure Initialize()
{02} $U = \emptyset$;
{03} $k_m = 0$;
{04} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{05} $rhs(s_{goal}) = 0$;
{06} U.Insert($s_{goal}$, CalcKey($s_{goal}$));
procedure UpdateVertex($u$)
{07} if $(u \neq s_{goal})$ $rhs(u) = \min_{s' \in \text{Succ}(u)}(c(u, s') + g(s'))$;
{08} if $(u \in U)$ U.Remove($u$);
{09} if $(g(u) \neq rhs(u))$ U.Insert($u$, CalcKey($u$));
procedure ComputeShortestPath()
{10} while (U.TopKey() $\dot{<}$ CalcKey($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
{11}   $k_{old}$ = U.TopKey();
{12}   $u$ = U.Pop();
{13}   if ($k_{old} \dot{<}$ CalcKey($u$))
{14}     U.Insert($u$, CalcKey($u$));
{15}   else if $(g(u) > rhs(u))$
{16}     $g(u) = rhs(u)$;
{17}     for all $s \in$ Pred($u$) UpdateVertex($s$);
{18}   else
{19}     $g(u) = \infty$;
{20}     for all $s \in$ Pred($u$) $\cup \{u\}$ UpdateVertex($s$);
procedure Main()
{21} $s_{last} = s_{start}$;
{22} Initialize();
{23} ComputeShortestPath();
{24} while $(s_{start} \neq s_{goal})$
{25}   /* if $(g(s_{start}) = \infty)$ then there is no known path */
{26}   $s_{start} = \arg\min_{s' \in \text{Succ}(s_{start})}(c(s_{start}, s') + g(s'))$;
{27}   Move to $s_{start}$;
{28}   Scan graph for changed edge costs;
{29}   if any edge costs changed
{30}     $k_m = k_m + h(s_{last}, s_{start})$;
{31}     $s_{last} = s_{start}$;
{32}     for all directed edges $(u, v)$ with changed edge costs
{33}       Update the edge cost $c(u, v)$;
{34}       UpdateVertex($u$);
{35}     ComputeShortestPath();
```

Figure 3: D* Lite

two kinds of estimates of the goal distance of each vertex s, namely a g-value $g(s)$ and an rhs-value $rhs(s)$. The rhs-value of a vertex is based on the g-values of its successors and thus potentially better informed than them. It always satisfies the following relationship: $rhs(s_{goal}) = 0$ and $rhs(s) = \min_{s' \in Succ(s)}(c(s, s') + g(s'))$ for $s \in S \backslash \{s_{goal}\}$. A vertex s is called consistent if $g(s) = rhs(s)$ (meaning that the estimate of its goal distance as given by its g-value is consistent with the estimates of the goal distances of its successors as given by their g-values), otherwise it is called inconsistent. An inconsistent vertex s is overconsistent if $g(s) > rhs(s)$ and underconsistent if $g(s) < rhs(s)$. If all vertices are consistent then all of their g-values are equal to their respective goal distances and a cost-minimal path from the start to the goal vertex can be found easily. However, D* Lite does not make every vertex consistent after some of the edge costs have changed. Instead, it uses consistent heuristics $h(s, s')$, that approximate the cost of a cost-minimal path from vertex s to vertex s', to focus the search and update only the g-values that are relevant for computing a cost-minimal path from the start to the goal vertex. D* Lite maintain a priority queue U that always contains exactly the inconsistent vertices. These are the vertices whose g-values D* Lite potentially needs to change to make the vertices consistent. The priority $k(s)$ of a vertex s in the priority queue can be a vector $[k_1(s); k_2(s); \ldots; k_n(s)]$ of n non-negative integers, which D* Lite calculates on line $\{01\}$. D* Lite always chooses a vertex in the priority queue (= an inconsistent vertex) with the smallest priority according to a lexicographic ordering and expands it. D* Lite defines priorities not only for the vertices in the priority queue but for all vertices because its termination criterion can then be stated in terms of these priorities. The correctness of D* Lite has been proved for all priorities that satisfy the following two properties (Likhachev 2005), where $c^*(s, s')$ denotes the cost of a cost-minimal path from vertex s to vertex s':

(a) for any two vertices $s, s' \in S$ if $c^*(s_{start}, s') < \infty$, $g(s') \geq rhs(s')$, $g(s) > rhs(s)$ and $rhs(s') > c^*(s', s) + rhs(s)$, then $k(s') > k(s)$; and

(b) for any two vertices $s, s' \in S$ if $c^*(s_{start}, s') < \infty$, $g(s') \geq rhs(s')$, $g(s) < rhs(s)$ and $rhs(s') \geq c^*(s', s) + g(s)$, then $k(s') > k(s)$.

The version of D* Lite from Figure 3 breaks ties in favor of cells with smaller g-values and uses priorities that are pairs of integers. The version of D* Lite that breaks ties in favor of cells with larger g-values (Likhachev & Koenig 2005) uses priorities that are triples of integers. The large number of possible priorities limits how one can implement the priority queue of D* Lite. Priority queues are commonly implemented with buckets or binary heaps. The amount of memory needed to store the priority queue depends in both cases on the number of elements in the priority queue. However, in case of buckets, it also depends on the number of possible priorities because buckets reserve at least one pointer for each possible priority, namely to a list (also called bucket) of elements with that priority. Thus, priority queues cannot reasonably be implemented with buckets in case the number of possible priorities is large, as was the case for D* Lite so far, even though buckets tend to result in much smaller runtimes than binary heaps.

D* Lite with Buckets

To implement D* Lite with buckets in case the costs and heuristics are integers (as is the case for planning with the freespace assumption), we now define simpler priorities for D* Lite and modify the calculation on line $\{01\}$ accordingly, namely $k(s) = [rhs(s) + h(s_{start}, s); 1]$ for consistent or overconsistent vertices $s \in S$ and $k(s) = [g(s) + h(s_{start}, s); 0]$ for underconsistent vertices $s \in S$.[2] These priorities are equivalent to using $k(s) = 2 * (rhs(s) + h(s_{start}, s)) + 1$ for consistent or overconsistent vertices $s \in S$ and $k(s) = 2 * (g(s) + h(s_{start}, s))$ for underconsistent vertices $s \in S$. They can thus be implemented with scalars that are at most about twice as large as the f-values of A* rather than vectors, which makes it possible to implement this version of D* Lite with buckets. It only remains to be shown that our priorities satisfy the two properties given above:

(a) Consider any two vertices $s, s' \in S$ with $c^*(s_{start}, s') < \infty$, $g(s') \geq rhs(s')$ (that is, vertex s' is either consistent or overconsistent), $g(s) > rhs(s)$ (that is, vertex s is overconsistent) and $rhs(s') > c^*(s', s) + rhs(s)$. It then follows from the definition of our priorities that $k(s') = [rhs(s') + h(s_{start}, s'); 1] > [rhs(s) + c^*(s', s) + h(s_{start}, s'); 1] \geq [rhs(s) + h(s_{start}, s); 1] = k(s)$. The strict inequality holds because $rhs(s') > c^*(s', s) + rhs(s)$ and $h(s_{start}, s') < \infty$ (since $c^*(s_{start}, s') < \infty$ and the heuristics are consistent and thus admissible). The non-strict inequality holds due to the consistency of the heuristics.

(b) Consider any two vertices $s, s' \in S$ with $c^*(s_{start}, s') < \infty$, $g(s') \geq rhs(s')$ (that is, vertex s' is either consistent or overconsistent), $g(s) < rhs(s)$ (that is, vertex s is underconsistent) and $rhs(s') \geq c^*(s', s) + g(s)$. It then follows from the definition of our priorities that $k(s') = [rhs(s') + h(s_{start}, s'); 1] > [g(s) + c^*(s', s) + h(s_{start}, s'); 0] \geq [g(s) + h(s_{start}, s); 0] = k(s)$. The strict inequality holds because $rhs(s') \geq c^*(s', s) + g(s)$ and $h(s_{start}, s') < \infty$ (since $c^*(s_{start}, s') < \infty$ and the heuristics are consistent and thus admissible) and the second element of the priority on the left (one) is strictly larger than the second element of the priority on the right (zero). The non-strict inequality holds due to the consistency of the heuristics. ■

	Expanded Cells	Runtime (μsec)
D* Lite (Larger G-Values)	11416.4 ± 229.1	10439.7 ± 210.9
D* Lite (Buckets)	11632.5 ± 232.7	4448.8 ± 89.2

Table 1: Experiments in Random Mazes

Experiments

We perform experiments in randomly generated terrain of size 201×201 that is solvable. We generate the corridor structure of the maze-like terrain with a randomized depth-first search and then remove 750 walls. We average over 5000 random mazes with randomly chosen start and goal cells on a computer with a Pentium IV 2GHz processor and 2 GByte of RAM. Table 1 reports the number of expanded cells as well as the runtime until the goal cell is reached for both D* Lite with a binary heap and D* Lite with buckets, that we both implemented with all standard optimizations (Koenig & Likhachev 2005). The runtime of D* Lite with buckets is only half of the runtime of D* Lite with a binary heap, which correlates well with similar speed-ups for other versions of A*. We also show the 95-percent confidence intervals (under the assumption that the distribution is normal) to demonstrate the statistical significance of our results.

References

Koenig, S., and Likhachev, M. 2005. Fast replanning for navigation in unknown terrain. *Transactions on Robotics* 21(3):354–363.

Likhachev, M., and Koenig, S. 2005. A generalized framework for Lifelong Planning A*. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 99–108.

Likhachev, M. 2005. *Search-based Planning for Large Dynamic Environments*. PhD dissertation, School of Computer Science, Carnegie Mellon University.

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

Tovey, C.; Greenberg, S.; and Koenig, S. 2003. Improved analysis of D*. In *Proceedings of the International Conference on Robotics and Automation*, 3371–3378.

[2] D* Lite uses $h(s_{start}, s)$ instead of the more familiar $h(s, s_{goal})$ because it is an incremental version of Backward A* rather than Forward A*.