Memory-Bounded D* Lite

Denton Cockburn and Ziad Kobti
School of Computer Science
University Of Windsor
Windsor, Ontario, Canada

Abstract

A* is a well-established pathfinding algorithm. Focussed D* is one of the more popular variations thereof. D* Lite in turn is based upon Focussed D*. Being based upon A*, D* Lite suffers from similar high memory usage as A*, as all nodes being expanded need to remain in the algorithm's OPEN list. D* Lite repairs the solution path as changes are detected, but oftentimes without any bounds on the amount of items placed on the update queue, creating a situation in which it is possible that all nodes within the map are placed on the queue, in the worst case. Memory-Bounded D* Lite (MD* Lite) aims to provide the same advantages of D* Lite, while applying memory usage constraints.

Introduction

Pathfinding algorithms plan paths from a starting location to one or more goal locations within an environment. The environment may be static or dynamic. Algorithms such as A* (Hart, Nilsson, and Raphael 1968) are popular for static environments. The Focussed D* algorithm (Stentz, 1995) was designed to handle dynamic environments where the traversability of nodes within the environment may only be partially known. The D* Lite algorithm (Koenig and Likhachev 2002) is a hybrid algorithm, taking aspects from both Lifelong-Planning A* (LPA*) (Koenig, Likhachev, and Furcy 2004) and Focussed D*. It is designed for agents operating in environments that are incompletely known, while the agent is navigating. Like D* and LPA*, D* Lite is optimal and complete. In this paper, we examine a variant of D* Lite, called Memory-Bounded D* Lite, that uses less memory while maintaing optimality and completeness.

D* Lite is algorithmically different, while being at least as efficient as Focussed D* . D* Lite also determines the same paths as Focussed D*. On every iteration, the agent plans the shortest path to the goal, with the assumption that unknown nodes are traversible. It is thus necessary that the agent knows the layout of the map, even if not the actual accessibility status of nodes. When new information is discovered regarding the accuracy of the map, the agent refines the solution if necessary. The cause of the discovery is irrelevant, as the changes could be globally created, or created by the effects of the agent or other agents in the environment. The discovery of these changes is also irrelevant, ranging from sensors of the agent, to global sensors reporting all changes. The agent continues until the goal is found, or all paths are determined to be untraversable.

D* Lite must terminate in all cases, because the agent either follows the path to the goal, or the increased knowledge regarding edge-costs results in knowing that no path exists. The agent replans the shortest path from the current location to the goal when the current path is untraversable. The algorithm makes no assumptions about whether node traversal costs are increasing or decreasing. It is also irrelevant to the algorithm where these nodes are changing, close to the current location or not. It is also irrelevant whether the change is real, as in the cost has actually changed, or the change is simply perceived to be so, such as might be the case if another agent is occupying a node that can only hold one agent. If a node is untraversible, the edge cost is set to infinity. One result of such an approach is that D* Lite is able to plan with the existence of other agents also navigating the environment.

LPA* is an incremental version of A* designed to work in dynamic environments. LPA* is usually used in fully known dynamic terrains. The initial search of LPA* is identical to that of A*, while subsequent searches reuse information from previous searches. LPA* would replan completely when costs changed, which was inefficient behaviour as most edges are unlikely to change between replanning episodes. While this is so, LPA* uses heuristics to focus its replanning, only updating nodes with costs that can possibly affect the shortest path. The algorithm uses a priority queue that only contains locally inconsistent vertices, which are nodes whose costs have changed. These inconsistent vertices may affect the LPA* algorithm, causing it to recompute its path. LPA* continually expands vertices until the goal node is consistent and the next node to expand does not have a lower cost than the goal node (as determined by LPA*). If the cost attributed to the goal node is infinite, then there exists no path to the goal. LPA* traces back through nodes to identify the path from the start location to the goal. This is the same method used by A* when it doesn't maintain a list of previously expanded elements (called the CLOSED list).

D* Lite switches the search direction of LPA*. The goal node becomes the start node and vice versa. The direction of all edges within the environment are reversed. The shortest path from the start location to the goal location can be decided by continually minimizing the cost function C(s,s') and the goal-estimate distance g(s') for the current location s and any successor node s'. Another change that D* Lite makes to LPA* is to dynamically move the agent while updating the keys of vertices in its priority queue. This is necessary because heuristics change as the agent moves, and also because the heuristics were calculated based on the previous location of the agent.

Focussed D* is an incremental and heuristically driven derivative of A*. The algorithm is fully optimal, always finding the most efficient paths, while taking into effect changes within the environment. The efficiency of a path is context dependent, including such things as distance travelled, amount of nodes navigated, and energy expended. Focussed D* is much faster than LPA*, as it modifies previous searches locally. Focussed D* is used in real-world applications such as Nomad robots (Koenig, Tovey, and Halliburton 2001).

D* Lite Notation

The D* Lite algorithm is listed below. Most of the notation for D* Lite is taken from the LPA* algorithm. S represents the set of vertices within the graph. Succ(s) is a subset of S, representing the successor vertices of a vertex s, which is also within S. Pred(s) is similar to Succ(s), being the set of predecessors of s. The start vertex is sstart and the goal vertex is sgoal. The priority queue needed to maintain the vertices that need updating is U. The estimated distance from the goal to a node s is g(s), while h(s,s') is the heuristic estimate of the distance from s to s'. The heuristic function needs to be non-negative and forward-backward consistent. That means that $h(s, s'') \leq h(s, s') + h(s', s'')$ for all vertices s, s', and $s'' \in S$. The heuristic used must also be admissible at all times. Thus $h(s, s') \leq c^*(s, s')$ for all vertices s, $s' \in S$, where c*(s,s') is the minimum cost of moving from s to s'. D* Lite uses a right-hand-side value, rhs(s), which is based on the *g* values of the predecessors of s. The value of rhs(s) is 0 if s = sstart, otherwise it is $\min_{s'} \in$Pred(s) (g(s') + c(s' , s)). D* Lite maintains a key modifier km, which is added to the first component of items when their keys are computed. The algorithm below is slightly modified from that presented in (Koenig and Likhachev 2002).

D* Lite Algorithm

The Initialize function sets the initial *g* and *rhs* values of all vertices to infinity. The goal vertex is inserted into the priority queue because it is initially inconsistent. The agent then makes a transition of only one vertex along the shortest path. The new vertex location is reflected in the updating of sstart, as this vertex is the start of the subsequent search. When an edge cost changes, the *rhs* values and keys of potentially affected vertices are updated in the UpdateVertex function. Vertices that become locally consistent or inconsistent due to this are either removed from the queue or placed on it respectively. The keys of all items within the priority queue

```
Procedure CalcKey(s)
1. return [min(g(s), rhs(s)) + h(sstart , s) + km; min(g(s), rhs(s))];

Procedure Initialize()
1. U = $\emptyset$
2. km = 0
3. for all s in S, rhs(s) = g(s) = $\infty$
4. rhs(sgoal) = 0
5. U.Insert(sgoal , CalcKey(sgoal))

Procedure UpdateVertex(u)
1. if (u $\neq$ sgoal) rhs(u) = min(s' in Succ(s): (c(s', s) + g(s')))
2. if (u $\epsilon$ U) U.Remove(u)
3. if (g(u) $\neq$ rhs(u)) U.Insert(u, CalcKey(u))

Procedure ComputeShortestPath()
1. while (U.TopKey() $<$ CalcKey(sstart) OR rhs(sstart) $>$ g(sstart))
   (a) kold = U.TopKey();
   (b) u = U.Pop();
   (c) knew = CalcKey(u));
   (d) if(kold $<$ knew) U.Insert(u, knew)
   (e) else if (g(u) $>$ rhs(u))
       i. g(u) = rhs(u);
       ii. for all s in Pred(u) UpdateVertex(s)
   (f) else
       i. g(u) = $\infty$
       ii. for all s in Pred(u) U [u] UpdateVertex(s)

Procedure Main
1. slast = sstart
2. Initialize()
3. ComputeShortestPath()
4. while (sstart $\neq$ sgoal)
   (a) sstart = arg min(s' in Succ(s): (c(s', s) + g(s')))
   (b) Move to sstart
   (c) scan graph for changed edge costs
   (d) if any edge costs changed
       i. km = km + h(slast, sstart)
       ii. slast = sstart
       iii. for all directed edges (u, v) with changed edge costs
            A. Update the edge cost c(u, v)
            B. UpdateVertex(u)
       iv. ComputeShortestPath()
```

are updated, and the shortest path is recalculated. The recalculation may be rather quick, as the changed vertices may not have affected, or only slightly have affected, the shortest path.

Memory-Bounded D* Lite

Memory-Bounded D* Lite (MD* Lite) is an algorithm that operates in dynamic environments where agents have restricted memory. The algorithm is based on D* Lite, using the same notation. The changes required to D* Lite are presented below. Procedures that are the same as presented in D* Lite are simply mentioned. ComputeShortestPath() includes only the change to know when the algorithm has ran out of memory. That change is mentioned, but not presented.

The algorithm works by managing what elements are to be added to the priority queue, U. A parameter to the algorithm is the amount of items allowed on U simultaneously, which we call M. Items in U are of the form (node, key). Items to be processed by ComputeShortestPath() are kept in U.

When an element n is to added to U during UpdateVertex(), we check whether the number of elements in U is less than M. If there are less than M items in U, then n is added. If there are M items already in U, then we need to create space such that n will be accounted for. We do this by selecting an item w, which is the element in U for which key(w) is minimum. If w is worse than n, then w is removed, and n is added to U. A node s is worse than another node s' if key(s) > key(s'). If n is worse than w, then we assign w to n. Thus, w will be the the worst node in the union of U and [n].

This node w has to be accounted for in U. We therefore select the parent p of w. The parent of a node n is the member s of Pred(n) for which g(s) is minimum. Thus the parent of a node is its predecessor through which lies the minimum cost path from the node to the goal. We maintain a value F that is the key of w, for whatever node is currently assigned as w. If p is already in U, then the key of p in U, which we call K(p) here, is set to min(K(p), F). Since ComputeShortestPath() processes nodes in U in order of key, p is guaranteed to be expanded before w. If p itself needed to be processed, then K(p) = key(p), and p will be processed before its children. When p is processed by ComputeShortestPath(), its children will be expanded again, and we will again attempt to place them on U.

If p is not in U, then we attempt to add it. If p is not the worst node, then p is added to U. After this addition, we set K(p) = F. Thus, p will not be processed until its descendant w would have been. If p is again the worst node, then we set p to the parent of p, and leaving F at its current value. If UpdateVertex() is called with a consistent node, then that node is removed from U if it is there. UpdateVertex() also sets the *rhs* value of a node, which is the current minimum cost path from the goal to the node. To ensure that p above is processed, we set its *g* value to infinity, making it inconsistent, which we correct within UpdateVertex().

MD* Lite is complete when M is sufficiently large, allowing all changes to be accounted for in U. If M is not large enough, then the algorithm will fail. It is possible to allow the algorithm to adjust the size of M when a node is unable to be accounted for. Failure is detected when a node is the top node for two consecutive iterations without its queue key changing. If M is greater than the maximum amount of nodes that D* Lite needs, then MD* will run exactly like D* Lite. If M is less than the amount of nodes that are needed, then MD* Lite will not exceed M until instructed to. Therefore, MD* Lite is guaranteed to store the same or less nodes than D* Lite.

Procedure CalcKey(u)

Procedure Initialize()

Procedure UpdateVertex(u)

```
1. if u $\neq$ goal then rhs(u) = min(s' in Succ(s): (c(s', s) + g(s')))
2. if u in U then U.Remove(u)
3. if g[u] $\neq$ rhs(u)
  (a) if F then g[u] = rhs(u)
  (b) if size(U) $<$ M then U.Insert(u, CalcKey(u))
  (c) else
      i. w = item s in U where f(s) is maximum
     ii. if key(w) $>$ key(u)
         A. U.Remove(w)
         B. if F then U.Insert(u, F)
         C. else U.Insert(u, key(u))
         D. F = None
         E. w = u
    iii. if F then key(u) = F and F = None
     iv. p = Parent(u)
      v. if p in U then U.Update(parent, min(K(p), key(u)))
     vi. else if CalcKey(p) $<$ CalcKey(start)
         A. g(p) = $\infty$
         B. F = key(u)
         C. UpdateVertex(p)
```

Procedure Parent(n)

```
1. avail = { x | x in Succ(n) and x in U }
2. if avail is empty then avail = Succ(n)
3. return x in avail for which (g(x) + c(x, n)) is minimum.
```

Procedure ComputeShortestPath() same as D* Lite, except if U.top() is the same as previous iteration of while loop, with the same key, then we are out of memory. In which case if M is adjustable then adjust M.

Procedure Main(M, Adjustable) - M specifies the allowed memory. Adjustable specifies whether M is adjustable if we run out of memory.

Testing Environment

For the purposes of these experiments, a four-connected graph was used. Traversal costs between nodes are node-based as opposed to edge-based. The result of this is that for x,y neighbours of z, C(x,z) = C(y,z), where C is the cost-function for edge-traversal. A side effect of this is that reverse traversal costs between nodes will frequently differ. Thus, in the previous example, C(x, z) is not necessarily equal to C(z, x). All edges within the environment can be blocked/unblocked with the exception of the goal and start nodes (as the search would be immediately failed). The costs of traversing nodes increase and decrease randomly, with the use of a global agent completely responsible for such acts.

In the paper on D* Lite, that algorithm is compared to many established algorithms such as Focussed D*, Backward/Forward A*, and Breadth-first search. We'll forgo the comparison to those algorithms in lieu of comparing MD* Lite to D* Lite. The items to be compared are the total amount of nodes stored in U simultaneously, the amount of nodes expanded, the run-time of the algorithms, and the amount of accesses to U. An access is the insertion, update, or deletion of a node. The run-time tests are implementation dependent, and were not optimized, but should provide some perspective on the time impact suffered by MD* Lite in comparison to D* Lite.

The grid to be used will be 100x100. The starting and ending locations are randomly chosen, with a requirement that the minimum path between them is 50 nodes measured by manhattan distance. To reduce the amount of changed nodes detected by the agent, the on-board sensors are to be limited to a range of 3 nodes measured by euclidean distance. It is not possible to accurately predict the amount of memory that MD* Lite will need before searching, so we initially alloted an M of 25 + the manhattan distance from the start to the goal. We made use of the previously mentioned capability for MD* Lite to increase its M to avoid failure. The M was increased by 10% when the algorithm ran out of space. MD* Lite will use less than the alloted memory if possible.

At the start of each test, a certain amount of nodes within the grid were blocked. In addition, during each movement of the agent, we allowed each unblocked node to increase or decrease its cost. Each node selected to change had a 50% chance of increasing, with an equal chance of decreasing. Unblocked nodes were not blocked after the initial setup. We ran tests over a range of blockade and change percentages. 100 tests were ran at each blockade percentage from 1% to 10%. So we blocked 100 nodes for the first set, then 200 nodes, etc. On each timestep, each node had a probability that its cost would change, this is what is referred to as the change percentage. These tests were divided into groups of 20 each per change percentage from 1% to 5%. So for when we had 100 nodes blocked, we had 20 simulations with 1% of nodes changing per iteration, then 20 with 2%, etc. We ran a total of 1000 tests (20 x 5 change groups x 10 blockade groups).

Test Results

MD* Lite used significantly less memory than D* Lite. Over the 1000 tests, MD* Lite stored only 22% as many nodes simultaneously when compared to D* Lite, with a standard deviation of 6%. The results of the node storage comparison is illustrated in Figure 1. MD* Lite expanded an average of 5% more nodes than D* Lite, with a 46% standard deviation. Thus, it can be expected in most cases that MD* Lite will expand less than 50% more nodes.

MD* Lite accessed the queue more often than D* Lite. Queue accesses were an average of 211% that of D* Lite, with a standard deviation of 90%. This is illustrated in Figure 2. Due to the increase in queue accesses, performance speed of the queue is important. There are methods that are available to provide O($\sqrt{\log n}$) and better time complexity for queue operations (Fredman and Willard 1994). As earlier stated, neither algorithm was implemented with runtime performance as a concern. MD* Lite took an average of 45% more time than D* Lite. The standard deviation for this measure was 97%. The results are illustrated in Figure 3. The standard deviation is also illustrated in the second and third images.

While MD* Lite stored significantly less items on the queue than D* Lite, it suffered increases in queue accesses and runtime. This can be improved if more nodes are allowed to be stored on the queue. In such cases, MD* Lite will need to remove and re-add less nodes. This would in turn reduce the runtime of the algorithm. Our testing method was designed to allow MD* Lite to use a very small amount of memory. The result of this was reflected in the increases in queue accesses and runtime cost. In actual implementations, these factors can be evaluated based on importance.

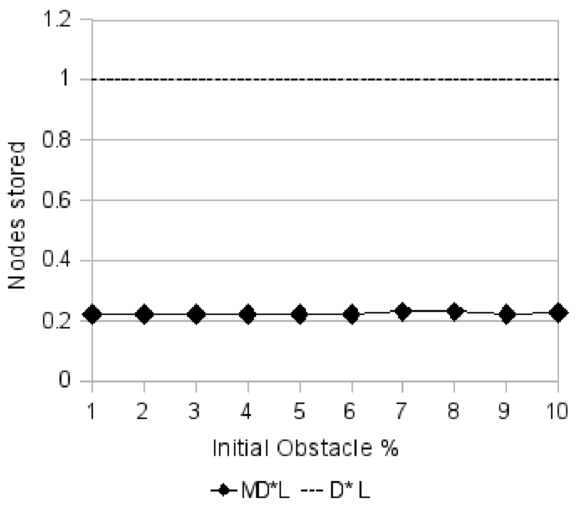

Figure 1: Ratio of maximum nodes stored simultaneously in queue. Normalized to D* Lite.

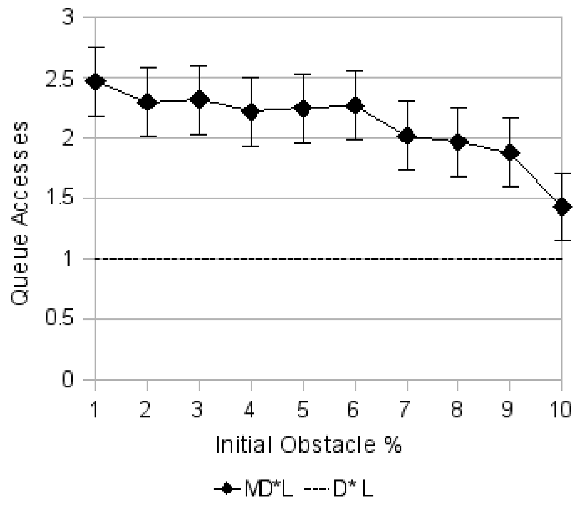

Figure 2: Ratio of queue accesses. Normalized to D* Lite.

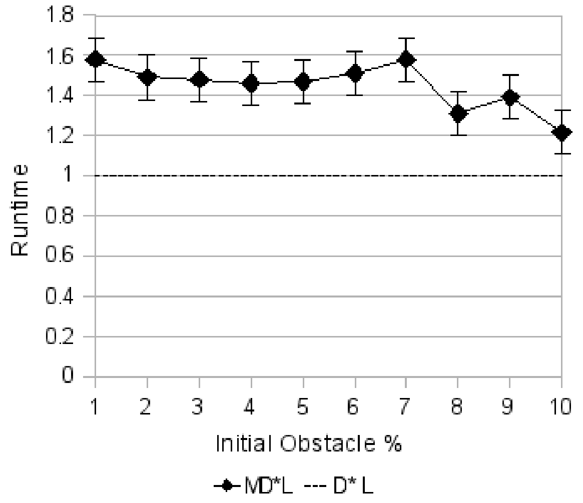

Figure 3: Ratio of runtime performance. Normalized to D* Lite.

Conclusion and future work

MD* Lite is always able to use the same or less memory than D* Lite, while producing the same path, but will also expand at least as many nodes as D* Lite. Similar to the relationship between A* and Simplified Memory-Bounded A* (Russell and Norvig 1994), there are cases where using MD* Lite is intractable if there is insufficient allowance of queue space. This is because switching between too many different path solutions with insufficient memory can result in too many nodes being forgotten and needing to be added again. This can be alleviated by allowing the algorithm to increase the amount of nodes it is allowed to store.

There exists a delayed version of D* called Delayed D* (Furguson and Stentz 2005). It outperforms D* Lite in terms of node expansions and runtime. It does this by optimizing the order in which nodes are expanded, resulting in the avoidance of some unnecessary expansions. In the future, MD* Lite can be modified to follow a similar strategy. This change should result in MD* Lite outperforming D* Lite with regards to node expansions and runtime.

MD* Lite removes nodes from the priority queue after their expansion. When the algorithm runs out of space and needs to account for additional nodes, it recursively computes parent nodes, adding suitable ones to the queue until all nodes are again accounted for. An idea is to keep nodes that are on the solution path in the queue. This may result in less computation when we need to create space to account for a node.

While Focussed D* and its variants are extremely popular in dynamic search environments, they are not aimed at being constrained by memory. We were able to present the MD* Lite algorithm, which is constrained by memory, yet is optimal and complete. MD* Lite has a place in environments in which memory usage is a concern. This has usage in gaming development, as well as in memory-restricted devices such as cell-phones and PDAs.

References

Hart, P., Nilsson, N., Raphael, B. (1968) A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 2 pg.100-107

Stentz, A. (1995) The focussed D* algorithm for real-time replanning. Proceedings of the International Joint Conference on Artificial Intelligence-95, Montreal, Quebec

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

Koenig, S., Likhachev, M., and Furcy, D., (2004) Lifelong planning A*. Artificial Intelligence Journal, 155(1-2) pg. 93-146

Koenig, S., Tovey, C., Halliburton, W. (2001) Greedy mapping of terrain. In Proceedings of the International Conference on Robotics and Automation, pg. 3594-3599

Fredman, M.L., and Willard, D. E. (1994) Surpassing the information theoretic bound with fusion trees. Journal of Computer and System Sciences, 48(3) pg. 533-551

Russell, S. and Norvig, P. (1994) Artificial Intelligence: A Modern Approach. Prentice-Hall

Ferguson, D. and Stentz, A. (2005) The Delayed D* Algorithm for Efficient Path Replanning, In Proceedings of the IEEE International Conference on Robotics and Automation, April, pg. 2045 - 2050.