2007

Memory-constrained pathfinding algorithms for partially-known environments

Denton Cockburn
University of Windsor

Memory-Constrained Pathfinding Algorithms for Partially-Known Environments

by

Denton Cockburn

A Thesis
Submitted to the Faculty of Graduate Studies
through The School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2007

Your file Votre référence
ISBN: 978-0-494-34975-5
Our file Notre référence
ISBN: 978-0-494-34975-5

ABSTRACT

Pathfinding is the search for a goal state given a start state, within either static or dynamic environments. Many pathfinding algorithms exist, including established algorithms such as A*, SMA*, and D*. These algorithms all provide optimal solution paths, using all available memory. Consequently, Algorithms such as A* and D* are known to be inefficient in terms of memory space usage. SMA* and similar algorithms provide a means by which optimal solution paths can be found while being memory efficient. SMA* and such algorithms are restricted to static environments, in which state traversal costs never change. This is a severe limitation, as one of the primary fields for search algorithms are games, many of which involve dynamic environments. Presented in this paper is a dynamic variant of the established D* Lite algorithm, that is able to provide an optimal solution path, if given sufficient memory, while using as little memory as possible.

It is also the case that in some areas, an optimal solution is not needed. This may be the case for robotics. Many algorithms already exist in this area, such as Anytime algorithms for time-limited searches. Real-Time algorithms for when the agent needs to move while planning its path. Also presented is an algorithm for real-time planning when an agent does not have a priori knowledge of the environment, and also limited memory capacity. This algorithm sacrifices optimality, but in turn is highly memory efficient, even in comparison to other algorithms designed for memory efficiency.

DEDICATION

I would like to dedicate this work to my two sets of parents. I am fortunate to have two great families that love me so much. My dad and stepmother, thank you so much for all that you have put into me and my education. I couldn't be who I am today without you. My mother and my stepfather, thank you for helping me be the man I am today. When I get stressed with the winter months and the school burnout, I always have a tropical home to return to. I'd like to thank you all for all that you have done for me, I love you.

ACKNOWLEDGEMENTS

I would like to thank first of all my advisor Dr. Kobti. Your guidance has helped me through times when I doubted myself. You have also been an inspiration in showing me that I can continue my education to even higher heights. I'd also like to thank Dr. Goodwin. Your class is what started me along this research area. Thank you for making your class fun to be in and learn from. Also thank you for the advice and patience you have given and shown. I'd also like to thank Dr. Ngom. Even though I wasn't a student of yours, you spent a lot of your time advising me. I thank you for suggesting I continue with this research area. Finally, I'd like to thank the ladies in the computer science office. You made handling school bureaucracy a breeze.

Table of Contents

ABSTRACT....3
DEDICATION....4
ACKNOWLEDGEMENTS....5
Chapter 1: Introduction....1
1.1 Background....1
1.2 Motivation....1
1.3 Thesis Contribution....3
1.4 Thesis Organization....3
Chapter 2: Review of Literature....5
2.1 A*....5
2.2 SMA*....7
2.3 LPA*....7
2.4 Focused D*....9
2.5 D* Lite....9
2.6 Anytime Repairing A*....10
2.7 Anytime D*....11
2.8 Adaptive A*....12
2.9 Fringe Saving A*....13
2.10 Delayed D*....14
Chapter 3: Range-Limited A*....17
3.1 Introduction....17
3.2 Initial comparisons....19
3.3 Admissible Heuristics....21
3.4 Blockades....22
3.5 Dynamic Environments....30
3.6 Completeness....31
3.7 Final RLA* Algorithm....31
3.8 Conclusion....33
Chapter 4: Memory-Bounded D* Lite....35
4.1 Introduction....35
4.2 Detailed Review of D* Lite and SMA*....35
4.2.1 D* Lite overview....35
4.2.2 D* Lite Notation....37
4.2.3 D* Lite Algorithm....38
4.3 SMA*....41
4.3.1 SMA* overview....41
4.3.2 SMA* algorithm:....43
4.4 Memory-Bounded D* Lite....44
4.5 Theorems....51
4.6 Testing Environment:....51
4.7 Test Results....52
4.7.1 Nodes Stored....52
4.3.2 Nodes Expanded....53
Chapter 5: Conclusions and Future Work....55

5.1 Conclusions.....55
5.2 Future work.....56
REFERENCES.....57
VITA AUCTORIS.....63

Chapter 1: Introduction

1.1 Background

Pathfinding is the search for a solution path from a given starting location to one or more goal locations. In the context of this document, pathfinding occurs in both static and dynamic environments. In static environments, the cost of traversing each state never changes from its initial value. In dynamic environments, state traversal costs can increase or decrease, as well as states can become blocked or unblocked between time units. There are several algorithms for pathfinding in static environments. There are also algorithms for pathfinding in dynamic environments. Some of these algorithms involve many domain-specific constraints. Some are required to handle a limited amount of time to perform the search. Some have to handle maps with imperfect information. Also, there are cases where the algorithm has to perform incrementally to handle real-time traversal, or even simply because the environment changes enough to warrant it. These algorithms progressively improve upon other approaches. What is lacking is the existence of a real-time, dynamic pathfinding algorithm that focuses on minimizing the amount of memory used.

1.2 Motivation

Pathfinding is a very important field within artificial intelligence. Pathfinding algorithms such as D* (Stentz, 1995) are being used by NASA for the Mars Rover (Stentz, 1995). Some of the most popular computer games involve the use of some pathfinding for non-player characters. Many popular video games make use of pathfinding algorithms. Outside of the field of gaming, pathfinding is used for areas such as route planning, for example in making a telephone call. Car navigation systems make use of pathfinding algorithms to plot potential paths for drivers that minimize travel time or distance.

Pathfinding algorithms such as Best-First Search (Pearl, 1984) have given way to more heuristically driven algorithms such as A* (Hart et al, 1968). While A* is optimal and complete, it has its limitations, being memory intensive as well as only working in static environments. Algorithms such as SMA* (Russell and Norvig, 1994) were created to deal with the memory issues of A*, while maintaining completeness and optimality. These algorithms still do not address the dynamic environment issue, for which algorithms such as LPA* (Koenig et al, 2004a) were created. LPA* dynamically updates the search path to handle the changing environment, while the agent is navigating, thus at all times providing an optimal path based on the heuristic and the cost of traversing the path.

A new class of problems involved real-time pathfinding. In these problems, the algorithm would actually be applied to an agent traversing within the environment. Thus incremental versions of search algorithms were necessary, as replanning the perfect path at all times would be highly computationally intensive in dynamic environments that change frequently. D* is one such algorithm. Some algorithms were limited based on how much time was available to perform the search, thus resulting in anytime algorithms that try to find a path quickly, refining it according to the amount of time allotted to perform the search. One such algorithm is Anytime Repairing A* (Likhachev et al, 2003).

As robotics become a regular part of the lives of citizens of industrialized nations, pathfinding plays a higher role. Robots that vacuum rooms for example, need to apply some form of pathfinding algorithm, even if simplistic. Children's toys are continuously being made more advanced and computerized. Computerized companion toys for children are cheaper as component prices are reduced. Cell phones are becoming more powerful and feature rich. Most cell phones support the playing of computer games. Even though memory is relatively abundant in desktop and laptop usages, some cell phones are still restricted by the amount of memory available for gaming. This means that there still exist a place for memory efficient dynamic pathfinding algorithms. An algorithm is said to

be memory efficient when it uses significantly less memory in comparison to other established algorithms. The goal of this thesis is the creation of two dynamic, incremental pathfinding algorithms that will use significantly less memory than established algorithms such as D*, D* Lite (Koenig and Likhachev, 2002a) and A*.

1.3 Thesis Contribution

For my thesis I intend to explore the creation of a range-limited variant of A* that uses significantly less memory. I will explore the development of this algorithm, addressing problems that the algorithm would have to overcome. The performance of the algorithm, which I call RLA* (Range-Limited A*), will be observed in static as well as dynamic environments. I will perform experiments to demonstrate the behaviour of the algorithm, while highlighting its shortcomings. RLA* will be compared to other algorithms including A*, SMA*, and D* Lite and D*. In the end, I intend to have an algorithm that is efficient in terms of memory usage, with only a reasonable sacrifice to the quality of the path found.

I also intend to develop a second algorithm. It will be called Memory-bounded D* Lite (MD* Lite). The algorithm will be able to find the same paths as D* Lite, if given enough memory allocation. The aim of MD* Lite is to be for D* Lite what SMA* is to A*. Like SMA*, MD* Lite suffers from the inability to pre-determine the amount of memory it will need to be able to find the optimal path, or any sufficient path. MD* Lite is compared to RLA*, D* and D* Lite in dynamic testing environments, with the goal of observing how much memory it uses in comparison to these algorithms. We will also test to see how many more nodes MD* Lite needs to expand in comparison to D* Lite to maintain the memory restrictions in its given environment.

1.4 Thesis Organization

I present a survey of some of the existing literature in this area in chapter 2. In chapter 3, I present the RLA* algorithm, a suboptimal memory efficient search algorithm. In chapter 4, I present memory-bounded D* Lite, a variation of the D* Lite algorithm that is aimed at memory constrained environments. Experiments for both these algorithms are presented in the same chapters as the algorithms. Finally in chapter 5 are the conclusions and plans for future work.

Chapter 2: Review of Literature

*2.1 A**

The A* algorithm (Hart et al., 1968) defines for each node a value f(n) = g(n) + h(n). g(n) is the minimal cost of reaching the current state n from the start state. The function h(n) is the estimated cost to reach a goal state from the current location n. A* requires that its heuristic is admissible, meaning that given any state n, h(n) <= h*(n), where h*(n) is the true cost from n to the goal. This requirement of admissibility is what discerns A* from the 'A' algorithm (Hart et al., 1968), from which A* is derived. A heuristic is said to be admissible if never overestimates the cost to a goal state. A heuristic is said to be perfect when h(n) = h*(n) for all n, and is said to be null when h(n) = 0 for all n. A* is complete when there are only finitely many nodes, and admissible due to its requirement on the heuristic function. So given a finite search space and an admissible heuristic, A* will always find the optimal path because every potentially optimal path will be checked until the goal is reached.

Even though A* will always find the optimal path, the definition of that optimal path will depend upon the heuristic used. If a different heuristic is used, it is possible that a different path will be chosen as optimal. The number of nodes expanded in an A* search depends on the heuristic used. If 0 <= $h_1(n)$ <= $h_2(n)$ <= h*(n) for all n, then h_2 is said to be more informed than h_1. A* using h_2 will expand possibly fewer and no more nodes than A* using h_1. The closer h(n) is to h*(n), the fewer nodes that need to be expanded by A*. c(n, n') refers to the cost to traverse from state n to n'. When provided with a monotone heuristic, A* does not require a CLOSED list. Monotone heuristics are heuristics whereby given any successor n of m, then h(m) – h(n) <= c(m, n), where c(m,n) is the cost to traverse from m to n. The A* algorithm is defined as such, with the use of a CLOSED list:

```
1.Put the start node S on the nodes list, called OPEN
```

2.If OPEN is empty, exit with failure

3.Remove from OPEN and place on CLOSED a node n for which f(n) is minimal.

4.If n is a goal node, exit (trace back from n to S)

5.Expand n, generating all its successors and attach pointers back to n. For each successor n' of n

a) If n' is not already on OPEN or CLOSED estimate h(n'), g(n')=g(n)+c(n,n'), f(n')=g(n')+h(n'), and place on OPEN.

b) If n' is already on OPEN or CLOSED, then check if g(n') is lower for the new version of n'. If so then:

i. Redirect pointers backward from n' along path yielding lower g(n')

ii. Put n' on OPEN

6.Goto 2

*2.2 SMA**

Simplified Memory-Bounded A* (SMA*) (Russell and Norvig, 1994) is aimed at addressing the memory problems of A*, while restricting the amount of memory used. SMA* allows for setting a restriction on the amount of nodes that are allowed in memory at any one point. In SMA*, the initial node is added, all its successors are then added, with the cost of the initial node being that of its lowest cost successor. This process is then repeated for each child until the maximum memory limit is reached. At this point if the new node being investigated has a lower cost than a node currently in memory, then the node in memory is dropped, with its parent taking note of its cost in case it needs to be re-explored later, and the new node is added. If the node to be added is not of lower cost than an existing node, then its parent takes note, possibly changing the lowest cost of forgotten children. SMA* always expands the lowest cost deepest-node, re-expanding forgotten nodes if they turn out to have the currently lowest cost of the nodes in memory.

Given enough memory, SMA* will find an optimal path, and will be complete. The calculation of what is enough memory cannot be predetermined, as the depth of a possible solution is unknown. SMA* is also a static algorithm, and thus unable to handle dynamic environments with changing edge costs or accessibility.

*2.3 LPA**

LPA* (Koenig and Likhachev, 2002) is a replanning method that is an incremental version of A*. The name is an analogy to "life-long learning". Life-long learning may also be referred to as continuous planning. It repeatedly finds the shortest path between the agent's current location and the

goal state. It does this while taking in consideration changing edge costs, obstacles within the environment, and nodes being added or removed. The initial search of LPA* is the same as that of A*, breaking ties in favour of nodes with smaller g-values if they have the same f-values. Subsequent searches are faster due to LPA* reusing parts of the previous search tree which are identical to the current one.

There are a lot of search environments which are not static. Environmental edge costs change, and obstacles can also play a part, blocking access to nodes. As these changes occur, static path-planning algorithms such as A* are not able to adapt. LPA* aims to maintain an optimal path as new information is perceived. LPA* also aims to find these shortest paths more quickly than A* and DynamicSWSF-FP (Ramalingam and Reps, 1996). It is a modification of DynamicSWSF-FP that searches from the start vertex to the goal vertex, stopping when it has found a shortest path, at which point it becomes an incremental version of breadth-first search. LPA* applies to pathfinding problems on known finite graphs where edge costs increase or decrease. Unlike A*, LPA* does not maintain a CLOSED list as it uses consistency checks to avoid vertex re-expansion.

LPA* is tested using a simplified route planning scenario. Testing is done on an 8-connected grid world with cells whose traversability changes over time. All nodes are either traversable with an edge cost of 1, or not traversable at all. The algorithm always determines the shortest path between two given nodes, knowing both the topology of the grid and the block states of nodes. Euclidean distance is the heuristic used to determine the distance between two vertices. LPA* was compared to Breadth-first search, A*, and DynamicSWSF-FP. The authors conclude that LPA* takes more time than A* per vertex expansion. They also claim that LPA* is more efficient than A* in some situations in terms of both runtime and vertex expansions, especially if the changes are slight and are close to the goal.

*2.4 Focused D**

Focused D* (D*) (Stentz, 1995) plans optimal traverses in real-time by using newly discovered information to refine paths. Focused D* is an extension of D* that focuses the repairs to significantly reduce the total time to plan an initial path, as well as subsequent planning operations. For dynamic search algorithms such as D* and LPA*, a path is considered optimal if it is the shortest path, assuming that all knowledge about edge costs and nodes states are correct (and will remain so). Focused D* uses the same heuristics as A* to propagate cost increases and Focus cost reductions. The algorithm uses a biasing function to compensate for agent movement between replanning operations. Focused D* places a range-restriction on D*. Whereas D* accounts for all changes in the search space, Focused D* assumes the agent can only detect local changes. According to the authors' tests, Focused D* is shown to be more efficient in terms of runtime than normal D*, while maintaining optimality (according to its knowledge). Focused D* and D* have become interchangeable, in a manner similar to the A algorithm and A*.

2.5 D Lite*

D* Lite is a is a sensor-based replanning variant of D*. The algorithm is targeted towards environments which are not completely known. In D* Lite, the agent always plans the shortest path from its current location to the goal, assuming that any unknown terrain is traversable. Terrain is assumed to be unknown if the sensors of the agent have not confirmed its traversability. Even though the algorithm is designed for incompletely known terrain, it still has to know the location of nodes within the environment, even though not knowing their traversability. The algorithm terminates eventually because either it follows the planned path to the goal, or the agent discovers the true costs

and traversability of the nodes, which can happen only once per edge. It will terminate in the latter case when it can determine that no path exists.

The algorithm replans its path when it detects that the current path is not traversable. While algorithmically different than D*, D* Lite presents the same paths as D*. Unlike A*, D* Lite does not demand an admissible heuristic. D* Lite is also based upon LPA*, extending it to where the goal of the search changes between replanning stages. It also switches the search direction used in LPA*, searching instead from the goal state to the current state of the agent. According to the authors' experiments, D* Lite appears to be even more efficient than Focused D* in some cases.

*2.6 Anytime Repairing A**

Anytime Repairing A* (Likhachev et al, 2003) is an anytime algorithm based upon A*. Sometimes a planning problem is complex, and there is a limited amount of time available for the computation of paths. In these cases, an agent must be willing to settle for a suboptimal solution. Anytime algorithms initially produce highly suboptimal solutions quickly, and refine these as time allows (Dean & Boddy 1988, Zilberstein & Russell 1995; Zhou & Hansen 2002, Likhachev et al 2003). A*-based anytime algorithms usually inflate the heuristic used by A*, which often provides significant speed-ups (Bonet and Geffner, 2001). To achieve this speedup, these algorithms need to sacrifice optimality. A property of A* is that if a consistent heuristic is multiplied by a factor ε, then the generated solution is guaranteed to be within ε times of the optimal solution (Pearl, 1984).

One of these A*-based anytime algorithms is that produced by Zhou and Hansen (Zhou & Hansen 2002). This algorithm would quickly produce a bounded solution and refine it over time. The problem with their algorithm is that it maintains no control over suboptimality bounds while the path is improved. Anytime Repairing A* (ARA*) performs a succession of A* searches, reducing the bounds

used in each successive search. Each search reuses information from previous searches. The authors were able to show that this algorithm is more efficient than other similar algorithms such as Zhou and Hansen's.

Initially, ARA* performs an A* search with an inflation factor of ε_0, expanding each node at most once. On subsequent searches, ARA* only considers states whose costs at the previous search are now invalidated by the new value of ε. If a state becomes inconsistent because the value of a neighbour has changed, then the state is placed on a list, INCONS. This list contains all the inconsistent states which have been already expanded. On conclusion of the current search, the items in INCONS are placed on the fresh priority queue, which is based on the new value of ε and is used for the new search. Only a small amount of computation is required between searches, as a) since only the inconsistent states are reconsidered, most of the previous search data is reused. and b) each state is expanded at most once per iteration.

*2.7 Anytime D**

Anytime D* (Likhachev et al, 2005) is aimed at environments that require both complex planning, and are dynamic. Anytime D* (AD*) is an attempt to combine ideas from both ARA* and D*. It performs searching using progressively decreasing inflation factors similar to ARA*. It handles environment changes in the same manner as D* Lite, placing affected states on the OPEN queue with priority equal to the minimum of its previous value and its new key. When edge costs changes substantially, it may be too expensive to repair the current solution to maintain ε bounded suboptimality. In that case, AD* increases ε to produce a less optimal solution quickly. The determination of what is is substantial change is dependent upon the application. Edge cost increases can cause states to become underconsistent, requiring them to be placed back on the OPEN queue.

Uninflated key values must be used to ensure that changes caused by underconsistent states are propagated. This leads to under- and over- consistent states needing different methods to calculate their keys.

AD* is able to support the dynamic traversal of the environment while handling the changes dynamically. AD* has an advantage over both ARA* and D* Lite in that it only processes those states that were inconsistent either at the beginning of the current search, or during the search. There exist cases where it may be better to replan from scratch as opposed to fixing the current search. This may happen if the environment changes were significant and the last complete search was a time ago. Another problem with AD* is that every time ε changes, the OPEN queue needs to be reordered due to the new recomputed key values. This operation can be computationally expensive.

*2.8 Adaptive A**

Incremental versions of A* are able to guarantee the quality of subsequent searches. For this reason, many new incremental search algorithms are based upon A*. LPA*, D*, and D* Lite are some well-known cases of this type of algorithm. These algorithms suffer from a problem in that it is difficult to prove them correct, and also that there are cases where they are slower than standard A*. This paper aims to address this problem by creating a principle whereby algorithms update the heuristics over time to make them more informed, thus making future searches more Focused. This principle is incorporated into A* to create Adaptive A* (Koenig and Likhachev, 2006). Adaptive A* (AA*) is shown to expand no more nodes than standard A*, and thus cannot be slower than A*. This claim does not take into effect some small setup and other actions that AA* need to operate. The principle behind AA* is based on properties of A*, whereby given the following:

1. Let gd[s] be the cost of a minimal path from s to the goal state

2. f* = gd[start], and is the cost of the minimal path produced by A*
3. For any state s expanded by A*, h[s] is the consistent heuristic of said state
4. g[s] and f[s] denote the g-value and f-value respectively
5. Therefore, $f* \leq g[s] + gd[s]$ and $f^* - g[s] \leq gd[s]$.

This means that f* - g[s] is an admissible estimate of the goal distance of s. Thus it can be used as a new heuristic for state s. This heuristic is shown to be no smaller than h[s], and thus dominates it. The result of this is that any subsequent A* search using this heuristic is guaranteed to expand no more nodes than the same search using h[s].

Adaptive A* is aimed at environments where the goal location does not change, while edge costs may increase dynamically. It uses A* to search, while providing more Focused heuristics for subsequent searches. As AA* expands fewer nodes on each subsequent search, and thus becomes faster over time. There are two versions of AA*. Eager AA* does not update the heuristics of states that are still in the A* OPEN list at the completion of the search since their new heuristic cannot be larger than their old one. The disadvantage of updating these states after the search is that some of them may have no effect on future searches. The second version of AA*, Lazy AA*, remembers the g-values of expanded nodes as well as the cost of the path returned by A*. This information is then reused to adjust the heuristic of these nodes if they are to be expanded in subsequent searches. Experimentally, AA* is shown to be about 10% faster than A*, even though those results are implementation-dependent.

*2.9 Fringe Saving A**

Fringe-Saving A* (Sun and Koenig, 2007) is an incremental version of A*. The algorithm repeatedly finds shortest paths in dynamic environments where edge costs increase or decrease. The initial search of Fringe-Saving A* (FSA*) is the same as that of A*. The algorithm then reuses a part of

the search in the subsequent search. FSA* restores the OPEN list of A* at the point where it would deviate from the current search (when a changed edge cost is perceived). FSA* then proceeds to continue the A* search from this point.

The state of an A* search is determined by the contents of the OPEN and CLOSED lists and the g-values and parent pointers of their nodes. FSA* keeps track of when nodes are expanded within the A* search using a function ExpandedId(s), which is the order in which s was expanded. Thus, ExpandedId(start) = 1 and ExpandedId(n) for an unexpanded state n is infinity. If a given state s' changes between searches, then FSA* restores all states s, with ExpandedId(s) < ExpandedId(s'). FSA* does not need to replan if the goal cell is reusable. This is because the shortest path from the previous search is also a shortest path for the current search. If the start cell is not reusable and blocked, then there does not exist a path and the algorithm can stop. If the start cell is not reusable and unblocked, then the algorithm performs a new complete A* search.

FSA* places cells that are unblocked and reusable on the OPEN list if they border one or more reusable cells. The algorithm identifies the anchor cell by following parent pointers of reusable cells from the goal to the start cell until a non-reusable cell is reached. This non-reusable cell is the anchor cell. FSA* then identifies fringe cells, which are the reusable cells from the previous search. FSA* stops when it is about to leave the anchor cell for the second time in the same direction. The g-values and parents pointers are then restored for reusable cells, and corrected for the cells on the OPEN list, if necessary. A* is then restarted from this position, and proceeds as usual. According to the authors experiments, FSA* is faster than LPA* and A* in some cases.

*2.10 Delayed D**

Delayed D* (Ferguson and Stentz, 2005) is a variant of D* that requires about half the

computation time. Delayed D* (DD*) incrementally repairs previous paths, focusing these repairs towards the agent's current position. As initial paths planned by variants of A* are not likely to remain accurate in dynamic environments, incremental planning algorithms are necessary. Focused D* and D* Lite are the most widely used of these algorithms. This is due to their efficient use of heuristics and incremental updates. D* has been experimentally shown to be between 0 and 50% more efficient than replanning from scratch when path environment information changes.

DD* focuses on solving the exact same problems as D* and D* Lite. The goal of DD* is to solve these problems more efficiently. DD* aims to do this by being even more restrictive in terms of which nodes are expanded than D* Lite. It is possible that in D* Lite, many more states may be expanded than is necessary. The logic is that, even if an inconsistent state has a lower priority than the start state, which would normally cause it to be expanded by D* Lite, its expansion may not affect the optimality of the current solution. DD* aims to determine whether an inconsistent state is of consequence to the validity of the current solution without propagating the inconsistency.

When a state becomes underconsistent due to cost increases, they are guaranteed not to have an effect on the previous solution if they are not on the solution path. This cannot be guaranteed for states made overconsistent due to cost decreases. DD* thus processes underconsistent states much more selectively than overconsistent states. As in D* Lite, when cost changes occur, the rhs-values of all affected states are updated and overconsistent states processed immediately. Rhs-values are dependent upon the distance from the starting location of its predecessors. The value of rhs(s) is 0 if s is the starting location, otherwise it is $\min_{s} \in P$ red(s) (g(s) + c(s , s)). The processing of underconsistent states is ignored. After all the overconsistent state changes are propagated, the environment is then checked for underconsistent states. These states are then added to the priority queue and their updated values propagated through the state space. This new solution path then needs to be repeatedly checked for underconsistent states until only consistent states are returned.

The advantages over D* Lite are twofold. Because DD* delays processing of underconsistent states, some of these states may never need to be processed. Also, due to the delay of the processing, it is possible to perform in one propagation changes that would take multiple propagations in D* Lite. According to the experiments of the authors, DD* performs much better than D* Lite. DD* expands expands less than half the nodes that D* Lite does, as well as taking half the time computationally.

Chapter 3: Range-Limited A*

3.1 Introduction

Range-Limited A* (RLA*) is a variant of the A* algorithm. RLA* is an investigation of performing pathfinding searches in grid environments while using as little memory as possible. The investigation ranges from using no memory at all, to using less memory than memory Focused algorithms like SMA*. The investigations cover its performance within static environments, with uniform and non-uniform edge costs. The final intent is to compare RLA* to other established pathfinding algorithms such as A* and SMA*.

Initially, the goal of RLA* was to explore the possibility of non-player characters (NPC) within games being able to find the goal state without complete knowledge of the environment. The range factor attempts to restrict A* within a sight range, as is the case with agents using local sensors. The alloted range can be dependent upon the character and its context. In the context of grid environments without obstacles, RLA* can be described as such: Starting from the initial state, the agent looks around for the best location observable. This best location takes into account the cost to get there, as well as the estimated cost to the goal. The agent then navigates to this chosen location, and continues the process, eventually reaching the goal. In this environment with no obstacles, the agent would always arrive at the goal location, even though not necessarily in optimal time. Like A*, RLA* is able to perform without a CLOSED list, but this is not the general case, wherein the given heuristic may not be monotone. Below is the algorithmic description of this initial version of RLA*:

FIND-PATH

1. Put the current node S on the nodes list, called OPEN.
2. If OPEN is empty
    1. If BEST is None, exit with failure
    2. exit, tracing back pointers from BEST to S as P, setting expected values
3. Remove from OPEN and place on CLOSED a node n for which f(n) is minimum.
4. If n is out of range
    1. set q = parent of n
    2. if f(q) < f(BEST) or BEST is none
        1. BEST = q
    3. Goto 2
5. If n is a goal node, exit (trace back pointers from n to S as P, setting expected values)
6. Expand n, generating all its successors and attach to them pointers back to n. For each successor n' of n.
    1. If n' is not already on OPEN or CLOSED estimate h(n'), g(n')=g(n)+c(n,n')
    2. If n' is already on OPEN or CLOSED, then check if g(n') is lower for the new version of n', if so then:
        1. Redirect pointers backward from n' along path yielding lower g(n')
        2. Put n' on OPEN.

```
   7. Goto 2

NEXT-STEP  - current path is specified as P
   1. If current state is goal state exit
   2. Set n to next node in P
   3.  current state = n
```

3.2 Initial comparisons

It can be noticed that the initial RLA* algorithm is very similar to the A* algorithm, in fact this algorithm uses A* to locate the best location, but within the allowed range of sight. It is trivial to show that RLA* is complete given a static grid environment. This is the sort of environment in which A* and SMA* are designed for, so I compare RLA* to these algorithms. To compare, 1000 test environments are created, each consisting of a 100x100 grid. The starting and ending nodes are randomly chosen and are the same for each of the algorithms. Even though it is possible to run A* without the use of a CLOSED list, we test it using one. Node-traversal costs in the environment are all 1. Comparisons are made regarding the path cost from the start to the goal, as well as the maximum amount of memory used at any time during the search. The calculation of the amount of memory used is the sum of the amount of items stored on the OPEN and CLOSED lists for each algorithm. The range of sight alloted for an RLA* agent is 3 units, with distance being determined by euclidean distance.

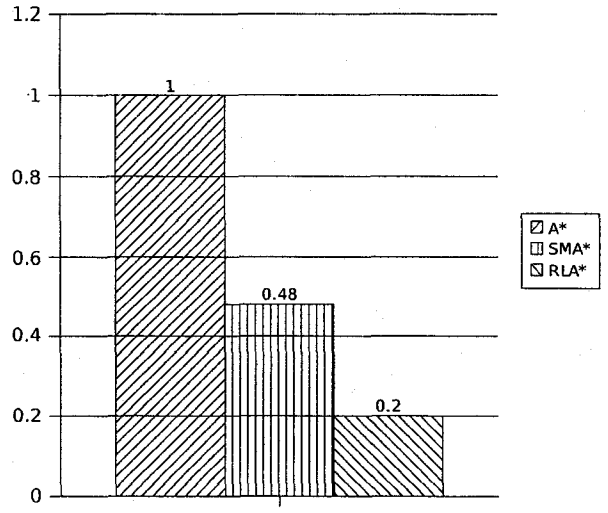

Figure 1: Memory Comparison in static, uniform, non-blocked environment

In the experiments, RLA* is able to find the same-cost paths as A* and SMA* using significantly less memory as shown in figure 1. RLA* used only 48% of the amount of memory used by SMA* and 20% of that used by A*. SMA* is always able to find the same path as A* when allowed the same or more nodes than are in the A* path. In the experiments, the reason that RLA* is able to find the exact same path is more a reflection of the testing environment than of the algorithm. The costs of the nodes in the environment do not vary significantly enough for there to be a path that is significantly better than the ones found by RLA*. It is possible for there to exist a significantly faster path after an initial rough patch beyond the horizon of the agent's range of sight, in which case the agent will take a path that is globally suboptimal. This case can be ignored as the RLA* agent is not able to detect these nodes.

While A* is globally optimal, RLA* is necessarily not so, because it cannot perceive things beyond its range of sight. Different incremental algorithms use different criteria to determine

optimality, including optimality based on the agent's perception, as opposed to global optimality. Within its range of sight, RLA* is optimal, it will always find the most efficient path to its ending location. This decision making process being limited by the current knowledge of the agent.

3.3 Admissible Heuristics

A* requires that the heuristic used is admissible. The algorithm will still be complete if the heuristic is not admissible, but it will not be optimal. A* with a null heuristic is equivalent to best-first search. This means that A* will still work if given a heuristic that has minimal effect on the f-value of nodes. Take for example an environment where edge costs average 25 units, while the heuristic is 1 for each edge. Unlike A*, RLA* will not work if the heuristic is insignificant in comparison to the true cost, h*(n). When a very weak heuristic is used for RLA*, for example a null heuristic, it becomes the case that the algorithm is driven primarily by edge costs. This, in addition to RLA* range restriction, further reduces the completeness of the algorithm.

This problem was demonstrated by running 100 tests with a null heuristic on a 100x100 grid with uniform costs of 1 and no blockades. The starting and ending locations were selected at random. Any path found by the algorithm should be within 200 units. The purpose of the experiment is to observe how many of the tests fail, with failure being when the path of RLA* has surpassed 200 nodes. After running the tests, RLA* was able to find the goal within the constraints only 52% of the time, with some of those being because they were close enough that the algorithm 'lucked out' in finding the goal. To demonstrate that it was not simply a result of the fact that it was a null heuristic used, the tests were reran, this time with a minimal heuristic. The edge-costs were non-uniform, ranging from 1 to 100 for this experiment, and the heuristic was 1 for each edge between the location and the goal. The results were even more dire in this case, showing only 8% of cases succeeding.

Whereas A* is driven by f-values, a solution for RLA* is to be driven by heuristic, with lower edge-costs being the tie-breaking factor. The algorithm will still be unable to work reliably in cases where a null-heuristic is used, but would work in other cases. Unlike A*, which needs an admissible heuristic, RLA* then would not. As long as monotonicity is maintained, the algorithm would still be able to navigate its way towards the goal state. To test this, the previous experiment was re-ran, but this time, the algorithm was driven using the new criteria. With the change, the algorithm was able to find the goal state in 100% of cases. The only effective difference is that RLA* now chooses the node at the edge of its horizon based on heuristic first and then f-value.

3.4 Blockades

Up to this point the grid environment being tested consisted of nodes that were all unblocked. A grid with blocked nodes is similar to a grid without such nodes. Simply remove the edges between unblocked nodes and blocked nodes, after which at least one graph component will remain with all nodes being unblocked. The component containing the start state is the primary component and the only one that would matter. If the goal node is not within this component then there exists no solution to the problem. For the sake of testing, the assumption will be that a solution does exist, this being verified by A* finding a path. The problem that arises within grids with blocked nodes is that there may exist only a few edges connecting components. In such cases it may be necessary for the algorithm to backtrack to properly navigate to the goal. As RLA* is currently, it is possible for the algorithm to become stuck in an infinite loop.

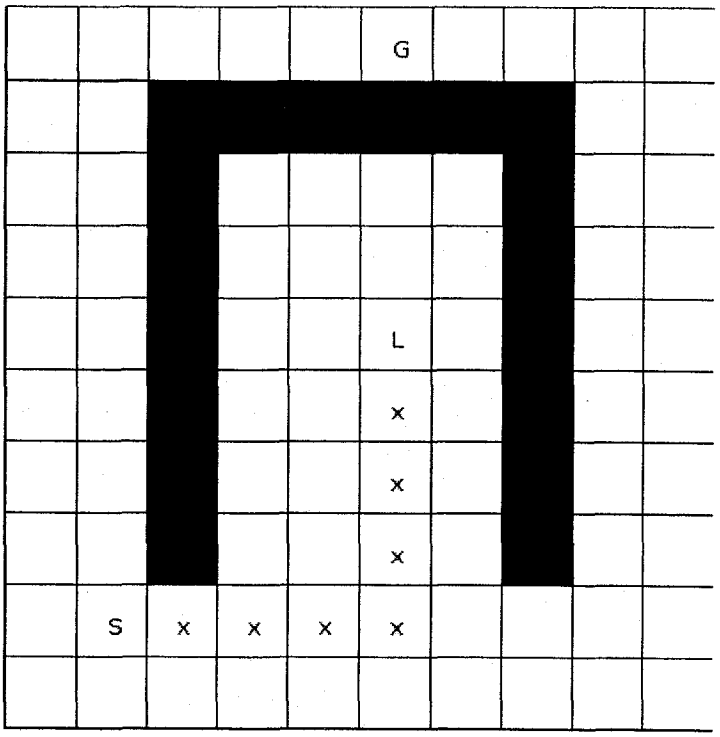

Figure 2a: RLA becoming stuck*

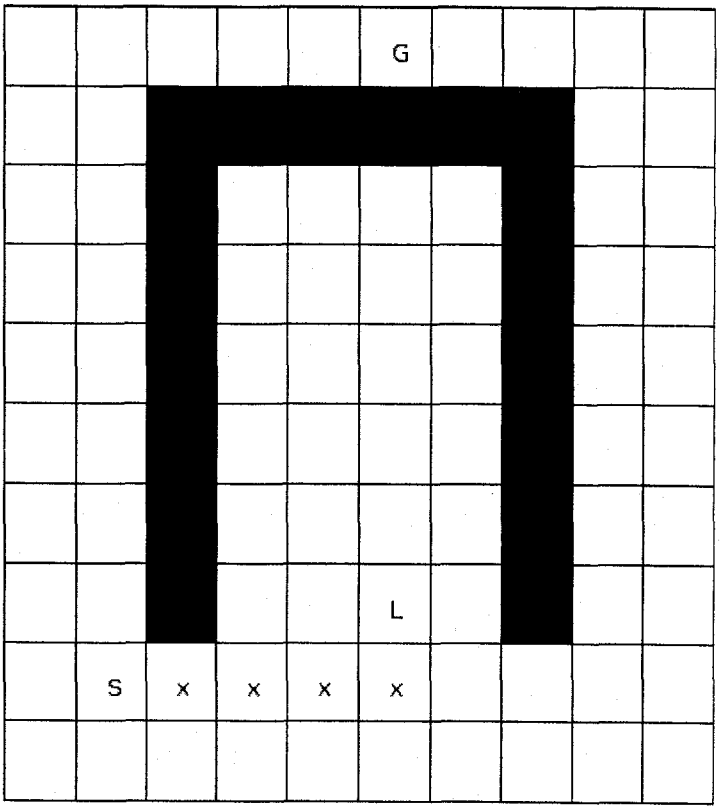

Figure 2b. RLA becoming stuck*

Above is one such such scenario in which the algorithm is unable to proceed. S and G are the start and goal states respectively. States visited by the agent are marked with an x, and L indicates the current position of the agent. The agent is assumed to have a range of sight of 3 nodes measured by euclidean distance. From its current location, the agent can observe that the obvious nodes leading towards the goal are dead ends. By the algorithm, the best location for the agent to proceed to is that shown in Figure 2b. The problem is that because the agent does not currently possess any memory, when the algorithm is ran from the state in Figure 2b, the algorithm will return to the state shown in Figure 2a. This process would continue, with each state returning the other as the best path to take.

The solution is the implementation of a backtracking method. Other pathfinding algorithms such as Hill-Climbing and Greedy best-first search also need backtracking methods. One such backtracking method is Tabu List (Glover, 1989). Tabu lists maintain a set of states that should not be revisited, usually for a specified period of time. Placing a restriction, or tenure, on the amount of time a node is allowed to remain within the Tabu list would lessen the effect maintaining such a list would have on memory. This possibility was explored, but a simple flaw was found that prevented such a method from working. In Figure 3a, we can see the agent navigating in a normal RLA* fashion, upon reaching the location illustrated, the agent can see that its path is blocked. Using the tenure method, these nodes would be placed on the Tabu list and the agent would return to the start state, exploring in the other direction as illustrated in Figure 3b. At the new location the agent detects no forward movement, placing the current path on the Tabu list and returning to start again. The potential problem is that depending on the tenure, the items that were placed on the Tabu list in Figure 3b, may have been released based on the expiration of their tenure. This would cause the agent to alternately retry both these failed paths perpetually.

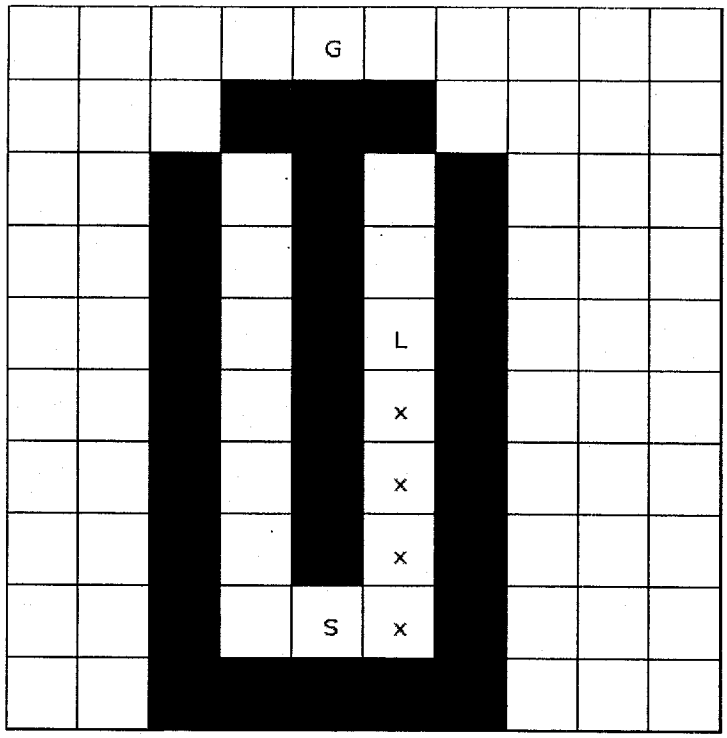

Figure 3a. Tabu initial search

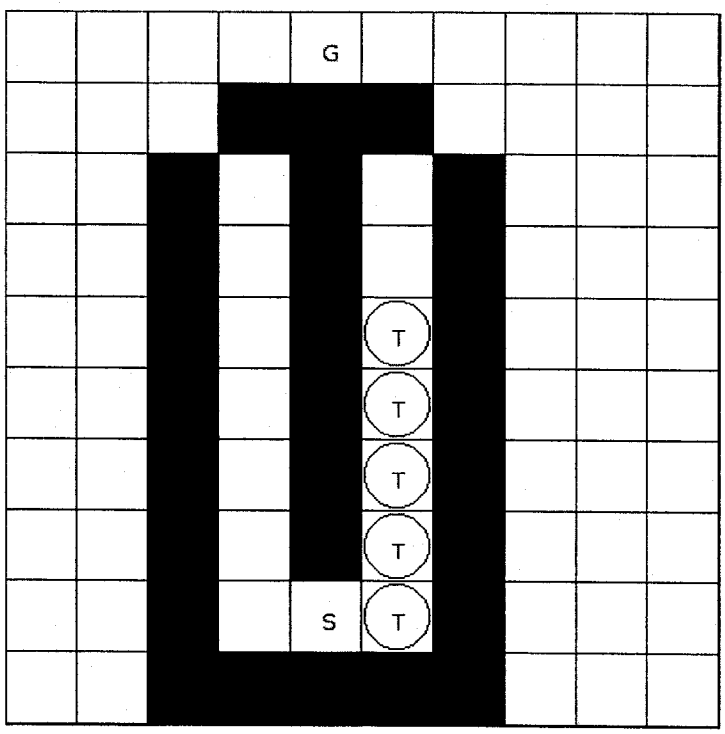

Figure 3b. Tabu after backtracking

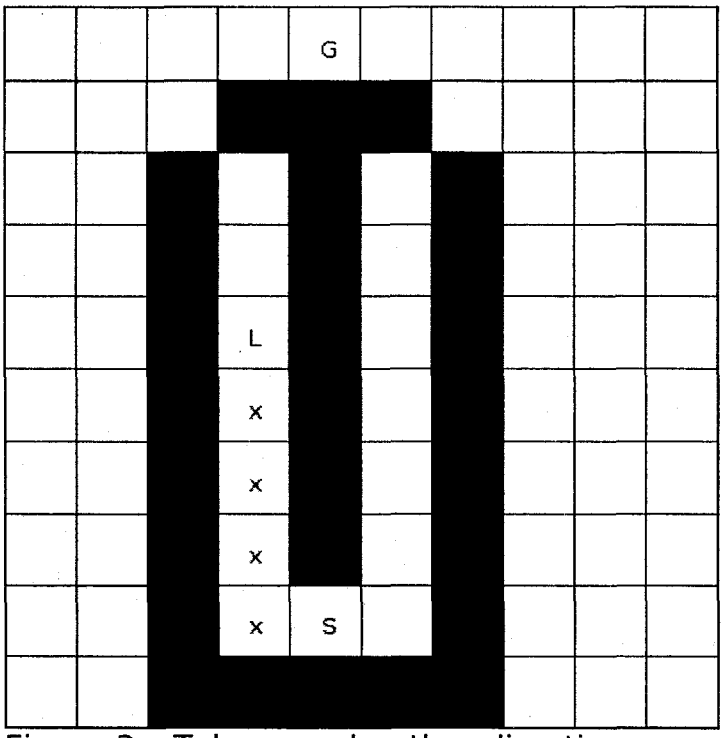

Figure 3c. Tabu search, other direction

The approach chosen is dubbed Landmark Tabu. This approach works by placing beginning search locations on a landmark list. When horizon nodes are found in FIND-PATH, we check if it is already on the landmark list. A horizon node is a node at the edge of the range of sight for the RLA* agent. A horizon node is also required to have accessible child nodes according to the local search. We maintain a list of these horizon nodes, while still searching for the best horizon node that is not on the landmark list. If a best non-landmark node is found, then it is the new destination for the agent.

If there doesn't exist a node that is not a landmark, then all nodes within the horizon list are placed on the tabu list. This indicates that all paths following the starting location lead nowhere. Nodes placed on the tabu list are not allowed to be ending nodes for future searches. Introduced to FIND-PATH are the TABU, HORIZON and LANDMARKS lists. Below is the modified FIND-PATH procedure with changes in bold.

FIND-PATH

1. Put the current node S on the nodes list, called OPEN.
2. **If not the first search**
    1. **BEST = previous search start location**
3. If OPEN is empty
    1. If BEST is none, exit with failure
    2. **If BEST is in LANDMARKS**
        1. **add all from HORIZON to TABU**
        2. **add S to TABU**
        3. **remove S from LANDMARKS**
    3. **add BEST to LANDMARKS**
    4. exit, tracing back pointers from BEST to S as P, setting expected values
4. Remove from OPEN and place on CLOSED a node n for which f(n) is minimum.
5. If n is out of range
    1. set q = parent of n
    2. **if q is in TABU**
        1. **Goto 3**
    3. **Put q in HORIZON if it's not in LANDMARKS**
    4. **if BEST is none**
        1. **BEST = q**
    5. **else if BEST is in LANDMARKS**
        1. **If q is not in LANDMARKS or f(q) < f(BEST)**

1. **BEST = q**

6. **else if f(q) < f(BEST)**

1. **BEST = q**

7. Goto 3

6. If n is a goal node, exit (trace back pointers from n to S as P, setting expected values)
7. Expand n, generating all its successors and attach to them pointers back to n. For each successor n' of n.
   1. If n' is not already on OPEN or CLOSED estimate h(n'), g(n')=g(n)+c(n,n')
   2. If n' is already on OPEN or CLOSED, then check if g(n') is lower for the new version of n', if so then:
      1. Redirect pointers backward from n' along path yielding lower g(n')
      2. Put n' on OPEN.
8. Goto 3

To test the performance of RLA* with the use of a Tabu List, we ran it on a 20x20 grid. The environment for our tests is static, with all blockades being generated before the start of the experiment. The starting and ending nodes are not allowed to be blocked, as this would mean that there exists no path. A total of 500 tests were ran, with 100 of them with 5%, 10%, 15%, 20%, and 25% of nodes blocked. The goal of the experiment is to observe the performance of RLA* in comparison to the the established A* and SMA* algorithms. To get a proper reflection of the path cost comparison, all nodes will have a uniform traversal cost of 1. We'll also ensure that the minimum path returned by A* is 20

nodes, as this will make the results more significant.

As memory is one of RLA* primary focuses, the maximum amount of nodes stored will also be measured. In the case of A*, this will be the maximum amount of nodes on the open list. The length of the path returned by A* will be used as the memory allowance for SMA*, guaranteeing the minimum. The memory used by RLA* will be measured as the maximum of nodes on its open list plus the size of the TABU and LANDMARKS list for any iteration of its search. The performance of A* will determine if a particular experiment is valid. If A* is unable to find a path through the generated grid, then the test is invalid and discarded.

In the experiments, RLA* was able to find the path to the goal in all cases. There was a pattern whereby the length of the RLA* path got longer as the number of blocked nodes increased. This can be attributed to the fact that RLA* would meet more dead-ends that it is unable to predict. This is due to the blockade being beyond the horizon of the agent. RLA* used significantly less memory than both A* and SMA*. The amount of memory in relation to A* also increased as the amount of blocked nodes increased. There was no apparent relation between RLA* memory use and SMA* memory use as blocked node count increased. It was simply coincidental that no paths were greater than 39 nodes in the experiments.

Table 1: RLA* with Landmark Tabu over multiple obstacle percentages

block %	path	mem - A*	mem - SMA*
5%	1.02	0.09	0.5
10%	1.04	0.1	0.5
15%	1.06	0.11	0.49
20%	1.32	0.13	0.55
25%	1.36	0.13	0.54

length range: 20-39

3.5 Dynamic Environments

In dynamic environments, edge-traversal costs may change, as well as the blockage state of a vertex. Since RLA* is restricted within its range of sight, it only has to handle such local changes. Three ways in which edge-traversal cost changes can be handled are:

1. Ignore the cost change and proceed to the end-point as planned
2. Maintain the optimal path to the end-point, similar to D*
3. Replan, create a new path to the new best end-point visible

The problem with the first approach is that it does not work for when the next node becomes blocked. It also does not take into account the magnitude of the cost change. It may make sense to completely replan if a change is significant enough. The second approach works, but the overhead required to maintain the optimal path may be more expensive than simply replanning, especially in environments that do not contain many nodes within the agents range of sight. The third approach's drawback is that it also doesn't take into account the magnitude of the cost change. It will replan if a small change is made, resulting in many replanning episodes in an environment where many small, yet insignificant, changes occur.

The solution chosen is a combination of the first and third approaches. We institute the idea of a change threshold. Thus, if an edge-cost changes significantly, a new path will be planned from the current location. The definition of a significant change is equivalent to the threshold, which is context dependent. In an environment where the range of the agent is large, The second approach may be more sensible, in the same way that D* is shown to be better in large environments than full-replanning algorithms. (Stentz, 1995)

3.6 Completeness

One of the properties of the Tabu list is that any node on it, must have been unblocked when added. Thus, no nodes that are placed on the Tabu list will need to be removed. If a path did not previously exist from the Tabu node, then the agent would be unable to detect that a new path now exists from such a node. This is due to the fact that the agent never initiates a search from a Tabu location. Therefore, while it is possible that a previously dead-end path is now the path to the goal state. It is not possible for our agent to perceive this, and as far as the agent knows, there is still no path from such a location. As a result of this, the agent will report that a path does not exist to the goal, which may be incorrect due to changes unknown to the agent. The only required change to RLA* is in NEXT-STEP, where we check if the next node has changed significantly since we searched, in which case we search again from the current location. Below is the final RLA* algorithm. A comparison between RLA*, my own MD* Lite, and the well-established D* and D* Lite is performed later in the section on MD* Lite.

3.7 Final RLA Algorithm*

```
FIND-PATH
  1. Put the current node S on the nodes list, called OPEN.
  2. If not the first search
     1. BEST = previous search start location
  3. If OPEN is empty
     1. If BEST is none, exit with failure
     2. If BEST is in LANDMARKS
```

1. add all from HORIZON to TABU
2. add S to TABU
3. remove S from LANDMARKS

3. add BEST to LANDMARKS
4. exit, tracing back pointers from BEST to S as P, setting expected values

4. Remove from OPEN and place on CLOSED a node n for which f(n) is minimum.
5. If n is out of range
   1. set q = parent of n
   2. if q is in TABU
      1. Goto 3
   3. Put q in HORIZON if it's not in LANDMARK
   4. if BEST is none
      1. BEST = q
   5. else if BEST is in LANDMARKS
      1. If q is not in LANDMARKS or f(q) < f(BEST)
         1. BEST = q
   6. else if f(q) < f(BEST)
      1. BEST = q
   7. Goto 3
6. If n is a goal node, exit (trace back pointers from n to S as P, setting expected values)
7. Expand n, generating all its successors and attach to them

pointers back to n. For each successor n' of n.

1. If n' is not already on OPEN or CLOSED estimate h(n'), g(n')=g(n)+c(n,n')
2. If n' is already on OPEN or CLOSED, then check if g(n') is lower for the new version of n', if so then:
   1. Redirect pointers backward from n' along path yielding lower g(n')
   2. Put n' on OPEN.
   8. Goto 3

NEXT-STEP - current path is specified as P

1. If current state is goal state exit
2. Set n to next node in P
3. **if n's expected cost has significantly changed or n has become blocked**
   1. call FIND-PATH
   2. Set n to next node in P
4. current state = n

3.8 Conclusion

The originally presented version of RLA* (version 1) was unable to work accurately with null or weak heuristics. The new version is also unable to perform in these situations due to the lack of direction from the heuristic function. As the agent is not able to remember previous search locations,

the lack of a significant heuristic is detrimental. On the other hand, RLA* does not require an admissible heuristic. A heuristic that overestimates the cost to the goal location is valid in RLA*, and even preferred, as this gives the agent an even more useful guide toward the goal.

In its final form within static environments, RLA* is very similar to Depth-First Search. If only the landmark nodes are considered, then RLA* is similar to Depth-First Search with the addition of a heuristic-influenced cost function. Nodes already on the Landmark list are considered visited. Thus, in static environments, RLA* is similar to heuristic driven Depth-First Search (DFS), With DFS nodes generated by an A* range-based local search. As node state is not changed within RLA*, the Tabu list is simply an extension of the Landmark list, but for nodes that have been determined to have no possible path to the goal. RLA* therefore has a time complexity of $O(|V| + |E|)$ between starting search locations. The local search complexity is dependent upon the heuristic used, which can be exponential in the worse case, but is polynomial in terms of path length when the deviation between h(n) and h*(n) does not grow faster than log(h*(n)). This cannot be expanded to include the general case due to the backtracking that the algorithm needs to do.

Given a sufficiently informed heuristic, RLA* is complete in static environments. It is able to use significantly less memory than both A* and SMA*, while suffering reasonable increases in path length. The algorithm is appropriate for environments in which storing the entire search in memory is not feasible. It is also appropriate for cases in which state transitions are not able to be determined initially, or only when the agent is close enough to the state. The major difference between RLA* and other informed search algorithms such as D* and LPA* lie in RLA*'s lack of knowledge about the state of the map. D* for instance, deals with incomplete information, but still having the knowledge of the map, even if not accurate information regarding costs and blockage status.

Chapter 4: Memory-Bounded D* Lite

4.1 Introduction

Proposed is a variant of D* Lite with memory bounded features akin to SMA*. D* Lite is based upon Focused D*, which in turn is a dynamic A* variant. Being based upon A*, D* Lite suffers from similar high memory usage as A*, as all nodes which have been discovered but not yet expanded need to remain in the algorithms' 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 most nodes within the map are placed on the queue, in the worse case. Memory-Bounded D* Lite (MD* Lite) aims to provide the same advantages of D* Lite, while applying memory usage constraints.

4.2 Detailed Review of D Lite and SMA**

4.2.1 D Lite overview*

D* Lite is algorithmically different, while being at least as efficient as Focused D* . D* Lite also determines the same paths as Focused D*, which is one of the most popular dynamic variants of A*. The D* Lite algorithm is for agents operating in environments that are incompletely known, while the agent is navigating. On every iteration, the agent plans the shortest path to the goal, with the assumption that unknown nodes are traversable. 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 are 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. D* Lite is a hybrid algorithm taking aspects from both Lifelong-Planning A* (LPA*) and Focused D*. D* Lite inherits all the properties of both LPA* and Focused D*.

LPA* is an incremental version of A* designed to work in dynamic environments. Unlike A*, LPA* can work with inadmissible heuristics. LPA* is usually used in fully known dynamic terrain. 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 to the effect of it needing to recompute its path. LPA* continually expands vertices until the goal node is consistent and the next node to expand doesn't 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* in the absence of a CLOSED list.

Focused D* is an incremental and heuristically driven derivative of A*. The algorithm is fully

optimal, always finding the shortest paths, while taking into effect changes within the environment. Focused D* is much faster than LPA*, as it modifies previous searches locally. Focused D* is used in real-world applications such as Nomad robots (Koenig et al., 2001).

D* Lite deals with the real-time traversal of dynamic environments. The algorithm makes no assumptions about whether nodes are increasing or decreasing. It is also irrelevant to the algorithm where these nodes are changing, whether close to the current location or otherwise. 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 untraversable, the edge cost is set to infinity. The result of such an approach is that D* Lite is able to plan with the existence of other agents also navigating the environment.

D* Lite switches the search direction of LPA*. The goal node becomes the start node and vice versa. All g-values of nodes now become estimates of the distance of the goal. 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 the 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.

4.2.2 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 s' predecessors. The value of rhs(s) is 0 if s = sstart, otherwise it is $\min_{s} \in P$ red(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.

4.2.3 D Lite Algorithm*

```
procedure CalcKey(s)
{01"'} return [min(g(s), rhs(s)) + h(sstart , s) + km ; min(g(s),
rhs(s))];

procedure Initialize()
{02"'} U = ∅;
{03"'} km = 0;
{04"'} for all s ∈ S rhs(s) = g(s) = ∞;
{05"'} rhs(sgoal ) = 0;
{06"'} U.Insert(sgoal , [h(sstart , sgoal ); 0]);
```

procedure UpdateVertex(u)

{07"'} if (g(u) not = rhs(u) AND u ∈ U) U.Update(u, CalcKey(u));

{08"'} else if (g(u) not = rhs(u) AND u not ∈ U) U.Insert(u, CalcKey(u));

{09"'} else if (g(u) = rhs(u) AND u ∈ U) U.Remove(u);

procedure ComputeShortestPath()

{10"'} while (U.TopKey()<CalcKey(sstart) OR rhs(sstart) > g(sstart))

{11"'} u = U.Top();

{12"'} kold = U.TopKey();

{13"'} knew = CalcKey(u));

{14"'} if(kold <knew)

{15"'} U.Update(u, knew);

{16"'} else if (g(u) > rhs(u))

{17"'} g(u) = rhs(u);

{18"'} U.Remove(u);

{19"'} for all s ∈ Pred(u)

{20"'} rhs(s) = min(rhs(s), c(s, u) + g(u));

{21"'} UpdateVertex(s);

{22"'} else

{23"'} gold = g(u);

{24"'} g(u) = ∞;

{25"'} for all s ∈ Pred(u) ∪ {u}

```
{26"'}            if (rhs(s) = c(s, u) + gold )
{27"'}                 if (s not = sgoal ) rhs(s) = min_{s' ∈Succ(s)} (c(s,
s') + g(s'));
{28"'}            UpdateVertex(s);
```

procedure Main()

```
{29"'} slast = sstart ;
{30"'} Initialize();
{31"'} ComputeShortestPath();
{32"'} while (sstart not = sgoal )
{33"'}     /* if (rhs(sstart ) = ∞) then there is no known path */
{34"'}     sstart = arg min_{s ∈Succ(sstart)} (c(sstart , s' ) + g(s'));
{35"'}     Move to sstart ;
{36"'}     Scan graph for changed edge costs;
{37"'}     if any edge costs changed
{38"'}         km = km + h(slast , sstart );
{39"'}         slast = sstart ;
{40"'}         for all directed edges (u, v) with changed edge costs
{41"'}            cold = c(u, v);
{42"'}            Update the edge cost c(u, v);
{43"'}            if (cold > c(u, v))
{44"'}                rhs(u) = min(rhs(u), c(u, v) + g(v));
{45"'}            else if (rhs(u) = cold + g(v))
```

{46"'} if (u not = sgoal) rhs(u) = $\min_{s \in Succ(u)}$ (c(u, s') + g(s'));

{47"'} UpdateVertex(u);

{48"'} ComputeShortestPath();

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. Nodes in the priority queue are then expanded until the start node is expanded. 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 affected items within the priority queue 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.

*4.3 SMA**

4.3.1 SMA overview*

The other foundation to the proposed algorithm, henceforth called Memory-bounded D* Lite, or simply MD* Lite, is the SMA* algorithm. SMA* (Simplified Memory-Bounded A*), is aimed at addressing the performance of A* in memory constrained settings. Like A*, SMA* is also designed for static environments that are completely known. The algorithm uses as much memory as it needs

and is made available to it. SMA* manages to avoid expanding previously expanded states as long as memory allows. It is complete when there is enough memory to store the shallowest solution path. It is also optimal if enough memory is available to store the shallowest optimal solution. If there is not enough memory to store the optimal solution, then the best solution that can be attained with the available memory is returned instead. This means it is not possible to determine whether the solution given by SMA* is in fact the optimal solution without other information.

SMA* assigns a depth to expanded nodes. The first expanded node (the starting location) is assigned a depth of 1. The successor vertices of the root are assigned a depth of 2, and their successors 3, and so on. If a node has a depth that is equal to the maximum amount of nodes allowed in the queue and it is not the goal, then it is ignored. This is due to the fact that it cannot be a member of path to the goal that can fit within memory. By necessity, the maximum allowed amount of items must be greater than or equal to 1, else the algorithm automatically fails.

SMA* makes use of the concept of forgotten nodes. The algorithm has to handle when a successor to a node needs to be expanded when no memory is available in queue. To do this, it determines the worst item in the queue. The worst item is the node n for which f(n), the estimated cost to the goal, is maximum. If that item is worse than the node that is to be added, then it is removed, and the new item is added. If the new item is worse, then it is considered the worst item that is to be forgotten. The element to be forgotten has a parent, which is the node that is its predecessor. The parent node maintains a value that is the minimum f-value of its forgotten children. This forgotten value is updated to take into account the worst child node, which may affect the value of the forgotten cost. The forgotten cost is a way for the algorithm to keep track of the best forgotten paths. If a forgotten path has a lower f-value than the next non-forgotten node to be expanded, then it is expanded instead.

SMA* performs well on problems with very accurate heuristics and also problems with many

highly-connected nodes. It does have problems in very complicated environments where not enough memory is allowed, wherein the algorithm has to often switch back and forth between solution paths. This in turn leads to many nodes being forgotten, and then having to be re-expanded. If such a node has many children, then they all have to be expanded and processed again. There is no accurate general-case method to determine what the minimum amount of memory is needed to obtain an optimal solution given just the start and goal location.

4.3.2 SMA algorithm:*

```
1. Put the start node S on the nodes list, called OPEN
2. If OPEN is empty, return failure
3. Set n as deepest node in OPEN where f(n) is minimum
4. If n is goal, return success (trace back from n to S)
5. For each successor n' of n
   a. if n' is not goal, and n' is at maximum depth, f(n') = infinity.
   b. else f(n') = max(f(n), g(n') + h(n'))
   c. If all of n' have been evaluated, set f(n) = lowest cost of
      child node, do same for ancestors if necessary, remove n from
      OPEN.
   d. If memory is full
      i. Remove shallowest node s where f(s) is maximum from OPEN
      ii.Remove s from n's successor list
      iii.Place n back in OPEN if necessary
      iv.Insert n' in OPEN
```

6. Goto 3

4.4 Memory-Bounded D* Lite

Memory-Bounded D* Lite (MD* Lite) will effectively merge the principles behind the SMA* and D* Lite algorithms to create an algorithm that operates in dynamic environments where agents have restricted memory. Below is the optimized version of D* Lite, with the changes required for MD* Lite highlighted.

```
procedure CalcKey(s)
{01"'} return [min(g(s), rhs(s)) + h(sstart , s) + km ; min(g(s),
rhs(s))];

procedure Initialize()
{02"'} U = ∅;
{03"'} km = 0;
{04"'} for all s ∈ S rhs(s) = g(s) = ∞;
{05"'} rhs(sgoal ) = 0;
{06"'} U.Insert(sgoal , [h(sstart , sgoal ); 0]);

procedure UpdateVertex(u) - Using D as the priority queue for things
not stored (forgotten nodes)
{07"'} if (g(u) not = rhs(u) AND u ∈ U ) U.Update(u, CalcKey(u));
```

1. if key of u has increased

1. call UpdateVertex(s) for all s in succ(u)

{08"'} else if (g(u) not = rhs(u) AND u not ∈ U)

1. If depth of u is less than max nodes allowed

1. if max allowed nodes reached

1. worst = item s in U where f(s) is max

2. if Key(s) > Key(u)

1. remove s from U

2. insert u in U

3. AddWorstToDelayed(s)

3. else

1. AddWorstToDelayed(u)

2. else U.Insert(u, CalcKey(u));

{09"'} else if (g(u) = rhs(u) AND u ∈ U) U.Remove(u);

Procedure AddWorstToDelayed - takes node, n

1. If no more room available in delayed queue

1. clean up delayed queue, removing parents whose children already updated

2. If still no room, return failure

3. else goto 2

2. if parent of n is in U, do nothing

3. else if parent of n is in delayed queue

1. update value of parent to be min(Key(parent) in delayed, Key(n))

4. else if there's room in delayed queue

1. add parent of n to delayed queue with Key(parent) = Key(n)

5. else

1. AddWorstToDelayed(U.pop())

2. add parent of n to delayed queue with Key(parent) = Key(n)

Procedure EnsureConsistent()

1. While D.TopKey() < U.TopKey() or memory available and items on D

1. for x in Succ(D.pop())

1. UpdateVertex(x)

2. return True

```
procedure ComputeShortestPath()
{10"'} while (EnsureConsistent() AND U.TopKey()<CalcKey(sstart ) OR
rhs(sstart ) > g(sstart ))
{11"'}      u = U.Top();
{12"'}      kold = U.TopKey();
{13"'}      knew = CalcKey(u));
{14"'}      if(kold < knew )
{15"'}          U.Update(u, knew );
{16"'}      else if (g(u) > rhs(u))
{17"'}          g(u) = rhs(u);
{18"'}          U.Remove(u);
```

```
{19"'}          for all s ∈ Pred(u)
{20"'}              rhs(s) = min(rhs(s), c(s, u) + g(u));
{21"'}              UpdateVertex(s);
{22"'}      else
{23"'}          gold = g(u);
{24"'}          g(u) = ∞;
{25"'}          for all s ∈ Pred(u) ∪ {u}
{26"'}              if (rhs(s) = c(s, u) + gold )
{27"'}                  if (s not = sgoal ) rhs(s) = min_{s ∈ Succ(s)} (c(s,
s') + g(s'))
{28"'}              UpdateVertex(s);

procedure Main()
{29"'} slast = sstart ;
{30"'} Initialize();
{31"'} ComputeShortestPath();
{32"'} while (sstart not = sgoal )
{33"'}      /* if (rhs(sstart ) = ∞) then there is no known path */
{34"'}      sstart = arg min_{s' ∈Succ(sstart)} (c(sstart , s' ) + g(s'));
{35"'}      Move to sstart ;
{36"'}      Scan graph for changed edge costs;
{37"'}      if any edge costs changed
{38"'}          km = km + h(slast , sstart );
{39"'}          slast = sstart ;
```

```
{40"'}          for all directed edges (u, v) with changed edge costs
{41"'}             cold = c(u, v);
{42"'}             Update the edge cost c(u, v);
{43"'}             if (cold > c(u, v))
{44"'}                 rhs(u) = min(rhs(u), c(u, v) + g(v));
{45"'}             else if (rhs(u) = cold + g(v))
{46"'}                   if (u not = sgoal ) rhs(u) = min_{s' ∈Succ(u)} (c(u,
s') + g(s')
{47"'}             UpdateVertex(u);
{48"'}          ComputeShortestPath();
```

Memory-Bounded D* Lite (MD* Lite) works by using 2 priority queues. Firstly, it is not that simple to determine what a memory unit is that needs to be constrained. D* Lite for instance maintains the rhs- and g-values of each node, yet these are not generally considered memory units. In MD* Lite, the second priority queue contains keys similar to the normal priority queue, and because those are considered memory units, these ones should necessarily be considered the same.

The amount of nodes in the regular priority queue, U, and the new priority queue count towards the total amount of memory units allowed. When nodes are to be added to the the priority queue, if it happens that there is no more available space, then this affects the second memory queue, D. If the node to be added, n, has a higher key than the worst node in U, then the key of the parent of n is updated within D. The worst node is defined is the node w for which key(w) > key(p) for all p in queue. The parent p of a node s is the member of Pred(s) for which g(p) + c(p, s) is minimum. Thus the parent of a node is its predecessor in the shortest path from the starting state to that node. The key of the parent in D is the minimum key of its successors. There is an exception to the definition of a

parent node. If it happens that there are already predecessors of s already in D, then the one in that list with the minimum cost (including traversal cost to s) is the parent. This is to prevent adding a new node to D if an existing node can serve the same purpose of accounting for the forgotten node.

If n has a lower key than the worst item in U, then that worst item is removed from U. The parent of that item is updated within D, and n is added to U. Because D* Lite allows for global sensors, it is possible that enough nodes change within the environment, it can easily overload the allowed memory. If these changed vertices are sufficiently disconnected, which is to say they do not share enough common ancestors, it is possible that not all parents can fit on D, and thus memory is exhausted. This is not a big problem if enough memory is allowed, and the amount of detected changed nodes is limited in some way, as is usually the case.

The consistency of U and D must always be maintained. This is to say that the following relationship must hold at all times, key(worst(U)) < key(best(D)). All nodes maintained in D must have a key that is worst than all nodes in U. Thus when nodes are expanded in U, it is guaranteed that there are no forgotten nodes (which are accounted for in D) which should have been expanded first. This is guaranteed in ComputeShortestPath, in which we check the status of the two priority queues. If it happens that this relationship is violated, or there are nodes in D and none in U, then the best node in D is processed. The processing of a node in D is simply removing the vertex from D, then updating each of its successors, adding to U if they are inconsistent and are good enough to be added. If a node is not good enough to be placed on U, it will return to being accounted for in D.

The processing of nodes in D cannot add already consistent nodes to U. The reason for this is that a successor s that did not need updating, will have g(s) = rhs(s) and thus will not have already been in U. As a result of this, UpdateVertex will simply ignore it. A vertex s' that was previously inconsistent, will be updated normally, and as g(s) not = rhs(s), will either be added to U, or its parent updated, again in this case, in D. It is sometimes necessary to repeatedly remove the worst nodes from

U to create enough memory space to add nodes to D. If memory is exhausted and thus a node needs to be added to D, but its parent is not already present, then another memory space needs to be created. Again, the worst node is dropped from U to create room to update its parent in D.

In the case that the parent of a node to be added to the delay queue is already in U, then we can safely ignore it. This is due to the fact that when ComputeShortestPath processes a node, the children of this node are updated. Thus a node that is forgotten, will be updated when its parent is processed. In the case that a change is detected in the parent before it is processed, the key of the forgotten child may change. We thus update any successors of said parent node, in which case they may end up with a different parent.

It is possible for there to be orphaned parents in D. If a parent node has changed, then it may no longer satisfy the relationship of being the predecessor on the lowest cost route to the child. Thus, if the child also changes, yet is still not good enough to be added to U, then a different node in D may be selected to be updated as its parent. These orphaned parents are accounted for when D and U are checked for consistency. Thus if an orphaned parent node is to be processed, then this child, which is no longer its child, will simply be accounted for under its new parent, and the old parent will be removed if it has no other forgotten children. This orphaning of parents does allow memory in D to be exhausted by information that is no longer valid, and we thus account for that by verifying the parent-child relationships of all nodes in D when we have exhausted all space. This is tedious, but as the situation should rarely happen if enough memory is allocated, or there is some restriction on the amount of changed nodes detected, it should be a rare occurrence. It therefore would have a rarely significant effect on the performance of the algorithm.

4.5 Theorems

MD* Lite does not meet any of the theorems related to D* Lite. Due to the fact that nodes may be added and removed repeatedly from the priority queue, it is impossible to guarantee how many times a node is to be expanded. Regardless of this, several theorems related to MD* Lite hold.

1. Theorem 1: MD* Lite produces the same path as D* Lite when all changes can be accounted for in memory.

   Proof: Because the top node of the priority queue is still expanded by ComputeShortestPath, the same order of node processing occurs. There are no changes that affect the values of the nodes expanded.

4.6 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 $\forall x,y$ neighbours of z, $C(x,z) = C(y,z)$, where C is the cost-function for edge-traversal. 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 Focused D*, Backward/Forward A*, and Breadth-first search. In this document we will compare MD* Lite to RLA*, Focused D* and D* Lite. Since SMA* is based on static environments, it is not possible to compare it here. We compare the maximum number of nodes stored on the priority queue(s) simultaneously. This tells us how much space the algorithm needs to operate. We ignore space

considerations for the program itself in terms of implementation, as this would be dependent upon such an implementation and cannot be easily generalized. We also compare the amount of nodes expanded. This is one way to measure the amount of work each algorithm is doing. Initially all nodes within the environment are given a random traversal cost between 1 and 2. The tests involve two levels of dynamism. Firstly, the cost to traverse to a node has a 5% chance of increasing or decreasing by up to 30% on each iteration, both numbers assigned at random. Secondly, each node can become blocked or unblocked on each iteration. The tests were ran with different blockage percentages, ranging in the natural numbers between 1% and 10% inclusive, but it was allowed that each node would only be blocked for at most 1 iteration.

The grid to be used will be 100x100. The starting and ending locations are randomly chosen, with a requirement that the minimum distance between is 50 nodes. The system the tests are ran on is irrelevant as they are not machine dependent. 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, thus enabling it to detect changes in a total of 29 nodes including its current location. It is not possible to accurately predict the amount of memory that MD* Lite will need before searching. For the purposes of testing, an A* search is ran on the initial state, with the result of that, plus 25 to handle changed nodes, is the allowed maximum for MD* Lite. This is just an arbitrary allowance, as MD* Lite will use less than the alloted amount of memory if possible. This is of course not guaranteed to be enough, in which case we simply ignore that grid, and perform another test. A total of 200 tests were ran, with 20 for each blockage percentage.

4.7 Test Results

4.7.1 Nodes Stored

The comparisons presented are average, and all relative to D*. Memory stored is based on the maximum amount of nodes maintained simultaneously. MD* Lite stored 27% less nodes than D*. D* Lite stored roughly the same as D*, storing 4% more nodes. RLA* was surprisingly able to find the optimal path 100% of the time, while only needing to store an average of 19% the amount of nodes maintained by D*. It should be noted that the heuristic used was not monotonic, due to the possible cost decreases. This does not seem to have been a factor in the performance of RLA*, but the algorithm would have been able to store even less nodes simultaneously, due to not needing a CLOSED list, if it was.

4.3.2 Nodes Expanded

RLA* was able to expand less total nodes than the other algorithms on average. It expanded 50% of the nodes expanded by D*, 33% of that expanded by MD* Lite, and 45% of that expanded by D* Lite. MD* Lite as expected, expanded more nodes than D* Lite and D*, expanding 50% more than D*. D* Lite ended up expanding 10% more nodes than D*.

All the algorithms remained consistent in performance across the different blockade ranges. No noticeable patterns appeared with increases in the amount of obstacles on the map.

Table 2: Performance over multiple blockage percentages relative to D*

According to block percentages

	NS – MD*L	NS – D*L	NS – RLA*	E – RLA*	E – MD* L	E – D* L
1	0.72	0.91	0.19	0.47	1.41	1.05
2	0.74	1.08	0.2	0.46	1.4	1.04
3	0.71	0.89	0.18	0.48	1.54	1.11
4	0.73	1.18	0.18	0.5	1.41	1.11
5	0.75	1.06	0.19	0.54	1.28	1.05
6	0.71	0.91	0.2	0.47	1.6	1.09
7	0.71	1.07	0.17	0.48	1.62	1.13
8	0.78	1.16	0.2	0.54	1.71	1.19
9	0.76	1.05	0.19	0.54	1.34	1.07
10	0.71	1.13	0.18	0.48	1.69	1.17
Average:	0.73	1.04	0.19	0.5	1.5	1.1
Std dev:	0.19	0.4	0.05	0.24	0.68	0.24

NS – Nodes stored* E – Nodes expanded*
* All relative to D*

Chapter 5: Conclusions and Future Work

5.1 Conclusions

The result regarding the memory comparison of RLA* and MD* Lite is misleading though, seeing that MD* Lite was not optimized for such an environment. As the environment was small, perhaps adding 25 nodes to the length of the A* path was too much. MD* Lite will use as much memory as it needs and has access to, resulting in the misleadingly high memory usage. This does point out an advantage that RLA* has over MD* Lite though. The only way to restrict how much memory MD* Lite uses is to set the amount of nodes it is allowed to store. The problem is that there is no a priori way to determine this accurately within dynamic environments. RLA* on the other hand, has no way to limit its memory usage, other than restricting its range of sight. This still falls short though, as the size of the Tabu list cannot be constrained.

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 SMA*, there are cases where using MD* Lite is intractable in terms of node expansions, as switching between too many different path solutions with insufficient memory can result in too many nodes being forgotten and needing to be expanded again. One of the negatives regarding RLA* is it is likely to search for a longer period in environments where a goal is unreachable, and will remain so. RLA* will search the all accessible pathways before finally realizing that the goal is unreachable.

While Focused D* and its variants are extremely popular in dynamic search environments, they are not aimed at being constrained by memory. MD* Lite produces the same path as D* Lite, while being able to work for agents with restricted memory. RLA* works very well in environments in which a strong heuristic can be provided. It also does well in which a goal is reachable, but doesn't

perform very well in environments in which it is not. Thus, smaller environments, in which it will not be too costly to fully explore, or environments with a high likelihood of a path existing, are appropriate for RLA*. MD* Lite and RLA* have been shown to be well suited for robots, or other agent implementations, where memory availability is a concern.

5.2 Future work

One of the variants of D* is Delayed D*. Delayed D* processes all overconsistent elements, propagating their changes. After the overconsistent elements are processed, underconsistent states are checked for and processed. This process is repeated until all states are consistent. Delayed D* is claimed to expand less than 50% of the nodes expanded by D* Lite. This also means that Delayed D* will expand less than 50% of the nodes expanded by MD* Lite. It is possible to incorporate the changes required by Delayed D* into the MD* Lite algorithm, resulting in the algorithm expanding significantly less nodes. The amount of nodes stored by the algorithm is likely to decrease, as items are placed on the queue in waves as opposed to all at once. Therefore it is my intention to create a delayed variant of MD* Lite.

In terms of RLA*, the next step is the implementation of the algorithm in practical situations. I intend to implement the algorithm on robotic navigation systems. The performance of the algorithm in these situations should provide further direction with regards to its capabilities.

REFERENCES

1. Baginski, B. (1996) The Z3 -Method for Fast Path Planning in Dynamic Environments.
   In Proceedings of the IASTED Conference on Applications of Control and Robotics
2. Barbehenn, M. and Hutchinson, S. (1995) Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest path trees. IEEE Transactions on Robotics and Automation, 11(2) pg. 198–214
3. Barto, A., Bradtke, S. and Singh, S. (1995) Learning to Act Using Real-Time Dynamic Programming.
   Artificial Intelligence vol. 72, pg. 81-138
4. Boddy, M. and Dean, T. (1989) Solving time-dependent planning problems.
   In Proceedings of the International Joint Conference on Artificial Intelligence, pg. 979–984
5. Bonet, B. and Geffner, H. (2001a) Heuristic search planner 2.0.
   Artificial Intelligence Magazine 22(3) pg. 77–80
6. Bonet, B. and Geffner, H. (2001b) Planning as heuristic search. Artificial Intelligence – Special Issue on Heuristic Search 129(1) pg. 5–33
7. Bulitko, V. (2003) Lookahead pathologies and meta-level control in real-time heuristic search.
   In Proceedings of the Euromicro Conference on Real-Time Systems, pg. 13–16
8. Carsten, J., Ferguson, D. and Stentz, A. (2006) 3D Field D*: Improved Path Planning and Replanning in Three Dimensions.
   Submitted to IROS
9. Cockburn, D., Kobti, Z., and Goodwin, S. (2006) Range-Limited A*

Future Play Conference, London, Ontario.

10. Dean, T. and Boddy, M. (1988) An analysis of time-dependent planning.

In Proceedings of the National Conference on Artificial Intelligence (AAAI)

11. Dechter, R. and Pearl, J. (1985) Generalized best-first search strategies and the optimality of A*.

Journal of ACM 32, 3, pg. 505-536

12. Dijkstra, E. (1959) A note on two problems in connexion with graphs.

Numer. Math. 1, pg. 269-271

13. Edelkamp, S. (1998) Updating shortest paths.

In Proceedings of the European Conference on Artificial Intelligence, pg. 655–659

14. Edelkamp, S. and Schrodl, S. (2000) Localizing A*.

In Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), pg. 885-890.

15. Ersson, T. and Hu, X. (2001) Path planning and navigation of mobile robots in unknown environments.

In Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS)

16. Ferguson, D. and Stentz, A. (2004a) Planning with imperfect information.

In proceedings of Intelligent Robots and Systems International Conference, Sendai, Japan

17. Ferguson, D. and Stentz, A. (2004b) Delayed D*: The Proofs

Carnegie Mellon Robotics Institute, Tech. Rep. CMU-RI-TR-04-51, September

18. Ferguson, D. and Stentz, A. (2005a) The Delayed D* Algorithm for Efficient Path Replanning.

In Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain

19. Ferguson, D. and Stentz, A. (2005a) Field D*: An Interpolation-based Path Planner and

Replanner.

In Proceedings of the International Symposium on Robotics Research (ISRR)

20. Ferguson, D. and Stentz, A. (2006) Multi-resolution Field D*.

In Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)

21. Fiorini, P. and Shiller, Z. (1996) Time optimal trajectory planning in dynamic environments.

In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)

22. Fujimura, K. (1991) Motion Planning in Dynamic Environments.

Springer-Verlag, Tokyo

23. Gelperin, D. (1977) On the optimality of A*.

Artificial Intelligence , 8(1) pg. 69–76

24. Glover, F. (1989) Tabu Search — Part I,

ORSA Journal on Computing 1: 3 pg. 190-206.

25. Hansen, E. and Zilberstein, S. (1996) Anytime heuristic search: Preliminary report.

In AAAI Fall Symposium on Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities, pg. 55-59 Cambridge, MA.

26. Hansen, E., Zilberstein, S. and Danilchenko, V. (1997) Anytime heuristic search: First results.

Tech. rep. 97-50, University of Massachusetts at Amherst, Department of Computer Science.

27. Hart, P., Nilsson, N. and Raphael, B. (1968) A formal basis for the heuristic determination of minimum cost paths.

IEEE Transactions on Systems Science and Cybernetics, 2:100–107

28. Hoffmann, J., and Nebel, B. (2001) The FF planning system: Fast plan generation through heuristic search.

Journal of Artificial Intelligence Research 14 pg. 253–302.

29. Kaelbling, L., Littman, M. and Cassandra, A. (1998) Planning and acting in partially observable

stochastic domains.

Artificial Intelligence

30. Koenig, S. (2001) Agent-centered search.

Artificial Intelligence Magazine, 22(4) pg. 109–131

31. Koenig, S., Tovey, C., and Halliburton, W. (2001) Greedy mapping of terrain

In Proceedings of the International Conference on Robotics and Automation pg. 3594–3599.

32. Koenig, S. and Likhachev, M. (2002a) D* Lite.

In Proceedings of the National Conference on Artificial Intelligence, pg. 476–483

33. Koenig, S. and Likhachev, M. (2002b) Incremental A*

Advances in Neural Information Processing Systems. MIT Press

34. Koenig, S. and Likhachev, M. (2002c) Improved fast replanning for robot navigation in unknown terrain.

In Proceedings of the IEEE International Conference on Robotics and Automation

35. Koenig, S., Tovey, C. and Smirnov, Y. (2003). Performance bounds for planning in unknown terrain.

Artificial Intelligence, 147 pg. 253–279

36. Koenig, S., Likhachev, M., and Furcy, D., (2004a) Lifelong planning A*.

Artificial Intelligence Journal, 155(1–2) pg. 93–146

37. Koenig, S., Likhachev, M., Liu, Y., and Furcy, D. (2004b) Incremental heuristic search in artificial intelligence.

Artificial Intelligence Magazine, 25(2) pg. 99–112

38. Koenig, S. and Likhachev, M. (2005) Fast replanning for navigation in unknown terrain.

Transaction on Robotics, 21(3):354–363

39. Koenig, S. (2004) A comparison of Fast Search Methods for Real-Time Situated Agents.

In Proceedings of the International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS), pg. 864-871.

40. Koenig, S. and Likhachev, M. (2006) A new principle for incremental heuristic search: Theoretical results.

In Proceedings of the International Conference on Autonomous Planning and Scheduling, pg. 402–405

41. Korf, R. (1990) Real-Time Heuristic Search

Artificial Intelligence, Vol. 42, No. 23, pg. 189-211

42. Korf. R. (1993) Linear-space best-first search.

Artificial Intelligence , 62 pg. 41–78

43. LaValle, S. (1998) Rapidly-exploring Random Trees: A new tool for path planning.

Technical report, Computer Science Dept., Iowa state University, October

44. Leven, P. and Hutchinson, S. (2002) Real-time motion planning in changing environments.

In Proceedings of the International Symposium on Robotics Research (ISRR)

45. Likhachev, M., Gordon, G. and Thrun, S. (2004) ARA*: Anytime A* with provable bounds on sub-optimality.

In Advances in Neural Information Processing Systems. MIT Press

46. Likhachev, D. et al. (2005) Anytime Dynamic A*: An Anytime, Replanning Algorithm.

In Proceedings of the International Conference on Automated Planning and Scheduling.

47. Pearl, J. (1984) Heuristics: Intelligent Search Strategies for Computer Problem Solving.

Addison-Wesley

48. Podsedkowski, L., Nowakowski, J., Idzikowski, M. and Vizvary, I. (2001) A new solution for path planning in partially known or unknown environments for nonholonomic mobile robots

Robotics and Autonomous Systems, vol. 34, pg. 145–152

49. Ramalingam, G. and Reps, T. (1996), An incremental algorithm for a generalization of the shortest-path problem.

Journal of Algorithms 21, pg. 267–305

50. Ratner, D. and Pohl, I. (1986) Joint and LPA*: Combination of approximation and search.

In Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86),

pg. 173-177.

51. Russell, S. (1992) Efficient memory-bounded search methods.

In Proceedings of the Tenth European Conference on Artificial Intelligence,

ECAI-92, Vienna, Austria

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

Prentice-Hall

53. Stentz, A. (1993) Optimal and efficient path planning for unknown and dynamic environments.

Carnegie Mellon Robotics Institute Technical Report CMU-RI-TR-93-20

54. Stentz, A. (1994) Optimal and efficient path planning for partially-known environments.

Proceedings of the IEEE International Conference on Robotics and Automation

55. Stentz, A. (1995) The Focused D* algorithm for real-time replanning

Proceedings of the International Joint Conference on Artificial Intelligence-95, Montreal, Quebec

56. Yoshizumi, T., Miura, T., & Ishida, T. (2000). A* with partial expansion for large branching factor problems.

In Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), pg. 923-929.

VITA AUCTORIS

Name: Denton Cockburn
Date Of Birth: August 10, 1982
Place Of Birth: St. Andrew, Jamaica
Degrees:
Bachelor Of Computer Science
Bachelor Of Science, Software Engineering