



Defence Research and
Development Canada

Recherche et développement
pour la défense Canada



Path Planning with *D*Lite*

*Implementation and Adaptation of the D*Lite Algorithm*

D. Mackay
DRDC Suffield

Technical Memorandum
DRDC Suffield TM 2005-242
December 2005

Canada

Path Planning with *D*Lite*

*Implementation and Adaptation of the D*Lite Algorithm*

D. Mackay
DRDC Suffield

Defence R&D Canada – Suffield

Technical Memorandum

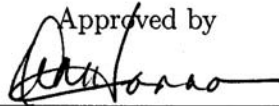
DRDC Suffield TM 2005-242

December 2005

Author

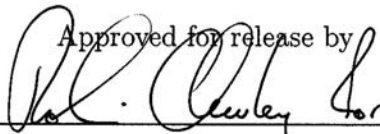

D. Mackay

Approved by



D. M. Hanna
Head/AISS

Approved for release by



Dr. P. A. D'Agostino
Head/Document Review Panel

Abstract

Incremental search methods reuse information from previous searches to find solutions to a series of similar tasks much faster than is possible by solving each search task from scratch. Heuristic search methods, such as A^* , use task-specific information in the form of approximations of goal distances to focus the search and typically solve search problems much faster than uninformed search methods. In LPA^* , incremental and heuristic search were combined to further reduce replanning times. D^*Lite is an application of a modified LPA^* to the goal-directed robot navigation task in unknown terrain. Conventional graph search methods, such as repeated applications of A^* , could be used when replanning paths after discovering previously unknown obstacles; however, resulting planning times could be prohibitively long. D^* uses a clever heuristic to speed up replanning by one or two orders of magnitude over repeated A^* searches by modifying previous search results locally. D^*Lite , building on LPA^* , implements the same navigation strategy as D^* , but is algorithmically different. It is an attractive alternative to D^* which is simpler to understand, implement, and debug. It involves only a single tie-breaking criterion when comparing priorities and does not need the nested *if* statements with complex conditions required in D^* . Furthermore, D^*Lite has been demonstrated to be at least as efficient as D^* . In this paper, an implementation of D^*Lite is described and its performance is compared with repeated A^* searches in solving goal-directed robot navigation tasks in unknown terrain.

Résumé

Les méthodes de recherche incrémentale réutilisent l'information obtenue des recherches antérieures pour trouver des solutions à une série de tâches similaires de manière plus rapide qu'il le serait de résoudre chaque tâche de recherche depuis le début. Les méthodes de recherche heuristiques telle que A^* , utilisent l'information spécifique à une tâche sous forme d'approximations de la distance des objectifs pour axer la recherche. Normalement, ces méthodes solutionnent les problèmes de recherche beaucoup plus rapidement que les méthodes de recherche non informée. Dans LPA^* (zone commune des modules rentrants), on a combiné les recherches incrémentales et heuristique pour mieux réduire les durées de re-planification. D^*Lite est une application d'une LPA^* modifiée, pour une tâche de navigation robotique guidée par un but, dans un terrain inconnu. Les méthodes de recherches en graphes traditionnelles, telles que les applications successives de A^* , pourraient être utilisées pour re-planifier les parcours après avoir découvert les obstacles antérieurement inconnus ; les durées prolongées de planification qui en résultent pourraient cependant être prohibitives. D^* utilise une heuristique ingénieuse qui accélère la re-planification de un à deux ordres de grandeur par rapport aux recherches successives A^* , ceci en modifiant localement les résultats des recherches antérieures. D^*Lite , construisant à partir de LPA^* , implémente la même stratégie de navigation que D^* , mais est différente au niveau de l'algorithme. Cette alternative D^* est intéressante parce que plus facile à comprendre, à implémenter et à mettre au point. Elle consiste en un seul critère décisif concernant la comparaison des priorités et elle n'exige pas les énoncés imbriqués *si* ni les conditions complexes requises en D^* . De plus, D^*Lite s'est révélé être au moins aussi efficient que D^* . Cet article comprend la description d'une implémentation de D^*Lite et on y compare son rendement avec les recherches successives A^* concernant les tâches de navigation robotique guidée par un but, dans un terrain inconnu.

This page intentionally left blank.

Executive summary

Path Planning with *D*Lite*

D. Mackay; DRDC Suffield TM 2005-242; Defence R&D Canada – Suffield;
December 2005.

Background

Autonomous vehicle navigation, the facility of moving from one position to another without operator supervision, is a key component of Unmanned Ground Vehicle (UGV) systems. Path planning, the Artificial Intelligence (AI) approach to autonomous vehicle navigation, enables the incorporation of *a priori* knowledge of the environment. It allows a UGV to act *intelligently*, obeying out-of-bounds restrictions, etc., and not simply react to perceived obstacles in its path. Drawing on the wealth of the AI literature in this area, the path planning algorithm *D*Lite* has been identified as a candidate for incorporation into Defence R&D Canada's UGV platforms. The *D*Lite* path planner combines aspects of *A** search, the classic AI heuristic search method, and incremental search to plan near-optimal paths in partially known environments. Incremental search methods reuse information from previous searches to find solutions to series of similar tasks much faster than is possible by solving each search task from scratch. *D*Lite* is reported to be at least two orders of magnitude faster than repeated *A** searches. *D*Lite* is algorithmically similar to the very successful *D** path planner but is much simpler to understand and thus to extend. *D*Lite* is at least as efficient as *D** (in terms of the number of nodes examined in the process of finding a path to a goal) and is quicker. In this paper, the development and implementation of *D*Lite* is described and the results of some performance tests conducted in simulation are presented.

Principal results

The *D*Lite* algorithm was tested in simulation, navigating through partially known, eight-connected grid worlds of varying sizes, fractions of traversable cells, and numbers of unknown obstacles. The performance of the *D*Lite* algorithm was compared to that of repeated *A** searches on the same grids. The performance metrics used were the number of node expansions and allocations required. The number of node expansions and allocations required by *D*Lite* were always less than or equal to that required by repeated *A** searches, becoming dramatically less with additional replanning episodes.

Significance of results

The *D*Lite* algorithm has been shown to be a much more efficient path planner than repeated *A** searches in solving the robot navigation problem in partially known terrain.

Future work

The performance tests need to be repeated, obtaining execution times, in addition to the node expansion and allocation metrics, to allow a direct comparison with the performance of other implementations of *D*Lite* and similar path planners. More importantly, the algorithm needs to be exercised outside of simulation in the real world.

Sommaire

Path Planning with *D*Lite*

D. Mackay; DRDC Suffield TM 2005-242; R & D pour la défense Canada – Suffield; décembre 2005.

Contexte

La navigation d'un véhicule autonome, la facilité avec laquelle ce dernier se déplace d'une position à une autre sans être surveillé par un opérateur est la composante clé des systèmes de Véhicules terrestres sans pilote (UGV). La planification du parcours qui est la méthode d'intelligence artificielle de navigation des véhicules autonomes, permet d'incorporer la connaissance *a priori* du milieu. Elle permet à un UGV d'agir *intelligemment*, en obéissant aux restrictions hors limites, etc., et ne réagit pas simplement aux obstacles qu'il a perçus sur son parcours. On a tiré parti de la richesse de la documentation en intelligence artificielle qui existe dans ce domaine pour identifier l'algorithme de la planification de parcours *D*Lite* comme étant un candidat pouvant être incorporé dans les plateformes UGV de R & D pour la défense Canada. Le planificateur de parcours *D*Lite* combine les aspects de la recherche *A**, la méthode de recherche heuristique classique d'IA avec la recherche incrémentale pour planifier un parcours quasi optimal dans des milieux partiellement connus. Les méthodes de recherche incrémentale réutilisent l'information obtenue des recherches antérieures pour trouver des solutions à des séries de tâches similaires de manière plus rapide qu'il le serait de résoudre chaque tâche de recherche depuis le début. On indique que *D*Lite* est plus rapide d'au moins deux ordres de grandeur que les recherches successives *A**. Au niveau de son algorithme, *D*Lite* est similaire au planificateur de parcours très efficace *D** mais il est plus facile à comprendre et par conséquent à étendre. *D*Lite* est au moins aussi efficace que *D** (en termes du nombre de nœuds examinés durant le processus de trouver un parcours vers un but) et il est aussi plus rapide. Cet article décrit la mise au point et l'implémentation de *D*Lite* et présente les résultats de certains tests de rendement en simulation.

Résultats principaux

L'algorithme *D*Lite* a été testé en simulation, navigant à travers huit maillages de mondes connexes partiellement connus, d'une variété de tailles, de fractions de cellules traversables et d'obstacles inconnus. On a comparé le rendement de l'algorithme *D*Lite* à celui des recherches successives *A** sur les mêmes maillages. Les paramètres de rendement utilisés étaient le nombre d'expansions et d'attributions de nœuds requis. Le nombre d'expansions et d'attributions de nœuds requis par *D*Lite* était toujours inférieur ou égal à celui requis pour les recherches successives *A**, diminuant énormément avec les épisodes additionnels de re-planification.

La portée des résultats

L'algorithme *D*Lite* s'est révélé être un planificateur de parcours beaucoup plus efficace que les recherches successives *A** pour résoudre le problème de la navigation robotique dans les terrains partiellement connus.

Les travaux futurs

Il faut répéter les tests de rendement pour obtenir des durées d'exécution en plus des paramètres d'expansion et d'attribution de nœuds, ceci pour permettre de comparer directement le rendement avec celui des autres implémentations de *D*Lite* et autres planificateurs de parcours similaires. Il est surtout important que l'algorithme soit expérimenté sans simulation dans le monde réel.

Table of contents

Abstract	i
Résumé	i
Executive summary	iii
Sommaire	v
Table of contents	vii
List of figures	viii
1 Introduction	1
2 Background	2
2.1 Graph Search	2
2.2 Uninformed Search	4
2.3 Heuristic Search	4
3 <i>D*Lite</i> Algorithm	9
3.1 Notation	9
3.2 Forward <i>LPA*</i>	11
3.2.1 Similarity to <i>A*</i>	11
3.2.2 Algorithm Description	13
3.3 Backward <i>LPA*</i>	14
3.4 <i>D*LiteBasic</i>	15
3.5 <i>D*Lite</i>	15
4 Implementation	19
5 Testing	21
6 Discussion and Conclusions	26
7 Future Work	27
References	28

List of figures

Figure 1:	Graph Search	3
Figure 2:	A^* Search	6
Figure 3:	Propagate	7
Figure 4:	Simple Maze Search	10
Figure 5:	Forward LPA^* Search	12
Figure 6:	Backward LPA^* Search	14
Figure 7:	D^*Lite Basic Search	16
Figure 8:	D^*Lite Search	17
Figure 9:	Effect of traversable fraction on the number of node expansions and allocations for a 100x100 grid world.	23
Figure 10:	Effect of grid size on the number of node expansions and allocations.	24
Figure 11:	Effect of the number of obstacles discovered during search in a 500x500 grid world on the number of node expansions and allocations.	25

1 Introduction

For the past several years, under the Autonomous Land Systems project, the guidance and control of Unmanned Ground Vehicles (UGVs) has been investigated at Defence R&D Canada – Suffield. Autonomous vehicle navigation, the facility of moving from one position to another without operator supervision, is a key component of a UGV system.

A UGV, tasked with navigating its way from its current location to a goal location, can simply follow the heading to the goal and reactively avoid obstacles as it encounters them, always returning to head directly towards the goal. This approach is simple and does not require building a map of the world, but without an hierarchy of behaviours to control the UGV, it is problematic. The UGV can become stuck in a cul de sac, turning one way and the other, and yet not able to back out. Also, without a representation of the world, the UGV is not able to handle *a priori* knowledge. This is an important shortcoming in a military context in which a UGV should reasonably be expected to obey out-of-bounds areas.

Alternatively, autonomous vehicle navigation can be thought of in terms of an Artificial Intelligence (AI) search task. Again, consider a UGV tasked with navigating its way from its current location to a goal location. However, in this case, the UGV has *a priori* knowledge about its environment and incorporates this into a map of its environment. The UGV begins by searching the map for the best (shortest, least costly, etc.) path and then attempts to follow this path to the goal. As the UGV tracks the path, it continually senses its environment and incrementally adds to its map. Should it discover that its path is blocked, it then plans a modified path to the goal. This approach to autonomous vehicle navigation, having AI search as its theoretical underpinnings, is provably complete, i.e., if a path exists to the goal, it will (eventually) be found. It also allows *a priori* information to be seamlessly incorporated into the map.

In this paper, path planning, the AI search approach to autonomous vehicle navigation is explored. A particularly effective search technique, called *D*Lite* [1], is examined in detail. The remainder of this document begins with some background into the general area of graph search as it relates to path planning. Next, the development of *D*Lite* from classic *A** search is briefly described. Then the implementation of *D*Lite* is presented, followed by some performance comparisons between *D*Lite* with repeated *A** searches solving the robot navigation through unknown terrain problem. And, finally, future work is proposed to address some unresolved questions regarding the implementation of the *D*Lite* algorithm.

2 Background

Before describing particular graph search techniques applicable to the autonomous vehicle navigation problem, we need to consider generic graph search and A^* search. In so doing, we will introduce some of the terminology found in the literature that will be useful in describing the development of the D^*Lite search algorithm. The following description of generic graph search and the A^* algorithm[2] is largely adapted from Nilsson[3], some specifics related to the A^* algorithm are taken from Rich and Knight[4].

2.1 Graph Search

A graph is a (not necessarily finite) set of nodes or vertices; pairs of nodes are connected by arcs or edges. If these arcs are directed from one member of a pair to the other, the graph is referred to as a directed graph. If a node s_i is connected to a node s_j by a directed arc from s_i to s_j , node s_j is said to be a successor or child of node s_i and node s_i is said to be a predecessor or parent of node s_j . A tree is a special case of a graph in which each node has at most one parent. A node in the tree without a parent is called a root node. A node in a tree having no successor is called a tip node or a leaf. The depth of a node in a tree is the depth of its parent plus one; the depth of a root node is zero.

A sequence of nodes $(s_i, \dots, s_{i+j}, \dots, s_k)$ with each s_{i+j} a successor of s_{i+j-1} for $j = 1, \dots, k$ is called a path of length k from node s_i to node s_k . If a path exists from node s_i to node s_k , then node s_k is said to be accessible from node s_i . Node s_k is also said to be a descendant of node s_i and node s_i is said to be an ancestor of node s_k .

Often it is convenient to assign positive costs to the arcs in a graph, representing the cost of transitioning from a node to one of its successors. The cost of an arc from node s_i to node s_j is denoted $c(s_i, s_j)$.

A graph may be specified either explicitly or implicitly. In an explicit specification, the nodes and arcs (with associated costs) are explicitly given in a table. The table might list every node in the graph, its successors, and the costs of the associated arcs. Obviously, an explicit specification is impractical for large graphs and impossible for those having an infinite set of nodes.

In typical applications, the control strategy generates (makes explicit) part of an implicitly specified graph. This implicit specification is given by the start node s , representing the initial database, and the rules that alter databases. A successor operator applied to a node gives all of the successors of the node (and the costs of the associated arcs) by applying the rules to that node. This process of applying the successor operator to a node is referred to as expanding the node. Expanding s , and the successors of s , *ad infinitum* makes explicit the graph specified implicitly by node s and the successor operator. A graph search control strategy, then, can be viewed as making explicit a sufficiently large portion of a graph so as to include a goal node. In the context of the autonomous vehicle navigation problem, a database is the state of the vehicle in the world representation and the rules essentially enumerate the possible moves the vehicle can make within the world representation.

A general graph search algorithm can be informally defined as shown in Figure 1. This procedure

```

Graph Search()
1 Create two empty lists Open and Closed.
2 Create a search graph G consisting solely of  $s_{start}$ ; put  $s_{start}$  on Open.
3 repeat
4   if (Open is empty) then
5     return error
6   Select the first node u on Open, remove it from Open and put it on Closed.
7   if ( $u == s_{goal}$ ) then
8     return u
9     (Following the backpointers from u to  $s_{start}$  in G will recover the path.)
10  Expand u, generating the set  $\{Succ(u)\}$ , the successors of u and install them as
11  successors of u in G.
12  foreach ( $s \in \{Succ(u)\}$ ) do
13    if ( $s \notin Open$  and  $s \notin Closed$ ) then
14      Establish a backpointer from s to u.
15      Add s to Open.
16    else
17      Decide whether or not to redirect its backpointer to u.
18    if ( $s \in Closed$ ) then
19      Decide for each successor of s, whether or not to redirect its
20      backpointer.
21  Reorder the Open list according to some arbitrary scheme or heuristic merit.
until a goal node is found

```

Figure 1: Graph Search

is general enough to encompass a wide variety of special graph search algorithms. The procedure generates an explicit graph, *G*, called the search graph, and a subset, *T*, of *G*, called the search tree. The search tree is defined by the back pointers set up at {1:10}.¹ Each node (except s_{start}) in *G* has a pointer directed to just one of its parents in *G*, which defines its unique parent in *T*. At {1:3} in the algorithm, the nodes on *Open* are those (tip) nodes of the search tree not yet selected for expansion. The nodes on *Closed* are either tip nodes selected for expansion which did not produce successors in the search graph or non-tip nodes of the search tree. The procedure orders the nodes on *Open* at {1:14} such that the best of these is selected for expansion. The ordering can be based on various arbitrary criteria or on some heuristic. Heuristic information can be regarded as a *rule of thumb* guiding the search in a fruitful direction. Whenever the node selected for expansion is a goal node, the algorithm terminates successfully. The successful path can be recovered by tracing backward via each node's backpointers from the goal node to the start. The algorithm terminates unsuccessfully when the search tree has no remaining tip nodes that have not yet been selected for expansion, i.e., when the *Open* list is empty.

If the search graph was a tree, then none of the successors generated in {1:7} could have been generated previously. Every node with the exception of the root node is the successor of only one node in a tree and is thus generated only once when its parent is expanded. Thus, in this special case, the members of the set of successors at {1:7&8} are not on either *Open* or *Closed*, so each member of the set of successors can be added to *Open* and installed in the search tree as a successor of the current node, *u*. The search graph is the search tree throughout the execution of the algorithm

¹This refers to source line 3 in Figure 1; this notation is used throughout the balance of the document.

and there is no need to change the parents of the nodes in T .

However, if the implicit graph being searched is not a tree, it is possible that some of the members of the set of successors have already been generated, i.e., they may already be members of *Open* or *Closed*. The problem of determining whether a newly generated node is identical to one generated previously can be computationally expensive. When the search process generates a node that it had generated before, it finds a path to it other than the one already recorded in the search tree. If this path is less costly than the one found so far, then the search tree is adjusted by changing the parentage of the regenerated node to its more recent parent.

If a node s on *Closed* has its parentage in T changed, a less costly path has been found to s . The less costly path may be part of less costly paths to the successors of s in the search graph. In this case, a change in the parentage in T of some of the successors of s in G may be in order. Because G is finite, the process of propagating the costs of the new paths downward to the successors of s in G is straightforward and finite.

2.2 Uninformed Search

If no heuristic information for the problem domain is used in ordering the nodes on *Open*, some arbitrary scheme must be used at $\{1:14\}$. The resulting search scheme is called uninformed. Two types of uninformed search are described here for purposes of comparison, depth-first and breadth-first search.

The first type of uninformed search orders nodes on *Open* in descending order of their depth in the search tree. The deepest nodes are put on the list first; nodes of equal depth are ordered arbitrarily. The search that results from such an ordering is called depth-first search because the deepest node in the search tree is always selected first for expansion. To prevent the search from following some fruitless path forever, a depth bound is typically provided.

The second type of uninformed search orders nodes on *Open* in increasing order of their depth in the search tree. The search that results is called breadth-first because expansion of nodes in the search tree proceeds along contours of equal depth. Breadth-first search is guaranteed to find an optimal path to a goal node, if such a path exists. If no path exists, the algorithm will exit with failure for finite graphs or will never terminate for infinite graphs.

2.3 Heuristic Search

Uninformed searches are exhaustive and, in principle, will find a solution to a path finding problem if one exists. However, they are often infeasible because of the size of the problem space; expanding too many nodes before a solution is found. A more efficient alternative to uninformed search uses task-dependent information to limit the number of nodes selected for expansion before a solution is found. In the context of the goal-directed robot navigation problem, the euclidean distance to the goal is an example of task-dependent information that can focus the search. Information of this sort is usually called heuristic information and searches using it are called heuristic search methods.

The particular heuristic search method described here is the A^* search. As in the generic graph search, two lists of nodes are employed:

- *Open* - nodes that have been generated and have had the heuristic applied to them but which have yet to be examined (i.e., had their successors generated). *Open* is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function.
- *Closed* - nodes that have already been examined. These nodes need to be kept because in searching a graph rather than a tree, each node generated must be checked to see that it has not already been generated.

Heuristic information can be used to order the nodes on *Open* so that the search expands along those sectors of the frontier thought to be most promising. In order to apply such an ordering, a method of estimating the merit or promise of each node generated is required. Call this function f to indicate that it is an approximation of a function f^* that gives the true merit of the node. For many applications, it is convenient to define this function as a sum of two components g^* and h . The function g^* is a measure of the cost of getting from the initial state to the current state. Note that g^* is not an estimate, it is the actual cost of traversing from the initial state along the optimal path to the current node. The function h is an estimate of the additional cost of traversing from the current node to the goal. This is the place where knowledge about the problem domain is exploited. The combined function f , then, represents an estimate of the cost of getting from the initial state to the goal along the path that generated the current node. If more than one path is generated for the current node then the algorithm records the best one. Note that because g^* and h are added, it is important that h must be a measure of the cost of getting from the current node to the goal rather than a measure of the goodness of the current node.

Each node must also store a backpointer to its parent or generating node in order to trace the optimal path from the goal back to the start node, s_{start} . The pseudocode for the A^* algorithm² is presented in Figure 2. When a better path has been found to a node $r \in Closed$, the improvement must be propagated to node r 's children. Node r maintains pointers to its children; each child in turn points to its children, until each branch either terminates with a node that is on *Open* or has no children. The purpose of the propagate function is to migrate the new cost information down through the *Closed* list. This is accomplished by performing a depth-first traversal of the underlying graph starting at node r , changing each node's g^* function value (and hence f function value), terminating each branch when a node with no children is reached or when a node is encountered for which an equivalent or better path has already been found. This is an easy condition to check. Each node, that has been explored, has a pointer back to its best known parent. As the propagation proceeds from a node, check to see if the child points back to the node being explored as its best parent. If so, continue the propagation. If not, then the child's g^* value already reflects a better path of which it is a part. The propagation may stop here; but it's possible that with the new value of the g^* function being propagated downward, the path that is being followed may become better than the path through the current best parent. Compare the two; if the path through the current parent is still

²Nilsson[3] refers to this algorithm as A , reserving the name A^* for the algorithm in which the heuristic function h is a lower bound on the perfect estimator h^* .

```

1  Open.insert( $s_{start}$  )
2  Closed  $\leftarrow$  empty
3   $s_{start}.g^* \leftarrow 0$ 
4   $s_{start}.f \leftarrow s_{start}.h$ 
5  repeat
6      if Open is empty then
7          return error
8       $u \leftarrow X \in \{Open | X = \arg \min_{Open}(X.f)\}$ 
9      Open.delete( $u$  )
10     Closed.insert( $u$  )
11     if  $u$  is the goal then
12         return  $u$ 
13         {Following the backpointers from  $u$  will recover the path.}
14     foreach  $s \in \{Succ(u)\}$  do
15          $s.parent \leftarrow u$ 
16          $s.g^* \leftarrow u.g^* + cost(u, s)$ 
17         if  $s \in Open$  then
18              $s_{old} \leftarrow Y \in \{Open | s == Y\}$ 
19              $u.child(s_{old})$  //Make  $s_{old}$  a child of  $u$ .
20             if  $s.g^* < s_{old}.g^*$  then
21                  $s_{old}.parent \leftarrow u$  //Make  $u$  the parent of  $s_{old}$ .
22                  $s_{old}.g^* \leftarrow s.g^*$ 
23                  $s_{old}.f \leftarrow s_{old}.g^* + s_{old}.h$ 
24                 delete( $s$ )
25         else
26             if  $s \in Closed$  then
27                 {If a better path to  $s_{old}$  is found, the improvement must be propagated
28                 to  $s_{old}$ 's children.}
29                  $s_{old} \leftarrow Y \in \{Closed | s == Y\}$ 
30                  $u.child(s_{old})$ 
31                 if  $s.g^* < s_{old}.g^*$  then
32                      $s_{old}.parent \leftarrow u$ 
33                      $s_{old}.g^* \leftarrow s.g^*$ 
34                      $s_{old}.f \leftarrow s_{old}.g^* + s_{old}.h$ 
35                     propagate( $s_{old}$  )
36                 delete( $s$ )
37             else
38                 { $s$  was not already on either Open or Closed.}
39                 Open.insert( $s$ )
40                  $u.child(s)$ 
41                  $s.f \leftarrow s.g^* + s.h$ 
42 until a goal node is found

```

Figure 2: A^* Search

```

propagate( $r$ )
foreach  $s \in \{Succ(r)\}$  do
  if  $s.parent == r$  then
    {The current child,  $s$ , is a member of the candidate path being updated (i.e.,
    its parent is node  $r$ ).}
     $s.g^* \leftarrow r.g^* + cost(r, s)$ 
     $s.f \leftarrow s.g^* + s.h$ 
    propagate( $s$ )
  else
    {The current child,  $s$ , is a member of another candidate path (i.e., its best
    parent is not node  $r$ ). Compare the cost to arrive at this child via node  $r$ 
    against the cost to arrive via its best parent node.}
    if  $r.g^* + cost(r, s) < s.g^*$  then
       $s.parent \leftarrow r$ 
       $s.g^* \leftarrow r.g^* + cost(r, s)$ 
       $s.f \leftarrow s.g^* + s.h$ 
      propagate( $s$ )

```

Figure 3: Propagate

better, stop the propagation. If the currently propagating path is better, reset the parent and continue propagation. The pseudocode for the propagate function is presented in Figure 3.

Several interesting observations can be made concerning the A^* algorithm. The first concerns the role of the g^* function. It lets one choose which node to expand next based not only on how good the node itself looks (as measured by the h function) but also on how good the path is thus far. By incorporating g^* into f , the node that appears closest to the goal will not always be chosen as the next node to expand. This is useful if the path itself is of concern. If it is not, and the manner in which the goal is reached is of no concern, then g^* can be defined to be always 0, thus always choosing the node which appears to be closest to the goal. If the path involving the fewest steps is desired, then the cost of going from a node to a successor can be set to a constant value, usually 1. If, on the other hand, the cheapest path is desired, then the g^* function should reflect the actual costs. Thus the A^* algorithm can be used to determine a minimal-cost path or simply any path as quickly as possible by varying the g^* function.

The second observation involves the h function, the estimator of h^* , the distance of a node from the goal. If h is a perfect estimator of h^* , then A^* will converge immediately to the goal with no search at all. The closer h is to the perfect estimator, the closer the convergence will be to the direct approach. If, on the other hand, the value of h is always 0, the search will be controlled by g^* . If the value of g^* is 0, the search strategy will be completely random. If the value of g^* is always 1, the search will be breadth-first. All nodes on one level will have lower g^* values, and thus lower f values, than will all nodes on the next level. If, on the other hand, h is neither perfect nor 0 and can be guaranteed to never overestimate h^* , then the A^* algorithm is guaranteed to find an optimal (as determined by g^*) path to a goal, if one exists. In practice, this is of very little use since the only way to guarantee that h never overestimates h^* is to set it to zero, resulting in a breadth-first search, admissible but inefficient. In the autonomous vehicle navigation problem, however, if the

shortest path is being sought, then an admissible heuristic is the Euclidean distance to the goal as it will never overestimate the actual distance that must be travelled.

A third observation that can be made concerning the A^* algorithm has to do with the relationship between trees and graphs. The algorithm was stated in its most general form as it applies to graphs. It can, of course, be simplified to apply to trees by not bothering to check whether a new node already exists on *Open* or *Closed*. This makes it faster to generate nodes but may result in the same search being conducted many times if nodes are often duplicated.

Under certain conditions, the A^* algorithm can be shown to be optimal in that it generates the fewest nodes in the process of finding a solution to a problem.

3 *D*Lite* Algorithm

The *D*Lite* algorithm [1] was devised to solve the autonomous vehicle navigation problem. It is an adaptation of Koenig's Lifelong Planning A^* (LPA^*) [5] which in turn is a derivation of A^* search incorporating incremental search. Incremental search methods reuse information from previous searches to find solutions to similar problems much faster than is possible by solving each search task from scratch.

By way of a concrete introduction to the problem, consider a goal-directed robot navigation task in unknown terrain in which a robot always observes which of its adjacent cells are traversable and then moves into one of them. From a start cell, the robot has to move to a goal cell. It always computes the shortest (or least cost in some sense) path from its current location to the goal under the assumption that cells with unexplored cells are traversable. It then follows this path until either it successfully reaches the goal or it observes an untraversable cell and is forced to recompute a shortest path from its current location to the goal. Figure 4, adapted from Koenig and Likhachev [1], shows the goal distances of all traversable cells and the shortest path from the robot's current cell to a goal cell. The top maze shows the initial path and the lower maze the situation after the robot has moved and discovered that its initial planned path is blocked. All of the cells in the world, with the exception of those adjacent to the start location, are unexplored before the robot has moved and are assumed to be traversable; these cells are painted white. Some cells, however, are known to impassable *a priori*, hence the robot plans an initial path around them; these cells are painted black.

In the top maze, the shortest path from the start location, S , to the goal location, G , is shown. This is determined by greedily decreasing the goal distance. In the lower maze, the cells whose goal distances have changed as a result of discovering that the planned path is blocked are shown in gray. Note that the majority of these changed cells are irrelevant to the replanned path. *D*Lite* is an efficient replanner because it identifies those cells that have changed and are relevant to the replanning task.

In what follows, the incremental development of the *D*Lite*, starting from the LPA^* , is presented.

3.1 Notation

Let S denote the finite set of vertices of the graph. Let $Succ(s) \subseteq S$ denote the set of successors of vertex $s \in S$ and similarly, $Pred(s) \subseteq S$ denote the predecessors of vertex $s \in S$. The cost of moving from vertex s_a to $s_b \in Succ(s_a)$ is denoted by $0 < c(s_a, s_b) \leq \infty$. The task of LPA^* is to maintain a shortest ³ path from a given start vertex $s_{start} \in S$ to a given goal vertex $s_{goal} \in S$, knowing both the topology of the graph and the current edge costs. The start distance of a vertex s is the length of the shortest path from the start vertex s_{start} to the vertex s and is denoted by $g^*(s)$.

³Although reference is made to the shortest path, it should be understood that LPA^* is equally applicable to finding a path minimizing some other cost metric, as determined by the edge costs, not just distance.

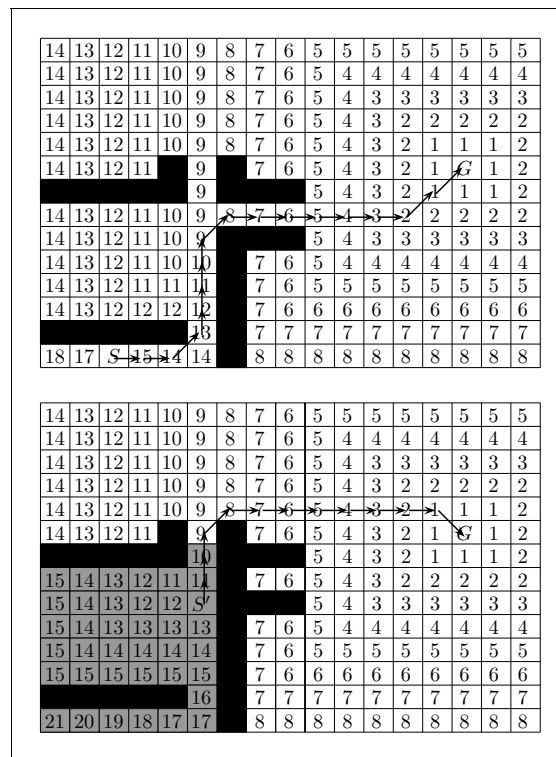


Figure 4: Simple Maze Search

3.2 Forward LPA^*

The LPA^* algorithm is shown in Figure 5. LPA^* maintains an estimate $g(s)$ of the start distance $g^*(s)$ of each vertex s , analogous to the g -values of an A^* search. LPA^* carries them forward from search to search. LPA^* also maintains a second kind of estimate of the start distances; the rhs -values are one step lookahead values based on the g -values and thus potentially better informed than the g -values. The rhs -values always satisfy the following relationship

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start} \\ \min_{s_a \in Pred(s)} (g(s_a) + c(s_a, s)) & \text{otherwise.} \end{cases} \quad (1)$$

A vertex is said to be locally consistent if and only if its g -value is equal to its rhs -value, otherwise it is locally inconsistent. If and only if all vertices are locally consistent, then the g -values of all vertices are equal to their start distances. However, LPA^* does not make all vertices locally consistent after some edge costs have changed. First, it does not recompute start distances that have been computed before and have not changed. And, second, it uses heuristic information, in the form of an estimate of the distance to the goal, to focus the search and updates only the g -values that are relevant to the computation of a shortest path. The heuristic estimate $h(s, s_a)$ of the distance between vertices s and s_a must satisfy

$$h(s, s_{goal}) \begin{cases} = 0 & \text{if } s = s_{goal} \\ \leq c(s, s_a) + h(s_a, s_{goal}) & \text{otherwise.} \end{cases} \quad (2)$$

for all vertices $s \in S$ and $s_a \in Succ(s)$.

The LPA^* algorithm is provably complete; it terminates and finds a shortest path if one exists. If $g(s_{goal}) = \infty$ after the search then no path exists between s_{start} and s_{goal} . Otherwise, one can trace a shortest path from s_{start} to any vertex s_u by, starting at vertex s_u , and always moving from the current vertex s to any predecessor s_{pred} of s that minimizes $g(s_{pred}) + c(s_{pred}, s)$ (ties can be broken arbitrarily) until s_{start} is reached. Thus, LPA^* doesn't explicitly maintain a search tree, instead it uses the g -values to encode it implicitly.

3.2.1 Similarity to A^*

Both A^* and LPA^* maintain a priority queue with heuristic information such that the most promising vertices are expanded first. LPA^* 's priority queue contains only the locally inconsistent vertices, those whose g -values LPA^* potentially needs to update to make them locally consistent. The priority of a vertex s in the queue is its key value $k(s)$, a vector of two components:

$$\begin{aligned} k(s) &= [k_1(s); k_2(s)] \\ k_1(s) &= \min(g(s), rhs(s) + h(s, s_{goal})) \\ k_2(s) &= \min(g(s), rhs(s)). \end{aligned} \quad (3)$$

The first component of the key, $k_1(s)$, corresponds to the f -value, $f(s) = g^*(s) + h(s, s_{goal})$, used by A^* and the second component, $k_2(s)$, corresponds to the g -values used by A^* . Keys are compared (and maintained in the priority queue) in lexicographic order;

$$k(s) \leq k(s_a) \text{ iff } \begin{cases} \text{either } k_1(s) < k_1(s_a) \\ \text{or } k_1(s) = k_1(s_a) \text{ and } k_2(s) \leq k_2(s_a). \end{cases} \quad (4)$$

```

1  CalculateKey(s)
   return [min(g(s), rhs(s)) + h(s, sgoal); min(g(s), rhs(s))]

```

```

2  Initialize()
   U ← 0
   for (s ∈ S) do rhs(s) ← g(s) ← ∞
   rhs(sstart) ← 0
   U.Insert(sstart, CalculateKey(sstart))

```

```

3  UpdateVertex(u)
   if (u ≠ sstart) then rhs(u) ←  $\min_{s \in \text{Pred}(u)} (\mathbf{g}(s) + c(s, u))$ 
   if (u ∈ U) then U.Remove(u)
   if (g(u) ≠ rhs(u)) then U.Insert(u, CalculateKey(u))

```

```

4  ComputeShortestPath()
   while (U.TopKey() < CalculateKey(sgoal) or rhs(sgoal) ≠ g(sgoal)) do
   | u ← U.Pop()
   | if (g(u) > rhs(u)) then
   | | g(u) ← rhs(u)
   | | for (s ∈ Succ(u)) do UpdateVertex(s)
   | else
   | | g(u) ← ∞
   | | for (s ∈ Succ(u) ∪ {u}) do UpdateVertex(s)

```

```

5  Main()
   Initialize()
   while (1) do
   | ComputeShortestPath()
   | Wait for changes in edge costs.
   | for (all directed edges (u, v) with changed costs) do
   | | Update the edge cost c(u, v).
   | | UpdateVertex(v)

```

Figure 5: Forward *LPA** Search

Thus, when LPA^* expands the vertex in the priority queue with the smallest key value first. This is similar to A^* which also expands the vertex in the priority queue with the smallest f -value if it breaks ties towards the smaller g -value. In addition, the keys of vertices expanded by LPA^* are nondecreasing over time just like the f -values of vertices expanded by A^* , if the heuristic information is consistent.

3.2.2 Algorithm Description

The function *Main()* first calls *Initialize()* at $\{5:17\}$ to start the search. *Initialize()* sets the g -values of all vertices to infinity and the rhs -values according to Equation 1 at $\{5:3\&4\}$. Initially, s_{start} is the only consistent vertex and is inserted into the empty priority queue at $\{5:5\}$. This initialization guarantees that the first call to *ComputeShortestPath()* performs an A^* search, i.e., it expands the same vertices as A^* would, in exactly the same order. It is important to note that in an actual implementation, *Initialize()* only needs to initialize those vertices that are encountered during the search and not all of the vertices in the search space. This is significant because search spaces are typically large and only a small fraction of the vertices contained therein may be visited during the search. The member function *TopKey()* returns the key value of the vertex at the top of the priority queue or $[\infty; \infty]$ if the queue is empty, and *Pop()* removes and returns the vertex with the minimum key from the priority queue.

At this point, LPA^* then waits for changes in the edge costs at $\{5:20\}$. If any edge costs have changed, *UpdateVertex()* is called to recompute the rhs -values and keys of the vertices potentially affected by the changed edge costs. In addition, if any of the vertices potentially affected have become locally consistent or inconsistent, their membership in the priority queue is adjusted.

The member functions *Remove()* at $\{5:7\}$ and *Insert()* at $\{5:8\}$, called by *UpdateVertex()*, remove a vertex from and insert a vertex onto, respectively, the priority queue.

Finally, *ComputeShortestPath()* at $\{5:19\}$ is called which repeatedly expands locally inconsistent vertices in the order of their priorities. A locally inconsistent vertex is called locally overconsistent if and only if $g(s) > rhs(s)$. A locally inconsistent vertex is called locally underconsistent if and only if $g(s) < rhs(s)$. When *ComputeShortestPath()* expands a locally overconsistent vertex at $\{5:12\&13\}$, it sets its g -value equal to its rhs -value, making it consistent. When expanding a locally underconsistent vertex at $\{5:15\&16\}$, *ComputeShortestPath()* sets its g -value to infinity. This makes the vertex either locally consistent or overconsistent. If the expanded vertex was locally overconsistent, then the change of its g -value can affect the local consistency of its successors. Similarly, if the expanded vertex was locally underconsistent, then the change of its g -value can affect the local consistency of itself and its successors. Thus, *ComputeShortestPath()* must invoke *UpdateVertex()* for all of the vertices potentially affected by the change in their g -values, modifying their rhs -values, checking their consistency, and adding them to or removing them from the priority queue as appropriate at $\{5:6-8\}$. *ComputeShortestPath()* expands vertices until s_{goal} is locally consistent or until the key of the next vertex to be expanded is no less than that of s_{goal} . This is similar to A^* that expands vertices until it expands s_{goal} at which point the g -value of s_{goal} equals its start distance and the f -value of the node to expand next is no less than the f -value of s_{goal} .

If $g(s_{goal}) = \infty$ after the search, then there is no path from s_{start} to s_{goal} . Otherwise one can trace

```

1  CalculateKey( $s$ )
   return [ $\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))$ ]

```

```

2  Initialize()
    $U \leftarrow 0$ 
3  for ( $s \in S$ ) do  $rhs(s) \leftarrow g(s) \leftarrow \infty$ 
4   $rhs(s_{goal}) \leftarrow 0$ 
5   $U.Insert(s_{goal}, CalculateKey(s_{goal}))$ 

```

```

6  UpdateVertex( $u$ )
   if ( $u \neq s_{goal}$ ) then  $rhs(u) \leftarrow \min_{s \in Succ(u)} (c(u, s) + g(s))$ 
7  if ( $u \in U$ ) then  $U.Remove(u)$ 
8  if ( $g(u) \neq rhs(u)$ ) then  $U.Insert(u, CalculateKey(u))$ 

```

```

9  ComputeShortestPath()
   while ( $U.TopKey < CalculateKey(s_{start})$  or  $rhs(s_{start}) \neq g(s_{start})$ ) do
10  |  $u \leftarrow U.Pop()$ 
11  | if ( $g(u) > rhs(u)$ ) then
12  | |  $g(u) \leftarrow rhs(u)$ 
13  | | for ( $s \in Pred(u)$ ) do UpdateVertex( $s$ )
14  | else
15  | |  $g(u) \leftarrow \infty$ 
16  | | for ( $s \in Pred(u) \cup \{u\}$ ) do UpdateVertex( $s$ )

```

```

17 Main()
   Initialize()
18 while (1) do
19 | ComputeShortestPath()
20 | Wait for changes in edge costs.
21 | for (all directed edges  $(u, v)$  with changed costs) do
22 | | Update the edge cost  $c(u, v)$ .
23 | | UpdateVertex( $u$ )

```

Figure 6: Backward LPA^* Search

back a shortest path from s_{start} to s_{goal} by always transitioning from the current vertex s , starting at s_{goal} , to any predecessor s_{pred} that minimizes $g(s_{pred}) + c(s_{pred}, s)$, breaking ties arbitrarily, until s_{start} is reached. This is similar to A^* if it does not use backpointers.

3.3 Backward LPA^*

It was pointed out in the simple maze search example, that during execution of goal-directed navigation as the robot moves towards the goal and discovers obstacles, many goal distances remain unchanged. By reversing the search direction in LPA^* , we can take advantage of this situation to improve the efficiency of the search. In the forward version of LPA^* , presented in Figure 5, the search starts with the start vertex and proceeds to the goal vertex and thus its g -values estimate the start distances. The backward version, presented in Figure 6, searches in the opposite direction, starting with the goal location and proceeding to the start location; its g -values are estimates of the goal distances. The backward version is derived from the original by reversing all edges of the graph

and exchanging the start and goal vertices. The heuristic information $h(s, s_a)$ now must satisfy

$$h(s_{start}, s) \begin{cases} = 0 & \text{if } s = s_{start} \\ \leq c(s_a, s) + h(s_{start}, s_a) & \text{otherwise.} \end{cases} \quad (5)$$

for all vertices $s \in S$ and $s_a \in Pred(s)$. Following execution of the backward LPA^* , if $g(s_{start}) = \infty$ then no path exists between s_{start} and s_{goal} . Otherwise, one can trace a shortest path from s_{start} to any vertex s_u by always moving from the current vertex s , starting at s_{start} , to any successor s_a of s that minimizes $c(s, s_a) + g(s_a)$, until s_u is reached.

3.4 *D*LiteBasic*

*D*LiteBasic* is built on the backward LPA^* algorithm. It repeatedly determines shortest paths between the current vertex and a goal vertex in the face of changing edge costs as the robot moves towards the goal vertex. As with the LPA^* variants, it makes no assumptions about the manner in which the edge costs change; they can increase or decrease, in close proximity to the robot or far from it, and they can change as a result of sensing a change in the world or simply by revising a priori estimates. As a result, *D*LiteBasic* can be used to solve the goal-directed navigation problem in unknown terrain. The terrain is modeled as a directed graph with edge costs representing the cost of edge traversal; unexplored edges have a small nonnegative cost and untraversable edges have a large positive cost. The goal-directed navigation strategy can be implemented by applying *D*LiteBasic* to this graph with s_{start} being the current vertex of the robot and s_{goal} the goal vertex. In implementing *D*LiteBasic*, shown in Figure 7, to solve the goal-directed navigation problem in unknown terrain, the main routine was extended to take into account the movement of the robot.

Since the robot moves and thus changes s_{start} , the heuristic estimate needs to satisfy

$$h(s_{start}, s) \begin{cases} = 0 & \text{if } s = s_{start} \\ \leq c(s_a, s) + h(s_{start}, s_a) & \text{otherwise.} \end{cases} \quad (6)$$

for all vertices $s \in S$ and $s_a \in Pred(s)$ and for all $s_{start} \in S$. This only changes the priorities of the vertices in the priority queue but does not affect which vertices are consistent and therefore the makeup of the priority queue itself is unchanged. When an edge cost change has occurred, *D*LiteBasic* must update the affected vertices and then perform a reordering of the priority queue. Given that the number of vertices on the queue could be very large, this could be a very expensive operation. Many of the vertices affected may also be irrelevant to the evolution of the planned path.

3.5 *D*Lite*

An improvement to *D*LiteBasic*, shown in Figure 8, borrows from D^* [6] a method to avoid repeatedly reordering the priority queue. After the robot has moved from vertex s to vertex s_a where it detects changes in edge costs, the first element of the priority keys can have decreased by at most $h(s, s_a)$, the heuristic estimate of the distance from vertex s to vertex s_a . The second component of the key does not depend on the heuristic estimate and is thus unchanged. If $h(s, s_a)$ was subtracted from the first element of the priorities of each of the vertices in the priority queue,

```

1  CalculateKey(s)
   return [min(g(s), rhs(s)) + h(sstart, s); min(g(s), rhs(s))]

```

```

2  Initialize()
   U ← ∅
   for (s ∈ S) do rhs(s) ← g(s) ← ∞
   rhs(sgoal) ← 0
   U.Insert(sgoal, CalculateKey(sgoal))

```

```

6  UpdateVertex(u)
   if (u ≠ sgoal) then rhs(u) ← mins ∈ Succ(u)(c(u, s) + g(s))
   if (u ∈ U) then U.Remove(u)
   if (g(u) ≠ rhs(u)) then U.Insert(u, CalculateKey(u))

```

```

9  ComputeShortestPath()
   while (U.TopKey < CalculateKey(sstart) or rhs(sstart) ≠ g(sstart)) do
10    u ← U.Pop()
11    if (g(u) > rhs(u)) then
12      g(u) ← rhs(u)
13      for (s ∈ Pred(u)) do UpdateVertex(s)
14    else
15      g(u) ← ∞
16      for (s ∈ Pred(u) ∪ {u}) do UpdateVertex(s)

```

```

17 Main()
   Initialize()
   ComputeShortestPath()
   while (sstart ≠ sgoal) do
20     // if (g(sstart) = ∞) then there is no known path
21     sstart ← arg mins ∈ Succ(sstart)(c(sstart, s) + g(s))
22     Move to sstart
23     Scan the graph for changed edge costs
24     if (any edge costs changed) then
25       for (all directed edges (u, v) with changed costs) do
26         Update the edge cost c(u, v)
27         UpdateVertex(u)
28       for (s ∈ U) do U.Update(s, CalculateKey(s))
29       ComputeShortestPath()

```

Figure 7: *D* Lite Basic Search*

```

1  CalculateKey(s)
   return [ $\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))$ ]

```

```

2  Initialize()
    $U \leftarrow \emptyset$ 
    $k_m \leftarrow 0$ 
   for ( $s \in S$ ) do  $rhs(s) \leftarrow g(s) \leftarrow \infty$ 
    $rhs(s_{goal}) \leftarrow 0$ 
    $U.Insert(s_{goal}, CalculateKey(s_{goal}))$ 

```

```

3  UpdateVertex(u)
   if ( $u \neq s_{goal}$ ) then  $rhs(u) \leftarrow \min_{s \in Succ(u)} (c(u, s) + g(s))$ 
   if ( $u \in U$ ) then  $U.Remove(u)$ 
   if ( $g(u) \neq rhs(u)$ ) then  $U.Insert(u, CalculateKey(u))$ 

```

```

4  ComputeShortestPath()
   while ( $U.TopKey < CalculateKey(s_{start})$  or  $rhs(s_{start}) \neq g(s_{start})$ ) do
7      $k_{old} \leftarrow U.TopKey()$ 
8      $u \leftarrow U.Pop()$ 
9     if ( $k_{old} < CalculateKey(u)$ ) then
10         $U.Insert(u, CalculateKey(u))$ 
11     else
12        if ( $g(u) > rhs(u)$ ) then
13             $g(u) \leftarrow rhs(u)$ 
14            for ( $s \in Pred(u)$ ) do UpdateVertex(s)
15        else
16             $g(u) \leftarrow \infty$ 
17            for ( $s \in Pred(u) \cup \{u\}$ ) do UpdateVertex(s)

```

```

21 Main()
    $s_{last} = s_{start}$ 
   Initialize()
   ComputeShortestPath()
   while ( $s_{start} \neq s_{goal}$ ) do
22     // if ( $g(s_{start}) = \infty$ ) then there is no known path
23      $s_{start} \leftarrow \arg \min_{s \in Succ(s_{last})} (c(s_{last}, s) + g(s))$ 
24     Move to  $s_{start}$ 
25     Scan the graph for changed edge costs
26     if (any edge costs changed) then
27          $k_m = k_m + h(s_{last}, s_{start})$ 
28          $s_{last} = s_{start}$ 
29         for (all directed edges ( $u, v$ ) with changed costs) do
30             Update the edge cost  $c(u, v)$ 
31             UpdateVertex(u)
32             ComputeShortestPath()

```

Figure 8: *D** Lite Search

one would obtain lower bounds on the priorities used in the *D*LiteBasic* algorithm. However, since $h(s, s_a)$ is the same value for each vertex in the queue, it's not necessary to actually perform the subtraction since it doesn't affect the order of vertices in the queue. Then, when new priorities are computed, their first components will be $h(s, s_a)$ too small relative to the priorities already in the priority queue. Thus, $h(s, s_a)$ must be added to the first component of the priority as each new vertex is added to the priority queue. When the robot moves again and detects further edge cost changes, the new $h(s, s_a)$ plus the previous value must be added. A new variable k_m is used to hold this running sum (at $\{8:30\}$) and whenever new priorities are computed k_m must be added to the first component (in *CalculateKey()* at $\{8:1\}$). In this way, the order of vertices in the priority queue is unaffected when the robot moves and the priority queue doesn't need to be reordered. The priorities, on the other hand, after k_m has been added to the first component, will always be lower bounds on the priorities used in *D*LiteBasic*.

4 Implementation

The *D* Lite* algorithm is coded in C++ as a template class, *DStarLiteSearch*. The template parameter, class *State*, is a user-defined type encapsulating the state variables of an element in the search space. Internal to the *DStarLiteSearch* class, an element of the search space is represented by an instance of the class *Node*. The *Node* class encapsulates:

- the g , rhs , and h values,
- the key values, computed from the previous three, used to order the priority queue,
- a pointer to the *State* class instance to which this *Node* corresponds,
- a backpointer to the predecessor of this instance, and
- a pointer to a drawable representation so that the developing path can be displayed.

In this way, there is a clear separation between the workings of the *D* Lite* algorithm and the representation of the planning space. The *DStarLiteSearch* class needs to know very little about its template parameter, class *State*, in order to update path costs and expand the search. The *State* class only needs to provide functions to:

- compute the heuristic estimate of the distance from the current node to the start node⁴,
- determine the neighbours of this *State*, and
- return the *a priori* and sensed costs of moving to an adjacent *State*.

Currently, the path planner maintains a global traversability map, updated as the robot moves towards the goal; the *State* class interfaces with this global map and gathers traversal costs from it.

Defence R&D Canada's *D* Lite* implementation departs from Koenig's in a number of respects. In this implementation, backpointers are employed, simplifying walking the planned path. Once the start node, the robot's current location, has been reached, backpointers identify the optimal path to the goal. This avoids the recomputation, inherent in Koenig's description because it lacks backpointers, associated with walking a planned path. In addition to the open list, *Open*, *D* Lite* maintains a closed list, *Closed*, as in *A** search. Both lists are implemented as `std::vector<Node*>` instances. *D* Lite* maintains *Open* as a priority queue (a minimum heap) using the Standard Template Library[7] functions *push_heap*, *pop_heap*, and a binary functor, *HeapCompare()*, for pairwise comparison of the key values. Examined nodes that are no longer candidates for expansion are removed to *Closed*. *D* Lite* dynamically allocates *Node* instances using a fixed block size allocator to minimize memory allocation expense. Every *Node** pointer is either a member of *Open* and a candidate for further expansion or on *Closed* and may be placed on *Open* if its costs change. This minimizes the number of allocations to find a path to the goal and simplifies deallocation.

⁴Recall that *D* Lite*'s search direction is reversed from that in *A** search.

Separating the D^* *Lite* algorithm from the planning representation simplifies experimentation with alternate planning spaces. An abstract planning space also eases the development of more sophisticated planners incorporating either additional dimensions or more complex node traversal costs. For example, incorporating vehicle dynamics to build a dynamics-aware path planner.

5 Testing

In an effort to gauge the performance of the *D* Lite* in solving the autonomous vehicle navigation problem in unknown terrain, a number of simulations were conducted in a rectangular 2D grid world. These included examinations of the effect of the size of the world and the fraction of free space in the world. The performance of the *D* Lite* algorithm is compared to that of repeated *A** searches.

Grid World

The grid world is eight-connected, the robot can move from its current cell forward or backward, left or right, or diagonally into an adjacent free cell. If a neighbour cell can not be reached, the cost of traversal to that cell is 10; otherwise, the cost of traversal is ≤ 10 , in fact for these tests, it is 1. It may be known *a priori* that a neighbour cell can not be reached, i.e., the neighbour cell in question is an obstacle, or it may be discovered to be an obstacle as the robot approaches it. To provide this capability, each cell of the grid world maintains two arrays of traversal costs. The first contains the *a priori* cost of traversal that would be incurred by the robot in moving from this cell to each of its neighbour cells. The second contains the costs of traversal that can be *sensed*. As the robot moves, it *senses* the costs of traversal to the cells within its sensing radius of its current location by querying those cells for their sensed costs of traversal. For all of the simulations described here, the sensing radius was set to its default value of 1.

To construct a grid world instance, an array of traversability values is generated automatically in a pseudorandom fashion while maintaining a specific fraction of traversable cells. Two arrays are actually used; the first, the plan array, contains *a priori* traversability information and the second, the sense array, will contain modified traversability information that the robot can sense as it moves in the world. Initially, the second is just a copy of the first. The sense array is edited manually to provide additional obstacles that can be discovered as the robot moves. The plan and sense arrays are then used to generate each cell's cost of traversal arrays, as described above. Each time the simulation is started, the grid world is constructed anew from the plan and sense arrays.

The start and goal locations for these tests were set to the lower left cell and the upper right cell of the grid world, respectively. For a traversable fraction of $\leq 50\%$, the likelihood of generating a world in which no path exists between the start and goal locations is significant. Grid worlds in which this was the case were edited manually to provide a clear path in proximity to the start and goal locations.

Discovered obstacles are provided in a two-step process. First, an initial path was planned between start and goal. Then a location on the path was chosen and manually marked as a discovered obstacle. When the planner is rerun, the robot will discover that its initial path is blocked and will be forced to replan.

Performance Metrics

Both A^* and D^*Lite maintain a priority queue with the most promising node for expansion always at the front of the queue. In this implementation, the underlying representation of the priority queue is a heap. Although this is an efficient representation, heap operations are nonetheless expensive. Both *push_heap* and *pop_heap* require reordering the heap, typically an $O(n \log_2 n)$ operation[8]. For A^* and similar searches that maintain a priority queue, one can reasonably expect that the number of node expansions (equivalent to the number of insertions and hence reorderings of the priority queue) would be a good metric for comparison of algorithm performance. Searching *Closed* to determine if a node has already been instantiated is also an expensive operation, $O(n)$. A direct measure of the length of time spent searching *Closed* isn't available.⁵ However, every node allocated is either on the *Open* or *Closed*, so the number of allocations gives some indication of the level of effort required to obtain a solution to a search problem and thus provides another useful metric for comparison of algorithm performance. In the tests described in the following sections, the number of node expansions and the total number of node allocations required by D^*Lite and repeated A^* searches are compared.

Effect of Traversable Fraction

To examine the effect on traversable fraction on path planning algorithm performance, 100x100 grid worlds were constructed with three traversable fractions: 50%, 60%, and 70%. Five replicates were generated for each traversable fraction. For each grid world, a single discovered obstacle was provided to force replanning using the method described in section 5. With repeated A^* search, an initial path was planned, then the planner was restarted from the point at which the obstacle was discovered. The number of node expansions and allocations is the sum of the values from both the initial and the replanning searches. The D^*Lite planner was simply run from the start location to the goal, allowed to discover the obstacle and forced to replan. The number of node expansions and allocations for the D^*Lite planner include the replanning episode, but arise from a single execution of the planner. In Figure 9, the number of node expansions and node allocations in a search in a 100x100 grid world with a single discovered obstacle are plotted for three traversable fractions: 50%, 60%, and 70%. Each datum is the average of the five replicates and the error bar represents ± 1 sample standard deviation.

As the traversable fraction increases, both the number of nodes expanded and allocated decreases. This simply reflects the greater ease with which a path can be planned when less of the terrain is impassable. With a 100x100 grid world and only a single discovered obstacle, there appears to be little difference in the performance of the D^*Lite and repeated A^* searches, measured with these performance metrics. The number of node expansions required is essentially the same for both D^*Lite and repeated A^* . The only significant difference is the higher number of node allocations required by repeated A^* in comparison to D^*Lite . Again, this is to be expected since D^*Lite and A^* plan exactly the same initial path. After encountering the discovered obstacle, D^*Lite retains all of the nodes allocated during the initial search, and, in particular, the nodes on the segment of the initial planned path lying beyond the discovered obstruction. During the replanning episode,

⁵ An oversight; it should have been recorded.

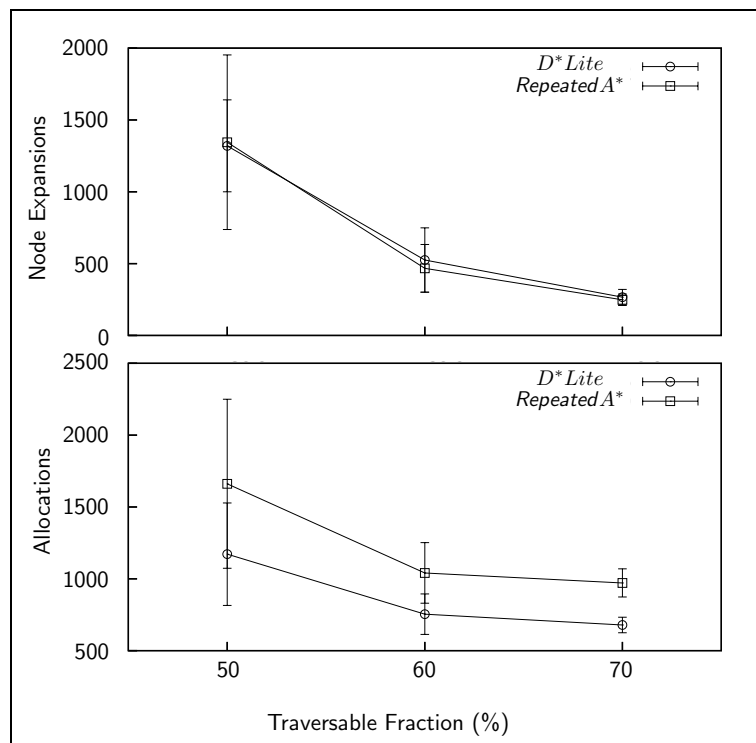


Figure 9: Effect of traversable fraction on the number of node expansions and allocations for a 100x100 grid world.

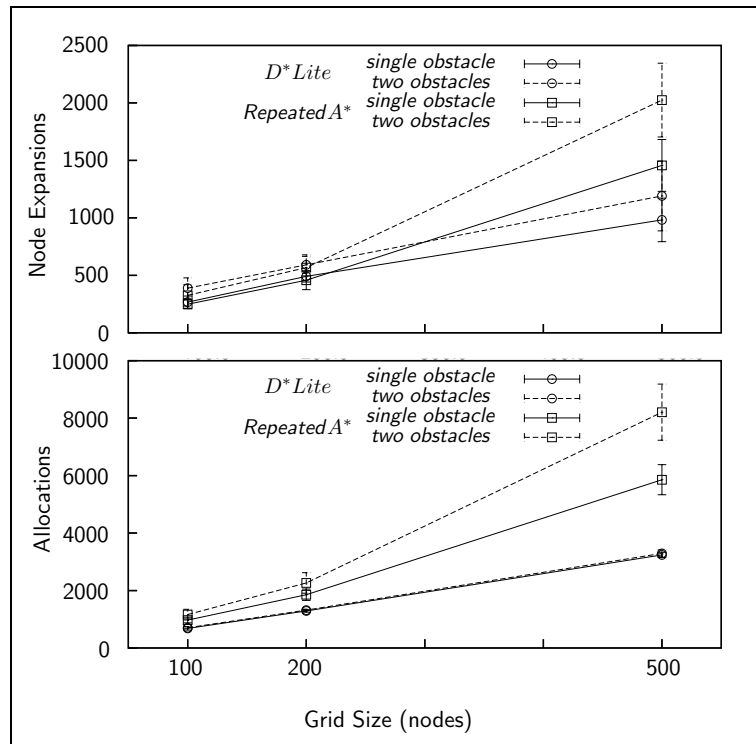


Figure 10: Effect of grid size on the number of node expansions and allocations.

D Lite* only need expand enough nodes to find a traversable connection between the robot's position at the point it discovered that the path was obstructed and a point beyond the obstruction on the minimal cost tree rooted at the goal node. Whereas the planner using a repeated *A** search during replanning must find a completely new path from the point at which the obstacle is detected. Thus with repeated *A** searches many nodes are reallocated during a replanning episode.

Effect of Grid World Size

The grid world used in the previous section to examine the effect of the traversable fraction was quite small, only 100x100 cells. To determine the effect of grid size on algorithm performance, path planning searches were conducted on three square grids: 100x100, 200x200, and 500x500. As in the traversability fraction simulations, five replicates were used. In this case, however, a path was planned in each replicate grid world, first with a single discovered obstacle and then repeated with an additional discovered obstacle. The results of these tests are shown in Figure 10. With an increasing grid size, both the number of node expansions and the number of allocations during the search are increased. This is the case for both *D* Lite* and repeated *A** searches. The numbers of node expansions and allocations required in repeated *A** searches in a 500x500 grid world are both clearly significantly higher than those required by *D* Lite*. With the addition of a second discovered obstacle, this effect is even more pronounced. However, with the addition of a second

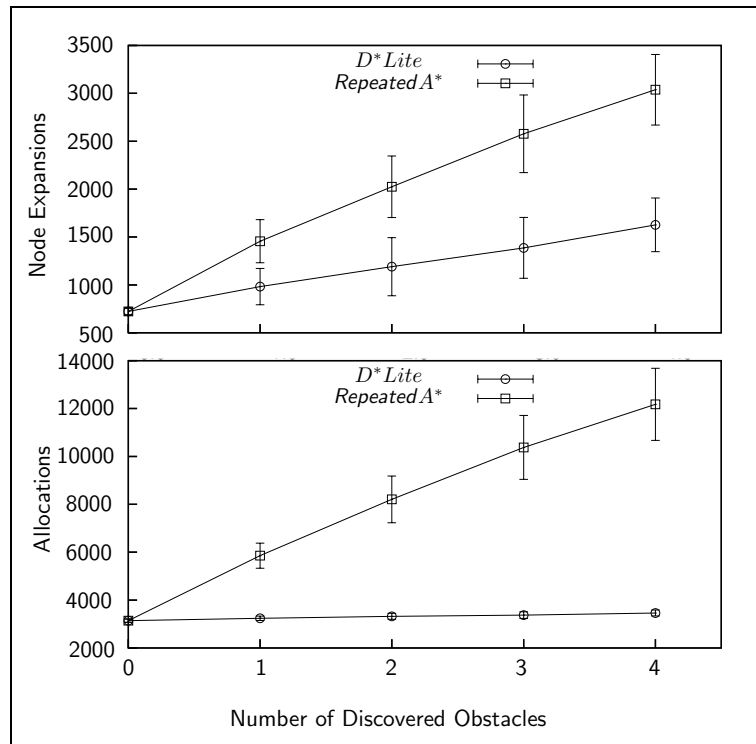


Figure 11: Effect of the number of obstacles discovered during search in a 500x500 grid world on the number of node expansions and allocations.

obstacle the numbers of node expansions and allocations required in the *D* Lite* search appeared not to change significantly.

Multiple Discovered Obstacles

To examine the effect of multiple discovered obstacles on the algorithm performance in greater detail, path planning searches were conducted on 500x500 grid worlds with up to four discovered obstacles. As with the other tests, five replicates of each grid world were used. The number of node expansions and allocations for zero through four discovered obstacles are shown in Figure 11.

The number of node expansions required increased with the number of discovered obstacles in a linear fashion for both repeated *A** and *D* Lite* searches, however, the rate of increase observed with repeated *A** searches was twice that observed with *D* Lite*. The number of node allocations increased with the number of discovered obstacles with repeated *A** searches as expected from the discussion in section 5. In contrast, the number of node allocations required with *D* Lite* appeared to be almost constant.

6 Discussion and Conclusions

D^* [6] has been reported to solve the robot navigation problem in unknown terrain one to two orders of magnitude faster than repeated A^* planning from scratch when the previously planned path is discovered to be blocked. Similar or slightly better performance has been reported for D^*Lite [1].

A direct comparison with the results reported in this paper is not possible because execution times were not recorded. Also, the nature of the grid worlds used here may invalidate a direct comparison. The pseudorandom placement of impassable cells, while maintaining a specified fraction of the total traversable, results in grid worlds that appear uniformly cluttered. There are neither large discrete obstacles nor large open spaces, except those generated by chance and this doesn't happen often. In particular, there aren't complex structures like cul de sacs that do occur in the real world. The uniform nature of the grid worlds and the fact that the states of the vast majority of the cells are known *a priori* (Only a few cells were changed to created obstacles that would be discovered as the robot follows the initial path to the goal.) resulted in an overly large number of nodes being examined during the initial planning episode. This is in stark contrast to most real world scenarios in which the granularity of *a priori* knowledge of the environment, consisting of large areas of the terrain assumed to be traversable interspersed with discrete obstacles, would be high. In this case, an initial planning episode would result in significantly fewer nodes having been examined since, with only a relatively few obstacles known *a priori*, the majority of those nodes examined would have been assumed to be traversable. Not until the robot actually traced the planned path would the previously unknown obstacles be discovered. During replanning, additional nodes would need to be examined, but in general, one could expect that more nodes would be examined during a traverse of a uniformly cluttered grid world than would be examined during a traverse of grid world of comparable size generated from real world terrain.

Regardless of the nature of the grid worlds used here, the same conclusion can be drawn, D^*Lite is much more efficient than repeated A^* searches in solving the robot navigation problem in partially known terrain.

A major question remains unanswered; how well will the D^*Lite algorithm scale to a real world problem? A 1 km square discretized with 0.5 m resolution, currently the resolution of the Raptor vehicle's traversability map, results in a planning grid with 4×10^6 nodes; much larger than the grids investigated here. Unfortunately, this implementation of D^*Lite has not yet been tested on a real platform negotiating real terrain. Due to a problem with the stability and accuracy of the heading estimation on the Raptor vehicle, consecutive local traversability maps obtained with the vehicle driving slowly forward in a straight line did not align at all well in the world frame. D^*Lite was thus not able to build a coherent internal world map using the local traversability map data and without a coherent internal world map, path planning is impossible. If it turns out that scaling is a problem, i.e., D^*Lite isn't able to provide planning information in a timely manner, alternate world representations or simply a coarser planning space resolution may provide some help. Framed quadrees[9], which effectively compress the planning space, have proven to be particularly useful representations for sparse environments.

7 Future Work

There remains much to be done to validate this implementation of *D*Lite* and to determine if it will ultimately prove to be useful.

First, the tests described in this paper should be repeated, recording execution times to allow a direct comparison with the performance of other implementations of *D*Lite* and similar path planners. Second, more realistic grid worlds with larger discrete obstacles should be employed. In addition, the effect of the proportion of the environment that is known *a priori* on the performance of *D*Lite* should be determined. And, most importantly, the algorithm needs to be exercised on a real vehicle negotiating real terrain.

If *D*Lite* scales well to the real world problem, the potential exists that it could be extended. There's no reason to restrict the *D*Lite* algorithm to the domain of a grid world. Like *A**, its domain is not the physical model or an abstraction of it, such as an eight connected grid world or occupancy grid, it's the set of transitions from one state in the world to another state in the world. And it's the cost of these transitions that are used in the planning heuristic. One could in principle plan using as the robot's adjacent states the set of states that are reachable (kinematically or dynamically). For instance, at time t , the robot's adjacent states could be the set of states whose predicted locations are the endpoints, at some time $t + \delta t$, of the path segments resulting from a discrete set of steering angles.

References

- [1] Koenig, S. and Likhachev, M. (2002). D* Lite, *Proceedings of the National Conference on Artificial Intelligence*, pp. 476–483.
- [2] Hart, P. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on SSC*, Vol. 4.
- [3] Nilsson, N. (1980). Principles of Artificial Intelligence, Morgan Kaufman Publishers Inc., Los Altos.
- [4] Rich, Elaine and Knight, Kevin (1991). Artificial Intelligence 2 ed, McGraw-Hill Inc., New York.
- [5] Koenig, S. and Likhachev, M. (2001). Lifelong Planning A*, (Technical Report GIT-COGSCI-2002/2), Georgia Institute of Technology.
- [6] Stentz, A. (1995). The Focussed D* Algorithm for Real-Time Planning, *Proceedings of the International Joint Conference on Artificial Intelligence*.
- [7] Stepanov, A. and Lee, M. (1994). The Standard Template Library, (Technical Report HPL-94-34(R1)), HP Labs.
- [8] Shiflet, A. and Martin, R. (1996). Data Structures in C++, West Publishing Co., Minneapolis / St. Paul.
- [9] Yahja, A., Stentz, A., Singh, S., and Brummitt, B. (1998). Framed-Quadtree Path Planning for Mobile Robots Operating In Sparse Environments, *Proceedings of IEEE Conference on Robotics and Automation*.

DOCUMENT CONTROL DATA		
(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)		
1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) Defence R&D Canada – Suffield Box 4000, Station Main, Medicine Hat, Alberta, Canada T1A 8K6	2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable). UNCLASSIFIED	
3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title). Path Planning with <i>D*Lite</i>		
4. AUTHORS (last name, first name, middle initial) Mackay, D.		
5. DATE OF PUBLICATION (month and year of publication of document) December 2005	6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc). 38	6b. NO. OF REFS (total cited in document) 9
7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered). Technical Memorandum		
8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include address).		
9a. PROJECT NO. (the applicable research and development project number under which the document was written. Specify whether project).	9b. GRANT OR CONTRACT NO. (if appropriate, the applicable number under which the document was written).	
10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique.) DRDC Suffield TM 2005-242	10b. OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor.)	
11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification) (X) Unlimited distribution () Defence departments and defence contractors; further distribution only as approved () Defence departments and Canadian defence contractors; further distribution only as approved () Government departments and agencies; further distribution only as approved () Defence departments; further distribution only as approved () Other (please specify):		
12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution beyond the audience specified in (11) is possible, a wider announcement audience may be selected). Unlimited		

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

Incremental search methods reuse information from previous searches to find solutions to a series of similar tasks much faster than is possible by solving each search task from scratch. Heuristic search methods, such as A^* , use task-specific information in the form of approximations of goal distances to focus the search and typically solve search problems much faster than uninformed search methods. In LPA^* , incremental and heuristic search were combined to further reduce replanning times. D^*Lite is an application of a modified LPA^* to the goal-directed robot navigation task in unknown terrain. Conventional graph search methods, such as repeated applications of A^* , could be used when replanning paths after discovering previously unknown obstacles; however, resulting planning times could be prohibitively long. D^* uses a clever heuristic to speed up replanning by one or two orders of magnitude over repeated A^* searches by modifying previous search results locally. D^*Lite , building on LPA^* , implements the same navigation strategy as D^* , but is algorithmically different. It is an attractive alternative to D^* which is simpler to understand, implement, and debug. It involves only a single tie-breaking criterion when comparing priorities and does not need the nested *if* statements with complex conditions required in D^* . Furthermore, D^*Lite has been demonstrated to be at least as efficient as D^* . In this paper, an implementation of D^*Lite is described and its performance is compared with repeated A^* searches in solving goal-directed robot navigation tasks in unknown terrain.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

path planning
robotics
navigation