

Implementing D Star Lite as a Pathfinding Solution to a Partially-known or Changing Grid-based Map

Seldi Kurnia Trihardja 13516042

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13516042@std.stei.itb.ac.id
xeldkt@gmail.com

Abstract—Pathfinding is a common and important problem in Computer Science. Pathfinding are used in many ways, from GPS to games. One of the most popular ways to solve it is using the A* algorithm, but in many cases we need to replan the route based on the current condition of the terrain. In this paper we discusses D* Lite, an incremental and heuristic graph search, in comparison with A* where the information about the environment is incomplete or the environment changes when the entity that uses the pathfinding, moves. As a representation we use a uniform cost grid based map as the environment

Keywords—pathfinding; A*; LPA*; D* Lite; Replanning; Grid-based Map

I. INTRODUCTION

Pathfinding in computer science is, as its name suggest, the plotting of a route between two points to create a path between them. Pathfinding as a field of research primarily consists of two problems, finding a path between two nodes in a graph or finding the “optimal” path between two nodes. These problems originally are solved using Breadth First Search or Depth First Search algorithms to exhaust many or even all possibilities of paths that are possible to find the best path through the graph. However, in reality, graphs can be massive in size making Depth First Search and Breadth First Search algorithms unfeasible to use as the time it would take to go through the possibilities would expand exponentially as the graph got bigger. And so, strategies and new ways are created to make an algorithm that can find the optimal path through a graph as fast as possible.

One of the most popular algorithms that are used as a solution to pathfinding with acceptable results are Dijkstra’s algorithm or its closely related variation, A* algorithm. Dijkstra’s algorithm find the shortest path by selectively choosing the path that, in total, cost the lowest in relation to all other paths that are open, and then expanding that path and compare all the paths again to expand the path with minimum cost and so on until it reaches the destination. Since Dijkstra’s expand the minimum path, when it finds the destination, the path will be the path with the least cost.

A* expanded on Dijkstra’s algorithm to include a heuristic that acts as some kind of guide so the algorithm doesn’t need to examine paths that are “roundabout” or paths that move away from the goal. Normally, the heuristic is the estimated distance between the node and the goal. With the cost of the edge of the node and the heuristic, A* modify the behaviour of Dijkstra’s algorithm to be more efficient by examining fewer nodes on average.



Figure 1 . A* pathfinding through a grid based map (red means examined)

Since then, many kinds of algorithm are created to cater to specific needs of many fields. While A* is still widely used in many software, there exist many algorithm that take advantage of varying techniques like dynamic programming, preprocessing the graph, exploiting the nature of grids, and so on. For our purposes, we are going to look at an algorithm that saves information from the first search to use later much like dynamic programming but also uses heuristics that are created by Koenig and Likachev called D* Lite[2] that uses LPA*(Lifelong Planning A*) that is also created by Koenig and Likachev with David Furcy[3] to mimic D* algorithm which uses dynamic programming to pathfind through partially known environment.

II. ALGORITHM BACKGROUND

A. D*

D* or Dynamic A* as coined by its founder Anthony Stentz[3], is an incremental heuristic search method that are used to address pathfinding where the environment is unknown or partially-known. At the time, most pathfinding algorithm assumes complete and accurate model of its environment, while in reality many situations occur where an environment changes without warning or the information present are incomplete or even non-existent. Automated machines that operate within unknown environment or partially-unknown environment, such as an exploratory robot in Mars or even an entity in games that have changing environment, need the capability of fast and efficient replanning method so it can move more intelligently especially in difficult terrain or time critical movement.

An outline of D* algorithm is as follows. Like A*, D* algorithm keep a list of nodes to be evaluated known as the "OPEN list". Nodes can be marked as several different states such as NEW, OPEN, CLOSED, RAISE, and LOWER. NEW means that the node has never been placed on the open list. OPEN means that it is currently on the open list. CLOSED means that it is no longer on the open list. RAISE means that the node cost is higher than the last time it was on the open list. While LOWER means that the node cost is lower than the last time it was on the open list. The algorithm works by expanding nodes from the goal until it reaches the start node. every node has a backpointer so that every node knows the way to the target. Every node also knows the exact cost to the target. This way we know all paths that lead to the target by

using the backpointers. When an obstacle presents itself where it obstructs the original path, the nodes that are affected are put in the open list again with the state RAISE and reevaluated on whether its neighbor can reduce its cost or not. If it can, the backpointer is updated and it passes the LOWER state to its neighbor. If it cant, it passes the RAISE state to its neighbor. The passing of states then continue forming a "wave" of RAISE and LOWER

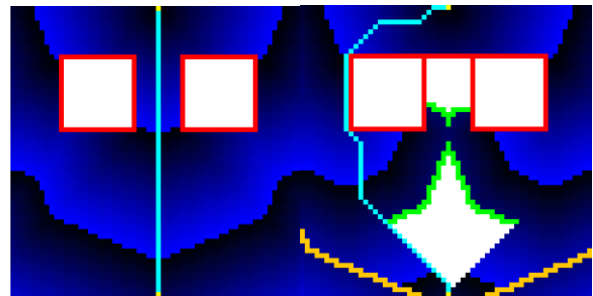


Figure 2 . Visualization of D* algorithm (Red is obstacle, blue are nodes with the brightness indicating cost, cyan is the path, green are lower states, and yellow are raise states)

B. LPA*

Most of the search methods available mainly focus on one-shot type of planning, where you plan the path once at the beginning. However, in reality, often times algorithms need to adapt their planning continuously as the model of the world changes. Without a technique to specifically deal with replanning, an algorithm needs to be ran again from scratch to change its planning. This approach might not be preferable as if there are many changes then the performance of the algorithm decreases significantly. Sometimes, we also need to run a pathfinding algorithm repeatedly to a series of similar world, or if the path needs to be continuously refined or learned.

LPA*(Lifelong Planning A*), also known as Incremental A*, is an incremental heuristic search that combines DynamicSWSF-FP and A*. LPA*, unlike the original A*, can adapt to changes to the graph without planning all the way from scratch. LPA* does this by using two estimates of distances $g(n)$ and $rhs(n)$, where g is the previously calculated cost and rhs is the minimum value of the g of its "parents", or formally known as predecessors, plus the cost of moving from that predecessor to the node. LPA* also uses heuristics in determining which nodes to update or expand using a system of two dimensional keys as the determining factor for its priority queue.

LPA* expands its nodes with the following rule. If the rhs -value of a node equals its g -value, the node is "locally consistent" and is removed from the queue. If the rhs -value of a node is less than its g -value, the node is "locally overconsistent" and the g -value is changed to match the rhs -value, making the node locally consistent. The node is then removed from the queue. If the rhs -value of a node is greater than its g -value, the node is a "locally underconsistent" node and the g -value is set to infinity (which makes the node either

locally overconsistent or locally consistent). If the node is then locally consistent, it is removed from the queue, else its key is updated by recalculating its rhs-value and inserting it back to the queue. Since changing the g values can cause the rhs values of other nodes to change, the nodes that are changed are also considered and updated. When an aspect of the graph changes, like the edges cost, LPA* recognizes all the nodes that are affected by the change and updates its values to the proper value and nodes that are locally consistent are removed from the queue and those that are inconsistent are added back to the queue to be updated. The algorithm finishes if the goal is locally consistent or the node to examine next according to the queue has a key bigger than the goal, where it means that the goal is unreachable.

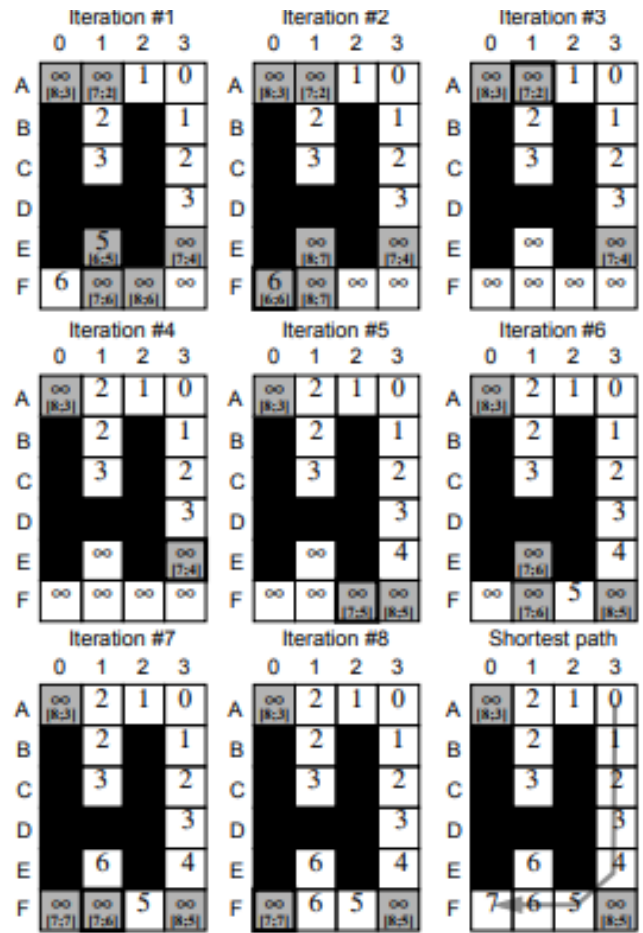


Fig. 4. An Example - Second Search

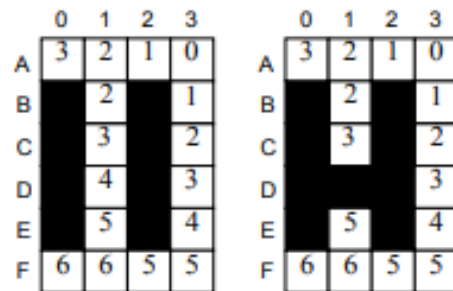


Figure 3. Visualization of LPA* Algorithm

The following are LPA* pseudocode:

```

procedure CalculateKey(s)
{01} return [min(g(s), rhs(s)) + h(s); min(g(s),
rhs(s))];
procedure Initialize()
{02} U = ∅;
{03} for all s ∈ S rhs(s) = g(s) = ∞;
{04} rhs(sstart) = 0;

```

```

{05} U.Insert(sstart, [h(sstart); 0]);
procedure UpdateVertex(u)
{06} if (u != sstart) rhs(u) =
mins0 ∈ pred(u)(g(s') + c(s', u));
{07} if (u ∈ U) U.Remove(u);
{08} if (g(u) != rhs(u)) U.Insert(u,
CalculateKey(u));
procedure ComputeShortestPath()
{09} while (U.TopKey() < CalculateKey(sgoal)
OR rhs(sgoal) != g(sgoal))
{10} u = U.Pop();
{11} if (g(u) > rhs(u))
{12} g(u) = rhs(u);
{13} for all s ∈ succ(u) UpdateVertex(s);
{14} else
{15} g(u) = ∞;
{16} for all s ∈ succ(u) U {u}
UpdateVertex(s);
procedure Main()
{17} Initialize();
{18} forever
{19} ComputeShortestPath();
{20} Wait for changes in edge costs;
{21} for all directed edges (u, v) with
changed edge costs
{22} Update the edge cost c(u, v);
{23} UpdateVertex(v);

```

III. D* LITE

At the time, even though D* have the attractive capability as a real-time pathfinding algorithm to artificial intelligence that can handle the difficult problem of a changing world model, D* is infamous as a complex algorithm and as such didn't get much popularity outside of specialized fields. Intending to combine the capability of D* to replan paths in an unknown or partially-known model of the world and the replanning robustness of LPA* that they created, Koenig and Likhachev proposed D* Lite in 2005.

As a result D* Lite uses LPA* to mimic the searching behavior of D* algorithm, but algorithmically much simpler and at least as fast as D* algorithm, making it easier to analyse understand and extend the algorithm itself opening many more possibilities.

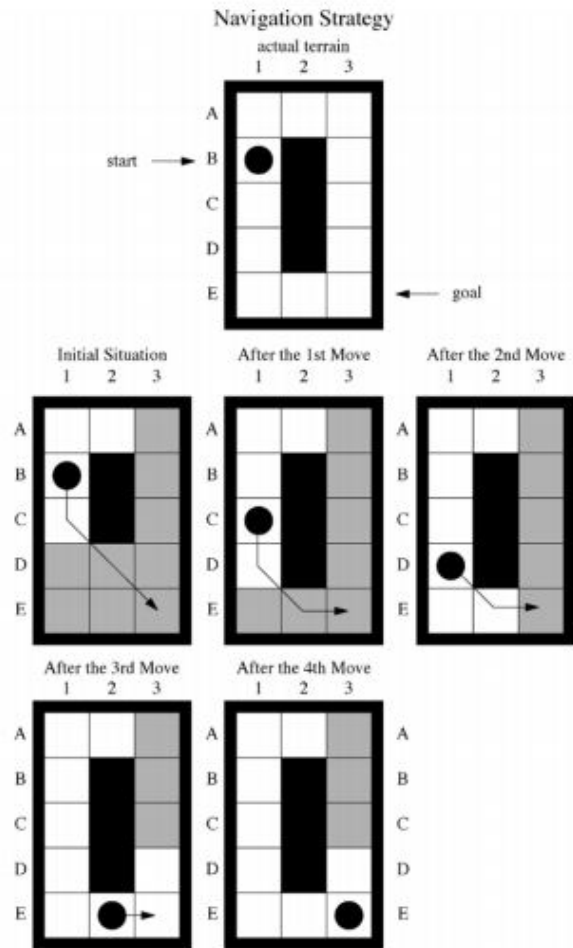


Figure 4. Visualization of navigation strategy

D* Lite mimics D* using LPA*, in short, by making the start position the current position of the robot. However, since LPA* uses the estimate of the distance to the start position as an integral part of its algorithm while the start position always moves around, we need to reverse how LPA* works so that it pathfinds from the goal to the start position. In effect, the $g(n)$ in LPA* now becomes the distance to the goal and since the goal position doesn't change it is able to work. After computing the shortest path using it, we can find the shortest path by following the minimum cost and $g(n)$. While running the reversed LPA*(D* Lite), if it notices any changes in the costs of the graph the algorithm need to reorder its priority queue every time it notices any changes.

Because the difference of D* and LPA*, other than it's algorithm, is slight (since D* can be considered as the map to change everytime the entity that uses the algorithm discovers new information that affect the path and LPA* deal with changing paths, it's just that D* doesn't care about the original start position but instead the current position of the entity) LPA* can be derived from and to a D* variant in D* lite with little difficulty. Further details for the D* Lite algorithm can be read on the reference papers written in the reference

IV. IMPLEMENTATION

For the implementation, we use C as the programming language. The model that are used are a grid based map that are visualized onto the command prompt using the stdout functions.

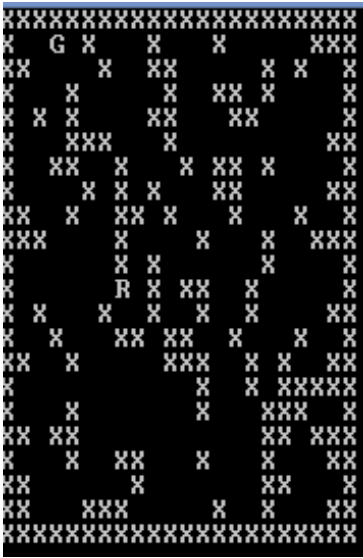


Figure 5 . An example of complete map, in this case we assume R only knows the spaces that are adjacent and towards the goal

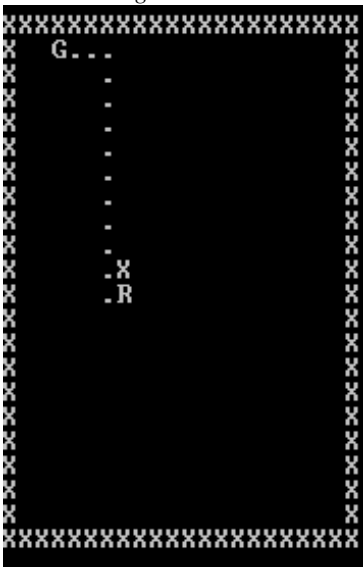


Figure 6 . The first iteration, unknown spaces are considered without obstacles until set otherwise

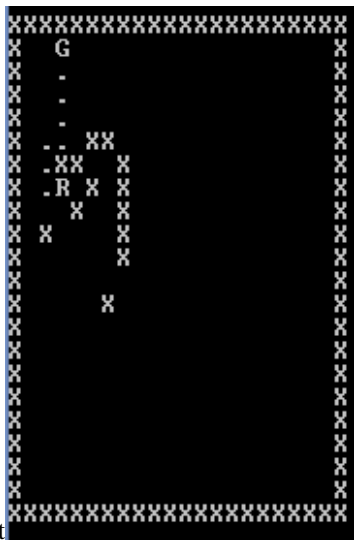
From Figure 6, we can see that D* Lite works by first swiftly finding a shortest path between R(Start position) and G(Goal position) where if there is unknown information we consider it passable for the time being. In implementing graph-search algorithm, we usually need to break ties. Ties are situations when we uses search algorithm in a graph, and we find more than one shortest path to get to a specific place. Since computers are deterministic, we need to decide which

path to go through(expand) first. To do that, we do what is known as Tie Breaking where we put a specific mechanism of choosing, in this case the implementation favors path that have larger g-values. For example in figure 6 there are two path with equal cost, either going left first then going up or going up first and then going left as in figure 6.



Figure 7 . The Robot encounters an obstacle in its original path and replan the path starting from the current position

As told before in the explanation of the algorithms, when the algorithm detects a change in the environment, in this case the new obstacle that R detected means that particular space g value has become infinity(impassable), D* Lite replan the path from the goal to the current position, now with the added information that a once empty space is now impassable. On the second picture of Fig.7. we see that the original direction is actually a dead end and the algorithm successfully changes its own route to reach the goal using the priority queue. R will continue moving towards the goal according to the previously planned path until it find a complication where it is unable to follow the original path or it detect a change in the cost of its path and need replanning to make sure it is the shortest path where it will replan. D* Lite will continue to run until R current position are locally



consistent
 Figure 8 . R found a clear path

On Fig.8. we can see that R planned a path with no obstacles when using the complete map to manually determine it.. To get the shortest path, all we need to do is traceback R steps through the graph and get the optimal path. However, even though it is not shown here, it is very possible to not have a valid path to the goal, in fact when the implementation generated a random maze and random position for the goal and start position it surprisingly happens somewhat often. Another observation that we can see from figure 8 are overall D* Lite does not need to examine that many nodes even though the grid is a 20x20 grid

In General, D* Lite can solve pathfinding problems where the pathfinding knows the current position and the goal (goal-directed navigation). The advantages of D* Lite compared to other goal-directed navigation are obviously the capability to replan and adapt according to the information it has about the model of the world, whether it is partially-known, changes, or even unknown. Because of this D* Lite can also be used to solve mazes since mazes are generally assumed as having an unknown interior. The above example(Figure 5-8) can also be considered a maze. The only factor limiting it is if it knows where the exit is, if the exit is unknown then we cannot use the aforementioned D* Lite. The following is an example of a grid-based maze where the start position is considered the entrance and the goal position is the exit

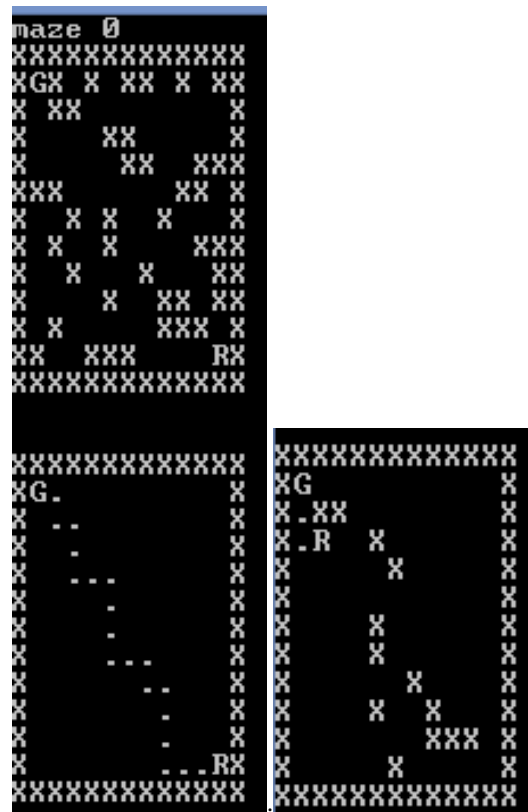


Figure 8 . A maze with D* Lite, initial plan, and final plan

V. CONCLUSION

D* Lite is a very good alternative to traditional graph-search methods like A* , especially when the world model often changes or it is an unknown environment. However it still has its own limitation with specific scenarios where papers have shown that D* Lite struggle to solve in terms of performance such as if the goal is also always changing. In general D* Lite can be implemented to any kind of pathfinding problem, from mazes, automated robot navigation, to games pathfinding.

ACKNOWLEDGMENT

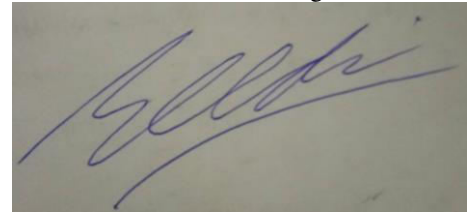
The author firstly would like to thank God for helping me give the power and will to finish this writing. I would like to also thank my parents for all the support they give so i can finish this paper

REFERENCES

- [1] Sven Koenig and Maxim Likhachev, " D* Lite, " Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI), pp. 476-483, 2002.
- [2] Sven Koenig and Maxim Likhachev, " Fast Replanning for Navigation in Unknown Terrain, " IEEE Transactions on Robotics (TRO), 21(3), pp. 354-363, 2005.

- [3] Stentz, Anthony (1994), "Optimal and Efficient Path Planning for Partially-Known Environments", Proceedings of the International Conference on Robotics and Automation: 3310–3317
- [4] Sven Koenig, Maxim Likhachev, and David Furcy, " Lifelong Planning A*", " Artificial Intelligence Journal (AIJ), 155(1-2), pp. 93-146, 2004.
- [5] Moving Target D* Lite, X. Sun, W. Yeoh and S. Koenig, Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010), van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. XXX-XXX
- [6] <http://idm-lab.org/> (diakses 11 Mei 2018)
- [7] <https://cstheory.stackexchange.com/questions/11855/how-do-the-state-of-the-art-pathfinding-algorithms-for-changing-graphs-d-d-1> (diakses 13 Mei 2018)

Bandung, 14Mei 2018



Seldi Kurnia Trihardja 13516042

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.