D* Lite Shortest Path Algorithm

Ian Gioffre
CPSC 450
Spring 2021

Summary

I looked at the D* Lite shortest path algorithm which finds the shortest path from a source to a destination given static and dynamic obstacles. My implementation approach was to develop a robust set of examples to run my algorithm on. All examples passed and worked as expected. In other words, I was able to find the shortest path from source to destination with moving and/or stationary obstacles.

1. ALGORITHM SELECTED

The problem I worked on was the dynamic shortest path problem. This is finding the shortest path from two nodes given obstacles that can move. The algorithm I selected to solve this problem was the D* Lite algorithm. This algorithm finds the shortest path from a source node to a destination node while scanning for and avoiding obstacles that are stationary or moving.

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

Figure 3: D* Lite.

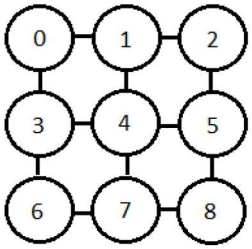
[1]

Since this is a dynamic algorithm, the complexity is not intuitive. In my implementation, since I do not scan for edge changes and instead assume all edges have changed, the time complexity is $O(n^2)$ where m is the number of nodes in the terrain graph. This is because for each step, all edges are updated and then the shortest path is found by stepping through adjacent vertices and using those nodes to find the next step in the shortest path.

2. BASE IMPLEMENTATION

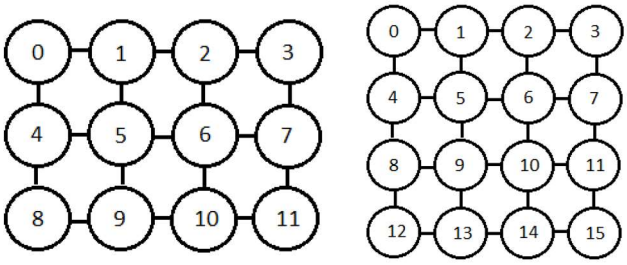
My implementation deviated from the pseudocode provided in one way. Instead of scanning for edges that changed, I assumed all edges changed and updated every node in the terrain. As for data structures I used. U is a list<pair<int, pair<int,int>>>. Thus, CalculateKey returns a pair<int,int>. All nodes are stored as integers in which the value of the node is the node number assigned to it at grid creation. Starting in the top left and going through each row, the nodes are numbered 0 to n – 1.

Figure 1. 3x3 Terrain Graph



I used many helper methods, so I first tested those. The helper methods tested were methods that created the grid, added and removed obstacles, and calculated the heuristic of the minimum number of edges between two vertices. I then tested the algorithm by adding several tests for 3x3 grids, 4x3grids, and 4x4 grids. I tested first without obstacles, then with static obstacles, then with variable obstacles.

Figure 2-3. 4x3 and 4x4 Terrain Graphs



3. EXTENDED IMPLEMENTATION

For my implementation plan, I chose to develop a robust set of examples. These examples overlapped with my testing from above. I added examples for no obstacles, static obstacles, and variable obstacles. I also added examples where there was no path from the source to the destination.

For all tests SO is static obstacle, VO is variable obstacle

Figure 4. 3x3 Test from 0 to 2 with SO(1)

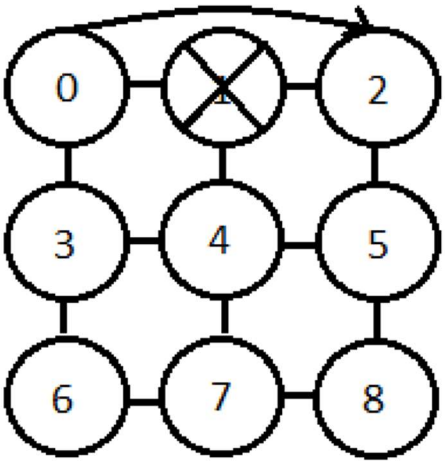


Figure 5. 4x3 Test from 1 to 9 with SO(5)

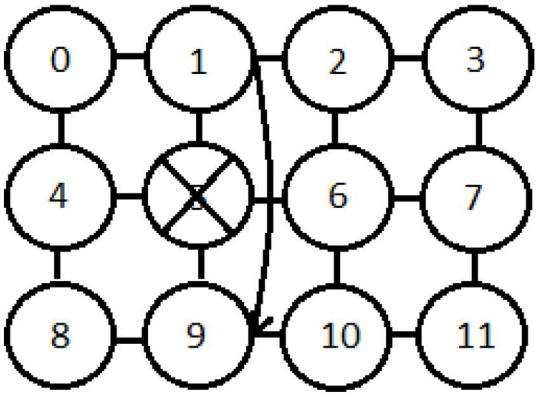


Figure 6. 4x3 Test from 0 to 3 with VO(2)

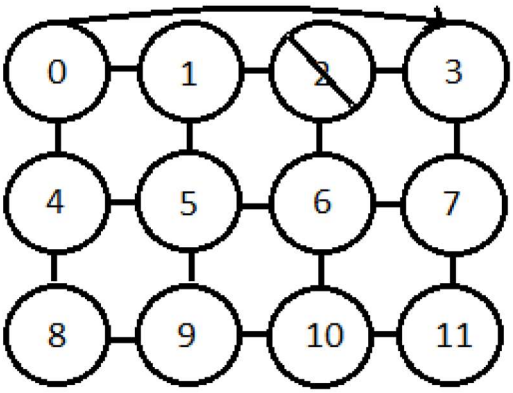


Figure 7. 3x3 Test from 0 to 2 with SO(5), VO(2)

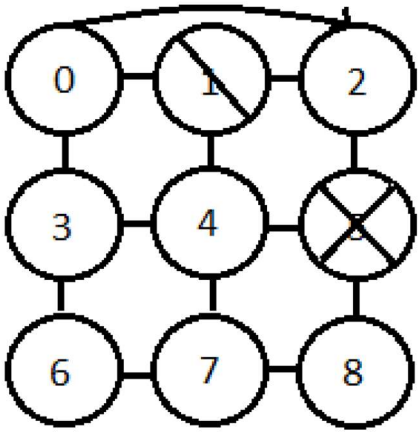
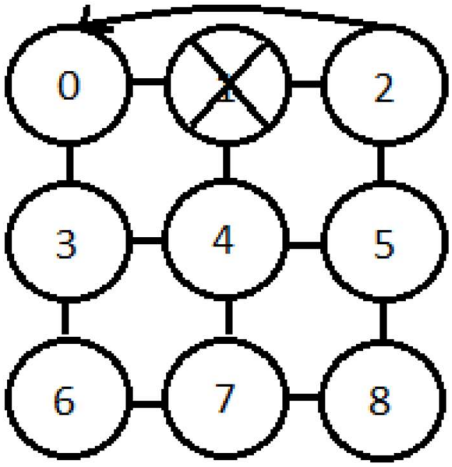


Figure 8. 3x3 Test from 2 to 0 with SO(1)



4. BUILD AND RUN INSTRUCTIONS

To run the tests and examples:

[1] cmake CmakeLists.txt

[2] make

[3] ./test_d_star

5. REFLECTION

My work was interesting because it is used in a lot of real-word robotics. D* Lite is used for robotic path finding where there is a dynamic environment. This algorithm is similar to Dijkstra's in that it is finding a shortest path between two nodes. The biggest challenge I faced was figuring out which data structures to use for C++. After meeting with Dr. Bowers, we were able to analyze the pseudocode to figure out which data structures would be the most appropriate. If I had more time, I would try to implement some of the implementations from the technical write-up [1].

6. RESOURCES

I used two main resources for this project. One was a technical write-up from Sven Koenig from Georgia Institute of Technology and Maxim Likhachev from Carnegie Mellon University. The other resource was a YouTube video of a presentation given by a group of students at MIT.

7. REFERENCES

[1] Koenig, S.; Likhachev, M. 2002. D* Lite. *American Association for Artificial Intelligence.*

http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf

[2] https://www.youtube.com/watch?v=_4u9W1xOuts