ALGORITHMS (CSE 211)

Searching & Shortest Path Project

SELECTED ALGORITHM

D* Lite

CANDIDATES

Md Abu Sauri Sufian (1930839)

Md. Rafid Mhotasin Bapi (1931181)

Md Shad Ashfaque Hossain (1930871)

Tahseen Ahmed Bhuiyan (1930731)

Introduction of the Paradigm of the Algorithm:

We know LPA*, which frequently finds the shortest pathways between the start and target vertices periodically. But now, we use LPA* to develop D* Lite, that frequently regulates the shortest paths between the current vertex and the target as the edge costs of a graph change as the edges change and adapt towards the goal vertex. D* Lite doesn’t make supposition about how the edge costs change, it doesn’t matter if they go up or down, whether it changes close to the contemporary vertex or far away from it. In an unknown terrain, D* Lite can also be utilized to solve the goal-directed navigation problem. It can also be used to determine the shortest path of a robot as it moves and changes its path towards its goal. The terrain of the navigation is designed as an eight-connected graph. Its edges have a cost of one. When the robot realizes they can't be traversed, they transform to infinity. You can practice the robot-navigation strategy by using D* Lite to this graph S_{start} being the contemporary vertex of the robot and S_{goal} being the target vertex.

For searching directions, we need to redirect the search directions of LPA*. LPA* g-values are estimations of the start distances as we know it searches from the start vertex to the target or goal vertex. D* Lite is a bit opposite, it searches from the goal vertex to the start vertex and hence its g-values are estimates of the target or goal distances. Basically, it is acquired from the LPA* algorithm by swapping the starting and goal vertex and countering all edges in the pseudo code.

Introduction of the Selected Algorithm:

DynamicSWSF-FP (Ramalingam & Reps 1996) is an incremental search method but are currently not used much in artificial intelligence. They use the information again and again from prior searches to solve a succession of similar search jobs much faster than performing each search task individually from the ground up. A*(Nilsson 1971) is a Heuristic search method which uses its knowledge to approximate goal distances to search and find a solution to the problems much faster compared to uninformed searching techniques or methods. LPA*(Lifelong Planning A*) was recently introduced and it deduces both DynamicSWSF-FP and A*. It uses these two methods to reduce its planning time. We apply LPA* to help robots navigate in unfamiliar terrains. If the robot does come upon a prior unknown obstacle, it can use conventional graph-search techniques to replan its path, though it can take a longer time. Another clever heuristic searching techniques is the Dynamic A* (D*), this method can acquire a speedup of two orders of magnitudes by modifying its prior searches locally over a repeated A* method. D* is being used on real robots. The mobile robot prototypes in the Mars Rover has currently D* being integrated into. We represent D* Lite building on LPA*, algorithmically different but a replanning method which has the same navigation plan as D*. D* lite is considerably quite a bit shorted than D*, when comparing priorities, it uses only one tie-breaking basis for which the maintenance of priorities gets simplified. Due to this, it also does need complex nested if-statements that occupy space which simplifies the program flow analysis. Properties like this allows us to extend it easily and gain efficiency. The properties we have gained from our theories shows that LPA* is efficient and almost the same as A* which is pretty well known and easily understandable algorithm. But, experimental properties also show that D* Lite at its worse is as efficient as D*.

Pseudocode:

```
cancomputeshortestpath
    if end->g < end->rhs
        update key with modifier
    else
        update key with modifier
    while minheap() && (end->rhs > end->g || minheap()<next)
        temp = minheap()
        updateoldkeyaccordinngto temp
        updatekey(temp)
```

```
updatecell()
if temp->g == temp->rhs
updateheap()
else if temp->g>temp->rhs
    for d to directions
        if temp can move to temp->[d]
            initializecell(newcell)
            if newcell is not start and newcell->rhs > temp->g + 1
                update newcell->rhs
                newcell->searchtree = temp
                updatecell(newcell)


else
    newcell->g = inf
    updatecell(temp)


    for d to directions
        if can move temp
            move newcell to temp
            initializecell(newcell)
        if newcell is not start and newcell searchtree equals temp
            updaterhsvalues()
```

```
if end->g < end->rhs
    update key values using modifier
else
    update key value using modifier


return start->rhs == inf
main()
    procssthemaze()
    last=end
    initializethemaze()
    while start != end
        if cannotcomputeshortestpath()
            break
        printknownmaze()
        end->tracertree=null
       updatethemaze()
```

<u>Implementation:</u>

```
void initializecell(cell *thiscell)//c
{

   if (thiscell->generated != mazeiteration)
   {
        thiscell->g = LARGE;
        thiscell->rhs = LARGE;
        thiscell->searchtree = NULL;
        thiscell->generated = mazeiteration;
   }
}

void updatecell(cell *thiscell)//logheap
{

   if (thiscell->g < thiscell->rhs)
   {

        thiscell->key[0] = thiscell->g + H(thiscell) + keymodifier;
        thiscell->key[1] = thiscell->g + H(thiscell) + keymodifier;
        thiscell->key[2] = thiscell->g;
        insertheap(thiscell);
    }
    else if (thiscell->g > thiscell->rhs)
    {

        thiscell->key[0] = thiscell->rhs + H(thiscell) + keymodifier;
        thiscell->key[1] = thiscell->rhs + H(thiscell) + keymodifier + 1;
        thiscell->key[2] = H(thiscell) + keymodifier;

        insertheap(thiscell);
    }
    else
        deleteheap(thiscell);
}

void updatekey(cell *thiscell)
{

    if (thiscell->g < thiscell->rhs)
    {
        thiscell->key[0] = thiscell->g + H(thiscell) + keymodifier;
        thiscell->key[1] = thiscell->g + H(thiscell) + keymodifier;
        thiscell->key[2] = thiscell->g;
    }
    else
    {
        thiscell->key[0] = thiscell->rhs + H(thiscell) + keymodifier;
        thiscell->key[1] = thiscell->rhs + H(thiscell) + keymodifier + 1;
        thiscell->key[2] = H(thiscell) + keymodifier;
    }
}
void updaterhs(cell *thiscell)
{
    int d;
    thiscell->rhs = LARGE;
    thiscell->searchtree = NULL;
    for (d = 0; d < DIRECTIONS; ++d)
    {

        if (thiscell->move[d] && thiscell->move[d]->generated == mazeiteration
&& thiscell->rhs > thiscell->move[d]->g + 1)
        {
            thiscell->rhs = thiscell->move[d]->g + 1;
            thiscell->searchtree = thiscell->move[d];
        }
    }
    updatecell(thiscell);
}

int computeshortestpath()
{
    cell *tmpcell1, *tmpcell2;
    int x, d;

    if (mazegoal->g < mazegoal->rhs)
    {
        goaltmpcell.key[0] = mazegoal->g + keymodifier;
        goaltmpcell.key[1] = mazegoal->g + keymodifier;
        goaltmpcell.key[2] = mazegoal->g;
    }
    else
    {
        goaltmpcell.key[0] = mazegoal->rhs + keymodifier;
        goaltmpcell.key[1] = mazegoal->rhs + keymodifier + 1;
        goaltmpcell.key[2] = keymodifier;
    }
```

```
   while (topheap() && (mazegoal->rhs > mazegoal->g || keyless(topheap(),
&goaltmpcell)))//m*logheap*logheap

   {

        tmpcell1 = topheap();//c

        oldtmpcell.key[0] = tmpcell1->key[0];

        oldtmpcell.key[1] = tmpcell1->key[1];

        oldtmpcell.key[2] = tmpcell1->key[2];


        updatekey(tmpcell1);//c

        if (keyless(&oldtmpcell, tmpcell1))

           updatecell(tmpcell1);//logheapsize

        else if (tmpcell1->g > tmpcell1->rhs)

        {

           tmpcell1->g = tmpcell1->rhs;

           deleteheap(tmpcell1);//logheapsize


           for (d = 0; d < DIRECTIONS; ++d)//logheapsize

           {

                if (tmpcell1->move[d])

                {

                   tmpcell2 = tmpcell1->move[d];

                   initializecell(tmpcell2);//c

                   if (tmpcell2 != mazestart && tmpcell2->rhs > tmpcell1->g + 1)
                {
                    tmpcell2->rhs = tmpcell1->g + 1;
                    tmpcell2->searchtree = tmpcell1;
                    updatecell(tmpcell2);//log heap*log heap
                }
            }
        }
    }
else//logheap
{
    tmpcell1->g = LARGE;
    updatecell(tmpcell1);//log heap

    for (d = 0; d < DIRECTIONS; ++d)
    {

        if (tmpcell1->move[d])
        {
            tmpcell2 = tmpcell1->move[d];
            initializecell(tmpcell2);//c
            if (tmpcell2 != mazestart && tmpcell2->searchtree == tmpcell1)
                updaterhs(tmpcell2);//logheap
        }
```

```
            }
        }

          if (mazegoal->g < mazegoal->rhs)
          {
              goaltmpcell.key[0] = mazegoal->g + keymodifier;
              goaltmpcell.key[1] = mazegoal->g + keymodifier;
              goaltmpcell.key[2] = mazegoal->g;
          }
          else
          {
              goaltmpcell.key[0] = mazegoal->rhs + keymodifier;
              goaltmpcell.key[1] = mazegoal->rhs + keymodifier + 1;
              goaltmpcell.key[2] = keymodifier;
          }

      }

  return (mazegoal->rhs == LARGE);
}
```

For the complete runnable code visit:

https://github.com/d3-c0d3/d-star-lite/blob/main/dstarlitecorecode.c

Time complexity analysis:

```
cancomputeshortestpath
    if end->g < end->rhs
        update key with modifier
    else
        update key with modifier
    while minheap() && (end->rhs > end->g || minheap()<next)---------------------> O(logv)
        temp = minheap()
        updateoldkeyaccordinngto temp
        updatekey(temp)
        updatecell()
        if temp->g == temp->rhs
        updateheap()
        else if temp->g>temp->rhs
            for d to directions
                if temp can move to temp->[d]
                    initializecell(newcell)-----------------------------------------------------> O (logv*logv)
                    if newcell is not start and newcell->rhs > temp->g + 1
                        update newcell->rhs
                        newcell->searchtree = temp
                        updatecell(newcell)
        else
```

```
            newcell->g = inf
            updatecell(temp)
            for d to directions
               if can move temp
                  move newcell to temp
                  initializecell(newcell)
               if newcell is not start and newcell searchtree equals temp
                  updaterhsvalues()
if end->g < end->rhs
   update key values using modifier
else
   update key value using modifier
return start->rhs == inf
main()
   procssthemaze()
   last=end
   initializethemaze()
   while start != end //search for all verteces --------------------------------------->   O(v)
      if cannotcomputeshortestpath() // ---------------------------------------------->   O(logv)*v
         break
      printknownmaze()
      end->tracertree=null
     updatethemaze()
```

Thus the total complexity is O(vlogv)

A brief description of how the code runs:

1. Initialize all nodes as un expanded.

2. search until start is consistent with neighbors and expanded.

3 move to next best vertex.

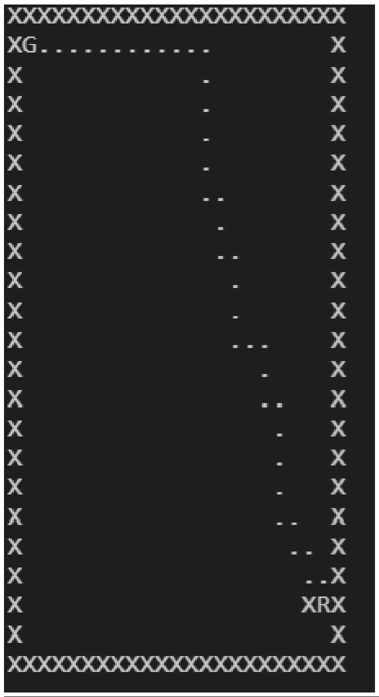
4. check if any edge costs change:

    a. track how heuristic have changed.

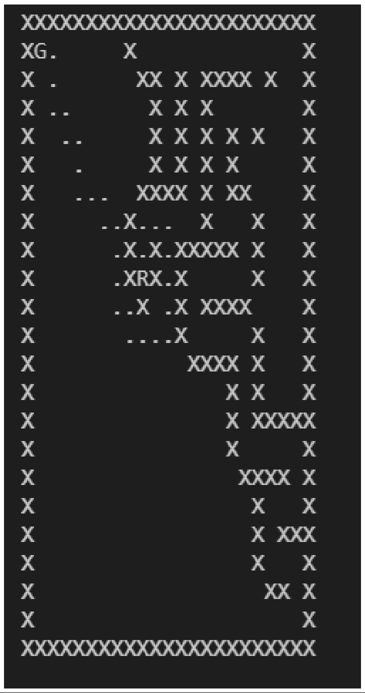
    b. update source nodes of changed edges.

Repeat from 2.

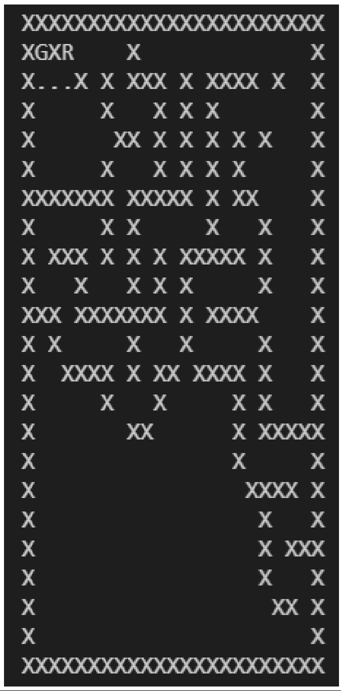
The maze that was formed:



```
XXXXXXXXXXXXXXXXXXXXXXXX
XG.............        X
X             .        X
X             .        X
X             .        X
X             .        X
X             ..       X
X              .       X
X              ..      X
X               .      X
X               .      X
X               ...    X
X                 .    X
X                 ..   X
X                  .   X
X                  .   X
X                  .   X
X                  ..  X
X                   .. X
X                    ..X
X                    XRX
X                      X
XXXXXXXXXXXXXXXXXXXXXXXX
```



```
XXXXXXXXXXXXXXXXXXXXXXX
XG.     X             X
X .      XX X XXXX X  X
X ..      X X X       X
X  ..     X X X X X   X
X   .     X X X X     X
X   ...  XXXX X XX    X
X     ..X...  X   X   X
X      .X.X.XXXXX X   X
X      .XRX.X     X   X
X      ..X .X XXXX    X
X       ....X     X   X
X            XXXX X   X
X               X X   X
X               X XXXXX
X               X     X
X                XXXX X
X                 X   X
X                 X XXX
X                 X   X
X                  XX X
X                     X
XXXXXXXXXXXXXXXXXXXXXXX
```



```
XXXXXXXXXXXXXXXXXXXXXXX
XGXR    X             X
X...X X XXX X XXXX X  X
X     X   X X X       X
X      XX X X X X X   X
X     X   X X X X     X
XXXXXXX XXXXX X XX    X
X     X X     X   X   X
X XXX X X X XXXXX X   X
X   X   X X X     X   X
XXX XXXXXXX X XXXX    X
X X     X   X     X   X
X  XXXX X XX XXXX X   X
X     X   X     X X   X
X       XX      X XXXXX
X               X     X
X                XXXX X
X                 X   X
X                 X XXX
X                 X   X
X                  XX X
X                     X
XXXXXXXXXXXXXXXXXXXXXXX
```

Without obstacle With less obstacle with most obstacle

Use Cases:

D* Lite is an algorithm which has a strong base on research on fast replanning methods in artificial intelligence, robotics and also supplement the research on symbolic planning methods in the field of artificial intelligence. Along with the research on incremental search methods together in algorithm theory and AI.

Practical use and the purpose for using the algorithm:

AGV also known as Automated Guided Vehicle functions as a robot which works following a definite path generated from its system, often used in industrial application. These AGV can also react on its own, for example if any obstacle appears suddenly it automatically detects and dodge the obstacle by itself. This all are possible because of path planning algorithms that have been researched for mobile robot such as A*, LPA*(Lifelong Planning A*), D* and D*Lite algorithms.

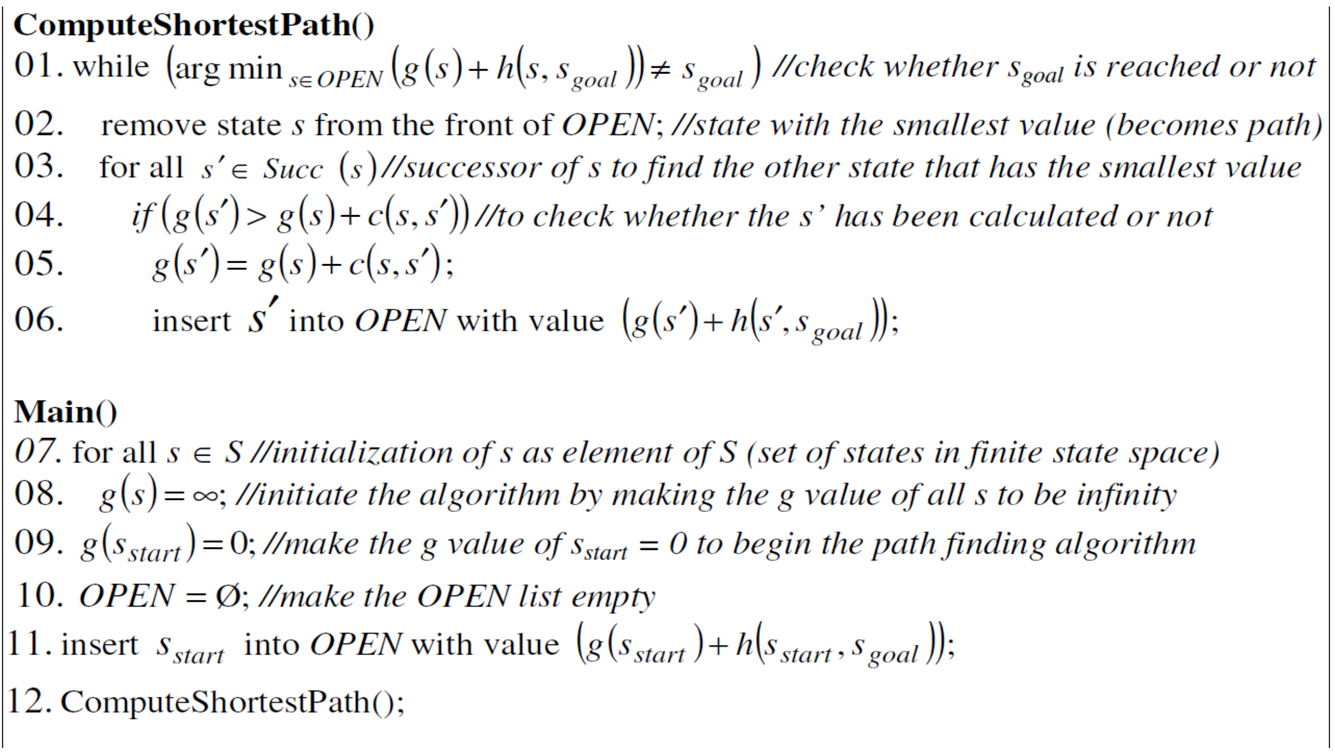
Among all, D* Lite algorithms are mostly used for path planning algorithms for having features like incremental updates and heuristics. Similar algorithm like A* are also noticed. But On the other hand D* lite algorithms are proved to be more efficient on replanning a new path rather than A* algorithm. The reason behind this is that A* replans path from the beginning but D* lite do not replan the path as it would have required information of ambience from the first search.

Recommended Use:

We can use it in the Line following robot(LFR). As we know every university has a tach competition and there we saw the LFR. If we use this algorithm in LFR, the LFR will run very smoothly by avoiding all the obstacles. Also now we are sending many robots in the space for gathering information. If we think about Mars, there's so many unknown obstacles. If we try this advanced algorithm in the robots which are sending in the space, so that may be it will select path more efficiently and we will get more and more information.

Comparison With similar algorithm:

A* algorithm is also a widely used as classical search algorithm which helps to calculated least-cost path on a weighted graphs, and this algorithm is very much similar to D* Lite algorithm.

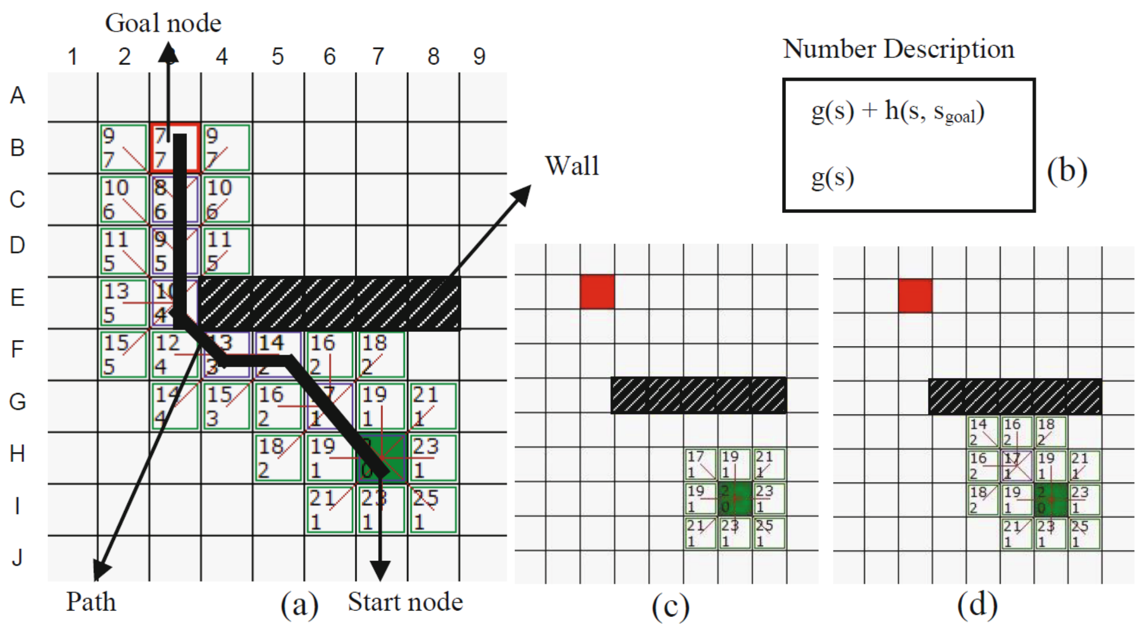


```
ComputeShortestPath()
01. while $(\arg\min_{s \in OPEN}(g(s) + h(s, s_{goal})) \neq s_{goal})$ //check whether $s_{goal}$ is reached or not
02.   remove state $s$ from the front of $OPEN$; //state with the smallest value (becomes path)
03.   for all $s' \in Succ(s)$//successor of s to find the other state that has the smallest value
04.     if $(g(s') > g(s) + c(s, s'))$//to check whether the s' has been calculated or not
05.       $g(s') = g(s) + c(s, s')$;
06.       insert $s'$ into $OPEN$ with value $(g(s') + h(s', s_{goal}))$;

Main()
07. for all $s \in S$ //initialization of s as element of S (set of states in finite state space)
08.   $g(s) = \infty$; //initiate the algorithm by making the g value of all s to be infinity
09. $g(s_{start}) = 0$; //make the g value of $s_{start}$ = 0 to begin the path finding algorithm
10. $OPEN = \emptyset$; //make the OPEN list empty
11. insert $s_{start}$ into $OPEN$ with value $(g(s_{start}) + h(s_{start}, s_{goal}))$;
12. ComputeShortestPath();
```

Pseudocode Of A* Algorithm

A* algorithm begins by calculating the values of the start node. The heuristic value is an estimated value from each node to the goal node. It can be measured by a number of ways i.e. manhattan distance, Euclidean distance etc. The algorithm will evaluate the total cost of the neighboring nodes from the starting node by adding the heuristic value of the node and the actual cost to travel to that node by the formula f(h)=g(h)+h(h). where "f(h)" is the total cost, "g(h)" is the actual cost to travel to that node and "h(h)" the heuristic value of that node. After this, we travel to the node with the least total cost value. Therefore, this node will be the chosen path and its neighbors will be evaluated to find the other node that has the smallest "f(h)" value again. This process will run until the goal node is reached, then the path planning is finished.



Example: A* Algorithm

Both of the algorithm works similarly but the difference stands between them is in the D* Lite algorithm which determines direction and is calculated from goal. As well as we can see both algorithms uses a heuristic and priority based queue in order to enforce its search relevantly reducing its costs which results efficiency. One more example is that if there's a gap of 4 nodes for A* and D* Lite, path re-planning works correspondingly.

The better algorithm can only be identified by examining the size of the state space. By which we can see D*Lite algorithm will have faster path replanning than A* algorithm. For this reason D* Lite algorithm take account of more node in its first search results in faster solutions rather than A*. But if D* Lite starts checking out too many nodes A* algorithm will obviously have an advantage.

Conclusion:

D*lite algorithm can be seen as an improved version of LPA*. It is mainly used in path planning where obstacles might be faced and the algorithm needs to change paths and adapt accordingly. D*lite is hard to implement but if implemented well, it will run more smoothly than any other algorithm if the problem is one which needs replanning due to facing unforeseen obstacles. The time complexity is O(V*logV) which is greater than that of A* (which is O(E)). But, A* fails to adapt according to the situation as quickly and efficiently.

Reference:

https://youtu.be/_4u9W1xOuts

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

http://idm-lab.org/code/dstarlite.tar

https://www.geeksforgeeks.org/search-algorithms-in-ai/?ref=lbp

https://www.geeksforgeeks.org/a-search-algorithm/

https://stackoverflow.com/questions/2900718/where-can-i-find-information-on-the-d-or-d-lite-pathfinding-algorithm

https://aaai.org/Papers/AAAI/2002/AAAI02-072.pdf

https://www.researchgate.net/figure/Illustration-of-D-Lite-algorithm-path-planning_fig4_321011666