ANALYSIS OF DIJKSTRA'S AND A* ALGORITHM TO FIND THE SHORTEST PATH

A Project submitted as J Component
for the Data structure and Algorithm of the
Degree of Bachelor of Computer Science and Engineering

Faculty of Computer Science and Engineering
Vellore Institute of Technology
Vellore , Tamilnadu-632014

CSE2003 – Data Structures and Algorithms

J Component - Project Proposal

Slot: L39+L40+L41+L442

24th June,2019

AKASH KUMAR YADAV

KISMAT KHATRI

SUSHAN GAUTAM

ABSTRACT

Introduction/Background

In the road network application, the A* algorithm can achieve better running time than Dijkstra's. The restricted algorithm can find the optimal path within linear time but the restricted area has to be carefully selected. The selection actually depends on the graph itself. This algorithm can be used in a way that allowing search again by increasing the factor if the first search fails.

Objective

Shortest Path problems are inevitable in road network applications such as city emergency handling and drive guiding system, in where the optimal routings have to be found. As the traffic condition among a city changes from time to time and there are usually a huge amounts of requests occur at any moment, it needs to quickly find the solution. Therefore, the efficiency of the algorithm is very important. Some approaches take advantage of preprocessing that compute results before demanding. These results are saved in memory and could be used directly when a new request comes up. This can be inapplicable if the devices have limited memory and external storage. This project aims only at investigate the single source shortest path problems and intends to obtain some general conclusions by examining three approaches, Dijkstra's shortest path algorithm, Restricted search algorithm and A* algorithm.To verify the three algorithms, a program was developed under Microsoft Visual C++environment. The two algorithms was implemented and visually demonstrated. The road network example is a graph data file containing partial transportation data of the Ottawa city.

Method

Dijkstra's Shortest Path Algorithm

The Dijkstra's shortest path algorithm is the most commonly used to solve the single sourse shortest path problem Today.

A* Search

The A* algorithm integrates a heuristic into a search procedure. Instead of choosing the next node with the least cost (as measured from the start node), the choice of node is based on the cost from the start node plus an estimate of proximity to the destination.

PROBLEM STATEMENT

- Path finding generally refers to finding the shortest path between any two locations. Many existing algorithms are designed precisely to solve the shortest path problem such as Genetic, Floyd algorithm.

- The method proposed in this study is A* algorithm and Dijkstra's, will probably find the shortest path solution in a very short amount of time and minimum distance.

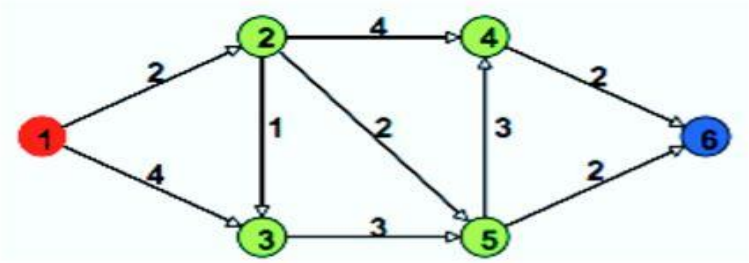
- A* algorithm, a kind of informed search, is widely used for finding the shortest path, because the location of starting and ending point is taken into account beforehand.

- A* use the heuristic function to speed up the runtime. The general purpose of heuristic algorithm is to find an optimal solution where the time or resources are limited. Dijkstra algorithm is simple and excellent method for path planning.

- Dijkstra's algorithm chooses one with the minimum cost until found the goal, but the search is not over as it calculates all possible paths from starting node to the goal, then choose the best solution by comparing which way had the minimum distance.
- Dijkstra's algorithm is a special case of A* star algorithm where heuristic is zero. It remains current because it is realistically fast and relatively easy to implement.

SAMPLE INPUT AND OUTPUT

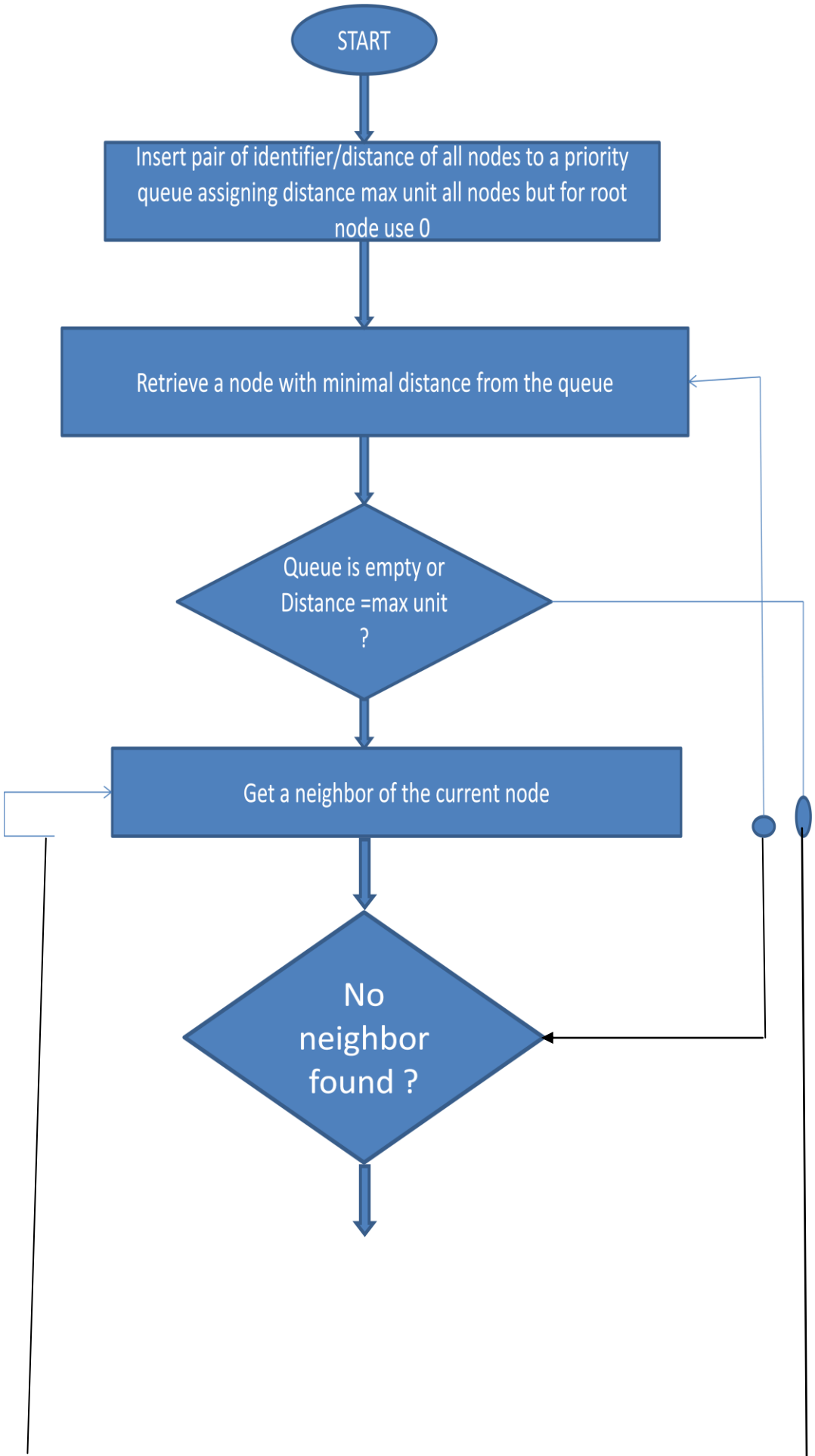
Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative.

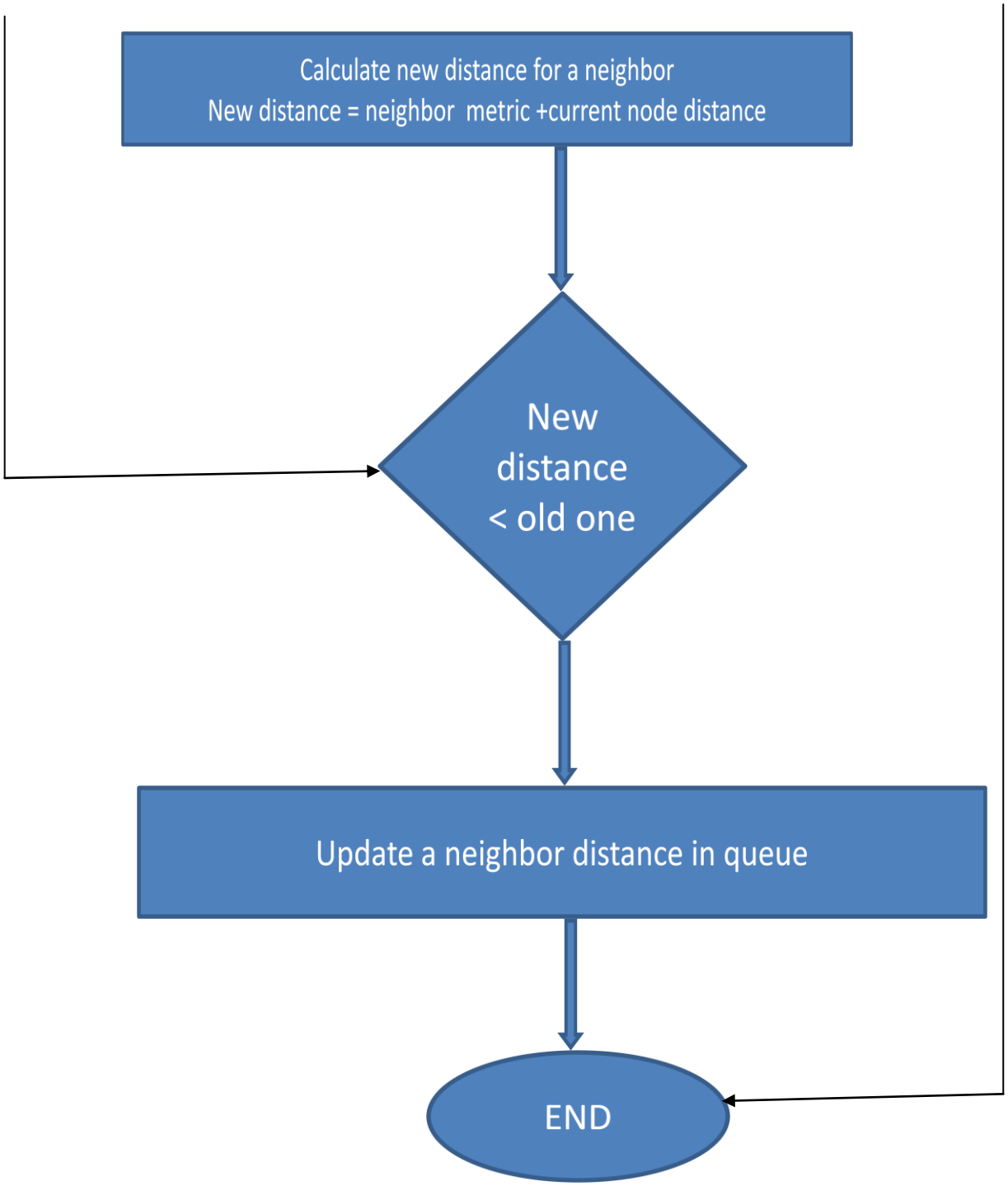


Output: Lengths and shortest paths from a given source vertex $v \in V$ to all other vertices.

Distance :- 6
Path :- {1,2,5,6}

FLOWCHART


START
Insert pair of identifier/distance of all nodes to a priority queue assigning distance max unit all nodes but for root node use 0
Retrieve a node with minimal distance from the queue
Queue is empty or Distance =max unit ?
Get a neighbor of the current node
No neighbor found ?



Dijkstra Flowchart

Algorithm & Working

- Dijkstra's algorithm is sometimes called the single-source shortest path because it solves the single-source shortest-path difficulty on a subjective, directed graph (G = V, E) where
- V is a set whose element is called vertices (nodes, junctions, or intersections) and E is a set of ordered pairs of vertices entitled directed edges (arcs or road segments). To find a shortest path from a source s vertex or location to a destination location d, Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the sources that already been determined. Knowing that w is the edge weight, the edge is an ordered pair (u, v) and assuming w (u, v) ≥ 0 for each edge (u, v) ϵ E, the algorithm recurrently chooses the vertex u ϵ V – S with the least shortest path approximation, adds u to S, and relaxes all edges leaving u.
  - Step 1: Node A is set to become current node. Zero is assigned to node A and infinity to all other nodes.

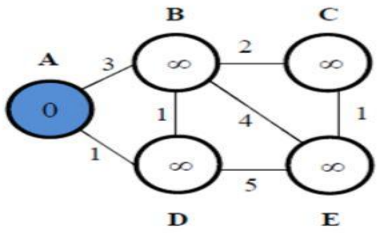


Figure2.10: First step of Dijkstra's Algorithm

- Step 2: Consider all unvisited neighbors and tentative distance will be calculated. Previously recorded value will be replaced since new value less than infinity.

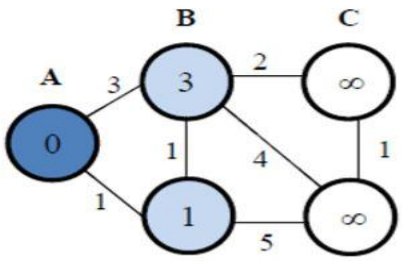


Figure2.11: Second step of Dijkstra's Algorithm

- Step 3: Since all neighbors of node A havebeen taken into account, it is struck as visited and will not be tested again.

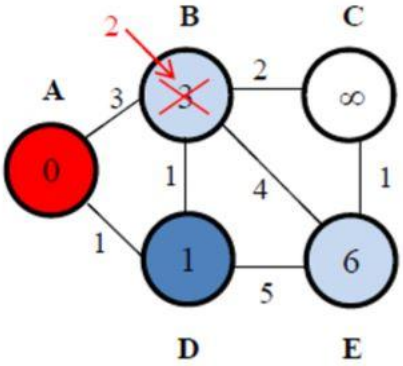


Figure 2.12: Third step of Dijkstra's Algorithm

- Step 4: Since all neighbors of node D have been taken into account, it is marked as visited and will not be checked over.

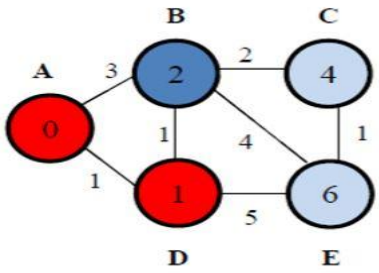


Figure 2.13: Step 4 of Dijkstra's Algorithm

- Step 5: Since all neighbours of node B have been accounted for, it is marked as visited and will not be testedover.

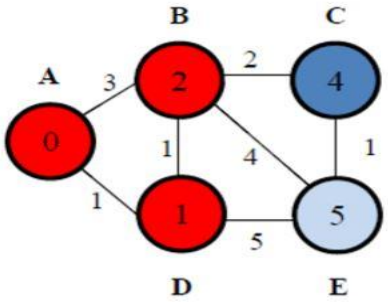


Figure 2.14: Fifth step of Dijkstra's Algorithm

- Step 6: Meanwhile, all neighbours of node C have been taken into account, it is marked as visited and will not be checked.

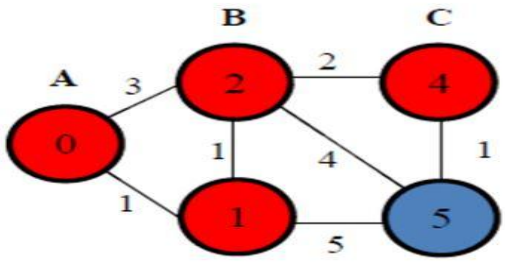


Figure2.15: Step 6 of Dijkstra's Algorithm

Dijkstra's shortest path Pseudo code

- for each vertex u in G:

```
        d[u] = infinity;
        parent[u] = NIL;
End for
    d[s] = 0;  // s is the start point
        H = {s};  // the heap
while NotEmpty(H) and targetNotFound:
        u = Extract_Min(H);
        label u as examined;
        for each v adjacent to u:
            if d[v] > d[u] + w[u, v]:
            d[v] = d[u] + w[u, v];
            parent[v] = u;
            DecreaseKey[v, H];
```

A* algorithm is a universal space-search algorithm that can be used to find the clarifications to many problems with shortest path as one of such,It has been used in several real-time strategy games and is perhaps the most popular shortest path algorithm.

A* is the most popular choice for path finding, because it is fairly flexible and can be used in a wide range of contexts.

- Step1 : A* algorithm uses a starting point and an endpoint point to produce the desired path, if it exists . In the given figure, the cell marked with “O” is the starting point/node and the cell marked with “X” is the destination. The white squares are walkable nodes and the black ones are walls, shelves or any other obstacles.

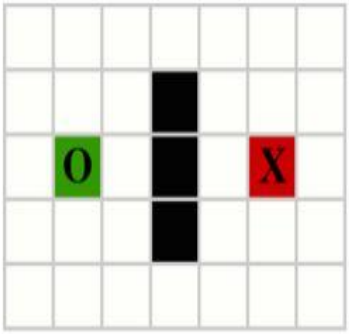


Figure 2.17: Starting and Destination Points in A* Search Algorithm

- Step2 : A* algorithm starts to perform by looking at the starting node first and then expansion to the surrounding nodes.

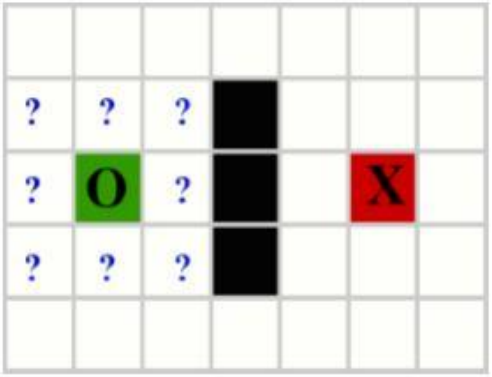


Figure 2.18: Starting the A* Search Algorithm

- Step3 : A* algorithm requests a way to keep track of the nodes. the nodes to be scrutinized are held in a list, called an Open List. after inspecting all of its surrounding nodes and place in another list named the Closed List.

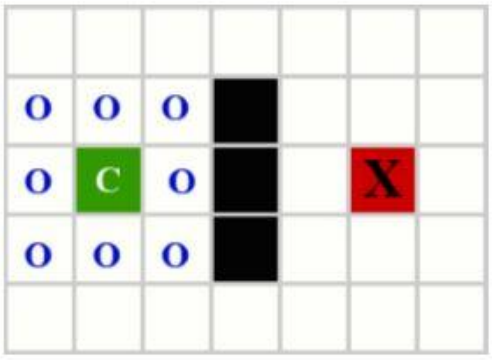


Figure 2.19:Putting the neighbouring cells to the Open List

- Step4 : A* loop However, need to track some supplementary information. Need to know how the nodes are linked . Although the Open List maintains a list of adjacent nodes, we need to know how the adjacent nodes are linked as well. Can do this by tracking the parent node of each node in the Open List.

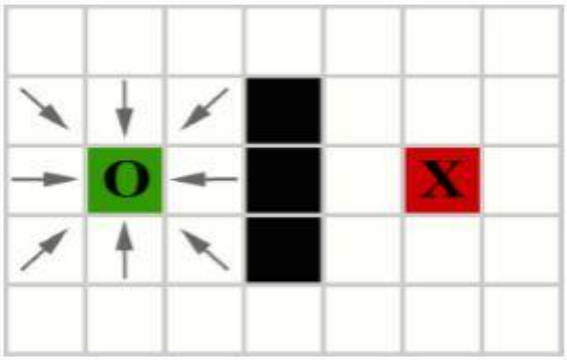


Figure 2.20: Parent relation of Starting Node

We'll use the parent links to trace a path back to the starting node when reach the destination. At this point the process was recommenced. Now have to choose a new node to check from the Open List.At the first iteration there is only a single node in the Open

List. Now have eight nodes in the Open List, and the node which will first be inspected is determined by assigning a score to each node. This score, f(n), is the combination of two scores:

$$f(n) = g(n) + h(n)$$

Where g(n) is the cost of the path from the starting node to any node n, h(n) is the heuristic estimated cost from any node n to the goal.

A* Search Algorithm Pseudocode

f(V) = distance from S to V + estimate of the distance to D.

$$= d(V) + h(V,D)$$

$$= d(V) + \text{sqrt}((x(V) - x(D))^2 + (y(V) - y(D))^2)$$

where x(V), y(D) and x(V), y(D) are the coordinates for node V and the destination node D.

The A* Search algorithm:

```
for each u  G:
        d[u] = infinity;
        parent[u] = NIL;
End for  d[s] = 0;
        f(V) = 0;
        H = {s};
while NotEmpty(H) and targetNotFound:
        u = Extract_Min(H);
label u as examined;
```

for each v adjacent to u:

if d[v] > d[u] + w[u, v] , then

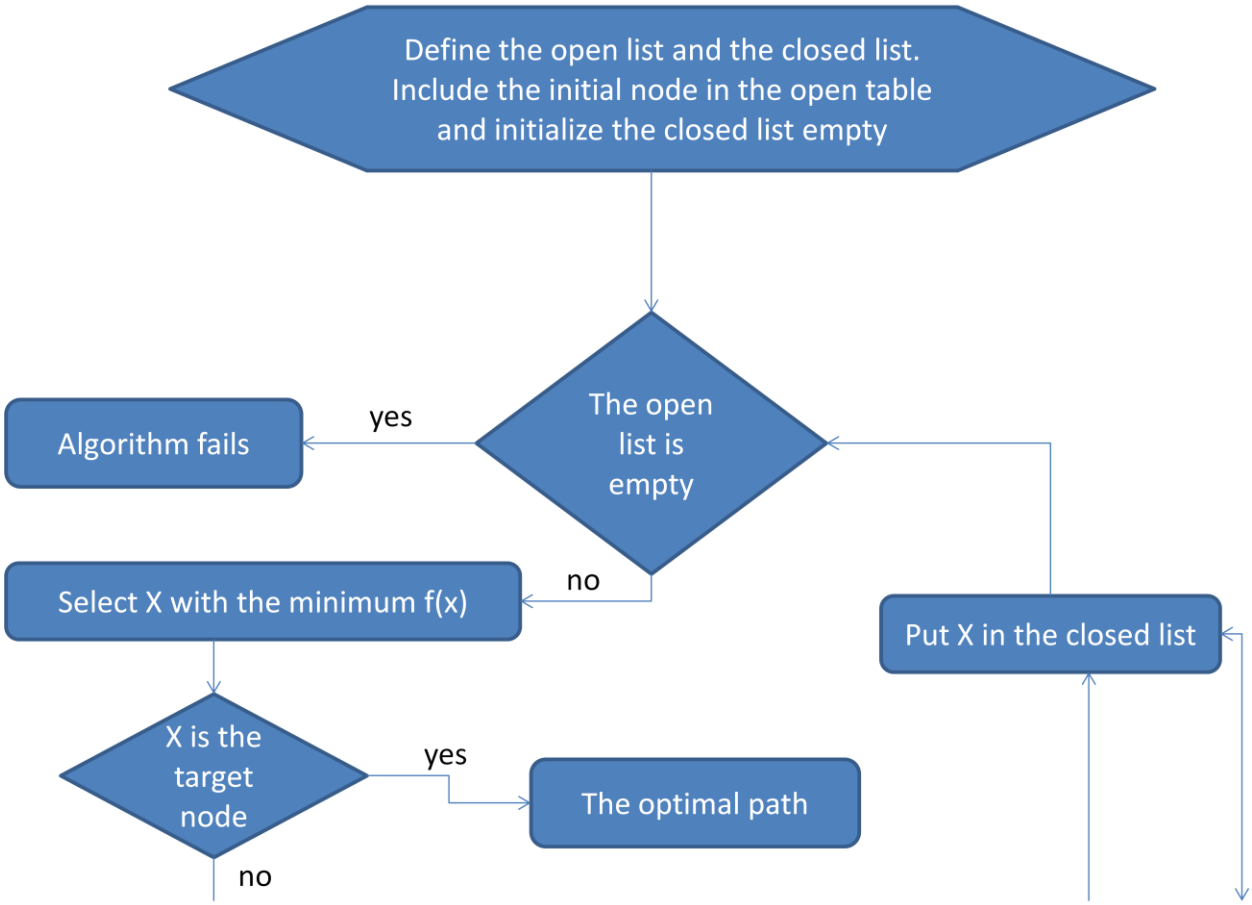
d[v] = d[u] + w[u, v];

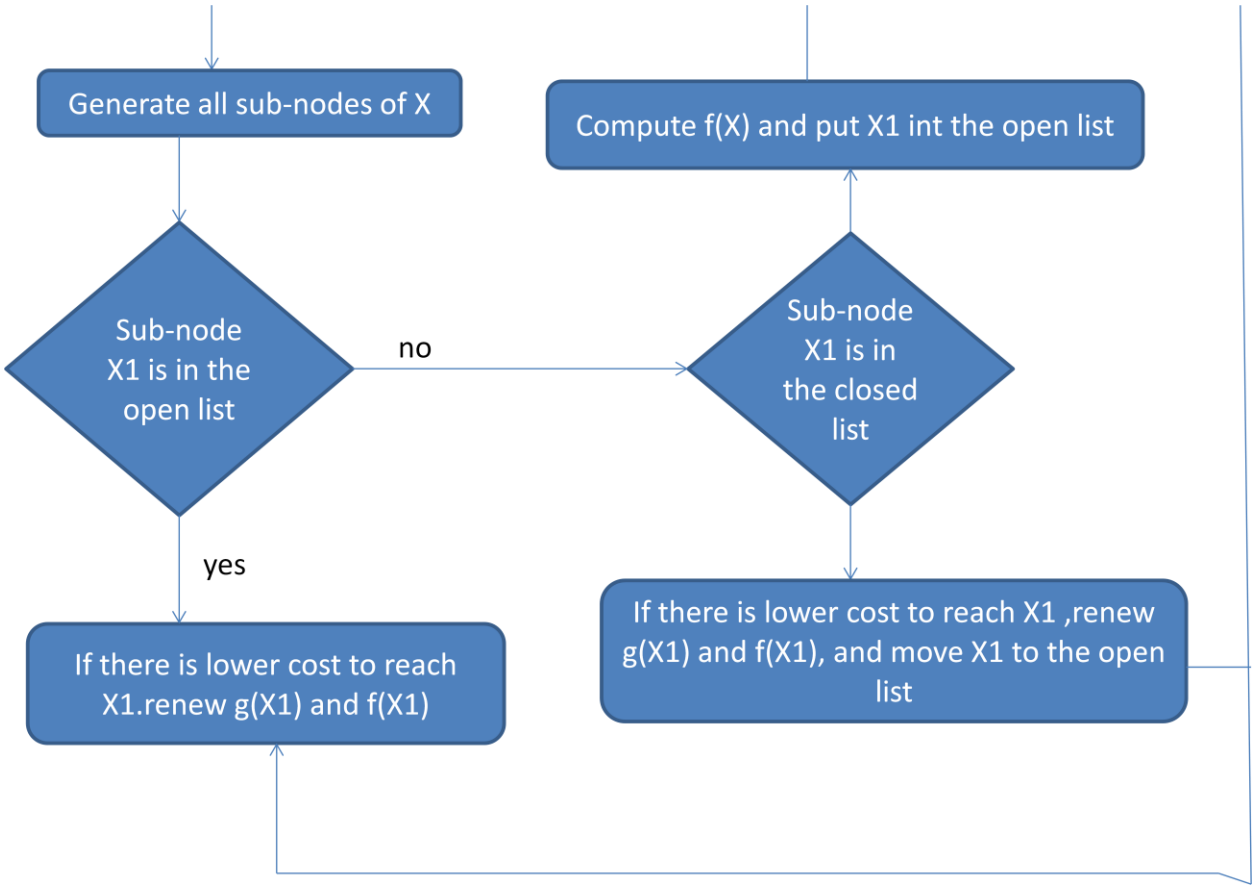
p[v] = u;

f(v) = d[v] + h(v, D);

DecreaseKey[v, H];

A* Flowchart





IMPLEMENTATION

A STAR CODE

```
%DEFINE THE 2-D MAP ARRAY

MAX_X=10;

MAX_Y=10;

MAX_VAL=10;

%This array stores the coordinates of the map and the

%Objects in each coordinate
```

```
MAP=2*(ones(MAX_X,MAX_Y));


% Obtain Obstacle, Target and Robot Position
% Initialize the MAP with input values
% Obstacle=-1,Target = 0,Robot=1,Space=2
j=0;
x_val = 1;
y_val = 1;
axis([1 MAX_X+1 1 MAX_Y+1])
grid on;
hold on;
n=0;%Number of Obstacles
% BEGIN Interactive Obstacle, Target, Start Location selection
pause(1);
h=msgbox('Please Select the Target using the Left Mouse button');
uiwait(h,5);
if ishandle(h) == 1
   delete(h);
end
xlabel('Please Select the Target using the Left Mouse
button','Color','black');
but=0;
while (but ~= 1) %Repeat until the Left button is not clicked
```

```
    [xval,yval,but]=ginput(1);
end
xval=floor(xval);
yval=floor(yval);
xTarget=xval;%X Coordinate of the Target
yTarget=yval;%Y Coordinate of the Target
 MAP(xval,yval)=0;%Initialize MAP with location of the target
plot(xval+.5,yval+.5,'gd');
text(xval+1,yval+.5,'Target')
pause(2);
h=msgbox('Select Obstacles using the Left Mouse button,to select the last obstacle use the Right button');
  xlabel('Select Obstacles using the Left Mouse button,to select the last obstacle use the Right button','Color','blue');
uiwait(h,10);
if ishandle(h) == 1
    delete(h);
end
while but == 1
    [xval,yval,but] = ginput(1);
    xval=floor(xval);
    yval=floor(yval);
    MAP(xval,yval)=-1;%Put on the closed list as well
```

```
    plot(xval+.5,yval+.5,'ro');
 end%End of While loop
 pause(1);
 h=msgbox('Please Select the Vehicle initial position using the Left Mouse button');
uiwait(h,5);
if ishandle(h) == 1
    delete(h);
end
xlabel('Please Select the Vehicle initial position ','Color','black');
but=0;
while (but ~= 1) %Repeat until the Left button is not clicked
    [xval,yval,but]=ginput(1);
    xval=floor(xval);
    yval=floor(yval);
end
xStart=xval;%Starting Position
yStart=yval;%Starting Position
MAP(xval,yval)=1;
 plot(xval+.5,yval+.5,'bo');
%End of obstacle-Target pickup
```

```
OPEN=[];
CLOSED=[];
 %Put all obstacles on the Closed list
k=1;%Dummy counter
for i=1:MAX_X
    for j=1:MAX_Y
        if(MAP(i,j) == -1)
            CLOSED(k,1)=i;
CLOSED(k,2)=j;
            k=k+1;
        end
    end
end
CLOSED_COUNT=size(CLOSED,1);
%set the starting node as the first node
xNode=xval;
yNode=yval;
OPEN_COUNT=1;
path_cost=0;
goal_distance=distance(xNode,yNode,xTarget,yTarget);
OPEN(OPEN_COUNT,:)=insert_open(xNode,yNode,xNode,yNode,p
ath_cost,goal_distance,goal_distance);
OPEN(OPEN_COUNT,1)=0;
```

```
CLOSED_COUNT=CLOSED_COUNT+1;
CLOSED(CLOSED_COUNT,1)=xNode;
CLOSED(CLOSED_COUNT,2)=yNode;
NoPath=1;
 while((xNode ~= xTarget || yNode ~= yTarget) && NoPath == 1)
%  plot(xNode+.5,yNode+.5,'go');
exp_array=expand_array(xNode,yNode,path_cost,xTarget,yTarget,C
LOSED,MAX_X,MAX_Y);
 exp_count=size(exp_array,1);
for i=1:exp_count
    flag=0;
    for j=1:OPEN_COUNT
        if(exp_array(i,1) == OPEN(j,2) && exp_array(i,2) == OPEN(j,3) )
            OPEN(j,8)=min(OPEN(j,8),exp_array(i,5));
%#ok<*SAGROW>
            if OPEN(j,8)== exp_array(i,5)
                %UPDATE PARENTS,gn,hn
                OPEN(j,4)=xNode;
                OPEN(j,5)=yNode;
                OPEN(j,6)=exp_array(i,3);
                OPEN(j,7)=exp_array(i,4);
            end;%End of minimum fn check
            flag=1;
```

```
        end;%End of node check
%         if flag == 1
%             break;
    end;%End of j for
    if flag == 0
        OPEN_COUNT = OPEN_COUNT+1;

OPEN(OPEN_COUNT,:)=insert_open(exp_array(i,1),exp_array(i,2),x
Node,yNode,exp_array(i,3),exp_array(i,4),exp_array(i,5));
     end;%End of insert new element into the OPEN list
 end;%End of i for
%Find out the node with the smallest fn
  index_min_node = min_fn(OPEN,OPEN_COUNT,xTarget,yTarget);
  if (index_min_node ~= -1)
   %Set xNode and yNode to the node with minimum fn
   xNode=OPEN(index_min_node,2);
   yNode=OPEN(index_min_node,3);
   path_cost=OPEN(index_min_node,6);%Update the cost of reaching
the parent node
  %Move the Node to list CLOSED
  CLOSED_COUNT=CLOSED_COUNT+1;
  CLOSED(CLOSED_COUNT,1)=xNode;
  CLOSED(CLOSED_COUNT,2)=yNode;
  OPEN(index_min_node,1)=0;
```

```
  else
      %No path exists to the Target!!
      NoPath=0;%Exits the loop!
  end;%End of index_min_node check
end;%End of While Loop
%Once algorithm has run The optimal path is generated by starting of at the
%last node(if it is the target node) and then identifying its parent node
%until it reaches the start node.This is the optimal path
 i=size(CLOSED,1);
Optimal_path=[];
xval=CLOSED(i,1);
yval=CLOSED(i,2);
i=1;
Optimal_path(i,1)=xval;
Optimal_path(i,2)=yval;
i=i+1;

if ( (xval == xTarget) && (yval == yTarget))
    inode=0;
   %Traverse OPEN and determine the parent nodes
   parent_x=OPEN(node_index(OPEN,xval,yval),4);%node_index returns the index of the node
```

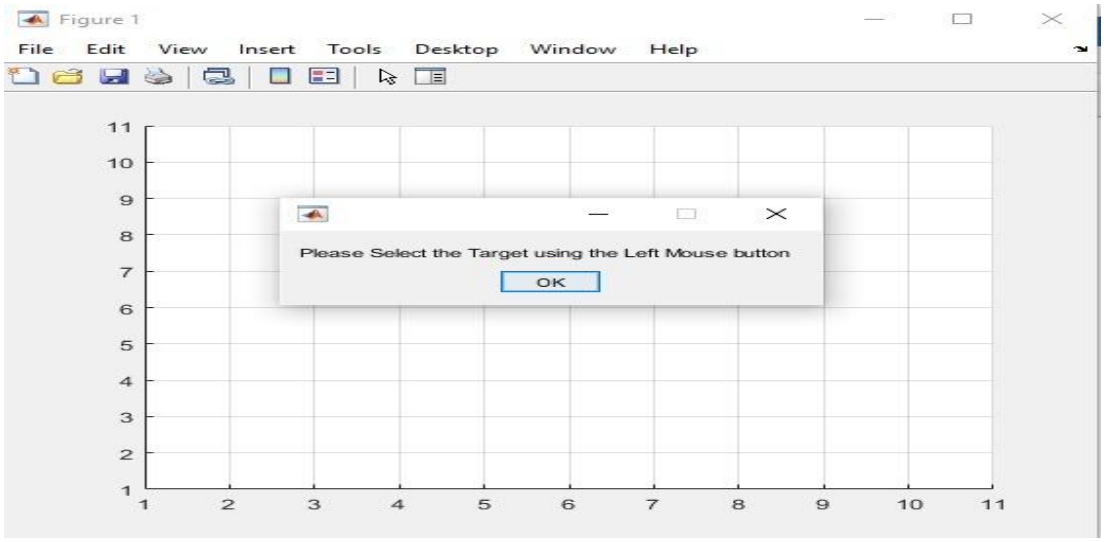
```
   parent_y=OPEN(node_index(OPEN,xval,yval),5);


   while( parent_x ~= xStart || parent_y ~= yStart)
          Optimal_path(i,1) = parent_x;
          Optimal_path(i,2) = parent_y;
          %Get the grandparents:-)
          inode=node_index(OPEN,parent_x,parent_y);
          parent_x=OPEN(inode,4);%node_index returns the index of
the node
          parent_y=OPEN(inode,5);
          i=i+1;
    end;
 j=size(Optimal_path,1);
 %Plot the Optimal Path!
p=plot(Optimal_path(j,1)+.5,Optimal_path(j,2)+.5,'bo');
 j=j-1;
 for i=j:-1:1
  pause(.25);
  set(p,'XData',Optimal_path(i,1)+.5,'YData',Optimal_path(i,2)+.5);
 drawnow ;
 end;
 plot(Optimal_path(:,1)+.5,Optimal_path(:,2)+.5);
else
```

```
 pause(1);
 h=msgbox('Sorry, No path exists to the Target!','warn');
 uiwait(h,5);
end
```

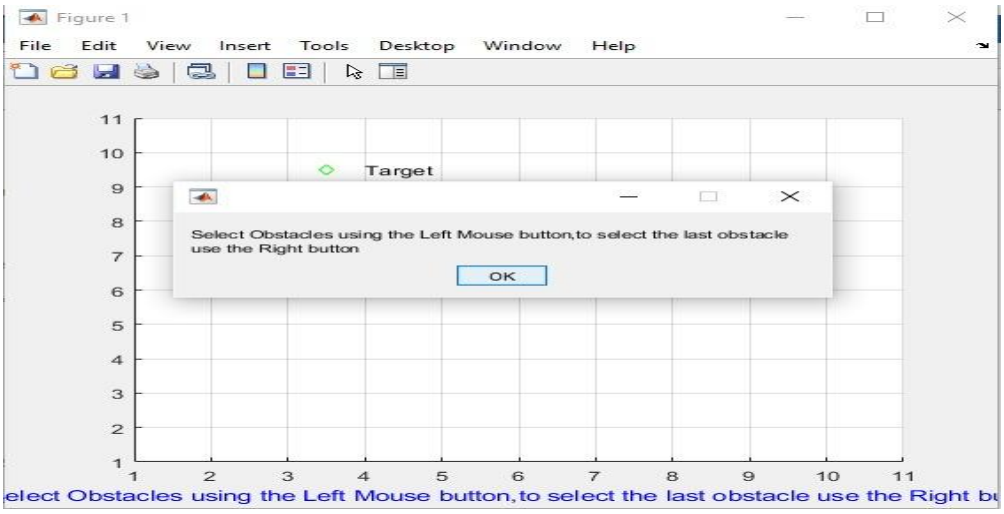
IMPLEMENTATION

On a Graph with Obstacles

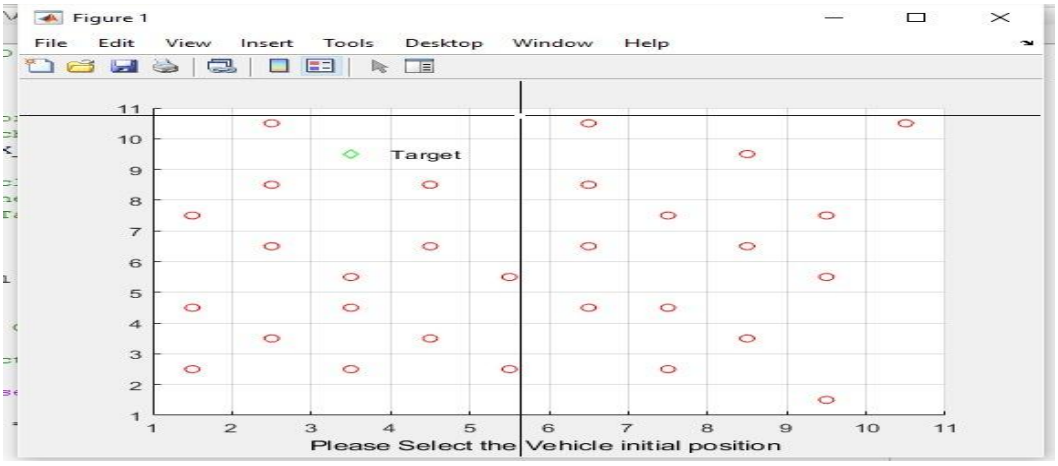
- Step 1: We created a graph :



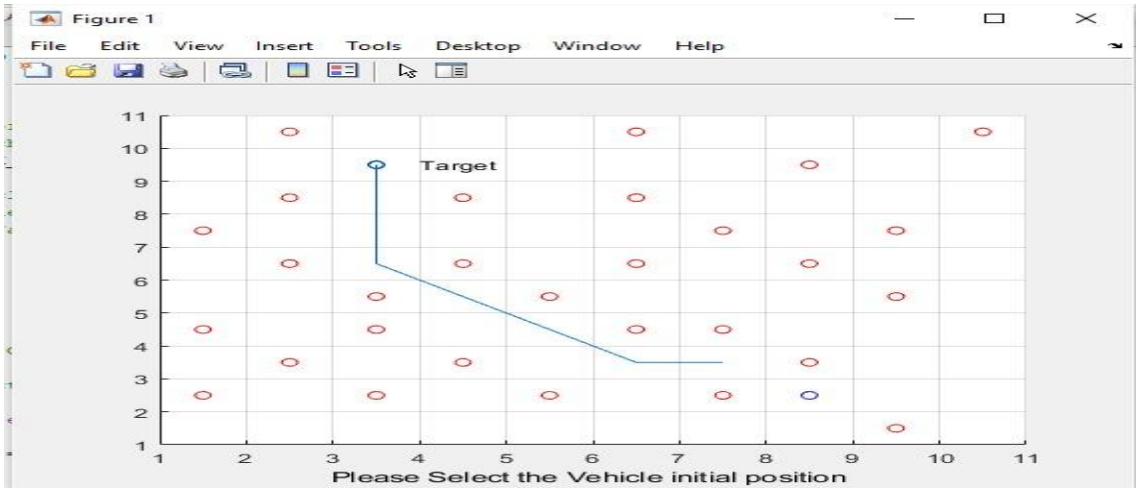
- Step 2: Select the destination where you want to go !
- Step 3: Put out the obstacles on the way !



- Step 4: Now, select the position of vechicle.



Output :



DIJKSTRA'S CODE

The Dijkstra's shortest path algorithm is the most commonly used to solve the single source shortest path problem today. For a graph G(V, E), where V is the set of vertices and E is the set of edges, the running time for finding a path between two vertices varies when different data structure are used. This project uses binary heap to implement Dijkstra's algorithm although there are some data structures that may slightly improve the time complexity, such as Fibonacci heap that can purchase time complexity of O(V*log(V)).

```
function P = createTransitionCostMat(T)
% This is the function to create a transition matrix.
% Consider a terrain map T of m x n, we will find the cost from one point
%in the map to the rest of other points in the map.
% P(toNode, fromNode): transition cost from fromNode to toNode
% Its value is ranging from 0 to inf.
% Cardinality of the nodes in the terrain map T of m x n:
%   1   2   3   4   5  ... n
%  n+1 n+2 n+3 n+4 n+5 ... 2n
%   .   .   .   .   .     .
%   .   .   .   .   .     .
```

```
%   .   .   .   .   .  ... mxn
% From one node, we can only move one step at a time to the sorrounding
% nodes (up, down, left, right, upper left, upper right, lower left, and
% lower right).
% manurung.auralius@gmail.com
% 17.11.2012
[m, n] = size(T);
P = inf * ones(m * n); % This generates (m*n) x (m*n) matrix
% For every point of terrain matrix T (pivot point), we will compute its
% cost to all other points of the same terrain matrix T.
for pivotR = 1 : m
    for pivotC = 1 : n
        fromNode = (pivotR - 1) * n + pivotC;
        % Upward
        if pivotR > 1
            r = pivotR - 1;
            c = pivotC;
            toNode = (r - 1) * n + c;
            P(toNode, fromNode) = max(0, T(r, c) - T(pivotR, pivotC));
        end
        % Downward
        if pivotR < m
```

```
            r = pivotR + 1;
            c = pivotC;
            toNode = (r - 1) * n + c;
            P(toNode, fromNode) = max(0, T(r, c) - T(pivotR, pivotC));
        end
    % Leftward
        if pivotC > 1
            r = pivotR ;
            c = pivotC - 1;
            toNode = (r - 1) * n + c;
            P(toNode, fromNode) = max(0, T(r, c) - T(pivotR, pivotC));
        end
        % Rightward
        if pivotC < n
            r = pivotR ;
            c = pivotC + 1;
            toNode = (r - 1) * n + c;
            P(toNode, fromNode) = max(0, T(r, c) - T(pivotR, pivotC));
        end
        % Down Rightward
        if pivotC < n && pivotR < m
            r = pivotR + 1;
            c = pivotC + 1;
```

```
            toNode = (r - 1) * n + c;
            P(toNode, fromNode) = max(0, T(r, c) - T(pivotR, pivotC));
        end
        % Upper Rightward
        if pivotC < n && pivotR > 1
            r = pivotR - 1;
            c = pivotC + 1;
            toNode = (r - 1) * n + c;
            P(toNode, fromNode) = max(0, T(r, c) - T(pivotR, pivotC));
        end
        % Down Leftward
        if pivotC > 1 && pivotR < m
            r = pivotR + 1;
            c = pivotC - 1;
            toNode = (r - 1) * n + c;
            P(toNode, fromNode) = max(0, T(r, c) - T(pivotR, pivotC));
        end
        % Upper Leftward
        if pivotC > 1 && pivotR > 1
            r = pivotR - 1;
            c = pivotC - 1;
            toNode = (r - 1) * n + c;
            P(toNode, fromNode) = max(0, T(r, c) - T(pivotR, pivotC));
```

```
        end
        P(fromNode, fromNode) = 0;
    end
end
function [stageCostMat, predMat, converged] = dpa(P, startNode, endNode, maxIteration)
% This is the function that will perform the DPA
% Assume we have n number of nodes. P matrix is the transition cost matrix
% with dimension of n x n(square matrix). P(toNode, fromNode) shows
% transition cost from fromNode to toNode.
% stageCostMat shows the cost at each node for current iteration.
% stageCostMat(c) = current stage cost matrix at node c.
%
% predMat shows parent/predecessor node of each node for every stage.
% predMat(c, s): parent of node c during stage s.
% Is the algortihm converged?
converged = 0;
% Cost matrix is a square matrix, m = n
[m, n] = size(P);
% Assume we will probably converge after n stages
stageCostMat = ones(1, m) * inf;
```

```
% Initial cost, no initial cost
stageCostMat(startNode) = 0;
% Predecessor matrix to trace back the optimum path, we will record parent of
% each node on each iteration
predMat = zeros(m, maxIteration);

% Stage-by-stage, we move from start node to terminal node
for stage = 2 : maxIteration
% Find connection from any nodes to any nodes, keep the smaller cost
    prevStageCostMat = stageCostMat;
    stageCostMat = ones(1, m) * inf;
    for fromNode = 1 : m
        for toNode = 1 : m
            aij = P(toNode, fromNode);
            dj = aij + prevStageCostMat(fromNode);
            if dj < stageCostMat(toNode)
                stageCostMat(toNode) = dj;
                predMat(toNode, stage) = fromNode;
            end
        end % end toNode
    end % end fromNode
```

```
    % Termination
%     if (stageCostMat == prevStageCostMat)
%         converged = 1;
%         break;
%     end
    if (predMat(endNode, stage) == endNode) || (predMat(endNode, stage-1) > 0) && (predMat(endNode, stage) == predMat(endNode, stage-1))
        converged = 1;
        break;
    end
end
predMat = predMat(:, 1:stage);
function drawTerrain(T)
[m,n] = size(T);
[X,Y] = meshgrid(1 : n, 1 : m);
surf(X, Y, T(1 : m, 1 : n));
end
% This is the main simulation file.
clc;
clear all;
syms x y
```

- x=input('starting = ');
- y=input('Destination = ');
- START_NODE = x; % Terrian1.mat, from 1 to 4225
- % Terrian2.mat, from 1 to 16641
- END_NODE = y;
-
- % Warning: Terrain2.mat is a huge map, it will takes quite a lot of time to
- % proceed.
-
- load ('Terrain2.mat');
- % load ('Terrain2.mat');
-
- figure;
- hold on;
- visualizeTerrain(T);
-
- P = createTransitionCostMat(T);
- [stageCostMat, predMat, converged] = dpa(P, START_NODE, END_NODE, 1000);
-
- % Keep in mind that DPA propagates through all the nodes. Basically DPA
- % tries to find optimal path from one nodes to all nodes.

```
[optimalPath, optimalCost] = processPredMat(stageCostMat, predMat, START_NODE, END_NODE);
visualizePath(T, optimalPath);
function visualizePath(T, optimalPath)

[m, n] = size(T);
l = length(optimalPath);

% Convert back the node cardinal number to the corresponding xyz coordinate.
x = zeros(1, l);
y = zeros(1, l);
z = zeros(1, l);
for i = 1 : l
    x(i) = mod(optimalPath(i) - 1, n) + 1;
    y(i) = abs((optimalPath(i) - 1 - mod(optimalPath(i) - 1, n)) / n) + 1;
    z(i) = 1.0 + T(y(i), x(i));
end
% Draw the optimal path as line.
plot3(x, y, z, 'r', 'LineWidth', 2)
% Draw asterisk symbol (*) at destination nodes.
plot3(x(l), y(l), z(l), '*m', 'LineWidth', 4)
```

- function visualizeTerrain(T)
- % Visualize the terrain.
- % Don't forget to hold the figure when you want to visualize the optimal
- % path using visualizePath function.
- [m, n] = size(T);
- [X,Y] = meshgrid(1 : n, 1 : m);
- surf(X, Y, T(1 : m, 1 : n));
- end

Dijkstra Time Complexity

- Time Complexity: The run time of first for loop is O(V). In each iteration of the while loop, Extract_Min of the heap is logV. The inner for loop iterates each adjacent node of the current node, thetotal run time is O(E). Therefore, the time complexity of this algorithm is O((V + E)*log(V) = O(E* log(V)). As the number of nodes in a graph increases, the running time of the applied algorithm will become longer and longer. Usually, a road network of a city has more than 10^4 nodes. A fast shortest path algorithm becomes more desirable.

A Star Time Complexity

- Time Complexity:This algorithm does not improve worst case time complexity, but it improves average time complexity. The shortest path search starts from start point and expands node that goes towards the destination.

Therefore, the run time is much shorter than the Dijkstra's algorithm.

Correctness

- Correctness: The algorithm uses the same approach as Dijkstra's except that it uses accumulated cost of edge plus the Euclidean distance from current node to the destination. This value is used to decide the position of a node in the min heap. The one with smallest value will beselected and removed from the heap. In the implementation, this value only affects the searching order. It doesn't modify the edge weights and accumulated distance. The accumulated distance is updated as the same way as Dijkstra when a node relax. Therefore, this algorithm is same as Dijkstra and it is correct.

IMPLEMENTATION

- Implementation: This program is developed under MATLAB environment. The two algorithms was implemented and visually demonstrated.

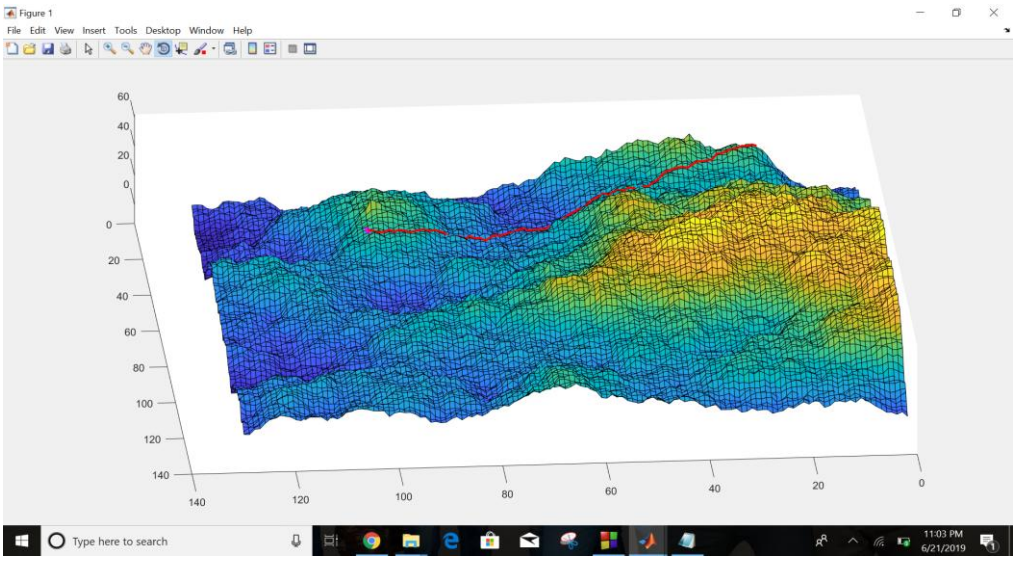


Figure 1
File Edit View Insert Tools Desktop Window Help
60
40
20
0
0
20
40
60
80
100
120
140
140
120
100
80
60
40
20
0
Type here to search
11:03 PM
6/21/2019

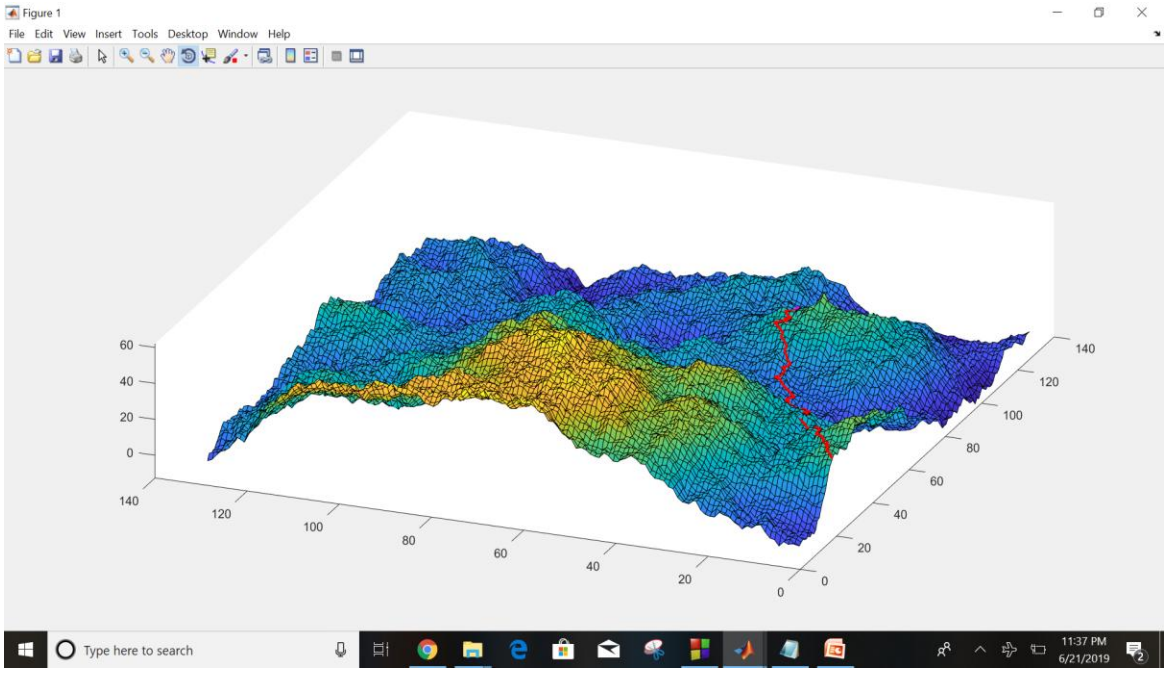
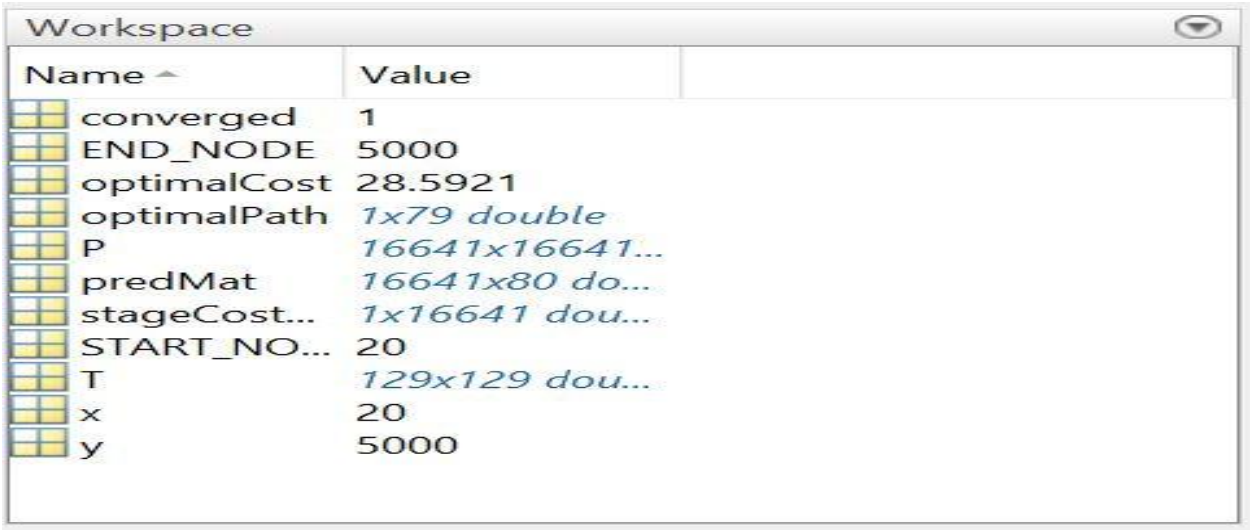


Figure 1
File Edit View Insert Tools Desktop Window Help
60
40
20
0
140
120
100
80
60
40
20
0
140
120
100
80
60
40
20
0
Type here to search
11:37 PM
6/21/2019



Advantages&Disadvantages:

A is faster as compare to Dijkstra's algorithm because it uses Best First Search whereas Dijkstra's uses Greedy Best First Search.*
Dijkstra's is Simple as compare to A.*
The major disadvantage of Dijkstra's algorithm is the fact that it does a blind search there by consuming a lot of time waste of necessary resources.
Another disadvantage is that it cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path.
Dijkstra's algorithm has an order of n2 so it is efficient enough to use for relatively large problems.
The major disadvantage of the algorithm is the fact that it does a blind search there by consuming a lot of time waste of necessary resources

Drawbacks of A *

- The main drawback of A* algorithm and indeed of any best-first searchis its memory requirement. Since at least the entire open list

must be sa ved,A* algorithm is severely space-limited in practice, and is no more practical thanbest-first search algorithm on current machines. For example, while it can berun successfully on the eight puzzle, it exhausts available memory in a matterof minutes on the fifteen puzzle.

Improvements

A* is a breadth first algorithm and as such consumes huge memory tokeep the data of current proceeding nodes. The search can be more efficient ifthe machine searches not just for the path from the source to the target, butalso in parallel for the path from the target to the source (the answer is foundwhen these two searches meet at some point)

CONCLUSION

The A* algorithm can achieve better running time by using Euclidean heuristic function although it theoretical time complexity is still the same as Dijkstra's. It can also guarantee to find the shortest path. The restricted algorithm can find the optimal path within linear time but the restricted area has to be carefully selected. The selection actually depends on the graph itself. The smaller selected area can get less search time but the tradeoff is that it may not find the shortest path or, it may not find any path. This algorithm can be used in a way that allowing search again by increasing the factor if the first search fails.