

ANALYSIS OF SHORTEST PATH ALGORITHMS

by

ANDREW JOHN	18BEC1278
ARJUN SHARMA	18BEC1117

A project report submitted to

Dr. Mirza Galib Anwarul Husain Baig

SCHOOL OF ELECTRONICS ENGINEERING

in partial fulfillment of the requirements for the course of

CSE2003 – Data Structures and Algorithms

in

B.Tech. ELECTRONICS AND COMMUNICATIONS ENGINEERING



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Vandalur – Kelambakkam Road

Chennai – 600127

June 2021

BONAFIDE CERTIFICATE

Certified that this project report entitled “**ANALYSIS OF SHORTEST PATH ALGORITHMS**” is a bonafide work of **ANDREW JOHN - 18BEC1278, and ARJUN SHARMA - 18BEC1117** who carried out the Project work under my supervision and guidance for **CSE2003 – Data Structures and Algorithms**.

Dr. Mirza Galib Anwarul Husain Baig

Associate Professor

School of Electronics Engineering (SENSE),

VIT University, Chennai

Chennai – 600 127.

ABSTRACT

In our day-to-day life, traveling from point A to point B is an essential part of our livelihood and we always prefer it to be less time-consuming. Hence, taking a route that is less time-consuming requires the path we take to be the shortest path and the most efficient path.

Thus, we always take the help of GPS or nowadays the famous Google Maps for finding the less time-consuming and shortest path from one point to another. Dijkstra's Algorithm, taken from its inventor's name, Edsger Dijkstra, one of the pioneering founders of modern computing, is the algorithm behind Google Maps and it is fascinating how it gives us various paths from one point to another depending upon the vehicle we use and the obstacles we will be facing such as traffic, diversions in roads, constructions sites, roadblocks, etc., and gives us various shortest routes taking into account all the things given above.

Therefore, in this project we have proposed an effective method to find the shortest distance between two points (starting point and stop or destination point) that's necessary and the project is aimed at doing this, considering roads to be laid in a grid-wise arrangement. The algorithm also involves the automatic generation of obstacles around which the nodes are traversed and thus plots various different shortest paths. The user is allowed to enter the source point and destination point from a matrix table through which the routing is done and the shortest path between the two points is graphically represented in a grid. The shortest path plotted is highlighted with a different colour for ease of visualization. The proposed program can be applied in real life for route optimizations.

Finally, finding the shortest path in a large-scale network analysis between any two nodes is a tough but very significant task. The shortest path can help us to analyse the information spreading performance and research the latent relationship in various applications, and so on. The applications of the shortest path algorithm are enormous ranging from, networking or telecommunications, gaming, operations research including plant and facility layout, robotic path planning, transportation & road networks, VLSI design for semiconductors, social media, and dating applications.

ACKNOWLEDGEMENT

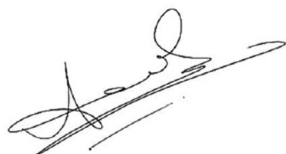
We wish to express our sincere thanks and deep sense of gratitude to our project guide, **Dr. Mirza Galib Anwarul Husain Baig**, Associate Professor, School of Electronics Engineering, for his consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work.

We are extremely grateful to **Dr. Sivasubramanian. A**, Dean of School of Electronics Engineering, VIT Chennai, for extending the facilities of the School towards our project and for his unstinting support.

We express our thanks to our Head of the Department **Dr. Vetrivelan. P** for his support throughout the course of this project.

We also take this opportunity to thank all the faculty of the School for their support and their wisdom imparted to us throughout the course.

We thank our parents, family, and friends for bearing with us throughout the course of our project and for the opportunity they provided us in undergoing this course in such a prestigious institution.



ANDREW JOHN



ARJUN SHARMA

TABLE OF CONTENTS

S. No.	Contents	Page No.
1	Abstract	3
2	Introduction	6
3	Problem Statement	9
4	Literature Survey	10
5	Approach Taken	12
6	Our Contribution	14
7	Results and Applications	24
8	Conclusion and Future Scope	26
9	References	27

INTRODUCTION

Routing is the process of finding the best path between two or more locations with a fixed order in a path or network given. The criterion according to which a path is the best can vary. The user may be looking for the shortest path (by distance), the fastest (by travel time), but also the most scenic or the safest path.

The shortest path problem can be defined for graphs whether undirected, directed, or mixed. It is defined here for undirected graphs; for directed graphs, the definition of path requires that consecutive vertices be connected by an appropriate directed edge.

In this project, we analyse Dijkstra's algorithm, A* Star Algorithm, and Greedy Algorithm. Dijkstra's algorithm solves the single-source shortest path problem with non-negative edge weight. A* search algorithm solves for single-pair shortest path using heuristics to try to speed up the search. The greedy algorithm is an algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.

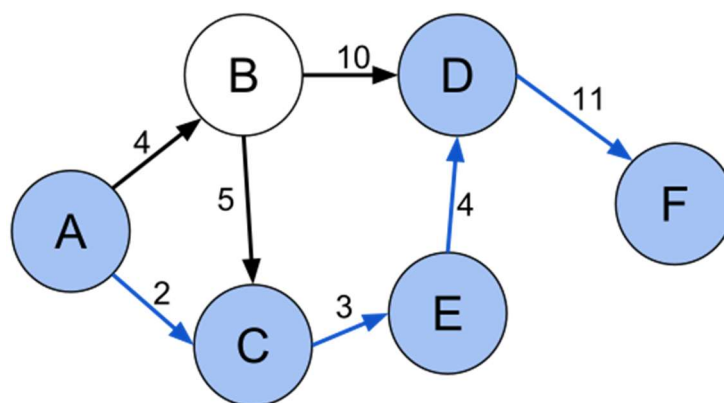


Fig-1: Shortest path (A, C, E, D, F) between vertices A and F in the weighted directed graph

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single

node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

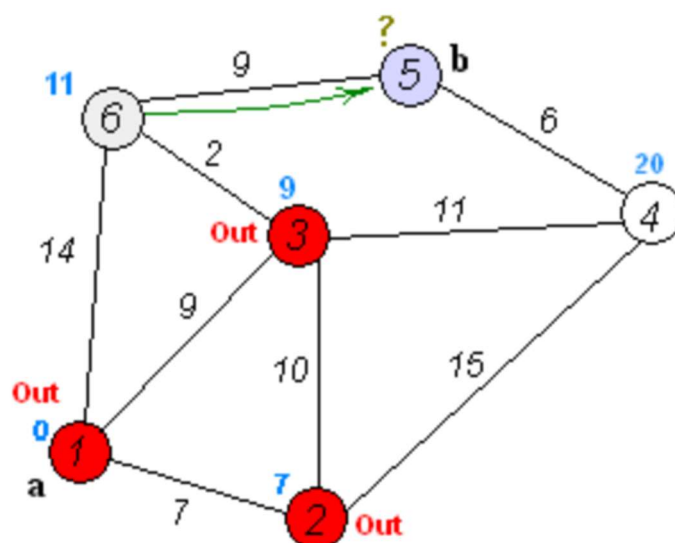


Fig-2: Dijkstra's algorithm to find the shortest path between a and b. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

A* is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. Thus, in practical travel-routing systems, it is generally outperformed by algorithms that can pre-process the graph to attain better performance, as well as memory-bounded approaches. Peter Hart, Nils Nilsson, and Bertram Raphael of Stanford Research Institute first published the algorithm in 1968. It can be seen as an extension of Dijkstra's algorithm. A* achieves better performance by using heuristics to guide its search.

What sets A* apart from a greedy best-first search algorithm is that it takes the cost/distance already traveled into account. Some common variants of Dijkstra's algorithm can be viewed as a special case of A* where the heuristic, $h(n) = 0$ for all nodes; in turn, both Dijkstra and A* are special cases of dynamic programming. A* itself is a special case of a generalization of branch and bound.

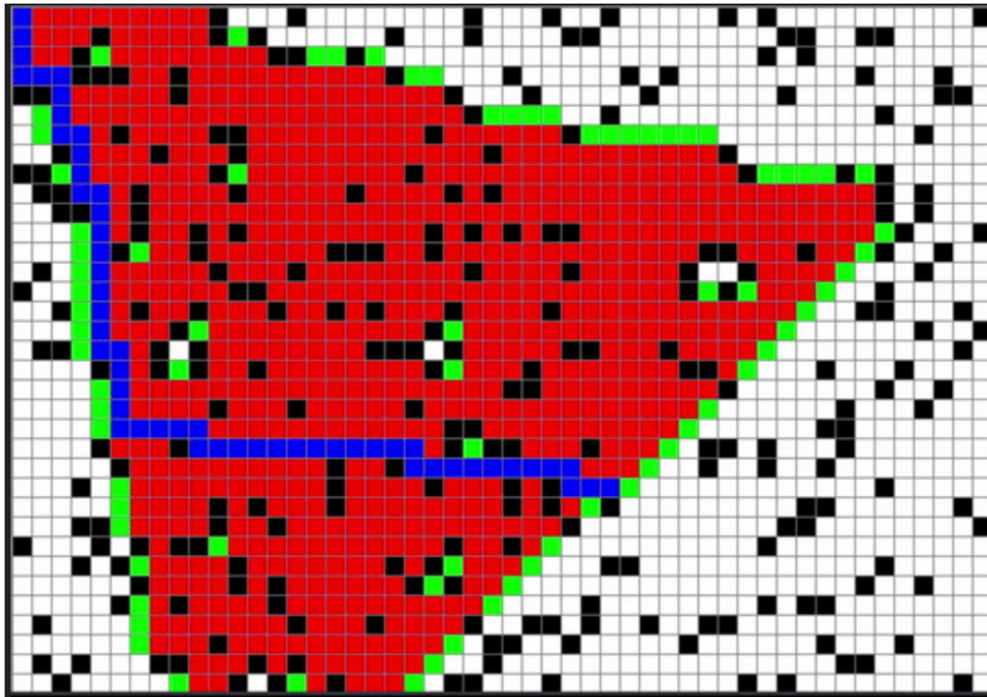


Fig-3: A pathfinding algorithm navigating around a randomly-generated maze*

A greedy algorithm is an algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless, a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

In general, greedy algorithms have five components: (1) A candidate set, from which a solution is created, (2) A selection function, which chooses the best candidate to be added to the solution (3) A feasibility function, that is used to determine if a candidate can be used to contribute to a solution, (4) An objective function, which assigns a value to a solution, or a partial solution, and (5) A solution function, which will indicate when we have discovered a complete solution. Greedy algorithms typically (but not always) fail to find the globally optimal solution because they usually do not operate exhaustively on all the data.

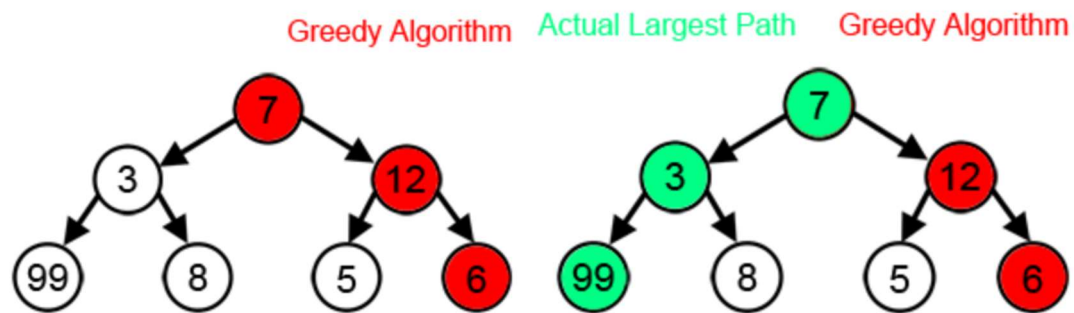


Fig-4: With a goal of reaching the largest sum, at each step, the greedy algorithm will choose what appears to be the optimal immediate choice, so it will choose 12 instead of 3 at the second step, and will not reach the best solution, which contains 99.

PROBLEM STATEMENT

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. The problem of finding the shortest path between two intersections on a road map may be modelled as a special case of the shortest path problem in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.

So, here we wanted to simplify major aspects of solving the problems in the shortest path algorithms with our novel approach and thus it can be implemented in various applications.

LITERATURE SURVEY

“Wang Shu-Xi (2012). The Improved Dijkstra's Shortest Path Algorithm and Its Application. *Procedia Engineering*, 29, 1186-1190” - Focuses on addressing the shortcoming of the Dijkstra Algorithm. The shortest path problem exists in a variety of areas. A well-known shortest path algorithm is Dijkstra's, also called "label algorithm". Experiment results have shown that the "label algorithm" has the following issues:

1. Its existing mechanism is effective to undigraph but ineffective to digraph, or even gets into an infinite loop;
2. It hasn't addressed the problem of adjacent vertices in the shortest path;
3. It hasn't considered the possibility that many vertices may obtain the "p-label" simultaneously.

“Sven Peyer, Dieter Rautenbach, & Jens Vygen (2009). A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing. *Journal of Discrete Algorithms*, 7(4), 377-390.” Illustrates the application of a generalized Dijkstra's algorithm for finding shortest paths in digraphs with non-negative integral edge lengths.

As an application, the VLSI routing problem is considered, where one needs to find millions of shortest paths in partial grid graphs with billions of vertices. The algorithm is applied twice, once in a coarse abstraction (where the labeled subgraphs are rectangles), and once in a detailed model (where the labeled subgraphs are intervals). Using the result of the first algorithm to speed up the second one via goal-oriented techniques leads to considerably reduced running time. This can be illustrated with a state-of-the-art routing tool on leading-edge industrial chips.

“H. Azis, R. d. Mallongi, D. Lantara, and Y. Salim, "Comparison of Floyd-Warshall Algorithm and Greedy Algorithm in Determining the Shortest Route," 2018 2nd East Indonesia Conference on Computer and Information Technology” - In this study, the method used to determine the shortest route is the conventional method and the Heuristic method.

These two methods will be compared to find out methods which can provide the best result. For conventional methods, the Floyd-Warshall algorithm is used whereas for the Heuristic method, the greedy algorithm is employed. The Floyd-

Warshall algorithm takes into account all possible routes so that there are some routes displayed while the greedy algorithm checks every node that is passed to select the shortest route (Local Optimum) so that the time needed in searching is faster. Based on the conducted testing, the final result obtained is the FloydWarshall algorithm provides a better solution. This result indicated that a longer time is required because it takes the distance to all points into account.

“Kairanbay, Magzhan & Mat Jani, Hajar. (2013). A Review and Evaluations of Shortest Path Algorithms. International Journal of Scientific & Technology Research.” - This paper’s main objective is to evaluate the Dijkstra’s Algorithm, Floyd-Warshall Algorithm, Bellman-Ford Algorithm, and Genetic Algorithm (GA) in solving the shortest path problem.

A short review is performed on the various types of shortest path algorithms. Further explanations and implementations of the algorithms are illustrated in graphical forms to show how each of the algorithms works. A framework of the GA for finding optimal solutions to the shortest path problem is presented. The results of evaluating the Dijkstra’s, Floyd-Warshall and Bellman-Ford algorithms along with their time complexity conclude the paper.

“Kairanbay, Magzhan & Mat Jani, Hajar. (2013). A Review and Evaluations of Shortest Path Algorithms. International Journal of Scientific & Technology Research.” - A* algorithm is a heuristic function based algorithm for proper path planning. It calculates the heuristic function’s value at each node on the work area and involves the checking of too many adjacent nodes for finding the optimal solution with zero probability of collision. Hence, it takes much processing time and decreases the work speed.

The modifications in A* algorithms for reducing the processing time are proposed in this paper. The proposed A* algorithm determines the heuristic function’s value just before the collision phase rather than initially and exhibits a good decrement in processing time with higher speed. This paper involves simulation of robot movement from source to goal. Several cases are considered with proposed A* algorithm which exhibits a maximum 95% reduction in processing time.

“Madkour, Amgad & Aref, Walid & Rehman, Faizan & Rahman, Abdur & Basalamah, Saleh. (2017). A Survey of Shortest-Path Algorithms.” - This paper presents a survey of shortest-path algorithms based on a taxonomy that is introduced in the paper. One dimension of this taxonomy is the various flavors of

the shortest-path problem. There is no one general algorithm that is capable of solving all variants of the shortest-path problem due to the space and time complexities associated with each algorithm.

Other important dimensions of the taxonomy include whether the shortest-path algorithm operates over a static or a dynamic graph, whether the shortest-path algorithm produces exact or approximate answers, and whether the objective of the shortest-path algorithm is to achieve time-dependence or is to only be goal directed. This survey studies and classifies shortest-path algorithms according to the proposed taxonomy. The survey also presents the challenges and proposed solutions associated with each category in the taxonomy.

APPROACH TAKEN

The programming language used in this project is JavaScript and the visualizer is visualized in the markup language, HTML and JavaScript p5 Library

In this project, we are performing a pathfinder for the Shortest path algorithm, and a visualizer for visualizing the route in which the process of displaying the shortest path from the source vertex and destination vertex. An $n \times n$ matrix is formed, and the shortest path between the source and destination is calculated and visualized using the implemented algorithms that are Dijkstra, A* and Greedy Algorithms

Input (as code): Vectors/Position of start and end points, probability of occurrence of obstacles/walls.

Output: Shortest path between source and destination visualized in a grid of $n \times n$ matrix.

The Algorithm used by each Shortest Path are as follows:

Generalised Dijkstra Algorithm for Shortest Path:

Between two vertices of a graph where all edges have non-negative weight. It is based on Dijkstra's algorithm for computing the shortest path repeatedly expanding the closest vertex which has not yet been reached. Since it traverses through all nodes of the graph, it is considered less efficient.

A briefer approach of the algorithm is as follows:

1. Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.
2. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
3. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.
4. For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
5. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
6. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
7. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 4.

Generalised A* Algorithm for Shortest Path:

A* algorithm picks the node according to a value -'f' which is a parameter equal to the sum of two other parameters - 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and processes that node/cell. We define 'g' and 'h' as simply as possible below g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there. h = the estimated movement cost to move from that given square on the grid to the final destination.

A briefer approach of the algorithm is as follows:

1. The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED. OPEN contains those nodes that have been evaluated by the heuristic function and CLOSED contains those nodes that have already been visited.
2. Define a list OPEN. Initially, OPEN consists solely of a single node, the start node S.
3. If the list is empty, return failure and exit.
4. Remove node n with the smallest value of $f(n)$ from OPEN and move it to list CLOSED. If node n is a goal state, return success and exit.
5. Expand node n.
6. If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S. Otherwise, go to Step-07.
7. For each successor node, apply the evaluation function f to the node. If the node has not been in either list, add it to OPEN.
8. Go back to Step-03.

Greedy Algorithm:

The Algorithm selects the next best node whose distance to destination is minimal. Unlike Dijkstra, it only traverses towards the direction of the end vertex. Hence it is more efficient than Dijkstra, but may not provide the most efficient path.

OUR CONTRIBUTION

Functions Used:

- function setup()
- function removeElement()
- function shortestPath()
- function heuristic()
- function draw()
- function drawfinal()

❑ Pseudocode for Dijkstra Shortest Path:

The programming language used for visualisation purposes is JavaScript
Initialise with frame rate of 60 frames per second (for visualisation purposes).

initialise rows = 25

initialise cols = 25

grid = new Array(rows)

int w, h;

let visited = []

let lookingAt = [];

function setup() [*This function also takes visualisation steps into account*]

w = width/cols

h = height/rows

begin loop

for(i = 0; i < rows; i++)

grid[i]=new Array(cols)

end loop

begin loop

for(i = 0; i < rows; i++)

for(j = 0; j < cols; j++)

grid[i][j] = new cell(i,j)

end loop

start = grid[5][5];

end = grid[cols - 5][rows - 10];

start distance = 0

start previous = start

current = start

lookingAt.push(start)

end.wall = start.wall = false;

end function

function removeElement(array, elemt)

begin loop

for(i=0; i<array length;i++)

if arr[i] == element

remove element

end loop

end function

function shortestPath(start, end)

current = prev.end

path=[]

begin loop

while (current.prev!=current)

path.unshift(current)

current = current.prev

end loop

begin loop

for(i=0;i<path length; i++)

path[i].show(255,255,0) [choose yellow color]

end loop

end function

function draw()

begin loop

for(i = 0; i < rows; i++)

begin loop

for(let j = 0; j < cols; j++)

if(grid[i][j].wall)

grid[i][j].show(0, 0, 0) [Black colour to denote wall]

else

grid[i][j].show(255, 255, 255) [Colour code for regions that aren't walls for visualisation]

end loop

end loop

begin loop

for(i = 0; i < visited.length; i++)

visited[i].show(242, 141, 0)

end loop

begin loop

for(i = 0; i < visited.length; i++)

visited[i].show(242, 141, 0) [Showing visited path using dark yellow]

end loop

begin loop

for(i = 0; i < lookingAt.length; i++)

lookingAt[i].show(18,228,248)

end loop

start.show(0, 255, 0) [Show starting point in green colour]

end.show(255, 0, 0) [Show ending point in red colour]

if(current == end)

shortestPath(start, end)

visited.push(current);

removeElement(lookingAt, current)


```
let neighbours = current.getNeighbours()
```

```
begin loop
```

```
for(i = 0; i < neighbours.length; i++)
```

```
  neighbour = neighbours[i]
```

```
  if(visited.includes(neighbour))
```

```
    continue
```

```
  consider node only if its not a wall and if it's not being looked at already
```

```
  if(!lookingAt.includes(neighbour) && !neighbour.wall){
```

```
    neighbour.dist = current.dist + 1;
```

```
    neighbour.prev = current;
```

```
    lookingAt.push(neighbour);
```

```
tentativeDist = current.distance + 1
```

```
if(!lookingAt.includes(neighbour) && !neighbour.wall) [consider the node if its not a wall or not looked at previously]
```

```
  lookingAt.push(neighbour)
```

```
if(tentativeDist < neighbour.distance)
```

```
  neighbour.distance = tentativeDist;
```

```
  neighbour.prev = current;
```

```
end loop
```

```
variable minnode = 0;
```

```
variable mindist = infinity
```

```
if(lookingAt.length == 0) [The shortest path has been found]
```

```
  return;
```

```
else
```

```
begin loop
```

```
if(lookingAt[i].total < lookingAt[minNode].total)
```

```
  minDist = lookingAt[i].dist;
```

```
  minNode = lookingAt[i];
```

```
end loop
```

```
current = minNode;
```

```
if(minDist == Infinity)
```

```
  return;
```

```
end function
```

```
end pseudocode
```

❑ Pseudocode for A* Shortest Path:

The programming language used for visualisation purposes is JavaScript
Initialise with frame rate of 60 frames per second (for visualisation purposes).

```
initialise rows = 25
initialise cols = 25
```

```
grid = new Array(rows)
int w, h;
let visited = []
let lookingAt = [];
```

function setup() [This function also takes visualisation steps into account]

```
w = width/cols
h = height/rows
```

```
begin loop
for( i = 0; i < rows; i++)
grid[i]=new Array(cols)
end loop
```

```
begin loop
for( i = 0; i < rows; i++)
for(j = 0; j < cols; j++)
grid[i][j]= new cell(i,j)
end loop
```

```
start = grid[5][5];
end = grid[cols - 5][rows - 10];
```

```
start distance = 0
start previous = start
current = start
lookingAt.push(start)
```

```
end.wall = start.wall = false;
```

```
end function
```

function removeElement(array, element)

```
begin loop
for(i=0; i<array length;i++)
if arr[i] == element
remove element
end loop
end function
```

```

function shortestPath(start, end)
  current = prev.end
  path=[]
  begin loop
  while (current.prev!=current)
  path.unshift(current)
  current = current.prev
  end loop
  begin loop
  for(i=0;i<path.length; i++)
  path[i].show(255,255,0) [choose yellow color]
  end loop
end function

```

```

function heuristic(cell,end)
  return abs(cell.i - end.i) + abs(cell.j - end.j)
end function

```

```

function draw()
  begin loop
  for(i = 0; i < rows; i++)
  begin loop
  for(let j = 0; j < cols; j++)
  if(grid[i][j].wall)
    grid[i][j].show(0, 0, 0) [Black colour to denote wall]
  else
    grid[i][j].show(200, 200, 200) [Colour code for regions that aren't walls for visualisation]
  end loop
  end loop
end loop

```

```

  begin loop
  for(i = 0; i < visited.length; i++)
  visited[i].show(242, 141, 0)
  end loop

```

```

  begin loop
  for(i = 0; i < visited.length; i++)
  visited[i].show(242, 141, 0) [Showing visited path using dark yellow]
  end loop

```

```

  begin loop
  for(i = 0; i < lookingAt.length; i++)
  lookingAt[i].show(18,228,248)
  end loop

```

```

  start.show(0, 255, 0) [Show starting point in green colour]
  end.show(255, 0, 0) [Show ending point in red colour]

```

```

  if(current == end)
    shortestPath(start, end)
  end if

```

return

visited.push(current);
removeElement(lookingAt, current)

let neighbours = current.getNeighbours()

begin loop
for(i = 0; i < neighbours.length; i++)
neighbour = neighbours[i]
if(visited.includes(neighbour))
 continue

tentativeDist = current.distance + 1
if(!lookingAt.includes(neighbour) && !neighbour.wall) [consider the node if its not a wall or
not looked at previously]
 lookingAt.push(neighbour)

if(tentativeDist < neighbour.distance)
 neighbour.distance = tentativeDist;
 neighbour.prev = current;
 neighbour.total = neighbour.distance + heuristic(neighbour, end)

end loop

variable minnode = 0;
variable mindist = infinity

if(lookingAt.length == 0) [The shortest path has been found]
 return;

else

begin loop
if(lookingAt[i].total < lookingAt[minNode].total)
 minNode = i
 minDist = lookingAt[i].total

end loop
current = lookingAt[minNode]

if(minDist == Infinity)
 return;

end function

end pseudocode

❑ Pseudocode for Greedy Shortest Path:

The programming language used for visualisation purposes is JavaScript
Initialise with frame rate of 30 frames per second (for visualisation purposes).

initialise rows = 25

initialise cols = 25

grid = new Array(rows)

int w, h;

let visited = []

let lookingAt = [];

function setup() [*This function also takes visualisation steps into account*]

w = width/cols

h = height/rows

background(255,0,200);

frameRate(30);

begin loop

for(i = 0; i < rows; i++)

grid[i]=new Array(cols)

end loop

begin loop

for(i = 0; i < rows; i++)

for(j = 0; j < cols; j++)

grid[i][j] = new cell(i,j)

end loop

start = grid[5][5];

end = grid[cols - 5][rows - 10];

start distance = 0

start previous = start

current = start

lookingAt.push(start)

end.wall = start.wall = false;

end function

function removeElement(array, elemt)

begin loop

for(i=0; i<array length;i++)

if arr[i] == element

remove element

end loop

end function

```
function shortestPath(start, end)
  current = prev.end
  path=[]
  begin loop
  while (current.prev!=current)
  path.unshift(current)
  current = current.prev
  end loop
  begin loop
  for(let i = 0; i < path.length; i++){
    drawfinal(path[i]);
  }
  end loop
end function
```

```
function heuristic(cell,end)
  return abs(cell.i - end.i) + abs(cell.j - end.j)
end function
```

```
function drawfinal(path){
  path.show(255,255,0);
}
```

```
function draw()
  begin loop
  for(i = 0; i < rows; i++)
  begin loop
  for(let j = 0; j < cols; j++)
  if(grid[i][j].wall)
    grid[i][j].show(0, 0, 0) [Black colour to denote wall]
  else
    grid[i][j].show(200, 200, 200) [Colour code for regions that aren't walls for visualisation]
  end loop
  end loop
```

```
  begin loop
  for(i = 0; i < visited.length; i++)
  visited[i].show(242, 141, 0)
  end loop
```

```
  begin loop
  for(i = 0; i < visited.length; i++)
  visited[i].show(242, 141, 0) [Showing visited path using dark yellow]
  end loop
```

```
  begin loop
  for(i = 0; i < lookingAt.length; i++)
  lookingAt[i].show(18,228,248)
  end loop
```

start.show(0, 255, 0) [Show starting point in green colour]
end.show(255, 0, 0) [Show ending point in red colour]

if(current == end)
 shortestPath(start, end)
return

visited.push(current);
removeElement(lookingAt, current)
let neighbours = current.getNeighbours()

begin loop
for(i = 0; i < neighbours.length; i++)
 neighbour = neighbours[i]
 if(visited.includes(neighbour))
 continue

tentativeDist = current.distance + 1
if(!lookingAt.includes(neighbour) && !neighbour.wall) [consider the node if its not a wall or not looked at previously]
 lookingAt.push(neighbour)

if(tentativeDist < neighbour.distance)
 neighbour.distance = tentativeDist;
 neighbour.prev = current;
 neighbour.total = heuristic(neighbour, end)

end loop

variable minnode = 0;
variable mindist = infinity

if(lookingAt.length == 0) [The shortest path has been found]
 return;

else

begin loop
if(lookingAt[i].total < lookingAt[minNode].total)
 minNode = i
 minDist = lookingAt[i].total

end loop
current = lookingAt[minNode]

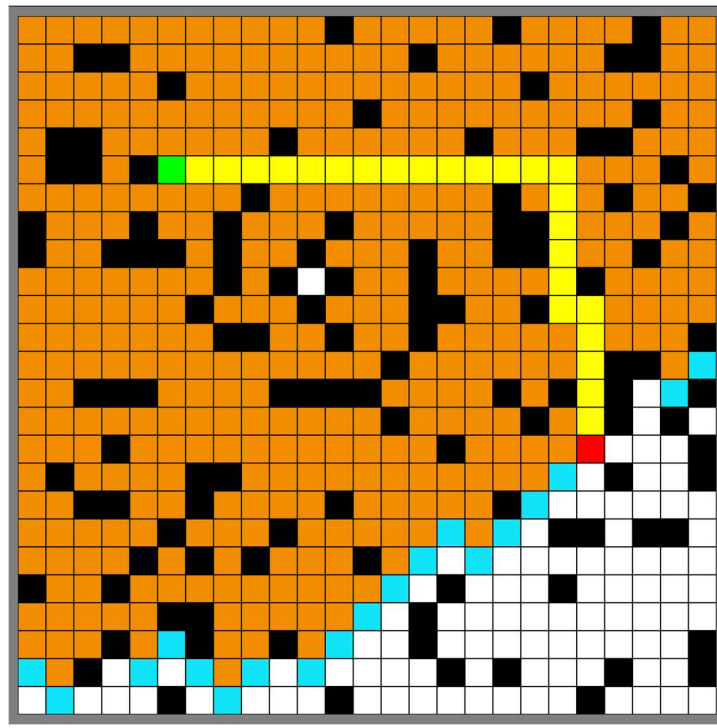
if(minDist == Infinity)
 return;

end function

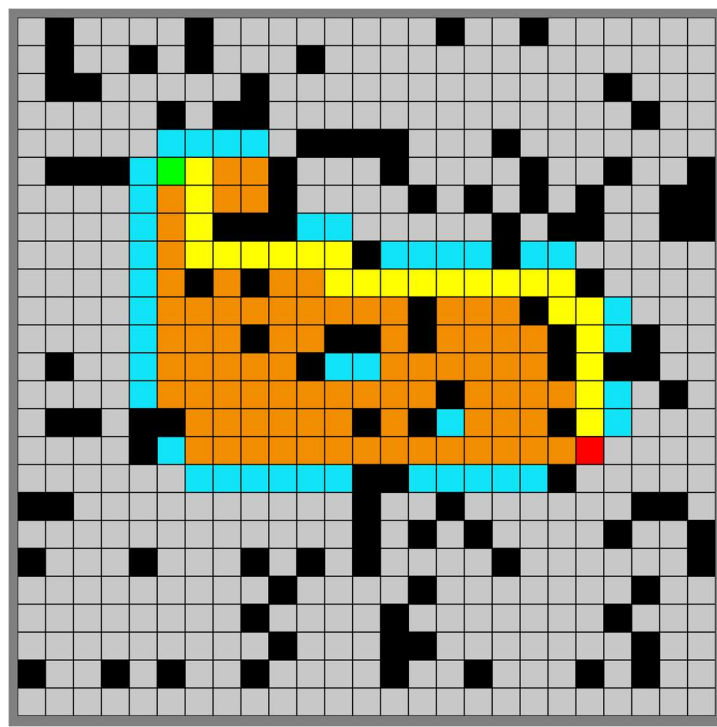
end pseudocode

RESULTS AND APPLICATIONS

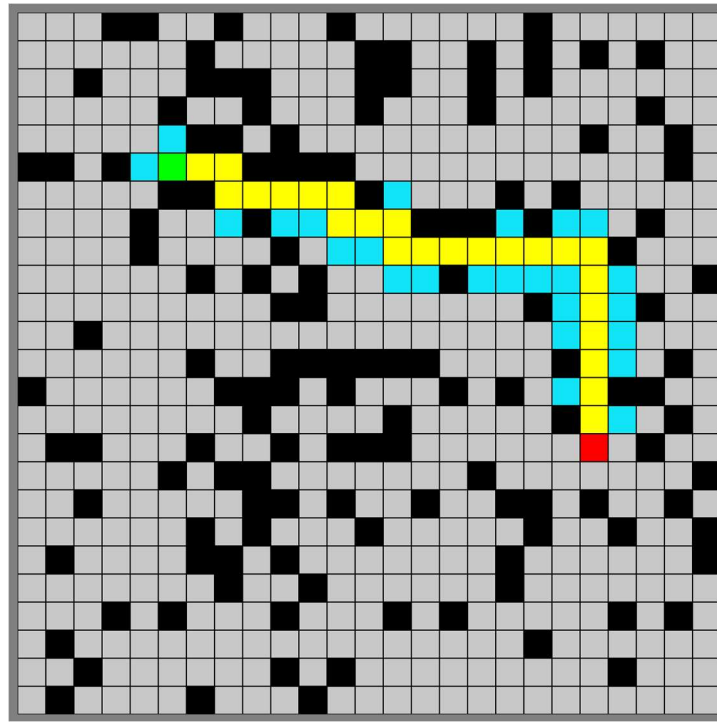
➤ Results for Dijkstra's Shortest Path Algorithm:



➤ Results for A* Shortest Path Algorithm:



➤ **Results for Greedy Shortest Path Algorithm:**



❑ **Applications of Shortest Path Algorithms:**

Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on web mapping websites like MapQuest or Google Maps. For this application fast specialized algorithms are available

If one represents a nondeterministic abstract machine as a graph where vertices describe states and edges describe possible transitions, shortest path algorithms can be used to find an optimal sequence of choices to reach a certain goal state, or to establish lower bounds on the time needed to reach a given state. For example, if the vertices represent the states of a puzzle like a Rubik's Cube and each directed edge corresponds to a single move or turn, shortest path algorithms can be used to find a solution that uses the minimum possible number of moves.

❑ Real-Life Applications of the Discussed Algorithms

Dijkstra's Algorithm

1. Digital Mapping Services in Google Maps
2. Social Networking Applications
3. Telephone Network
4. IP routing to find Open Shortest Path First
5. Flighting Agenda
6. Designate file server
7. Robotic Path

A* Algorithm

1. Originally designed as a general graph traversal algorithm.
2. Pathfinding in Video Games.
3. Using stochastic grammars in NLP.
4. Informational search with online learning.

Greedy Algorithm

1. Finding an optimal solution.
2. Finding close to the optimal solution for NP-Hard problems like TSP.

CONCLUSIONS AND FUTURE SCOPE

The path finder and visualizer programs for Analysis of Shortest Path Algorithms responds correctly to all sources and destinations as per their definitions and provides the shortest path between the former and latter.

The highlighted route helps distinguish itself from the grid and makes it easier for the user to comprehend.

1. Increasing the working of the programme to any size of matrix is the first future scope and extends analysis for the same if achieved.
2. Multiple destination points can be added for a single traversal and is the second future scope.

REFERENCES

- [1] Wang Shu-Xi (2012). The Improved Dijkstra's Shortest Path Algorithm and Its Application. *Procedia Engineering*, 29, 1186-1190.
- [2] Sven Peyer, Dieter Rautenbach, & Jens Vygen (2009). A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing. *Journal of Discrete Algorithms*, 7(4), 377-390.
- [3] H. Azis, R. d. Mallongi, D. Lantara, and Y. Salim, "Comparison of Floyd-Warshall Algorithm and Greedy Algorithm in Determining the Shortest Route," 2018 2nd East Indonesia Conference on Computer and Information Technology (EIconCIT), Makassar, Indonesia, 2018, pp. 294-298, DOI: 10.1109/EIconCIT.2018.8878582.
- [4] Kairanbay, Magzhan & Mat Jani, Hajar. (2013). A Review and Evaluations of Shortest Path Algorithms. *International Journal of Scientific & Technology Research*. 2. 99-104.
- [5] Akshay Kumar Guruji, Himansh Agarwal, & D.K. Parsediya (2016). Time-efficient A* Algorithm for Robot Path Planning. *Procedia Technology*, 23, 144-149.
- [6] Madkour, Amgad & Aref, Walid & Rehman, Faizan & Rahman, Abdur & Basalamah, Saleh. (2017). A Survey of Shortest-Path Algorithms.