Optimisation of Dijkstra's Algorithm Using A* Search Algorithm

Harshit Soni

B.Tech in Computer Science and Engineering (SCOPE)

Vellore Institute of Technology , Vellore – 632014 , India

harshit.soni2019@vitstudent.ac.in

Abstract – This Documentation Contains the approach to how to optimise the Dijkstra's Algorithm. Dijkstra's Algorithm is used in many areas including Google Maps, Various Social networking Sites, Online Delivery and in a lot of fields. Due to Advancement in Technology as there are more & more users there will be more data and the time to evaluate the shortest path or distance using Dijkstra's Algorithm is more So We need a better approach than the Dijkstra's Algorithm. Here comes our A* Search Algorithm and the objective is to approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances. So we will be using Euclidean distance & Heuristics to find the shortest distance to destination point.

Keywords - A* algorithm, Dijkstra's Algorithm, Heuristic Function, Priority Queue, Adjacency Matrix, Manhattan Distance, Diagonal Distance , Euclidean Distance

I.INTRODUCTION

Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest path between nodes in a graph, which may represent, for example, road networks 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. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

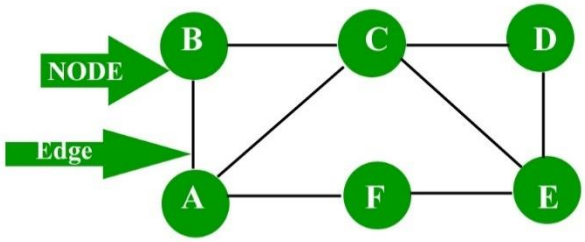
If the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithm is network routing protocol, most notably IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF). It is also employed as a Subroutine.

II. Background

GRAPHS

A graph is a data structure that is defined by two components :

1. A node or a vertex.
2. An edge E or ordered pair is a connection between two nodes u,v that is identified by unique pair(u,v). The pair (u,v) is ordered because (u,v) is not same as (v,u) in case of directed graph.The edge may have a weight or is set to one in case of unweighted graph.

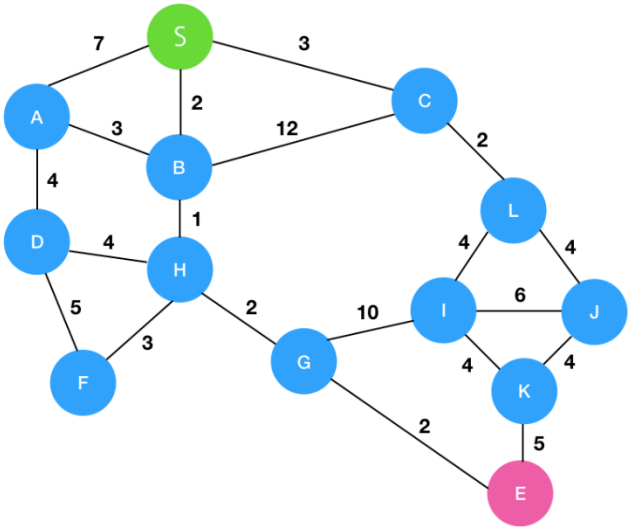


I. Dijkstra's Algorithm

Suppose you would like to find the shortest path between two city on a map: a starting point and a destination. Dijkstra's algorithm initially marks the distance (from the starting point) to every other intersection on the map with infinity. This is done not to imply that there is an infinite distance, but to note that those intersections have not been visited yet. Some variants of this method leave the intersections' distances unlabeled. Now select the current intersection at each iteration. For the first iteration, the current intersection will be the starting point, and the

distance to it (the intersection's label) will be zero. For subsequent iterations (after the first), the current intersection will be a closest unvisited intersection to the starting point (this will be easy to find).

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection and then relabeling the unvisited intersection with this value (the sum) if it is less than the unvisited intersection's current value. In effect, the intersection is relabeled if the path to it through the current intersection is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabeled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighbouring ineraction, mark the current intersection as visited and select an unvisited intersection with minimal distance (from the starting point) – or the lowest label—as the current intersection. Intersections marked as visited are labeled with the shortest path from the starting point to it and will not be revisited or returned to.

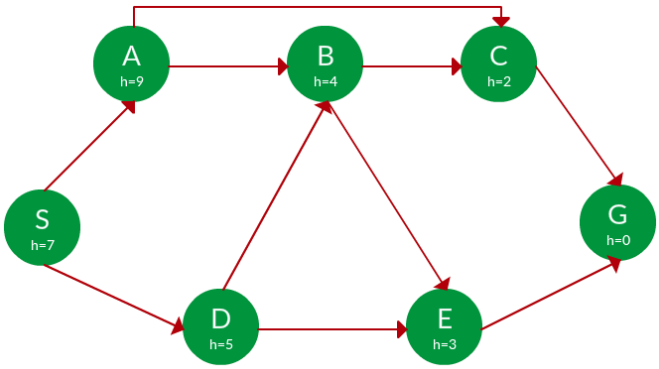


Continue this process of updating the neighbouring intersections with the shortest distances, marking the current intersection as visited, and moving onto a closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited (as is the case with any visited intersection), you have determined the shortest path to it from the starting point and can trace your way back following the arrows in reverse

II. A Search Algorithm*

A* (pronounced "A-star") 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.[1] One major practical drawback is its O(b^d) space complexity, as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, as well as memory-bounded approaches; however, A* is still the best solution in many cases.

A* was created as part of the Shakey project, which had the aim of building a mobile robot that could plan its own actions. Nils Nilsson originally proposed using the Graph Traverser algorithm for Shakey's path planning. Graph Traverser is guided by a heuristic function the estimated distance from node to the goal node: it entirely ignores the distance from the start node to Bertram Raphael suggested using the sum, Peter Hart invented the concepts we now
all admissibility and consistency of heuristic functions. A* was originally designed for finding least-cost paths when the cost of a path is the sum of its edge costs, but it has been shown that A* can be used to find optimal paths for any problem satisfying the conditions of a cost algebra.



The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

As an example, when searching for the shortest route on a map, h(x) might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points.

If the heuristic h satisfies the additional condition $h(x) \leq d(x, y) + h(y)$ for every edge (x, y) of the graph (where d denotes the length of that edge), then h is called monotone, or consistent. With a consistent heuristic, A* is guaranteed to find an optimal path without processing any node more than once and A* is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) = d(x, y) + h(y) − h(x)$.

III Implementation of Dijkstra's Algorithm

I. Using a priority queue

A min-priority queue is an abstract data type that provides 3 basic operations : add_with_priority(), decrease_priority() and extract_min(). As mentioned earlier, using such a data structure can lead to faster computing times than using a basic queue. Notably, Fibonacci heap () offer optimal implementations for those 3 operations. As the algorithm is slightly different, we mention it here, in pseudo-code as well :

Pseudocode for Dijkstra's Algorithm using Priority Queue

1) Initialize distances of all vertices as infinite.

2) Create an empty **p**riority_**q**ueue **pq**. Every item of pq is a pair (weight, vertex). Weight (or distance) is used used as first item of pair as first item is by default used to compare two pairs

3) Insert source vertex into pq and make its distance as 0.

4) While either pq doesn't become empty
a) Extract minimum distance vertex from pq. Let the extracted vertex be u.
b) Loop through all adjacent of u and do following for every vertex v.

// If there is a shorter path to v
// through u.
If dist[v] > dist[u] + weight(u, v)

(i) Update distance of v, i.e., do dist[v] = dist[u] + weight(u, v)
(ii) Insert v into the pq (Even if v is already there)

5) Print distance array dist[] to print all shortest paths.

Time Complexity

. If we take a closer look, we can observe that the statements in inner loop are executed ***O(V+E)*** times (similar to BFS). The inner loop has decreaseKey() operation which takes O(LogV) time. So overall time complexity is O(E+V)*O(LogV) which is ***O((E+V)*LogV) = O(ELogV)***
Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to O(E + VLogV) using Fibonacci Heap. The reason is, Fibonacci Heap takes O(1) time for decrease-key operation while Binary Heap takes O(Logn) time.

II. Using a Adjacency Matrix :
Algorithm:

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While *sptSet* doesn't include all vertices
….**a)** Pick a vertex u which is not there in *sptSet* and has minimum distance value.
….**b)** Include u to *sptSet*.
….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

X	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	5	0	0	0	0	0	5	0	0	0	0	0
B	5	0	5	0	0	0	0	0	2	0	0	0	0
C	0	5	0	5	0	0	0	0	0	2	0	0	0
D	0	0	5	0	5	0	0	0	0	0	0	0	0
E	0	0	0	5	0	5	0	0	0	0	0	0	0
F	0	0	0	0	5	0	5	0	0	0	2	0	0
G	0	0	0	0	0	5	0	5	0	0	0	2	0
H	5	0	0	0	0	0	5	0	0	0	0	0	0
I	0	2	0	0	0	0	0	0	0	0	0	2	1
J	0	0	2	0	0	0	0	0	0	0	2	0	1
K	0	0	0	0	0	2	0	0	0	2	0	0	1
L	0	0	0	0	0	0	2	0	2	0	0	0	1
M	0	0	0	0	0	0	0	0	1	1	1	1	0

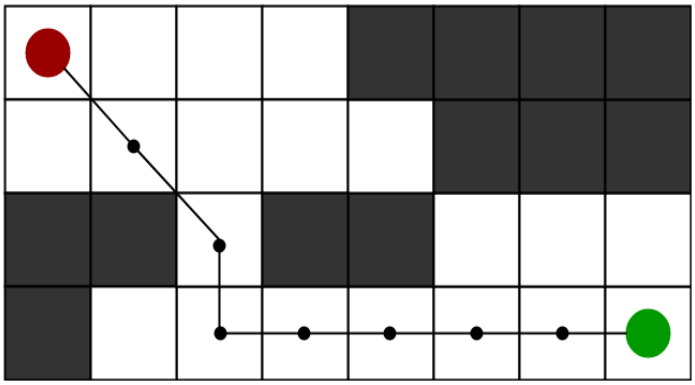
Time Complexity: **O(|V|^2).**
Total vertices: V, Total Edges : E

- O(V) – Iterate through all vertices to add them in SPT
- O(V) – Each time select a vertex with minimum distance.
- So over all complexity: **O(|V|^2).**

IV. A Search Algorithm*

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.
And it is also worth mentioning that many games and

web-based maps use this algorithm to find the shortest path very efficiently (approximation).



Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

What A* Search Algorithm does is that at each step it 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 process 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. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this 'h' which are discussed in the later sections.

Now to understand how A* works, first we need to understand a few terminologies:

- Node (also called State) — All potential position or stops with a unique identification
- Transition — The act of moving between states or nodes.
- Starting Node — Where to start searching
- Goal Node — The target to stop searching.
- Search Space — A collection of nodes, like all board positions of a board game
- Cost — Numerical value (say distance, time, or financial expense) for the path from a node to another node.
- g(n) — this represents the *exact cost* of the path from the starting node to any node n
- h(n) — this represents the heuristic *estimated cost* from node n to the goal node.
- f(n) — lowest cost in the neighboring node n

Each time A* enters a node, it calculates the cost, f(n) (being the neighboring node), to travel to all of the neighboring nodes, and then enters the node with the lowest value of f(n).

These values we calculate using the following formula:

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

I.Heuristics

A) Exact Heuristics –

We can find exact values of h, but that is generally very time consuming.

Below are some of the methods to calculate the exact value of h.

1) Pre-compute the distance between each pair of cells before running the A* Search Algorithm.

2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance

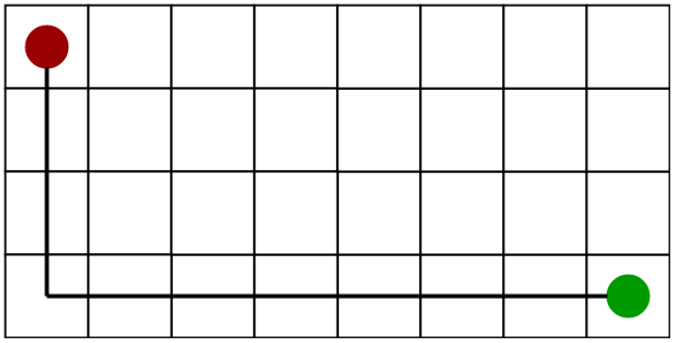
B) Approximation Heuristics –

There are generally three approximation heuristics to calculate h –

1) Manhattan Distance –

- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,
- h = abs (current_cell.x – goal.x) + abs (current_cell.y – goal.y)
- When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

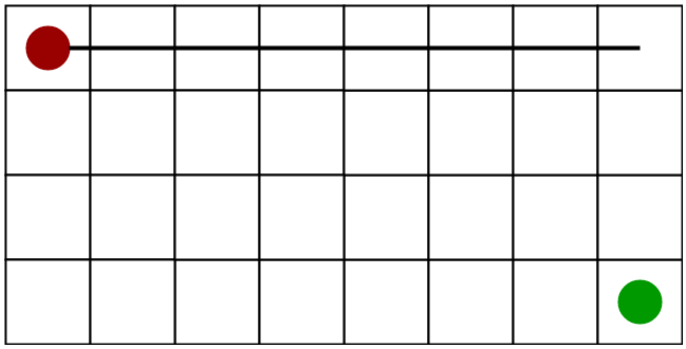
The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



2) Diagonal Distance-

- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,
- h = max { abs(current_cell.x – goal.x), abs(current_cell.y – goal.y) }
- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

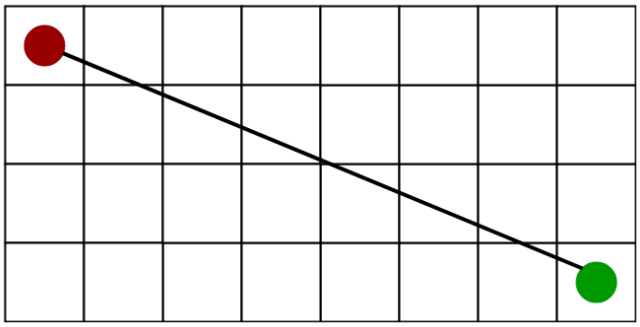
The Diagonal Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



3) Euclidean Distance

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula
- h = sqrt ((current_cell.x – goal.x)2 + (current_cell.y – goal.y)2)
- When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



II.Algorithm

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

// A* Search Algorithm

1. Initialize the open list
2. Initialize the closed list

   put the starting node on the open

   list (you can leave its f at zero)

3. while the open list is not empty
   a) find the node with the least f on the open list, call it "q"

   b) pop q off the open list

   c) generate q's 8 successors and set their parents to q

   d) for each successor
      i) if successor is the goal, stop search
      successor.g = q.g + distance between successor and q
      successor.h = distance from goal to successor (This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)

      successor.f = successor.g + successor.h

      ii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor

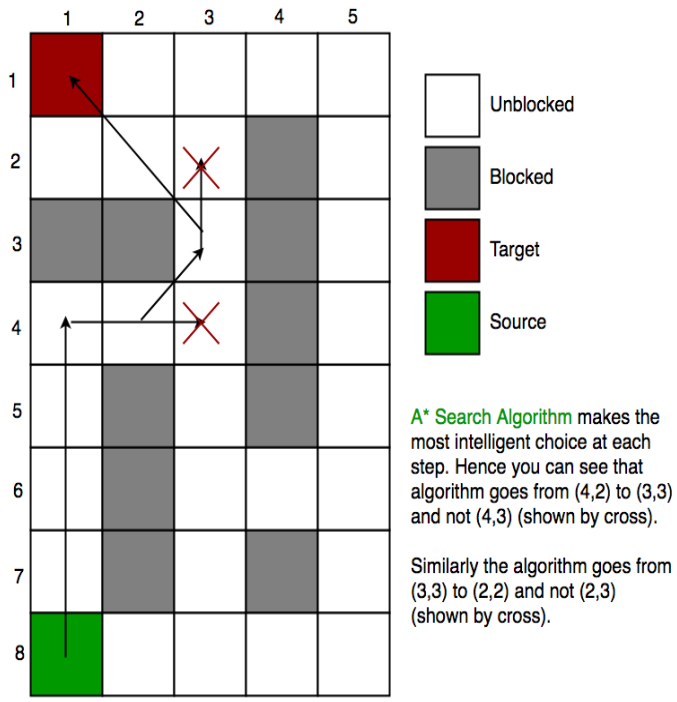
iii) if a node with the same position as
successor is in the CLOSED list which has
a lower f than successor, skip this successor
otherwise, add the node to the open list
end (for loop)

e) push q on the closed list
end (while loop)

So suppose as in the below figure if we want to reach the target cell from the source cell, then the A* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristics.

III.Relation (Similarity and Differences) with other algorithms-

Dijkstra is a special case of A* Search Algorithm, where h = 0 for all nodes.



IV.Limitations

Although being the best pathfinding algorithm around, A* Search Algorithm doesn't produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h

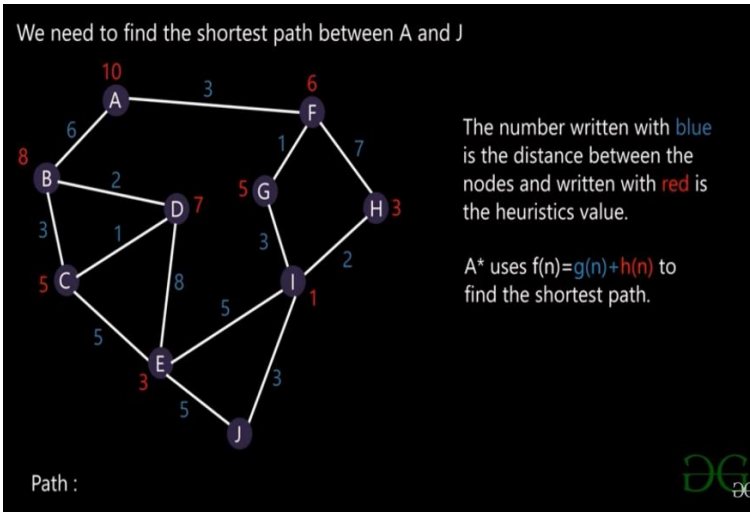
V.Time Complexity

Considering a graph, it may take us to travel all the edge to reach the destination cell from the source cell [For example, consider a graph where source and destination nodes are connected by a series of edges, like – 0(source) –>1 –> 2 –> 3 (target)
So the worse case time complexity is O(E), where E is the number of edges in the graph

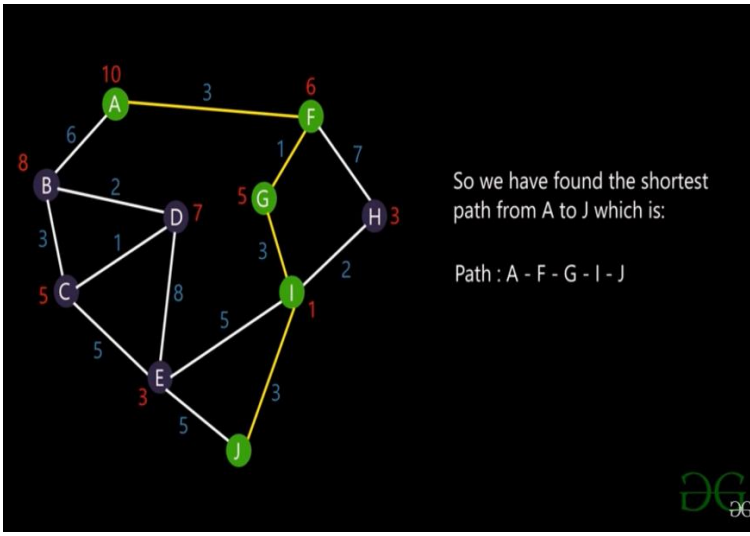
VI. Auxiliary Space

Auxiliary Space In the worse case we can have all the edges inside the open list, so required auxiliary space in worst case is O(V), where V is the total number of vertices.

VII. Diagram To find The Shortest Path Using A Search Algorithm*



Possible Shortest Path-



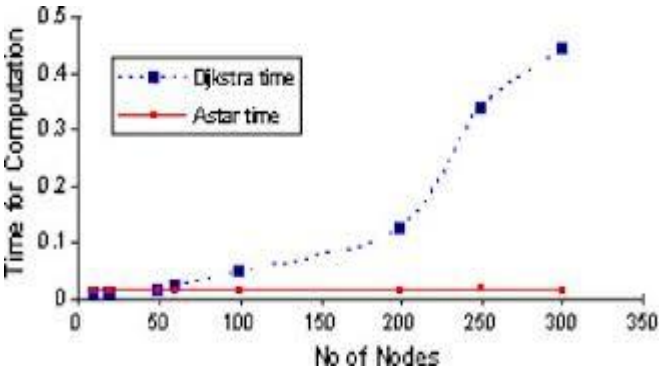
III. Comparison of A* and Dijkstra's

A* is just like **Dijkstra**, the only difference is that A* tries to look for a better path by using a heuristic function which gives priority to nodes that are supposed to be better than others while Dijkstra's just explore all possible paths.

Its optimality depends on the heuristic function used, so yes it can return a non optimal result because of this and at the same time better the heuristic for your specific layout, and better will be the results (and possibly the speed).

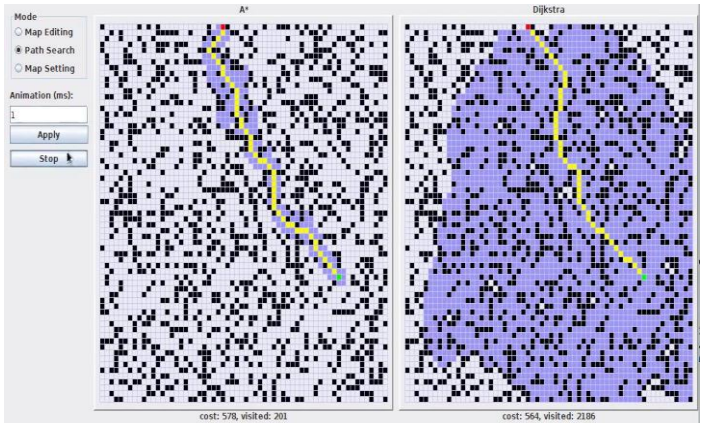
It is meant to be faster than Dijkstra even if it requires more memory and more operations per node since it explores a lot less nodes and the gain is good in any case.

Precomputing the paths could be the only way if you need realtime results and the graph is quite large, but usually you wish to pathfind the route less frequently



A* takes constant time whereas

Dijkstra's algorithm takes 5 times more time as compared to A star algorithm



Here in above example in this grid the black dots represent the area is blocked so we cannot go there so we can traverse through only white spaces using A star Algorithm We are Going to 201 grids whereas in dijkstra's algorithm it is visiting 2106 grids so it will take more time.

So A* search algorithm runs in order of n time complexity whereas

Parameters	A* Algorithm	Dijkstra's Algorithm
Search Algorithm	Best First Search	Greedy Best First Search
Time Complexity	Time complexity is O(n log n), n is the no. of nodes.	The time complexity is O(n2).
Heuristics Function	Heuristic Function, f(n)=g(n)+h(n), g(n) represents the cost of the path from the starting point to the vertex n. h(n) represents the heuristic estimated cost from vertex n to the g.	f(n)=g(n), g(n) represents the cost of the path from the starting point to the vertex n. Dijkstra's Algorithm is the worst case of A star Algorithm.

Table 1 Difference between A* Algorithm and Dijkstra's Algorithm

IV. Conclusion

A* is just like Dijkstra, the only difference is that A* tries to look for a better path by using a heuristic function which gives priority to nodes that are supposed to be better than others while Dijkstra's just explore all possible paths.
Its optimality depends on the heuristic function used, so yes it can return a non optimal result because of this and at the same time better the heuristic for your specific layout, and better will be the results (and possibly the speed).

It is meant to be faster than Dijkstra even if it requires more memory and more operations per node since it explores a lot less nodes and the gain is good in any case.

Precomputing the paths could be the only way if you need realtime results and the graph is quite large, but usually you wish to pathfind the route less frequently

Dijkstra's is essentially the same as A*, except there is no heuristic (H is always 0). Because it has no heuristic, it searches by expanding out equally in every direction, but A* scan the area only in the direction of destination. As you might imagine, because of this Dijkstra's usually ends up exploring a much larger area before the target is found. This generally makes it slower than A*. But both have their importance of its own, for example A* is mostly used when we know both the source and destination and Dijkstra's is used when we don't know where our target destination is. Say you have a resource-gathering unit that needs to go get some resources of some kind. It may know where several resource areas are, but it wants to go to the closest one. Here, Dijkstra's is better than A* because we don't know which one is closest. Our only alternative is to repeatedly use A* to find the distance to each one, and then choose that path. There are probably countless similar situations where we know the kind of location we might be searching for, want to find the closest one, but not know where it is or which one might be closest. So, A* is better when we

know both starting point and destination point. A* is both complete (finds a path if one exists) and optimal (always finds the shortest path) if you use an Admissible heuristic function. If your function is not admissible off - all bets are off.

Acknowledgement

With immense pleasure and deep sense of gratitude, I wish to express my sincere thanks to my faculty Prof. Seetha R., VIT University, without her motivation and continuous encouragement, this project would not have been successfully completed. I would also like to acknowledge my friends, without their help and support it would have not been possible to write and understand such a long code The paper format is the official IEEE Format Paper.

References

1.http://theory.stanford.edu/~amitp/GameProgramming/

2.https://en.wikipedia.org/wiki/A*_search_algorithm

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

4. https://stackoverflow.com/questions/13031462/

5. http://www.hindex.org/2014/p520.pdf

6. https://www.youtube.com/watch?v=g024lzsknDo

7https://www.youtube.com/watch?v=cSxnOm5aceA

8. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

9. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/

Video Link

https://youtu.be/dS-fCJUCZVc