[CS502]-Graph Theory and Applications **Fall 2021-22**

A* Search Algorithm

Instructor: Vinod Reddy *Authors: Swapnil Narad, Ambar Mutha*

1 Introduction

Search algorithms or be more specifically pathfinding algorithms are very useful in many real-time settings like in the areas of networking, video games, robotics etc. It is used to find the shortest route between two points. Pathfinding is closely related to the shortest path problem, within graph theory. These algorithms extend from well-known graph traversals like Breadth-First Search and Depth First Search, Dijkstra's Algorithm and Bellman's Ford Algorithm. A* (A Star) Search Algorithm is also one of these paths finding algorithms.

For a greater understanding, we can say that Dijkstra's Algorithm is a special case for the A* search algorithm where we take the heuristic value for each vertex as 0. In contrast to Dijkstra and others, in the A* algorithm, a heuristic approach is implemented into a standard graph search algorithm that effectively plans ahead to make the right decision. Due to its heuristic components and thus goal-oriented nature, it also has its significance in Artificial Intelligence studies. Unlike Dijkstra, A* helps us to find the shortest distance only between the starting point and the ending point.

2 Working of the algorithm

2.1 Working

In a graph with weighted edges, the A* search algorithm finds the shortest path between a source vertex S and a target vertex T or determine whether no path exists connecting S and T. It is used in a huge graph where any blind search algorithm like BFS will be very inefficient and would not be terminated in a feasible time. The algorithm tracks the vertices to check in the open set and the checked vertices in the closed set. The open set is initialized with the start vertex S and the closed set is empty.

Each vertex v has a value $f(v)$ associated to it such that

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

where $g(v)$ is the cost required to reach the vertex v from S and $h(v)$ is a heuristic value which denotes the estimated cost to reach target vertex T from v. For the algorithm to work correctly, the heuristic $h(v)$ can be underestimated but cannot be overestimated i.e. it should never be higher than the actual cost.

The algorithm iterates over vertices in the open set until it is empty. In each iteration, the vertex with the smallest $f(v)$ is chosen and moved from open set to closed set and its unvisited neighbors are added to the open set. If we reach the target vertex, we have found the shortest path. If the open set becomes empty without reaching the target then the source and target are not connected.

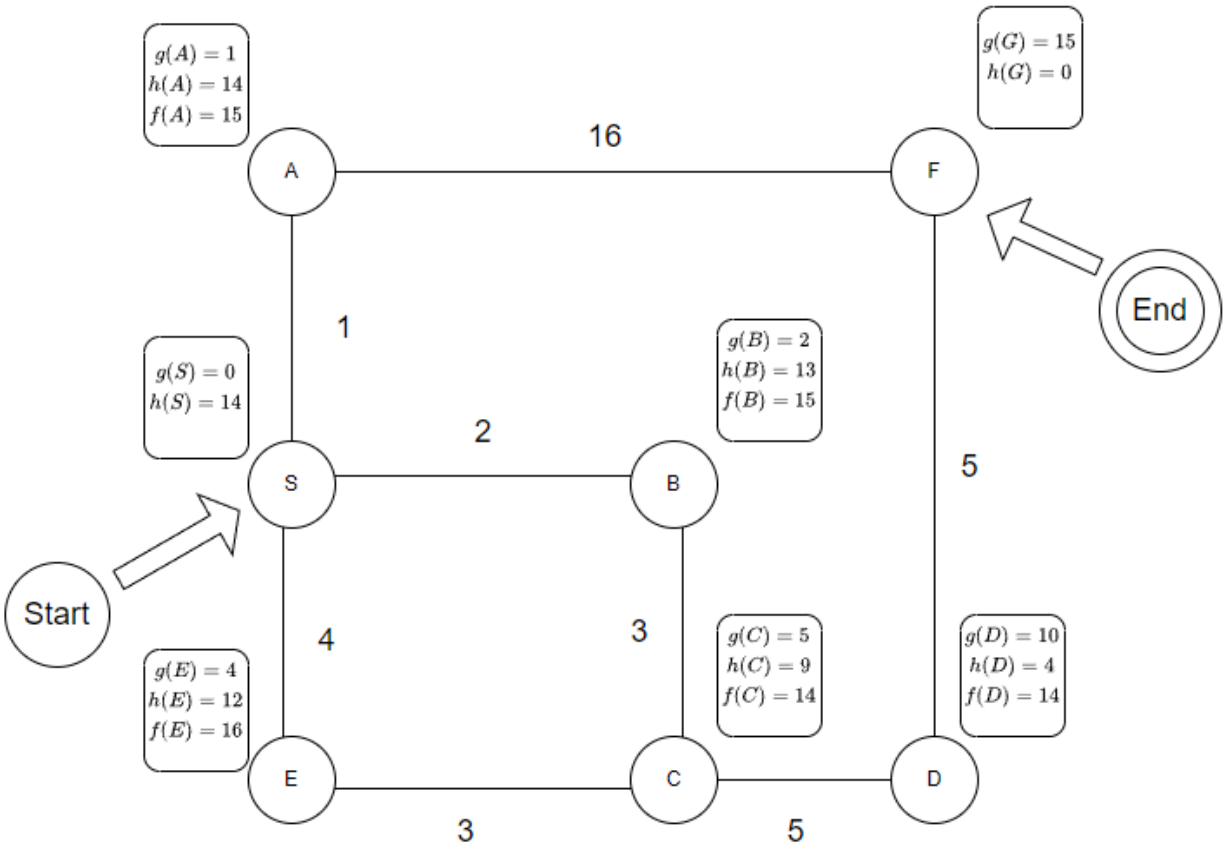
2.2 Pseudocode

Algorithm 1 Compute sum of integers in array

```
1:  procedure A* SEARCH(Node S,Node T, Graph G)
2:      S.f ← 0
3:      openSet ← {S}
4:      closedSet ← {}
5:      while openSet.size > 0 do
6:          current ← node from openSet having smallest f value
7:          openSet.remove(q)
8:          if current == T then
9:              return path from S to T
10:         end if
11:         for each neighbor in current.neighbors do
12:             if neighbor in closedSet then
13:                 continue
14:             end if
15:             new_g = current.g + cost of edge between current and neighbor
16:             if neighbor not in openSet then
17:                 openSet.add(neighbor)
18:             else
19:                 if new_g ≥ neighbor.g then
20:                     continue
21:                 end if
22:             end if
23:             neighbor.g = new_g
24:             neighbor.f = neighbor.g + Estimated cost of travelling from neighbor to T
25:             neighbor.parent = current
26:         end for
27:     end while
28:     return Error: Path does not exist
29: end procedure
```

2.3 Example

We are taking an example of a undirected graph G to do a dry run for the algorithm.



1. We will start with vertex S by initializing the algorithm with $g(S) = 0$ and $h(S) = 14$.
2. We will now add neighbours of vertex S.
   (a) We will first add A to open list with $g(A) = 1$, $h(A) = 14$ and $f(A) = 15$.
   (b) Then add B to open list with $g(B) = 2$, $h(B) = 13$ and $f(B) = 15$.
   (c) Finally add E to open list with $g(E) = 4$, $h(E) = 12$ and $f(E) = 16$.
3. Here we have $f(B)$ and $f(A)$ both having least and equal value $f(A) = f(B) = 15$, so we will be adding neighbours of A to open list as it entered the open list first, and therefore we will be adding F(which is also the end vertex) to the open list with $g(F) = 17$, $h(C) = 0$ and $f(C) = 17$. Since this is not the least value in open list, the algorithm will not stop here.
4. Now we have the least value of $f(B) = 15$ so now we will add neighbours of B to the open list, and therefore C will be added with $g(C) = 5$, $h(C) = 9$ and $f(C) = 14$.
5. Right now the least item on the open list is $f(C) = 14$, so we will be adding neighbours of it i.e. D with $g(D) = 10$, $h(D) = 4$ and $f(D) = 14$.

6. The least item in the open list is now $f(D)$, so we will add F to the open list with $g(F) = 15$, $h(F) = 0$ and $f(F) = 15$, which is also the least value in the open list, since F is also the end vertex or goal state and it is also having the least f value in the open list, the algorithm will be terminated and we will get the path as $S \longrightarrow B \longrightarrow C \longrightarrow D \longrightarrow F$ with total cost 15.

2.4 Complexity Analysis

Time Complexity: Consider a graph that starts as a straight line path with the starting vertex on one end and the ending vertex on the other, for example – 0 (source) $\rightarrow$ 1 $\rightarrow$ 2 $\rightarrow$ 3(target). So the worst case time complexity is $\mathcal{O}(E)$, where E is the number of edges in the graph.

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

2.5 Heuristic Functions

The choice of Heuristic Function is the most important part of determining the efficiency of the A* search algorithm. We can calculate $g(v)$ as it is definite, but as $h(v)$ is a guess, choosing the appropriate formula for determining the heuristic value should be done carefully. There are two ways to do so:

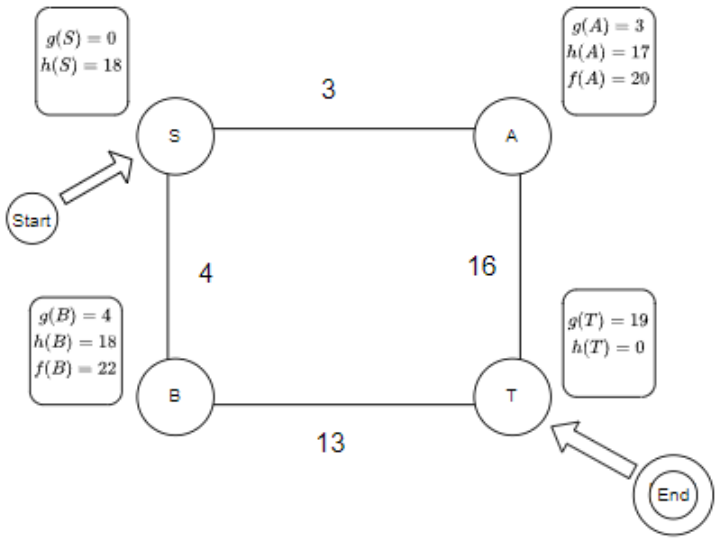
1. Calculate the exact value of $h(v)$: This will result in worst time and space complexities, this can be done in two ways:
   (a) We can compute the distances between every pair of nodes in advance(Pre-computation). This will be highly inefficient and infeasible with respect to space complexity $\mathcal{O}(V^2)$ that this method might demand.
   (b) If no obstacle is present in the graph, then we can easily calculate the distance between two nodes by the euclidean formula.
2. The other way of doing this will be by approximating the value of $h(v)$ using some heuristic formula: Based on the implementation, there can be various ways to approximate the value of $h(v)$. Some of these methods are: Consider $C \longrightarrow$ Current Vertex and $T \longrightarrow$ Target Vertex;
   (a) **Manhattan Distance**: This is defined as the sum of the absolute difference between the value of x coordinate of the current cell and the value of x coordinate of the target cell, with the absolute difference between the value of y coordinate of the current cell and the value of y coordinate of the target cell, i.e,
   $$h = \mid C.x - T.x \mid + \mid C.y - T.y \mid$$
   We can only consider movement in four directions i.e. (up, down, left, right) while using this formula.
   (b) **Diagonal Distance**: This is defined as the maximum of absolute values of differences in the target's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,
   $$= max(\mid C.x - T.x \mid, \mid C.y - T.y \mid)$$
   This formula can be used when we can explore in all the eight directions.
   (c) **Euclidean Distance**: This is the square root of the sum of the squares of the difference of x,y values of the current cell and that of the target cell, respectively., i.e,
   $$h = \sqrt{(C.x - T.x)^2 + (C.y - T.y)^2}$$
   This formula can be used when we are free to go in any directions.

2.6 Estimation

As we are guessing the value for our heuristic function $h(v)$, there are two cases possible other than guessing the correct distance between two points, these are:

2.6.1 Over-Estimation

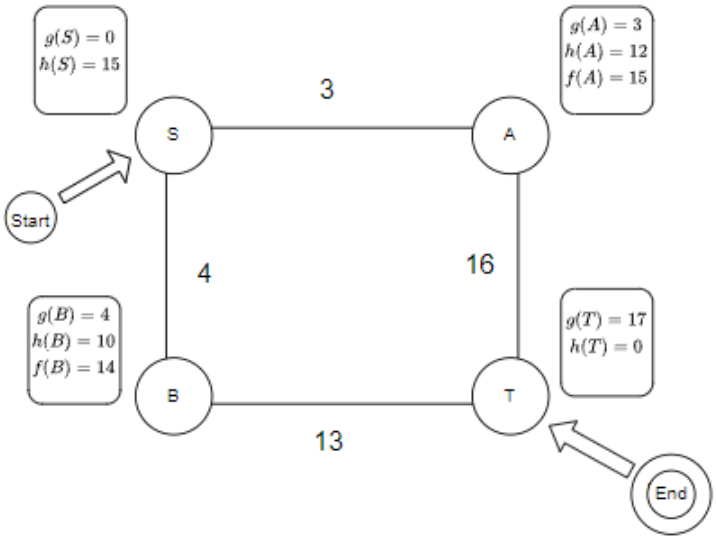
If the guess value is more than the actual value of the distance between the current node v and the target node t, i.e., $h(v) > d(v \to t)$, then the approximation is termed to be overestimated. If over-estimation is made while calculating the value of $h(v)$, then the algorithm may not return the optimum distance between the starting node and the target node. This can be explained with the example given below :



1. The open list will be initialised with the vertex S with $g(S) = 0$, $h(S) = 18$.
2. Now neighbours of S will be added to open list, A will be inserted with $g(A) = 3$, $h(A) = 17$ and $f(A) = 20$ and B will also be inserted with $g(B) = 4$, $h(B) = 18$ and $f(A) = 22$.
3. Since $f(A) < f(B)$, so neighbours of A will be taken and therefore T will be added to open list, with $g(T) = 19$, $h(t) = 0$ and $f(T) = 19$.
4. Since $f(T)$ has the least value in open list and also T is the target node and therefore the algorithm will be terminated here with path as $S \longrightarrow A \longrightarrow T$ with total cost as 19.
5. The optimal value of $d(S \longrightarrow T) = 17$, but the algorithm returns 19 this happened because of over-estimation in the heuristic function.

2.6.2 Under-Estimation

If the guess value is less than the actual value of the distance between the current node v and the target node t, i.e., $h(v) < d(v \to t)$, then the approximation is termed to be underestimated. Under-estimation guarantees the algorithm will return the optimum value, but more the underestimation, more tedious is the algorithm. This can be explained with the example given below :



1. The open list will be initialised with the vertex S with $g(S) = 0$, $h(S) = 15$.
2. Now neighbours of S will be added to open list, A will be inserted with $g(A) = 3$, $h(A) = 12$ and $f(A) = 15$ and B will also be inserted with $g(B) = 4$, $h(B) = 10$ and $f(A) = 14$.
3. Since $f(A) > f(B)$, so neighbours of B will be taken and therefore T will be added to open list, with $g(T) = 17$, $h(t) = 0$ and $f(T) = 17$.
4. Since $f(T)$ has the least value in open list and also T is the target node and therefore the algorithm will be terminated here with path as $S \longrightarrow B \longrightarrow T$ with total cost as 17.
5. The optimal value of $d(S \longrightarrow T) = 17$ which is also returned by the algorithm by using under-estimation.

3 Applications

Path finding have a very wide range of applications, there are several industries which use path finding to a great extent, some of the fields are Networking, Game development, Robotics, AI etc. Computer games like CS:GO, Age Of Empires, GTA series uses path finding extensively to design their bot movement, the game's bots or in some cases AI pedestrians design their route using A* search algorithm and its variants. Cui, Xiao & Shi, Hao [1] has already reviewed the various A* based gaming algorithms and addressed their uses in some of the popular games mentioned above in their paper. [2] and [3] have proposed robot path finding algorithms using A* algorithms.

3.1 Finding shortest route in a city

A common scenario where A* Search Algorithm can be used effectively is finding route between two points in a dense city with a large number of roads and intersections.

The problem can be modeled as a graph where road intersections are represented by vertices. The start and end points are vertices S and T respectively. The roads are represented by edges connecting these vertices. The weight of each edge is the length of the road in between. And the heuristic $h(v)$ for each vertex can be defined as the euclidean distance between v and T. Since the shortest route to T will always be at least as much as the euclidean distance, we are confident that our heuristic can never be an overestimate. Therefore, the A* search algorithm will always give the shortest path.

Similarly, the A* search algorithm can be used in any geographical path finding scenario.

4 Conclusions

To conclude, A* search algorithm is currently the best path searching algorithm and heavily optimises path searching in a large graph with obstacles between the start and target nodes. As a result, it has many real-world applications such as in navigating maps, video games and more. This algorithm's efficiency is due to the heuristic that determines which node next may provide the optimal path while ignoring any other irrelevant nodes.

5 References

1. *Cui2011AbasedPI* - A*-based Pathfinding in Modern Computer Games - Xiao Cui and H. Shi.
2. *Guruji2016* - Time-efficient A* Algorithm for Robot Path Planning - Akshay Kumar Guruji and Himansh Agarwal and D.K. Parsediya.
3. *Peng2015* - Robot Path Planning Based on Improved A* Algorithm - Jiansheng Peng and Yiyong Huang and Guan Luo.
4. *Wikipedia - Pathfinding*
5. *Wikipedia - A*-search-algorithm*
6. *GeeksForGeeks - A*-search-algorithm*