A* Algorithm Concepts and Implementation

.

A* (pronounced "A-star") is a graph traversal and path search algorithm, which is used in many fields of computer science due to its completeness, optimality, and optimal efficiency.[1] One major practical drawback is its space complexity, as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms that can pre-process the graph to attain better performance,[2] as well as memory-bounded approaches; however, A* is still the best solution in many cases.[3]

Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968.[4] It can be seen as an extension of Dijkstra's algorithm. A* achieves better performance by using heuristics to guide its search.

Compared to Dijkstra's algorithm, the A* algorithm only finds the shortest path from a specified source to a specified goal, and not the shortest-path tree from a specified source to all possible goals. This is a necessary trade-off for using a specific-goal-directed heuristic. For Dijkstra's algorithm, since the entire shortest-path tree is generated, every node is a goal, and there can be no specific-goal-directed heuristic.

What is an A* Algorithm?

It is a searching algorithm that is used to find the shortest path between an initial and a final point.

It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

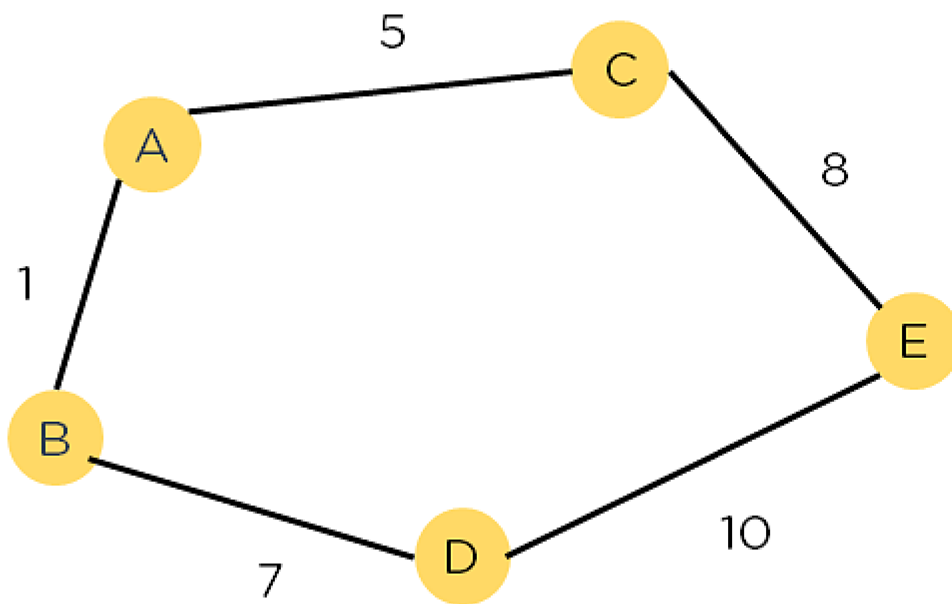

Figure 1: Weighted Graph

A major drawback of the algorithm is its space and time complexity. It takes a large amount of space to store all possible paths and a lot of time to find them.

Why A* Search Algorithm?

A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding. It is an extension of Dijkstra's shortest path algorithm (Dijkstra's Algorithm). The extension here is that, instead of using a priority queue to store all the elements, we use heaps (binary trees) to store them. The A* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. This function is used in conjunction with the f-heap data structure in order to make searching more efficient.

Let us now look at a brief explanation of the A* algorithm.

Explanation

In the event that we have a grid with many obstacles and we want to get somewhere as rapidly as possible, the A* Search Algorithms are our savior. From a given starting cell, we can get to the target cell as quickly as possible. It is the sum of two variables' values that determines the node it picks at any point in time.

At each step, it picks the node with the smallest value of 'f' (the sum of 'g' and 'h') and processes that node/cell. 'g' and 'h' is defined as simply as possible below:

- 'g' is the distance it takes to get to a certain square on the grid from the starting point, following the path we generated to get there.
- 'h' is the heuristic, which is the estimation of the distance it takes to get to the finish line from that square on the grid.

Heuristics are basically educated guesses. It is crucial to understand that we do not know the distance to the finish point until we find the route since there are so many things that might get in the way (e.g., walls, water, etc.). In the coming sections, we will dive deeper into how to calculate the heuristics.

Let us now look at the detailed algorithm of A*.

Algorithm

Initial condition - we create two lists - Open List and Closed List.

Now, the following steps need to be implemented -

- The open list must be initialized.
- Put the starting node on the open list (leave its f at zero). Initialize the closed list.
- Follow the steps until the open list is non-empty:

1. Find the node with the least f on the open list and name it “q”.
2. Remove Q from the open list.
3. Produce q's eight descendants and set q as their parent.
4. For every descendant:

i) If finding a successor is the goal, cease looking

ii)Else, calculate g and h for the successor.

successor.g = q.g + the calculated distance between the successor and the q.

successor.h = the calculated distance between the successor and the goal. We will cover three heuristics to do this: the Diagonal, the Euclidean, and the Manhattan heuristics.

successor.f = successor.g plus successor.h

iii) Skip this successor if a node in the OPEN list with the same location as it but a lower f value than the successor is present.

iv) Skip the successor if there is a node in the CLOSED list with the same position as the successor but a lower f value; otherwise, add the node to the open list end (for loop).

- Push Q into the closed list and end the while loop.

We will now discuss how to calculate the Heuristics for the nodes.

Heuristics

We can easily calculate g, but how do we actually calculate h?

There are two methods that we can use to calculate the value of h:

1. Determine h's exact value (which is certainly time-consuming).

(or)

2. Utilize various techniques to approximate the value of h. (less time-consuming).

Let us discuss both methods.

Exact Heuristics

Although we can obtain exact values of h, doing so usually takes a very long time.

The ways to determine h's precise value are listed below.

1. Before using the A* Search Algorithm, pre-calculate the distance between every pair of cells.

2. Using the distance formula/Euclidean Distance, we may directly determine the precise value of h in the absence of blocked cells or obstructions.

Let us look at how to calculate Approximation Heuristics.

Approximation Heuristics

To determine h, there are typically three approximation heuristics:

1. Manhattan Distance

The Manhattan Distance is the total of the absolute values of the discrepancies between the x and y coordinates of the current and the goal cells.

The formula is summarized below -

h = abs (curr_cell.x – goal.x) +

abs (curr_cell.y – goal.y)

We must use this heuristic method when we are only permitted to move in four directions - top, left, right, and bottom.

Let us now take a look at the Diagonal Distance method to calculate the heuristic.

2. Diagonal Distance

It is nothing more than the greatest absolute value of differences between the x and y coordinates of the current cell and the goal cell.

This is summarized below in the following formula -

dx = abs(curr_cell.x – goal.x)

dy = abs(curr_cell.y – goal.y)

h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)

where D is the length of every node (default = 1) and D2 is the diagonal

We use this heuristic method when we are permitted to move only in eight directions, like the King's moves in Chess.

Let us now take a look at the Euclidean Distance method to calculate the heuristic.

3. Euclidean Distance

The Euclidean Distance is the distance between the goal cell and the current cell using the distance formula:

h = sqrt ((curr_cell.x – goal.x)^2 +

(curr_cell.y – goal.y)^2)

We use this heuristic method when we are permitted to move in any direction of our choice.

The Basic Concept of A* Algorithm

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :

$f(n) = g(n) + h(n)$, where :

$g(n)$ = cost of traversing from one node to another. This will vary from node to node

$h(n)$ = heuristic approximation of the node's value. This is not a real value but an approximation cost

How Does the A* Algorithm Work?

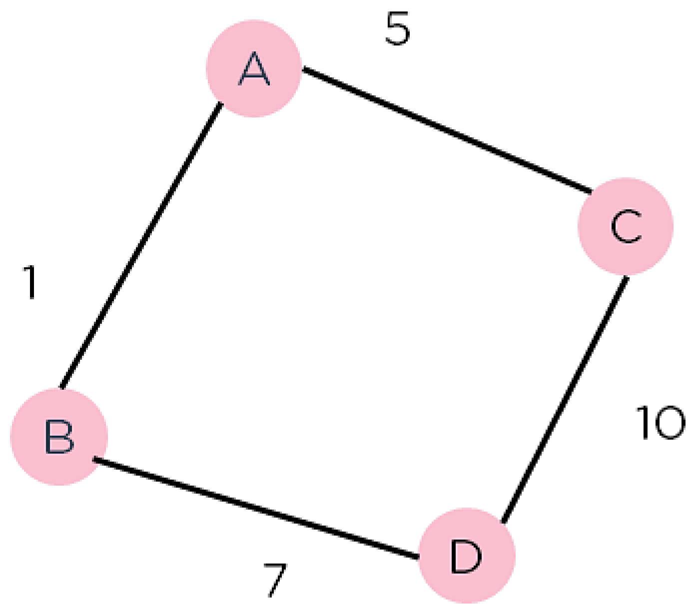

Figure 2: Weighted Graph 2

Consider the weighted graph depicted above, which contains nodes and the distance between them. Let's say you start from A and have to go to D.

Now, since the start is at the source A, which will have some initial heuristic value. Hence, the results are

f(A) = g(A) + h(A)
f(A) = 0 + 6 = 6

Next, take the path to other neighbouring vertices :

f(A-B) = 1 + 4

f(A-C) = 5 + 2

Now take the path to the destination from these nodes, and calculate the weights :

f(A-B-D) = (1+ 7) + 0

f(A-C-D) = (5 + 10) + 0

It is clear that node B gives you the best path, so that is the node you need to take to reach the destination.

Pseudocode of A* Algorithm

The text below represents the pseudocode of the Algorithm. It can be used to implement the algorithm in any programming language and is the basic logic behind the Algorithm.

- Make an open list containing starting node
  - If it reaches the destination node :
  - Make a closed empty list
  - If it does not reach the destination node, then consider a node with the lowest f-score in the open list

We are finished

- Else :

Put the current node in the list and check its neighbors

- For each neighbor of the current node :
  - If the neighbor has a lower g value than the current node and is in the closed list:

Replace neighbor with this new node as the neighbor’s parent

- Else If (current g is lower and neighbor is in the open list):

Replace neighbor with the lower g value and change the neighbor’s parent to the current node.

- Else If the neighbor is not in both lists:

Add it to the open list and set its g

How to Implement the A* Algorithm in Python?

Consider the graph shown below. The nodes are represented in pink circles, and the weights of the paths along the nodes are given. The numbers above the nodes represent the heuristic value of the nodes.

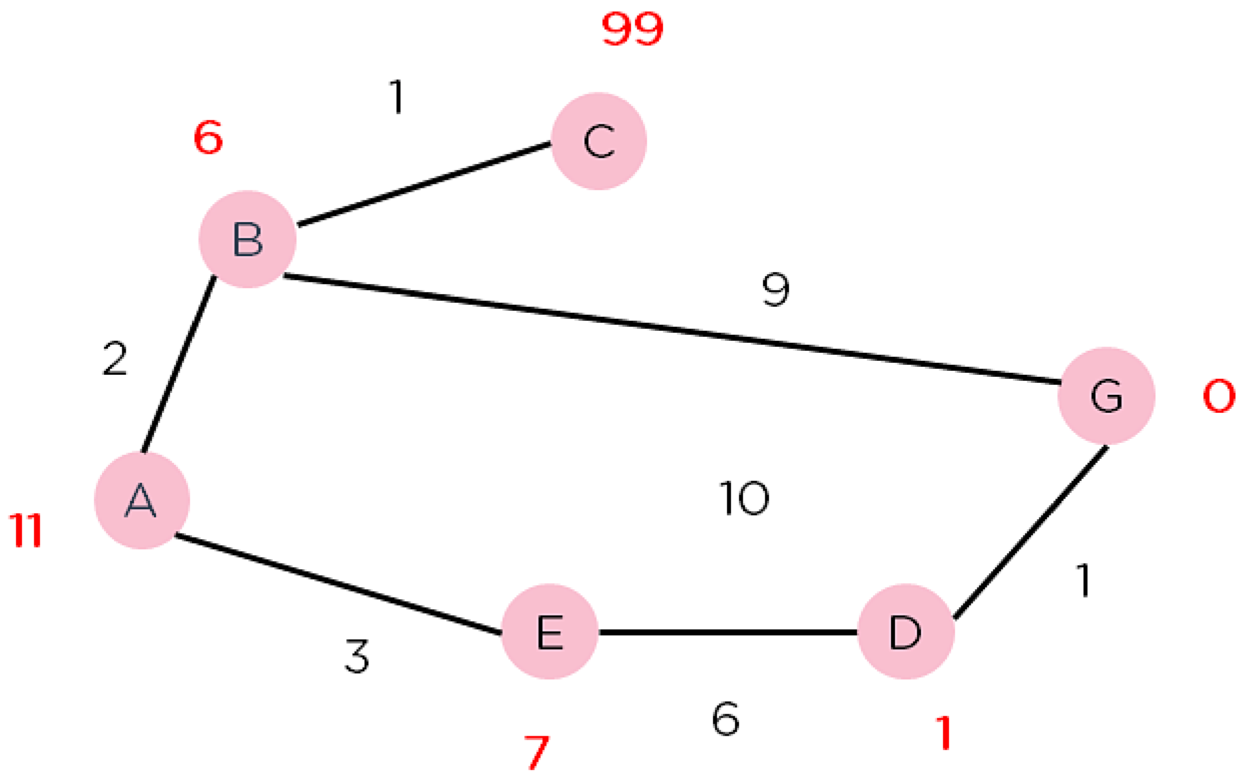

Figure 3: Weighted graph for A* Algorithm

You start by creating a class for the algorithm. Now, describe the open and closed lists. Here, you are using sets and two dictionaries - one to store the distance from the starting node, and another for parent nodes. And initialize them to 0, and the start node.

```
def aStarAlgo(start_node, stop_node):

        open_set = set(start_node)
        closed_set = set()
        g = {} #store distance from starting node
        parents = {}# parents contains an adjacency map of all nodes

        #ditance of starting node from itself is zero
        g[start_node] = 0
        #start_node is root node i.e it has no parent nodes
        #so start_node is set to its own parent node
        parents[start_node] = start_node
```

Figure 4: Initializing important parameters

Now, find the neighboring node with the lowest f(n) value. You must also code for the condition of reaching the destination node. If this is not the case, put the current node in the open list if it's not already on it, and set its parent nodes.

```
while len(open_set) > 0:
    n = None

    #node with lowest f() is found
    for v in open_set:
        if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
            n = v


    if n == stop_node or Graph_nodes[n] == None:
        pass
    else:
        for (m, weight) in get_neighbors(n):
            #nodes 'm' not in first and last set are added to first
            #n is set its parent
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight
```

Figure 5: Adding nodes to open list and setting parents of nodes

If the neighbor has a lower g value than the current node and is in the closed list, replace it with this new node as the neighbor's parent.

```
#for each node m,compare its distance from start i.e g(m) to the
#from start through n node
else:
    if g[m] > g[n] + weight:
        #update g(m)
        g[m] = g[n] + weight
        #change parent of m to n
        parents[m] = n

        #if m in closed set,remove and add to open
        if m in closed_set:
            closed_set.remove(m)
            open_set.add(m)
```

Figure 6: Checking distances and updating the g values

If the current g is lower than the previous g, and its neighbor is in the open list, replace it with the lower g value and change the neighbor's parent to the current node.

If the neighbor is not in both lists, add it to the open list and set its g value.

```python
        if n == None:
            print('Path does not exist!')
            return None

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []

            while parents[n] != n:
                path.append(n)
                n = parents[n]

            path.append(start_node)

            path.reverse()

            print('Path found: {}'.format(path))
            return path


        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)

    print('Path does not exist!')
    return None
```

Figure 7: Checking distances, updating the g values, and adding parents

Now, define a function to return neighbors and their distances.

```python
#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
```

Figure 8: Defining neighbors

Also, create a function to check the heuristic values.

```
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,

        }

        return H_dist[n]
```

Figure 9: Defining a function to return heuristic values

Let's describe our graph and call the A star function.

```
#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}
aStarAlgo('A', 'G')
```

```
Path found: ['A', 'E', 'D', 'G']

['A', 'E', 'D', 'G']
```

Figure 10: Calling A* function

The algorithm traverses through the graph and finds the path with the least cost

which is through E => D => G.