Project Name/Purpose- A* Search Algorithm

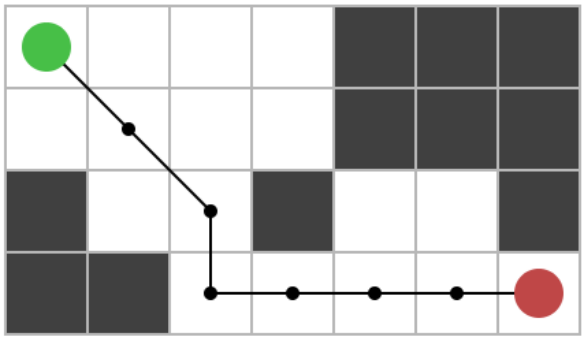
Project Category- Graph, Pathfinding, Greedy, Approximation, Dynamic Programming

Programming Paradigm/Algorithm Used- Approximation Heuristics, Dynamic Programming

Motivation-

To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances.

We can consider a 2D Grid having several obstacles and we start from a source cell (coloured green below) to reach towards a goal cell (coloured red below)



What is A* Search Algorithm ?-

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Why A* Search Algorithm ?-

Informally speaking, A* Search algorithms, unlike other traversal techniques has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact will be cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps uses this algorithm to find the shortest path very efficiently (approximation).

Explanation-

We will 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 will come 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 will 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 we will discuss in the later sections.

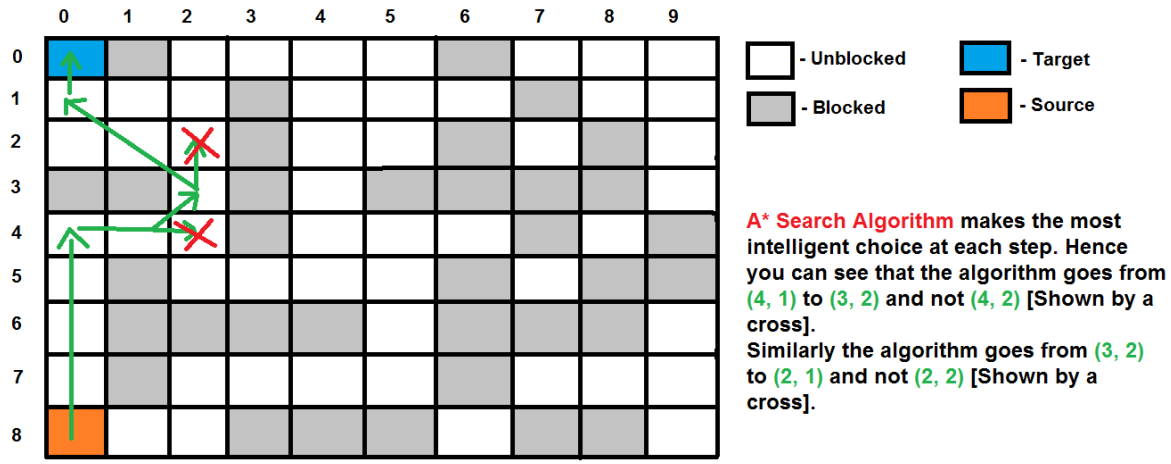
Algorithm -

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

```
1:      // A* Search Algorithm
2:      initialize the open list
3:      initialize the closed list
-       put the starting node on the open list (you can leave its f at zero)
4:
5:      while the open list is not empty
```

```
6:          find the node with the least f on the open list, call it "q"
7:          pop q off the open list
8:          generate q's 8 successors and set their parents to q
9:          for each successor
10:                 if successor is the goal, stop the search
11:                 successor.g = q.g + distance between successor and q
12:                 successor.h = distance from goal to successor (This can be done using many ways,
-                                  we will discuss three heuristics- Manhattan, Diagonal and
-                                  Euclidean Heuristics)
13:                 successor.f = successor.g + successor.h
-
14:           if a node with the same position as successor is in the OPEN list \
-               which has a lower f than successor, skip this successor
15:           if a node with the same position as successor is in the CLOSED list \
16:              which has a lower f than successor, skip this successor
17:           otherwise, add the node to the open list
18:         end
            push q on the closed list
        end
```

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. We will discuss about the heuristics in the later section below.



Heuristics-

We can calculate **g** but how to calculate **h** ?

We can do things.

A) Either calculate the exact value of **h** (which is certainly time consuming).

OR

B) Approximate the value of **h** using some heuristics (less time consuming).

We will discuss both of the methods.

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 [See this - https://en.wikipedia.org/wiki/Euclidean_distance]

B) ***Approximation Heuristics -***

There are generally three approximation heuristics to calculate **h** -

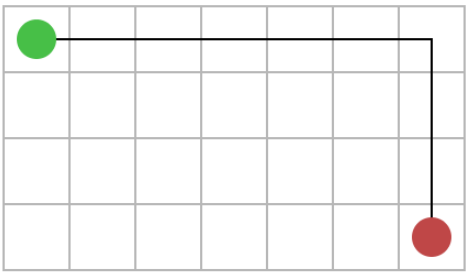
1) *Manhattan Distance -*

→ 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 ? – 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 green spot as source

cell and red spot as target cell).

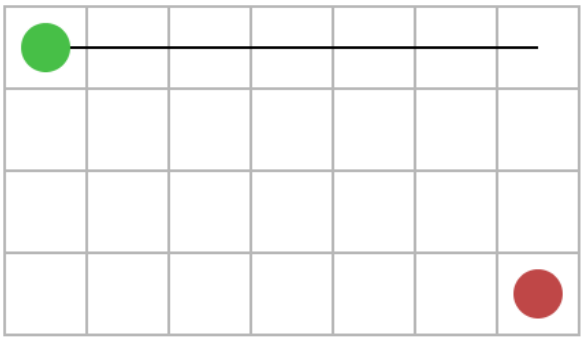


2) *Diagonal Distance-*

→ 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 ? – 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 green spot as source cell and red spot as target cell).



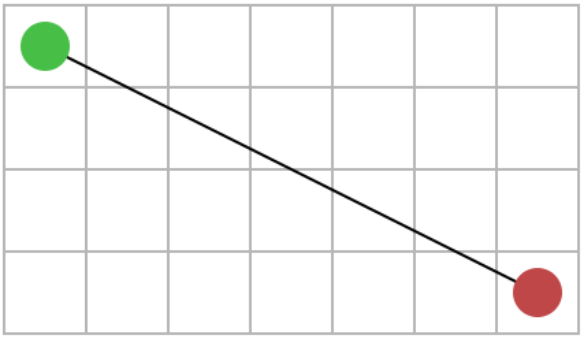
3) ***Euclidean Distance-***

→ 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

$h = \text{sqrt}\left((\text{current\_cell.x} - \text{goal.x})^2 + (\text{current\_cell.y} - \text{goal.y})^2 \right)$

→ When to use ? – When we are allowed to move in *any* directions

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



Relation (Similarity and Differences) with other algorithms-

Dijkstra is a special case of A* Search Algorithm, where - **h** = 0

Implementation-

We can use any data structure to implement open list and closed list but for best performance we use a ***set< >*** data structure of C++ STL(implemented as Red-Black Tree) and a boolean hash table for a closed list.

The implementations are similar to Dijsktra's algorithm. If we use a Fibonacci heap to implement the open list instead of a binary heap/self-balancing tree, then the performance will become better (as Fibonacci heap takes O(1) average time to insert into open list and to decrease key)

Also to reduce the time taken to calculate **g**, we will use dynamic programming.

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**

Applications-

This is the most interesting part of A* Search Algorithm. They are used in games !
But how?

Ever played ***Tower Defense Games*** ? [For more information go to- https://en.wikipedia.org/wiki/Tower_defense]

Tower defense is a type of strategy video game where the goal is to defend a player's territories or possessions by obstructing enemy attackers, usually achieved by placing defensive structures on or along their path of attack.

A* Search Algorithm is often used to find the shortest path from one point to another point. You can use this for each enemy to find a path to the goal.

One example of this is the very popular game- ***Warcraft III*** (see figure below)

What if the search space is not a grid and is a graph ?-

The same rules applies there also. The example of grid is taken for the simplicity of understanding. So we can find the shortest path between the source node and the target node in a graph using this A* Search Algorithm, just like we did for a 2D Grid.

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

Space Complexity-

In the worse case we can have all the edges inside the open list, so giving us a worse case space complexity as **O(V)**, where V is the total number of vertices

Exercise to the Readers-

Ever wondered how to make a game like- Pacman where there are many such obstacles.
Can we use A* Search Algorithm to find the correct way ?

Think about it as a fun exercise.

Articles for interested readers-

In our program, the obstacles are fixed. What if the obstacles are moving ?
Interested readers may go to -
http://theory.stanford.edu/~amitp/GameProgramming/MovingObstacles.html#recalculating-paths
for finding an excellent discussion on this topic.

Summary-

So when to use DFS over A*, when to use Dijkstra over A* to find the shortest paths ?

We can summarise this as below-

1) One source and One Destination-

→ Use A* Search Algorithm (For Unweighted as well as Weighted Graphs)

2) *One Source, All Destination -*

→ Use BFS (For Unweighted Graphs)
→ Use Dijkstra (For Weighted Graphs without negative weights)
→ Use Bellman Ford (For Weighted Graphs with negative weights)

3) *Between every pair of nodes-*

→ Floyd-Warshall
→ Johnson's Algorithm

References-

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

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