Course : ARTIFICIAL INTELLIGENCE (INT404)

- Email Address : kumarabhinav1777@gmail.com

- GitHub Link : https://github.com/itsabhinav98/PATH-FINDER
- Section : K18RH

STUDENT DECLARATION

This is to declare that this report has been written by me/us. No part of the report is copied from other sources. All information included from other sources have been duly acknowledged. I/We aver that if any part of the report is found to be copied, I/we are shall take full responsibility for it.

NAME : ABHINAV KUMAR

SECTION : K18RH

ROLL.No. : 28

NAME : SUDHANSHU RANJAN

SECTION : K18RH

ROLL.No. : 62

MY GITHUB LINK : https://github.com/itsabhinav98/PATH-FINDER

BONAFIDE CERTIFICATE

Certified that this project report " Path Finding in Ai" is the bonafide work of "ABHINAV KUMAR and SUDHANSHU RANJAN " who carried out the project work under my supervision.

PATH FINDER PROJECT

Here I will Discuss A* Path Finder Algorithm, how it works, and its implementation in pseudocode and real code with python

The A* search algorithm is an extension of Dijkstra's Algorithm useful for finding the lowest cost path between two nodes (vertices) of a graph. The path may traverse any number of nodes connected by edges (arcs) with each edge having an associated cost. The algorithm uses a heuristic which associates an estimate of the lowest cost path from this node to the goal node, such that this estimate is never greater than the actual cost.

The algorithm should not assume that all edge costs are the same. It should be possible to start and finish on any node, including ones identified as a barrier in the task.

A☆ Method Steps

1. Add the starting square (or node) to the open list.

2. Repeat the following:

A) Look for the lowest F cost square on the open list. We refer to this as the current square.

B). Switch it to the closed list.

C) For each of the 8 squares adjacent to this current square …

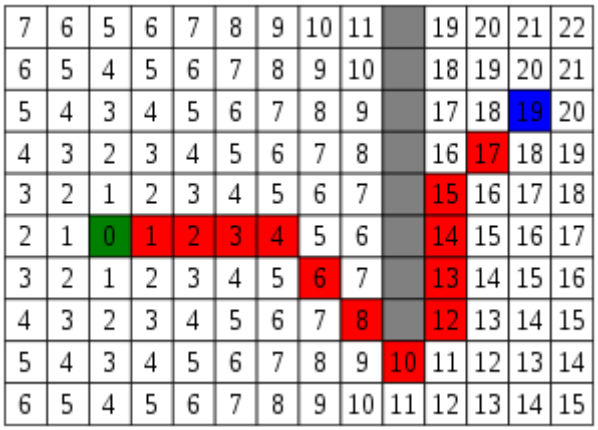
- If it is not walkable or if it is on the closed list, ignore it. Otherwise do the following.
- If it isn't on the open list, add it to the open list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
- If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.

D) Stop when you:

- Add the target square to the closed list, in which case the path has been found, or
- Fail to find the target square, and the open list is empty. In this case, there is no path.

3. Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

Let's take a look at a quick graphic to help illustrate this.



Task To Perform

Consider the problem of finding a route across the diagonal of a chess board-like 8x8 grid. The rows are numbered from 0 to 7. The columns are also numbered 0 to 7. The start position is (0, 0) and the end position is (7, 7). Movement is allow by one square in any direction including diagonals, similar to a king in chess. The standard movement cost is 1. To make things slightly harder, there is a barrier that occupy certain positions of the grid. Moving into any of the barrier positions has a cost of 100.

The barrier occupies the positions (2,4), (2,5), (2,6), (3,6), (4,6), (5,6), (5,5), (5,4), (5,3), (5,2), (4,2) and (3,2).

A route with the lowest cost should be found using the A* search algorithm (there are multiple optimal solutions with the same total cost).

Print the optimal route in text format, as well as the total cost of the route.

Optionally, draw the optimal route and the barrier positions.

Note: using a heuristic score of zero is equivalent to Dijkstra's algorithm and that's kind of cheating/not really A*!

Pseudocode

Following the example below, you should be able to implement A* in any language.

A* Search Algorithm Pseudocode

```
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)
```

----------Coding in python 

```
from __future__ import print_function
import matplotlib.pyplot as plt

class AStarGraph(object):
        #Define a class board like grid with two barriers

        def __init__(self):
                self.barriers = []
                self.barriers.append([(2,4),(2,5),(2,6),(3,6),(4,6),(5,6),(5,5),(5,4)
,2)])

        def heuristic(self, start, goal):
                #Use Chebyshev distance heuristic if we can move one square either
                #adjacent or diagonal
                D = 1
                D2 = 1
                dx = abs(start[0] - goal[0])
                dy = abs(start[1] - goal[1])
                return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)

        def get_vertex_neighbours(self, pos):
                n = []
                #Moves allow link a chess king
                for dx, dy in [(1,0),(-1,0),(0,1),(0,-1),(1,1),(-1,1),(1,-1),(-1,-1)]
                        x2 = pos[0] + dx
                        y2 = pos[1] + dy
                        if x2 < 0 or x2 > 7 or y2 < 0 or y2 > 7:
                                continue
                        n.append((x2, y2))
                return n

        def move_cost(self, a, b):
                for barrier in self.barriers:
                        if b in barrier:
                                return 100 #Extremely high cost to enter barrier squa
                return 1 #Normal movement cost

def AStarSearch(start, end, graph):

        G = {} #Actual movement cost to each position from the start position
        F = {} #Estimated movement cost of start to end going via this position

        #Initialize starting values
        G[start] = 0
        F[start] = graph.heuristic(start, end)

        closedVertices = set()
        openVertices = set([start])
        cameFrom = {}

        while len(openVertices) > 0:
                #Get the vertex in the open list with the lowest F score
                current = None
                currentFscore = None
                for pos in openVertices:
                        if current is None or F[pos] < currentFscore:
                                currentFscore = F[pos]
                                current = pos

                #Check if we have reached the goal
                if current == end:
                        #Retrace our route backward
                        path = [current]
```

Output Of Program

```
route [(0, 0), (1, 1), (2, 2), (3, 1), (4, 1), (5, 1), (6, 2),
(7, 5), (6, 6), (7, 7)]

cost 11
```