CS7050

Artificial Intelligence – Autumn 2022-23(CS7050)

Coursework

Heuristic Search Problem

Submitted By:

Divya Prackash Ravi | 22014173

Submitted To:

Prof. Vassil Vassilev

Submission Date: 11th December,2022.

ABSTRACT

Heuristic is a method of identifying a possible efficient path in a path solving problem, it may not provide the user with the best shortest/efficient path but it will find a path in a short duration of time [1]. Unlike other simplistic search algorithms like BFS and DFS where they explore almost all possible path and then choose the path with the least cost, a program with heuristic function wont explore a path if they have a higher cost to reach from starting point than an alternative with a lower cost. The cost to reach a point is the sum of the heuristic value of that point and the cost to get to the current position from the starting point. In this coursework, I aim to solve a maze solving problem by implementing A* algorithm which uses heuristics and other algorithms to explore the possibilities.

Contents

GROUP TASKS 4
DATA STRUCTURE: 4
SUCCESS CRITERIA: 4
PATH FUNCTION: 5
HEURISTIC FUNCTION: 6
A STAR 6
REPRESENTATION OF THE MAZE: 7
OUTPUT OF THE ALGORITHM: 7
CODE OF THE ALGORITHM: 8
ADDITIONAL TASKS 11
ADDITIONAL ALGORITHM 11
DESIGN A BOARD TO REPRESENT THE MAZE 11
VISUALIZE THE PATH 12
CODE OF THE ALGORITHM 14
REFERENCES 17

GROUP TASKS

Problem definition:

For a given maze of size 5 rows and 6 columns find the shortest path between the starting node (0,0) and the end node (4,5). The algorithm can only move Left, Right, Up and Down, it cannot move diagonally. The maze contains walls and no movement are permitted through them.

DATA STRUCTURE:

The data about the maze is stored in a list of lists which forms a matrix like structure, the cells with possible paths are represented with a value of zero, whereas the cells with walls are represented with the value of 1. An alternative data structure to store the data would be in a numpy matrix which is available in the numpy package.

SUCCESS CRITERIA:

The success criteria for reaching the end node is that, the algorithm should adhere by the rules mentioned above by only moving vertically and horizontally, it should not be able to pass through the walled of cells, it should ignore the longer paths or get trapped in a loop, it should not go to a node which has already been visited prior.

The algorithm should use the information gathered from the maze using the heuristic function, by choosing the cells with the least total cost to reach the end. The total cost to reach the end of a cell is the sum of the heuristic cost from that cell and the cost of getting to that cell from the starting node.

PATH FUNCTION:

The path function is developed in a way that it back tracks from the end point all the way to the start point via the path which the function has determined as the shortest path. The cost from the end to start should iteratively decreases as it gets closer we will use this information to check if we are headed in the correct path. The starting point has a cost of 0 hence we will check the cell which neighbouring cell follows the correct path , append that neighbouring cell to a list and jump to that cell until the current cell cost is 0

Here the reference matrix contains the cost details about the cells which have been explored by the algorithm, cccost is a variable which stores the current cell cost value, path is list data structure used to store the nodes to be followed for the shortest path.

```
def path_function(start_point,end_point,ref_matrix):
    path=[]

    i,j=end_point
    cccost=ref_matrix[i,j]
    path.append([i,j])

    while cccost>0:
        if i > 0 and ref_matrix[i-1,j]==cccost-1:
            i,j=i-1,j
            path.append([i,j])
            cccost-=1
        elif j>0 and ref_matrix[i,j-1]==cccost-1:
            i,j=i,j-1
            path.append([i,j])
            cccost-=1
        elif i<len(ref_matrix)-1 and ref_matrix[i+1,j]==cccost-1:
            i,j=i+1,j
            path.append([i,j])
            cccost-=1
        elif j<len(ref_matrix[i])-1 and ref_matrix[i,j+1]==cccost-1:
            i,j=i,j+1
            path.append([i,j])
            cccost-=1
    return path
```

HEURISTIC FUNCTION:

By following the constraints mentioned in the problem statement the perfect heuristic function to use for this use case would be the Manhattan Distance.

Manhattan distance is a measure of distance between any two points measured along the axes at a right angle. Eg: Manhattan distance between point 1 (x1,y1) and point 2 (x2,y2) is |x1 -x2| + |y1-y2| or it can be written mathematically as square root of $(x1-x2)^2 + (y1-y2)^2$

```
def heu(current,end):
    x1,y1=current
    x2,y2=end
    return abs(x1-x2)+abs(y1-y2)
```

A STAR

A* algorithm uses the heuristic values to guide itself to the end goal, it provides higher importance to the vertices near the starting the point and the ending point.

A* uses two functions to identify the shortest possible path, the g(n) function which represents the cost to reach the point n from the starting point, the h(n) function which is the heuristic function which gives an estimate cost from a vertex n to reach the goal state. It choses which cell to explore next based on the sum of g(n) and h(n), this is called as f(n). It explores cell with the lowest value of f(n) [3].

The A* algorithm can be implemented by using the PriorityQueue data structure available in queue library

Priority Queue is vastly different from the regular queue data structure. The queue data structure is a First In First Out data structure, whereas the priority queue works based on priority. The priority Queue takes in the data along with its priority assigned and then it will release the data with higher priority and then it will release the data with lower priority [4].

REPRESENTATION OF THE MAZE:

As mentioned in subsection ‘A’ the maze is stored in a list of lists wherein each list contains the information about the cells in a row. The wall cells are labelled as 1 and the traversable cells are labelled as 0.The starting node and ending node can be initialised by the user mentioning which cell they want it to be, to ensure that the user doesn’t place the starting or ending node within a wall, a preliminary check is made to confirm if the nodes are not in a wall.

```
maze
```

```
[[0, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 1, 0, 1, 0, 0],
 [0, 1, 0, 0, 1, 0],
 [0, 0, 0, 0, 1, 0]]
```

```
start_point=[0,0]
end_point=[4,5]
if(maze[start_point[0]][start_point[1]]==0):
    print('Eligible start point')
if(maze[end_point[0]][end_point[1]]==0):
    print('Eligible end point')
```

OUTPUT OF THE ALGORITHM:

The A* star algorithm is fed with the maze details , starting node , and end node. It generates two matrices of the same size to store the g score and f score, using the f score it chooses the optimal path.
Here the starting point is [0,0] and the ending point is [4,5] the path is shown by using ‘#’ symbol.

['#', 1, 0, 0, 0, 0]
['#', '#', '#', '#', '#', '#']
[0, 1, 0, 1, 0, '#']
[0, 1, 0, 0, 1, '#']
[0, 0, 0, 0, 1, '#']

CODE OF THE ALGORITHM:

As mentioned above the in ‘E’ the A* algorithm can be implemented using Priority Queue this package is available in the queue library in base python and the numpy library to create the g and f score matrices with infinite values

```
from queue import PriorityQueue
import numpy as np
```

Then the maze data are initialised

```
maze=[[0,1,0,0,0,0],
      [0,0,0,0,0,0],
      [0,1,0,1,0,0],
      [0,1,0,0,1,0],
      [0,0,0,0,1,0]
     ]
start_point=[0,0]
end_point=[4,5]
```

An object for the PriorityQueue is created and we push in a value of 0 as the priority, counter to keep track of how many items have been put inside, and then we insert the starting node as information to be used in the A* algorithm

```
count=0
open_set=PriorityQueue()
open_set.put((0,count,start_point))
previous={}
```

The g_score and f_score matrices are made using the numpy matrix package, we first create a ones matrix using numpy.ones package in the same dimension as the maze and it is multiplied with numpy.infinite to make a matrix filled with infinite values.

```
g_score=np.matrix(np.ones((len(maze),len(maze[1]))) * np.inf)
f_score=np.matrix(np.ones((len(maze),len(maze[1]))) * np.inf)
```

The starting node index is assigned as ‘0’ in the g_score matrix and the heuristic value is calculated then assigned in the f_score matrix.

```
g_score[start_point[0],start_point[1]]=0
f_score[start_point[0],start_point[1]]=heu(start_point,end_point)
```

Since open_set is a PriorityQueue object we wont be able to view the contents inside the queue without popping them all out, so a viewing list is created to store the indexes which are in the open_set

```
open_set_list=[start_point]
```

A temporary maze is made to be used within the algorithm, the A* algorithm is ran in a loop until there are no items in the priority queue. We pop one index out of the priority queue, and the same is done for the viewing list, a check is done if the popped index is the same as the ending index, if it is then the algorithm stops. The neighbours of the popped index is stored in a list if they are not a wall, now another loop is ran for each neighbour to check their g_score and f_score.

Since it takes a cost of 1 to move from one cell to another we create a temporary g_score value of 1 greater than the current popped index g_score value.

A comparison is done between the temporary g score of the neighbour and the actual g score of the neighbour in the g_score matrix is done. If the temporary g score has a lower value than the actual value, then we store the neighbour index in a dictionary as a key with the current index as value, update the g_score matrix with the temporary g score and a new f score is calculated for that neighbour.

If the neighbour is not in the viewing list (i.e that index isn't in the priority queue) then that neighbours f score, the counter, and the neighbour index is put inside the priority queue and the viewing list.

```
while not open_set.empty():
    current = open_set.get()[2]
    open_set_list.remove(current)
    if current == end_point:
        break

    x,y=current
    neighbors=[]
    if x-1>=0:
        if(tempmaze[x-1][y]==0):
            neighbors.append([x-1,y])
    if y-1>=0:
        if(tempmaze[x][y-1]==0):
            neighbors.append([x,y-1])
    if x+1<len(maze):
        if(tempmaze[x+1][y]==0):
            neighbors.append([x+1,y])
```

```
    if y+1<len(maze[1]):
        if(tempmaze[x][y+1]==0):
            neighbors.append([x,y+1])
    print(neighbors)

    for neighbor in neighbors:
        temp_g=g_score[current[0],current[1]]+1
        if temp_g<g_score[neighbor[0],neighbor[1]]:
            previous[str([neighbor[0],neighbor[1]])]=str([current[0],current[1
]])
            g_score[neighbor[0],neighbor[1]]=temp_g
            f_score[neighbor[0],neighbor[1]]=temp_g +
heu([neighbor[0],neighbor[1]],end_point)
            #tempmaze[current[0]][current[1]]=2
            if neighbor not in open_set_list:
                count+=1
                open_set.put((f_score[neighbor[0],neighbor[1]],count,neighbor)
)
                open_set_list.append(neighbor)
```

ADDITIONAL TASKS

ADDITIONAL ALGORITHM

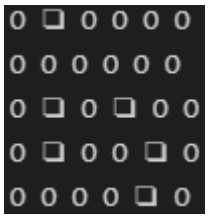
Dijkstra algorithm was chosen as the additional algorithm to implement, it is a type of greedy algorithm which finds the shortest distance in a graph.
It has two dictionaries, one to store the cost taken to visit that node and another dictionary to store through which node can the current node can be visited for the least cost.

For every node in the algorithm, it checks for the neighbouring nodes, it will calculate the cost to reach to that particular node from the start point, if the calculated distance is lesser than it has been recorded then update the dictionary with the node and distance, this process is repeated until we reach the end node. [5]

DESIGN A BOARD TO REPRESENT THE MAZE

We can represent a Board like structure in textual format by using the print method and ❑ text character to represent the walls

```
for i in range(len(maze)):
    for j in range(len(maze[1])):
        if(maze[i][j]==1):
            print('❑',end=' ')
        else:
            print('0',end=' ')
    print('')
```



VISUALIZE THE PATH

Using the path function explained in ‘C’ section of the group tasks, we can retrieve the path and with that information combined with time library can slowly show the updates done on a matrix using the print command.

```
import time
i,j=path[-1]
tempmaze[i][j]='#'
counter=-1
while [i,j]!= path[0] :
    counter-=1
    i,j=path[counter]
    tempmaze[i][j]='#'

    print('--------- iteration :' ,abs(counter+1) ,' ---------')

    for x in tempmaze:
        print(x,end='\n')
    time.sleep(2)
```

```
--------- iteration : 1  ---------
['#', 1, 0, 0, 0, 0]
['#', '0', '0', '0', '0', '0']
[0, 1, 0, 1, 0, '0']
[0, 1, 0, 0, 1, '0']
[0, 0, 0, 0, 1, '0']
--------- iteration : 2  ---------
['#', 1, 0, 0, 0, 0]
['#', '#', '0', '0', '0', '0']
[0, 1, 0, 1, 0, '0']
[0, 1, 0, 0, 1, '0']
[0, 0, 0, 0, 1, '0']
--------- iteration : 3  ---------
['#', 1, 0, 0, 0, 0]
['#', '#', '#', '0', '0', '0']
[0, 1, 0, 1, 0, '0']
[0, 1, 0, 0, 1, '0']
[0, 0, 0, 0, 1, '0']
```

```
--------- iteration : 4  ---------
['#', 1, 0, 0, 0, 0]
['#', '#', '#', '#', '0', '0']
[0, 1, 0, 1, 0, '0']
[0, 1, 0, 0, 1, '0']
[0, 0, 0, 0, 1, '0']
--------- iteration : 5  ---------
['#', 1, 0, 0, 0, 0]
['#', '#', '#', '#', '#', '0']
[0, 1, 0, 1, 0, '0']
[0, 1, 0, 0, 1, '0']
[0, 0, 0, 0, 1, '0']
--------- iteration : 6  ---------
['#', 1, 0, 0, 0, 0]
['#', '#', '#', '#', '#', '#']
[0, 1, 0, 1, 0, '0']
[0, 1, 0, 0, 1, '0']
[0, 0, 0, 0, 1, '0']
```

```
--------- iteration : 7  ---------
['#', 1, 0, 0, 0, 0]
['#', '#', '#', '#', '#', '#']
[0, 1, 0, 1, 0, '#']
[0, 1, 0, 0, 1, '0']
[0, 0, 0, 0, 1, '0']
--------- iteration : 8  ---------
['#', 1, 0, 0, 0, 0]
['#', '#', '#', '#', '#', '#']
[0, 1, 0, 1, 0, '#']
[0, 1, 0, 0, 1, '#']
[0, 0, 0, 0, 1, '0']
--------- iteration : 9  ---------
['#', 1, 0, 0, 0, 0]
['#', '#', '#', '#', '#', '#']
[0, 1, 0, 1, 0, '#']
[0, 1, 0, 0, 1, '#']
[0, 0, 0, 0, 1, '#']
```

CODE OF THE ALGORITHM

The Dijkstra algorithm can be implemented either using a heap or a priority queue, I have chosen to use heap for this implementation. The heap operation can be accessed by importing the packages from the heapq library

```
from heapq import *
```

Since Dijkstra works best on graph like structures we need to first convert the maze matrix into a graph.
We have stored the maze information in a list of strings, where each digit represents a paths cost to pass through, 1 means it is a possible path and 9 means it is a wall. We then convert these into a list of lists containing the values as integers

```
grid=['191111',
      '111111',
      '191911',
      '191191',
      '111191']
listgrid=[[int(char) for char in string] for string in grid]
```

An empty dictionary is created to contain the node to node relationship data

```
graph={}
```

we then iteratively update the graph by using a method made called get_next_nodes

```
cols=len(listgrid[0])
rows=len(listgrid)
for y,row in enumerate(grid):
    for x, col in enumerate(row):
        graph[(x,y)]=graph.get((x,y),[])+get_next_nodes(x,y)
```

The get next nodes method takes the indeces as input and checks for the possible neighbours vertically and horizontally

```
def get_next_nodes(x,y):
    check=lambda x,y: True if 0<=x<cols and 0<=y<rows else False
    possible=[-1,0],[0,-1],[1,0],[0,1]
    return [(grid[y+dy][x+dx],(x+dx,y+dy)) for dx,dy in possible if
check(x+dx,y+dy)]
```

Now that the graph is available, the Dijkstra algorithm can be implemented.

A list is created for the heap to work with, the start index is pushed in the heap list with a cost set as 0.

A loop is created so that the following process repeats until the queue is empty
The heap list is popped and the cost and index is stored in 2 temporary variables cost and node.
If the node is the end index then the loop is broken and returns all the visited nodes, else the neighbouring nodes are retrieved from the graph dictionary.

For each neighbour a new cost is calculated as the sum of the cost to reach to the current index and the cost to jump to the neighbour. If the cost is lesser than existing cost then update the cost_visited dictionary, visited dictionary, and push the neighbour and the cost in the heap list

Once the heap list is empty or the end index has been reached return the visited dictionary

```
def dijkstra(start,end,graph):
    queue=[]
    heappush(queue,(0,start))
    cost_visited={start:0}
    visited={start: None}

    while queue:
        cost,node=heappop(queue)

        if(node==end):
            break

        next_nodes=graph[node]
        for next_node in next_nodes:
            neighbor_cost,neighbor_node= next_node
            new_cost = cost_visited[node]+int(neighbor_cost)

            if neighbor_node not in cost_visited or new_cost <
cost_visited[neighbor_node]:
                heappush(queue,(new_cost,neighbor_node))
                cost_visited[neighbor_node]=new_cost
                visited[neighbor_node] = node
    return visited
start=(0,0)
end=(5,4)
path=dijkstra(start,end,graph)
```

Since the returned value is a dictionary we can backtrack from the end point to the start point

```
a=end
print(a)
while a!=start:
    a=path[a]
    print(a)
```

REFERENCES

[1] https://users.cs.cf.ac.uk/Dave.Marshall/AI2/node23.html

[2] https://xlinux.nist.gov/dads/HTML/manhattanDistance.html

[3]http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[4] https://pages.cs.wisc.edu/~vernon/cs367/notes/11.PRIORITY-Q.html

[5] https://www.educative.io/answers/what-is-dijkstras-algorithm