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

ADDITIONAL TASKS

ADDITIONAL ALGORITHM

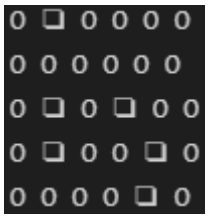
Dijkstra algorithm was chosen as the additional algorithm to implement, it is a type of greedy algorithm which finds the shortest distance in a graph.
It has two dictionaries, one to store the cost taken to visit that node and another dictionary to store through which node can the current node can be 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. [2]

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://www.educative.io/answers/what-is-dijkstras-algorithm