Tree Based Search

Mikkel Aguilar
103613812

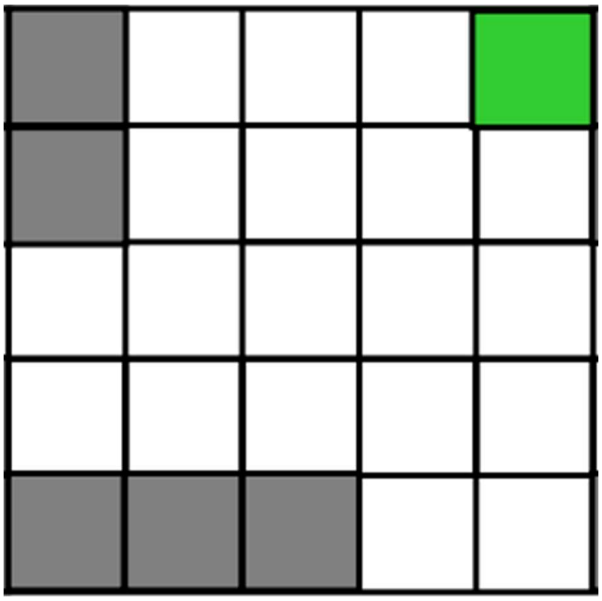
Due Date:

April 22

Subject Code

COS30019

Table of Contents



Instructions

The assignment is run on a DOS command line, written in the Python programming language. Here are the steps to using the program:

1. Navigate to the folder through folder through the command line
2. Input this format into the command line:
    - 'main.py'
    - 'data.txt' or any text file with the appropriate structure
    - 'BFS' or any acronym for a search method (DFS, GBFS, AS, CS1, CS2)
3. Press 'Enter', and you will see the directions to for a robot AI to reach a specific goal
4. User will be notified of invalid formats

Introduction

The Robot navigation problem is as follows: Given a grid, N(1) x N(2) where N is a positive integer, find a path using 4 directions (up, down, right, left) to reach a goal state where an entity (in this case a robot) reaches a goal square from its initial given position.

A single grid can have squares categorized into 4 groupings – a normal square, a goal square, a wall, and the starting square. The text file contains the data on which squares fall into each category.

For the robot to navigate through the graph, it will use common tree traversal concepts such as depth first search, breadth first search and other known or custom methods to find a valid path to a goal (if any is found).

Glossary

Node: A single point in the graph, can be a wall, goal, or a normal node
Neighbor: The nodes that are adjacent (left, right, up, down) to a given node
Stack: Data structure where the last element put in, is the first element out (LIFO)
Queue: Data structure where the first element put in, is the first element out (FIFO)
Heuristic: In AI, it is a value used by a machine to deduce which action to take to reach a solution (rather than a step-by-step approach)
Square: The position of a node (i.e. (0, 1))

Search Algorithms

Breadth First Search:
Uses a queue data structure (FIFO), examining nodes in the order of their level/distance from the starting node in a layered manner. This search method always finds the shortest path because by the time it reaches the goal, all the previous levels have already been visited. The downside of this, however, is that the total number nodes that have been visited is usually much higher than other search methods, meaning it is less efficient.

Depth First Search:
Uses a stack data structure (LIFO), where a robot keeps selecting a node to examine until it reaches a goal reaches a node with no more unvisited neighbors. In the latter case it keeps backtracking until it reaches a parent node with an unvisited neighbor. The robot follows this order when choosing nodes to expand to: UP, LEFT, DOWN, RIGHT meaning they initially try to move UP rather than DOWN if UP is possible. Depth first search is generally very inefficient as it may prioritize a direction that does not lead to the end goal, also very rarely finds the shortest path to a goal.

Greedy Best First Search:
This search method uses a heuristic f(n) = h(n) where h(n) is the Manhattan distance from a node to the goal node. Greedy search utilizes a list which contain all the potential nodes the robot can travel to; this list initially contains all the neighbors of the starting node, and as each node is examined, the open list appends all the unvisited neighbors of that node. The way in which nodes are chosen to be expanded is through the heuristic, where the node with the lowest heuristic among every node in the open list is chosen. GBFS is inconsistent when finding the shortest path, but most of the time finds the path the quickest among the searches.

A Star Search:
Uses the same heuristic as the GBFS with an added cost of moving from one node to another represented by g(n), hence f(n) = h(n) + g(n). A Star also uses a list of potential nodes to expand, the only difference is a node is chosen depending on their distance from the end goal and the cost to move which differ for each action. AS is more consistent in finding the shortest path but it is not guaranteed. If anything, it doesn't find the shortest path but finds a path close to the shortest.

Custom Search 1:
This custom search does the same thing as depth first search, just changes the order of the nodes to be examined. This search focuses on moving horizontally first before vertical movement (Order: RIGHT, LEFT, DOWN, RIGHT). This uninformed search is not optimal in an environment with walls and usually does not find the shortest path if a goal is not in the same horizontal axis.

Custom Search 2:
Works very similar to GBFS (same heuristic as well) but instead of just choosing the node with lowest heuristic from the open list, it also expands the neighbors of the node with he second lowest heuristic value. This can be useful in cases where the lowest heuristic node leads into a 'dead end' area. This informed search is a bit more consistent in finding the shortest path than GBFS but at the same time visits doubles the nodes, thus has problems with efficiency.

Implementation

Commonalities:
All search methods below return a list with 2 values; the first containing a nested list of the path using the coordinates of the nodes (i.e. [(0, 1), (0, 2), (1, 2)]), and the second containing another nested list of all the nodes visited when attempting to find a path. Given a path cannot be found, the methods return a list of 2 nested lists, in which both are empty. If the starting node given to any of these functions equal the end node, both nested lists will return the single starting node coordinate. Parameters include a graph where the values are the neighbors of the key, a starting node, and an end node.

Breadth First Search:

```
path list = [[start node]], index = 0, visited nodes = [start node]
if start == end then return [[start node], [start node]]
loop until index < length of path list
    current path = path list[index], last node = current path[-1],
    next nodes = graph[last node]
    if end node in next nodes then
        current path append end node
        return current path and visited nodes
    for every node in next nodes
        if node is not a visited node
            new path = copy of current path, new path append next
            node, path list append new path, visited nodes append
            node
    increase index by 1
return empty lists
```

Depth First Search:

```
stack = [starting node], visited = [starting node], path = [starting
node]
loop until stack is empty
    next = stack pop
    if next ==  end node
        return path and visited
    for neighbours in graph[next]
        if neighbour is not a visited node
            stack append neigbour, visited append neighbour, path
            append neighbour, break the loop
    if stack is empty
        if len(path) > 1
            tack append path[-2], path pop
        else return empty lists
```

Greedy Best First Search:

```
visited = [], open list = {}, parent = {}, found path = False
open list[starting node] = cost to reach goal (separate function)
loop until open list is empty
    lowest square = square with the minimum cost in the open list
    (keys)
    open list pop lowest square, visited append lowest square
    if lowest square == end node
        found path = True
        break loop
    for neighbour in graph[lowest square]
        if neighbour is not visited
            open list [neighbour] = cost to reach goal
            if neighbour not in parent
                parent[neighbour] = lowest square
if found path == True
    return backtraced (separate function) path and visited
else return empty lists
```

A Star Search:

Same as GBFS but the heuristic in line 2 and 11 adds the cost to move (separate function) as an additional cost

Custom Search 1:

Uses depth first search but adjusts the order of the graph values using an external function.

Custom Search 2:

Same as GBFS with adjustments to loop:

```
loop until open list is empty
    second lowest square = None
    lowest square = square with the minimum cost in the open list
    (keys)
    open list pop lowest square, visited append lowest square
    if lowest square == end node
        found path = True
        break loop
    if open list is still not empty (after popping earlier in the
    loop)
        second lowest square = square with the minimum cost in the
        open list (keys)
        open list pop second lowest square, visited append second
        lowest square
        if second lowest square == end node
            found path = True
            break loop
    for neighbour in graph[lowest square]
        if neighbour is not visited
            open list[neighbour] = cost to reach goal
        if neighbour not in parent
            parent[neighbour] = lowest square
    if second lowest square is not None
        for neighbour in graph[second lowest square]
            if neighbour is not visited
                open list[neighbour] = cost to reach goal
            if neighbour not in parent
                parent[neighbour] = second lowest square
```

Implementation (Utilities)

Convert Points to Moves:

```
parameters --- path
path moves = []
for i in the range from 0 to length of path
      current node = path[i]
      next node = path[i + 1]
      loop through neighbour keys and values (key is the direction,
      value is the neighbour node)
            if next node == value
                  path moves append key
return path moves
```

Order Graph:

```
parameters --- graph, order
loop through core node(key) and neighbours(value) in graph dictionary
      temporary var = []
      node = core node
      loop through directions in order
            try if node neighbours[direction] in  neighbours
                  temp append node neighbours[direction]
            except KeyError then pass
      graph[core node] = temp
return graph
```

Cost to Reach Goal:

```
parameters --- node, goal
x = absolute value of node[0] - goal[0]
y = absolute value of node[1] - goal[1]
return x + y
```

Cost to Move:

```
parameters --- node, origin node = None
cost to move = 0
if origin node is not None
      loop through origin node neighbours where move is the key and
      neighbour is the values
            if neighbour == node
                  if move is 'down'
                        cost to move = 1
                  if move is 'up'
                        cost to move = 1
                  if move is 'left'
                        cost to move = 1
                  if move is 'right'
                        cost to move = 1
return cost to move
```

Back trace:

```
parameters --- parent, start, end
path = [end]
loop while path[-1] is not start
      path append parent[path[-1]]
reverse path
return path
```

Features/Bugs/Missing

- Program can read data from a text file
- Program can convert data into a grid and the locations of specific nodes in a grid
- Program can create a hash table that stores a nodes location as a key and their neighbors as values
- Support for Breadth First Search
- Support for Depth First Search
- Support for Greedy Best First Search
- Support for A Star Search
- Support for a Custom Search 1
- Support for Custom Search 2
- Functions keep track of coordinates in a path
- Functions keep track of coordinates that have been visited by the robot
- Program can convert a coordinate in a path to directions (i.e. (0, 1), (0, 2) becomes down)
- Invalid user input formats are detected and rejected
- Program prints out a path and the number of visited nodes for each goal
- Program can find the shortest path to all goals
- Program assumes that format of the text file is follows the requirements
- Program does not run on an executable file (.exe)
- Runs on the command line
- No known bugs have been found

Research Initiative

As additional research I extended the program to include a function that finds the shortest path that reaches all goal nodes in the grid. To implement this as accurately as possible, I tested using all the different searches I had already made and tried searching for a single goal and then once the goal has been reached, started another search using the goal as the starting position for the next search. The nature of this research was to find the shortest path so I decided to settle on Breadth first search despite its inefficiency as it will always find an appropriate path.

One of the major issues I had when trying to create this function was finding a way for a robot to know in what order the robot should try searching through the goals. To solve this, I had to order the goals using a lambda function, where the 'key' for ordering is the Manhattan distance AKA the cost to reach goal from the starting location to each one of the goals.

The function returns a list of directions which basically contain the paths of multiple breadth first searches extended upon each other. When no path is found in any of the Breadth first searches, the function returns 'None'.

Overall, this function was not overly difficult to implement given I had already completed the difficult part of the assignment, the actual searches.

To run this function, you instead of putting in a search method into the command line, put in 'ALL'. It will look like this:

"main.py data.txt all"

Conclusion

An artificial intelligence robot has a multitude of ways to solve this grid navigation problem, whether it be an informed or uninformed search. Every method can be 'better' and 'faster' than another given the right problem and conditions but generally the most effective and efficient search method is the A Star search. It is no surprise that A.I's most commonly implement A star because the algorithm itself considers all relevant information in the form of its heuristic f(n) = g(n) + h(n). By using both the neighbor nodes' 'cost to reach goal' and the 'cost to move', the robot can find the move that takes him closest to the goal with the lowest cost. Although there are situations where A star does not find the shortest path 100% of the time, the number of nodes moves taken to find a path is usually lower than other searches.

The performance of the search algorithms can still be improved upon further, especially the informed searches. This can be done by using a better heuristic that considers positions of walls instead of the regular 'Manhattan' distance.

The program is very functional, but it could have been improved upon had Vectors been used to represent coordinates rather than tuples.

Acknowledgements/Resources

https://www.geeksforgeeks.org/ - Foundation code for the BFS and DFS searches

https://www.youtube.com/watch?v=A8pmud1UhoQ&ab_channel=ShaulMarkovitch – Understanding of GBFS concept

https://www.youtube.com/watch?v=71CEj4gKDnE&ab_channel=AnishKrishnan – Understanding of A star concept

https://docs.python.org/3/library/re.html - Understanding regular expression to properly interpret data from a file

https://www.javatpoint.com/heuristic-techniques - Learning heuristic and their relations to search methods

http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html - How heuristics are implemented in GBFS and A*

https://stackoverflow.com/questions/8922060/how-to-trace-the-path-in-a-breadth-first-search - Learn how to back track through visited nodes to find a path

References

'robot.py' – Contains the robot class with all utility and search functions

'main.py' – Takes in parameters from the command line to create a robot that completes a search

'nodes.py' – Contains the node class with all its variables (including neighbors)

'data.txt' – Main test case

'test.txt' – Extra test case (proof of working functionality)

'Report.docx' – This Document