h

Assignment – Option B

The Robot Navigation Problem

Swinburne University of Technology

COS30019 – Introduction to Artificial Intelligence

Justin Sargent 9989900

**Introduction**

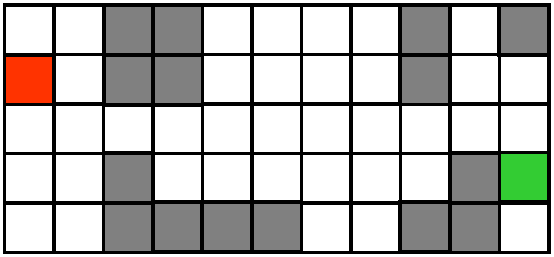
**Robot Navigation Problem**

The problem consists of a grid style map, with a start cell and a goal cell. The solution to the problem is to begin at the start cell and end at the goal cell whilst navigating a path around filled, inaccessible cells.

Many solutions exist, as there are many possible paths the theoretical robot can take.

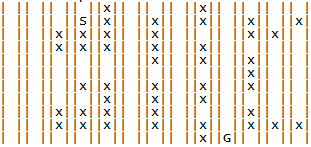
The solutions available for implementation will depend on the results of various AI searching algorithms, the paths that they generate, the amount of ‘nodes’ (cells) they expand (or visit) and the amount of cells which make up the resulting path.

**The initial testing map:**

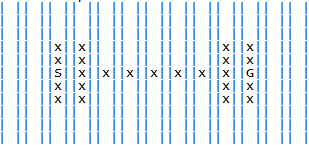


The red cell represents the start and the green cell represents the goal.

**Other testing maps:**



x : Filled cells, S : start, G : goal



**Glossary of Common AI Terminology**

**Node –** The subject of an AI search, connected to several other nodes, is part of a network which makes up critical characteristics of an environment. For example, each cell in the Robot Navigation Problem is a node.

**Node Expansion –** Expanding a node means taking information from that node which references other nodes along with how they are connected, whether by value, name, direction, distance or both.

**Search Tree –** If the starting node is expanded to find 3 more nodes and each node is placed in a line, below the starting node with a line connecting each node to the starting node, then the diagram begins to make a ‘tree’ shape. The second row, or ‘tier’ is then expanded to find more nodes, which are placed below in the third tier, linking back to the node which expanded it. Some of these new ‘branches’ may run into a dead end and stop, whilst other branches continue to expand into another tier. This shapes the tree into a structure which depicts the connections of every node.

**FIFO list** – A first in, first out list where the oldest element is evaluated and removed first and any additional elements are added into the back, waiting until the previous elements are dealt with before advancing to the front of the list

**Manhattan Distance Heuristic** – A method of cost calculation which involves counting horizontal and vertical square movement within a grid

**Euclidean Distance Heuristic** – A method of cost calculation which involves measuring the straight line distance between two points

**Search Algorithms**

**Depth First Search (DFS)**

This algorithms works by continuously chaining node expansion using the first built-in choice available until it reaches the bottom of the search tree. When all options are exhausted, it will either get stuck on the last node or move back to a previous node, depending on implementation.

Depth searching is known for being uncertain with regards to how many node expansions or path distance traversed. Some searches may find a goal towards the starting side of the search tree, taking much less time to find. Searches may also be unlucky and only find the goal when coming close to completing every possible node expansion.

Depth searches have no data available with regards to the goal cell, therefore it is an *uninformed* search.

The order of preference for node expansion can also have a large impact on the outcome of DFS on a single map.

Below are graphs which show the outcome of every possible permutation of node expansion preference against the resulting path count with regards to up, down, right and left.

The results of the Figure 1 test show that on average, the depth search will generate a slightly higher path count for most preferential orders of node expansion when run on the test map.

More importantly, it can be seen that there are a wide variety of efficient and less effective solutions available.

[Figure 1]

This next series of results are from the same test. This graph tracks the node expansions for every direction order possibility. It is clear to see that there are a higher number of iterations in the upper half of the number range, meaning that for this particular map DFS is more likely to have both an increased path length and a higher amount of node expansions before reaching goal.

[Figure 2]

**Breadth First Search (BFS)**

After this algorithm expands a node, it places every accessed node at the end of a FIFO (first in first out list), then removes the current node from the list and accessed the next node. When in operation, this algorithm will expand each node in the current tier and store nodes from the next tier at the end of the list, thus expanding the nodes in the environment one tier at a time.

Generally this algorithm will work best if the goal node is close to the starting node, as the goal node will be found during a search on the closer tier.

If processing time is not your concern, this algorithm is a safe option, as it will return the shortest path to the goal every time, since it will never expand past the first tier containing the goal node.

**Note:** BFS always contains the same path count for every preferential order of node expansion, therefore no graphical data is necessary.

The BFS algorithm varies little in the amount of node expansions when the preference of direction varies. This low variance can be explained by the fact that every iteration will find the goal node on the same tier, with the only variance being how the goal node is ordered within that tier. For some iterations, the goal node is expanded sooner and other iterations expanded it two nodes after.

[Figure 3]

**Greedy Best First Search (GBFS)**

The greedy algorithm is an informed search, meaning it has information about the goal node. Greedy best first chooses node expansion based on whatever node is closer to the goal. In this program’s application, each cell is given a number based on how many cells would have to be expanded to get to the goal, ignoring inaccessible cells. This is normally referred to as Manhattan distance. Therefore this algorithm follows the function F(n) = h(n), where h(n) is the Manhattan cost to goal node.

The algorithm in this application chooses the cell with the lowest path count to the goal cell. If cells are equal in minimal cell count from goal, it will preference up, right, down then left.

The advantage of greedy best first is it will always move in the general direction of the goal, however it can be tricked into expanding into dead ends, costing more processing time for node expansion.

Greedy will not always return the shortest path count like BFS. This is because it technically deepens itself to the goal based on informed preference.

**A\* Search (AS)**

This algorithm expands the idea of GBFS with an additional factor based on path cost from the start. Accessed nodes are analysed in terms of Manhattan distance from start and goal combined. The preferential decision will be F(N), the lowest total number of both values summed. F(n) = h(n) + g(n), where g(n) is the cost from start so far and h(n) is the cost to the goal node.

Advantages for this algorithm generally include having a shorter path to the goal.

In this application however A\* performs very chaotically, as the sum of cells large areas of cells where the cell cost combinations are equal. The algorithm delays itself by taking unnecessary detours as the similar cost of cells makes no clear preferential path available, making the search almost random.

Unfortunately, this version of A\* is not useful or beneficial for reducing path count or node expansions.

CUS2 is a custom informed searched named ‘Advanced A\*’ in this program which attempts to improve upon the downfalls of A\* in this application.

**CUS1 – Lowest Cost First Search (LCFS)**

LCFS expands nodes tier by tier like BFS does, however it prioritises the lowest cost cells to be searched first, generally attempting to lower node expansion by a small value, saving small amounts of processing time.

The path cost from start is more advanced than the value in the A\* search, it is based on the straight line centre to centre distance between the analysed cell and the start cell. This brings a big advantage over the previous application, as the program has greatly reduced occurrences of equivalent cost choices, being able to make more accurate path expansion choices.

**CUS2 – Advanced A\* Iteration (AAS)**

Advanced A\* completely overhauls how path cost is calculated.

Euclidean distance is used for both cost from start and distance to goal, creating much clearer choices of direction preference depending on the angle made from the both the start and goal cells. The lower cost cells are the ones which create a straight path between the start and goal cells, however there is much less chance of this algorithm getting lost or detoured as the majority of choices now deviate.

Implementation

The program is designed to run off a command line with passed in arguments. The arguments get processed and evaluated, returning error messages if not understood.

Features

* ASCII Map representations of outputs in search path
* Ability to count node expansions for every algorithm
* Each algorithm returns a stack of cell objects which makes up the path
* Series of tests which can output to either console or file
* Every algorithm has a built in backtracking ability to revisit the last expanded node once all options are exhausted
* Every algorithm tracks visited cells to improve performance
* Every algorithm always finds a solution

Bugs

* Map files must be valid, or program will not run as it should

Missing

**Yet to implement:**

* Output of a solution path in the form of directions, eg: right; up; up; down; ….

Need to analyse the returned stack and process a list of directions from the changes in cell position for each element of the stack. Can use one function to work any stack return by any algorithm.

Research

**Euclidean Distance Heuristic**

Conclusion

Resources

References