Using Implementations of Various Searching Methods to Solve 'Pacman' Maze Problems

Jordan Pottruff

I. Introduction

This report is an overview of my implementation of a "Pacman Search" program that determines a path from a pacman player to a food pellet within a maze. I wrote the program in Java due to my familiarity with designing data structures and algorithms using the language's object oriented structure. I also wanted to design the program using design patterns I had previously implemented in Java in order to make my code cleaner and more capable of meeting the objectives of the project.

II. Program Design

The solution uses a Search class in order to follow the requirement that a unified top-level search routine be implemented that is capable of handling different search behaviors. These behaviors are attached to the Search class using a strategy pattern, where the Search class holds a SearchBehavior (interface) object that can be polymorphically assigned to any given class that implements SearchBehavior. To implement the interface, each search strategy must implement three methods:

- *setup(...)* - the initial staging before the search commences.
- *getNextNode(...)* - handles mechanic of getting next node from the frontier.
- *processChildNode(...)* - encapsulates what the search method does for each child node.

The *search* method in the Search class then uses the above method implementations from the assigned behavior to conduct the search. The *search* method also implements shared structures like looping, goal checking, and solution metadata capture.

Maze and Tile classes exist for keeping track of data stored at each position, as well as getting the neighbors for each position. Neighbors are found in the order left, down, right, and up. This order has a drastic effect on depth first search due to branching, and a minimal effect on all due to goal checking.

III. Search Implementations

A. Depth First Search

The *setup* method for DFS adds the starting position onto the frontier. Then, each call to *getNextNode* takes the node/tile from the top of the frontier, saves it, adds it to a "closed" list, and returns it. When *processChildNode* is given a child of the top node, it skips it if it has already been closed (visited) and otherwise sets the top node as its parent and adds it to the end

of the frontier. Through these behaviors, depth first search branches out to find a non-optimal solution using relatively little space compared to the other search algorithms.

B. Breadth First Search

The *setup* method for BFS adds the start position onto both the frontier and the closed lists. The *getNextNode* method simply removes and returns the node at the bottom of the stack. Then in *processChildNode* the children of each removed node are skipped if already closed and otherwise the removed node is set as its parent, it is added to the closed list, and its added to the frontier. The combination of these methods is able to find an optimal solution, but often requires a large amount of space and expands many nodes.

C. Best First Search (Greedy)

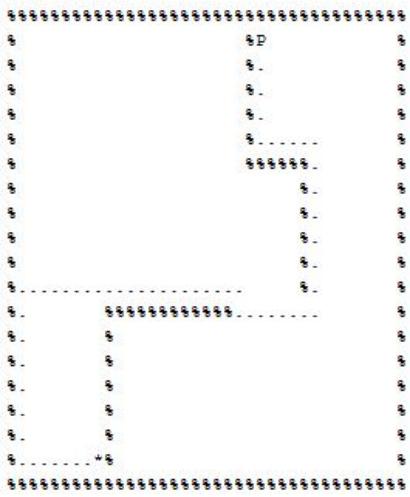
The *setup* method for Greedy just adds the start node to the frontier. Then, *getNextNode* is implemented by removing the node from the frontier with the smallest heuristic cost, calculated as the manhattan distance to the goal node. It then adds this node to the closed list before returning it. Then for processing children, *processChildNode* skips children that are closed or on the frontier, and otherwise sets the child's parent and adds to the frontier in the same fashion as in other search methods. These methods produce a solution that, while it may not be optimal, can generally be found in a lesser number of expanded nodes.

D. A Star Search

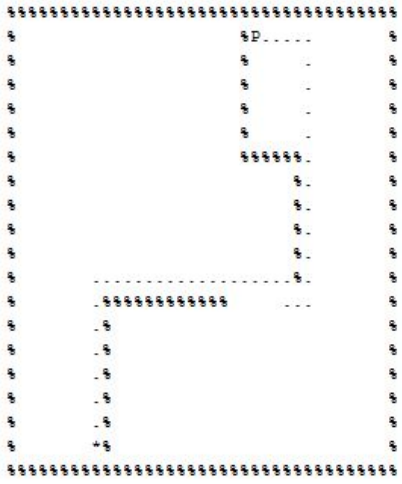
The *setup* and *getNextNode* methods for A Star are almost exactly the same as Greedy, but now the cost of any node is computed as the heuristic *plus the distance from the start*. To avoid poor performance, if the lowest cost is tied between two nodes, the node with the lowest heuristic only is preferred. The *processChildNode* method is complicated as it re-opens any previously closed nodes if a new shortest path to that node is discovered. To accomplish this, every node keeps track of its distance from the start, which is just its parents cost plus one. Together these methods produce an optimal solution while expanding a moderate number of nodes (in the given examples, at least).

IV. Example Solutions

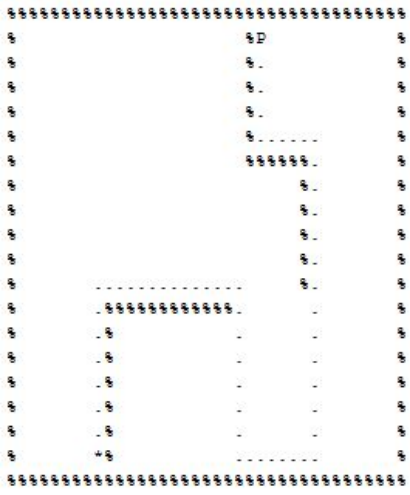
Depth First Search on 'open maze.txt'
Shortest path cost: 58
Number of Expanded Nodes: 530



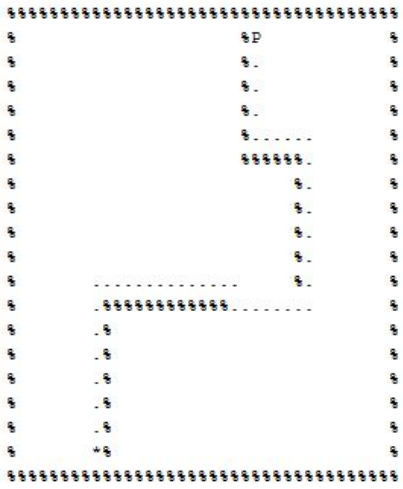
Breadth First Search on 'open maze.txt'
Shortest path cost: 44
Number of Expanded Nodes: 489



Best First Search on 'open maze.txt'
Shortest path cost: 56
Number of Expanded Nodes: 153



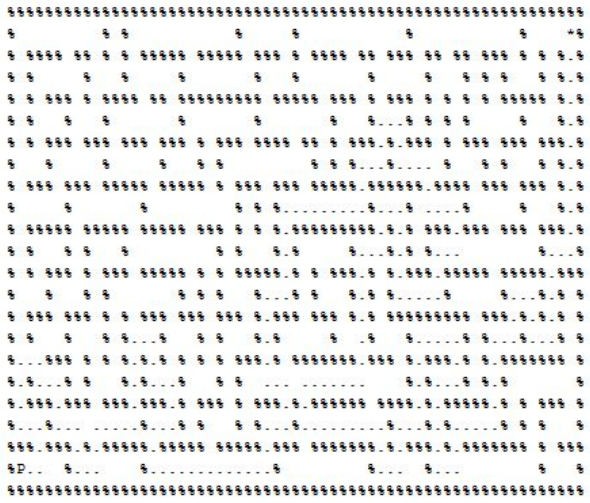
A* Search on 'open maze.txt'
Shortest path cost: 44
Number of Expanded Nodes: 178



Depth First Search on 'medium maze.txt'

Shortest path cost: 204

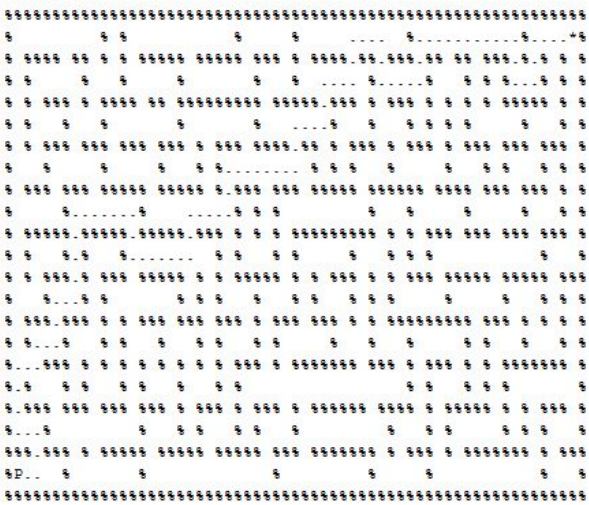
Number of Expanded Nodes: 284



Breadth First Search on 'medium maze.txt'

Shortest path cost: 94

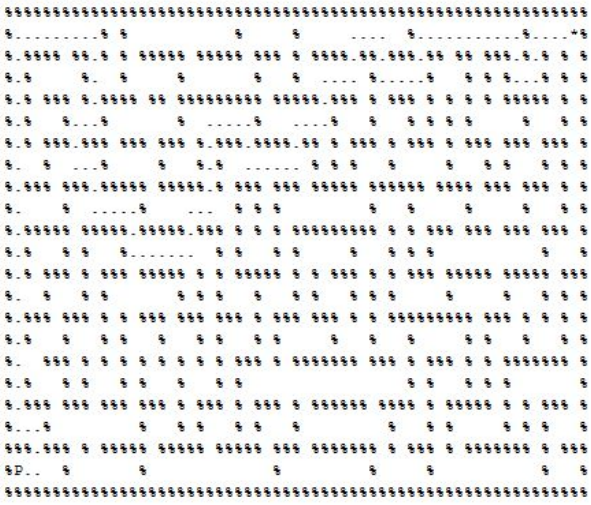
Number of Expanded Nodes: 603



Best First Search on 'medium maze.txt'

Shortest path cost: 118

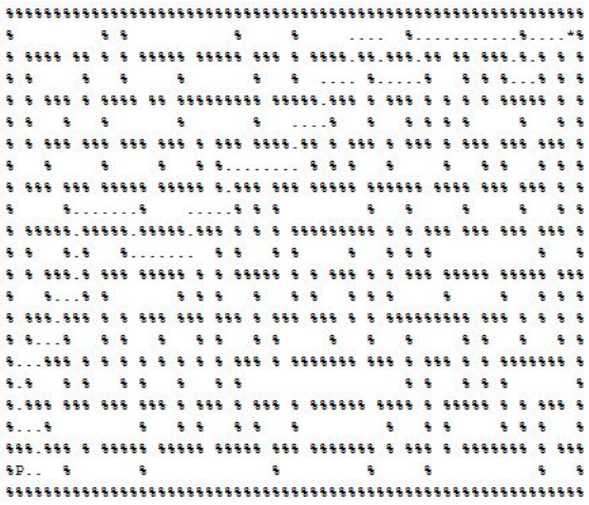
Number of Expanded Nodes: 133



A* Search on 'medium maze.txt'

Shortest path cost: 94

Number of Expanded Nodes: 300



Depth First Search on 'large maze.txt'

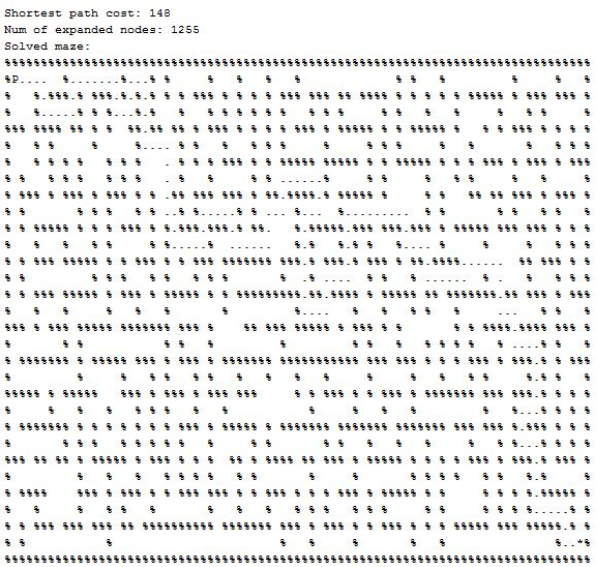
Shortest path cost: 474

Number of Expanded Nodes: 1070

Breadth First Search on 'large maze.txt'

Shortest path cost: 148

Number of Expanded Nodes: 1255



Best First Search on 'large maze.txt'

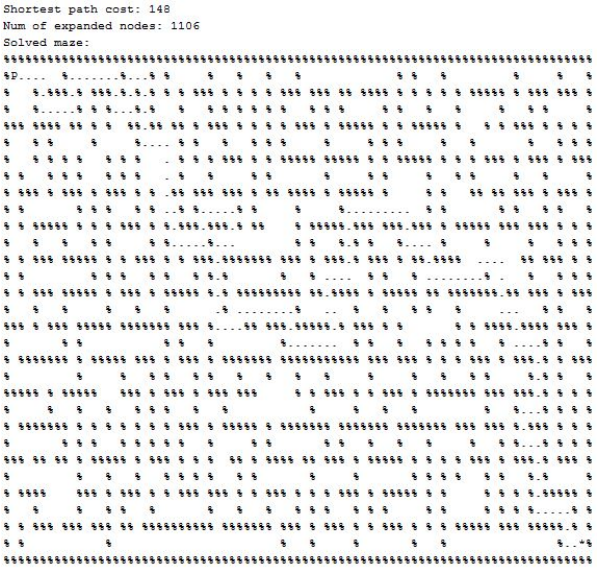
Shortest path cost: 222

Number of Expanded Nodes: 288

A* Search on 'large maze.txt'

Shortest path cost: 148

Number of Expanded Nodes: 1106



V. Contribution Outline

All elements of the assignment were completed by myself, Jordan Pottruff.