CS440 - Artificial Intelligence
Assignment 1 (3 Credit Hours)

Jon Reynolds (jdrynld2)
Matthew Krikorian (krikorn2)
Patrick McMahon (pfmcmah2)

September 26, 2017

1 Abstract

In this assignment we explore the different search algorithms that can be used to traverse a maze, and analyze the optimality's and general performance between the different implementations under a very heavy scope. In section 1.1 (basic pathfinding), we analyzed four different pathfinding algorithms. Specifically, Depth-first search, Breadth-first search, Greedy best-first search, and A* search. In section 1.2 (search with multiple dots), we were tasked with developing our own heuristic in the implementation of our A* search algorithm. There are also multiple goal positions in section 1.2 as well, which is why coming up with an admissible heuristic to optimally traverse the mazes was paramount.

2 Basic Pathfinding

In this part of the assignment we implemented four different search algorithms to analyze cost, number of nodes expanded, and overall solution optimality. We were given three different mazes to run our search algorithms on, demonstrating to us the potential pros and cons of each algorithm. **Breadth-first search** is a search algorithm where the root node is expanded first, then all of the root-notes children are expanded, and so forth until the goal successor (or state) is found.

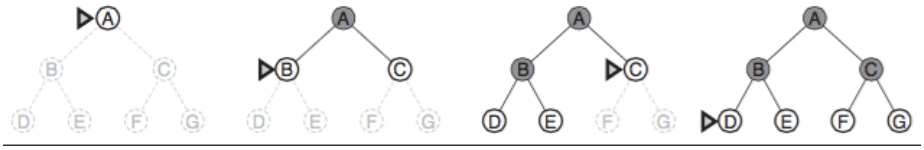

Figure 1: Breadth-first search algorithm, *Russell and Norvig*

Depth-first search is a search algorithm where the deepest node currently in the frontier is always chosen to be expanded with the hope of finding the goal state. If the goal state is not found once the algorithm finds a node with no successors, the algorithm will then backtrack back to the next deepest node that still has unexplored successors.

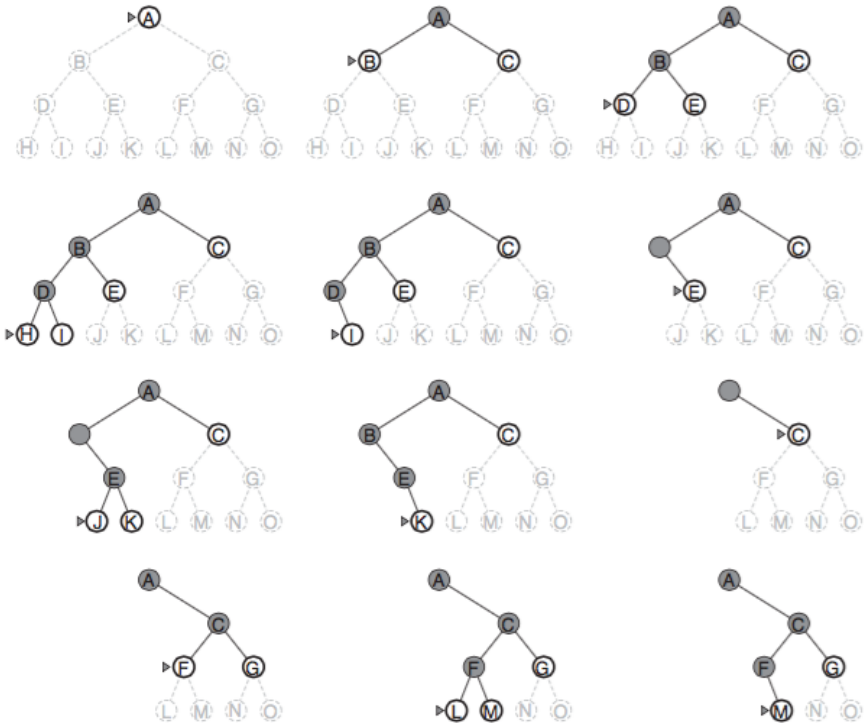

Figure 2: Depth-first search algorithm, *Russell and Norvig*

Greedy best-first search is a search algorithm that always chooses to go down the path that has the lowest cost with respect to the other neighbors of the current node it is on. The name greedy implies that it does not care about the actions of the future, but rather only chooses it's path on the best choice to it based on immediate cost. Greedy algorithms evaluate the nodes they are going to travel to using a heuristic function. This heuristic does not use past or future information to make any decisions, which sets it apart from both the BFS and DFS search algorithms. The algorithm chooses the next node it will travel to directly from the heuristic function it is given, so the choosing function $f(n)$ can be modeled as $f(n) = h(n)$ where $h(n)$ is the heuristic function.

Finally, **A* search** is a search algorithm where the function used to evaluate which node the algorithm will travel to next, $f(n)$, is dependent on both the cost from the start node to the goal node, $g(n)$, and the cost to get from the current node to the goal node, $h(n)$. The node selection function is actually modeled as the direct sum of $h(n)$ and $g(n)$, giving us $f(n) = h(n) + g(n)$. This algorithm is thought of as the algorithm amongst the four that often provides the most savings, and can be optimized heavily through a very good heuristic function $h(n)$.

2.1 Our Code, Explained

Our assignment is broken up into a *src* folder which holds all of our *.cpp* files, our *include* folder holds all of the *.h* files that we wrote, our *build* folder holds all of the *.o* files that are used in compilation, our *bin* folder contains all of the *.exe* files that we use to execute our search algorithms, and the *tex* folder holds all of the *.tex* that was used to compile the pdf that we submitted.

2.1.1 *frontier.cpp* and *frontier.h*

The file which handles keeping track of the frontier is *frontier.cpp* and *frontier.h*. Here we define a FrontierNode that acts as a wrapper for nodes that are kept in the running solution of the maze. The struct is defined as below.

```
struct FrontierNode{
    Node* node;
    Node* prevNode;
    FrontierNode* next;
    FrontierNode* prev;
    int pathCost;
    int heuristic;
};
```

During traversal, FrontierNodes are kept in a doubly linked-list so that adding and removing nodes that are deemed to not be a part of the solution can be removed easily. For example, when we perform Depth-first search, FrontierNodes are added to the linked list until the algorithm realizes there is no deeper node to traverse to, in which case we back-track and remove the last FrontierNode from the linked list and backtrack back to a FrontierNode where *prev* points to a FrontierNode that wraps a Node that has an unexplored Node. The FrontierNode struct also keeps track of the pathCost for the given node, as well as a value for that FrontierNode based on the heuristic function being used. In *frontier.cpp* and *frontier.h*, we define a Frontier class that holds all of the functions used to modify the frontier while traversing the maze.

```
class Frontier {
public:
    Frontier();
    ~Frontier();
    void push_back(Node* node, Node* prevNode, int val=0, int cost = 0);
    void push_front(Node* node, Node* prevNode, int val=0, int cost=0);
    FrontierNode* pop_back(std::unordered_map<Node*, Node*>& history);
    FrontierNode* pop_front(std::unordered_map<Node*, Node*>& history);
    FrontierNode* pop_min(std::unordered_map<Node*, Node*>& history);
    FrontierNode* getHead();
    FrontierNode* getTail();
    bool empty();
    FrontierNode* find(Node* node);
    void update(FrontierNode* fnode);
```

```
private:
    FrontierNode* head;
    FrontierNode* tail;
    int size;
    std::unordered_map<Node*, FrontierNode*> nodeMap;
```

The various push and pop functions in our Frontier class are used to easily manage the stack and queue when implementing DFS and BFS. When implementing DFS, we used a stack data structure to easily remove the most recently added frontier nodes from the top of the stack. For Greedy and A* search, *pop min* is used to find the lowest cost option for finding the next node based on each algorithms node choosing function.

2.1.2 *maze.cpp* and *maze.h*

```
class Maze {

public:
    Maze(std::string filename);
    ~Maze();
    Node* getStart();
    std::vector<Node*>* getGoals();
    void printSolution();
    std::string getName();

private:
    std::string name;
    Node* start;
    std::vector<Node*>* goals;
    Node*** maze;
    int w, h;
};
```

The purpose of *maze.cpp* and *maze.h* in our assigment is to take in an inputted maze text file, parse it, and allocate a 2-dimensional matrix which holds all the information related to the contents of the maze. After extracting the contents of the maze and putting it in a 2-dimensional array, the maze class writes the data in the array into Nodes which store the essential information of each position of the maze. This class can be thought of as the maze creation class, initializing all of the Nodes that are going to be used in the *main.cpp* file to build the frontier.

2.1.3 *node.cpp* and *node.h*

```
class Node {

public:
    Node(int x, int y);
    ~Node();
    std::vector<Node*>* getNeighbors();
    void addNeighbor(Node*);
    bool isVisited();
    void visit();
    int getX();
    int getY();

private:
    int x, y;
    std::vector<Node*>* neighbors;
    bool visited;
};
```

The Node class is the basis of how the maze is traversed and information relevant to the maze is stored. Every node is initialized with an x and y coordinate value, the neighbors of the given node in the maze stored in a vector of Node pointers, and a boolean value that determines whether any given search algorithm has already visited that Node. The Node class along with the FrontierNode class are the foundational Nodes used to traverse the maze and store the solution that is to be printed after the algorithm is done searching for the goal state.

2.1.4 *maze.cpp*

This file is the body where all of the Nodes and FrontierNodes are processed and the main algorithms are implemented. This file is where the functions to manage the stack for Depth-first search is managed as well as the queue for Breadth-first search are called and the overall algorithm for those two algorithms is implemented. This file is also where the solution frontier for both Greedy and A* Search are maintained and handled, and the manhattan distance heuristic is used to properly calculate the next Node to be traversed. We've also implemented the backtracking in this file which handles re-visiting the last Node with unexplored neighbors. After each maze is traversed and a solution is found, this file then goes and prints the respective solutions for each maze.

2.2 Depth-first Search Outputs

Below are the outputs we generated for each maze using the Depth-first search algorithm.

2.2.1 Medium Maze

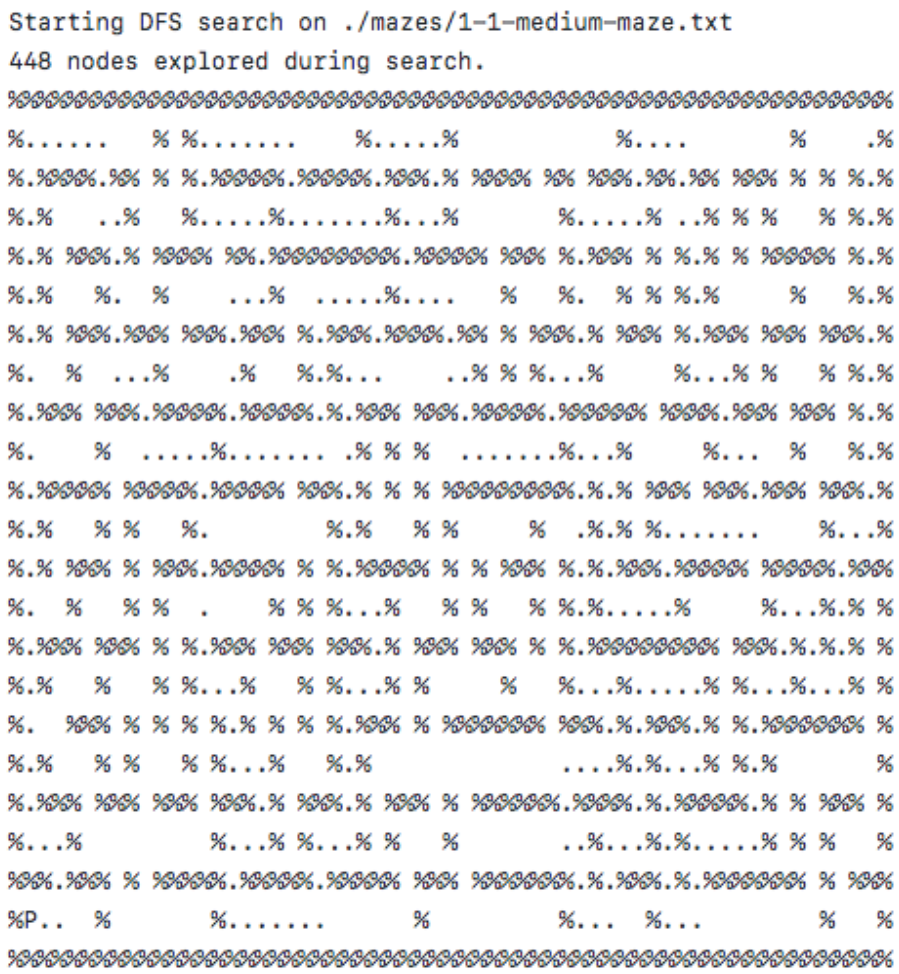

Figure 3: Depth-first search on the medium maze

2.2.2 Big Maze

```
Starting DFS search on ./mazes/1-1-big-maze.txt
967 nodes explored during search.
```

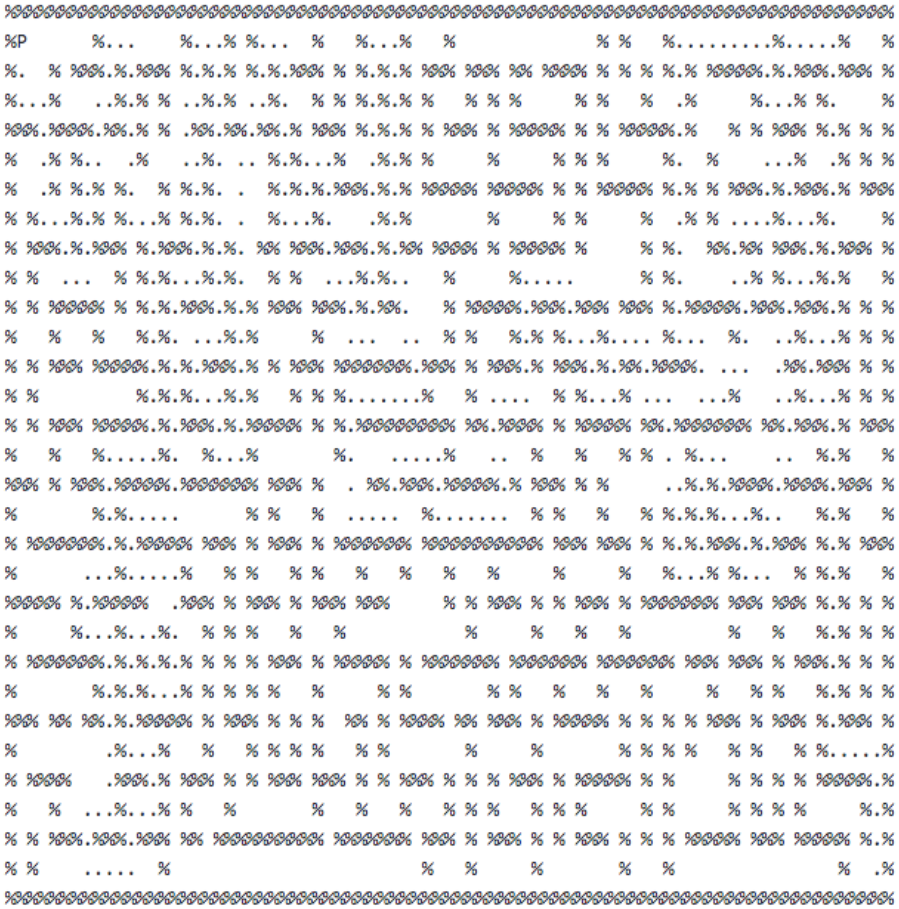

Figure 4: Depth-first search on the big maze

2.2.3 Open Maze

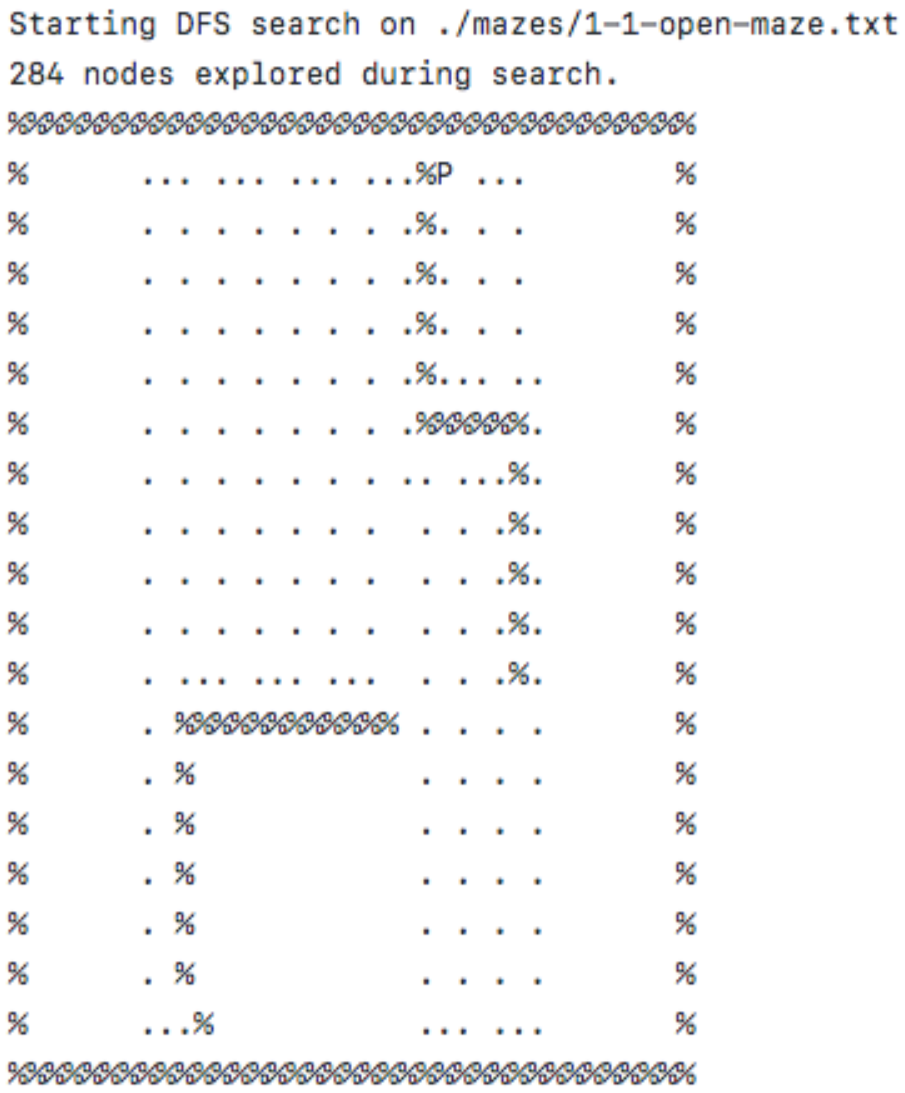

Figure 5: Depth-first search on the open maze

2.3 Breadth-first Search Outputs

Below are the outputs we generated for each maze using the Breadth-first search algorithm.

2.3.1 Medium Maze

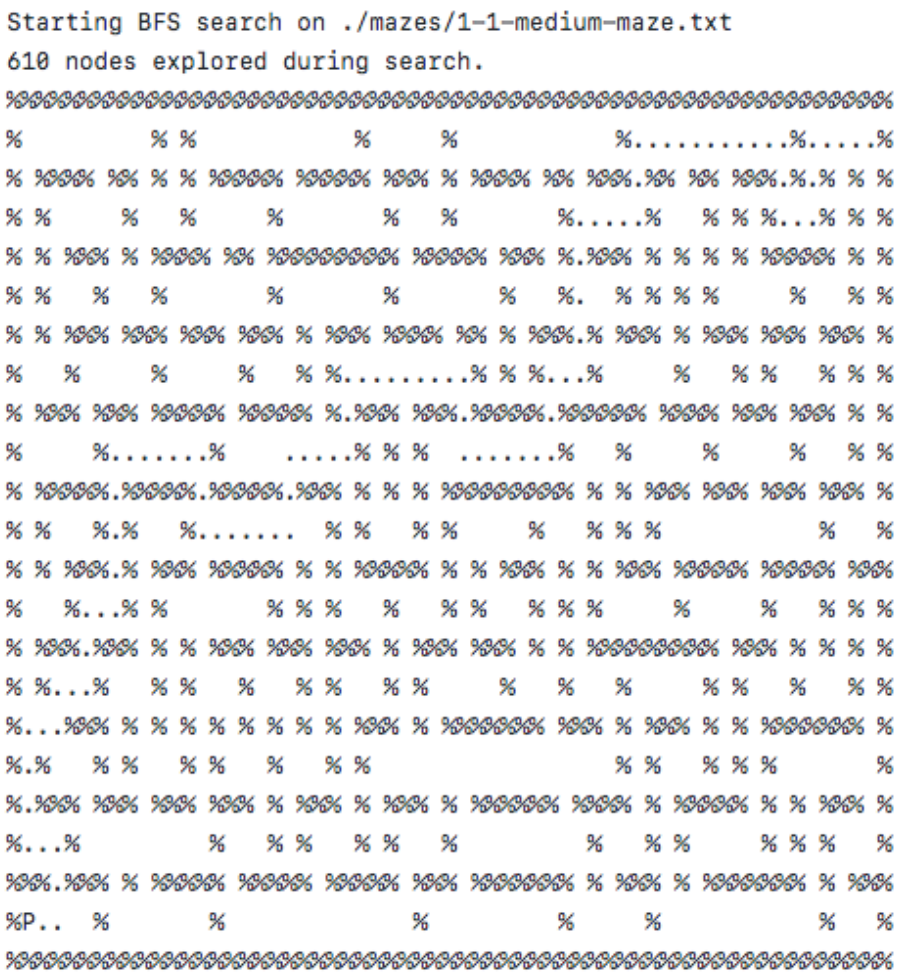

Figure 6: Breadth-first search on the medium maze

2.3.2 Big Maze

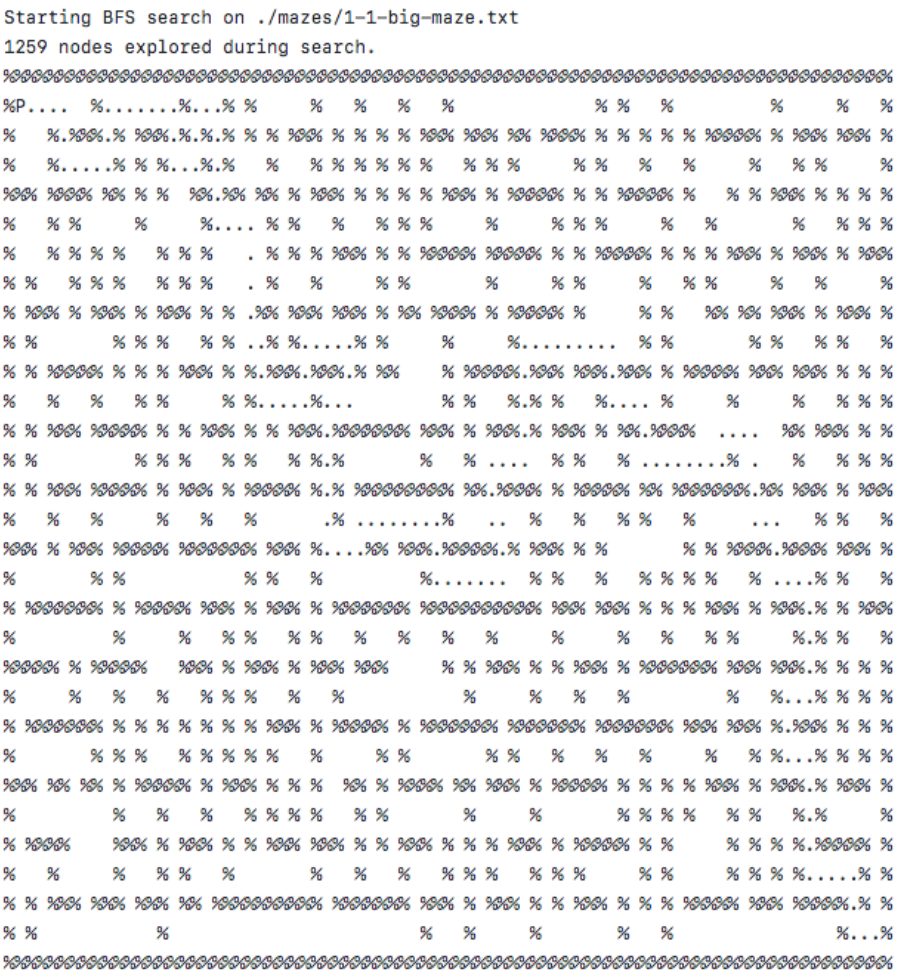

Figure 7: Breadth-first search on the big maze

2.3.3 Open Maze

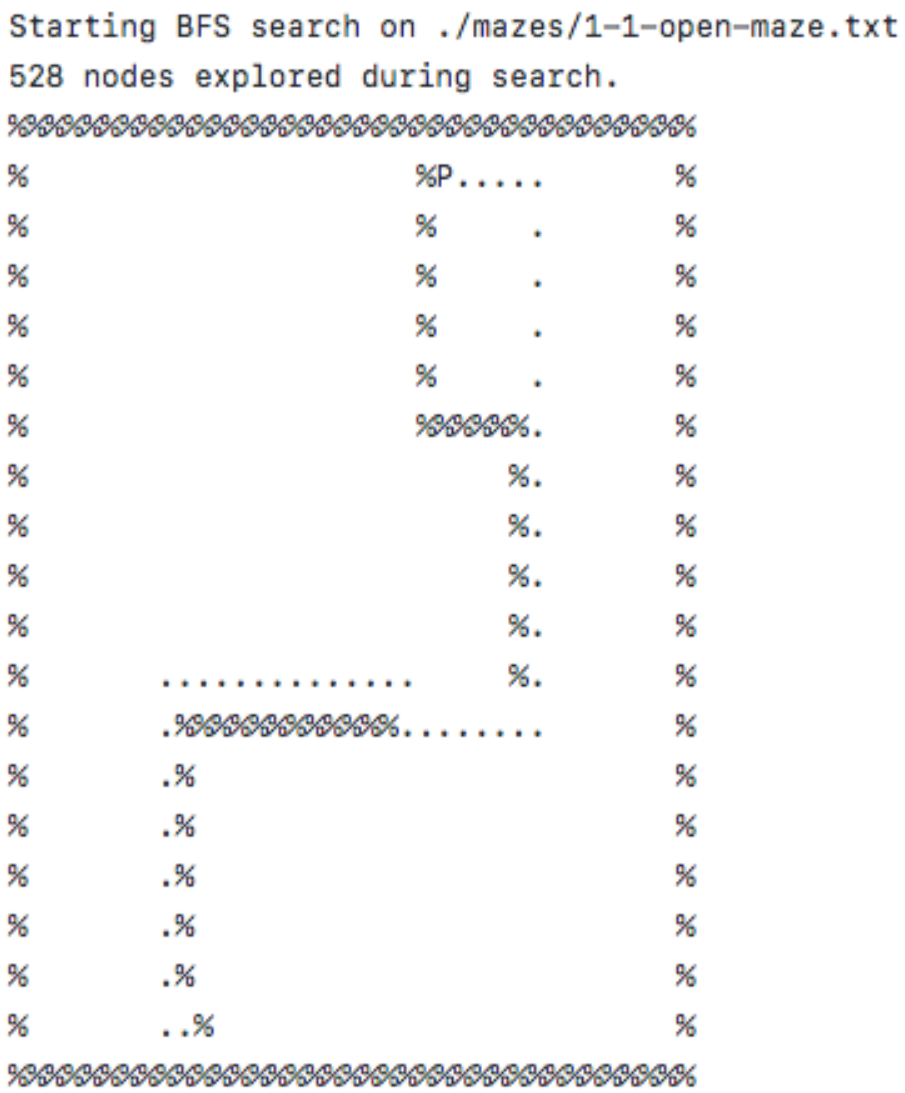

Figure 8: Breadth-first search on the open maze

2.4 Greedy Search Outputs

Below are the outputs we generated for each maze using the Greedy search algorithm.

2.4.1 Medium Maze

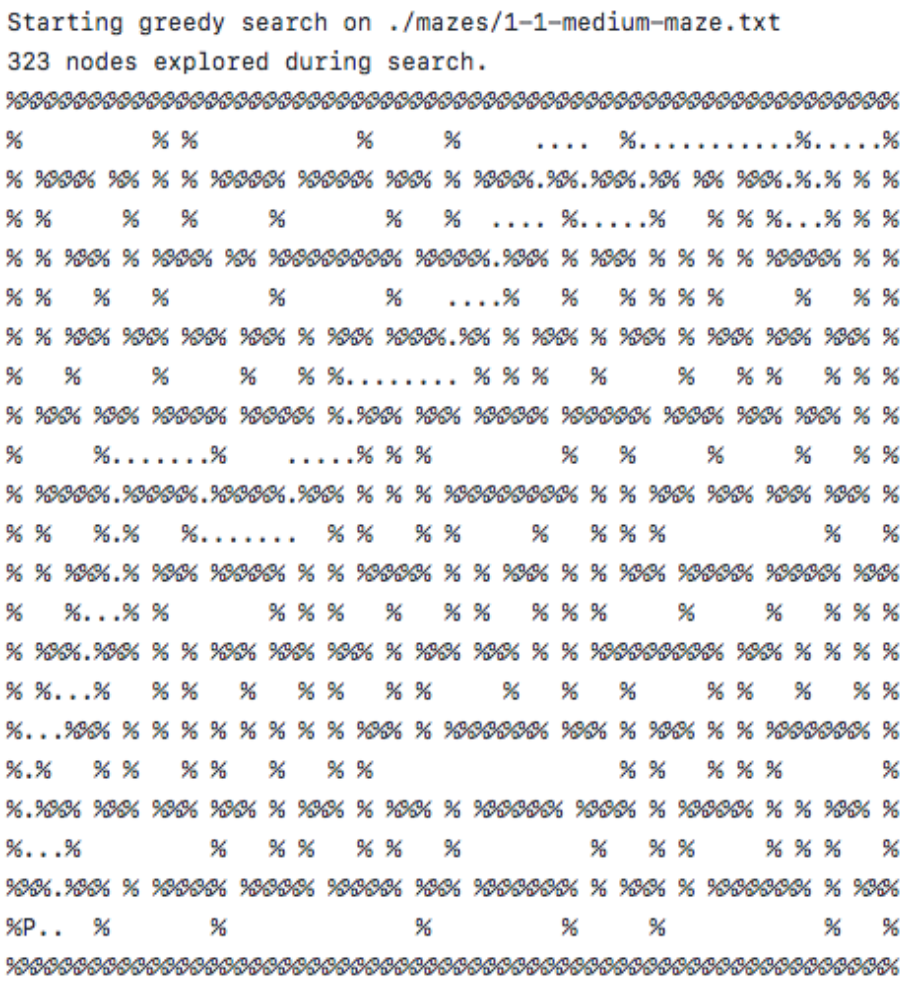

Figure 9: Greedy search on the medium maze

2.4.2 Big Maze

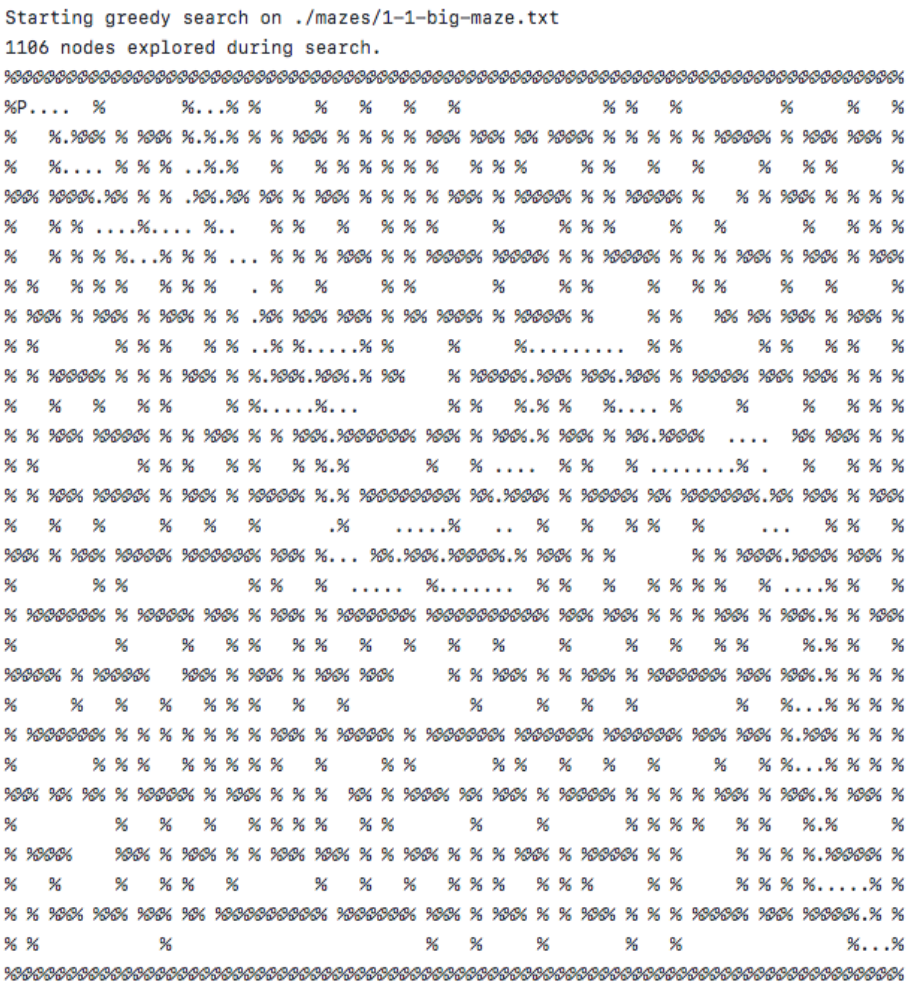

Figure 10: Greedy search on the big maze

2.4.3 Open Maze

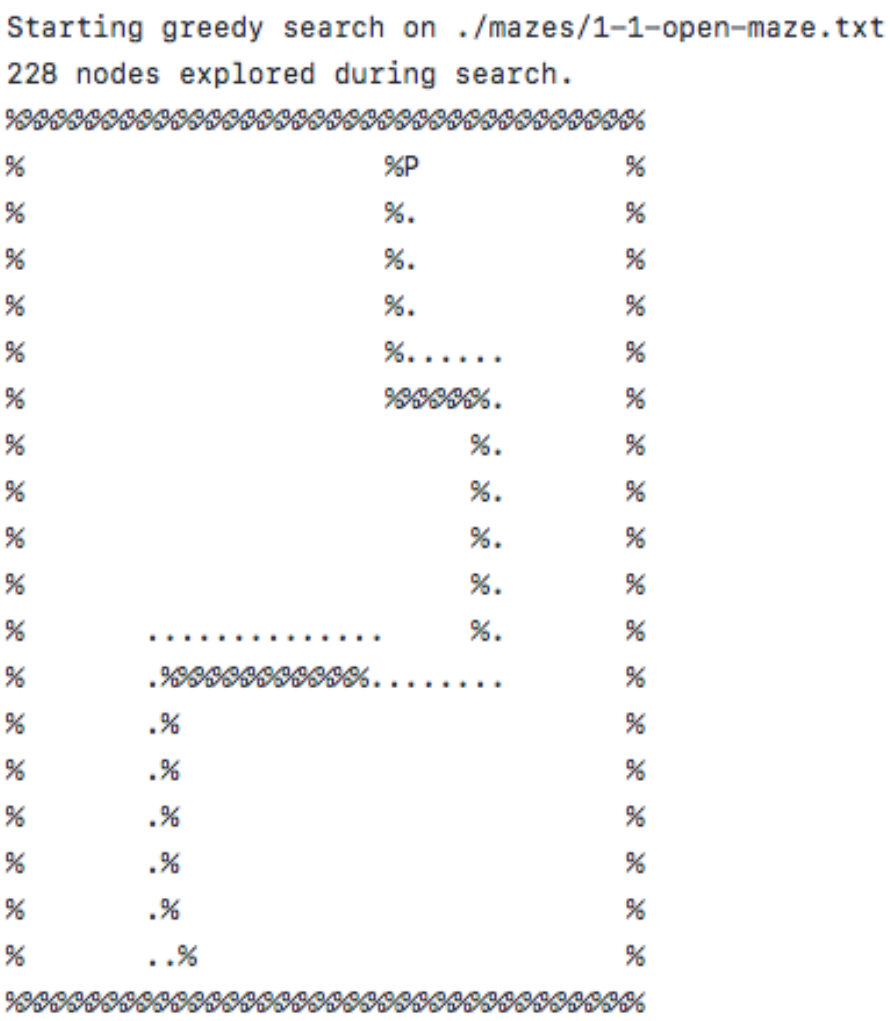

Figure 11: Greedy search on the open maze

2.5 A* Search Outputs

Below are the outputs we generated for each maze using the A* search algorithm.

2.5.1 Medium Maze

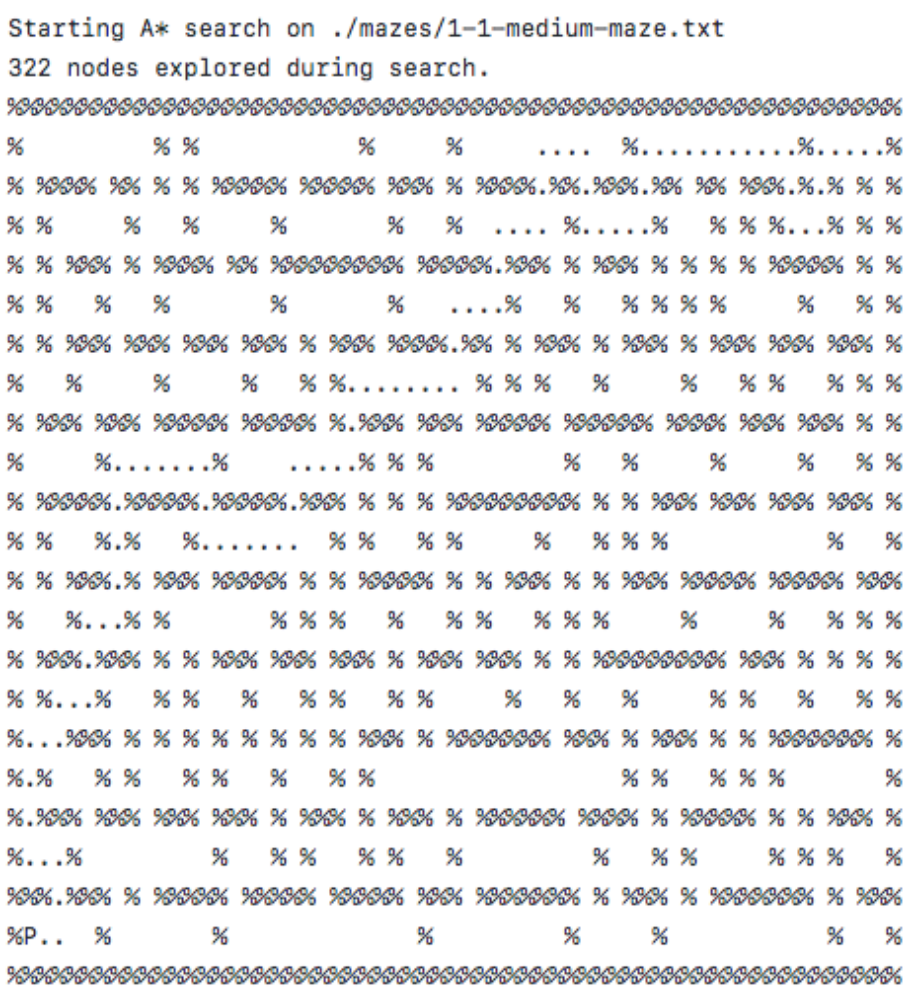

Figure 12: A* search on the medium maze

2.5.2 Big Maze

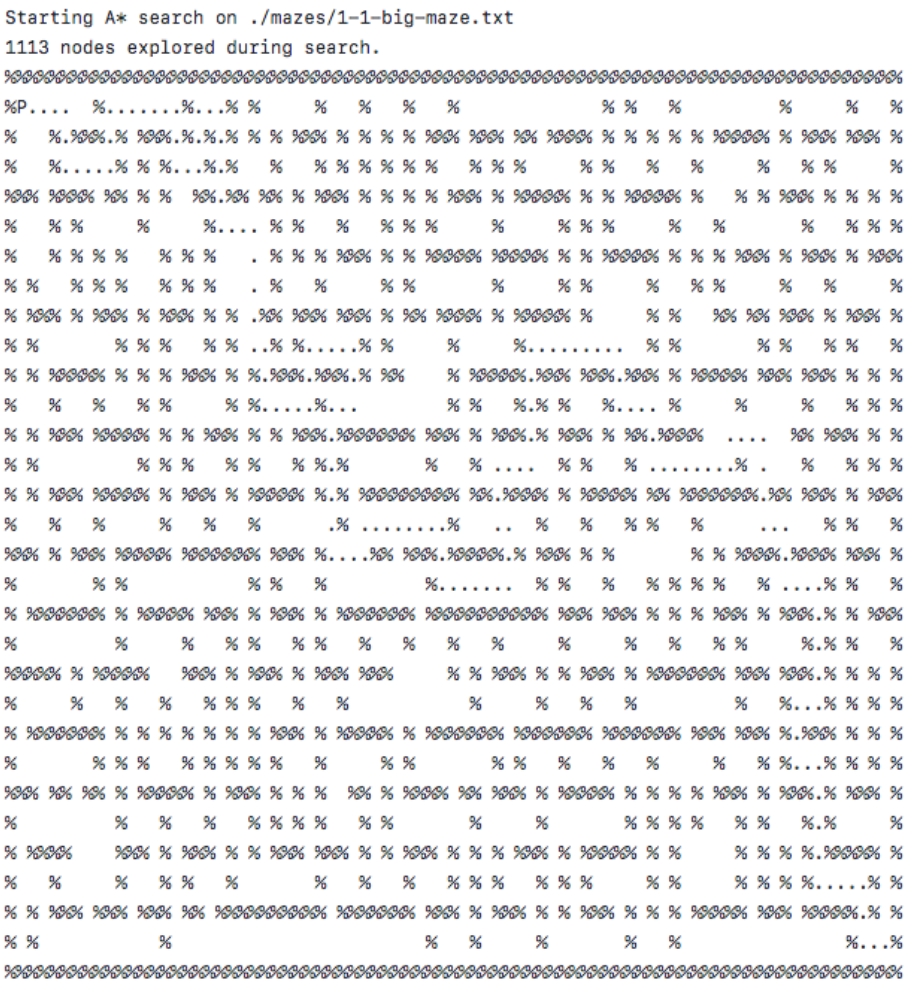

Figure 13: A* search on the big maze

2.5.3 Open Maze

Figure 14: A* search on the open maze