# CS 440 MP1
# Section Q4
# 4 Credits

Christian Howard
howard28@illinois.edu

Luke Pitstick
pitstck2@illinois.edu

Liuyi Shi
liuyis2@illinois.edu

**Abstract**

Within this report, an analysis was performed to compare and contrast a set of search algorithms, Breadth-First Search (BFS), Depth-First Search (DFS), Greedy Search, and A*. The analysis showed Greedy Search was not very robust, BFS was reliable but slow, DFS was fast but non-optimal generally, and A* tended to perform well with respect to both run-time and path cost. Heuristics were also designed for A* such that an agent could tackle the problem of touching multiple goal points with minimal moves. The heuristics designed ranged from a couple based on the Convex Hulls of the goal points and agent's location to a heuristic that blended a non-admissible and admissible heuristic together to achieve a balance between optimality and efficiency. After investigating multiple goal case, search algorithms were used to guide an agent to solve the Sokoban puzzle. Using both BFS and A* with a suitable heuristic, a set of Sokoban puzzles were solved and the associated statistics for each method against each puzzle were produced. Both A* and BFS solved the puzzles in an optimal number of moves, but A* was significantly faster for the tougher puzzles.

# Contents

# 1 Part 1 - Multi-Goal Search

Within this section, we investigate using search algorithms to tackle single-goal and multi-goal objectives. To handle this problem, the provided maze is processed into a graph, $G_m$, that can be used to figure out what actions a given agent can take and when they reach some goal node. To model the motion of some agent, we define the state, $\boldsymbol{x}$, as the following:

$$\boldsymbol{x} = (p, b_1, b_2, \cdots, b_n)$$

where $p$ is an index representing the graph node the agent is on and $b_k$ represents a Boolean variable that is 1 (true) when the agent has passed through the $k^{\text{th}}$ goal point at some point prior and 0 (false) otherwise. Note that this state is always implicitly dependent on the maze graph, $G_m$. We know an agent has completed a maze when it reaches a state that satisfies the following:

$$b_1 \wedge b_2 \wedge \cdots \wedge b_n = 1$$

To model the step by step motion of an agent, the transition function is defined as the following:

**Algorithm 1:** Transition Function - Multi-goal

**Data:** State, $\boldsymbol{x}$
**Data:** Action, $a$
**Data:** Maze Graph, $G_m$
**Data:** Goal Node Set, $V_g = \{p_1, p_2, \cdots, p_n\}$
**Result:** New State, $\boldsymbol{x}_n$

```
// Set the new state with the
// current state's boolean configuration
```
**for** $k \in \{1, 2, \cdots, |V_g|\}$ **do**
  $\boldsymbol{x}_n(b_k) = \boldsymbol{x}(b_k)$
**end**

```
// Update position in maze given the action a
```
$\boldsymbol{x}_n(p) = \text{getNextPosition}\,(\boldsymbol{x}, a, G_m)$

```
// Check if new position is in goal node set
// If so, find index associated with goal node and
// set corresponding boolean to 1
```
**if** $\boldsymbol{x}_n(p) \in V_g$ **then**
  Get $m \ni \boldsymbol{x}_n(p) = p_m$
  Set $\boldsymbol{x}_n(b_m) = 1$
**end**

**return** $\boldsymbol{x}_n$

Given the state representation and transition function described above, we are now able to implement and test our search-based agents. That said, we are now able to dive into solving each challenge of Part 1.

## 1.1 Part 1.1 - Basic Pathfinding

In this first challenge for Part 1, the goal is to implement search algorithms for navigating through a maze environment to a single goal location. For this challenge, we have implemented the following search algorithms:

- Depth-First Search
- Breadth-First Search
- Greedy Best-First Search
- A$^*$ Search

For Greedy and A$^*$, we have also set their Heuristic Function as the Manhattan Distance metric, defined as the following:

$$d_M(u, v) = |u - v|_1$$

where $u, v \in \mathbb{R}^2$ represent positions in the maze we want the distance between and $|\cdot|_1$ is the $L_1$ norm. Using this formula, the heuristic function will be:

$$h(n) = d_M(n, g)$$

where $n$ is the 2-D position of the current agent's position and $g$ is the 2-D position of the goal point. Given the above search strategies and heuristic, Table 1 and Table 2 show the stats between each method for each of the provided mazes:

| | Medium Maze | Big Maze | Open Maze |
|---|---|---|---|
| Depth-First Search | 124 | 474 | 59 |
| Breadth-First Search | 68 | 148 | 45 |
| Greedy Best-First Search | 68 | 222 | 77 |
| A$^*$ Search | 68 | 148 | 45 |

Table 1: Solution Path Cost

Note that Appendix A contains visuals for the solution paths for each case. Now as we can see in the tables, Depth-First Search (DFS) tended to arrive at a solution to a given maze quickly but at the expense of a sub-optimal path cost. We can note that Breadth First Search (BFS) and A* both, as expected, produced optimal path costs but it is interesting to note that for the Big Maze

| | Medium Maze | Big Maze | Open Maze |
|---|---|---|---|
| Depth-First Search | 198 | 1029 | 319 |
| Breadth-First Search | 345 | 1259 | 523 |
| Greedy Best-First Search | 77 | 311 | 27761 |
| A* Search | 202 | 1495 | 667 |

Table 2: Number of Nodes Expanded

and Open Maze, BFS actually reached the optimal solution faster. It seems feasible that the Manhattan distance heuristic would, at times, take A* in a path that would lead to dead ends, in turn requiring A* to expand more nodes than BFS would have to.

One other thing to note was the interesting characteristics of the Greedy Search. We can see that on one end, it reached an optimal result in a very low number of expanded nodes for the Medium Maze. On the other end, it reaches a sub-optimal path cost after expanding a huge number of states in the Open Maze. Greedy obviously lacks robustness and this can be suspected due to the fact it does not take into account how far it has come but only thinks about how far away it thinks it is. Given this one sided thinking, the agent can get too excited being "close" to the goal even though they may be winding through a lot of poor sub-paths.

## 1.2 Part 1.2 - Search with Multiple Dots

The goal of this part was to build heuristics for the A* algorithm that would allow an agent to tackle a set of multi-goal problems in a reasonable time frame. The heuristic being used for this section is one based on the Convex Hull.

Let us first define the distance metric, $d_{\text{ch}}(\cdot,\cdot,\cdot)$, as the following:

$$d_{\text{ch}}(x, y, \beta) = \beta|x - y|_2 + (1 - \beta)\,|x - y|_1$$

where $|\cdot|_1$ is the $L_1$ norm, $|\cdot|_2$ is the $L_2$, norm, and $\beta \in [0, 1]$ is a tuning parameter. Let us define $p$ as the current location of the agent and define $V_u = \{\hat{g}_1, \hat{g}_2, \cdots, \hat{g}_m\}$ as the set of $m$ unvisited goal points. Let us then compute the convex hull of $\{p\} \bigcup V_u$, defined as:

$$(V_{\text{ch}}, E_{\text{ch}}) = \text{ConvexHull}\left(\{p\} \bigcup V_u\right)$$

where $V_{\text{ch}}$ are the vertices on the convex hull of the input set and $E_{\text{ch}}$ is the edge set of 2-tuples such that $E_{\text{ch}} = \{(x, y) : x, y \text{ are ordered vertices of some edge}\}$. For this heuristic, we will remove an edge from $E_{\text{ch}}$ to ensure we only touch each point on the convex hull once. We will call the resulting edge set as $\hat{E}_{\text{ch}}$ and will define it in the following way.

If $p \notin V_{\text{ch}}$, $\hat{E}_{\text{ch}}$ will be equivalent to $E_{\text{ch}}$ with the largest edge from $E_{\text{ch}}$ removed and then an edge of minimal distance between $p$ and some $v \in V_{\text{ch}}$ added. If $p \in V_{\text{ch}}$, $\hat{E}_{\text{ch}}$ will be equivalent to $E_{\text{ch}}$ with the largest edge that has the vertex $p$ on it being removed. Given we have produced this new edge set $\hat{E}_{\text{ch}}$, the heuristic is then computed as:

$$h_{\text{ch}}(p) = \sum_{(x,y)\in\hat{E}_{\text{ch}}} d_{\text{ch}}(x, y, \beta) + |V_I| - \frac{20}{|V_u|}$$

where $V_I$ represents the set of goal points on the interior of the convex hull. This heuristic is admissible because, looking at the first term, the convex hull based distance using the edge set $\hat{E}_{\text{ch}}$ represents the shortest distance possible between points on the convex hull.

The secondary term is then just multiplying the number of goal points within the convex hull by the minimum distance of 1 to get to any neighboring goal point. This second term is a lower bound on the distance it will take to touch each of the interior goal points from any point along the convex hull. The last term removes from the distance based on the number of points the agent has yet to visit. This third term is meant to help add variation in the boolean portion of the state space while ensuring the heuristic is admissible. This last term essentially subtracts more when there are less goals left, making the distance to go appear less and in turn making a node get selected earlier. Now in all, these three terms together in turn under estimate the minimal distance the agent will need to travel to touch each goal point, making the heuristic admissible. The results using this heuristic can be found in Table 3 and Table 4.

| Tiny Search | Small Search | Medium Search |
|---|---|---|
| 39 | 153 | 207 |

Table 3: Solution Path Cost - A* with $h_{\text{ch}}(p)$

| Tiny Search | Small Search | Medium Search |
|---|---|---|
| 507 | 1,474,789 | 217,424,792 |

Table 4: Number of Nodes Expanded - A* with $h_{\text{ch}}(p)$

## 1.3 Part 1 Extra Credit - Sub-optimal Heuristics

### 1.3.1 Fast Convex Hull Heuristic

The heuristic in this section is another Convex Hull based Heuristic that modifies some of the logic used in the Part 1.2 definition that makes it not provably admissible. This heuristic can be defined as the following:

$$h_{\text{fch}}(p) = (1+\beta_1)\left(\sum_{(x,y)\in E_{\text{ch}}} |x-y|_2 - \max_{(x,y)\in E_{\text{ch}}} |x-y|_2\right) + \beta_2 |V_I|$$

where $\beta_1, \beta_2 \geq 0$ are tuning parameters, $E_{\text{ch}}$ is the unmodified convex hull edge set, as defined earlier, and $V_I$ represents goal points within the convex hull. For testing, $\beta_1 = \frac{1}{\sqrt{2}}$ and $\beta_2 = 2$. Table 5 and Table 6 show this heuristic produces great results in much faster speeds, compared to the admissible Convex Hull Heuristic from Part 1.2, while still achieving near optimal cost.

| Tiny Search | Small Search | Medium Search |
|---|---|---|
| 38 | 149 | 207 |

Table 5: Solution Path Cost - A* with $h_{\text{fch}}(p)$

### 1.3.2 Blended Heuristic

The idea of this heuristic was to blend two heuristics, one that is non-admissible and another that is admissible. The blending procedure implemented used the following formulation:

| Tiny Search | Small Search | Medium Search |
|---|---|---|
| 90 | 424,221 | 6,852,599 |

Table 6: Number of Nodes Expanded - A* with $h_{\text{fch}}(p)$

$$
\begin{aligned}
h_{\text{blend}}(p) &= (1-\beta)\, h_{MM}(p) + \beta h_{max}(p) \\
h_{\text{MM}}(p) &= \sum_{g \in V_u} |p-g|_1 \\
h_{\text{max}}(p) &= \max_{g \in V_u} |p-g|_1 \\
\beta &= \sigma\left(20\frac{|V_g| - |V_u|}{|V_g|} - 10\right) \\
\sigma(z) &= \left(1+e^{-z}\right)^{-1}
\end{aligned}
$$

Table 7 and Table 8 show the performance of this blended heuristic on the same mazes from Part 1.2. We can see that this Blended Heuristic achieved decent solutions for each of the mazes and did so after visiting a lot less nodes than the admissible heuristic from Part 1.2.

| Tiny Search | Small Search | Medium Search |
|---|---|---|
| 45 | 171 | 265 |

Table 7: Solution Path Cost - A* with $h_{\text{blend}}(p)$

| Tiny Search | Small Search | Medium Search |
|---|---|---|
| 1104 | 224,780 | 2,295,759 |

Table 8: Number of Nodes Expanded - A* with $h_{\text{blend}}(p)$

# 2 Part 2 - Sokoban

## 2.1 State and Transition Model Definitions

Within this section, the goal is to develop an agent that is capable of solving the Sokoban puzzle problem for a set of input puzzles. To handle this problem, the state representation has been defined as the following:

$$\boldsymbol{x} = (p, b_1, b_2, \cdots, b_n)$$

where in this case $p$ is the position of the agent, represented as a graph node index, and $b_k$ represents the position of the $k^{\text{th}}$ box's position, represented as a graph node index, within the puzzle environment. Note that, as before, the input puzzle is processed to construct a graph representation which is in turn an implicit part of the state and environment representation for this problem. To know when the agent has found a Goal State, the following condition must be met:

$$(b_1 \in V_g) \wedge (b_2 \in V_g) \wedge \cdots \wedge (b_n \in V_g) = 1$$

where $V_g = \{g_1, g_2, \cdots, g_n\}$ is the set of goal points within the puzzle we can place a box. The next step in formulating this problem is describing the Transition Model for the agent and what actions an agent can take based on its state. Algorithm 2 describes the Transition Model for Sokoban given our state representation, while Algorithm 3 refers to how we get the feasible action set for an agent given its current state. With these pieces described, we can readily approach solving Sokoban using the search methods discussed earlier.

**Algorithm 2:** Sokoban Transition Model Algorithm

**Data:** State, $\boldsymbol{x}$
**Data:** Action, $a$, assumed to be valid
**Data:** Maze Graph, $G_m$
**Result:** New State, $\boldsymbol{x}_n$

```
// Set the new state with the
// current state's box configuration
```
**for** $k \in \{1, 2, \cdots, |V_g|\}$ **do**
  $\boldsymbol{x}_n(b_k) = \boldsymbol{x}(b_k)$
**end**

```
// Update position in maze given the action a
```
$\boldsymbol{x}_n(p) = \text{getNextPosition}\,(\boldsymbol{x}, a, G_m)$

```
// Check if new position is in equivalent to
// the position of one of the boxes.  If so update
// the location of the box using the same action
```
**if** $\boldsymbol{x}_n(p) \in \{\boldsymbol{x}_n(b_1), \boldsymbol{x}_n(b_2), \cdots, \boldsymbol{x}_n(b_n)\}$ **then**
  Get $i \ni \boldsymbol{x}_n(p) = \boldsymbol{x}_n(b_i)$
  Set $\boldsymbol{x}_n(b_i) = \text{getNextPosition}\,(\boldsymbol{x}_n(b_i), a, G_m)$
**end**

**return** $\boldsymbol{x}_n$

**Algorithm 3:** Sokoban Action Set Retrieval Algorithm

**Data:** State, $\boldsymbol{x}$
**Data:** Maze Graph, $G_m$
**Data:** Goal Node Set, $V_g = \{g_1, g_2, \cdots, g_n\}$
**Result:** Action Set, $A$

```
// Get set of box positions
```
$B_p = \text{getBoxPositions}\,(\boldsymbol{x})$

```
// Initialize Action Set as the one based on the Connectivity
```
$A = \text{getActionsFromConnectivity}\,(\boldsymbol{x}(p), G_m)$

```
// Loop through possible actions and check they are valid
// given the rules of Sokoban
```
**for** $a \in A$ **do**
    `// Get the next node in the graph the`
    `// agent would be going to`
    $p_n = \text{getNextPosition}\,(\boldsymbol{x}, a, G_m)$

    `// Check if` $p_n$ `is the position of a box`
    **if** $p_n \in B_p$ **then**
        `// Update position of matching box using action` $a$
        `// and check if it is a valid position`
        $u = \text{getNextPosition}\,(p_n, a, G_m)$

        `// If new box position interferes with another box`
        `// or the wall, remove` $a$ `from the action set` $A$
        **if** $u \in B_p$ *or* $isWall(u, G_m)$ **then**
            $A \leftarrow A \setminus \{a\}$
        **end**
    **end**
**end**

```
// Return the desired Action Set
```
**return** $A$

## 2.2 Solving Sokoban with Breadth First Search

Using our state representation and the transition model for the agent, it becomes simple to apply BFS to tackling Sokoban problems. With this, the results of this BFS against the set of Sokoban puzzles provided can be found in Table 9. Animations of their solution, in the form of GIFs, can be found in the project zip with their identifiers as **bfs_sokoban*.gif**. As expected, BFS solved the Sokoban problems using how we modeled the state and environment, though Sokoban 4 did expand a considerable number of states.

| | Sokoban 1 | Sokoban 2 | Sokoban 3 | Sokoban 4 |
|---|---|---|---|---|
| Number of Moves | 8 | 144 | 34 | 72 |
| Nodes Expanded | 46 | 2,038,936 | 2,076,600 | 176,495,868 |
| Run Time | 0.13 s | 3.47 s | 4.70 s | 207.12 s |

Table 9: Sokoban solved by BFS

## 2.3 Solving Sokoban with A*

To tackle Sokoban, we go over a heuristic for A* and the results we achieve solving Sokoban. To keep things simple, the heuristic is defined as the following:

$$h_{\text{nearest}}(p) = \min_{b \in B_p} d_M(p, b) + \sum_{b \in B_p} \min_{g \in V_g} d_M(b, g)$$

where $B_p = \text{getBoxPositions}(\boldsymbol{x})$ and $V_g$ represents the set of goal point locations for the boxes. This heuristic is admissible. The first term, representing the distance the agent has left to move to reach the boxes, is underestimating the distance the agent has to go since it is only looking at the distance to the nearest box. The second term also acts as a sum of the lower bounds to the distance the agent would have to push a given box $b$ to some goal $g$. This is a lower bound in the second term, as formulated, because the nearest goal point to some box $b$ is not necessarily a goal point the box can reach, nor is that goal point necessarily one that can be used to solve the overall puzzle. The sum of these components together in turn produces an underestimate of the distance the agent has left to travel before completion of the puzzle, making the heuristic admissible.

With this, the results of this A* formulation against the set of Sokoban puzzles provided can be found in Table 10. Animations of their solution, in the form of GIFs, can be found in the project zip with their identifiers as **astar_nearest_sokoban*.gif**. Watching the animations, it is interesting to see how the agent tackled trickier puzzles like Sokoban 2 and Sokoban 4 where the solution requires more seemingly long term thinking, i.e. making sure moves you make now allow for you to make moves later on boxes that are more constrained. Fortunately, the heuristic managed to solve all these puzzles in a reasonable time

frame, though we expect there are superior heuristics for managing Sokoban 2 and Sokoban 4.

| | Sokoban 1 | Sokoban 2 | Sokoban 3 | Sokoban 4 |
|---|---|---|---|---|
| Number of Moves | 10 | 148 | 36 | 72 |
| Nodes Expanded | 36 | 517,584 | 146,259 | 11,404,258 |
| Run Time | 0.14 s | 2.49 s | 0.95 s | 40.16 s |

Table 10: Sokoban solved by A* using $h_{\text{nearest}}(p)$

## 2.4 Comparison of Results

By checking Table 9 and Table 10, we can draw some obvious comparisons. First, BFS always achieved an optimal result and A* was right near it, though it might have missed the absolute optimal solution due to the repeated state detection. Additionally, A* performed much faster than BFS in terms of number of nodes expanded. The run-time comparison between BFS and A* also followed that trend generally, though A* did not run as many multiples faster than BFS with respect to run-time as it did number of nodes expanded. This makes sense because the heuristic computation of A* is going to cost CPU time that BFS does not have to worry about.

# 3 Contributions

- Christian Howard
  - Built all the utility code, including code for reading and writing mazes, custom GIF wrapper for animating solutions, convex hull code, custom data structures, etc.
  - Built the state representation and transition model codes for both parts of the assignment
  - Built the OOP framework used to define search algorithms and heuristic functions in a flexible way
  - Built A* using template meta-programming to generalize to many search problem
  - Designed and built all the heuristics, successful or not, for A* for both parts of the assignment
  - Wrote and ran scripts to compile all the statistical data and generate output solutions and GIFs for each test case
  - Made custom tiles for use in making the GIF animations
  - Wrote the report
- Luke Pitstick
  - Built Depth-First Search code for Multi-Goal Search
  - Modified A* to have flag that can make A* into Greedy Search
  - Reviewed the report
  - Worked on sub-optimal algorithms for tackling Big Dots maze
- Liuyi Shi
  - Built Breadth-First Search code for Multi-Goal Search
  - Built Breadth-First Search code for Sokoban
  - Reviewed the report
  - Worked on sub-optimal algorithms for tackling Big Dots maze

# 4 List of Potential Extra Credit

- Animations for all solutions throughout the report, including Sokoban problems
- Extra sub-optimal heuristics for Part 1.2

# Appendices

## A Maze Solutions for Part 1.1

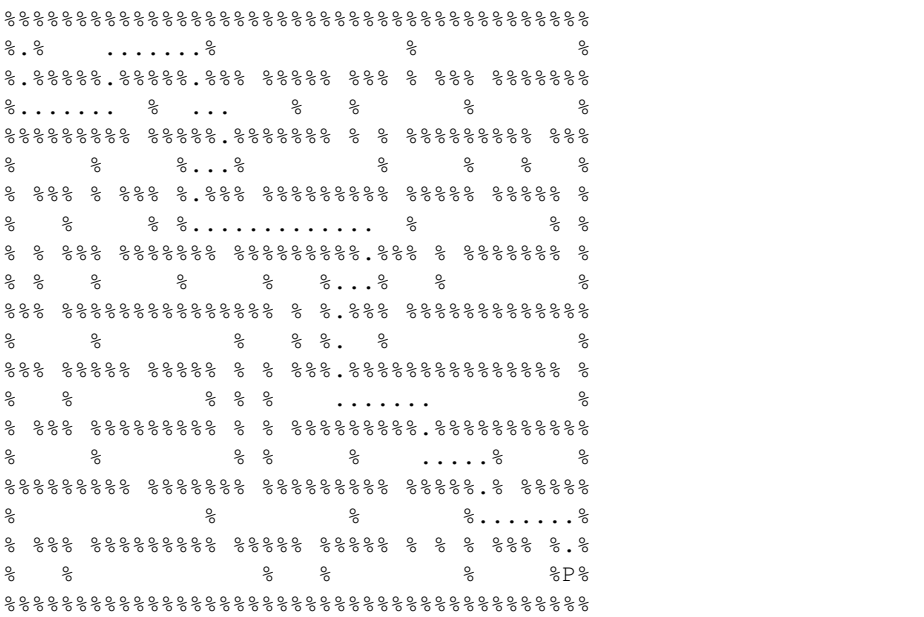

Figure 1: Solution to Medium Maze using BFS

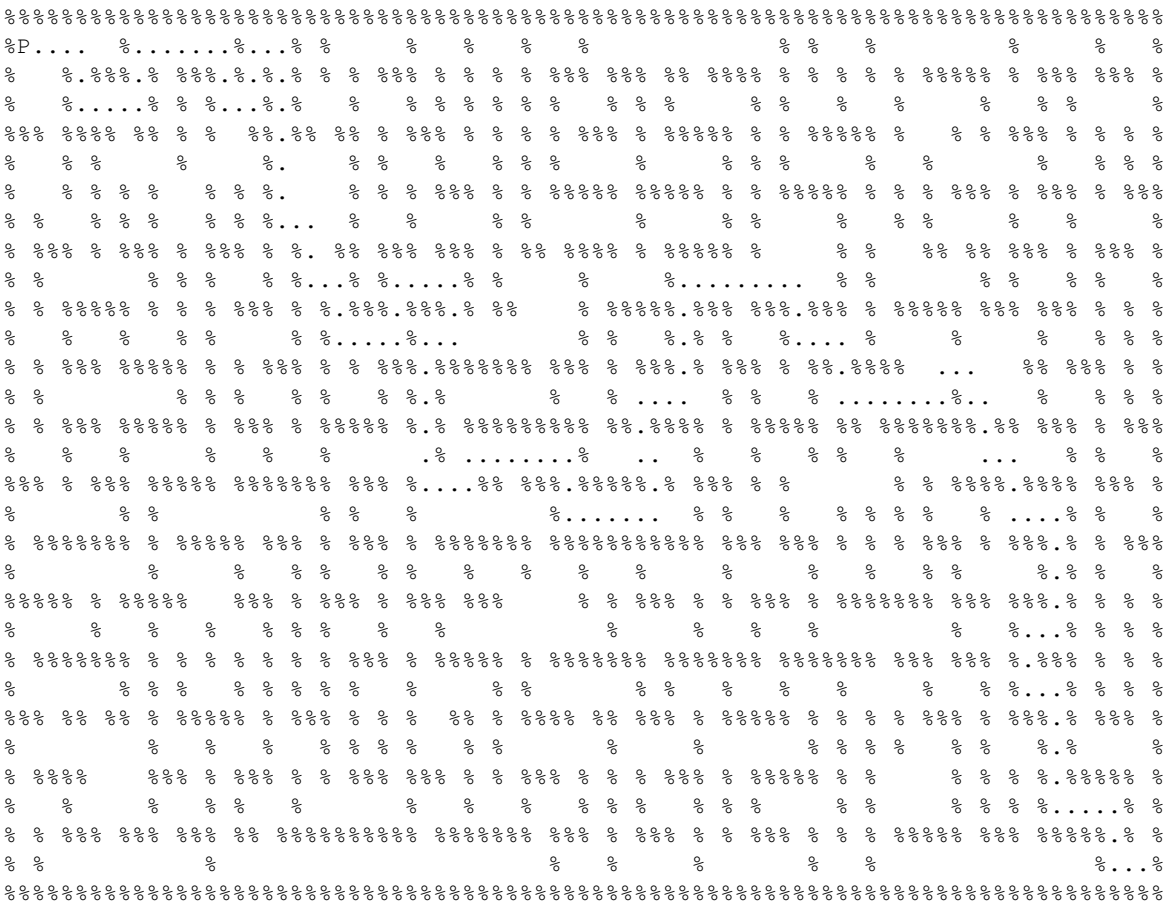

Figure 2: Solution to Big Maze using BFS

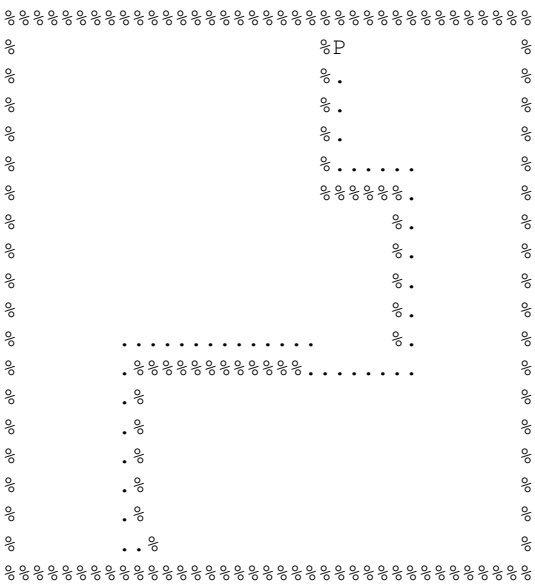

Figure 3: Solution to Open Maze using BFS

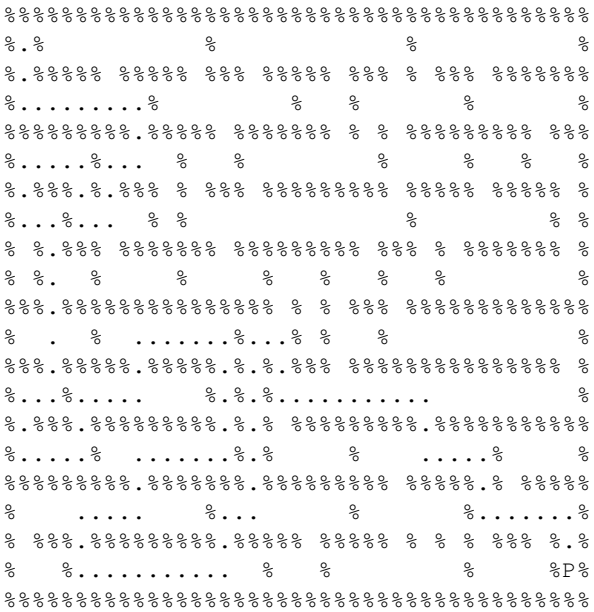

Figure 4: Solution to Medium Maze using DFS

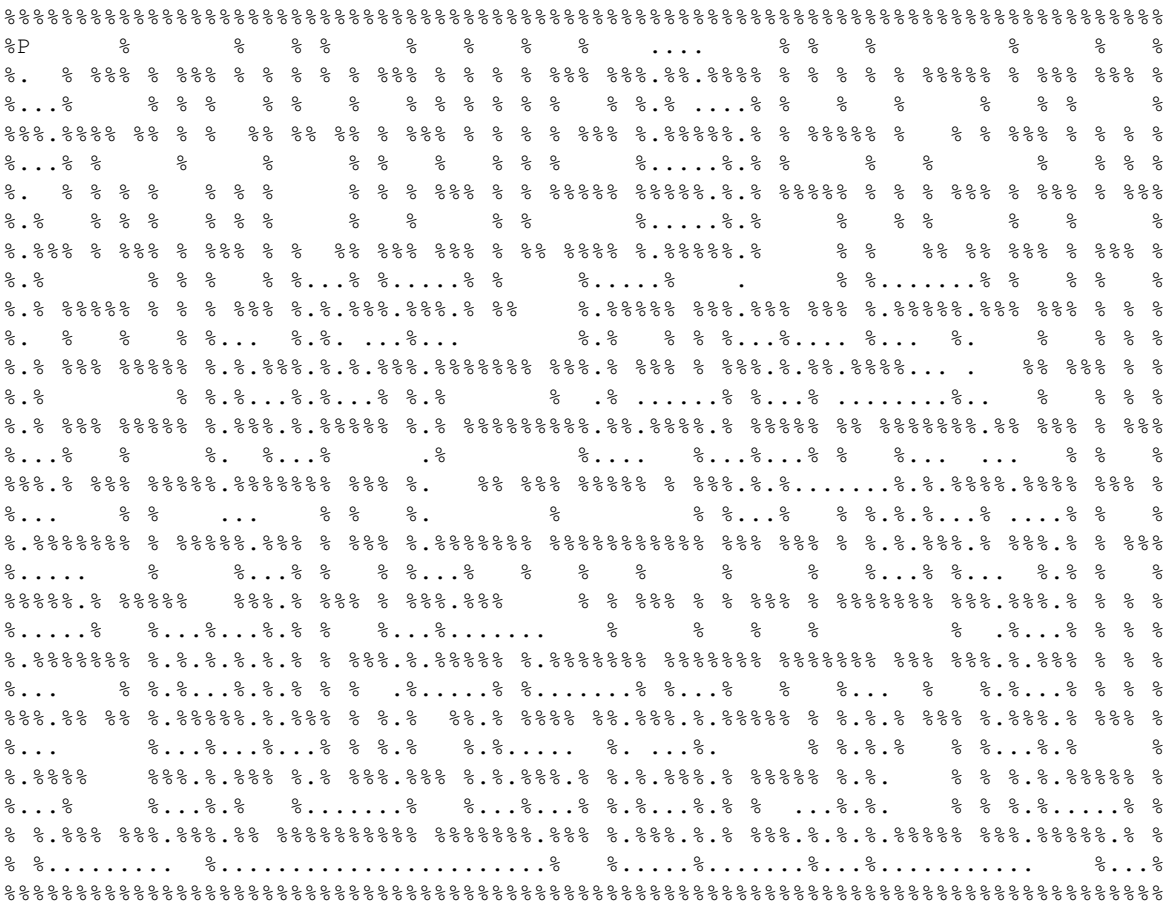

Figure 5: Solution to Big Maze using DFS

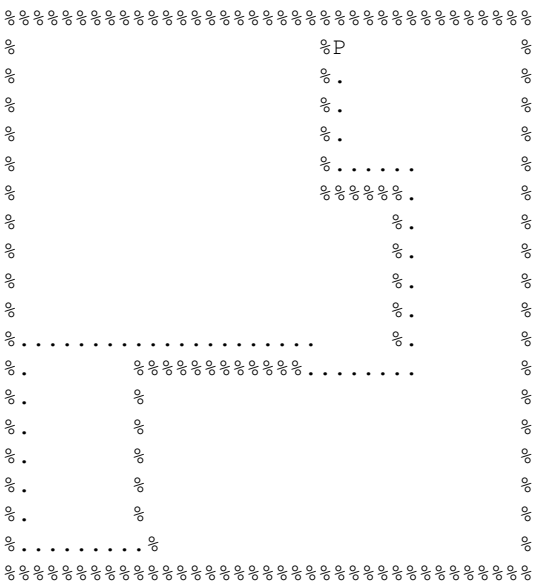

Figure 6: Solution to Open Maze using DFS

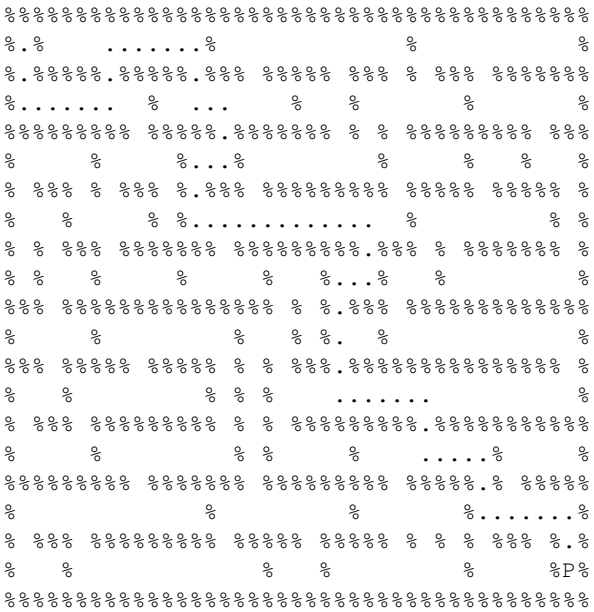

Figure 7: Solution to Medium Maze using Greedy

Figure 8: Solution to Big Maze using Greedy

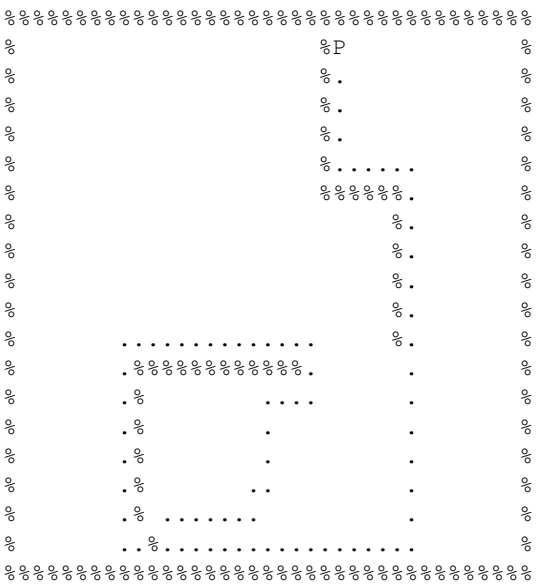

Figure 9: Solution to Open Maze using Greedy

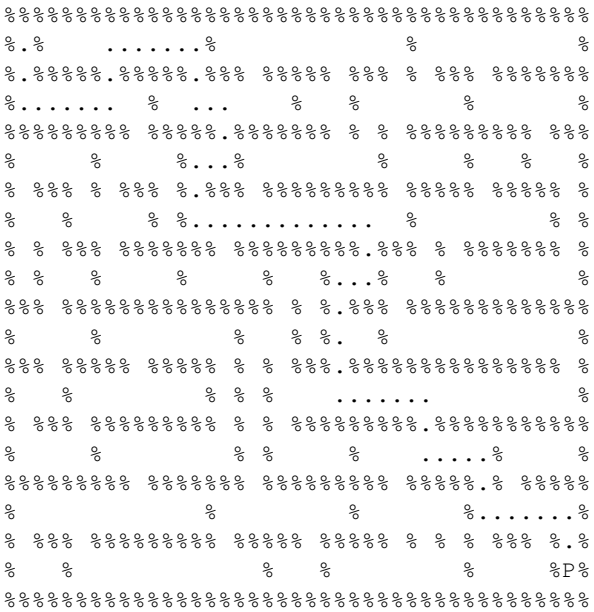

Figure 10: Solution to Medium Maze using A*

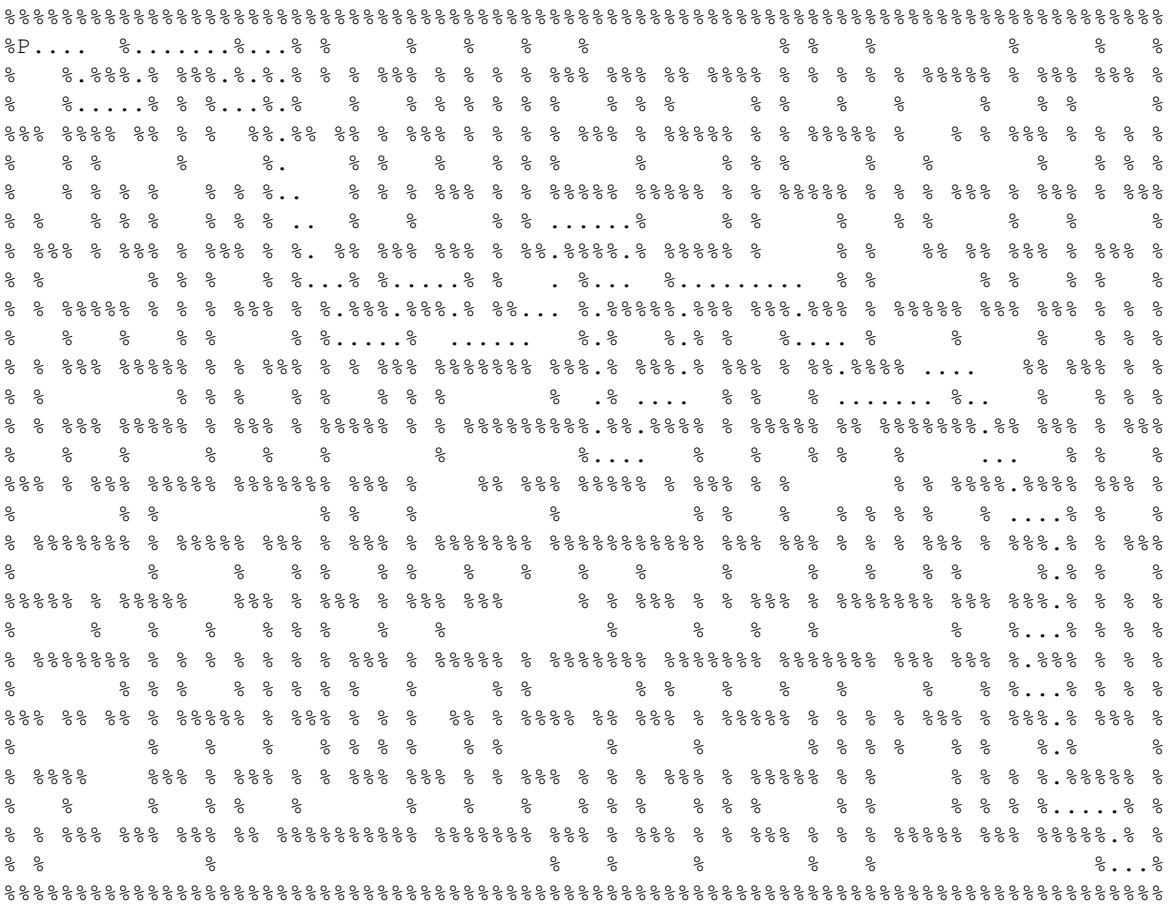

Figure 11: Solution to Big Maze using A*

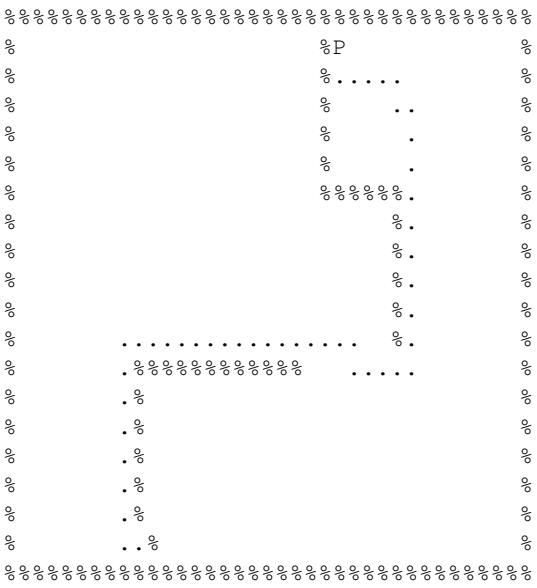

Figure 12: Solution to Open Maze using A*

# B Maze Solutions for Part 1.2

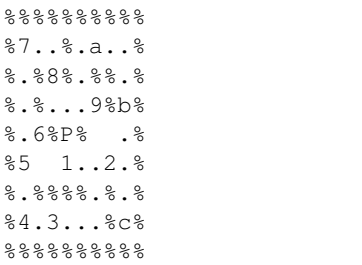

Figure 13: Solution to Tiny Search using A*

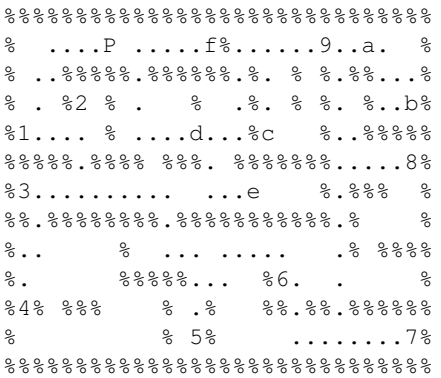

Figure 14: Solution to Small Search using A*

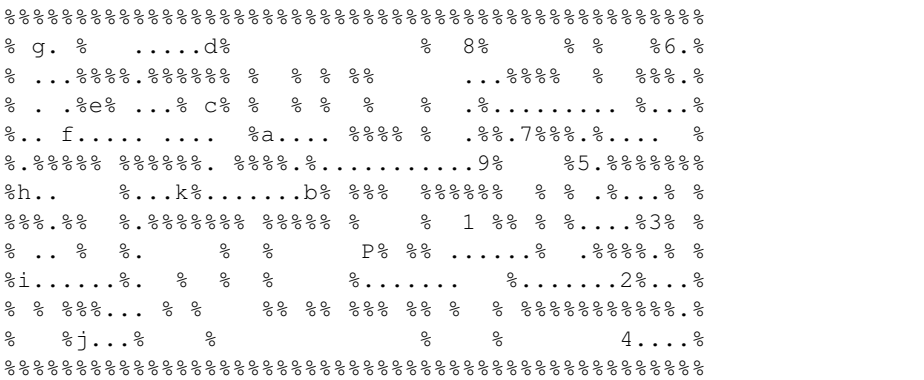

Figure 15: Solution to Medium Search using A*

# C Open Source Software Used

The following list of Open Source software was used to help with this project:

- **gif-h**
  - A single header in C that allows for basic GIF processing
  - https://github.com/ginsweater/gif-h
- **libpng**
  - C library for PNG manipulation
  - http://www.libpng.org/pub/png/libpng.html