# MP1 Report

Dennis J. McWherter, Jr. (dmcwhe2@illinois.edu) 4 credit
Joe Fetsch (jfetsch2@illinois.edu) 4 credit
Revanth Reddy (rreddy4@illinois.edu) 4 credit

September 26, 2016

## 1 Part 1.1

This section discusses our search algorithm implementation as well as performance metrics for each algorithm on its respective maze. It should be noted that $A*$ and Greedy BFS always use Manhattan distance for the heuristic in this section. The set of labeled ASCII maze solutions can be found in section 7 (Figures).

### 1.0.1 Implementation

Our maze solver is based around a unified top-level search algorithm. This algorithm was adapted from the pseudocode presented in class (lecture 6, slide 1 for the Fall 2016 semester). The most significant adaptation from the presented code is our ability to handle multiple goals. To achieve this, we first altered our termination condition. When a path exists, the original algorithm terminates upon reaching its *first* goal. However, in searching for multiple goals the algorithm must terminate only after reaching *all* goals. It is crucial that our set of visited positions be cleared each time we successfully reach a goal. If this is ignored, the solver would fail to solve mazes with goals that are only accessible through a single path since they would not be able to use the same path positions that were visited by a previous goal.

Aside from algorithmic adaptations, there were some practical concerns that we handled in our implementation. Specifically, we pass an initFrontier function which takes the start state as an argument. This function returns a $frontier$ type supporting operations such as *add* and *pop* (only expanded in-bounds, non-wall nodes are added to the frontier). In the case of BFS and DFS this is a simple *Queue* and *Stack*, respectively. Both Greedy BFS and A* use a Priority Queue. However, it is a subtle and important detail that the Priority Queue *add* must support *replacing*. That is, while inserting into the priority queue, if a node already exists with the same position, the node with the smallest score must be placed in the queue and the larger removed. If the priority queue allows duplicate positions with different scores, it is possible that there will be significant state explosion.

Each node in our $frontier$ holds necessary state. In particular, our maze state for any node would minimally require $(currentPath, position, score)$ where $currentPath$ is an ordered list of visited positions, $position$ is the current position, and $score$ is the estimated cost of this $(currentPath, position)$ pair. For convenience, we also include other pre-computations such as $currentPathLength$ to avoid computing it each time it is required. For all algorithms, this state is used to reconstruct the final

path when all goals have been met. Specifically, $currentPath + [position]$ is our final path. For Greedy BFS and A*, the $currentPath$ (used to calculate remaining goals) and $position$ are used in their heuristic functions and $score$ is used to insert the node into the Priority Queue.

In summary, our goal test checks whether or not the position of our current node is contained in the set of unvisited goal positions. Our state representation contains $(currentPath, position, score)$. Finally, our transition model is dependent on our specific frontier and heuristic functions. Specifically, BFS and DFS use a queue and stack, respectively while Greedy BFS and A* use their heuristic functions to score a node and add it to a Priority Queue.

#### 1.0.2 Medium Maze Metrics

| Algorithm | Path Cost | Expanded Nodes |
|---|---|---|
| BFS | 69 | 343 |
| DFS | 117 | 373 |
| Greedy | 125 | 158 |
| A* | 69 | 197 |

#### 1.0.3 Big Maze Metrics

| Algorithm | Path Cost | Expanded Nodes |
|---|---|---|
| BFS | 267 | 795 |
| DFS | 295 | 527 |
| Greedy | 267 | 631 |
| A* | 267 | 779 |

#### 1.0.4 Open Maze Metrics

| Algorithm | Path Cost | Expanded Nodes |
|---|---|---|
| BFS | 75 | 574 |
| DFS | 343 | 400 |
| Greedy | 83 | 346 |
| A* | 75 | 405 |

# 2 Part 1.2

## 2.1 Heuristic Discussion

Our heuristic in part 1.2 is a combination of two different heuristics. Formally:

$$\begin{aligned} h1(x) &= |G| \\ h2(x) &= \max_{g \in G}(\text{manhattan}(x, g)) \\ h(x) &= \max(h1(x), h2(x)) \end{aligned}$$

where $G$ is the set of unreached goal positions and $x$ is the position being estimated. In words, the heuristic takes the maximum value between the number of unreached goals and the largest

manhattan distance from point $x$ to any remaining goal $g \in G$. This heuristic is admissible but inconsistent. As a result, the solution produced is not necessarily optimal.

Without a formal proof, the admissibility of the heuristic can be described in the relaxation of problem constraints. First, $h1$ (i.e. number of remaining goals) must be admissible because it fully relaxes the cost-of-movement constraint. Particularly, this heuristic describes a situation where moving from position $(x, y)$ to $(x', y')$ results in a unit step cost for all $(x, y), (x', y')$ positions in the maze. As a result, this is the exact solution for any maze where the start point and all goals are within a single step, however, for any other maze this underestimates the cost of the true path which makes it admissible.

For heuristic $h2$, we calculate the MANHATTAN distance between a position and goals. This heuristic relaxes our wall constraints. Since we know that manhattan distance is actually the shortest distance between two points given our movement constraints, this would provide a straight line solution if we could move directly through walls from the start to finish points. However, since this is not the case for achieving an actual result for all mazes with a path requiring turns, the manhattan distance provides an optimistic estimate of the path cost. Similarly, for any maze where the solution path is contains no walls to the goals, the heuristic provides an exact estimate which shows that it is admissible.

Since both heuristics $h1$ and $h2$ are admissible, we know that $\max(h1, h2)$ is also admissible. Thus, since $h = \max(h1, h2)$, $h$ is an admissible heuristic.

As previously mentioned, though heuristic $h$ is admissible, it is not *consistent*. This is best demonstrated through a counter-example. Consider the following maze:

```
%%%%%
%.P..%
%%%%%
```

The first iteration of the algorithm will label $h(right) = 2$ since there are only two goals remaining at the point directly to the right of $P$ and $h(left-most) = 3$ since the $left-most$ point is a distance of 3 from the $right-most$ goal. This results in $f(right) = g(right)+h(right) = 0+2 = 2$. At this point, the algorithm chooses to move one step to the right to the lower cost node. The next iteration will calculate $h(P) = 2$ and $h(right - most) = 3$. This results in $f(P) = 1 + 2 = 3$ and $f(right - most) = 1 + 3 = 4$. Now, we have returned to position $P$ which will result in $f(left - most) = 2 + 3 = 5$ and $f(right - most) = 2 + 3 = 5$. Since this is a small example, we know the optimal solution to this maze is to first reach the left-most goal and then retrieve the remaining two resulting in a path cost $C^* = 4$. Consequently, since there is an $f(n) = 5$, we see that because $5 > 4$, $f(n) > C^*$ which breaks the consistency constraint.

## 2.2 Solutions

This section includes multi-goal search solution metrics. Please refer to section 8 (Search Solutions) for the ASCII solutions. Each solution was derived using the $A*$ algorithm with the heuristic function described in the section above.

# 3 Part 1.2

| Maze | Path Cost | Expanded Nodes |
|---|---|---|
| Tiny Search | 43 | 778 |
| Small Search | 416 | 31467 |
| Medium Search | 580 | 91002 |

# 4 Part 2.1

## 4.1 Implementation and Heuristic

We implemented our search using the $A*$ algorithm. The algorithm is implemented in a similar way as to part 1 where it is discussed in further detail. However, in this problem, expanding a node corresponded to generating nodes from the 12 different rotations (6 faces with CW and CCW each) from any given state rather than a path through a maze. After properly modeling the clockwise rotations about each face, we implemented each counter-clockwise rotation by simply performing three clockwise rotations.

Our heuristic was based on the number of misplaced tiles. Specifically, we had to divide this number by a factor of 8 to be admissible. In general, any face rotation affects 12 cubes total (4 cubes for the face + 2 cubes x 4 sides). However, in the shortest path we know there is only a single move required to complete the puzzle as the two opposing faces are complete and one rotation is required to align the other four faces. As a result, we must rotate about the completed face or else there would be more than a single move remaining. Since the completed face is a monochromatic, this can be treated as a noop leaving on the remaining 2 cubes x 4 sides = 8 cubes to be rotated.

## 4.2 Results

| Cube | Path | Expanded Nodes | Running Time |
|---|---|---|---|
| 1_1 | T' R' Ba F' | 156 | 0.08s |
| 1_2 | F' L L T' F' | 1785 | 0.80s |
| 1_3 | T' F' L' L' T' F' | 4305 | 2.00s |

# 5 Part 2.2

This section discusses our results for part 2.2. Part 2.2 differs from part 2.1 largely because rotation invariance allows for a greater solution space. As a result, our solver can find shorter paths to one of the goal states using fewer overall nodes.

## 5.1 Implementation Modifications

The major change between part 2.1 and 2.2 was calculating our solution across many cubes. Since we had implemented path-hashing previously, we had to adjust our heuristic for any valid solution. This resulted in changing our heuristic to iterate over the 24 possible solution rotations and taking the *minimum* number of misplaced tiles available for that configuration to any of the solutions. Moreover, we had to generate the solution cubes from the primary solution by applying several

global rotations. To aid in detecting repeated states in our closedSet, we hash using python's built-in hash function for a string. The input string is the current path for the given node. We had also tried tried other hashing strategies such as hashing the 24 globally rotated cube states for each unique configuration. To guarantee hash collisions would occur only for the same states, we used a monotonically increasing number which was increased once for each unique state as the hash value (i.e. any given state and one of its 23 rotations would have the same hash). However, by the end of our experiments we found that our path-hashing strategy resulted in the best overall performance.

### 5.2 Results

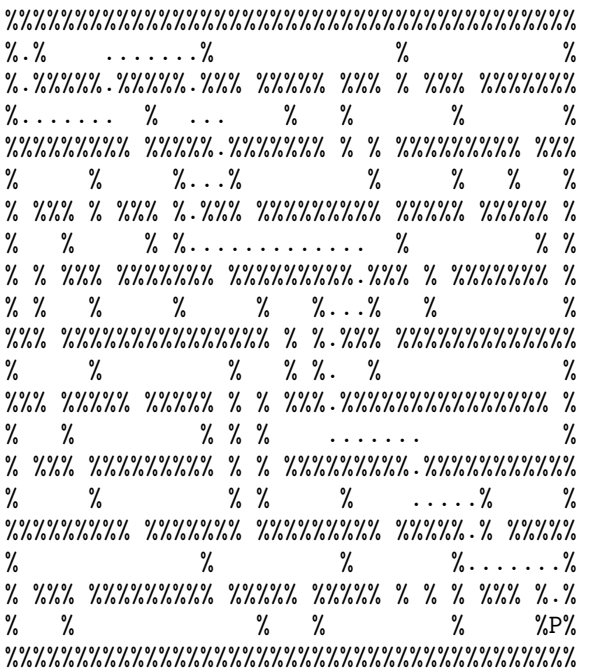
| Cube | Path | Expanded Nodes | Running Time |
|---|---|---|---|
| 2_1 | T' L' | 7 | 0.03s |
| 2_2 | F' T' L' L' T' F' | 13818 | 21.88s |
| 2_3 | F' R' Bo' T' F' | 8204 | 13.04s |

## 6 Statement of Contribution

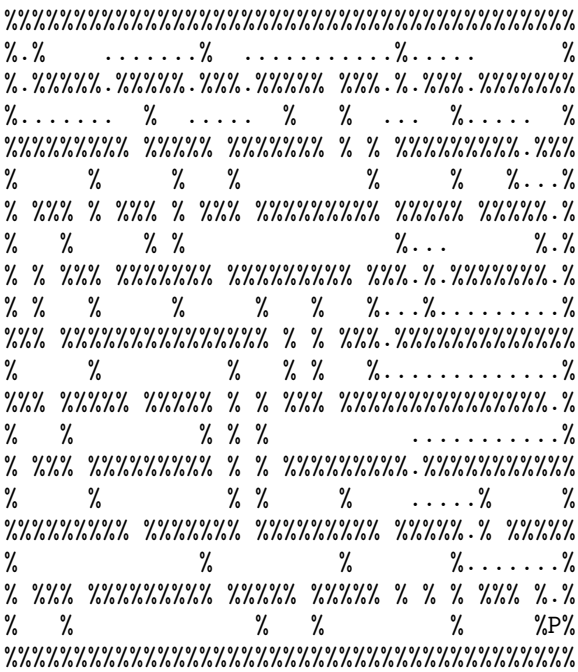
The three group members made a combined effort in discussing the logic for the implementations and completing this assignment requirements. Dennis was responsible for the implementation for parts 1.1 and 1.2 while Joe and Revanth reviewed the solution and write-up to check for mistakes. They also made necessary modifications for part 2.2. In part 2.1 Joe wrote the initial implementation and Revanth fixed the rotation code to be consistent while Dennis reviewed and helped debug the code.

# 7 Figures

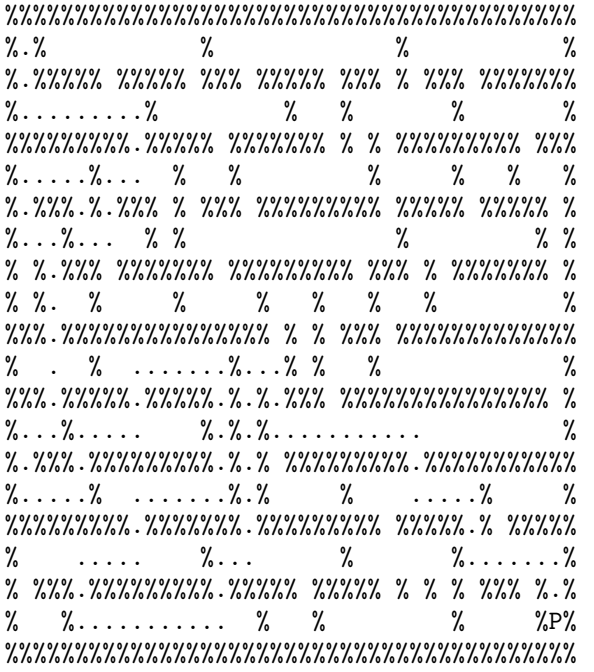
## 7.1 Medium Maze Solutions



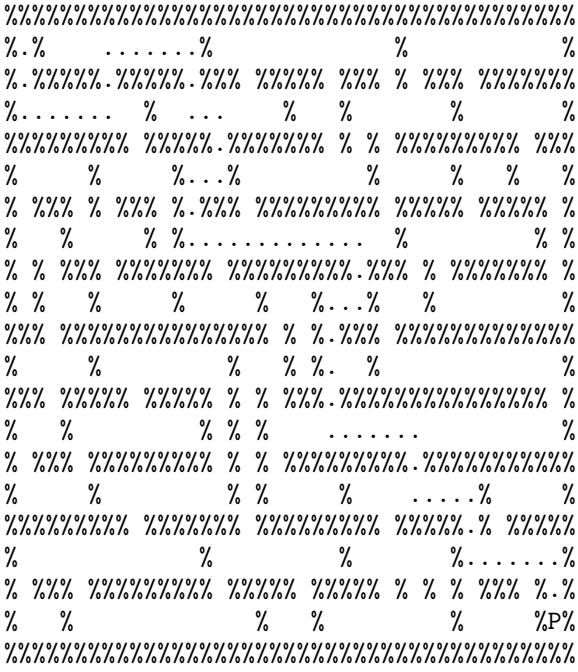
(a) BFS



(b) DFS



(c) Greedy BFS



(d) A*

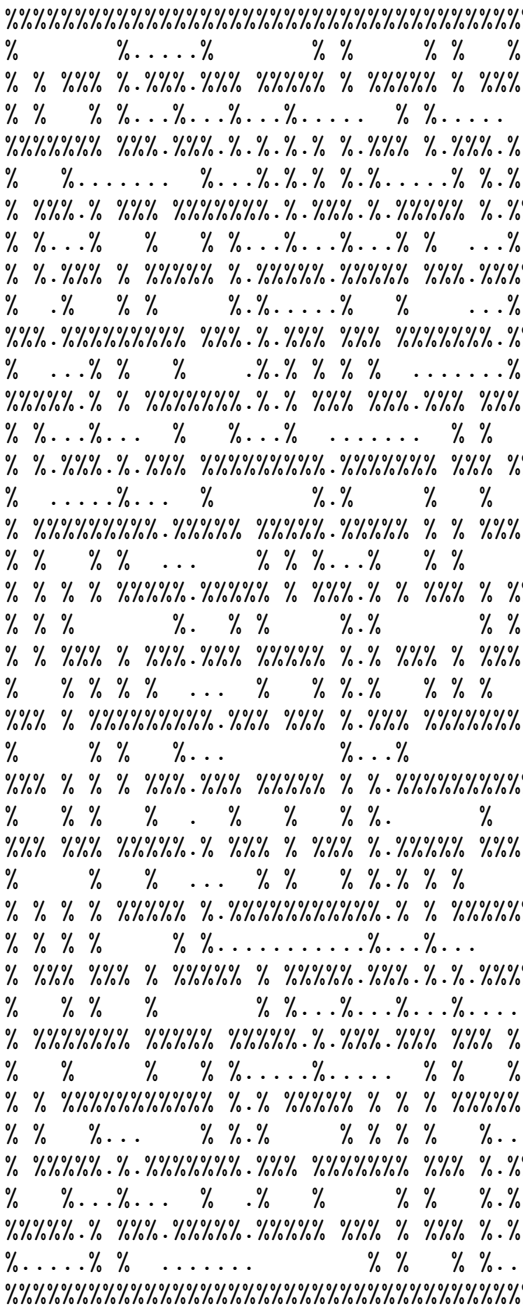
Figure 1: Medium maze solutions

## 7.2 Big Maze Solutions

(a) BFS

(b) DFS

Figure 2: Big maze solutions

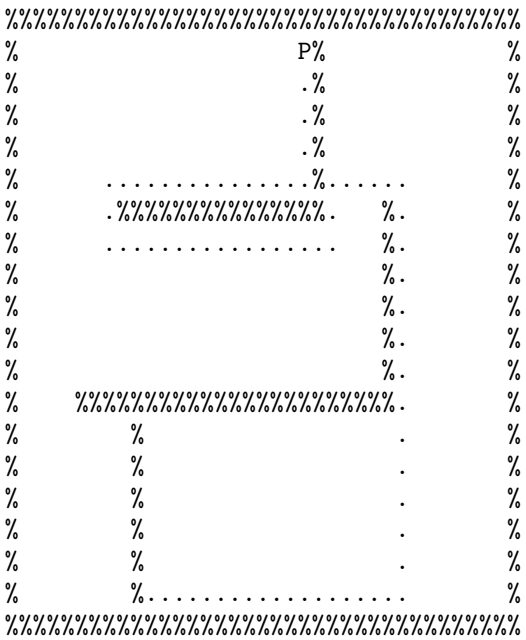


(a) Greedy BFS

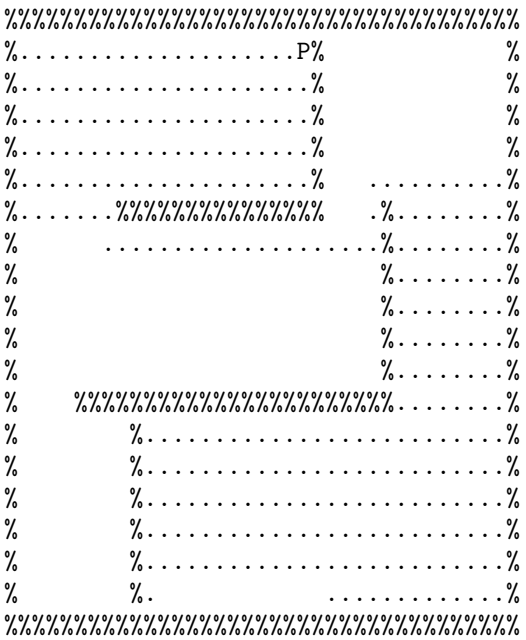
(b) A*

Figure 3: Big maze solutions

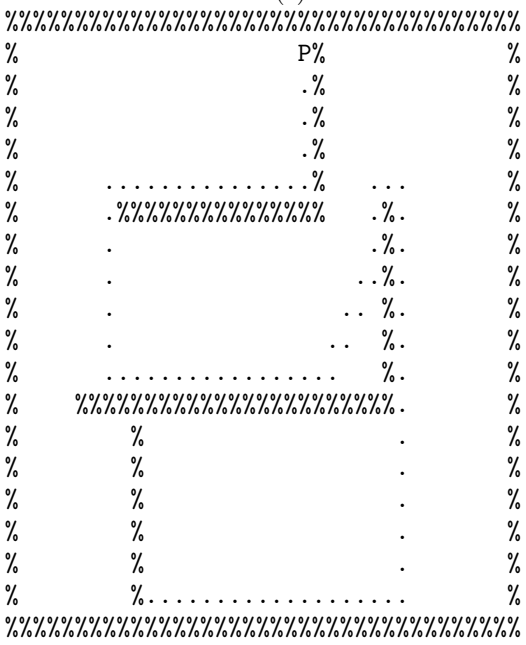
## 7.3 Open Maze Solutions



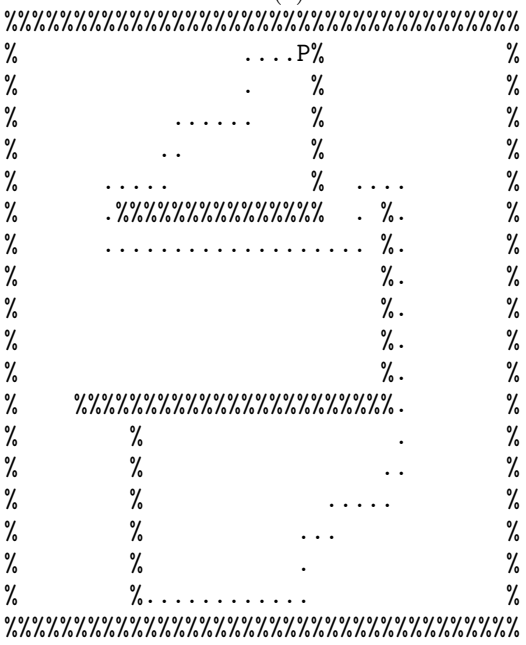
(a) BFS



(b) DFS



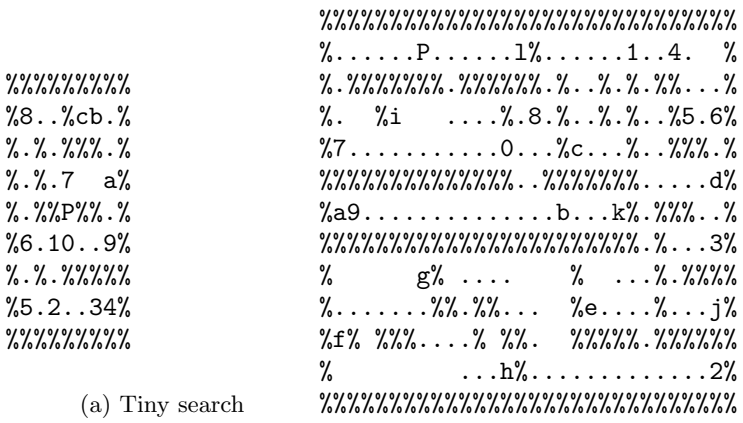
(c) Greedy BFS



(d) A*

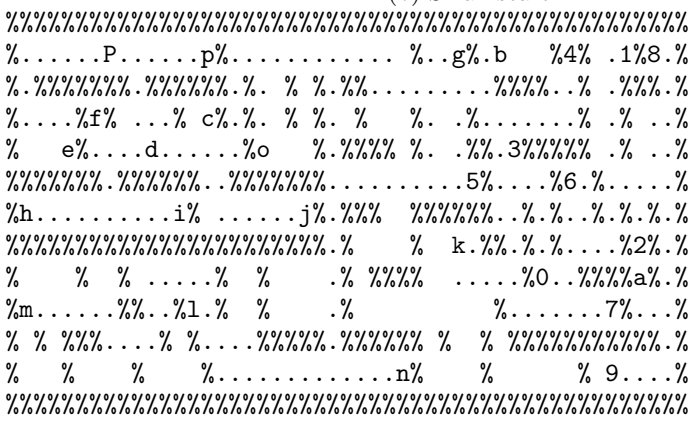
Figure 4: Open maze solutions

# 8 Search Solutions



(a) Tiny search

(b) Small search



(c) Medium search

Figure 5: Multi-goal search maze solutions