Rodney Shaghoulian – shaghou2 – (3 Units)

1.0 Setting Up The Environment

The backbone of the search algorithms is the **Maze** class:

```
public class Maze{
	int rows;
	int columns;
	Point [][]grid;
	Point startPoint;
	Point endPoint;
	Ghost ghost;			//Used in Part 1.3
	ArrayList<Point> dots;	//Used in Part 2

	//methods omitted
}
```

The Maze class relies on the **Point** class:

```
public class Point{
	public int x;
	public int y;
	public PointType pointType;		//EMPTY, WALL, DOT, START
	public Heuristic heuristic;

	//methods omitted
}
```

A **Maze** is constructed by reading in a text file line by line, and reading each character in the line. Every time a character is read, a **Point** is created, and it is assigned a value depending on the character that is read.

- '%' corresponds to *PointType.WALL*
- ' ' corresponds to *PointType.EMPTY*
- '.' corresponds to *PointType.DOT*
- 'P' corresponds to *PointType.START*

Two passes are done to read the necessary information from the text file:

Pass 1:

- Count number of rows and columns.
- Create grid

Pass 2:

- Create each **Point** in the grid.
- Find the startPoint and endPoint.
- Create the **Ghost**.
- Save all the dots in an **ArrayList<Point>** (for Part 2).

In addition to storing its *PointType*, a **Point** object also stores its location with two integers, one that corresponds to its x coordinate, and another corresponding to its y coordinate. It also saves **Heuristic** information (for use in Greedy and A* Search).

Tie-Breaking: Right, Left, Up, Down

Point first checks the **Point** to the right of it, then the **Point** to theleft of it, then the **Point** above it, and then the **Point** below it. To do this, **Point** has a method called "*getAdjacentPoints*" that returns an **ArrayList** of each **Point** that is next to the **Point** that calls the method. It makes sure that each **Point** it adds to the **ArrayList** is in a valid position (not out of bounds from our grid)

Console Output

The **Point** and **Maze** classes each have a *toString()* method, which is what we use to print the solution. The *toString()* method for **Point** simply returns its corresponding character value in the form of a String. The *toString()* method for **Maze** returns a String comprised of the *toString()* for each **Point** in the **ArrayList**.

1.1 Basic Pathfinding

Breadth-First Search

```
public class BFS{
	/* Data */
	public Queue<Point> frontier
	public HashSet<Point> visited;
	public HashMap<Point, Point> predecessor;
	public int nodesExpanded;
	public int solutionDistance;

	/* Constructor */
	public BFS(){ /* Details omitted */ }

	/* Maze Solver */
	public void findSolution(Maze maze){ /* Details omitted */ }
}
```

Notice we use a `Queue` for the frontier. It is important to note that we add a `Point` to "visited" when we **add** it to the frontier.

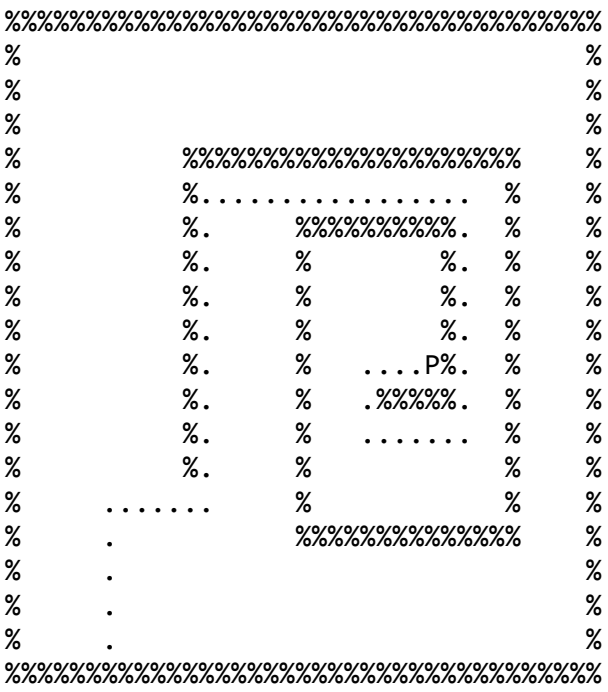
```
BFS Solution to mediumMaze.txt
%%%%%%%%%%%%%%%%%%%%%%%%%
%.%   %   %   %   % % %
%.  %   %% % %%%% %%  %
%.%   % % % %       % %
%.%% %%%     %% %% %%%%
%.% %   % % % % %     %
%.%%% % %%%  %    %% %%
%.% %...    % % %   % %
%.....%.% %%%   % %   %
% % % %.....% % %   % %
%   %%% %%%.% %   % %%%
% %     % %.....%     %
%% % %%    %%%%.% %% %%
% % %   % %    ...  % %
%   %% %% %% %%%%.%%  %
% % %     %     %.% % %
%%%% %% %%  %% % . % %%
% % %     %   % %.% % %
% %% % %%%%% %   ...  %
% %       %   % % %.% %
%   %% %% %%% %   %.%%%
% % %   %       % %..P%
%%%%%%%%%%%%%%%%%%%%%%%%%
BFS Nodes Expanded = 224
BFS Solution Distance = 42
```

BFS Solution to bigMaze.txt

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

BFS Nodes Expanded = 716
BFS Solution Distance = 62

BFS Solution to openMaze.txt



BFS Nodes Expanded = 274

BFS Solution Distance = 54

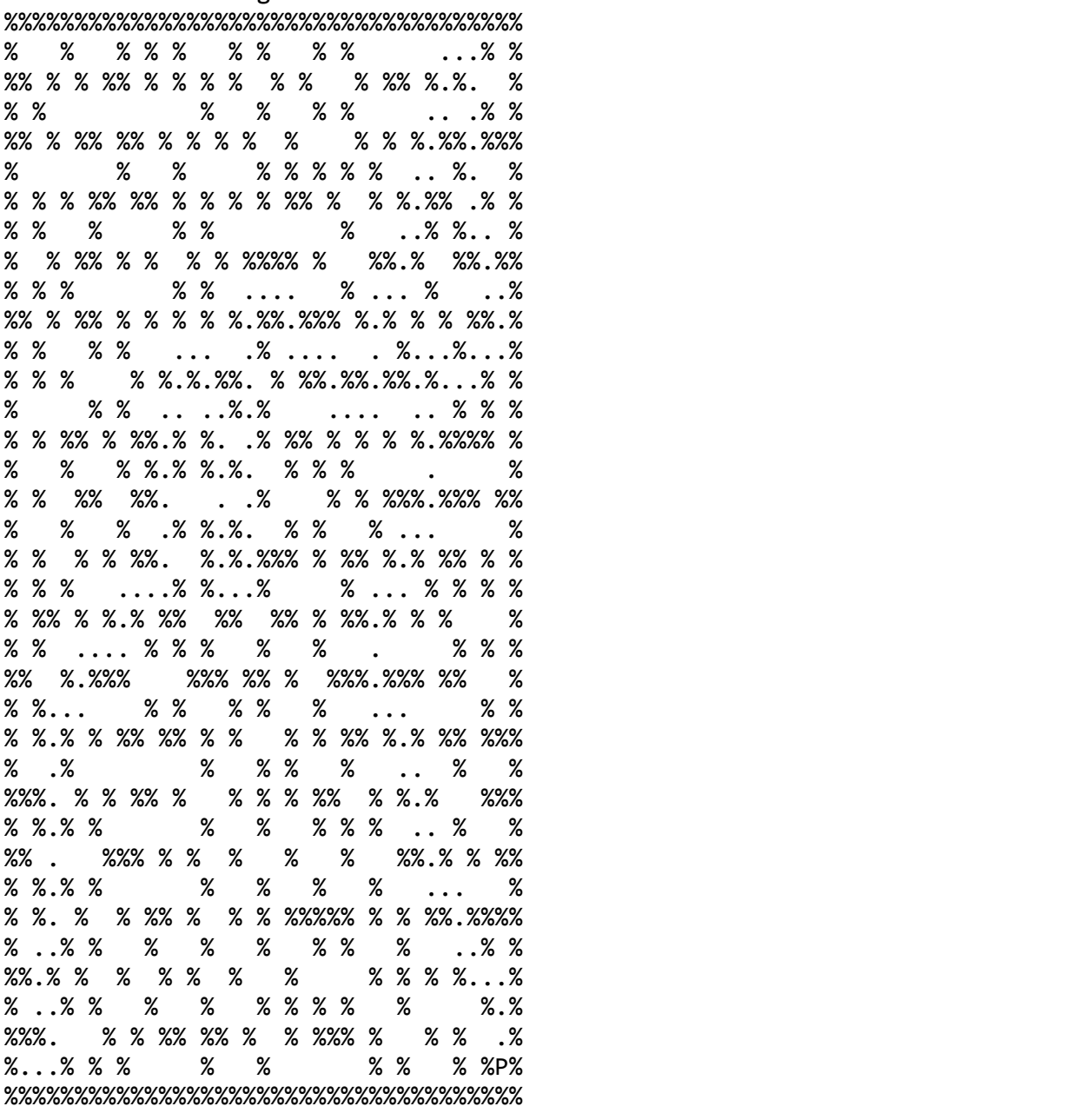
Depth-First Search

```
public class DFS{
	/* Data */
	public Stack<Point> frontier
	public HashSet<Point> visited;
	public HashMap<Point, Point> predecessor;
	public int nodesExpanded;
	public int solutionDistance;

	/* Constructor */
	public DFS (){ /* Details omitted */ }

	/* Maze Solver */
	public void findSolution(Maze maze){ /* Details omitted */ }
}
```

Notice the only difference between **BFS** and **DFS** is that, for the frontier for **DFS**, we use a **Stack** instead of a **Queue**.

```
DFS Solution to mediumMaze.txt
%%%%%%%%%%%%%%%%%%%%%%%%%
%.%   %   %   %   % % %
%.  %   %% % %%%% %%  %
%.%   % % % %  .... % %
%.%% %%%     %%.%%.%%%%
%.% %...% % % %.%..   %
%.%%%.%.%%%  % . .%% %%
%.% %. .    % %.%.  % %
%. ...%.% %%%  .%.%   %
%.%.% %.    % %.%.  % %
%...%%%.%%% % %. .% %%%
% %    .% %    .%.    %
%% % %%.   %%%%.%.%% %%
% % % ..% % .... .  % %
%   %%.%% %%.%%%%.%%  %
% % % ..  %..   %.% % %
%%%% %%.%% .%% % . % %%
% % % ..  %.. % %.% % %
% %% %.%%%%%.%   ...  %
% %   ....% ..% % %.% %
%   %% %%.%%%.%   %.%%%
% % %   %.....  % %..P%
%%%%%%%%%%%%%%%%%%%%%%%%%
DFS Nodes Expanded = 110
DFS Solution Distance = 96
```

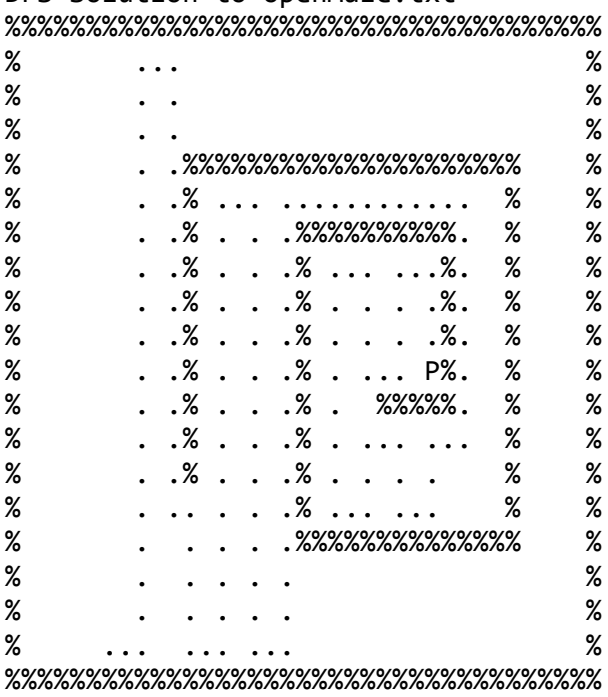
DFS Solution to bigMaze.txt



DFS Nodes Expanded = 427

DFS Solution Distance = 150

DFS Solution to openMaze.txt



DFS Nodes Expanded = 139

DFS Solution Distance = 138

Greedy Search

```
public class Greedy{
	/* Data */
	public PriorityQueue<Point> frontier
	public HashSet<Point> visited;
	public HashMap<Point, Point> predecessor;
	public int nodesExpanded;
	public int solutionDistance;

	/* Constructor */
	public Greedy(){ /* Details omitted */ }

	/* Maze Solver */
	public void findSolution(Maze maze){ /* Details omitted */ }
}
```

Notice the main difference between **Greedy** and **BFS/DFS** is the use of a **PriorityQueue** for our frontier. The **PriorityQueue** sorts elements by utilizing the following **Comparator**:

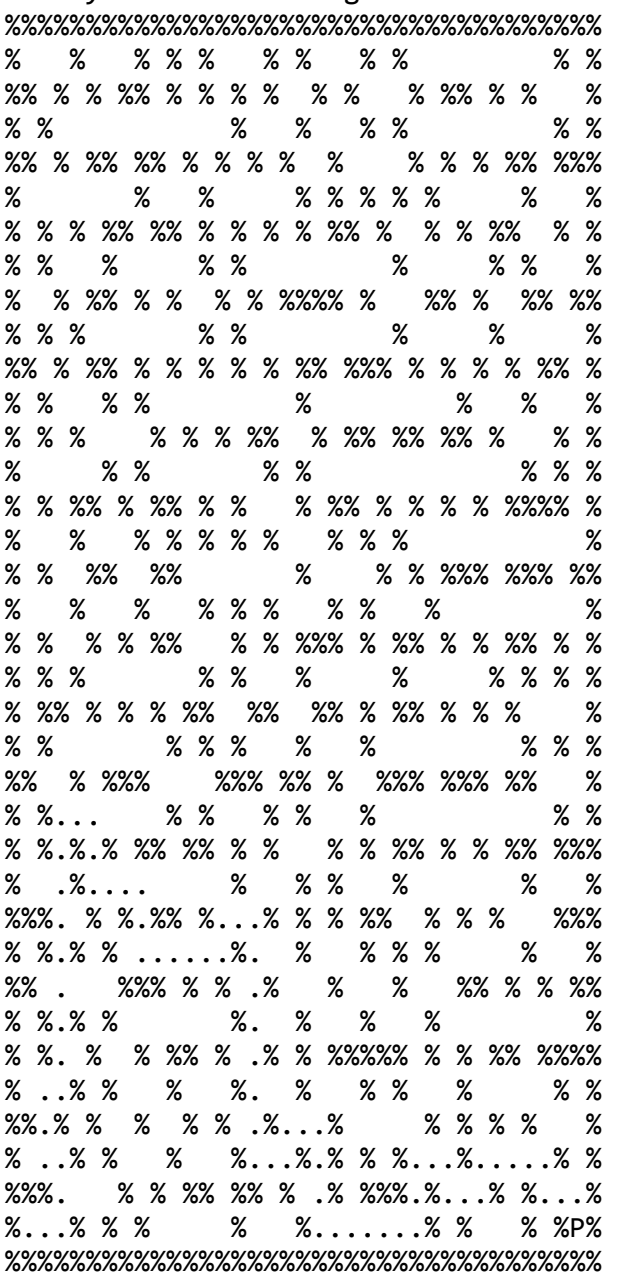
```
public class GreedyComparator implements Comparator<Point>{
	@Override
	public int compare(Point p1, Point p2){
		return p1.heuristic.distanceHeuristic –
			p2.heuristic.distanceHeuristic;
}
```

Greedy Solution to mediumMaze.txt

Greedy Best-First Nodes Expanded = 57

Greedy Best-First Solution Distance = 42

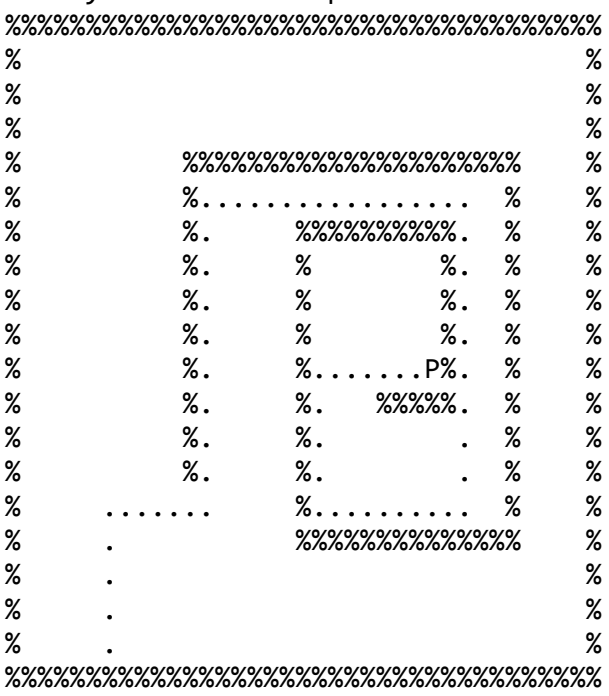
Greedy Solution to bigMaze.txt



Greedy Best-First Nodes Expanded = 117

Greedy Best-First Solution Distance = 70

Greedy Solution to openMaze.txt



Greedy Best-First Nodes Expanded = 124
Greedy Best-First Solution Distance = 64

A* Search

```
public class Astar{
	/* Data */
	public PriorityQueue<StateAstar> frontier
	public HashSet<Point> visited;
	public HashMap<StateAstar, StateAstar> predecessor;
	public int nodesExpanded;
	public int solutionDistance;

	/* Constructor */
	public Astar(){ /* Details omitted */ }

	/* Maze Solver */
	public void findSolution(Maze maze){ /* Details omitted */ }
}
```

We have now switched from using a **Point** to using a **StateAstar** to represent each state

```
public class StateAstar{
	public Point pacmanLocation;
	public int costSoFar;
	public Heuristic heuristic;

	//methods omitted
}
```

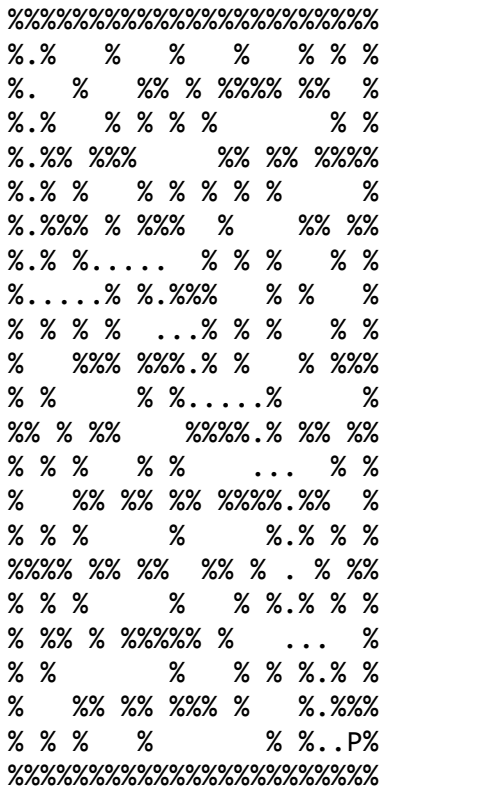
Notice the main difference between **Astar** and **Greedy** is way the **StateComparator** tells us to sort values for our **PriorityQueue**. It takes into account both the **Heuristic** and the path cost. This gives us f(n) = g(n) + h(n)

The **PriorityQueue** sorts elements by utilizing the following **Comparator**:

```
public class StateComparator implements Comparator<StateAstar>{
	@Override
	public int compare(StateAstar s1, StateAstar s2){
		return (s1.heuristic.distanceHeuristic + s1.costSoFar) –
			(s2.heuristic.distanceHeuristic + s2.costSoFar);
}
```

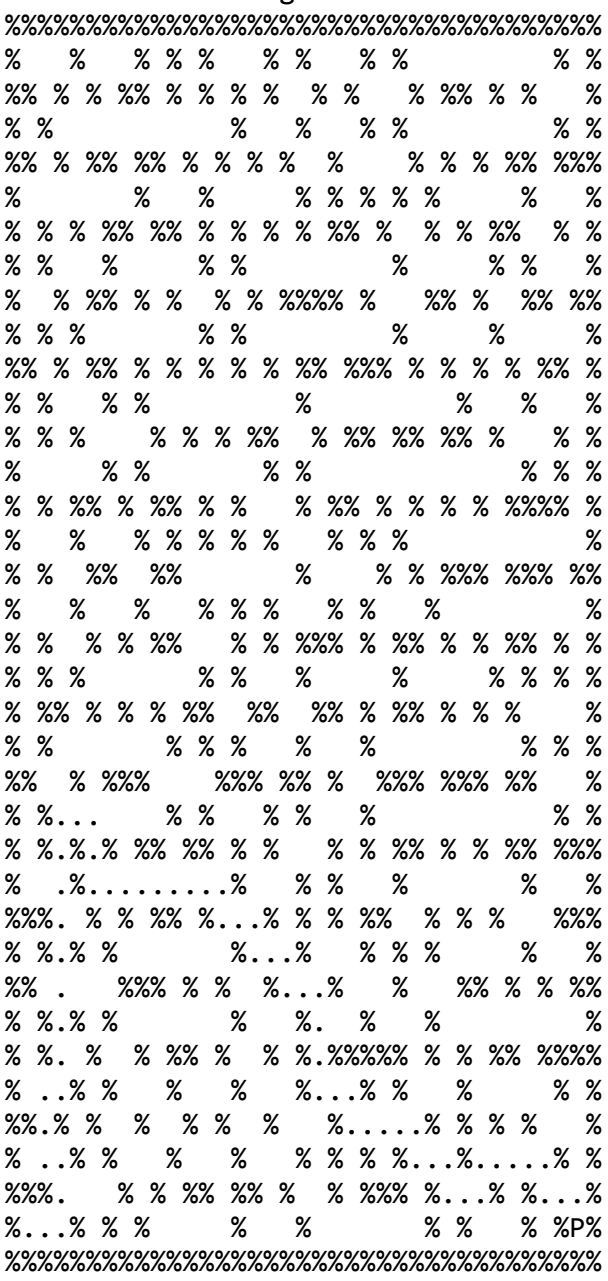
It is very important to note that we mark a state as visited when we **remove** it from the frontier. This implementation is different than how we did it for **BFS**/**DFS**/**Greedy**. This difference arises since we are using a different state representation than in **BFS**/**DFS**/**Greedy**.

A* Solution to mediumMaze.txt



A* Nodes Expanded = 105
A* Solution Distance = 42

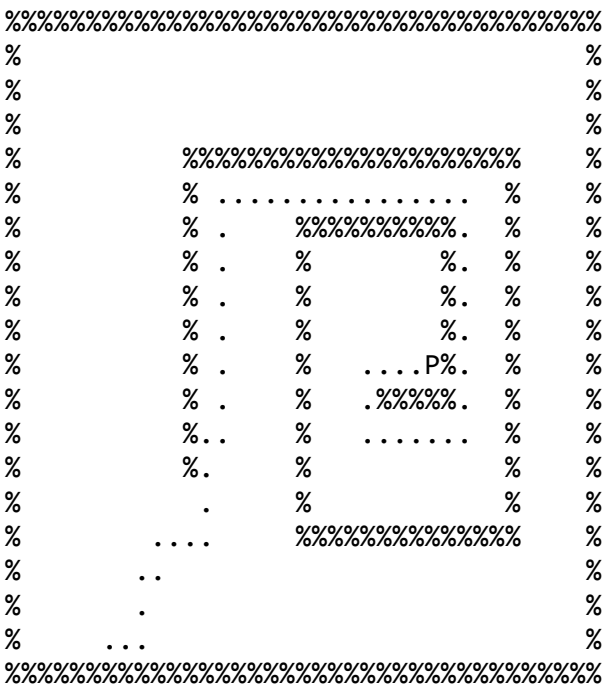
A* Solution to bigMaze.txt



A* Nodes Expanded = 275

A* Solution Distance = 62

A* Solution to openMaze.txt



A* Nodes Expanded = 169

A* Solution Distance = 54

1.2 Penalizing Turns

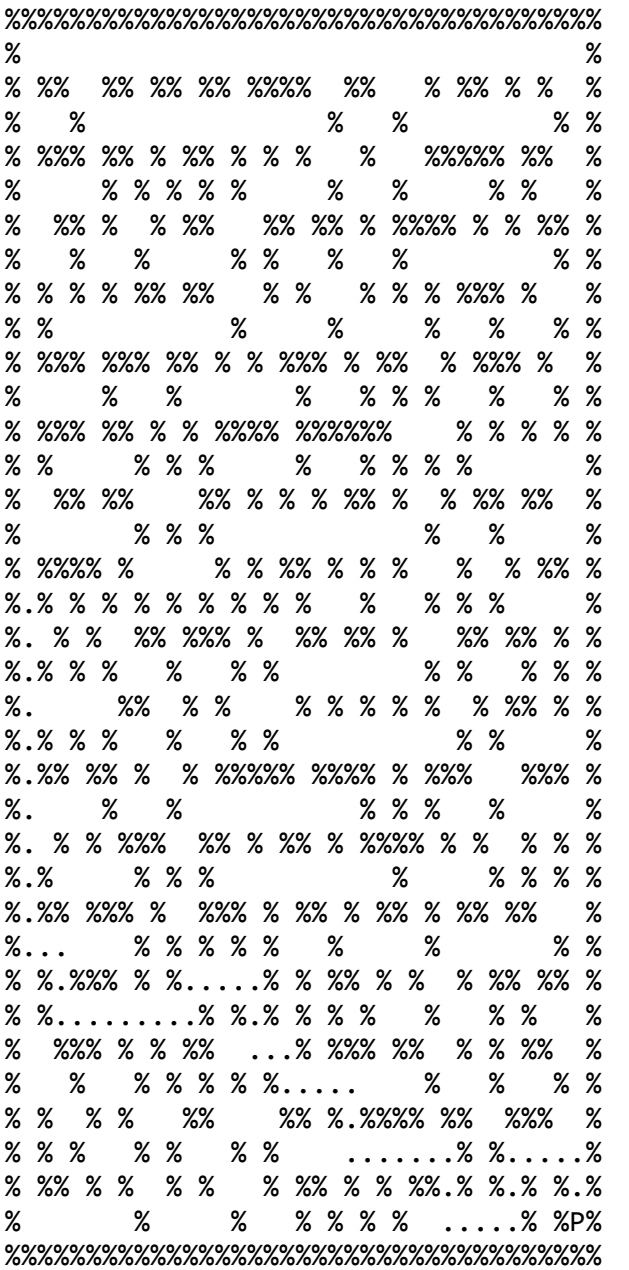
For Part 1.2, our representation of a “state” is updated to include the *Direction* that pacman is facing.

```
public class StateTurns{
	public Point pacmanLocation;
	public Direction direction;		//added for Part 1.2
	public int costSoFar;
	public Heuristic heuristic;

	//methods omitted
}

public enum Direction{
	NORTH, SOUTH, EAST, WEST
}
```

Here are the results for using manhattan distance as a heuristic

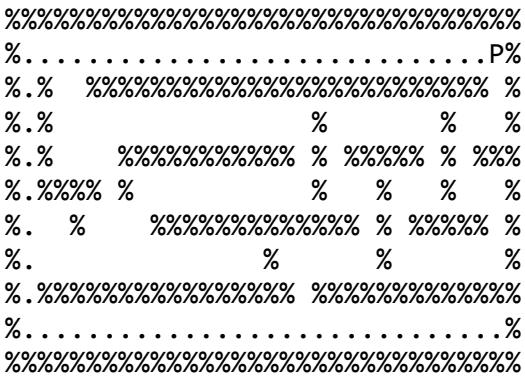
```
A* Solution to smallTurns.txt (moveCost = 2, turnCost = 1)
Heuristic: Manhattan Distance
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                P%
% %  %%%%%%%%%%%%%%%%%%%%%%%%%%.%
% %                  %.......%...%
% %    %%%%%%%%%%% %.%%%%%.%.%%%
% %%%% %            %...%...%...%
%   %    %%%%%%%%%%%%%.%.%%%%%.%
%                % .....%.......%
% %%%%%%%%%%%%%%%.%%%%%%%%%%%%%
%                    ............%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
AstarTurns Nodes Expanded = 626
AstarTurns Solution Cost = 120
```

A* Solution to bigTurns.txt (moveCost = 2, turnCost = 1)
Heuristic: Manhattan Distance



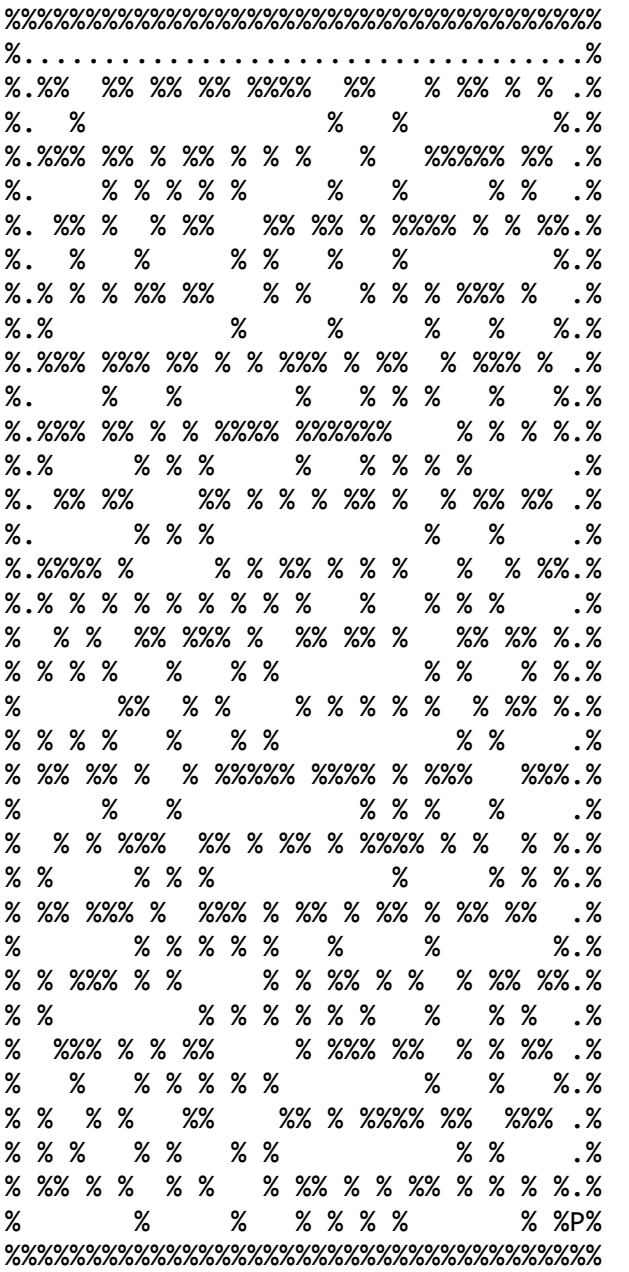
AstarTurns Nodes Expanded = 2022
AstarTurns Solution Cost = 133

A* Solution to smallTurns.txt (moveCost = 1, turnCost = 2)
Heuristic: Manhattan Distance



AstarTurns Nodes Expanded = 458
AstarTurns Solution Cost = 74

A* Solution to bigTurns.txt (moveCost = 1, turnCost = 2)
Heuristic: Manhattan Distance



AstarTurns Nodes Expanded = 1614
AstarTurns Solution Cost = 90

To design a better heuristic, we have to take into account the cost of each "move" and "turn". Our state representation is now updated with that information. We can change the values of moveCost and turnCost values to generate different cost scenarios.

```
public class StateTurns{
      public Point pacmanLocation;
      public Direction direction;         //added for Part 1.2
      public int costSoFar;
      public Heuristic heuristic;

      private int moveCost;         //added for Part 1.2
      private int turnCost;         //added for Part 1.2

      //methods omitted
}
```

The heuristic that will be used is:

```
heuristic = (moveCost * manhattanDistance) + (turnCost * minimumTurns);
```

Explanation of heuristic:

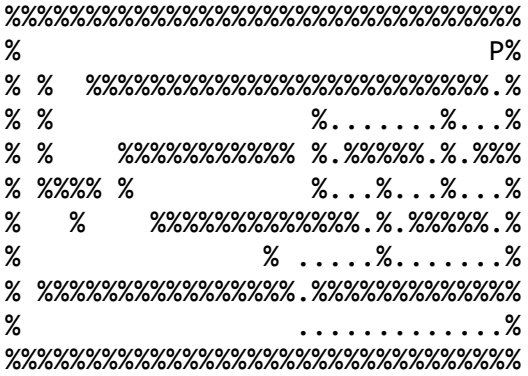
- First, we should observe that when moveCost = 1 and turnCost = 0, we get heuristic = manhattanDistance. This is what we used in Part 1.1
- We already know that manhattanDistance is **admissible**. The multiplication by "moveCost" now lets us allow for move costs that are greater than 1. This product is still **admissible** since it never overestimates the cost of reaching the goal.
- To make our heuristic **more informed**, we also have to take turnCost into account. In addition to the cost of moving, there are a minimum number of turns required to reach the goal. Adding this value to our heuristic (with a factor of turnCost), still keeps our heuristic **admissible** since we are using "minimumTurns" and never overestimating the number of turns needed to reach the goal.
- Our final heuristic is still **admissible** since it
  1) never overestimates the number of steps (with moveCost taken into account) to reach the goal
  2) never overestimates the number of turns (with turnCost taken into account) to reach the goal

The trick to make our heuristic **more informed** is to notice that a "dead end" requires 2 turns to get out of. We will define a "dead end" as any spot in the maze that has 3 walls surrounding it.

By simply setting "minimumTurns" to 2 for dead ends, and "minimumTurns" to 0 for all other spots, we can have Pacman be inclined to avoid dead ends. This simple improvement (combined with taking moveCost into account) gives us better results in all 4 scenarios of mazes.

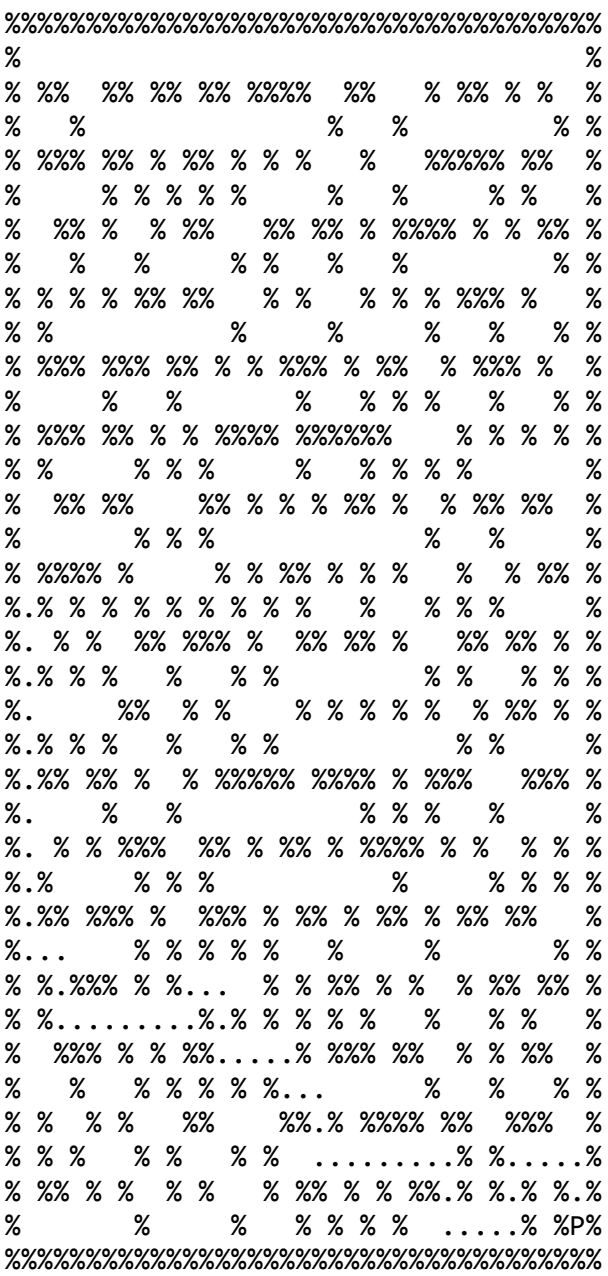
Here are the 4 updated results using our “More Informed” Heuristic:

A* Solution to smallTurns.txt (moveCost = 2, turnCost = 1)
Heuristic: More Informed



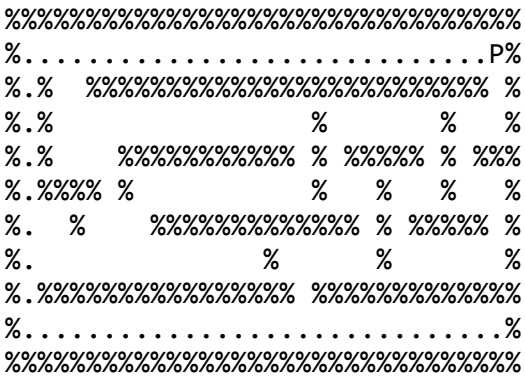
AstarTurns Nodes Expanded = 272
AstarTurns Solution Cost = 120

A* Solution to bigTurns.txt (moveCost = 2, turnCost = 1)
Heuristic: More Informed



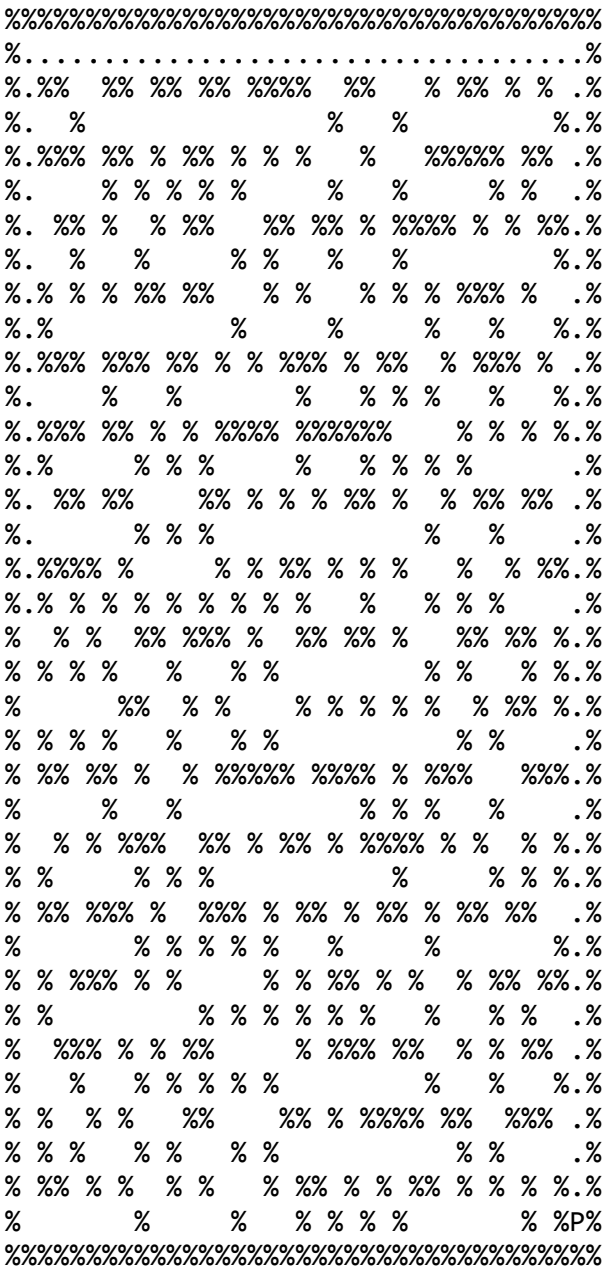
AstarTurns Nodes Expanded = 1075
AstarTurns Solution Cost = 133

A* Solution to smallTurns.txt (moveCost = 1, turnCost = 2)
Heuristic: More Informed



AstarTurns Nodes Expanded = 455
AstarTurns Solution Cost = 74

A* Solution to bigTurns.txt (moveCost = 1, turnCost = 2)
Heuristic: More Informed



AstarTurns Nodes Expanded = 1563
AstarTurns Solution Cost = 90

Notice that the optimal solutions have been found in all 8 solutions. The more informed (yet still admissible) heuristic also expanded fewer nodes in all 4 of the scenarios that we used it for.

1.3 Pacman with a Ghost

To create organized code, a **Ghost** class was created to represent everything about the **Ghost**

```
public class Ghost{
	public Point location;
	public int minX;
	public int maxX;
	Direction direction;

	//methods omitted
}
```

All of the necessary variables are updated when the **Maze** class reads in a text file maze.

The **Ghost** also has a function to move it called "getNewMovedGhost". It actually returns a new **Ghost** so that each **StateGhost** can have it's own **Ghost**. The "getNewMovedGhost" function makes it convenient to have the **Ghost** determine its own movement.

In addition, our state representation is updated (compared to Part 1.1) to include the **Ghost** information:

```
public class StateGhost{
	public Point pacmanLocation;
	public Ghost ghost;                    //added for Part 1.3
	public int costSoFar;
	public Heuristic heuristic;

	//methods omitted
}
```

Our A* search algorithm is very similar to before. The main difference is that it now deals with **StateGhost** as state representations.

To account for the **Ghost**, our **StateGhost** class has to a method called "getAdjacentStates" which only returns valid adjacent states where

1) the pacman and **Ghost** don't share the same location, and
2) where the pacman doesn't "walk through" the **Ghost** go get to its next position

Here are the results (Note: For each solution, the Pacman goes backward 1 step, then forward 1 step, to avoid the **Ghost** when he encounters the **Ghost**:

AstarGhost for smallGhost.txt

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

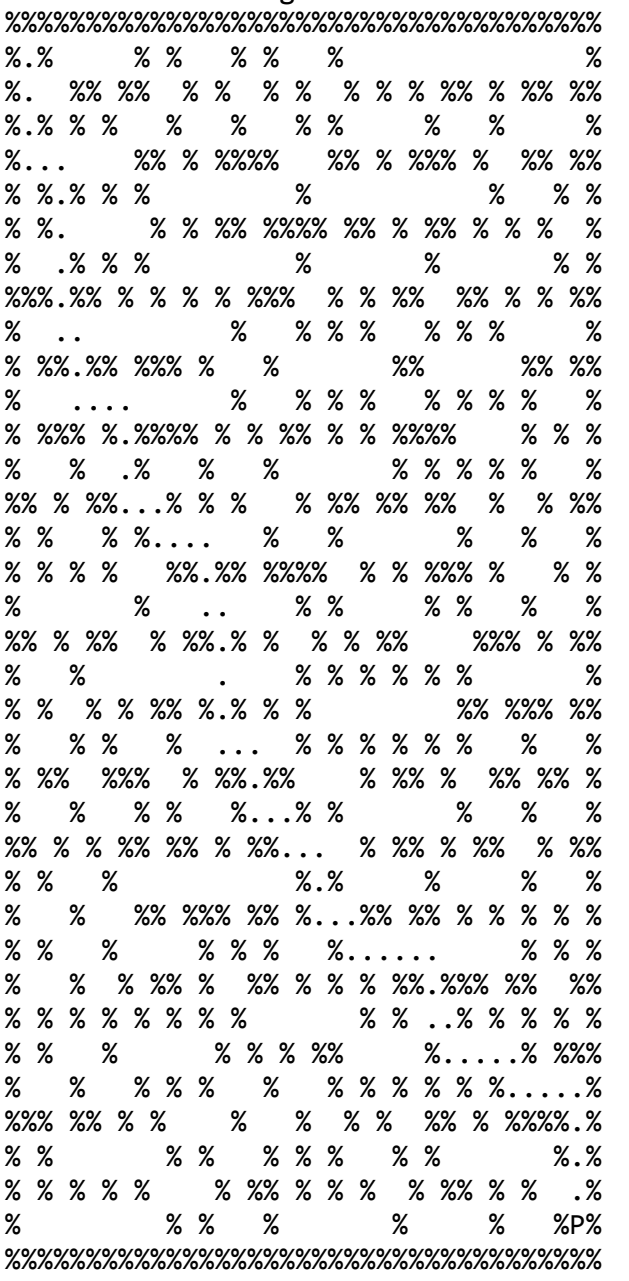
AstarGhost Nodes Expanded = 121
AstarGhost Solution Distance = 21

AstarGhost for mediumGhost.txt

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

AstarGhost Nodes Expanded = 74
AstarGhost Solution Distance = 26

AstarGhost for bigGhost.txt



AstarGhost Nodes Expanded = 308
AstarGhost Solution Distance = 70

Here are the results for running A* search on the same mazes, with the **Ghost** removed:

AstarGhost for smallGhost.txt (with the Ghost removed)

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

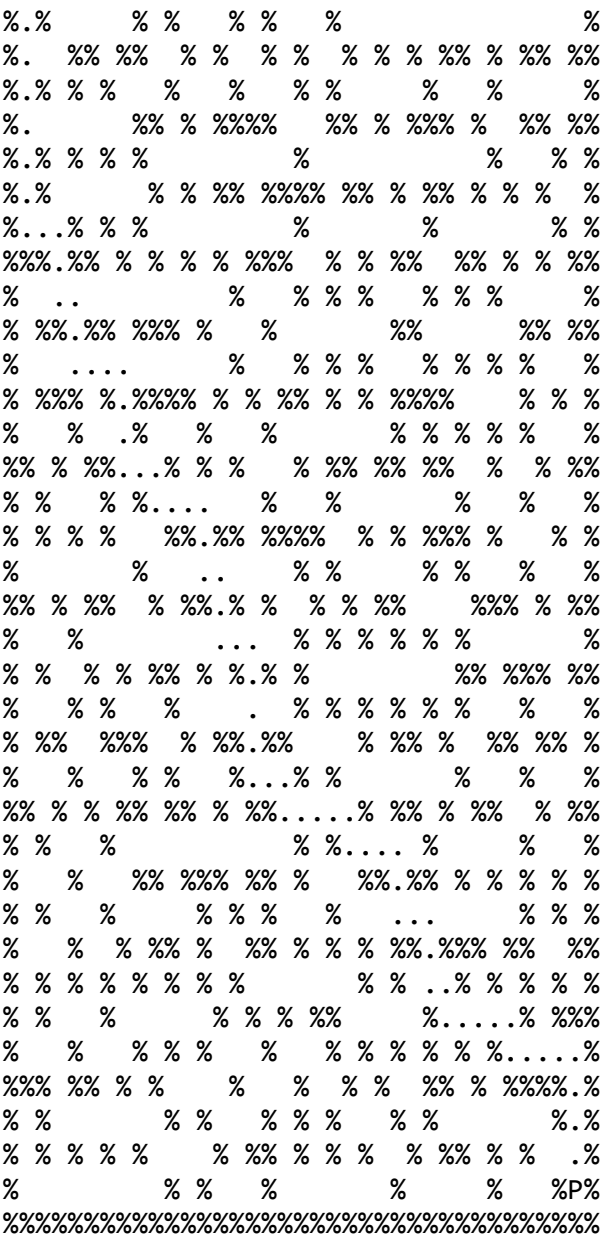
Astar Nodes Expanded = 52
Astar Solution Distance = 19

AstarGhost for mediumGhost.txt (with the Ghost removed)

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

Astar Nodes Expanded = 31
Astar Solution Distance = 24

AstarGhost for bigGhost.txt (with the Ghost removed)



Astar Nodes Expanded = 164
Astar Solution Distance = 68

In each of the 3 mazes, when the **Ghost** is removed, pacman is able to find a shorter solution to the goal. Coincidentally, it is shorter by 2 steps in each maze (which corresponds to pacman taking a step back, and a step forward so that the **Ghost** can get out of his way). The number of nodes expanded is also less for all 3 mazes.

Part 1.3 Extension – Advanced Heuristic

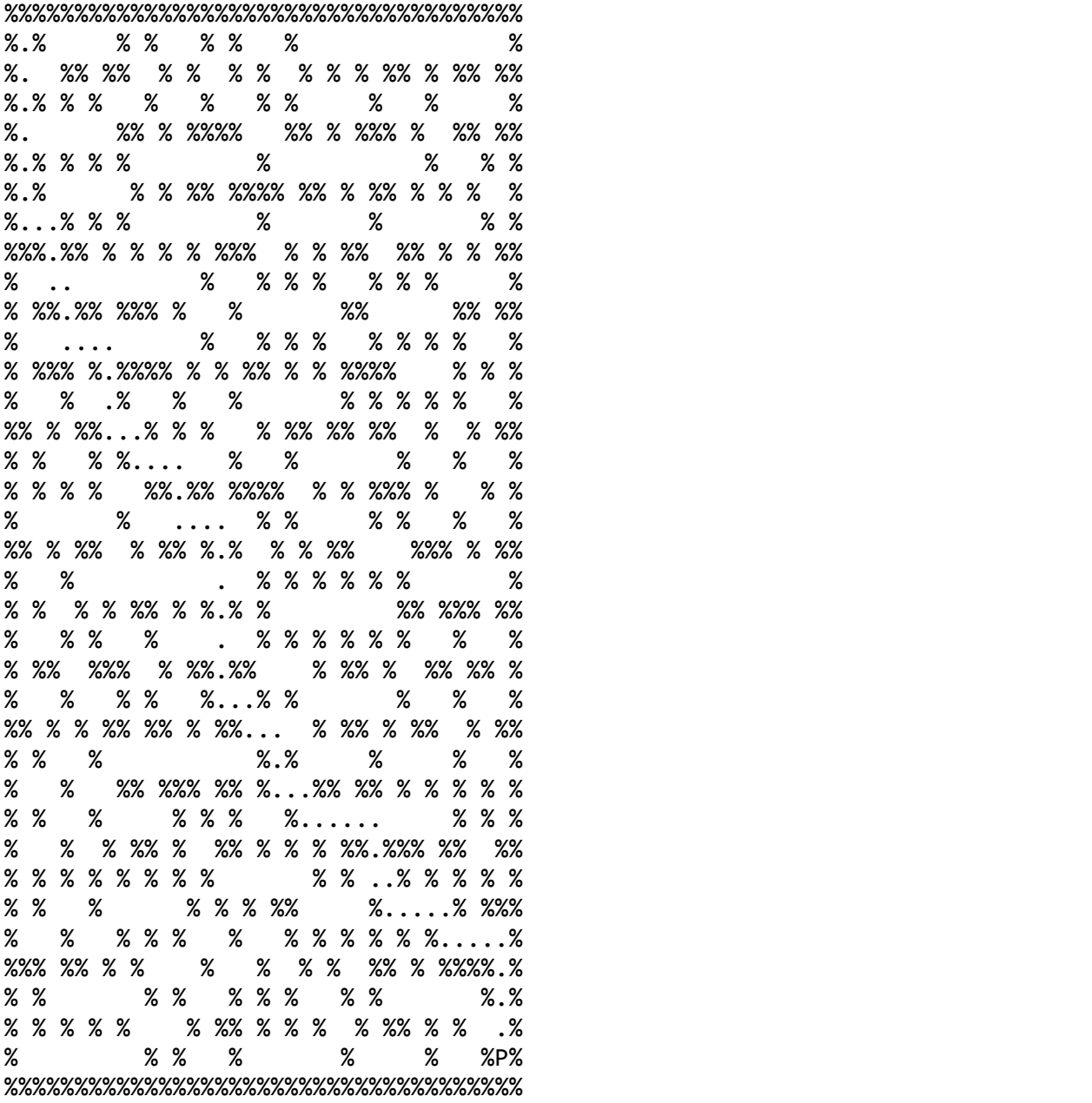
The heuristic we will use is based on the idea that we want to avoid going into a “dead end”, which we will define as a spot in the maze that is surrounded by 3 walls (and does not have a dot in it). Our heuristic is:

```
heuristic = manhattanDistance + 100*(deadEnd ? 1 : 0);
```

This will enable us to avoid dead ends. Here are the results it produces:

```
AstarGhost for smallGhost.txt
%%%%%%%%%%%%%%%%%%%%%%%%
% %%         % %       %
%    %%%%%% %.%%%%%% %
%%%%%%      P..%       %
%    % %%%%%%.%% %%%%%
% %%%% %     ..    %   %
%         %%%.%%%   % %
%%%%%%%%%%... %%%%%% %
%..........%%         %
%%%%%%%%%%%%%%%%%%%%%%%%
AstarGhost Nodes Expanded = 120
AstarGhost Solution Distance = 21


AstarGhost for mediumGhost.txt
%%%%%%%%%%%%%%%%
%.       %     %
%.%%%% %%% %  %
%.%   % %   % %
%...%% %% %%  %
% %.....  % % %
%% % %%.%%%%  %
% %     ...  % %
%   %%% %.%%% %
% %   % %.... %
% %%%     %%.%%
%   % % % % ..%
%%%%%  %%   %.%
%     %   % %P%
%%%%%%%%%%%%%%%%
AstarGhost Nodes Expanded = 57
AstarGhost Solution Distance = 26
```

AstarGhost for bigGhost.txt



AstarGhost Nodes Expanded = 242
AstarGhost Solution Distance = 70

Notice that in all 3 mazes, we expand fewer nodes, while still obtaining optimal solutions.

Part 1.3 Extension – Animations (Extra Credit)

All 3 mazes have corresponding animations (in .gif format as an “animated gif”):

- smallGhost.gif
- mediumGhost.gif
- bigGhost.gif

These animations can be found in the ZIP file in the “Animations -> Part 1.3” folder

2.1 Search with Multiple Dots (Extra Credit)

Max-of-8 Heuristic

This was the most challenging part of the assignment. I used the maximum of 8 different heuristics to try to solve the mazes.

1) Number of dots left
2) Manhattan distance to closest dot
3) Manhattan distance to farthest dot
4) (Manhattan distance to closest) + (number of dots left) – 1
5) Maxium manhattan distance between 2 dots
6) Maximum xyRange for the dots. Defined as
   xyRange = (xMax - xMin) + (yMax - yMin);
7) A heuristic I call "fancyHeuristic" (which I got the idea from reading about the "Held-Karp algorithm" for solving the traveling salesman problem). I solve the traveling salesman problem for the 1st column of dots (which is trivial). I then include the dots in subsequent columns, one column at a time, **to find a lower bound** on the minimum distance required to connect the dots.
8) A heuristic I call "fancyHeuristic2". This is the same as the previous heuristic, except it goes by rows instead of columns.

All 8 of these heuristics are **admissible**. Here is an explanation for each heuristic:

1) If there are "n" dots left, pacman needs at least "n" moves to collect the dots
2) Traveling to the closest dot is part of the solution of traveling to all the dots, so it underestimates to total solution cost
3) Traveling to the closest dot is part of the solution of traveling to all the dots, so it underestimates to total solution cost
4) Without loss of generality, traveling to the closest dot is part of the solution of traveling to all the dots. After that dot is eaten, we still have to eat "n-1" dots, which each require at least 1 step, so we can add up the distance to the closest dot and "n-1" and get an admissible heuristic. Notice that Heuristic #4 dominates Heuristic #1 and Heuristic #2
5) Pacman will have to travel this distance to collect both dots. He likely has to travel more than this distance, but underestimating the total distance he has to travel makes this heuristic admissible
6) The same reasoning for Heuristic #5 can be applied to Heuristic #6. Pacman needs to be at xMin, at xMax, at yMin, and at yMax, to collect all the dots. The number of steps required to do this underestimates the solution cost, so this heuristic is admissible.

7) Since we found a lower bound on the minimum distance to collect the dots, this heuristic is admissible.
8) This heuristic is admissible for the same reasoning as heuristic #7.

This max-of-8 admissible heuristics was able to solve the 2 mazes from last semester optimally.

```
%%%%%%%%%%%%%%%%%%%%%%
%.            ...P .%
%.%%.%%.%%.%%.%% %.%
% %% %.....      %.%
%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 333
Solution Distance = 34
```

```
%%%%%%%%%%%%%%%%%%%%%%
%.            ..%   %
%.%%.%%.%%.%%.%% % %
%         P       % %
%%%%%%%%%%%%%%%%%% %
%.....              %
%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 7769
Solution Distance = 60
```

It was also able to solve tinySearch.txt from this semester **optimally**!

tinySearch.txt

```
%%%%%%%%%%%%%%%%%%%%%%%%
%      P       .%   .. %
% %%%%%%% %%%%%% %  % %
%   .%..       .% %  % %
%.         .      %.   %
%%%%%%%%%%%%%  %%%%%%%
%..                .   .%
%%%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 157019
Solution Distance = 91
```

Branch-and-Bound Heuristic

It is also worth mentioning another excellent admissible heuristic that was used to generate solutions. Using the “branch-and-bound” technique (which I learned from Wikipedia), I tried solving the “traveling salesman problem” for the remaining dots (by ignoring the walls between them). This number was then used as an admissible heuristic. Results were cached in a HashMap for fast lookup. This method was able to solve the mazes from last semester optimally:

```
%%%%%%%%%%%%%%%%%%%%
%.            ...P .%
%.%%.%%.%%.%%.%% %.%
% %% %.....      %.%
%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 8374
Solution Distance = 34

%%%%%%%%%%%%%%%%%%%%
%.            ..%   %
%.%%.%%.%%.%%.%% % %
%        P       % %
%%%%%%%%%%%%%%%%%% %
%.....             %
%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 6699
Solution Distance = 60
```

I had very high hopes for this technique, but it was not fast enough to solve the 2 harder mazes from this semester. Let’s resort to weighted A* search.

Weighted A* search

To solve smallSearch.txt and mediumSearch.txt in a reasonable amount of time, weighted A* search was used to find excellent (although non-optimal) solutions in a matter of seconds.

The following heuristic was used:

heuristic = 100 * dotsLeft

Here are the solutions it generated:

smallSearch.txt

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

Nodes Expanded = 726
Solution Distance = 279

mediumSearch.txt

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

Nodes Expanded = 1323
Solution Distance = 459

All 3 mazes (from this semester) have corresponding animations (in .gif format as an “animated gif”):

- tinySearch.gif
- smallSearch.gif
- mediumSearch.gif

These animations can be found in the ZIP file in the “Animations -> Part 2.1” folder

2.2 Suboptimal Search (Extra Credit)

The following heuristic was used to solve the mazes:

```
heuristic = Math.max(25 * dotsLeft - farthestDot, 0);
```

It generates superb results. The requirement was to expand less than 72000 nodes for bigDots.txt. The above heuristic expands only 633 nodes!

Explanation of heuristic:

- The (25 * dotsLeft) portion is there to reward pacman for collecting dots. This makes states with fewer dots much more appealing than states with many dots.
- Subtracting "farthestDot" is a unique trick I came up with (I did mention it in a post to the instructors on Piazza, so maybe it is no longer unique). The effect of **subtracting** farthestDot is to guide pacman to eat nearby dots before it eats the farther dots.
- The "Math.max()" is there to make sure the heuristic is never negative, since our definition required the heuristic to never be negative. It is worthy to note that removing the Math.max() does not alter the results in any way.

Here are the results:

mediumDots.txt

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%...%........%%%%.............%
%%%.%...%%%.........%.%...%.%%%
%...%%%.%.%%%%.%.%%%%%%.%%%...%
%.%.....%..P...%......%.....%.%
%.%%%.%%%.%%%%%%%%%.%%%.%.%%%%%
%.....%.........%...%...%.....%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 262
Solution Distance = 170
```

bigDots.txt

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

Nodes Expanded = 633
Solution Distance = 276

Both mazes have corresponding animations (in .gif format as an animated gif):

- mediumDots.gif
- bigDots.gif

These animations can be found in the ZIP file in the “Animations -> Part 2.2” folder

(Extra Credit)

As an added bonus, I created a super complicated maze with hundreds of dots that is about 6 times bigger than any of the mazes in the assignment (it is 32 x 78, with 1500-2000 dots to eat). My Weighted A* algorithm finds a solution to this maze in about 2 minutes. The maze is shown on the next page.

Here are the results for the maze:

Nodes Expanded = 19661
Solution Distance = 2070

The animation for the solution is omitted as it is 2000+ images totaling 300+ megabytes

Here is a screenshot of the txt file of the complicated maze.