Abhishek Deep Nigam – adnigam2 (3 units)
Sakshi Agarwal – sagarwl8 (3 units)

CS 440: MAZE SEARCH

ENVIRONMENT

The environment that we set up is quite simple. We created **Maze** and **Location** objects for the search methods to take in. The bulk of the work is done in the **Maze** class. A **Maze** is formed by reading in each character from each row of a text file. Each **Location** is assigned a value based on the character read. An empty space is a space, a "%" is a wall, a "P" is the start position, and a "." is the goal position. To account for the ghost condition, a "G" will indicate the starting position of a ghost. A "g" will indicate a position that is in the range of motion of the ghost. Each **Location** in the **Maze** is then added to a two-dimensional array of **Location**s based on where it is located in the text file. As the **Maze** is being built, references to the start and goal positions are additionally stored as variables in the **Maze** object. Again, to account for the ghost condition, if a ghost is found, a reference to the start position of the ghost is stored as a variable in the **Maze** object.

A **Location** object stores each position using two integers, one that corresponds to the x-coordinate, and another that corresponds to the y-coordinate. Additionally, it contains a classifier that details the object type, and a heuristic for the A* search. The **Location** class also has a *getAdjacent* method that returns an **ArrayList** of **Location**s that neighbor the **Location** called by the method. The *getAdjacent* method ensures that only valid locations will be added to the **ArrayList.** This allows to freely access all members of the **ArrayList** without worrying about if they are valid locations in the **Maze.** A **Location** is deemed valid if it is within the bounds of the maze.

Both the **Maze** and **Location** classes also have a *toString* method which is used to print the solution. **Location**'s *toString* returns each position's character as a string, and **Maze**'s *toString* returns a string of all the *toString*'s of the **Location**'s in the array.

BASIC PATHFINDING (1.1)

Depth-First Search

Our depth-first search uses two methods: *findSolution* and *printSolution.*

Our depth-first search uses a **Stack** to maintain the frontier. A **Stack** was chosen because it allows us to maintain the first-in last-out structure needed for a Depth-First Search. In addition, we maintain an **ArrayList** named “visited” to keep track of what **Location**s in the **Maze** have already been visited. This prevents us from looking at the same node many times over. We used a **HashMap** that maps **Location** keys to **Location** values. The key is a **Location** object, and the value associated with that key is a **Location** that is predecessor of the given key.

findSolution takes in a **Maze** object and functions by pushing the starting **Location** to a **Stack** called “frontier” and adding it to an ArrayList called “visited,” which keeps track of **Location**s that have already been visited.

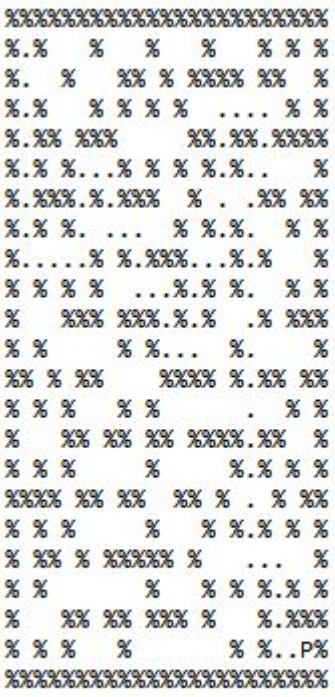
findSolution then calls a while loop which performs the actual search. The search terminates when the **Stack** is empty. A variable “cur” of type **Location** (initially the starting point) is assigned to the **Location** popped off the stack, and it is the current location being inspected. Each time a **Location** is popped off the stack, an integer is incremented to keep track of how many **Location**s have been traversed. This enables us to track the number of nodes expanded by the search. Another **ArrayList** is also created to store all the **Location**s adjacent to “cur.” A for loop that goes through these adjacent **Location**s is then called. If the **Location** being examined is not a wall, and has not been visited before, it is added to a **HashMap** called “predecessors” which takes “temp” (the **Location** going through the for loop) as the key, and “cur” as the value. The **Location** will then be pushed onto the stack, and added to the **ArrayList** of visited **Location**s. If the adjacent point is the goal, then this **Location** will be returned, and the search is complete, otherwise the process will be repeated.

printSolution is called after *findSolution*. It consists of a while loop that goes through “predecessor,” starting from the end **Location**, and sets every **Location** in the HashMap to a

dot. We ensure that it does not set the starting **Location** to a dot, allowing the starting **Location** to remain a “P.” The loop terminates once the start **Location** is reached and there are no predecessors left.

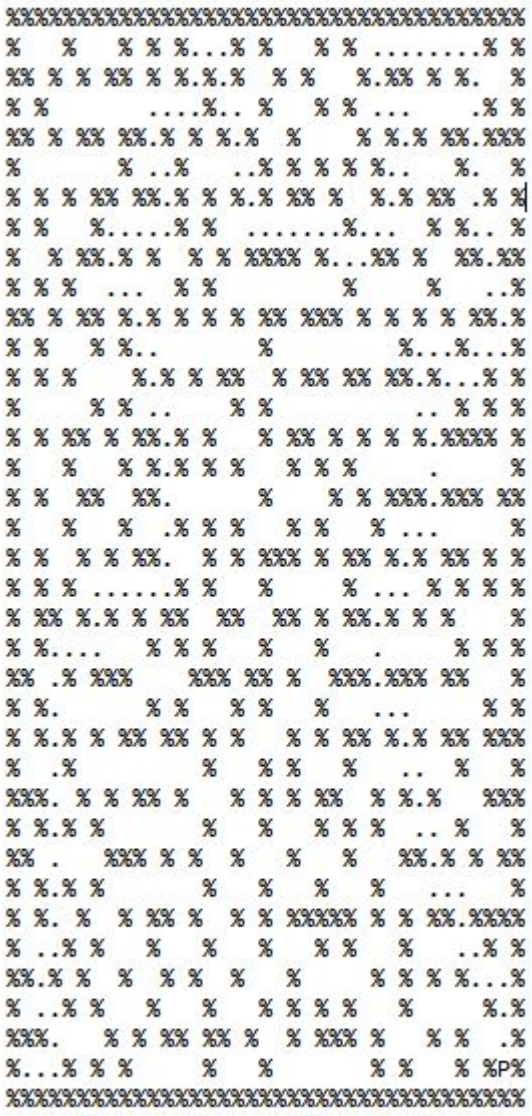
The results of DFS on the three mazes provided:

Medium Maze



```
Nodes Expanded: 74
Path cost: 64
```

DFS Big Maze



```
Nodes Expanded: 293
Path cost: 154
```

Abhishek Deep Nigam – adnigam2 (3 units)
Sakshi Agarwal – sagarwl8 (3 units)

DFS Open Maze

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

Nodes Expanded: 156
Path cost: 146
```

Breadth-First Search

Our breadth-first search is similar to our depth-first search, but it uses a **Queue** instead of a **Stack**. Despite this slight difference, BFS still uses two methods: *findSolution* and *printSolution.* The *printSolution* method is identical to that of the DFS.

Our breadth-first search uses a **Queue** backed by a **LinkedList** to maintain the frontier. A **Queue** was chosen because it allows us to maintain the first-in first-out structure needed for a Breadth-First Search. In addition, we maintain an **ArrayList** named "visited" to keep track of what **Location**s in the **Maze** have already been visited. This prevents us from looking at the same node many times over. We used a **HashMap** that maps **Location** keys to **Location** values. The key is a **Location** object, and the value associated with that key is a **Location** that is predecessor of the given key.

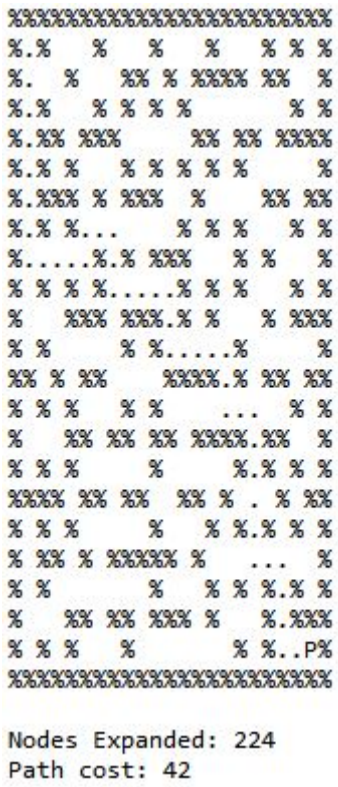
findSolution takes in a **Maze** object and functions by adding the starting **Location** to a **Queue** called "frontier" and adding it to an ArrayList called "visited," which keeps track of Locations that have already been visited.

findSolution then calls a while loop which performs the actual search. The search terminates when the **Queue** is empty. A variable "cur" of type **Location** (initially the starting point) is assigned to the **Location** popped off the **Queue**, and it is the current location being inspected. Each time a **Location** is popped off the **Queue**, an integer is incremented to keep track of how many **Location**s have been traversed. This enables us to track the number of nodes expanded by the search. Another **ArrayList** is also created to store all the **Location**s adjacent to "cur." A for loop that goes through these adjacent **Location**s is then called. If the **Location** being examined is not a wall, and has not been visited before, it is added to a **HashMap** called "predecessors" which takes "temp" (the **Location** going through the for loop) as the key, and "cur" as the value. The **Location** will then be pushed onto the **Queue**, and added to the **ArrayList** of visited **Location**s. If the adjacent point is the goal, then this **Location** will be returned, and the search is complete, otherwise the process will be repeated.

printSolution is called after *findSolution*. It consists of a while loop that goes through "predecessor," starting from the end **Location**, and sets every **Location** in the HashMap to a dot. We ensure that it does not set the starting **Location** to a dot, allowing the starting **Location** to remain a "P." The loop terminates once the start **Location** is reached and there are no predecessors left.

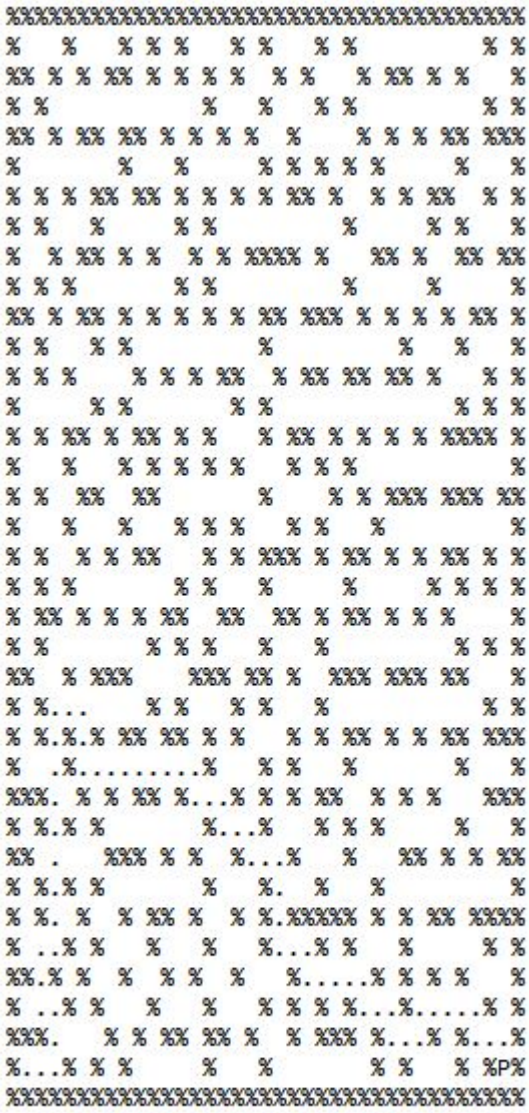
The results of BFS on the three mazes provided:

Medium Maze



```
Nodes Expanded: 224
Path cost: 42
```

BFS Big Maze



```
Nodes Expanded: 716
Path cost: 62
```

Abhishek Deep Nigam – adnigam2 (3 units)
Sakshi Agarwal – sagarwl8 (3 units)

BFS Open Maze

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

Nodes Expanded: 272
Path cost: 54
```

Greedy Best-First Search

Our greedy best-first search is similar to our breadth-first search, but it uses a **Priority Queue** instead of a queue. Unlike in the BFS, this search uses a **Comparator** based on the Manhattan Distance from current **Location** to end **Location** to determine which point is to be returned. Despite these differences, the greedy best-first search still uses two methods: *findSolution* and *printSolution.* The *printSolution* method is identical to that of the BFS.

Our greedy best-first search uses a **Priority Queue** to maintain the frontier. A **Priority Queue** was chosen because it allows us to use a custom **Comparator** (described above) to maintain the ordering of the **Priority Queue**. In addition, we maintain an **ArrayList** named "visited" to keep track of what **Location**s in the **Maze** have already been visited. This prevents us from looking at the same node many times over. We used a **HashMap** that maps **Location** keys to **Location** values. The key is a **Location** object, and the value associated with that key is a **Location** that is predecessor of the given key.

findSolution takes in a **Maze** object and functions by adding the starting **Location** to a **Priority Queue** called "frontier" and adding it to an ArrayList called "visited," which keeps track of Locations that have already been visited

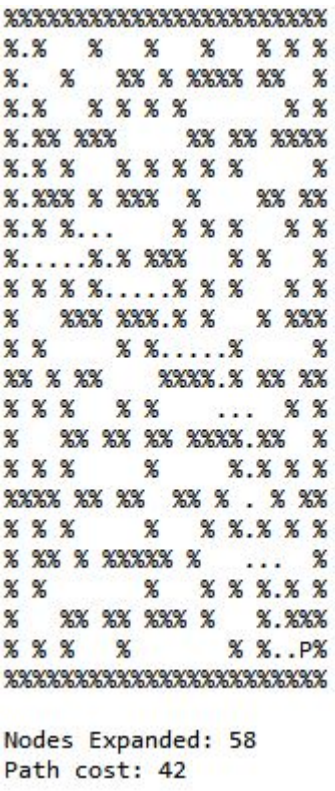
findSolution then calls a while loop which performs the actual search. The search terminates when the **Priority Queue** is empty. A variable "cur" of type **Location** (initially the starting point) is assigned to the **Location** popped off the **Priority Queue**, and it is the current location being inspected. Each time a **Location** is popped off the **Priority Queue**, an integer is incremented to keep track of how many **Location**s have been traversed. This enables us to track the number of nodes expanded by the search. Another **ArrayList** is also created to store all the **Location**s adjacent to "cur." A for loop that goes through these adjacent **Location**s is then called. If the **Location** being examined is not a wall, and has not been visited before, it is added to a **HashMap** called "predecessors" which takes "temp" (the **Location** going through the for loop) as the key, and "cur" as the value. The **Location** will then be pushed onto the **Priority Queue**,

and added to the **ArrayList** of visited **Location**s. If the adjacent point is the goal, then this **Location** will be returned, and the search is complete, otherwise the process will be repeated.

printSolution is called after *findSolution*. It consists of a while loop that goes through "predecessor," starting from the end **Location**, and sets every **Location** in the HashMap to a dot. We ensure that it does not set the starting **Location** to a dot, allowing the starting **Location** to remain a "P." The loop terminates once the start **Location** is reached and there are no predecessors left.

The results of Greedy Best-First Search on the three mazes provided:

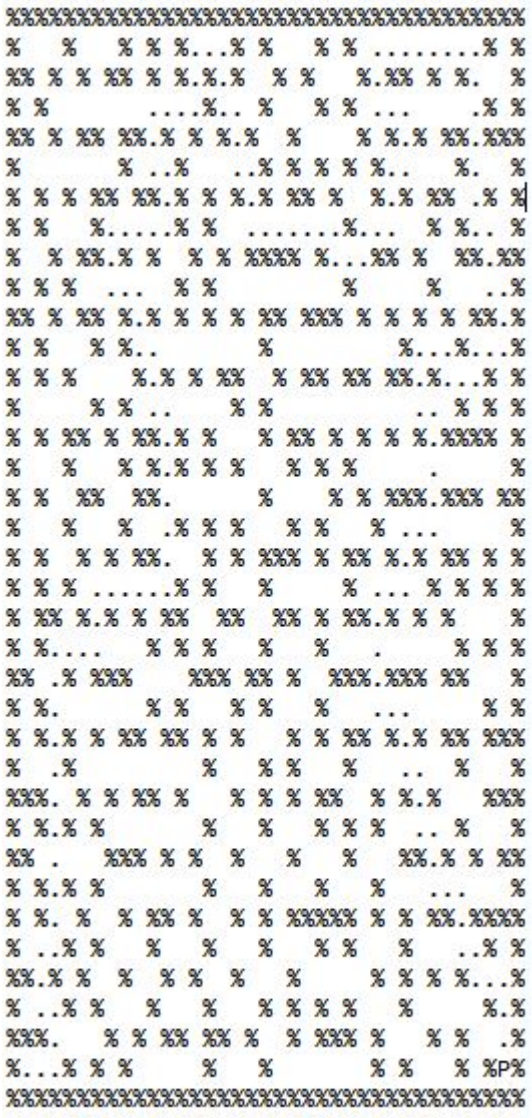
Medium Maze



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

```
Nodes Expanded: 58
Path cost: 42
```

Big Maze



```
Nodes Expanded: 293
Path cost: 154
```

Open Maze

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

```
Nodes Expanded: 122
Path cost: 58
```

A* (Astar) Search

Our A* search is similar to our greedy best-first search, but the comparator differs slightly. The comparator still takes into account the Manhattan Distance from the current location to the end point, but A* additionally uses distance travelled to reach the current **Location**. The two values are added together to determine which **Location** is to be removed from the priority queue. Despite this difference, the implementation of A* still uses two methods: *findSolution* and *printSolution.* The *printSolution* method is identical to that of the greedy best-first search.

Our A* search uses a **Priority Queue** to maintain the frontier. A **Priority Queue** was chosen because it allows us to use a custom **Comparator** (described above) to maintain the ordering of the **Priority Queue**. In addition, we maintain an **ArrayList** named "visited" to keep track of what **Location**s in the **Maze** have already been visited. This prevents us from looking at the same node many times over. We used a **HashMap** that maps **Location** keys to **Location** values. The key is a **Location** object, and the value associated with that key is a **Location** that is predecessor of the given key.

findSolution takes in a **Maze** object and functions by adding the starting **Location** to a **Priority Queue** called "frontier" and adding it to an ArrayList called "visited," which keeps track of Locations that have already been visited.

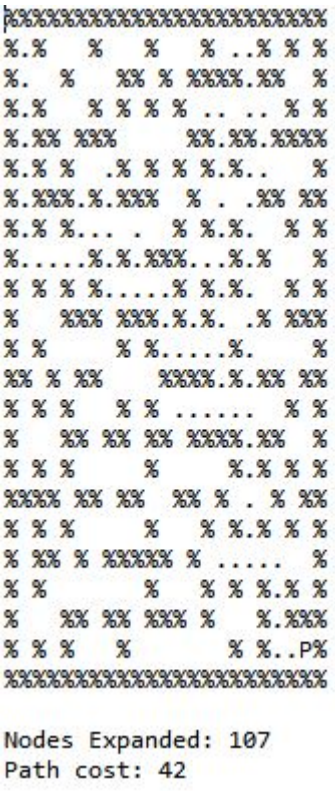
findSolution then calls a while loop which performs the actual search. The search terminates when the **Priority Queue** is empty. A variable "cur" of type **Location** (initially the starting point) is assigned to the **Location** popped off the **Priority Queue**, and it is the current location being inspected. Each time a **Location** is popped off the **Priority Queue**, an integer is incremented to keep track of how many **Location**s have been traversed. This enables us to track the number of nodes expanded by the search. Another **ArrayList** is also created to store all the **Location**s adjacent to "cur." A for loop that goes through these adjacent **Location**s is then called. If the **Location** being examined is not a wall, and has not been visited before, it is added to a **HashMap** called "predecessors" which takes "temp" (the **Location** going through the for loop) as the key, and "cur" as the value. The **Location** will then be pushed onto the **Priority Queue**,

and added to the **ArrayList** of visited **Location**s. If the adjacent point is the goal, then this **Location** will be returned, and the search is complete, otherwise the process will be repeated.

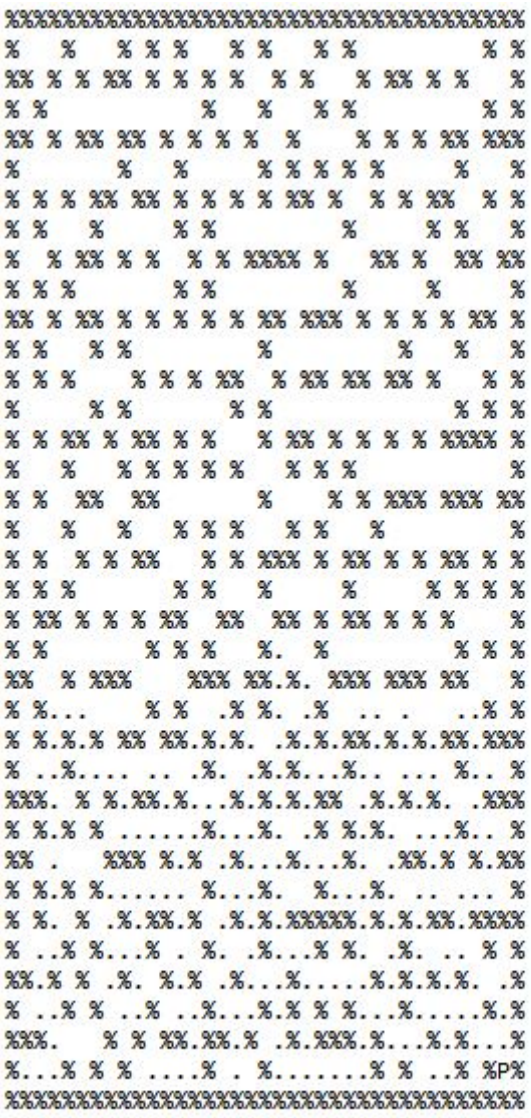
printSolution is called after *findSolution*. It consists of a while loop that goes through "predecessor," starting from the end **Location**, and sets every **Location** in the HashMap to a dot. We ensure that it does not set the starting **Location** to a dot, allowing the starting **Location** to remain a "P." The loop terminates once the start **Location** is reached and there are no predecessors left.

The results of A* Search on the three mazes provided:

Medium Maze

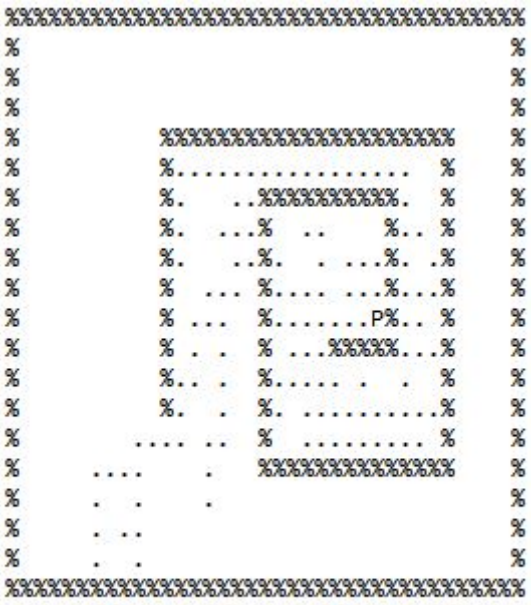


A* Big Maze



```
Nodes Expanded: 285
Path cost: 64
```

A* Open Maze



PENALIZING TURNS (1.2)

Our implementation of 1.1 was modified to be able to set specific costs to turning and moving forward. In most ways, the search is performed in the same way as it originally was performed. However, the A* search was modified so that each A* object can hold a turnCost and forwardCost. When you take into account turn cost and forward cost, the cost of your solution is not merely equal to the amount of **Location**s in the solution path. You must calculate solution cost by taking into account the turn cost and forward cost. That is exactly what we did here.

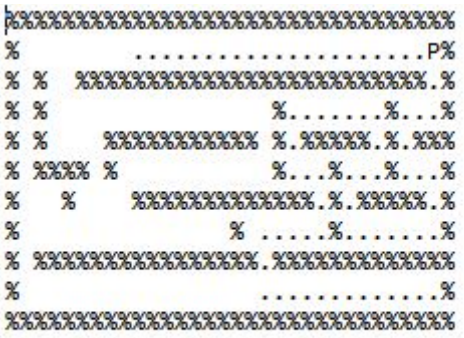
To calculate solution cost, we traverse from the starting point to the ending point. When a turn is necessary, the amount of turns necessary is calculated. For example, if the current direction is "right," and the next point in the solution path is above the current point, then a turn is necessary. You would need to turn to be facing "up" and then move forward. Only 1 turn is necessary to turn from "right" to "up." Then, to calculate the cost of that turn, you multiply 1 by the turnCost. You must add this value as well as the cost of moving forward to the solutionCost variable. This algorithm allows us to calculate the cost of a turn as well as the cost of moving one step forward after the turn. The method that determines how many turns are necessary also takes into account if no turns are necessary. For example, if you are facing "right," and the next point on the solution path is to the right, then no turns are necessary. This condition is accounted for, which would mean only the cost of moving forward is added to the solutionCost.

The next task in 1.2 is to implement a heuristic that is more informed than Manhattan distance. We implemented a heuristic that takes the original heuristic for A*, and weights it based on the cost of reaching the current node. This heuristic is admissible because it never overestimates the cost to reach the goal. The heuristic is more informed than the Manhattan distance because it takes into account the cost (including turns and moves forward) to reach the current node. Because you know the cost of reaching your current node, there is no reason you should not factor that cost into the heuristic you choose to use.

If you notice below, our new heuristic, designated Heuristic 2, always expands fewer nodes than the standard heuristic, designated Heuristic 1. In addition, Heuristic 2 has a smaller solution cost than Heuristic 1.

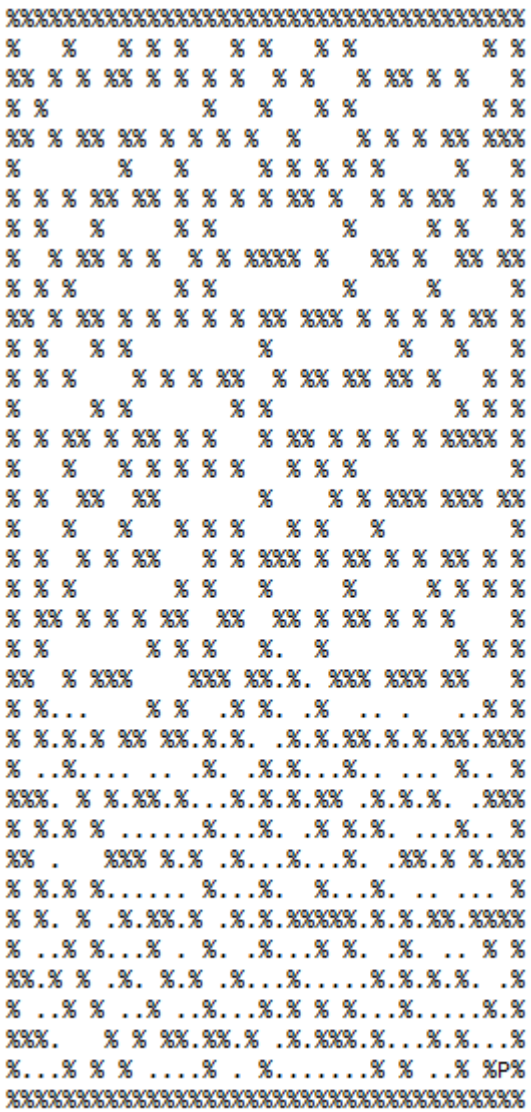
Case 1 below is where forward movement has cost 2 and any turn has cost 1. Case 2 below is where forward movement has cost 1 and any turn has cost 2.

Small Turns, Case 1, Heuristic 1



```
Nodes Expanded: 74
Solution cost: 120
```

Big Maze, Case 1, Heuristic 1

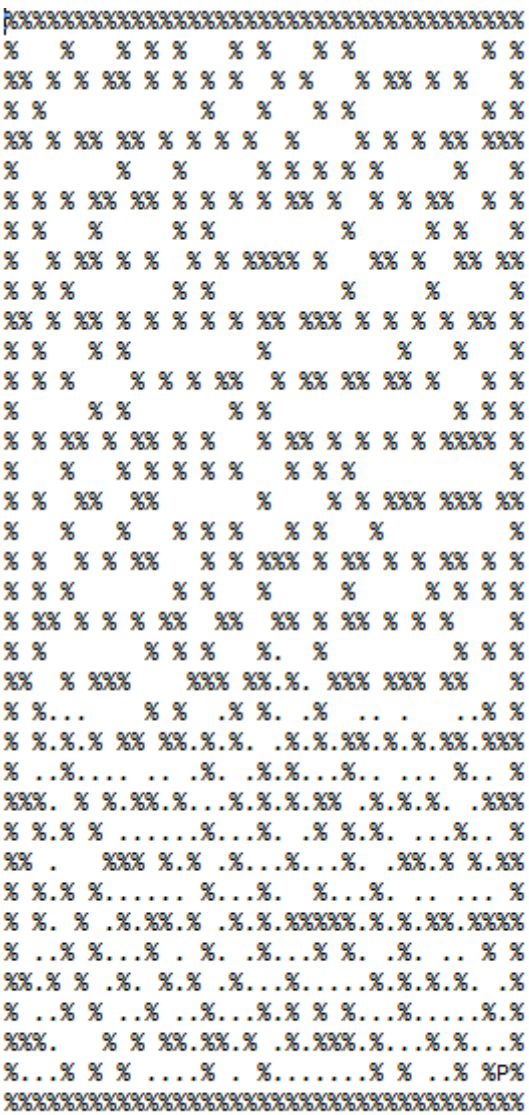


```
Nodes Expanded: 285
Solution cost: 379
```

Small Turns, Case 2, Heuristic 1

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

Nodes Expanded: 74
Solution cost: 84
```

Big Maze, Case 2, Heuristic 1

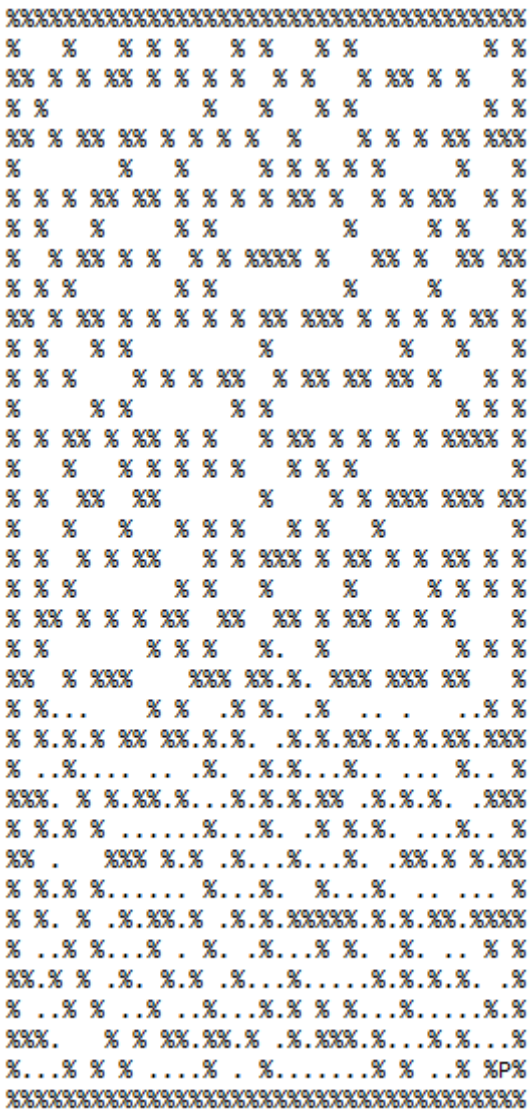


```
Nodes Expanded: 285
Solution cost: 312
```

Small Turns, Case 1, Heuristic 2

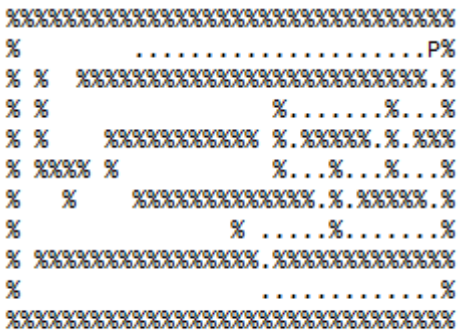
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        .....................P%
% %  %%%%%%%%%%%%%%%%%%%%%%%%%.%
% %                %.......%...%
% %    %%%%%%%%%%% %.%%%%%.%.%%%
% %%%% %           %...%...%...%
%   %    %%%%%%%%%%%%%.%.%%%%%.%
%               % .....%.......%
% %%%%%%%%%%%%%%%%.%%%%%%%%%%%%%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

Nodes Expanded: 56
Solution cost: 99
```

Big Maze, Case 1, Heuristic 2



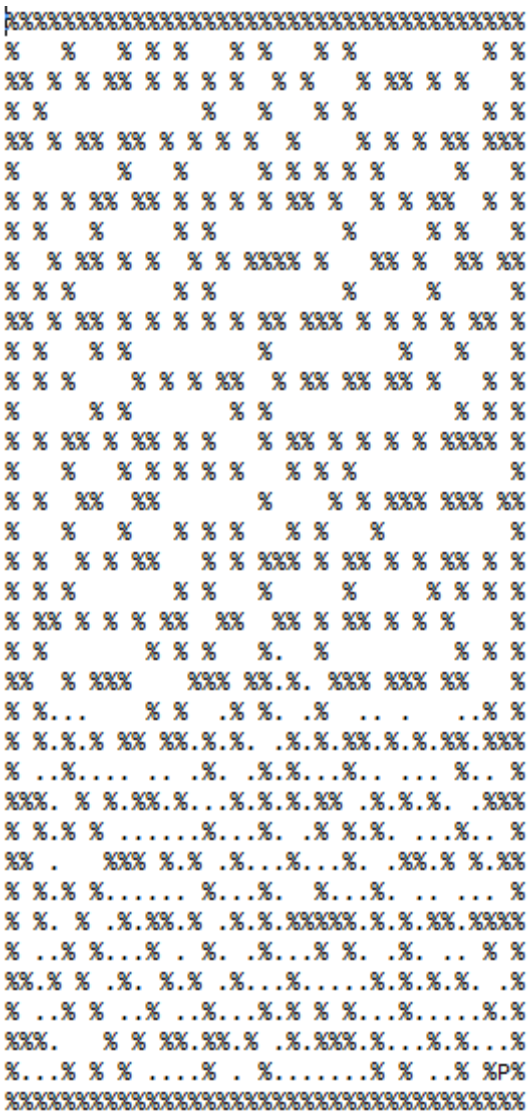
```
Nodes Expanded: 192
Solution cost: 226
```

Small Turns, Case 2 Heuristic 2




```
Nodes Expanded: 56
Solution cost: 68
```

Big Maze, Case 2 Heuristic 2



```
Nodes Expanded: 192
Solution cost: 206
```

PACMAN WITH A GHOST (1.3)

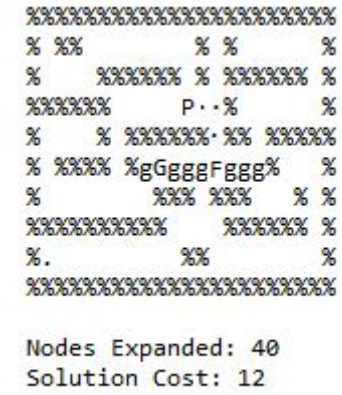
The task for 1.3 was to allow the maze to have a ghost. The ghost will have a fixed range of movement, which is left to right. We modified our maze class to accommodate for the ghost. That is, when the **Maze** is being constructed, it stores the 'g' and 'G' characters into the **Location** where that classifier was encountered. In addition, when a **Location** with a 'G' is encountered, that **Location** is assigned a reference named ghostStart within the **Maze** object. This allows us to access the starting point of the ghost when needed.

To account for the ghost, we implemented a method named *ghostSolver*. The *ghostSolver* method traverses through the maze along the calculated solution path. Every time the current **Location** of the Pacman is updated, the **Location** ghost is updated as well. If the current **Location** of the Pacman is ever the current **Location** of the ghost, then the game is over. In addition, if Pacman and the ghost ever pass through each other, the game also ends. We accommodated for this by stating that if the last **Location** of the Pacman is ever the current **Location** of the ghost, then the game is over. In the game over case, we still print the full solution path. This is the path that Pacman intended to take had the ghost not got in the way. The **Location** at which Pacman 'died,' (when the game ended) is labelled with an 'F' to signify the failure state. To reiterate, the complete planned path is printed, but the 'F' signifies where Pacman collided or passed through the ghost. If Pacman collides with the ghost or passes through the ghost, then the *ghostSolver* method returns the number of steps Pacman took before the game ended. If Pacman reaches the solution, *ghostSolver* returns the number of steps Pacman took to reach the solution. This is the solution cost.

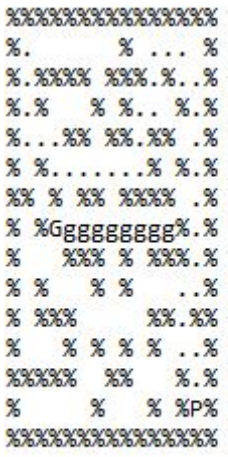
We included support for more complicated ghost movements as well as for multiple ghosts. We modified our code for the original ghost movements to allow the ghost to not only move left and right, but to also move up and down. In addition, we added to our functionality to allow for two ghosts. Both of these conditions could make the environment more difficult for the Pacman.

The results of A* Search with Manhattan Distance on the three mazes provided:

Small Ghost

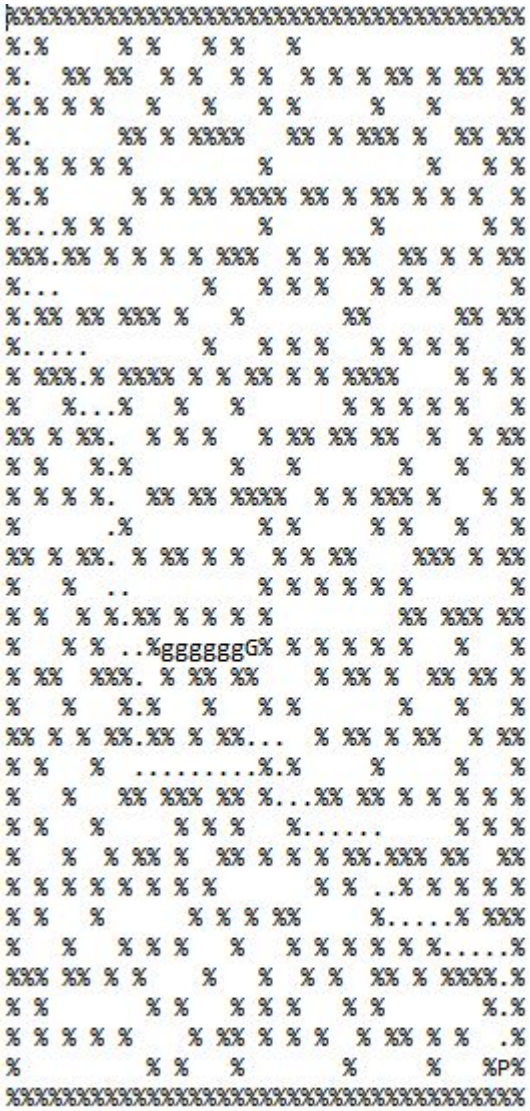


Medium Ghost



```
Nodes Expanded: 73
Solution Cost: 34
```

Big Ghost



```
Nodes Expanded: 136
Solution Cost: 74
```

For the Big maze, the number of nodes expanded is less for the maze with the ghosts, but the solution cost is greater. For both the Small and Medium mazes, the number of nodes expanded and solution costs are less for the mazes with ghosts.

Modifying this code to accommodate for even more ghosts would actually be possible. We have implemented two ghosts that support movements in the four cardinal directions. However, you could easily define an **ArrayList** of ghosts. When constructing the maze, every time a 'G' was encountered, you would add to the **ArrayList.** Then, when running the *ghostSolver*, you would compare the current **Location** of Pacman and the last **Location** of Pacman to the **Location** of every single ghost in the **ArrayList**. You could even define ghosts that were not limited to a defined path; they could move around all over the maze. Currently, the ghost can only move in the locations specified by the characters 'g' and 'G.' You could remove this restriction and allow the ghost to move throughout all locations. Doing that would force the Pacman to consider the location of the ghost in its heuristic. The above implementation of an **ArrayList** of ghosts could easily be combined with ghosts that had their own heuristic function. For example, the ghost's path could be determined based on the ghost's Manhattan Distance to the Pacman. These ghosts would then target the Pacman. The possibilities for expansion are endless when considering the ghosts in Pacman. This is for bonus points! ☺

We've also included three cutesy little gifs as animations. For your convenience, these gifs can be found at the following public URL on Dropbox. The account is linked to the email address adn5327@gmail.com, which is Abhishek's personal account. Link: Pacman GIFs. This is for bonus points! ☺

We also implemented a heuristic that calculates the Euclidean Distance rather than the Manhattan Distance. We considered using this in the assignment as well, and ultimately chose not to. However, we did want to implement it because it is a closer representation of the 'straight-line distance' that we have seen in the textbook and in class. This heuristic is no more informed than the Manhattan Distance, as they both solely take into account the distance from a point to the goal. We brainstormed several heuristics that could potentially make Pacman more informed. If Pacman was run with a ghost, you could design a heuristic that factors in the

position of the ghost. This could potentially be the original heuristic used in A* plus the Manhattan Distance from the Pacman to the Ghost. This is for bonus points! ☺

INDIVIDUAL CONTRIBUTION

Abhishek Nigam (adnigam2)

I was responsible for developing the environment. This includes the **Location** and **Maze** classes. I also implemented the **Location Comparator** and **Astar Comparator**. I implemented the four search algorithms, and integrated them tightly with the **Maze** and **Location** classes. To determine how to accurately manage a list of predecessors, I explored the option of using a **HashMap** as well as using a **Two-Dimensional Array**. I implemented the Manhattan Distance heuristic. I created the animated gifs seen above and explored the extra content for how this game could be expanded in the future.

Sakshi Agarwal (sagarwl8)

I implemented the heuristic that takes into account the cost of reaching the current node in addition to the standard Manhattan Distance. I also implemented a Euclidean Distance heuristic as a potential future heuristic. I did a majority of testing as well as debugging. I wrote all the test cases and verified the correctness of their output. I implemented the methods that would traverse the **Maze** from start to finish to calculate solution cost when taking into account the turnCost and forwardCost. I designed the new mazes for 1.3.