Maze Search

Aravind Sundaresan – sundrsn2 (3 Units)
Krish Masand – masand2 (3 Units)
Rodney Shaghoulian – shaghou2 (3 Units)

1.0 Setting up the Environment

To make our search methods more intuitive, we created `Point` and `Maze` objects for our search method to take in. 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. A "%" is a wall, a space is empty space, a "." is a dot that is used either as the end `Point` or one of the `Point`s that must be traverse depending on the search methods, and a "P" is the starting `Point`. We determine the number of columns in the `Maze` by counting the number of characters in the first line of the text file, and we determine the number of rows in the `Maze` by counting the number of lines in the text file. Each `Point` in a `Maze` is assigned to a certain spot in a two-dimensional array of `Point`s based on its location in the text file.

A `Maze` also stores a reference to the start `Point`, the end `Point` (only used in 1.1 and 1.2) and an ArrayList of references to each dot (only used in 1.3).

In addition to storing its value, a `Point` object also stores its location with two integers, one that corresponds to its x coordinate, and another corresponds to its y coordinate. A `Point` also has a method called "*getAdjacentPoints*" that returns a Vector of the `Point`s that are next to the `Point` that calls the method. *getAdjacentPoints* checks to see if the elements next to the `Point` that called the method in the array are valid `Point`s that are in valid positions (not out of bounds) and adds them to the Vector that it returns. It first checks the `Point` that should be to the right of it, then the left of it, then the `Point` below it, and then above of it.

The `Point` and `Maze` classes both have a *toString()* method, which is what we use to print the solution. `Point`'s *toString* simply returns its corresponding character value in the form of a String, and `Maze`'s *toString* returns a String comprised of all of the *toString*s of the `Point`s in the array of `Point`s, with a new line after each row.

1.1 Basic Pathfinding

Depth First Search

Our depth first search uses two methods: "*findSolution*" and "*getSolution*."

findSolution operates by taking in a `Maze` object and pushing the starting `Point` to a `Stack` and adding it to a Vector called "visited," which keeps track of the `Point`s that have already been traversed.

findSolution then calls a while loop, which performs the actual searching algorithm. The loop lasts until the `Stack` is empty. A variable "`currentPoint`" of type `Point` is created that keeps track of the `Point` that is currently being inspected by being assigned to the `Point` that is popped off of the `Stack`. (`currentPoint` is initially the starting `Point` since the starting `Point` was the only `Point` on the `Stack` before the while loop started). Every time a `Point` is popped off of the `Stack`, an integer that keeps track of the number of `Point`s that have been traversed is incremented. A nested for loop that goes through the current `Point`'s adjacent `Point`s is then called. If the `Point` that the loop is going through is empty (not a wall or a dot) or is a dot, and if the `Point` has not previously been visited, the following things will happen:

- The `Point` will be pushed to the `Stack`
- The `Point` will be added to the Vector of visited `Point`s
- The `Point` is added to a `HashMap` called "predecessor" which takes the `Point` that the for loop was going through as the value as the key, and `currentPoint` (the `Point` whose *getAdjacentPoints* was called).
    - The reasoning behind this is that *predecessor(*`Point`*)* will return the `Point` that was visited before the currently visited `Point`.
    - predecessor is used again in *getSolution*
- If the `Point` is the end `Point`, the `Point` will be returned.

The "last in, first out" nature of `Stack`s means that the last adjacent `Point` added to the `Stack` will be the next `Point` acted on by the loop, making this perfect for depth first search. By the end of *findSolution*, the end will have been found (and will be what `currentPoint` is currently pointing to), and the path from the start to the end will be stored in predecessor.

getSolution is then called. *getSolution* starts a while loop that goes through predecessor starting from the end `Point`, and sets everything on the path to a dot. The loop ends on the starting `Point`, as there is no predecessor to the starting `Point`.

The result of Depth First Search on the three provided Mazes is:

```
Small Maze
%%%%%%%%%%%%%%%%%%%%%%
% %%        % %      %
%    %%%%%% % %%%%%% %
%%%%%%.....P  %      %
%    %.%%%%%% %% %%%%%
% %%%%.%.....    %   %
%     ...%%%.%%%   % %
%%%%%%%%%%... %%%%%% %
%..........%%        %
%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 53
Path Cost = 29


Medium Maze
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                 P%
% %%%%%%%%%%%%%%%%%%%%%%% %%%%%%%%.%
% %%   %   %......%%%%%%%   %%.....%
% %% % % % %.%%%%.%%%%%%%%% %%.%%%%%
% %% % % % %...  ....    %% %%.....%
% %% % % % % %.%%%% .%%%    %%%%%%.%
% %  % % %   %....%%.%%%%%%%% .....%
% %% % % %%%%%%%%.%%........%%.%%%%%
% %% %   %%.......%%%%%%%%%.%%.....%
%    %%%%%%.%%%%%%%......%%.%%%%%%.%
%%%%%%......%.......%%%%.%%.% .....%
%......%%%%%%.%%%%% %....%%.%%.%%%%%
%.%%%%%%......%.......%%%%%.%%.....%
%........%%%%%%.%%%%%%%%%%%.%%  %%.%
%%%%%%%%%%......           .%%%%%%.%
%..........%%%%%%%%%%%%%%%%........%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 254
Path Cost = 162
```

Big Maze

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

Nodes Expanded = 469
Path Cost = 210

Breadth First Search

Our breadth first search is very similar to our depth first search with one main difference – it uses a `Queue` instead of a `Stack`. The reasoning behind this is that because the `Point` returned from removing it from a `Queue` will be the next `Point` that was added to the `Queue` after the previous `Point` that was acted on, making it more suitable for Breadth First Search. Despite this, it still uses the two methods: "*findSolution*" and "*getSolution*." The latter is exactly the same as *getSolution* used in Depth First Search, while the former slightly differs.

findSolution operates by taking in a `Maze` object and adding the starting `Point` to a `Queue` and adding it to a Vector called "visited," which keeps track of the `Point`s that have already been traversed.

findSolution then calls a while loop, which performs the actual searching algorithm. The loop lasts until the `Queue` is empty. A variable "`currentPoint`" of type `Point` is created that keeps track of the `Point` that is currently being inspected by being assigned to the `Point` that is removed from the `Queue`. (`currentPoint` is initially the starting `Point` since the starting `Point` was the only `Point` on the `Queue` before the while loop started). Every time a `Point` is removed from the `Queue`, an integer that keeps track of the number of `Point`s that have been traversed is incremented. A nested for loop that goes through the current `Point`'s adjacent `Point`s is then called. If the `Point` that the loop is going through is empty (not a wall or a dot) or is a dot, and if the `Point` has not previously been visited, the following things will happen:

- The `Point` will be added to the `Queue`
- The `Point` will be added to the Vector of visited `Point`s
- The `Point` is added to a `HashMap` called "predecessor" which takes the `Point` that the for loop was going through as the value as the key, and `currentPoint` (the `Point` whose *getAdjacentPoints* was called).
    - The reasoning behind this is that *predecessor(*`Point`*)* will return the `Point` that was visited before the currently visited `Point`.
    - predecessor is used again in *getSolution*
- If the `Point` is the end `Point`, the `Point` will be returned.

The "first in, first out" nature of `Queue`s means that the first adjacent `Point` added to the `Queue` will be the next `Point` acted on by the loop followed by the next adjacent `Point`, making this perfect for breadth first search. By the end of *findSolution*, the end will have been found (and will be what `currentPoint` is currently pointing to), and the path from the start to the end will be stored in predecessor.

getSolution is then called, and the `Maze` solution is ready to be printed.

The result of Breadth First Search on the three provided Mazes is:

```
Small Maze
%%%%%%%%%%%%%%%%%%%%%%
% %%        % %      %
%    %%%%%% % %%%%%% %
%%%%%%     P..%      %
%    % %%%%%%.%% %%%%%
% %%%% %    ..   %   %
%        %%%.%%%   % %
%%%%%%%%%%... %%%%%% %
%..........%%        %
%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 90
Path Cost = 19
```

```
Medium Maze
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                        .........P%
% %%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%% %
% %%   %   %      %%%%%%%...%%     %
% %% % % % % %%%% %%%%%%%%%.%% %%%%%
% %% % % % %        .....%%.%%     %
% %% % % % % % %%%% .%%%....%%%%%% %
% %  % % %   %    %%.%%%%%%%%      %
% %% % % %%%%%%%% %%........%% %%%%%
% %% %   %%       %%%%%%%%%.%%     %
%    %%%%%% %%%%%%%      %%.%%%%%% %
%%%%%%      %       %%%% %%.%      %
%      %%%%%% %%%%% %    %%.%% %%%%%
% %%%%%%      %       %%%%%.%%     %
%        %%%%%% %%%%%%%%%%%.%%  %% %
%%%%%%%%%%..................%%%%%% %
%..........%%%%%%%%%%%%%%%%        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 266
Path Cost = 68
```

Big Maze

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

Nodes Expanded = 618
Path Cost = 210
```

Greedy Best-First Search

Our Greedy Best-First Search is very similar to our Breadth First Search, with one exception – instead of a `Queue`, our Greedy Best-First used a `PriorityQueue`. Unlike a `Queue`, it uses a Comparator to determine which element is returned when remove() is called. In this case, our comparator is based on Manhattan Distance from the `Point` to the end `Point`. Therefore, the `Point` from the `PriorityQueue` that is removed and acted on in the loop will always be the `Point` closest to the end `Point`. Despite the differences, it still uses two similar methods: "*findSolution*" and "*getSolution*." The latter is exactly the same as *getSolution* used in Depth First Search, while the former slightly differs.

findSolution operates by taking in a `Maze` object and adding the starting `Point` to a `PriorityQueue` and adding it to a Vector called "visited," which keeps track of the `Point`s that have already been traversed.

findSolution then calls a while loop, which performs the actual searching algorithm. The loop lasts until the `PriorityQueue` is empty. A variable "`currentPoint`" of type `Point` is created that keeps track of the `Point` that is currently being inspected by being assigned to the `Point` that is removed from the `PriorityQueue`. (`currentPoint` is initially the starting `Point` since the starting `Point` was the only `Point` on the `Queue` before the while loop started). Every time a `Point` is removed from the `PriorityQueue`, an integer that keeps track of the number of `Point`s that have been traversed is incremented. A nested for loop that goes through the current `Point`'s adjacent `Point`s is then called. If the `Point` that the loop is going through is empty (not a wall or a dot) or is a dot, and if the `Point` has not previously been visited, the following things will happen:

- The `Point` will be added to the `PriorityQueue`
- The `Point` will be added to the Vector of visited `Point`s
- The `Point` is added to a `HashMap` called "predecessor" which takes the `Point` that the for loop was going through as the value as the key, and `currentPoint` (the `Point` whose *getAdjacentPoints* was called).
    - The reasoning behind this is that *predecessor(*`Point`*)* will return the `Point` that was visited before the currently visited `Point`.
    - predecessor is used again in *getSolution*
- If the `Point` is the end `Point`, the `Point` will be returned.

The nature of the `PriorityQueue` means that the `Point` closest to the end `Point` that hasn't been acted on yet will be the next `Point` acted on by the loop followed by the next adjacent `Point`, making this perfect for Greedy Best-First search. By the end of *findSolution*, the end will have been found (and will be what `currentPoint` is currently pointing to), and the path from the start to the end will be stored in predecessor.

getSolution is then called, and the `Maze` solution is ready to be printed.

The result of Greedy Best-First Search on the three provided Mazes is:

```
Small Maze
%%%%%%%%%%%%%%%%%%%%%%
% %%        % %      %
%    %%%%%% % %%%%%% %
%%%%%%.....P  %      %
%    %.%%%%%% %% %%%%%
% %%%%.%.....    %   %
%     ...%%%.%%%   % %
%%%%%%%%%%... %%%%%% %
%..........%%        %
%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 39
Path Cost = 29
```

```
Medium Maze
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%.................................P%
%.%%%%%%%%%%%%%%%%%%%%%%% %%%%%%%% %
%.%%...%...%      %%%%%%%   %%     %
%.%%.%.%.%.% %%%% %%%%%%%%% %% %%%%%
%.%%.%.%.%.%...          %% %%     %
%.%%.%.%.%.%.%.%%%%  %%%    %%%%%% %
%.% .%.%.%...%....%% %%%%%%%%      %
%.%%.%.%.%%%%%%%%.%%        %% %%%%%
%.%%.%...%%.......%%%%%%%%% %%     %
%....%%%%%%.%%%%%%%      %% %%%%%% %
%%%%%%......%.......%%%% %% %      %
%......%%%%%%.%%%%%.%    %% %% %%%%%
%.%%%%%%......%.....  %%%%% %%     %
%........%%%%%%.%%%%%%%%%%% %%  %% %
%%%%%%%%%%......            %%%%%% %
%..........%%%%%%%%%%%%%%%%        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 156
Path Cost = 152
```

Big Maze
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       % % %  .......  %   %     % %
% %%%%%%% % %%%.% %%%.%%% %%%%%%% % %
%       %  .....% %  .  %     % %   %
%%%%% %%%%%.%%% % % %.%%% %%%%% % %%%
%   % % % %.  % % % %.  % %   % %   %
% %%% % % %.%%% %%%%%.%%% % %%% %%% %
%       %  ...%   %  .%     % % %...%
%%% %%%%%%%%%.%%%%%%%.%%% %%% % %.%.%
%  ...........%.......% %   %.....%.%
% %.%%%%% % %%%.% % %%% % %%%.%%% %.%
% %.%     % % %.% %     %   %.% % %.%
% %.% %%%%%%% %.%%%%%%%%% %%%.% %%%.%
% %.% %     %  .%     %     %.  %  .%
%%%.%%% % %%%%%.%%%%% %%% %%%.%%%%%.%
%  .  % % %  ...% %     % %...% % %.%
% %.% % % %%%.%%% %%% %%% %.% % % %.%
% %.% % %    .............%.% %.....%
%%%.%%%%%%% % % %%%%% %%%.%.%%%.%%%%%
%  ...  % % % %     %   %...  %.%   %
%%%%%.% % %%%%%%%%% %%%%%%%%%%%.% %%%
%   %.%           % %     %...%.%   %
% %%%.%%%%% %%%%%%%%% %%%%%.%.%.%%% %
% %...%      % .......%.....%...    %
% %.% %%%%% %%%.% % %.%.%%%%%%%%%%%%%
% %.%   %     %.% % %...    %   % % %
% %.%%% %%% % %.% %%%%%%%%% %%% % % %
% %...% %   % %.%   % %   % % %     %
% %%%.%%% %%%%%.%%% % % %%%%% % %%%%%
%  ...  %   %  ...% %     %   % %   %
%%%.% %%%%% %%%%%.%%% %%% % %%% % %%%
% %.% % % % % %...  % %   % %...% % %
% %.%%% % % % %.%%%%%%%%% % %.%.% % %
%...%   %   %  ...............%.....%
%.% % % %%% %%% %%%%%%% %%% %%% %%%.%
%.% % %       %   %       %   % %  P%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 452
Path Cost = 210
```

A* Search

Our A* Search is very similar to our Greedy Best-First Search, with one exception – while the comparator still takes into account Manhattan distance to the end `Point`, it also takes into account distance already traveled to get to the `Point` in the `PriorityQueue`. It adds the two values, and that value is what is used to choose which `Point` in the `Queue` is removed next. Despite the differences, it still uses two similar methods: "*findSolution*" and "*getSolution*." The latter is exactly the same as *getSolution* used in Depth First Search, while the former slightly differs.

findSolution operates by taking in a `Maze` object and adding the starting `Point` to a `PriorityQueue` and adding it to a Vector called "visited," which keeps track of the `Point`s that have already been traversed.

findSolution then calls a while loop, which performs the actual searching algorithm. The loop lasts until the `PriorityQueue` is empty. A variable "`currentPoint`" of type `Point` is created that keeps track of the `Point` that is currently being inspected by being assigned to the `Point` that is removed from the `PriorityQueue`. (`currentPoint` is initially the starting `Point` since the starting `Point` was the only `Point` on the `Queue` before the while loop started). Every time a `Point` is removed from the `PriorityQueue`, an integer that keeps track of the number of `Point`s that have been traversed is incremented. A nested for loop that goes through the current `Point`'s adjacent `Point`s is then called. If the `Point` that the loop is going through is empty (not a wall or a dot) or is a dot, and if the `Point` has not previously been visited, the following things will happen:

- The `Point` will be added to the `PriorityQueue`
- The `Point` will be added to the Vector of visited `Point`s
- The `Point` is added to a `HashMap` called "predecessor" which takes the `Point` that the for loop was going through as the value as the key, and `currentPoint` (the `Point` whose *getAdjacentPoints* was called).
    - The reasoning behind this is that *predecessor(*`Point`*)* will return the `Point` that was visited before the currently visited `Point`.
    - predecessor is used again in *getSolution*
- If the `Point` is the end `Point`, the `Point` will be returned.

The nature of the `PriorityQueue` means that the next `Point` removed from the `Queue` will be the `Point` that has the smallest sum of Manhattan distance to the end `Point` and the number of steps already taken to get to that `Point`, making it perfect for A* search. By the end of *findSolution*, the end will have been found (and will be what `currentPoint` is currently pointing to), and the path from the start to the end will be stored in predecessor.

getSolution is then called, and the `Maze` solution is ready to be printed.

The result of A* Search on the three provided Mazes is:

```
Small Maze
%%%%%%%%%%%%%%%%%%%%%%
% %%        % %      %
%    %%%%%% % %%%%%% %
%%%%%%     P..%      %
%    % %%%%%%.%% %%%%%
% %%%% %    ..   %   %
%        %%%.%%%   % %
%%%%%%%%%%... %%%%%% %
%..........%%        %
%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 52
Path Cost = 19


Medium Maze
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                        .........P%
% %%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%% %
% %%   %   %      %%%%%%%...%%     %
% %% % % % % %%%% %%%%%%%%%.%% %%%%%
% %% % % % %        .....%%.%%     %
% %% % % % % % %%%% .%%%....%%%%%% %
% %  % % %   %    %%.%%%%%%%%      %
% %% % % %%%%%%%% %%........%% %%%%%
% %% %   %%       %%%%%%%%%.%%     %
%    %%%%%% %%%%%%%      %%.%%%%%% %
%%%%%%      %       %%%% %%.%      %
%      %%%%%% %%%%% %    %%.%% %%%%%
% %%%%%%      %       %%%%%.%%     %
%        %%%%%% %%%%%%%%%%%.%%  %% %
%%%%%%%%%%..................%%%%%% %
%..........%%%%%%%%%%%%%%%%        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 221
Path Cost = 68
```

Big Maze

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

Nodes Expanded = 549
Path Cost = 210

1.2 Designing "Difficult" Inputs for Different Search Algorithms

Our Maze that A* Search runs through well but Greedy Best-First Search has some difficulty with is

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

The results are:

```
Greedy Best-First Search
%%%%%%%%%%%%%%%%%%%%%%
%....................%
%.%%%%%%%%%%%%%%%%%%.%
%.%                %.%
%.%              % %.%
%.%              % %.%
%.%              % %.%
%.%%%%%%%%%%%%%%%% %.%
%P                 %.%
%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 112
Solution Cost = 33

A* Search
%%%%%%%%%%%%%%%%%%%%%%
%....................%
%.%%%%%%%%%%%%%%%%%%.%
%.%                %.%
%.%              % %.%
%.%              % %.%
%.%              % %.%
%.%%%%%%%%%%%%%%%% %.%
%P                 %.%
%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 60
Solution Cost = 33
```

The reasoning that Greedy Best-First Search struggles with this Maze is that it first goes right and then goes through all of the empty space in the middle, since those Points are closer to the end Point than the Points that are on the right path. A* search does the same thing at first, but then the sum of the distance from the active Point to the end Point and the number of steps already taken to get to the Point ends up exceeding that of the Point above the starting Point, so it proceeds to continue on that path and get to the end Point while expanding less nodes than Greedy Best-First Search. Greedy Best-First search expanded almost twice as many nodes.

Our `Maze` that Greedy Best-First Search runs through well but A* Search has some difficulty with is

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

Greedy Best-First Search
%%%%%%%%%%%%%%%%%%%%%%
%                 ...%
%                %.%.%
%                %.%.%
%                %.%.%
%                %.%.%
%                %.%.%
%            %%%%%.%.%
%P.................%.%
%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 34
Path Cost = 33

A* Search
%%%%%%%%%%%%%%%%%%%%%%
%                 ...%
%                %.%.%
%                %.%.%
%                %.%.%
%                %.%.%
%                %.%.%
%            %%%%%.%.%
%P.................%.%
%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 142
Path Cost = 33
```

The reasoning that Greedy Best-First Search does better with this `Maze` is that it first goes right and then continues up to each free `Point` closest to the end `Point`, which in this case is the best solution. A* search does the same thing at first, but then the sum of the distance from the active `Point` to the end `Point` and the number of steps already taken to get to the `Point` ends up exceeding that of other `Point`s, so it proceeds to traverse those instead of continuing on the right path for some time. A* search had to expand four times as many nodes.

1.3 Search With Multiple Dots

For our complete traversal to collect all our dots, we used an A* approach along with modified heuristics from our single target search. The heuristics we used were benchmarking the number of dots left with the closest dot by Manhattan distance. An important part of this task was that we had to make sure our heuristics were admissible. The number of dots left is an admissible heuristic because it takes at least that many steps if not more to collect all those dots, and the closest dot by Manhattan distance is correct because it takes at least the Manhattan distance to traverse to that dot.

We represented the search with a `State` rather than a `Point`. A State object contained the current Point we are on in the maze, a list of the remaining dots in the maze, the path cost so far to reach that point, and the heuristic to compare. Instead of having a visited list like we did in our single target A* search, we compared each new state with our previous states, not allowing any duplicates. The main advantage to this is that while we can be at the same point in the maze, the states might be different because there would be less dots left and a higher path cost. This allows us to backtrack, a crucial part of a complete traversal, but not be stuck in an infinite loop.

Our algorithm for searching is very similar to the single target search:

- Add the state to the `PriorityQueue`
- Remove the first element in the `PriorityQueue`
- Expand the node and inspect all adjacent States
- If there are no more dots in the maze, terminate
- If the frontier doesn't contain the adjacent state, add it to the PQ

The nature of our algorithm as well as the results show us that our solution was optimal. Our heuristic takes into account both the current path we travelled, and the distance to the nearest node. It is admissible on both counts and optimal in that it finds a good path to take. We combined the two heuristics and subtracted one, the movement that we traverse in that one step. The sum is still admissible.

The result of A* complete traversal on the two provided Mazes is:

```
----------------------
Tests for Section 1.3
----------------------
%%%%%%%%%%%%%%%%%%%%
%.           ...P .%
%.%%.%%.%%.%%.%% %.%
% %% %.....      %.%
%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 34896
Solution Distance = 34

%%%%%%%%%%%%%%%%%%%%
%.           ..%   %
%.%%.%%.%%.%%.%% % %
%        P       % %
%%%%%%%%%%%%%%%%%% %
%.....             %
%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 126156
Solution Distance = 60

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%............%%%%%............%
%%%.%...%%%.........%.%...%.%%%
%...%%%.%.%%%%.%.%%%%%%.%%%...%
%.%.....%......%......%.....%.%
%.%%%.%%%%%.%%%%%%%.%%%.%.%%%%%
%.....%........P....%...%.....%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 614404
Solution Distance = 294

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%.....%.................%.....%
%.%%%.%.%%%.%%%%%%%.%%%.%.....%
%.%...%.%......%......%.%.....%
%...%%%.%.%%%%.%.%%%%...%%%...%
%%%.%.%.%.%......%..%.%...%.%%%
%...%.%%%.%.%%% %%%.%.%%%.%...%
%.%%%.......%     %.......%%%.%
%...%.%%%%%.%%%%%%%.%.%%%.%...%
%%%.%...%.%....%....%.%...%.%%%
%...%%%.%.%%%%.%.%%%%.%.%%%...%
%.......%......%......%.....%.%
%.....%.%%%.%%%%%%%.%%%.%.%%%.%
%.....%........P....%...%.....%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Nodes Expanded = 399677
Solution Distance = 448
```

For the larger mazes, an optimal solution can be found faster with a weighted A* search, which would heavily reduce the nodes expanded and run much quicker. However, our optimal solution can work on all sized mazes with a significant amount of time.