Maze Search

1. 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 points 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 Points 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 Points 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 Points 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, then the Point below it, and then above.

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 toStrings of the Points in the array of Points, 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 Points 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 Points that have been traversed is incremented. A nested for loop that goes through the current Point’s adjacent Points is then called, and 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 Points
* 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 Stacks 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 smallMaze.txt is:

Depth First Search

%%%%%%%%%%%%%%%%%%%%%%

% %% % % %

% %%%%%% % %%%%%% %

%%%%%%.....P % %

% %.%%%%%% %% %%%%%

% %%%%.%..... % %

% ...%%%.%%% % %

%%%%%%%%%%... %%%%%% %

%..........%% %

%%%%%%%%%%%%%%%%%%%%%%

## 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 Points 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 Points that have been traversed is incremented. A nested for loop that goes through the current Point’s adjacent Points is then called, and 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 Points
* 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 Queues 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:

Breadth First Search

%%%%%%%%%%%%%%%%%%%%%%

% %% % % %

% %%%%%% % %%%%%% %

%%%%%% P..% %

% % %%%%%%.%% %%%%%

% %%%% % .. % %

% %%%.%%% % %

%%%%%%%%%%... %%%%%% %

%..........%% %

%%%%%%%%%%%%%%%%%%%%%%

## Greedy Best-First Search

Our Greedy Best-First 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 Points 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 Points that have been traversed is incremented. A nested for loop that goes through the current Point’s adjacent Points is then called, and 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 Points
* 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:

## A\*Star Search

To replace the placeholder text on this page, just select a line of text and start typing. But don’t do that just yet!

First check out a few tips to help you quickly format your report. You might be amazed at how easy it is.

* Need a heading? On the Home tab, in the Styles gallery, just tap the heading style you want.
* Notice other styles in that gallery as well, such as for a numbered list, or a bulleted list like this one.

## Look Great Every Time

For best results when selecting text to copy or edit, don’t include space to the right of the characters in your selection.

1. Here is a numbered list which you can find in the Styles gallery.

### Heading 3

Replace this text with your text.

Breadth First Search

%%%%%%%%%%%%%%%%%%%%%%

% %% % % %

% %%%%%% % %%%%%% %

%%%%%% P..% %

% % %%%%%%.%% %%%%%

% %%%% % .. % %

% %%%.%%% % %

%%%%%%%%%%... %%%%%% %

%..........%% %

%%%%%%%%%%%%%%%%%%%%%%

Depth First Search

%%%%%%%%%%%%%%%%%%%%%%

% %% % % %

% %%%%%% % %%%%%% %

%%%%%%.....P % %

% %.%%%%%% %% %%%%%

% %%%%.%..... % %

% ...%%%.%%% % %

%%%%%%%%%%... %%%%%% %

%..........%% %

%%%%%%%%%%%%%%%%%%%%%%

Greedy Best First Search

%%%%%%%%%%%%%%%%%%%%%%

% %% % % %

% %%%%%% % %%%%%% %

%%%%%%.....P % %

% %.%%%%%% %% %%%%%

% %%%%.%..... % %

% ...%%%.%%% % %

%%%%%%%%%%... %%%%%% %

%..........%% %

%%%%%%%%%%%%%%%%%%%%%%

A\* Search

%%%%%%%%%%%%%%%%%%%%%%

% %% % % %

% %%%%%% % %%%%%% %

%%%%%% P..% %

% % %%%%%%.%% %%%%%

% %%%% % .. % %

% %%%.%%% % %

%%%%%%%%%%... %%%%%% %

%..........%% %

%%%%%%%%%%%%%%%%%%%%%%

BFS Nodes Expanded = 90

DFS Nodes Expanded = 53

Greedy Best-First Nodes Expanded = 39

A\* Nodes Expanded = 52

BFS Path Cost = 19

DFS Path Cost = 29

Greedy Best-First Path Cost = 29

A\* Path Cost = 19

----------------------

Tests for Section 1.2

----------------------

Special Maze: Should be BAD for Greedy Search

%%%%%%%%%%%%%%%%%%%%%%

%....................%

%.%%%%%%%%%%%%%%%%%%.%

%.% %.%

%.% % %.%

%.% % %.%

%.% % %.%

%.%%%%%%%%%%%%%%%% %.%

%P %.%

%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 112

Path Cost = 33

Special Maze: Should be GOOD for Astar Search

%%%%%%%%%%%%%%%%%%%%%%

%....................%

%.%%%%%%%%%%%%%%%%%%.%

%.% %.%

%.% % %.%

%.% % %.%

%.% % %.%

%.%%%%%%%%%%%%%%%% %.%

%P %.%

%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 60

Path Cost = 33

Special Maze: Should be GOOD for Greedy Search

%%%%%%%%%%%%%%%%%%%%%%

% ...%

% %.%.%

% %.%.%

% %.%.%

% %.%.%

% %.%.%

% %%%%%.%.%

%P.................%.%

%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 34

Path Cost = 33

Special Maze: Should be BAD for Astar Search

%%%%%%%%%%%%%%%%%%%%%%

% ...%

% %.%.%

% %.%.%

% %.%.%

% %.%.%

% %.%.%

% %%%%%.%.%

%P.................%.%

%%%%%%%%%%%%%%%%%%%%%%

Nodes Expanded = 142

Path Cost = 33