# 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 row by row in 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. Each **Location** in the **Maze** is then added to a two-dimensional array of Locations based on where it is located in the text file. As the maze is being built, the start and goal positions are additionally stored in separate **Location** variables.

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. Before a **Location** is added to the ArrayList, the *getAdjacent* also checks to see if it is valid (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*

*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 Locations 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 start position)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. 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, its coordinates will be stored in a two dimensional array of **Location**s called “predecessors”. 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 ArrayList to a dot. The loop terminates once the start **Location** is reached and there are no predecessors left.

### 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, out BFS still uses two methods: *findSolution* and *printSolution.* The *printSolution* method is identical to that of the DFS.

*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 start position)is assigned to the **Location** removed from the queue, and it is the current location being inspected. Each time a **Location** is removed from the queue, an integer is incremented to keep track of how many **Location**s have been traversed. 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, its coordinates will be stored in a two dimensional array of **Location**s called “predecessors”. The **Location** will then be added to 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 ArrayList to a dot. The loop terminates once the start **Location** is reachedand there are no predecessors left.

### Greedy Best-First Search

### A\* (Astar) Search

## Penalizing Turns (1.2)

## Pacman with a GHOST (1.3)