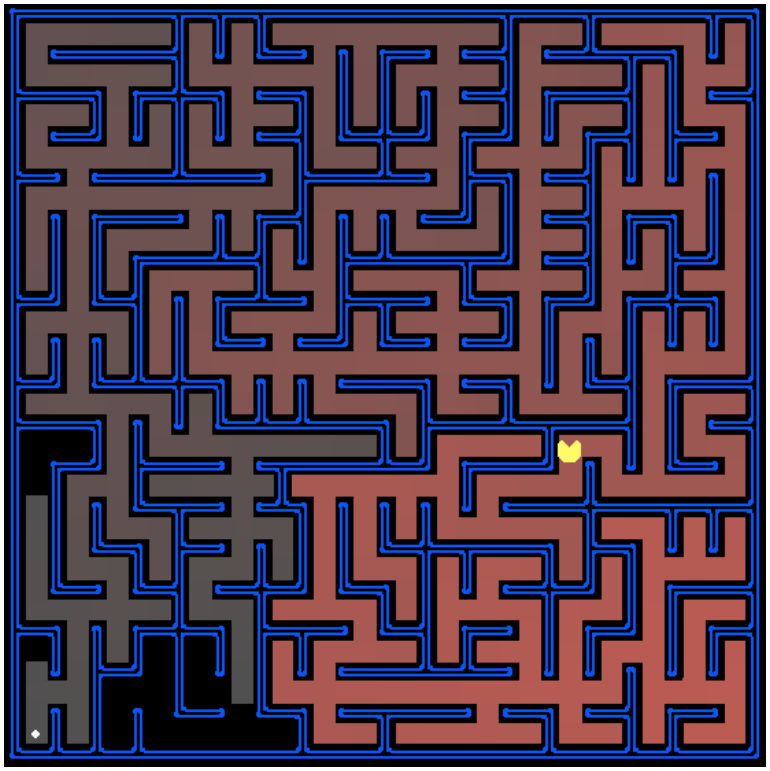
CMSC 373 Artificial Intelligence

Lab/Assignment#2
Uninformed Search in Pac-Man[i]
Due Wednesday, September 22



There are two exercises in this lab:

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)

Introduction

As in the first assignment, you will continue to work with your partner.

In this assignment, your Pac-Man agent will find paths through his maze world to reach a particular location. You will build general search algorithms and apply them to many different Pac-Man scenarios.

Files you'll edit:

`search.py`	Where all your search algorithms will reside.
`searchAgents.py`	Where all your search-based agents will reside.

Files you should look at but NOT edit:

`util.py`	Useful data structures for implementing search algorithms.
`pacman.py`	The main file that runs Pac-Man games. This file describes a Pac-Man `GameState` type, which you use in this lab.
`game.py`	The logic behind how the Pac-Man world works. This file describes several supporting types like `AgentState`, `Agent`, `Direction`, and `Grid`.

Finding a fixed food dot using Uninformed Search

In `searchAgents.py`, you'll find a fully implemented `SearchAgent`, which plans out a path through Pac-Man's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented -- that's your job. As you work through the following questions, you might need to refer to the glossary of objects in the code (at the end of this handout).

First, test that the `SearchAgent` is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command above tells the `SearchAgent` to use `tinyMazeSearch` as its search algorithm, which is implemented in `search.py`. This simply follows a fixed sequence of actions to demonstrate how the code works. Pac-Man should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pac-Man plan routes! Pseudocode for the depth-first search and breadth-first search algorithms you'll write is shown below.

```
function UninformedSearch(problem) returns a list of actions
   initialize the frontier using the initial state of the problem
   initialize explored to empty
   # For explored, use Pacman position as the key with a value True
   # initialize a dictionary of states already explored
   while frontier is not empty do
      choose a leaf node and remove it from the frontier
      if the node contains a goal state
         return list of actions from start state to goal state
      add the state key to the explored dictionary
      for each successor of the node state
         if the key of the successor state is not in explored
            add node of the successor onto the frontier
   return an empty list (i.e. no solution!)
```

Important note*:* All your search functions need to return a list of *actions* that will lead the agent from the start to the goal. These actions must be legal moves (valid directions, no moving through walls).

Hint: Algorithms for DFS and BFS differ only in the details of how the frontier is managed. So, concentrate on getting DFS right and then BFS should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. Your implementation need *not* be of this form to receive full credit.

Hint: Make sure to check out the `Stack`, `Queue`, and `PriorityQueue` types provided to you in `util.py`.

EXERCISE 1: Implement the depth-first search algorithm in the `depthFirstSearch` function in `search.py`. You should begin by creating a `Node` class to use in all your search algorithms. Recall that a search node contains:

- current state
- parent node
- action taken to get to the state
- step cost
- total path cost

Although DFS and BFS ignore the costs, you'll need them for later search methods. Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

The Pac-Man board will show an overlay of color for the states explored and the order in which

they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pac-Man actually go to all the explored squares on his way to the goal?

Hint: The solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the frontier in the order provided by `getSuccessors()`; you might get 244 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

EXERCISE 2: Implement the breadth-first search algorithm in the `breadthFirstSearch` function in `search.py`. Use the same algorithm as shown in the above pseudocode. Test your code the same way you did for depth-first search.

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=bfs
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Does BFS find a least cost solution?

Hint: If Pac-Man moves too slowly for you, try the option `--frameTime 0`.

Object Glossary

Here's a glossary of the key objects in the code base related to search problems, for your reference:

`SearchProblem (search.py)`
: A `SearchProblem` is an abstract object that represents the state space, successor function, costs, and goal state of a problem. You will interact with any `SearchProblem` only through the methods defined at the top of `search.py`

`PositionSearchProblem (searchAgents.py)`
: A specific type of `SearchProblem` that you will be working with --- it corresponds to searching for a single pellet in a maze.

Search Function
: A search function is a function which takes an instance of `SearchProblem` as a parameter, runs some algorithm, and returns a sequence of actions that lead to a goal. Example of search functions are `depthFirstSearch` and `breadthFirstSearch`, which you have to write. You are provided `tinyMazeSearch` which is a very bad search function that only works correctly on `tinyMaze`

`SearchAgent`
: `SearchAgent` is a class which implements an Agent (an object that interacts with the world) and does its planning through a search function. The `SearchAgent` first uses

the search function provided to make a plan of actions to take to reach the goal state, and then executes the actions one at a time.

What to Hand in

1. Fill out the table below:

	Depth-First Search			Breadth-First Search		
	#nodes explored	**Solution length**	**Is it optimal?**	**#nodes explored**	**Solution length**	**Is it optimal?**
tinyMaze						
mediumMaze						
bigMaze						

2. Based on the above, a short discussion/reflection of how the searches compare.

3. A printout of all the code you wrote. Note, ONLY the code you wrote, not the contents of the entire file in which you wrote it. This should include your code for the search node, DFS, and BFS.
4. Staple, clearly indicating your names on the front page of your report.

[i] The Pac-Man code was developed by John DeNero and Dan Klein at UC Berkeley.