**Lab 2**

The purpose of this lab is to get you to implement search algorithms.

**Introduction**

On Blackboard you will find the complete source files of Pacman (courtesy of UC Berkley). Setup a source code directory, copy the folder in there and start examining the code. It will take a little while to understand how it all hangs together, but dig in and ask questions.



The plan is to write an automated AI Agent which will control Pacman. Pacman lives in a maze and must find the food dots to survive.

So, the aim of our AI Agent is to search the maze to find the food dot and then navigate Pacman there.

When everything is installed, at the cmd prompt change into the Pacman directory and you should be able to play a game of Pacman by running:

python pacman.py

To understand how an Agent works, try the following pre-written Agent called GoWestAgent:

python pacman.py --layout testMaze --pacman GoWestAgent

However, this agent is pretty unintelligent since it only moves West and can’t turn. Try it again in a different Maze:

python pacman.py --layout tinyMaze --pacman GoWestAgent

Note that pacman.py supports a number of options that can each be expressed in a long way (e.g., --layout) or a short way (e.g., -l). You can see the list of all options and their default values via:

python pacman.py -h

**File that you will edit:**

search.py Where all of your search algorithms will reside.

**Files you might want to look at:**

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

pacman.py The main file that runs Pacman games.

This file describes a Pacman GameState type, which you use in this

project.

game.py The logic behind how the Pacman world works. This file describes

several supporting types like AgentState, Agent, Direction, and Grid.

util.py Useful data structures for implementing search algorithms.

**Supporting files you can ignore:**

graphicsDisplay.py Graphics for Pacman

graphicsUtils.py Support for Pacman graphics

textDisplay.py ASCII graphics for Pacman

ghostAgents.py Agents to control ghosts

keyboardAgents.py Keyboard interfaces to control Pacman

layout.py Code for reading layout files and storing their contents

test\_cases/ Directory containing the test cases for each question

**Exercise1**:

In searchAgents.py, you'll find a fully implemented SearchAgent, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented -- that's your job.

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. Pacman should navigate the maze successfully.

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

I have included an implementation of the depth-first search (DFS) algorithm in search.py called depthFirstSearch. Try it with this:

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

Examine it and then **implement the breadth-first search (BFS) algorithm** in the breadthFirstSearch function in search.py.

**Hint**: The algorithms we discussed in class are very similar. Algorithms for DFS, BFS, and A\* differ only in the details of how the list of nodes is managed (stack v queue v priority queue).

When you are done, test your algorithm on the larger mazes:

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

There are some cool parameters you can pass into Pacman when you run it. A couple of useful ones include:

Zoom: on some screens the bigMaze does not fit. You can zoom out e.g. to zoom out 50% and make the whole game window smaller use:

-z 0.5

Speed: to speed up Pacman you can specify the time between each frame being drawn on the screen. The default time is 0.1 seconds e.g. to double the speed we can do the following:

--frameTime 0.05 (note the double -- in this case)

**Exercise2**:

Implement the A\* search algorithm in the aStarSearch function in search.py

A\* takes a heuristic function as an argument. Heuristic functions take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). By default, the nullHeuristic heuristic function in search.py is used but us a trivial example that simply returns a value of 0.

Hint: The order that you process the nodes should be in order of minimum cost. You can make use a PriorityQueue (in util.py) to do this where the cost is used as the priority. A priority queue pops the items in order of priority (low number = high priority).

Try it with:

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

python pacman.py -l mediumMaze -p SearchAgent -a fn=astar

python pacman.py -l bigMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

**Exercise3**:

You can test your A\* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as manhattanHeuristic in searchAgents.py).

python pacman.py -l tinyMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

python pacman.py -l mediumMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

python pacman.py -l bigMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

**Exercise 4:**

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: FoodSearchProblem in searchAgents.py (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. If you have written your general search methods correctly, A\* with a null heuristic should quickly find an optimal solution to testSearch with no code change on your part (total cost of 7).

python pacman.py -l testSearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic

The foodHeuristic function in searchAgents.py currently does nothing and just returns 0, so it works as is.

Try it on a more challenging board:

python pacman.py -l trickySearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic

For comparison, try the DFS and BFS searches also

python pacman.py -l trickySearch -p SearchAgent -a fn=dfs,prob=FoodSearchProblem

python pacman.py -l trickySearch -p SearchAgent -a fn=bfs,prob=FoodSearchProblem

How do they compare?

Now, fill in the foodHeuristic function in searchAgents.py with a consistent heuristic for the FoodSearchProblem. Try your agent on the trickySearch board:

python pacman.py -l trickySearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic