COEN 266 Artificial Intelligence

Homework #2 – Part 2

The code for this homework consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download and unzip all the code and supporting files search_and_games.zip.

Files to Edit: You will edit portions of search.py, where all of your search algorithms will reside. There is no need to change the other files/code in the source code folder.

Submission:

1. Submit a pdf file to Camino (for the format of the pdf file, please refer to Homework2_Part2_submission_sample.pdf).
2. Submit all source code needed (with search.py modified by you) to generate all results of the **Experiments** in Problem 1 and Problem 2 as a .zip file to Camino. We will test run your submitted code, so make sure it works.

Grading: The grade depends on both the submitted pdf and source code.

Files you might want to look at

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.
searchAgents.py	Where all of your search-based agents will reside.
util.py	Useful data structures for implementing search algorithms.

Files you will not edit

agentTestClasses.py	Autograding test classes
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
autograder.py	Project autograder
testParser.py	Parses autograder test and solution files
testClasses.py	General autograding test classes
test_cases/	Directory containing the test cases for each question

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know, and we will pursue the strongest consequences available to us.

This assignment is based on the Pacman AI projects developed at UC Berkeley, http://ai.berkeley.edu.

Welcome to Pacman

After downloading the code (search_and_games.zip), unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

The simplest agent in `searchAgents.py` is called the `GoWestAgent`, which always goes West (a trivial reflex agent). This agent can occasionally win:

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

But, things get ugly for this agent when turning is required:

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

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

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
```

Also, all of the commands that appear in this portion of the project also appear in `commands.txt`, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with `bash commands.txt`.

Problem 1: Iterative Deepening

In the iterativeDeepeningSearch function in search.py, implement an iterative-deepening search algorithm (iterative deepening depth-limited depth-first **graph search**). You will probably want to make use of the Node class in search.py. Figure 3.17 and Figure 3.18 from the textbook (also given at the end of this document) may help.

Experiments: Test your code using:

```
python pacman.py -l threeByOneMaze -p SearchAgent -a fn=ids

python pacman.py -l testMaze -p SearchAgent -a fn=ids

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

python pacman.py -l smallMaze -p SearchAgent -a fn=ids

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

python pacman.py -l bigMaze -p SearchAgent -a fn=ids
```

A few additional notes:

- If Pacman moves too slowly for you, try the option --frameTime 0.
- All of your 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).
- We are implementing **graph search**, not tree search, so IDS might not return the optimal path.

Problem 2: A* Search

Implement A* graph search in the empty function aStarSearch in search.py. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The nullHeuristic heuristic function in search.py is a trivial example.

You will probably want to make use of the Node class in search.py and the PriorityQueue class in util.py.

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).

Experiments:

```
python pacman.py -l tinyMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

python pacman.py -l smallMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

python pacman.py -l mediumMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic --frameTi
me 0

python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic --frameTime 0
```

Note: The following (from the textbook) may be helpful for you to implement the Iterative Deepening depth-first search algorithm.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.