COMP7404 - Assignment 1 (Part A)

19/20 Semester 1

This assignment is based on materials by http://ai.berkeley.edu.

Introduction

In this part of the assignment, you will build informed, uninformed and local search algorithms and apply them to Pacman and the 8-Queens problem.

Like in Assignment 0, this project includes an autograder for you to mark your answers. This can be run with the command

```
python autograder.py
```

See the autograder tutorial in Assignment 0 for more information about using the autograder.

The code for this project 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 all the code and supporting files from a1A.zip.

Related Files

Files you'll edit and submit

File	Description
`search.py`	Where all of your search algorithms will reside
`searchAgents.py`	Where all of your search-based agents will reside
`solveEightQueens.py`	Where all you local search algorithms go

Files you might want to look at

File	Description
`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

File	Description
`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
`searchTestClasses.py`	Autograding test classes

Requirements

Files to Edit and Submit: You will fill in portions of `search.py`, `searchAgents.py` and `solveEightQueens.py` during the assignment. You should submit these files with your code and comments. Please do not change the other files in this distribution or submit any of our original files other than these files.

Evaluation: Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you may wreak havoc on the autograder.

Academic Dishonesty: We will be checking your code against other submissions in the class and from the Internet for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try.

We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, please submit questions to the forum.If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Discussion Forum: Please be careful not to post spoilers. Please don't post any code that is directly related to the assignments. However you are welcome and encouraged to discuss general ideas.

Submission: Submit your code `search.py`, `searchAgents.py`, `solveEightQueens.py` as `a1A_UID.zip` file to moodle (UID is your student ID).

You will get zero mark if

- you submit the wrong files
- you copy another student's answer
- your zip file's name does not follow the format `a1A_UID.zip`
- Your zip uses any other file format than ZIP
- your program contains an infinite loop

Check your files before the submission.

Welcome to Pacman

Playing Pacman with the keyboard

After downloading the code, 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.

A predefined simple agent

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
```

Pacman options

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`. Below I list some of the most useful options in this assignment.

```
Usage:
    USAGE:      python pacman.py <options>
    EXAMPLES:   (1) python pacman.py
                    - starts an interactive game
                (2) python pacman.py --layout smallClassic --zoom 2
                    - starts an interactive game on a smaller board, zoomed in

Options:
  -h, --help            show this help message and exit
  -l LAYOUT_FILE, --layout=LAYOUT_FILE
                        the LAYOUT_FILE from which to load the map layout
                        [Default: mediumClassic]
  -p TYPE, --pacman=TYPE
                        the agent TYPE in the pacmanAgents module to use
                        [Default: KeyboardAgent]
  -z ZOOM, --zoom=ZOOM  Zoom the size of the graphics window [Default: 1.0]
  -a AGENTARGS, --agentArgs=AGENTARGS
                        Comma separated values sent to agent. e.g.
                        "opt1=val1,opt2,opt3=val3"
  --frameTime=FRAMETIME
                        Time to delay between frames; <0 means keyboard
                        [Default: 0.1]
```

Different kinds of layout

In the `layouts/` directory, you can find multiple predefined layouts. You can set the `--layout` option to one of the following layout (please do not include `.lay`).

```
bigCorners.lay contestClassic.lay mediumMaze.lay openClassic.lay
smallSafeSearch.lay tinyMaze.lay bigMaze.lay contoursMaze.lay
mediumSafeSearch.lay openMaze.lay smallSearch.lay tinySafeSearch.lay
bigSafeSearch.lay greedySearch.lay mediumScaryMaze.lay openSearch.lay
testClassic.lay tinySearch.lay  bigSearch.lay mediumClassic.lay
mediumSearch.lay originalClassic.lay testMaze.lay trappedClassic.lay
boxSearch.lay mediumCorners.lay minimaxClassic.lay smallClassic.lay
testSearch.lay trickyClassic.lay capsuleClassic.lay mediumDottedMaze.lay
oddSearch.lay smallMaze.lay tinyCorners.lay trickySearch.lay
```

Exit the Game

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

Soon, your agent will solve not only `tinyMaze`, but any maze you want.

Also, all of the commands that appear in this assignment also appear in `commands.txt`, for easy copying and pasting.

Finding a Fixed Food Dot using Search Algorithms

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. As you work through the following questions, you might find it useful to refer to the object glossary. Here's a glossary of the key objects in the code 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.
- `CornersProblem` (`searchAgents.py`)
  A specific type of `SearchProblem` that you will define --- it corresponds to searching for a path through all four corners of a maze.
- `FoodSearchProblem` (`searchAgents.py`)
  A specific type of SearchProblem that you will be working with --- it corresponds to searching for a way to eat all the pellets in a maze.

- Search function (`search.py`)
  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` (`searchAgents.py`)
  `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.

Next, 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`. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note 1: 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).

Important note 2: Make sure to use the `Stack`, `Queue` and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the frontier is managed. So, concentrate on getting DFS right and the rest 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).

Question 1 (2 points): Depth First Search

Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. To make your algorithm complete, write the graph search version of DFS, which

avoids expanding any already visited states.

(cost, node expended)
T (10, 15)
M (130, 146)
B (210, 390)

Your code should quickly find a solution for

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=dfs
python pacman.py -l mediumMaze -p SearchAgent # -a fn=dfs can be omitted, default option
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

The Pacman board will show an overlay of 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 Pacman actually go to all the explored squares on his way to the goal?

Hint: If you use a Stack as your data structure, 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 246 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.

Question 2 (2 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

(cost, node expended)
M (68, 269)
B (210, 620)

```
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? If not, check your implementation.

Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

```
python eightpuzzle.py
```

Question 3 (2 points): Varying the Cost Function (Uniform Cost Search)

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best"

in other senses. Consider `mediumDottedMaze` and `mediumScaryMaze`.

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`. We encourage you to look through `util.py` for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

(cost, node expended)
M (68, 269)
M-D (186, 646)
M-S (68719479864, 108)

Note: You should get very low and very high path costs for the `StayEastSearchAgent` and `StayWestSearchAgent` respectively, due to their exponential cost functions (see `searchAgents.py` for details).

Question 4 (2 points): 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 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 bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on `openMaze` for the various search strategies?

(cost, node expended)
B (210, 549)

Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

In corner mazes, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first! Hint: the shortest path through `tinyCorners` takes 28 steps.

Question 5 (2 points): Representation for Corners Problem

Note: Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.

Implement the `CornersProblem` search problem in `searchAgents.py`. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

```
python pacman.py -l tinyCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```

To receive full credit, you need to define an abstract state representation that does not encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman GameState as a search state. Your code will be very, very slow if you do (and also wrong).

Hint: The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Our implementation of `breadthFirstSearch` expands just under 2000 search nodes on `mediumCorners`. However, heuristics (used with A* search) can reduce the amount of searching required.

Question 6 (2 points): Heuristics for Corners Problem

Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.

Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic` in `searchAgent.py`.

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Note: AStarCornersAgent is a shortcut for

```
-p SearchAgent -a
fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be admissible, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be consistent, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most c.

Remember that admissibility isn't enough to guarantee correctness in graph search -- you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to receive any

points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded

Number of nodes expanded	Marks
more than 2000	0/3
at most 2000	1/3
at most 1600	2/3
at most 1200	3/3

Remember: If your heuristic is inconsistent, you will receive no credit, so be careful!

Eating All The Dots

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. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, A* with a null heuristic (equivalent to uniform-cost search) 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 AStarFoodSearchAgent
```

Note: `AStarFoodSearchAgent` is a shortcut for

```
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

You should find that UCS starts to slow down a bit even for the seemingly simple `tinySearch`. As a reference, our implementation takes 0.7 seconds to find a path of length 27 after expanding 5057 search nodes.

Question 7 (4 points): Food Heuristic

Note: Make sure to complete Question 4 before working on Question 7, because Question 7 builds upon your answer for Question 4.

Fill in `foodHeuristic` in `searchAgents.py` with a consistent heuristic for the `FoodSearchProblem`. Try your agent on the `trickySearch` board.

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Our UCS agent finds the optimal solution in about 3 seconds, exploring over 16,000 nodes.

Any non-trivial non-negative consistent heuristic will receive 1 point. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll get additional points.

Number of nodes expanded	Grade
more than 15000	1/4
at most 15000	2/4
at most 12000	3/4
at most 9000	4/4 (full credit)
at most 7000	5/4 (optional extra credit)

Remember: If your heuristic is inconsistent, you will receive no credit, so be careful!

Question 8 (6 points): Local Search

Reference: Artificial Intelligence: A Modern Approach (3rd Edition) P120-124

The 8-Queens problem places eight chess queens on an 8×8 chessboard such that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

In this question you will implement a simple local search method. Read the code of the class `SolveEightQueens` in the file `solveEightQueens.py`. The solve method of this class will call the helper function `search` to solve the 8-Queens problem by iteratively changing the location of a selected queen on the board. Type

```
python solveEightQueens.py -l
```

to generate the 8-Queens problem shown in the lecture. The following output should appear

```
iteration 0
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . q . . . .
q . . . q . . .
. q . . . q . q
. . q . . . q .
. . . . . . . .

*** Method not implemented: getNumberOfAttacks at line 115 of
solveEightQueens.py
```

Implement `getBetterBoard`

Next, implement the method `getBetterBoard` in `solveEightQueens.py`. A correct implementation will return a tuple consisting of a better 8-Queen configuration, its corresponding number of attacking queens and the selected row and column. The method should return the best 8-Queen configuration obtained by moving a single queen along its column.

Important: Please move a single queen along its column at each iteration to pass the autograder.

There are a number of correct outputs for the example above. Here is one (just the first 3 iterations are shown):

```
python solveEightQueens.py -l

iteration 0
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . q . . . .
q . . . q . . .
. q . . . q . q
. . q . . . q .
. . . . . . . .

# attacks: 17
 18 12 14 13 13 12 14 14
```

```
 14 16 13 15 12 14 12 16
 14 12 18 13 15 12 14 14
 15 14 14  q 13 16 13 16
  q 14 17 15  q 14 16 16
 17  q 16 18 15  q 15  q
 18 14  q 15 15 14  q 16
 14 14 13 17 12 14 12 18

iteration 1
. . . . . . . .
. . . . q . . .
. . . . . . . .
. . . q . . . .
q . . . . . . .
. q . . . q . q
. . q . . . q .
. . . . . . . .

# attacks: 12
 13  7 10 10 13  9 10  9
 11 11 10 12  q 11  9 11
 10  7 13 10 15  9  9  9
 11  9 11  q 13 11 10 11
  q  9 12 10 17  9 11 11
 14  q 12 13 15  q 11  q
 14  9  q 11 15 10  q 11
 10  8  9 13 12 10  8 12

iteration 2
. q . . . . . .
. . . . q . . .
. . . . . . . .
. . . q . . . .
q . . . . . . .
. . . . . q . q
. . q . . . q .
. . . . . . . .

# attacks: 7
 10  q  7  7  9  6  5  6
  8 11  7  8  q  6  4  7
  6  7  9  7  9  5  4  5
```

```
  7  9  7  q  9  7  5  7
  q  9  7  6 12  6  6  7
  9 12  7  8  9  q  6  q
  9  9  q  7 10  6  q  8
  6  8  5  8  7  6  3  8

iteration  3
...
```

Once you are confident that both your implementations of `getBetterBoard` and `getNumberOfAttacks` are correct, you may attempt solving a larger number of 8-Queens problems that are randomly generated.

```
python solveEightQueens.py -n 100 -q
Solved: 11 / 100
```

However, the current algorithm stops if it reaches a plateau where the best successor has the same value as the current state. Starting from a randomly generated 8-queen state, hill-climbing search can only solve 11% of the problem instances. One common solution is to allow a certain number of consecutive sideways moves. For example, we could allow up to 100 consecutive moves in the 8-queens problem. Hopefully, this will raise the success rate of local search.

Modify the stop criterion

Modify the stop criterion in the current algorithm to allow up to 100 consecutive moves even though the best successor has the same value as the current state. The algorithm should stop immediately if there is no attack between the queens.

After modification, you can run the above command again to check whether the success rate will rise.

Important note: The autograder for this question will randomly generate 30 problem instances, and your program should at least solve 25 of them to get this credit.

Acknowledgments

This work is based on previous work by John DeNero and Dan Klein et al. of berkeley.edu