Computer Science 384
St. George Campus

Friday, January 19, 2018
University of Toronto

Homework Assignment #1: Search
— Part I —
Due: Monday, February 5, 2018 by 11:59 PM

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due, i.e., no question about this assignment will be answered, whether it is asked on the discussion board, via email or in person.*

Late Policy: 10% per day after the use of 3 grace days.

Overview: Assignment #1 is comprised of two parts. Part I is a programming assignment. Part II is a small set of short answer questions that accompany Part I. This document details Part I, the programming assignment.

Total Marks: Part I has total of 50 marks.
Assignment #1 (Part I and Part II) represent 10% of the course grade.

Handing in this Assignment

What to hand in on paper: Nothing.

What to hand in electronically: Download the assignment files from http://www.teach.cs.toronto.edu/~csc384h/winter/Assignments/A1/A1.html. Modify `search.py`, `searchAgents.py` and `acknowledgment_form.pdf` as described in this document and submit your modified versions using MarkUs. Your login to MarkUs is your teach.cs username and password. You can submit a new version of any file at any time, though the lateness penalty applies if you submit after the deadline. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission. More detailed instructions for using MarkUs are available at: http://www.teach.cs.toronto.edu/~csc384h/winter/markus.html.

We will test your code electronically. You will be supplied with a testing script that will run a **subset** of the tests. If your code fails all of the tests performed by the script (using Python version 2.7.13), you will receive a failing grade on the assignment.

When your code is submitted, we will run a more extensive set of tests which will include the tests run in the provided testing script and a number of other tests. You have to pass all of these more elaborate tests to obtain full marks on the assignment.

Your code will not be evaluated for partial correctness; it either works or it doesn't. It is your responsibility to hand in something that passes at least some of the tests in the provided testing script.

- *Make certain that your code runs on teach.cs using Python (version 2.7.13) using only standard imports.* This version is installed as "python" on teach.cs. Your code will be tested using this version and you will receive zero marks if it does not run using this version.
- *Do not add any non-standard imports from within the python file you submit (the imports that are already in the template files must remain).* Once again, non-standard imports will cause your code to fail the testing and you will receive zero marks.
- *Do not change the supplied starter code.* Your code will be tested using the original starter code, and if it relies on changes you made to the starter code, you will receive zero marks. See Section 1 for a description of the starter code.

For the purposes of this assignment, we consider the standard imports to be what is included with Python 2.7.13 plus NumPy 1.13.1 and SciPy 0.19.1 (which are installed on the teach.cs machines). If there is another package you would like to us, please ask and we will consider adding it to the list.

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 1 Clarification page:

`http://www.teach.cs.toronto.edu/~csc384h/winter/Assignments/A1/a1_faq.html`.

*** You are responsible for monitoring the A1 Clarification page.***

Help Sessions: There will be approximately three help sessions for this assignment. Dates and times for these sessions will be posted to the course website and to Piazza.

Questions: Questions about the assignment should be posed on Piazza:

`https://piazza.com/utoronto.ca/winter2018/csc384`.

If you have a question of a personal nature, please email the A1 TA, Andrew Perrault, at perrault at cs dot toronto dot edu or the instructor at csc384prof at cs dot toronto dot edu placing 384 and A1 in the subject line of your message.

Warnings: We are aware that solutions to the original Berkeley project exist on the internet. Do not use these solutions as this would be plagiarism. To earn marks on this assignment you must develop your own solutions. Also please consider the following points.

- You are to implement the search algorithms following the pseudocode in algorithms.pdf. These algorithms differ in subtle but important ways from other presentations of this material. If you implement your search based on other non-course material it might give the wrong answers. If you try to use solutions found on the internet the same problem might occur.
- We will check for answers that would arise from solutions to the original Berkeley project. Such answers indicate that your solution is incorrect, reproducing the errors of the Berkeley lectures. This will cause you to fail some of our additional tests and you will lose marks for those tests.
- If we find evidence of plagiarism we will investigate thoroughly and we will send your case to the University Academic Offenses Office.
- Please do not implement your own "improvement" to the search algorithms: it will wreak havoc with the automarker. (Note, if you have invented a significant improvement we would be happy to hear about it, but don't use it in this assignment).
- You are asked to write, sign, scan, and submit a statement acknowledging that the code you submitted was written by you (`acknowledgment_form.pdf`).
- Although the assignment includes an autograder, additional tests will be run on your code after submission.
- Be careful to not post spoilers on Piazza.

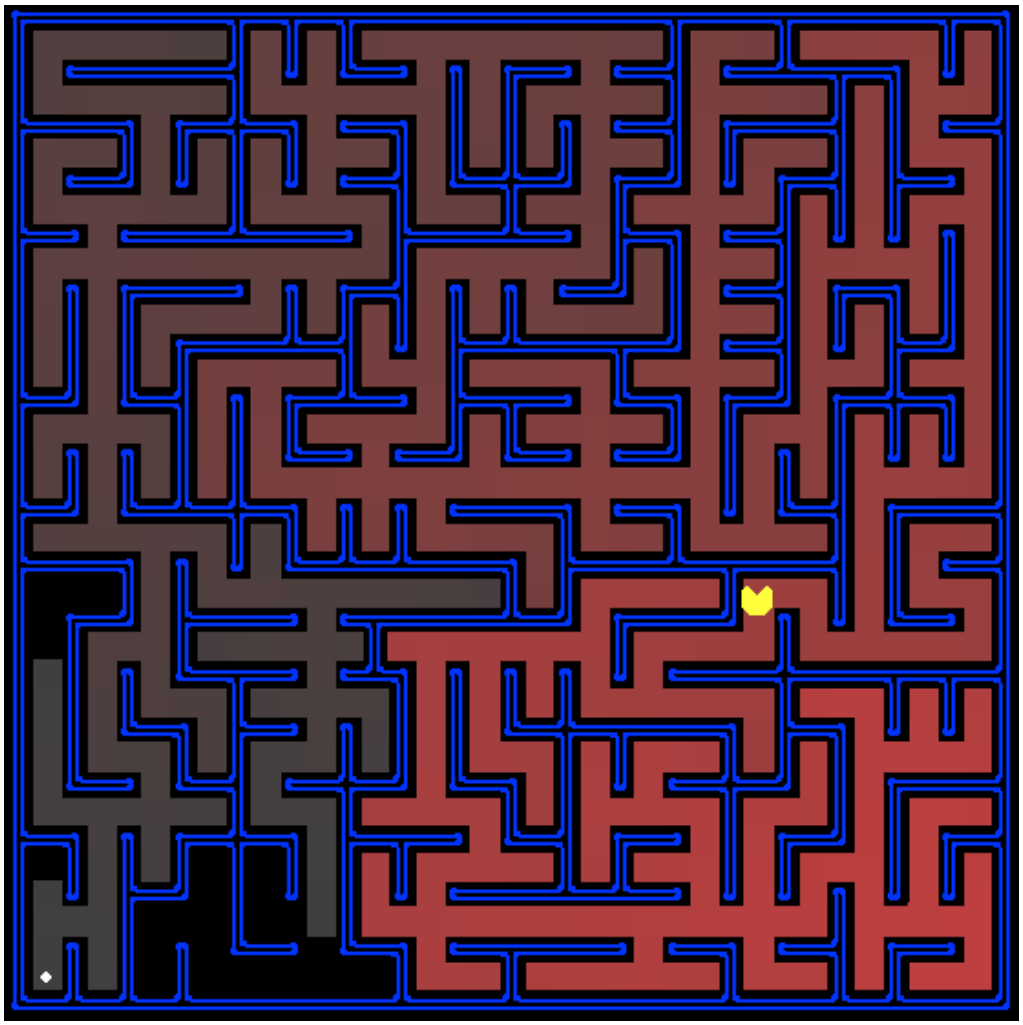

Figure 1: All those colored walls,
Mazes give Pacman the blues,
So teach him to search.

1 Introduction

In this assignment, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

This assignment includes an autograder for you to grade your answers on your machine. This can be run with the command: `python autograder.py`. Note: **other tests will be run on your submitted code beyond the tests the autograder runs.** See the autograder tutorial in Assignment 0 for more information about using the autograder.

The code for this assignment 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 as a zip archive. In that zip you will find the following files.

Files you'll edit:

- search.py: where all of your search algorithms will reside.
- searchAgents.py: where all of your search-based agents will reside.

Files you should review:

- pacman.py: the main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this assignment.
- 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.
- algorithms.pdf This file contains pseudocode for algorithms covered in class. You should follow this pseudocode in the development of your 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.
- autograder.py: assignment autograder.
- testParser.py: parses autograder test and solution files.
- testClasses.py: general autograding test classes.
- test_cases/: directory containing some test cases for each question.
- searchTestClasses.py: Assignment 1 specific autograding test classes.

2 Welcome to Pacman

After downloading the code (search.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
```

Note: **if python2.7 is not the default python on the machine you are using you will have to change python to the command that invokes python2.7**. You can also configure the Wing IDE (if you use this) to invoke `python2.7`.

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.

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

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 assignment 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`. (Note if python2.7 is not the default on your machine you might have to change commands in this file).

Note: if you have a non-graphics connection you can run the `pacman.py` code with the `-t` argument. If you install python2.7 on your personal machine it should come with the correct graphics support, and the graphics should also work if you are connected to the teaching labs with an X Window connection.

Note: In the various parts below we ask a number of questions. You **do not** have to hand in answers to these questions, rather they are designed to help you understand what is going on with search.

Question 1 (6 points): Finding a Fixed Food Dot using Depth First Search

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.

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`. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. 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: 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: 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 OPEN 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).

Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. To ensure that DFS does not run around in circles, implement **path checking** to prune cyclic paths during search. **Do not** use full cycle checking for DFS (we will use full cycle checking for the other searches)

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 Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Check that the exploration order is 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 fringe 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 (6 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in search.py. This time you must implement full cycle checking in your search algorithm to avoid the overhead of cyclic paths. Test your code the same way you did for depth-first search.

```
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.

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

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 (6 points): Varying the Cost Function

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 search algorithm with full cycle checking 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
```

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 (6 points): A* search

Implement A* search with full cycle checking 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 by expanding slightly fewer nodes 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?

You will find that DFS will take too long on `openMaze` maze since it only does path-checking. If you use full cycle checking with DFS it will find a solution (a very unusual one!).

Question 5 (6 points): 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.

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 (6 points): Corners Problem: Heuristic

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, admissible heuristic for the `CornersProblem` in `cornersHeuristic`.

```
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`.

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 admissible 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	Grade
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 inadmissible, you will receive no credit, so be careful!

Question 7 (8 points): 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 assignment, 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 assignment.) 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 even for the seemingly simple `tinySearch`. As a reference, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search nodes.

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 an admissible 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 13 seconds, exploring over 16,000 nodes.

Any non-trivial non-negative admissible 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; medium)
at most 7000	5/4 (optional extra credit; hard)

Remember: If your heuristic is inadmissible, you will receive no credit, so be careful! Can you solve `mediumSearch` in a short time? If so, we're either very, very impressed, or your heuristic is inadmissible.

Question 8 (6 points): Suboptimal Search

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard (even the layout `mediumSearch` is probably too hard for A*). In these cases, we'd still like to find a reasonably good path, quickly. In this section, you'll write an agent that always greedily eats the closest dot. `ClosestDotSearchAgent` is implemented for you in `searchAgents.py`, but it's missing a key function that finds a path to the closest dot.

Implement the function `findPathToClosestDot` in `searchAgents.py`. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Hint: The quickest way to complete `findPathToClosestDot` is to fill in the `AnyFoodSearchProblem`, which is missing its goal test (its goal should get to any food location). Then, solve that problem with an appropriate search function. The solution should be very short!

Your `ClosestDotSearchAgent` won't always find the shortest possible path through the maze. Make sure you understand why and try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots.

GOOD LUCK and HAVE FUN!