CS 4260/5260: Introduction to AI
Homework 1 [points]: (Search).
Due: 2018, Central Time on BrightSpace

General Instructions:
If anything is ambiguous or unclear.
1. Discuss possible interpretations with other students, your TA, and instructor
2. Send e-mail to your TA first, and to your instructor if an issue is not resolved to your satisfaction.
3. Make use of web sources.

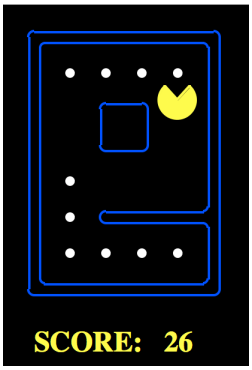
Remember that after general discussions with others, you are required to work out the problems by yourself. All submitted work must be your own. Please refer back to the Honor code for clarifications.
Write legibly, be sure to staple all your answer sheets together, and write your name, and the honor pledge on the top of the first answer sheet.
Start early, and avoid last minute stress!

Introduction

In this project, you will code a Pacman agent that finds paths through a 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.

The code for this project is stored in a few Racket files inside the racket-solver folder. The file you will be editing is called **student.rkt** Example mazes are stored in text files with .lay extensions inside the layouts folder.

The python code in the main folder provides visualizations of your final search path like the one shown below. If you don't have python on your machine, and don't want to see these visualizations, you can ignore these files (they are not needed to run your code and will not be used in grading).



How to Run Your Code

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 as a zip archive.

Your code should return a list of characters which are instructions for pacman (i.e. `'(#\S #\S #\W)` for south (down), south, west (left)).

To run your code in racket run:

```
racket racket-solver/main.rkt -l <path to layout in layouts dir>
-s <search strategy> -r <heuristic>
```

If you are using Windows change this to:

```
racket racket-solver\main.rkt -l <path to layout in layouts dir>
-s <search strategy> -r <heuristic>
```

For example:

```
racket racket-solver/main.rkt -l layouts/tinyMaze.lay -s
tinyMazeSearch
```

or for Windows:

```
racket racket-solver\main.rkt -l layouts\tinyMaze.lay -s
tinyMazeSearch
```

See more examples in racket-commands.txt

To visualize your code with python (if you have python installed), run:

```
python pacman.py -l <name of layout file> -p RacketAgent -a
fn=<search strategy>,heuristic=<heuristic>
```

Note: It is important that there be no space between the comma and heuristic.

See examples in commands.txt

Problems

There are three search strategies that you need to implement for this assignment: DFS, BFS and A*.

Depth First Search

For python visualization:

In searchAgents.py, you'll find a fully implemented RacketAgent, which visualizes 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 RacketAgent is working correctly by running:

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

The command above tells the RacketAgent to use tinyMazeSearch as its search algorithm, which is implemented in student.rkt at the top of the file. Pacman should navigate the maze successfully.

Racket Part:
Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode and examples for the search algorithms you'll write can be found in the lecture slides from week 2 and in section 3.5.2 of the textbook. 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: The functions provided in utils.rkt assume that nodes on the frontier are stored as tuples of type (<path to state>, state). If you choose to store your path differently, please include the functions needed to interpret your solution in student.rkt.

Important note: All of your search functions need to return a list of letters representing actions that will lead the agent from the start to the goal (for example `'(#\S #\S #\W)` for south-south-west). These actions all have to be legal moves (valid directions, no moving through walls). If you use the get-succ function provided in utils.py, each successor comes paired with the letter that describes its action.

Your function must return this list, not print it. If you are printing the list instead of returning it, your code will throw an error when you run the racket command to test your program.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward.

Implement the depth-first search (DFS) algorithm in the DFS function in student.rkt. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states (if you do not keep track of already visited states, your code will be very slow and may exceed the stack size allowed for racket).

To test your code, try the following commands in your command line (substitute \ for / if using Windows):

```
racket racket-solver/main.rkt -l layouts/tinyMaze.lay -s dfs
racket racket-solver/main.rkt -l layouts/tinySearch.lay -s dfs
racket racket-solver/main.rkt -l layouts/smallMaze.lay -s dfs
racket racket-solver/main.rkt -l layouts/greedySearch.lay -s dfs
racket racket-solver/main.rkt -l layouts/mediumMaze.lay -s dfs
racket racket-solver/main.rkt -l layouts/bigMaze.lay -s dfs
```

Each test should print a list of instructions in .5 – 3 seconds.

If you would like to see your output visualized using the python libraries, try the following commands:

```
python pacman.py -l tinyMaze -p RacketAgent
python pacman.py -l tinySearch -p RacketAgent
python pacman.py -l smallMaze -p RacketAgent
python pacman.py -l greedySearch -p RacketAgent
python pacman.py -l mediumMaze -p RacketAgent
python pacman.py -l bigMaze -z .5 -p RacketAgent
```

Hint: If you use a Stack as your data structure and add nodes to the frontier in the same order they are provided by get-succ, the solution found by your DFS algorithm for mediumMaze should have a length of 234 (if you push them in the reverse order you might get 130). Is this a least cost solution? If not, think about what depth-first search is doing to get a sub-optimal path.

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

Breadth First Search

Implement the breadth-first search (BFS) algorithm in the BFS function in student.rkt. 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.

(again substitute \ for / if using Windows):

```
racket racket-solver/main.rkt -l layouts/tinyMaze.lay -s bfs
racket racket-solver/main.rkt -l layouts/smallMaze.lay -s bfs
racket racket-solver/main.rkt -l layouts/greedySearch.lay -s bfs
racket racket-solver/main.rkt -l layouts/mediumMaze.lay -s bfs
racket racket-solver/main.rkt -l layouts/bigMaze.lay -s bfs
```

These commands should each take .5 - 2 seconds to run.

If you want to, you can also try tinySearch, but your code will probably take 10 seconds or more to find the solution using BFS:

```
racket racket-solver/main.rkt -l layouts/tinySearch.lay -s bfs
```

If you would like to see visuals of your code in action, try the following python commands:

```
python pacman.py -l smallMaze -p RacketAgent -a fn=bfs
python pacman.py -l mediumMaze -p RacketAgent -a fn=bfs
python pacman.py -l greedySearch -p RacketAgent -a fn=bfs
python pacman.py -l bigMaze -p RacketAgent -a fn=bfs -z .5
```

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

A-Star Search

Implement A* graph search in the empty function A-star in student.rkt. A* takes a heuristic function as an argument. The null-heuristic function in utils.rkt is a trivial example.

You can test that your A* implementation is working correctly by running the following racket commands (again substitute \ for / if using Windows):

```
racket racket-solver/main.rkt -l layouts/bigMaze.lay -s astar -r distance-to-food
racket racket-solver/main.rkt -l layouts/openMaze.lay -s astar -r distance-to-food
racket racket-solver/main.rkt -l layouts/greedySearch.lay -s astar -r count-food
racket racket-solver/main.rkt -l layouts/tinySearch.lay -s astar -r count-food
```

These commands should each take .5 - 2 seconds to run.

You should get the same results as from your BFS implementation, but in less time (note that tinySearch should now only take a few seconds.

If you would like to see visuals of your code in action, try the following python commands:

```
python pacman.py -l bigMaze -z .5 -p RacketAgent -a fn=astar,heuristic=distance-to-food
python pacman.py -l openMaze -z .5 -p RacketAgent -a fn=astar,heuristic=distance-to-food
python pacman.py -l greedySearch -p RacketAgent -a fn=astar,heuristic=count-food
python pacman.py -l tinySearch -p RacketAgent -a fn=astar,heuristic=count-food
```

Some Tips for Racket

There are several ways in which Racket works differently from object oriented languages, as you no doubt discovered in Programming Assignment 0. This section contains a few pointers on how to do some things in Racket that might feel a little non-intuitive.

You can find more notes on how to use functions in the utils.rkt file and examples in the slides from Thursday, September 13.

Assigning and Scoping Variables

Use **`define`** to make global variables, **`let`** to make locally scoped variables and **`set!`** to change the value of a variable. To assign multiple variables use **`define-values`**, **`let-values`** and **`set!-values`**.

While loops

To make a while loop in racket, add a #:break condition to an infinite for loop. For example:

```
(let ([x 0])
     (for ([i (in-naturals 1)]
          #:break (>= x 5))
               (print (list i x))
               (set! x (add1 x))))
```

You can also add "continue" type statements by filtering loops with the #:unless operator:

```
(let ([x 0])
     (for ([i (in-naturals 1)]
          #:unless (even? i)
          #:break (>= x 5))
               (print (list i x))
               (set! x (add1 x))))
```

Using Lists, Stacks, Queues and Priority Queues

Methods of stacks, queues and priority queues have been provided at the top of **utils.rkt**.

You will notice that push functions return the datatype and pop functions return the popped item and the data type. This is because in functional languages like Racket it is considered good practice not to mutate objects and instead re-assign them to the mutated version using functions like **`set!`** and **`set!-values`**.

You can check if an element is in a list using the **`ismember?`** function (this will come in handy when you want to check if a state was previously visited).

Acknowledgements

The python code and idea for this project comes from UC Berkley. You can read about their python version of this project at http://ai.berkeley.edu