Project 1 Report Specification

Due by 11:59pm 9/26/2018

### Third Puzzle: The Burnt Pancake Problem

This is a variation on the more straight-forward [pancake flipping problem](https://en.wikipedia.org/wiki/Pancake_sorting). In the original pancake flipping problem, you are given a stack of pancakes of varying diameters. You want to sort the pancakes so that they are in order from the largest (at the bottom) to the smallest (at the top). You are only allowed to flip the pancakes by sticking a spatula into the stack and flipping all pancakes above it. For example, if I have a stack of pancakes in the order of (top to bottom):

3 1 2 5 4

and if I stick my spatula between pancake size 2 and pancake size 5 and flip, the result will be:

2 1 3 5 4

In the burnt pancake variant, we distinguish between the top and bottom of the pancakes (which are apparently all burnt). We still want to sort the pancakes in order of their sizes, but we now also want all the burnt side to face down. If a pancake has its burnt side up, we’ll represent it with a negative sign. For example, the pancake stack:

3 -1 2 5 4

means that the pancake of size 1 (the second from the top position) has its burnt side up. And if we stick our spatula between 2 and 5 and flip, the result will be:

-2 1 -3 5 4

Now, the pancakes of diameter 2 and 3 have their burnt sides facing up.

We will use a configuration file (two test cases are provided for you in the Google Directory linked below) to specify the particular instance of the puzzle. It has the following format:

Line 1 is a keyword (**pancakes**) that tells you this is the pancakes problem

Line 2 is the initial state; e.g., (3, -1, 2, 5, 4)

Line 3 is the goal state; e.g., (1, 2, 3, 4, 5) // we don’t really need this line

You may assume that we will always give n pancakes, all with unique sizes ranging from 1 to n.

For this problem, you have the freedom to implement any heuristic function that isn’t h(node)=0. Your function does not necessarily have to be admissible and consistent, though you should discuss all these issues in your report (see below). You may either try to think up your own function or read up on some research papers about this problem (give citations) and implement something based on your readings.

### Experimentation

* Download the test configuration files from the shared [Google Directory](https://drive.google.com/drive/folders/16ezjCBaObnzET7UOhackAwuE-P1gOyGI?usp=sharing)
  + test\_jug.config
  + test\_cities.config
  + test\_pancakes1.config
  + test\_pancakes2.config
* Run your puzzle solver for these test scenarios:
  + water jugs (show runs on **bfs**, **dfs, greedy**) on test\_jug.config
    - for **greedy**, you need to supply one heuristic of your own design (see more in discussions)
    - Please try to adopt the following convention for enumerating your action list; this would be a great help for the grader:
      * fill jug actions first (ordered from largest to smallest)
      * empty jug actions next (from largest to smallest)
      * transfer between jugs actions (order sources from large to small)

Just to be clear, it’s a good idea to **explicitly enumerate** the order of actions explored for the grader.

* + path-finding (show runs on **unicost**, **greedy, astar**) on test\_cities.config
    - for **greedy** and **astar** use Euclidean distance as your heuristic function.
    - Please adopt the following convention for enumerating your action list:
      * for a city at location (x,y), consider neighboring cities starting from (x+delta,y) in clockwise fashion.

Just to be clear, it’s a good idea to explicitly describe how you order the actions, perhaps with an example so that there is no confusion.

* + pancakes (show runs on **iddfs,** **astar**, **idastar**) on both test\_pancakes1.config and test\_pancakes2.config
    - Please adopt the following convention for enumerating your action list:
      * Consider inserting the spatula from closest to the bottom to closest to the top.
* For each of the test cases listed above, capture your program’s outputs for the specified search strategies and store them in a “typescript” file. If the search strategy takes longer than 30 minutes on any of the test cases, you may interrupt it and report that the search strategy was not able to finish within the allotted time.

### Report Content

1. First, briefly describe your project setup: which version of Python? Basic computer configurations
2. If you discussed the project with someone, list their names and briefly describe what you talked about. Relatedly, if you consulted any sources outside of the class notes and the textbook, please give citations to them here.
3. Report any known bugs you have.
4. For each puzzle type:
   1. Enumerate your action expansion order
   2. Give the name of the heuristic function (so that the grader can supply it as an argument in the command line). If you made up any heuristic function(s) in addition to the ones we specified, explain what it is and why you designed it that way.
   3. show a transcript of the expected run.
5. Additional discussion points: Here, you may want to run some test cases on more search strategies than those required for the transcript. Address the following issues:
   1. For each of the three puzzle types, what do you think is the best search strategy, and why? You should take all four factors into considerations (completeness, optimality, time complexity, space complexity).
   2. For the water jugs problem and the burnt pancakes problem, is your heuristic admissible and consistent? Explain.
   3. You may also include additional discussions on any observations that you find surprising and/or interesting.