B551 Assignment 1: Searching

Spring 2015
Due: Friday September 18, 11:59PM
(You may submit up to 48 hours late for a 10% penalty.)

This assignment will give you practice with posing AI problems as search, and with un-informed and informed search algorithms. This is also an opportunity to dust off your coding skills. This assignment consists of two parts. Part One requires you to solve problems with pen and paper. Part Two requires you to write some programs. You may work with a partner (in a group of two) for this assignment. You should only submit **one** copy of the assignment for your team, through GitHub, as described below. Both people on the team will receive the same grade on the assignment, except in very unusual circumstances. Please read the instructions below carefully; we cannot accept any submissions that do follow the instructions given here. Most importantly: please **start early,** and ask questions on the OnCourse Forum or in office hours.

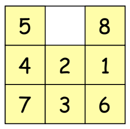
Academic integrity. You and your partner may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about Python syntax and features, etc. You may also consult printed and/or online references, including books, tutorials, etc., but you must cite these materials (e.g. in source code comments). However, the work and code that you and your partner must be your group's own work, which the two of you personally designed and wrote. You may not share written answers or code with any other students except your own partner, nor may you possess code written by another student who is not your partner, either in whole or in part, regardless of format.

Part 1: Written problems

Solve the following problems. You must show your work (derivations, calculations, etc.). Submit your solutions electronically, as PDF files. To do this, either type up your answers and all your work in a plain text or PDF file, or neatly write your solutions by hand on paper and scan them into a PDF file.

1. Consider a particularly unusual variant of tic-tac-toe, wherein the first player to complete a row, column, or diagonal *loses* the game. Draw the state graph of this game for a 2×2 board. Make sure to indicate the start state as well as the goal states.

2. Consider the heuristic function for the 8-puzzle given by: $h(s)$ = sum of permutation inversions. For example, $h(N) = 4+6+3+1+0+2+0+0 = 16$ (there are 4 numbers smaller than 5 that come after 5, 6 numbers smaller than 6 that come after 6, and so on) for the following board configuration N:



STATE(N)

   Is h admissible? Prove your answer.

Part 2: Programming problems

The following problems require you to write programs in Python. You may import standard Python modules for routines not related to AI, such as basic sorting algorithms and basic data structures like queues. You must write all of the rest of the code yourself. If you have any questions about this policy, please ask us. We recommend using the CS Linux machines (e.g. `burrow.soic.indiana.edu`). You may use another development platform (e.g. Windows), but make sure that your code works on the CS Linux machines before submission. If you've never used Python before, now you have an excuse to learn! There are many tutorials and references available online, as well as interactive sites like `http://www.codeacademy.com/`.

For each problem, please include a detailed comments section at the top of your code that describes: (1) a description of how you formulated the search problem, including precisely defining the state space, the successor function, the edge weights, and (for problems 2 and 3) the heuristic function(s) you designed for A* search, including an argument for why they are admissible; (2) a brief description of how your search algorithm works; (3) and discussion of any problems you faced, any assumptions, simplifications, and/or design decisions you made.

You'll submit your code via GitHub. We strongly recommend using GitHub as you work on the assignment, pushing your code to the cloud whenever you reach a milestone. If you have not used IU GitHub before, instructions for getting started with git are available on the OnCourse wiki and in many online tutorials.

0. For this project, we are assigning you to a team with another student, unless you choose to work alone. We will let you change these teams in future assignments. You can find your assigned teammate(s) by logging into IU Github, at `http://github.iu.edu/`. In the upper left hand corner of the screen, you should a pull-down menu. Select `cs-b551`. Then in the yellow box to the right, you should see a repository called *userid1-userid2*-p1, where the other user ID corresponds to your teammate.

   To get started, clone the github repository:

   `git clone https://github.iu.edu/cs-b551/`*your-repo-name*`-a1`

   where *your-repo-name* is the one you found on the GitHub website above. (If this command doesn't work, you probably need to set up IU GitHub ssh keys. See OnCourse wiki for help.)

1. An online retailer wants to create a delivery system in which packages are flown from the warehouse directly to each customer's front door. Their first idea (autonomous drones) was unreliable, so the company switched from robotic technology to a fleet of highly-trained Amazon Parrots. But while the birds are able to deliver packages safely, they can't read the address labels, which means they regularly deliver items to the wrong address. For instance, on the first day of testing, five packages were supposed to be delivered to five separate customers, each of whom lived on a separate street. But none of the customers got the correct order! The customer who ordered the Candelabrum received the Banister, while the customer who ordered the Banister received the package that Irene had ordered. Frank received a Doorknob. George's package went to Kirkwood Street. The delivery that should have gone to Kirkwood Street was sent to Lake Avenue. Heather received the package that was to go to Orange Drive. Jerry received Heather's order. The Elephant arrived in North Avenue; the person who had ordered it received the package that should have gone to Maxwell Street. The customer on Maxwell Street received the Amplifier. Who ordered each package, and where did each person live? Write a Python program that figures out and displays the answer.

2. It's September, which means you have only 6 months to make your Spring Break vacation plans to a dream destination! We've prepared a dataset of major highway segments of the United States (and parts of southern Canada and northern Mexico), including highway names, distances, and speed limits; you can visualize this as a graph with nodes as towns and highway segments as edges. We've also prepared a dataset of cities and towns with corresponding latitude-longitude positions. These files should be in your GitHub repo from when you cloned in step 0. Your job is to implement algorithms that find good driving directions between pairs of cities given by the user. Your program should be run on the commandline like this:

   ```
   python route.py [start-city] [end-city] [routing-option] [routing-algorithm]
   ```

   where:

   - `start-city` and `end-city` are the cities we need a route between.
   - `routing-option` is one of:
     - `segments` finds a route with the fewest number of "turns" (i.e. edges of the graph)

  - `distance` finds a route with the shortest total distance
  - `time` finds the fastest route, for a car that always travels at the speed limit

- `routing-algorithm` is one of:
  - `bfs` uses breadth-first search
  - `dfs` uses depth-first search
  - `astar` uses A* search, with a suitable heuristic function

The output of your program should be a nicely-formatted, human-readable list of directions, including travel times, distances, intermediate cities, and highway names, similar to what Google Maps or another site might produce. In addition, the *last* line of output should have the following machine-readable output about the route your code found:

```
[total-distance-in-miles] [total-time-in-hours] [start-city] [city-1] [city-2] ... [end-city]
```

Please be careful to follow these interface requirements so that we can test your code properly. For instance, the last line of output might be:

```
51 1.0795 Bloomington,_Indiana Martinsville,_Indiana Jct_I-465_&_IN_37_S,_Indiana Indianapolis,_Indiana
```

Like any real-world dataset, our road network has mistakes and inconsistencies; in the example above, for example, the third city visited is a highway intersection instead of the name of a town. Some of these "towns" will not have latitude-longitude coordinates in the cities dataset; you should design your code to still work well in the face of these problems.

In the comment section at the top of your code file, please include a brief analysis of the results of your program. Which search algorithm seems to work best for which routing options? Which algorithm is fastest in terms of the amount of computation time required by your program, and by how much, according to your experiments? Which heuristic function did you use, how good is it, and how might you make it better? (To measure time accurately, you may want to temporarily include a loop in your program that runs the routing a few hundred or thousand times.)

3. Consider a variant of the 15-puzzle, but with the following important changes. First, instead of 15 tiles, there are 16, so that there are no empty spaces on the board. Second, instead of moving a single tile into an open space, a move in this puzzle consists of either (a) sliding an entire row of tiles left or right, with the left- or right-most tile "wrapping around" to the other side of the board, or (b) sliding an entire column of the puzzle up or down, with the top- or bottom-most tile "wrapping around." For example, here is a sequence of two moves on such a puzzle:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2	3	4
8	5	6	7
9	10	11	12
13	14	15	16

1	14	3	4
8	2	6	7
9	5	11	12
13	10	15	16

The goal of the puzzle is to find a short sequence of moves that restores the canonical configuration (on the left above) given an initial board configuration. Write a program called solver16.py that finds a solution to this problem efficiently using A* search. Your program should run on the command line like:

```
python solver16.py [input-board-filename]
```

where `input-board-filename` is a text file containing a board configuration in a format like:

```
5 7 8 1
10 2 4 3
6 9 11 12
15 13 14 16
```

The program can output whatever you'd like, except that the last line of output should be a machine-readable representation of the solution path you found, in this format:

```
[move-1] [move-2] ... [move-n]
```

where each move is encoded as a letter `L`, `R`, `U`, or `D` for left, right, up, or down, respectively, and a row or column number (indexed beginning at 1). For instance, the two moves in the picture above would be represented as:

```
R2 D2
```

What to turn in

Turn in the four files required above (PDF from Part 1, and three programs from Part 2) by simply put the finished version (of the code and the report) on GitHub (remember to `add`, `commit`, `push`) — we'll grade whatever version you've put there as of 11:59PM on the due date. To make sure that the latest version of your work has been accepted by GitHub, you can log into the github.iu.edu website and browse the code online.