CS B551 - Assignment 1: Searching

Fall 2019
Due: Tuesday October 8, 11:59:59PM Eastern (New York) time
Resubmissions or late submissions due: Thursday October 10, 11:59:59PM Eastern (New York) time
(See *Submissions* below for details.)

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. Please read the instructions below carefully; we cannot accept any submissions that do not follow the instructions given here. Most importantly: please **start early,** and ask questions on Piazza or in office hours.

Guidelines for this assignment

Coding requirements. For fairness and efficiency, we use a semi-automatic program to grade your submissions. This means you must write your code carefully so that our program can run your code and understand its output properly. In particular: 1. You must code this assignment in Python 3, not Python 2. 2. Make sure to include a #! line at the top of your code. 3. You should test your code on one of the SICE Linux systems such as burrow.sice.indiana.edu. 4. Your code must obey the input and output specifications given below. 5. You may import standard Python modules for routines not related to AI, such as basic sorting algorithms and data structures like queues, as long as they are already installed on the SICE Linux servers. 6. Make sure to use the program file name we specify.

These requirements were discussed in more detail in the Assignment 0 handout.

For each of these problems, you will face some design decisions along the way. Your primary goal is to write clear code that finds the correct solution in a reasonable amount of time. To do this, you should give careful thought to the search abstractions, data structures, algorithms, heuristic functions, etc. that you use. To encourage innovation, we will conduct a competition among programs to see which can solve the hardest problems in the shortest amount of time.

Submissions. To help you make sure your code satisfies our requirements, we will give you the chance to revise your code based on some initial feedback from our grading program. If you submit your code by the **Due date** listed above, we will attempt to run your code through a simplified version of our grading program and send you feedback within 24 hours. You may then submit a revised version before the **Resubmission** listed above, with no grade penalty. If you do not submit anything by the **Due date** date, you may still submit by the **Resubmission** deadline, but it will count as a late submission and the 10% grade deduction will apply. You'll submit your code via GitHub.

Groups. You'll work in a group of 1-3 people for this assignment; we've already assigned you to a group (see details below) according to your preferences. You should only submit **one** copy of the assignment for your team, through GitHub, as described below. All the people on the team will receive the same grade on the assignment, except in unusual circumstances; we will collect feedback about how well your team functioned in order to detect these circumstances. The requirements for the assignment are the same whether you have 1 person, 2 people, or 3 people on your team, but we expect that teams with more people will submit answers that are more "polished" — e.g., better documented code, faster running times, more thorough answers to questions, etc.

Coding style and documentation. We will not explicitly grade based on coding style, but it's important that you write your code in a way that we can easily understand it. Please use descriptive variable and function names, and use comments when needed to help us understand code that is not obvious.

Report. Please put a report describing your assignment in the Readme.md file in your Github repository. For each problem, please include: (1) a description of how you formulated the search problem, including precisely defining the state space, the successor function, the edge weights, the goal state, and (if applicable) the heuristic function(s) you designed, 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. These comments are especially important if your code does not work as well as you would like, since it is a chance to document how much energy and thought you put into your solution. For example, if you tried several different heuristic functions before finding one that worked, feel free to describe this in the comments so that we appreciate the work that you did that might not otherwise be reflected in the final solution.

Academic integrity. We take academic integrity very seriously. To maintain fairness to all students in the class and integrity of our grading system, we will prosecute any academic integrity violations that we discover. *Before beginning this assignment, make sure you are familiar with the Academic Integrity policy of the course, as stated in the Syllabus, and ask us about any doubts or questions you may have.* To briefly summarize, you 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). We expect that you'll write your own code and not copy anything from anyone else, including online resources. *However, if you do copy something (e.g., a small bit of code that you think is particularly clever), you have to make it explicitly clear which parts were copied and which parts were your own. You can do this by putting a very detailed comment in your code, marking the line above which the copying began, and the line below which the copying ended, and a reference to the source.* Any code that is not marked in this way must be your own, which you personally designed and wrote. You may not share written answers or code with any other students, nor may you possess code written by another student, either in whole or in part, regardless of format.

Part 0: Getting started

For this project, we are assigning you to a team. 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 see a pull-down menu. Select `cs-b551-fa2019`. Then in the box below, you should see a repository called *userid1*-a1, *userid1-userid2*-a1, or *userid1-userid2-userid3*-a1, where the other user ID(s) correspond to your teammate(s). Now that you know their userid(s), you can write them an email at userid@iu.edu.

To get started, clone the github repository:

```
git clone git@github.iu.edu:cs-b551-fa2019/your-repo-name-a1
```

If that doesn't work, instead try:

```
git clone https://github.iu.edu/cs-b551-fa2019/your-repo-name-a1
```

where *your-repo-name* is the one you found on the GitHub website above. (If neither command works, you probably need to set up IU GitHub ssh keys. See Canvas for help.)

Part 1: The Luddy puzzle (20 pts)

Here are three variants of the 15-puzzle that we studied in class:

1. Original: This is the original puzzle. The game board consists of a 4x4 grid with 15 tiles numbered from 1 to 15. In each turn, the player can slide a tile into an adjacent empty space. Here are two sample moves of this puzzle:

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

2. Circular: In this variant, all of the moves of the original game are allowed but if the empty space is on the edge of the board, the tile on the opposite side of the board can be moved into the empty space (i.e., it slides off the board and "wraps around" to the other side). If an empty space is on a corner, then two possible "circular" moves are possible.

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

3. Luddy: This variant honors our school's building's namesake because all moves are in the shape of a letter "L". Specifically, the empty square may be filled with the tile that is two positions to the left or right and one position up or down, or two positions up or down and one position left or right.

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

The goal of the puzzle is to find the shortest sequence of moves that restores the canonical configuration (on the left above) given an initial board configuration. We've written an initial implementation of a program to solve these puzzles — find it in your github repository. You can run the program like this:

```
./solve_luddy.py [input-board-filename] [variant]
```

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 0
```

where 0 indicates the empty tile, and `variant` is one of `original`, `circular`, or `luddy`. We've included a few sample test boards in your repository. While the program works, the problem is that it is quite slow for complicated boards. Using this code as a starting point, implement a faster version, using A* search with a suitable heuristic function that guarantees finding a solution in is few moves as possible.

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, indicating the direction that a tile should be moved. For Original and Circular, the possible moves are `L`, `R`, `U`, or `D` for left, right, up, or down, respectively, indicating the direction that a tile should be moved. For example, the solution to the rightmost puzzle under Original (above) would be:

```
LU
```

For Luddy, the possible moves should be indicated with letters as follows: `A` for moving up two and left one, `B` for moving up two and right one, `C` for moving down two and left one, `D` for moving down two and right one, `E` for moving left two and up one, `F` for moving right two and up one, `G` for moving left two and down one, and `H` for moving right two and down one. For example, the two moves illustrated above in the Luddy section would be described with: `HD`.

Part 2: Road trip!

Besides baseball, McDonald's, and reality TV, few things are as canonically American as hopping in the car for an old-fashioned road trip. 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 the GitHub repo 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 command line like this:

```
./route.py [start-city] [end-city] [cost-function]
```

where:

- `start-city` and `end-city` are the cities we need a route between.
- `cost-function` is one of:
  - `segments` tries to find a route with the fewest number of "turns" (i.e. edges of the graph)
  - `distance` tries to find a route with the shortest total distance
  - `time` tries to find the fastest route, for a car that always travels at the speed limit
  - `mpg` tries to find the most economical route, for a car that always travels at the speed limit and whose mileage per gallon (MPG) is a function of its velocity (in miles per hour), as follows: $MPG(v) = 400\frac{v}{150}(1 - \frac{v}{150})^4$

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-segments] [total-miles] [total-hours] [total-gas-gallons] [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:

```
3 51 1.0795 1.9552 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.

Part 3: Choosing a team

In the dystopian future, AI will have taken over most labor and leadership positions. As a student in B551, you will assemble teams of robots to do the assignments for you. SICE will have a set of available robots for you to choose from. Each robot i will have an hourly rate P_i and a skill level S_i. You'll want to choose a team of robots that has the greatest possible skill (i.e., the sum of the skill levels is as high as possible), but you have a fixed budget of just B Intergalactic EuroYuanDollars (EYDs).

The robot names, skills, and rates will be given in a file format like this:

```
David 34 100.5
Sam 25 30
Ed 12 50
Glenda 50 101
Nora 1 5
Edna 45 80
```

We've prepared an initial solution for you, but it has a big problem. It is guaranteed to assemble the best possible team but only if you are allowed to hire fractions of robots:

```
[<>djcran@tank part3]$ ./choose_team.py people-small 200
Found a group with 4 people costing 200.000000 with total skill 69.029703
Nora 1.000000
Ed 1.000000
David 1.000000
Glenda 0.440594
```

As you can see, the program found a 4 person team that cost exactly the budget, but it hired only a portion (about 0.44) of Glenda's time. Your job is to write a program that assembles as skilled a team as possible but using only whole robots, and whose total cost is less than or equal to the budget. Please follow the same output format as above.

What to turn in

Turn in the three programs 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. **Your programs must obey the input and output formats we specify above so that we can run them, and your code must work on the SICE Linux computers.** We will provide test programs soon to help you check this.

Tip: These three problems are very different, but they can all be posed as search problems. This means that if you design your code well, you can reuse or share a lot of it across the three problems, instead of having to write each one from scratch.