CSC520 Fall 2019 Assignment 2

Due September 24^{th} at 11:59pm

This assignment includes **conceptual and code** questions. It must be completed individually. You may not collaborate with other students or exchange partial answers. Questions about your work must be emailed to the instructor or TAs, or discussed during office hours.

You must submit your code as a single self-contained zip file called Assign2.zip. This file must contain all code necessary to run along with a README file that specifies how to build and execute the code. Your grade will be based upon the execution of your code and on its compilation, clarity, and documentation. Your answers to the analysis questions should be uploaded as a single pdf file called Assign2.pdf. This will be graded on the completeness, correctness, coherence, and readability of your answers.

The reference platform for the code execution is the CSC520_VCL image which is available via the NCSU VCL platform. Verify that your code will run on that platform *before submission* as it will be used for grading.

Globe Puzzle

The Globe puzzle as described in Lecture 6 is a spherical tile rotation puzzle similar to a Rubik's Cube. The puzzle presents as a globe with three intersecting rings containing 12 tiles each. The rings are arraigned perpendicularly to one-another with one ring corresponding to the equator, and the other two corresponding to lines of longitude. The rings rotate freely with each tile occupying 30° of either latitude or longitude. In normal latitude and longitude the globe is divided into Noth and South Hemispheres with latitude being defined relative to the equator, while longitude is divided into East and West hemispheres relative to the prime meridian.

For the sake of this assignment we use a simpler notation where latitude coordinates run from 0° at the north pole to 180° at the south. Longitude goes for 360° around the globe. Thus in this coordinate system the vertical rings run along longitude 0° to 180° and 90° to 270°, and the equator ring contains tiles at latitude 90°. The vertical rings intersect at the "North Pole" (latitude 0°, longitude 0°) and the "South Pole" (latitude 180° and longitude 180°). And they intersect with the equator at the following points: (90°, 0°), (90°, 180°), (90°, 90°), and (90°, 270°).

We number the tiles around the rings as follows:

- Longitude 0/180: (0°, 0°), (30°, 0°), (60°, 0°), (90°, 0°), (120°, 0°), (150°, 0°), (180°, 180°), (150°, 180°), (120°, 180°), (90°, 180°), (60°, 180°), (30°, 180°)
- Longitude 90/270: (0°, 0°), (30°, 90°), (60°, 90°), (90°, 90°), (120°, 90°), (150°, 90°), (180°, 180°), (150°, 270°), (120°, 270°), (90°, 270°), (60°, 270°), (30°, 270°)
- Equator: (90°, 0°), (90°, 30°), (90°, 60°), (90°, 90°), (90°, 120°), (90°, 150°), (90°, 180°), (90°, 210°), (90°, 240°), (90°, 270°), (90°, 300°), (90°, 330°)

Because each tile covers 30° of either latitude or longitude then all rotations of the tiles will increment or decrement tile values by multiples of 30. Thus a single increment of the equator would increase all tiles

by 30° longitude save for (90°, 330°) which would become (90°, 0°). An increment of longitude 0/180 would increase the latitude of all tiles with longitude 0° by 30° and decrease all with longitude 180° by 30 save for (30°, 180°) which becomes (0°, 0°). A similar process takes place for the longitude 90/270 save that it also moves (0°, 0°) and (180°, 180°) to (30°, 90°) and (150°, 270°) respectively.

You have been given a set of files that describe marble puzzles. Each line of the file looks as follows:

```
Tile(30-180, (90,270), Exact(30,180))
```

This specifies a single tile. In this case we have a tile with ID `"30-180"` that is currently at latitude and longitude `(90,270)` and which has an exact target coordinates of latitude and longitude `(30,180)` to match. A puzzle is complete when all of its tiles are at their target locations.

The files are organized into two groups:

- `PathN-<N>.mb` specify a puzzle with a guaranteed solution of exactly N steps.
- `Puzzle-<N>.mb` specify a puzzle with a an unknown number of steps required.

The first set of files is intended to support debugging, the second set will be used to answer the questions below.

1 Heuristic (15 pts)

Define a novel heuristic function for this puzzle. Include an explanation of this heuristic in your report along with a justification for why it will guarantee an optimal solution with A* Search. Use this heuristic in your implementation below.

2 Implementation (65 pts)

Implement three search algorithms to solve the puzzle:

1. Breadth-First Search
2. A*
3. Recursive Best-First Search

Your code should be implemented in a single package called `Search.py` or `Search.java`. This code will be called as follows:

```
Search.py <ALG> <FILE>
```

Where `ALG` is one of: "BFS", "AStar", "RBFS". And `FILE` is the puzzle file name. When run your code should load the file, execute the search and return the following information:

- The number of states expanded.
- The maximum size of the queue during search.
- The final path length.
- The final path represented as a sequence of steps.

3 Analysis (20 pts)

Using the code above calculate solution results for each of your algorithms on the Puzzle files. Report the minimum, average, and maximum values for the number of states expanded and the queue size for each algorithm and identify the hardest puzzle for each. Consider the relative performance of the algorithms and state which algorithm is best for this problem. Justify your answer.