Issued: Jan. 23, 2018
Due: Mon., Feb. 5, 2018 (11:59pm to Canvas)

PROJECT #1: AI SEARCH WITH APPLICATION TO ROUTE PLANNING AND TSP

Please take care to complete all parts of this project independently. Do not use any solutions or code found online or written by other students. If you are struggling please go to office hours, post to Piazza, and/or send us email. We ask that you use one programming language for the search code in this assignment; either Python or C/C++ is fine, but not both.

Part I: Problem-solving without code (35 points)

Please submit a single PDF file `PartI.pdf` of all non-code problem solutions to Canvas. The PDF file can contain a scan of handwritten or typewritten/typeset solutions, and any supporting graphics.

1. UPDATED: R&N Problem 3.16: *Wooden (Toy) Railway Analysis* (10 points)

A basic wooden railway set contains the pieces shown in Figure 3.32 (see R&N text – both versions have this figure!). The task is to connect these pieces into a railway that has no overlapping tracks and no loose ends where a train could run off onto the floor.

a. Suppose that the pieces fit together exactly with no slack. Give a precise formulation of the task as a search problem.
b. Identify a suitable uninformed search algorithm for this task and explain your choice.
c. Explain why removing any one of the "fork" pieces makes the problem unsolvable.
d. Give an upper bound on the total size of the state space defined by your formulation. (Hint: think about the maximum branching factor for the construction process and the maximum depth, ignoring the problem of overlapping pieces and loose ends. Begin by pretending that every piece is unique.)

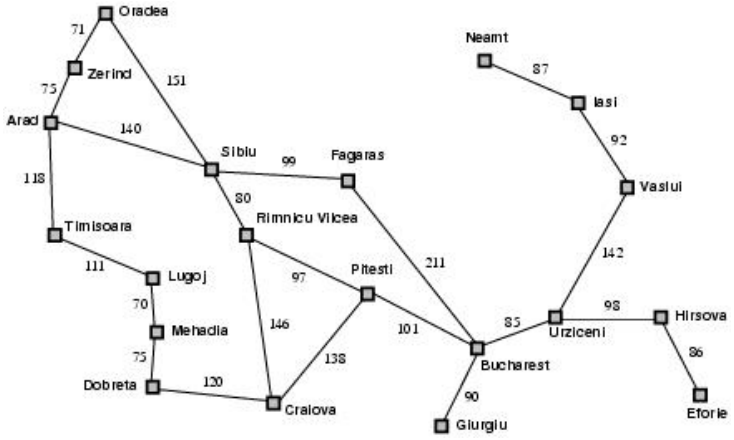
2. R&N Problem 3.8: *Analysis of Negative Path Costs* (10 points)

In Chapter 3 (p. 68), we said that we would not consider problems with negative path costs. In this exercise, we explore this decision in more depth.

a. Suppose that actions can have arbitrarily large negative costs; explain why this possibility would force any optimal algorithm to explore the entire state space.
b. Does it help if we insist that step costs must be greater than or equal to some negative constant c? Consider both trees and graphs.
c. Suppose that a set of actions forms a loop in the state space such that executing the set in some order results in no net change to the state. If all of these actions have negative cost, what does this imply about the optimal behavior for an agent in such an environment?
d. One can easily imagine actions with high negative cost, even in domains such as route finding. For example, some stretches of road might have such beautiful scenery as to far outweigh the normal costs in terms of time and fuel. Explain, in precise terms, within the context of state-space search, why humans do not drive around scenic loops indefinitely, and explain how to define the state space and actions for route finding so that artificial agents can also avoid looping.
e. Can you think of a real domain in which step costs are such as to cause looping?

3. "Manual" Search Tree Generation (15 points)

Consider the road map (graph) from R&N and our lecture slides:



Submit manually-generated search trees to plan a route from Hirsova to Sibiu as specified below. Label nodes in the sequence they are expanded (starting with node 1 at the tree root), and append the first letter of each city name (e.g., B = Bucharest) to indicate the state associated with each search tree node. Show the final search tree; shade in any "failure" (deadend) nodes removed during search, and label path costs. Specify each search strategy's solution as a route of travel (e.g., A→Z→O for travel from Arad to Oraclea) and total route cost. Indicate whether the identified solution is optimal. Note that you do not need a heuristic function for this problem.

a) Breadth-first Search
b) Depth-first Search
c) Uniform Cost Search
d) Iterative Deepening Search (show all generated trees starting with depth 3)

Part II: General Search Code with Case Studies (65 points)

Submit all your code for this project to Canvas as a single `tar`, `gzip` archive `search.tar.gz`. The grader will unzip your code into an empty directory and test it on CAEN Linux. Please make sure you create the archive from a local directory; do not include absolute directory pathnames in your archive. Include a `README` file that indicates use of C-11 (if applicable) and specific examples of code compilation (for C/C++) and execution for each domain. Any Python code you submit must be compatible with v2.7; to facilitate grading make sure you import any necessary modules at the top of your "main" python search code; these modules must be available on CAEN Python v2.7. You must write your search code from scratch, but you can use dictionaries, etc. per the Python lecture.

1. General (Generic) Search Code (35 points)

   Write a General Search Code in either C/C++ or Python. Your general search code should be capable of working with any particular "domain". In example C++ solution code we implement general search with two files: `mysearch.cpp` and `mysearch.h`. Each domain is specified in distinct `mydomain.cpp` and `mydomain.h` files. For Python we recommend a top-level file mysearch.py with any supporting files including the domain imported into this top-level file. You can give the grader specific instructions (ideally a one-line change) in README re: customizing imports or command line to switch between domains. If the grader struggles to follow your directions in configuring and executing your code on CAEN Linux you will lose points; please be clear and give examples in your README file.

   The general search code top-level function should accept two input (or command line) arguments: (1) the domain (e.g., file name, key word – indicate clearly in README), and (2) the search method to be used. The search method should be indicated as follows (case-sensitive): 'B' = breadth-first, 'D' = depth-first, 'I' = iterative deepening, 'U' = uniform cost, 'A' = A*. Only these five search strategies need to be implemented for this project. Feel free to specify internal to your code a depth limit value of, say 15, to assure termination in cases where a bug causes depth-first or iterative deepening to continue indefinitely.

2. Route planning (15 points): Test your code with the following route planning data from Southern Michigan, formatted in C but easily translatable to Python or CSV formats; note path segment distances are in miles:

```
const struct transition transition_set[] =
{{"Ann Arbor","Brighton", 19.2},
 {"Ann Arbor","Plymouth", 17.2},
 {"Ann Arbor","Romulus", 23.1},
 {"Brighton","Farmington Hills", 21.4},
 {"Brighton","Pontiac", 34.1},
 {"Plymouth","Romulus", 23.1},
 {"Plymouth","Farmington Hills", 14.0},
 {"Plymouth","Detroit", 27.9},
 {"Romulus","Detroit", 31.0},
 {"Farmington Hills","Royal Oak", 16.9},
 {"Farmington Hills","Detroit", 28.3},
 {"Farmington Hills","Pontiac", 15.5},
 {"Pontiac","Sterling Heights", 17.2},
 {"Pontiac","Royal Oak", 13.3},
 {"Romeo","Pontiac", 27.8},
 {"Romeo","Sterling Heights", 16.5}};
```

```
const struct latlon coords[] =
{ {"Ann Arbor", 42.280826,-83.743038},
  {"Brighton", 42.529477,-83.780221},
  {"Detroit", 42.331427,-83.045754},
  {"Farmington Hills", 42.482822,-83.418382},
  {"Plymouth", 42.37309,-83.50202},
  {"Pontiac", 42.638922,-83.291047},
  {"Romeo", 42.802808,-83.012987},
  {"Romulus", 42.24115,-83.612994},
  {"Royal Oak", 42.48948,-83.144648},
  {"Sterling Heights", 42.580312,-83.030203}};
```

Note that Latitude and Longitude data can be converted to Cartesian ECEF (Earth Centered Earth Fixed) coordinates with the following (python) code where `phi` = latitude (in radians), `theta` = longitude (in radians), and `R` = 3959 miles (radius of Earth). You can compute Cartesian coordinates and reasonably assume a "flat Earth" for this project with the following python code:

```
# python
x = math.cos(phi) * math.cos(theta) * R
y = math.cos(phi) * math.sin(theta) * R
z = math.sin(phi) * R # z is 'up'
```

The grader will need to run your code with the following three inputs: (1) Origin (e.g., Ann Arbor), (2) Destination (e.g., Detroit), and (3) Search algorithm choice (B, D, I, U, A). Clearly specify an example execution command in your README file. Your edge costs (*g(n)*) should be actual path length from origin to current location in miles, and your heuristic (*h(n)*) should be straight line (3-D ECEF) distance from current location to the goal based on the given latitude, longitude values. Once a solution is computed, your code must print to the screen the following information: (1) Total number of nodes expanded, (2) Solution path (e.g., Ann Arbor → Plymouth → Detroit), and (3) Total solution cost *g(n)*.

3. Travelling Salesman Problem (TSP) (15 points): Using the Southeast Michigan route map from the last problem, now set up your search code so that the user can enter the a starting location and the "goal" will be a path that visits all listed cities at least once. Path cost *g(n)* should still be total distance traveled along a TSP path from the initial city to node *n*. Note that you do NOT have to return to the initial city; instead you can remain in the last city visited. Specify your admissible heuristic function *h(n)* in the README file and explain why it is a good choice. The grader will need to run your code with the following two inputs: (1) Origin (initial state) (e.g., Ann Arbor), (2) Search algorithm choice (B, D, I, U, A). Clearly specify an example execution command in your README file. Once a solution is computed, your code must print to the screen the following information: (1) Total number of nodes expanded, (2) Solution path (same format as in last problem; all cities must appear at least once), and (3) Total solution cost *g(n)*.

UPDATE (Jan. 24): The grader has asked that the input for each search problem be specified in a file with the below format. Points will be deducted if your code does not allow this format. For problem 2, you should read input from file `route.txt`; for problem 3, you should read input from file `tsp.txt`.

Example `route.txt` (plan a route from Ann Arbor to Detroit using A* search):

```
Ann Arbor
Detroit
A
```

Example `tsp.txt` (start the TSP search from Ann Arbor using breadth-first search):

```
Ann Arbor
B
```