Issued: Jan. 27, 2020
Due: Mon., Feb. 10, 2020 (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 (40 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. R&N Problem 3.15 (10 points)

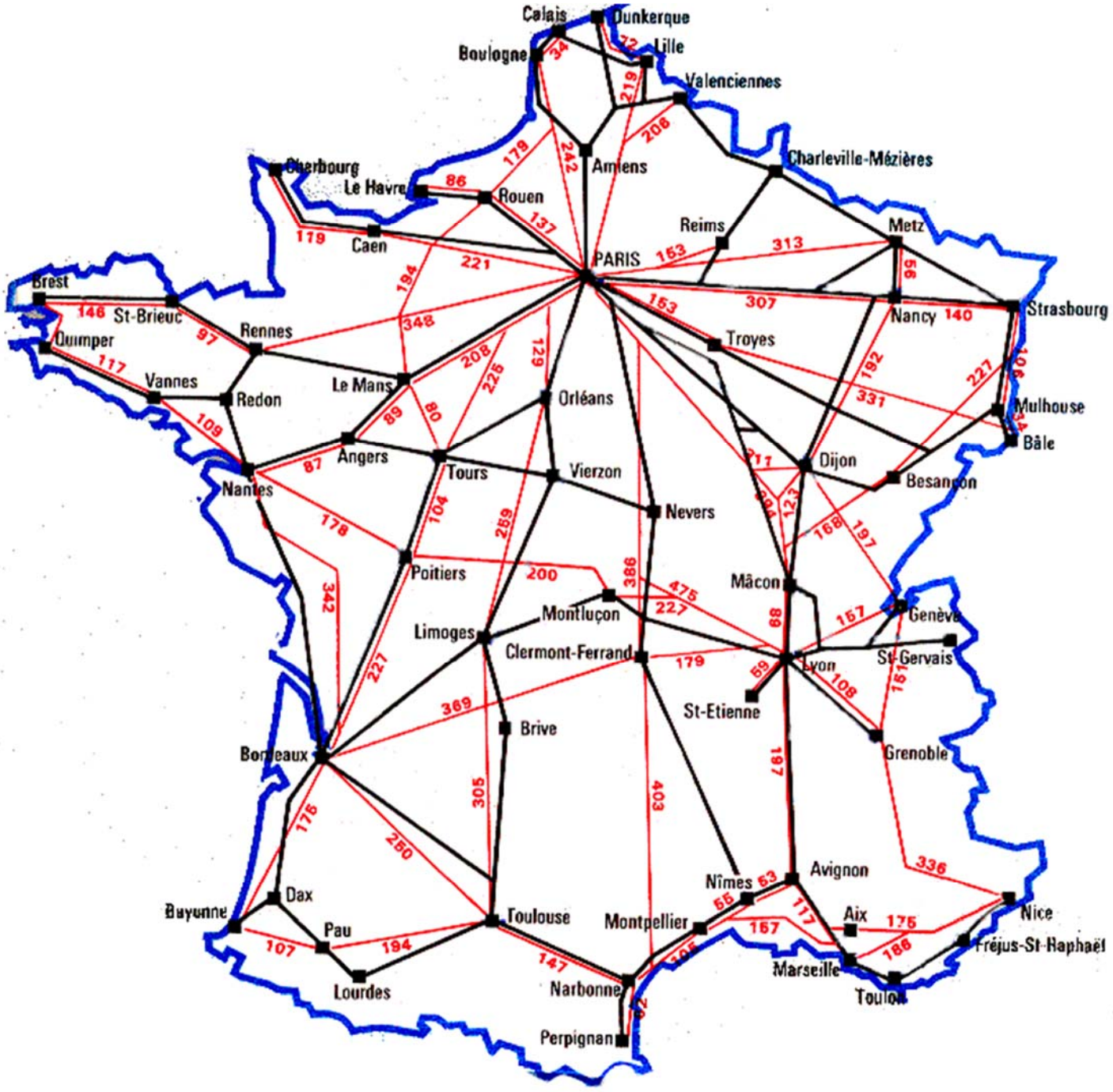
Consider a state space where the start state is number 1 and each state $k$ has two successors: numbers $2k$ and $2k + 1$.
a. Draw the portion of the state space for states 1 to 15.
b. Suppose the goal state is 11. List the order in which nodes will be visited for breadth-first search, depth-limited search with limit 3, and iterative deepening search.
c. How well would bidirectional search work on this problem? What is the branching factor in each direction of the bidirectional search?
d. Does the answer to (c) suggest a reformulation of the problem that would allow you to solve the problem of getting from state 1 to a given goal state with almost no search?
e. Call the action going from $k$ to $2k$ Left, and the action going to $2k + 1$ Right. Can you find an algorithm that outputs the solution to this problem without any search at all?

### 2. R&N Problem 3.27 (10 points)

$n$ vehicles occupy squares (1,1) through ($n$,1) (i.e. the bottom row) of an $n \times n$ grid. The vehicles must be moved to the top row but in reverse order; so the vehicle $i$ that starts in ($i$,1) must end up in $(n - i + 1, n)$. On each time step, every one of the $n$ vehicles can move one square up, down, left, or right, or stay put; but if a vehicle stays put, one other adjacent vehicle (but not more than one) can hop over it. Two vehicles cannot occupy the same square.
a. Calculate the size of the state space as a function of $n$.
b. Calculate the branching factor as a function of $n$.
c. Suppose that vehicle $i$ is at $(x_i, y_i)$; write a nontrivial admissible heuristic $h_i$ for the number of moves it will require to get to its goal location $(n - i + 1, n)$, assuming no other vehicles are on the grid.
d. Which of the following heuristics are admissible for the problem of moving all $n$ vehicles to their destinations? Explain
(i) $\sum_{i=1}^{n} h_i$.
(ii) $\max\{h_1, \ldots, h_n\}$.
(iii) $\min\{h_1, \ldots, h_n\}$.

3. “Manual” Search Tree Generation (20 points)

Consider the map (graph) shown below. The black segments are roads (ignore for this problem), and the red segments are train routes and distances (in km). Assume you can only actually reach the cities served by train for this problem, so you can ignore the others (e.g., Redon, Nevers). Use a ruler or online measurement device to determine approximate straight line distances from this map for your heuristic. For calibration purposes, assume the straight-line distance from Paris to Toulouse is 600 km.



Submit manually-generated search trees to plan a route from Nice to Paris 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., P = Paris) to indicate the state associated with each search tree node. If multiple cities have the same first letter, designate a key in your solution (e.g., P1=Paris, P2=Pau). 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 and total route cost. Indicate whether the identified solution is optimal. Note that you only need to specify heuristic estimates (straight-line distance) for A* so you only need to compute straight-line distances as you expand nodes in the particular search for a route from Nice to Paris.

a) Breadth-first Search
b) Depth-first Search
c) Uniform Cost Search
d) A* Search

*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 v3.6; 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 v3.6. You must write your search code from scratch, but you can use dictionaries, etc. per the Python lecture. We will check your code with cpplint/pylint.

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.

   Once your domain is selected, your code should then read from a domain specific problem .txt file, either route.txt (Route planning) or tsp.txt (Travelling Salesman Problem). The search method will 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. Example files are given below:

   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
```

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 using the Route Planning domain and will change route.txt to test your code (as explained in 1)). 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 to go from a starting location and have the "goal" 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 using the TSP domain and will change tsp.txt to test your code (as explained in 1)). 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)*.