**RAHUL JONUKUTI**

**16341703**

**Program 1: Guided Search**

For this assignment you’ll be doing a small example of a large problem—route-finding.

Given a list of cities and their adjacencies—from city A, what cities are next to it—can a route be found from city A to city X?

Obviously, an undirected brute-force approach **will** do this, but a brute-force approach is far from ideal for the general case. As we discuss in class, a pure breadth-first search, while guaranteed to work, is slow, and has a large memory requirement. We **can** do better!

* For this program, you’ll be implementing a heuristically guided search:   
  Maintain a list of cities you know how to reach.
* At first, this place will have a city: origin.
* The cities nearby the starting point are the ones you know how to reach; Cities that you may reach but have not yet visited (and therefore do not yet know whether they are on the route or not) form the border.
* Which border town should I choose next?
* To make this a little less complicated, we'll first go back and implement the search in the best way, you'll rank the cities you might visit (but haven't yet visited) depending on the distance from your starting point. point. In a more efficient A\* search, you choose the result that is closest or estimated to be closest to your destination.
* If you're trying to find a route from Kansas City to Denver, you'll always go west if possible. If you're already halfway there and looping to the east, you'll try to get back to the west as soon as possible, and so on.
* From the new city you choose, update the border and reevaluate which city on the border is closest to the destination.
* What if you find yourself at a dead end? Let's say a particular city looks promising, but you get there and find that there are no directions to your destination. In this case, you'll eventually run out of places you can access from, and the best-guess-first approach will keep you from going too far off course.
* Additionally, if the detour becomes too important, the algorithm will have to return to another border city and try again from there.
* For each city visited, you may want to record where you got to that city so the list can be used to reconstruct the route you've taken so far.

An example of path selection:   
We start with city A. It is adjacent to cities B, C and D, forming a boundary.

C is the closest place to our destination, so we chose C to visit.

C is adjacent to D (already bordered), F, M and P.

F, M and P are added to the boundary, and we note that we have arrived at C from A. M is closest to the destination, so we choose M, noting that we got to M from C.

From M can go to J, N and R.

R was closest to our destination, so we chose it noting that we were coming from M.

R is adjacent to W, P, Q and Z: our target.

Z is closest to our target (distance 0), so we add it to the visited cities, noting that we came here from R. We're on target, Z.

We come to Z from R.

We came to R from M.

We came to M from C.

We came to C from A. So... the route is A → C → M → R → Z.  
**Programming details:**

• You can write your program in any of: C, C++, Python, C#, Java, or Matlab. If you want to use some other language I’m willing to discuss it but you’ll need to come to me in advance.

• You’re given 2 data files.

◦ First is a list of all the towns we know of (mostly small towns in southern Kansas), along with the latitude and longitude of each town.

For this program, we're not interested in how much closer one city is than another, but only in finding out which city is closest. So we can take the coordinates as points in the XY plane and use the standard Euclidean distance formula.

# The name has been changed so that the city name consists of more than one word with underscores instead of spaces between words (South\_Haven instead of South Haven), to simplify typing.

◦ List of records for each municipality and neighboring municipalities. This file is line oriented. The first city on the path is the city of interest, and the remaining cities (various numbers) are adjacent to the first city (not necessarily each other).

# Note that this file is not complete. Cities must appear in each other's neighbors list because neighborhoods are symmetrical:

If A is adjacent to B then B is adjacent to A. This may not be listed correctly. [In other words, it is possible that A is listed adjacent to B, C, and D, and D adjacent to E, F, and Q; obviously it must also be adjacent to A.]

--> Take this into account when setting up your program’s data structures. If adjacency is listed in either direction, it should be considered present in both directions.

• Ask the user for their starting and ending towns, making sure they’re both towns in the database. Use either best-first or A\* search to find a route to the destination, if one exists; print the route you find in order, from origin to destination. You might be interested in looking at a map and see how the route your program finds compares with reality. Note that your database is very limited, and I left out a lot of roads and routes.

\*Also, all we’re looking at is adjacency, not distance between adjacent cities. If you put into the database that Topeka is adjacent to Denver, it’s going to find a route between them very quickly, and say it’s ‘only’ one step. On the other hand, a real mapping application often gives directions such as “get onto I-70, and go west until you reach the I-225 exit at Aurora, Colorado.” It’s only one step, but it’s a large one.   
  
**Submission**: Either zip up the project folder from your IDE and upload it to Canvas (or just your source code files), or a link to a GitHub or GitLab repo.