CS 461
Fall 2022
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, a brute-force approach like Dijkstra's algorithm 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 if we try to search more intelligently. For this program, you'll be implementing a *best-first* search: Maintain a list of cities you know how to reach. At the beginning, this will have one city—the origin. Cities adjacent to the origin are cities that you know how to reach; cities that you can reach, but have not yet visited (and so don't yet know if they're on the route) form the *frontier*. Which city on the frontier should you choose to visit next? In a pure best-first search, you'll order the cities you can get to (but haven't visited yet) by distance from your starting point. In the more-efficient A* search, you choose the one that is closest, or estimated as closest, to your destination. So if you're trying to find a route from Kansas City to Denver, you'll consistently head in a generally western direction if possible. If you're partway there and have to detour eastward, you'll try to return to westward travel as soon as possible, and so on. From the new city you select, update the frontier, and re-assess which city on the frontier is closest to the destination.

What if you hit a dead end? Suppose a particular city looks promising, but you get there and find there's no route to your destination. Then you will eventually exhaust the places you can get to from there, and the best-first heuristic will keep you from getting too far afield. If the detour is getting too large, the algorithm will return to a different frontier city and try from there.

For each city visited, you will probably want to store where you came to that city from. This can be used to reconstruct the path.

An example:

We start at city A. It's adjacent to B, C, and D, which form the frontier. C is closest to our destination, so we choose C to visit. C is adjacent to D (already on the frontier), F, M, and P. F, M, and P are added to the frontier, and we make a note that we came to C from A. M is closest to our destination, so we choose M, making a note that we came to M from C. From M, we can reach J, N, and R. R is closest to our destination, so we choose it, making a note that we came from M. R is adjacent to W, P, Q, and Z—our goal. Z is closest to our goal (distance 0) so we add it to the visited cities, noting we came here from R.

We're at the goal, Z.
We came to Z from R.
We came to R from M.
We came to M from C.
We came to C from A.
Therefore the route is A → C → M → R → Z.

Programming details:

- You can write your program in any of: C, C++, C#, Java, Python, Racket. If you want to use some other language I’m willing to discuss it but you’ll need a pretty good reason.
- You’re given 2 data files.
  - The first is a list of all of the cities we know about—mostly small towns in southern Kansas—and the latitude and longitude of each. For this program, we’re not interested in *how much* closer one town is than another, only finding which one *is* closer. Therefore we can take the coordinates as points in the XY plane and use the standard Euclidean distance formula. Names have been tweaked so that city names consisting of more than one word have an underscore rather than a space between the words (South_Haven rather than South Haven), to simplify input.
  - A file listing, for each town, towns that are adjacent to it. This file is line-oriented. The first town on the line is the town of interest, and the remaining towns (the number is variable) are adjacent to the first (not necessarily each other). Note that this file (being hand-constructed late at night) is not entirely complete. Towns should appear on each others’ adjacency list, since adjacency is symmetric: If A is adjacent to B, then B is adjacent to A. This may not be listed properly in all cases. That is, it’s possible A is listed as adjacent to B, C, and D, and D as adjacent to E, F, and Q; obviously it should 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.