Weighted Graph Searching with Train Routes

Eckel, TJHSST AI1, Fall 2020

Background & Explanation

So far, we have been searching on unweighted graphs, meaning graphs where any movement costs the same amount (one move is one move, regardless of the direction). This lab applies the same principles to a weighted graph search, where some movements cost more than others.

The weighted equivalent of BFS is called Dijkstra's algorithm. Put simply, this algorithm always chooses the node with the least total distance from the source as the next node to process. To implement this, we would use a heap that sorted only on total distance from source. This is analogous to a BFS search because it will try every node in an increasing distance from the source exhaustively. We can also apply A* to this situation. To implement this, we would use a heap that sorted on total distance from source plus a consistent, admissible heuristic estimate of the remaining distance.

After implementing each search, we will use a graphics library to animate the searches, seeing how they both work!

On the website, you will find several files:

- "rrNodes.txt" contains a list of nodes representing the Chicago Train Authority's actual dataset of train junctions in the US, Canada, and Mexico, with an id tag and then latitude and longitude location.
- "rrEdges.txt" contains a list of connected junctions as pairs of id tags, thus describing all available train routes. If two nodes appear as a pair in this file, there is a train track directly from one to the other. All tracks are bidirectional.
- "rrNodeCity.txt" identifies certain junctions with names (these are the junctions located in major cities). These cities will be the available options for starting and ending points you might be given. Note that some names contain **spaces**; your code should be able to read in the **full** city names. There is no city named "San".
- "distanceDemo.py" is a Python file containing the code you **MUST USE** to calculate distance between any two latitude/longitude pairs. This is called "great circle distance" and is necessary to replace the standard distance formula because our world is a planet, not a plane. If we all use the same code, all of our answers are guaranteed to match to a high degree of precision, so **USE THIS CODE. DO NOT WRITE YOUR OWN IMPLEMENTATION OF GREAT CIRCLE DISTANCE**. Thank you for making this easier for me to grade!
- "tkinterDemo.py" is a Python file demonstrating the use of Python's built in graphics library tkinter, graciously written by Tarushii Goel in 1st period to make this assignment take less time for everyone else. You can ignore this until you reach part 3 of this assignment. No need to play around with it yet.

Part 1: Modeling & Searching the Actual North American Train Network

Using this information to model the actual train network can be a slightly confusing process. Here are a few points to consider carefully before you begin coding. You'll want to create the entire network before you search.

- As a backing data structure I recommend a dictionary where each junction is matched with a list or set of tuples. Each tuple should contain one junction that the key junction can reach directly (ie, one junction paired with the key junction in rrEdges.txt) **and the distance between them**. This will be our weighted graph.
- You'll need to calculate those distances! This is because this data set contains lat/long pairs for each train junction but not lengths of the train tracks themselves. This isn't actually a problem; to find the length of any given track between junctions, **we will find the great circle distance between the two junctions** (see distanceDemo.py). Train tracks have to be very straight in order to work properly, so this is an approximation, but it's a good one. For the purposes of the rest of the assignment, the great circle distance from one junction to the other in a given pair will be considered the correct length of that track.
- Be sure to add each pair in rrEdges.txt to the dictionary twice; each node should be a child node of the other.

Once the weighted graph has been created, implement Dijkstra and A* searches to find ideal paths between any two named cities. Both searches should give correct answers; A* should be faster.

Dijkstra's Algorithm search on our weighted graph:

- This algorithm is analogous to BFS in that it exhaustively tries the nodes closest to the source first. However, because each edge of the graph has a different weight, when you code this it looks more like A* than BFS, in that the fringe should be a *heap*. Copy your A* code from a previous assignment and modify that rather than trying to copy BFS. The only difference between Dijkstra code and A* code is that Dijkstra does not use a heuristic – the heap simply sorts on depth, and the next node to be processed will be the node with the shortest total distance from the origin so far. The depth is no longer the number of moves, as it was in BFS; now, the depth is the **sum of the lengths of every track the path has traversed so far, from the starting node to the current node.** So, when generating children, each child should be added to the heap with its own depth at index 0 in the tuple, and that depth can be found by adding the parent's depth plus the distance from the parent to the child, which you should be able to look up in your backing dictionary.

A* Search on our weighted graph:

- Bear with me here; this is pretty confusing. Remember that A* uses the *actual* distance so far plus an *estimated* distance to the goal as its value for prioritizing. *Actual* distance so far should be the same as Dijkstra – the sum of the great circle distances along the actual lengths of track traversed so far. *Estimated* distance to the goal *is also great circle distance*, but used differently. This confuses students every year so read carefully! The estimated remaining distance is a new calculation of great circle distance from the current node *directly* to the goal, as if there was a single track connecting them. It stands to reason that the *actual* distance to the goal cannot be smaller than that, so it's a guaranteed underestimate, which is what A* mathematically requires. To be clear:
    - The great circle distance along each *actual* piece of track is *pre-calculated*, stored in a dictionary, looked up, and used to calculate *depth*.
    - The great circle distance from the current node directly to the goal is an *estimate* of the remaining distance, is calculated *when you put children onto the heap*, and is used to calculate the *heuristic* part of the A* algorithm.
    - Read this section at least twice, slowly. Ask questions early and often.

- I'd like to reiterate one crucial aspect of A* one more time. A* does *not* goal test children when they are added to the heap. The first time you get a path to a child might not be the fastest. This *will* happen on this assignment! You only guarantee the shortest path to any particular node when you pop it off the heap for the first time. It may be on the heap several times with different paths at that point – this is normal! Double check and make sure you only goal test, and only add to closed, when *removing* a node from the heap.

- **Warning:** the provided great circle distance code will sometimes crash when asked to calculate the distance from a node to itself. Either modify that function to detect this circumstance and return 0 specifically, or modify your search code to check if you ever are calling distance on the same node twice and skip the call in that case.

Once again: both searches should return identical distances in the final calculation, and A* should be faster. On the next page are some test cases to verify this.

Test Your Code So Far

Modify your code to take two city names as command line arguments, and run both searches and print the results and the time taken for each search. Along the way, it should also print the time taken to build the backing dictionary. Here is some sample output to verify against:

```
>python train_search.py Albuquerque Atlanta
Time to create data structure: 0.09579829999999999
Albuquerque to Atlanta with Dijkstra: 1457.5893495571672 in 0.015421400000000002 seconds.
Albuquerque to Atlanta with A*: 1457.5893495571672 in 0.005043700000000012 seconds.

>python train_search.py Leon Tucson
Time to create data structure: 0.09520499999999998
Leon to Tucson with Dijkstra: 1153.3469371549922 in 0.0022250000000000048 seconds.
Leon to Tucson with A*: 1153.3469371549922 in 0.0006297999999999859 seconds.

>python train_search.py "Ciudad Juarez" Montreal
Time to create data structure: 0.09547239999999999
Ciudad Juarez to Montreal with Dijkstra: 2125.627229333468 in 0.026318800000000003 seconds.
Ciudad Juarez to Montreal with A*: 2125.627229333468 in 0.009051500000000018 seconds.
```

Important notes:

- Notice these searches are all quite fast, but A* is definitely faster, and the distances match precisely.
- In this lab, even small errors in the distance calculations are almost certainly indicative of wrong code (since we have the same implementation of great circle distance). Any differences are solely due to differences in OS default precision & Python versions, which is to say, there won't be much difference at all. Your answers must match mine to at least three decimal places, and are likely to match more than that. Of course, feel free to check additional test cases against your peers.
- Note the use of quotation marks in the third example to submit a command line argument that contains a space.

Specification for Part 1

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code does all of the following (see sample run below):
  - Automatically build the weighted graph. You can assume I will have the three data files in the directory where I'm running your code, and they will have the same names as I gave on the front of this sheet. This means you **MUST NOT RENAME THESE FILES IN YOUR OWN CODE.** Sorry for yelling, but it's been a common mistake in previous years!
  - Outputs the time taken to build the complete weighted graph.
  - Accept two **command line arguments** – the names of two cities.
  - Run each algorithm to find the distance between the two cities. There should be two lines of output, one that says "Dijkstra: " and then a distance and then a run time; another that says "A*: " and then a distance and a run time. The distances should match, and both run times should be quite short.
- Total runtime is less than 10 seconds. (My code builds the data structure and runs both algorithms on all three examples above in less than a quarter of a second total; this is very achievable.)

Part 2: Animation!!

Now let's have some fun.

Tkinter is Python's traditional default graphics package; it doesn't need to be downloaded (it's included with Python's standard distribution) so it can be added to your code by just putting "import tkinter" at the top of your code.

A demo is provided for you to see an example of how to do animation in tkinter. Go ahead and run it, then take a close look at the code.

Your goal here is to use the latitude and longitude pairs given to locate coordinates and draw each edge of the graph as a black line. Then, as a search algorithm runs, you'll color each connection it processes. Once your search is complete, you'll turn the final path found a different color as well. So, at minimum, there should be three colors – one for untraversed edges, one for edges that have been traversed while adding children to the fringe, and a third color for the correct final path that appears once the search is completed.

Let's break that down into steps.

- Just as the demo begins with a black grid on the screen before edges start turning red, you'll want to begin with every train track shown in a complete black network on the screen before the search algorithms start coloring in edges. Each pair of nodes in rrEdges.txt should produce a line drawn between them on the screen.
  - This will be sort of annoying to get right. You'll need to come up with some math that will take the given latitude, longitude coordinates and convert them to x, y pairs within the space shown on the screen. Remember that latitude is y and longitude is x.
  - As a hint, you can swap a whole set of points up and down if you replace y with (max_y_value – y) instead; ie, if your window is 1000 pixels tall, swapping each y with (1000 – y) will turn the image upside down.
  - You'll need to draw each line in black as you create it. You'll also need to be able to *access* each line later, so you can change its color. In the demo file, Tarushii just looped over every line she had created in the order they were generated. You'll need to store the lines you created differently, so you can access them individually while the search process takes place. Perhaps a dictionary, pairing a key that is a tuple of two node id numbers with a value that is a canvas line object? That's one idea, but feel free to find a solution you like better!
  - See an example of what this part should look like on the next page.

- Then, animate a search algorithm. Each time you add a child to the fringe, you should turn the connection between it and its parent a different color (eg, processed connections turn red during Dijkstra, then blue during A*). If this slows your graphics down too much, feel free to only call update every 10 steps. Or 50. Or 2000. Or whatever.

- Once the search algorithm finds the goal, it should turn the final path a different color (eg, green). So, you'll need to keep track of your path (with information stored on the fringe, for these algorithms, not some kind of overall dictionary) and at the end trace the path and turn those connections a final color.

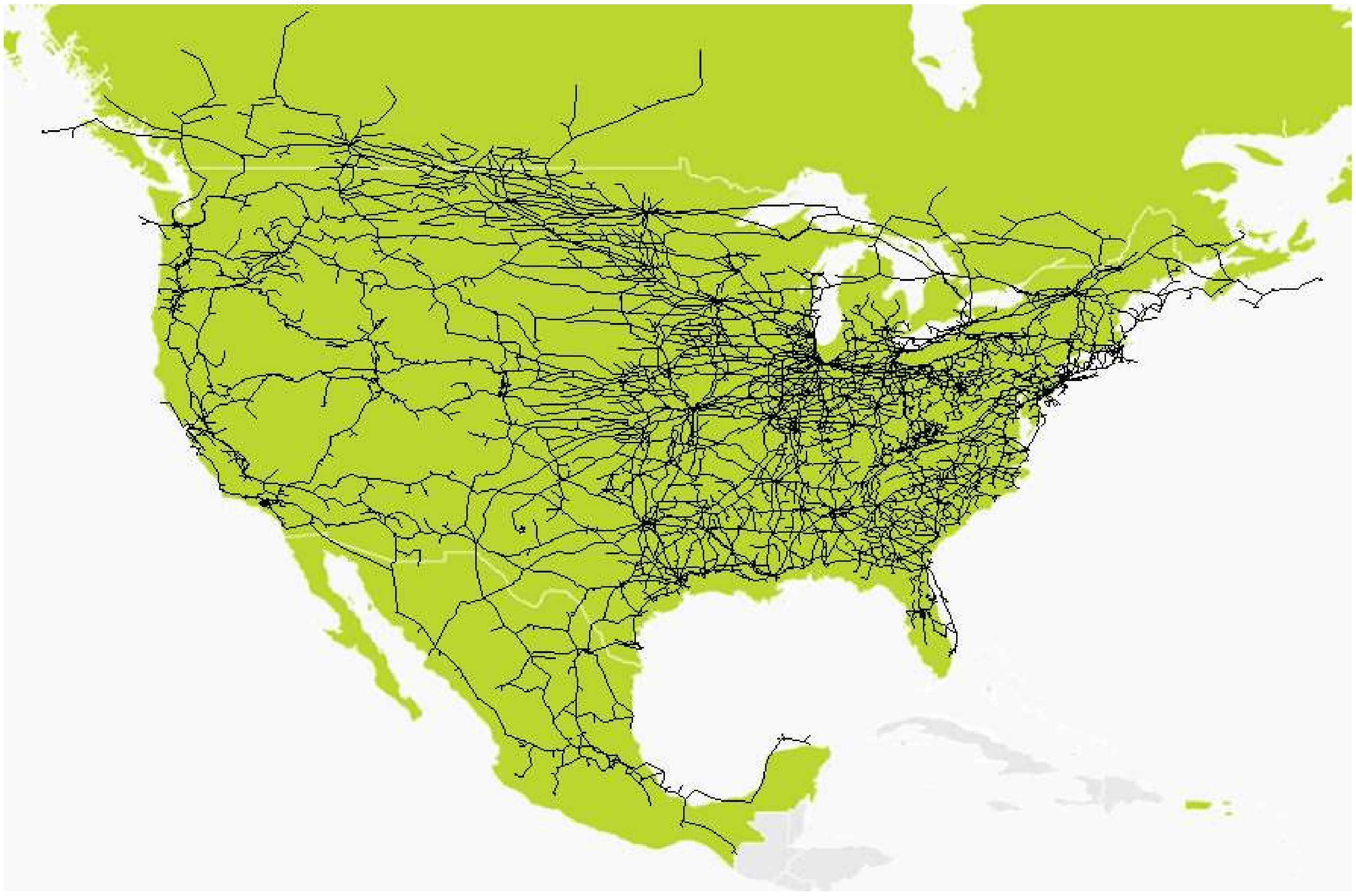
- The easiest assignments for me to grade are the ones that run one search and freeze at the end, waiting for me to close the window (just like the demo). Then, when I close one window, open another window and do the other search algorithm, which itself freezes until I close the window. A little experimentation should find this!

I give you complete freedom to tinker with this until you like it. Speed it up by modifying how often the screen updates. Change the colors. Label the graph. Whatever makes you happy. The only conditions are that we can see the difference between the algorithms and it doesn't take forever to run.

Example network map

This is an example of what your map might look like. Here, a student from last year added an image of a map of North America behind the train network; **you don't have to do that**. And if you do add a map, it's also likely not to line up perfectly, as you'll notice this student's doesn't – some of these tracks appear to head off into the ocean!

When I run your code, before the search algorithm starts running I'm expecting to see a network that looks like this but without the backing image (unless you want to add it as part of Outstanding Work, or just for fun.)



Specification for Part 2

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code takes two city names as command line arguments (see previous page).
- Your code runs Dijkstra, and shows the algorithm progressing in animated form in a graphics window.
- Your code runs A*, and shows the algorithm progressing in animated form in a graphics window.
- Total runtime for both algorithms combined is less than 2 minutes. (Test Ciudad Juarez to Montreal for a good double check that you're fast enough; update less frequently if you aren't.)
- You don't import any graphics libraries aside from tkinter.

Specification for Outstanding Work: Unusually Awesome Animation

One way to extend this assignment and receive outstanding work credit is to work with Tkinter to make this something truly extraordinary; something that I could show to students next year to really impress them. Submit to the link on the website.

This extension **receives outstanding work credit** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- You still don't import any graphics libraries aside from tkinter. (This is just so I can make sure I can run it on my end! I'm willing to accept requests for leniency here, but unless I specifically say yes, assume just tkinter.)
- City selection and algorithm selection is done via a GUI. No command line arguments needed.
- I can reset the map and run a different pair of cities / different algorithm without restarting the script.
- There are speed settings that allow me to watch a search between two fairly close cities slowly, seeing every step, or speed it up to watch a search from one side of the continent to another.
- There is a map of North America lined up as nicely as possible behind the city grid. (Due to differing projections it's unlikely this will be perfect, but you can come pretty close.)
- There is a fourth color for the fringe. At any moment while any algorithm is running, I should be able to see unvisited connections, connections to nodes on the fringe, and connections to nodes in the closed set in different colors. Once the algorithm finishes, a fourth color identifies the final path.
- There is a third animation option for a naïve DFS search. This should wander randomly around the map, and is usually pretty entertaining to watch.
- This is totally optional, but it would also be insanely cool to make a Bidirectional Dijkstra's algorithm (a la Bidirectional BFS); I'd love to see that run!