CSC 301 Project 4: MST/TSP Spring 2025

Due: Wednesday, April 16th

In this project, you will implement both Prim's and Kruskal's algorithms for finding a Minimum Spanning Tree. You will then use that MST to approximate the solution for the Traveling Salesman Problem. You will be required to implement your solutions in their respective locations in the provided `prim.cpp`, `kruskal.cpp`, and `tsp.cpp` files.

Contents

1 Problem Statement

You will have three primary tasks in this project:

1. Implement Prim's algorithm to find the MST.
2. Implement Kruskal's algorithm to find the MST.
3. Implement a DFS function to find the tour once you have found the MST.

2 Traveling Salesman Problem

Suppose you are a salesperson who will need to meet with a number of clients living in various cities. To give you as much time with the clients as possible, you would like to minimize the time you spend traveling between the different cities. Specifically: we can create a *complete* graph where the cities are the vertices and the edge weights are given as the distance (on the sphere) between all the pairs of cities. We want to find the cheapest **tour** of the graph that visits every vertex only once outside of returning to a given starting city.

This is a variation of the classic Traveling Salesman Problem (TSP), where you would like to minimize the *cost* of the tour. The variation we are solving in this project defines the 'cost' between two cities to be the great circle distance between them.

The Traveling Salesman Problem is an example of an NP-Complete problem. These problems are very hard to solve. In fact, there are no known polynomial time algorithms that can solve any of them. All known algorithms perform in exponential time, or worse. As an example, the naive approach (called **brute force**) would be to simply check every possible tour looking for a minimum tour. If there are n cities, this will take $(n-1)!$ time (number of permutations of the non-start cities).

While we do not know of an efficient algorithm to optimally solve this problem, there is an approach that promises an approximation within a factor of 2 of optimal, i.e., if the optimal solution has cost $|\sigma|$, then this approximation has cost $|A| < 2|\sigma|$.

The idea for this approach is based on an important observation: the minimum spanning tree of a graph will always have less cost than a tour. This is because the minimum spanning tree is the smallest set of $|V| - 1$ edges in the graph that do not contain a cycle, and removing an edge from the tour gives a spanning tree of size $|V| - 1$. This resulting spanning tree must cost at least as much as the minimum spanning tree, and therefore the original tour costs more, i.e., $|\sigma| > |M|$ where $|M|$ is the cost of the minimum spanning tree and $|\sigma|$ is the cost of the optimal tour.

We would like to somehow use the MST to find a low cost tour of the graph. We know that the MST has to connect to our start city, which means we could start there and traverse the entire tree using depth first search, which visits each city along each of the MST edges twice: once during the unwind (pre-step) and once during the rewind (post-step). So this traversal will cost $2|M|$. Note that it is not a proper tour because we would visit each vertex twice.

However, since every city is connected to every other city, we can traverse our cities in the same order that DFS would, but we can skip any cities we have already visited. As long as the costs in the graph satisfy the triangle inequality, our approximation will be at worst $2|M|$. Therefore

$$|M| < |\sigma| < |A| < 2|M| < 2|\sigma|,$$

and so our approximation will be within a factor of 2 of the optimal solution. We will discuss this derivation further in class.

3 Provided Code

Your goal in this project will be to implement both Prim's and Kruskal's algorithms to find Minimum Spanning Trees, and then use these MSTs to approximate the solution to the Traveling Salesman Problem.

To aid you with this, you have been provided with fully functioning Vertex and Edge classes.

3.1 Vertex Class (in `Vertex.hpp` and `Vertex.cpp`)

The first of the fully functioning classes is the Vertex class. This class has 4 class attributes:

- `int label`: the label of the given vertex
- `string city`: the string label of the city associated with the vertex
- `vector<int> neighbors`: the vector of the neighboring vertices
- `vector<int> mstNeighbors`: the neighbor vector for the MST

Along with these are two implemented member functions:

- `==`: overloaded equality test that compares the labels of the two vertices.
- `printVertex`: produces a readable string with the vertex's info.

3.2 Edge Class (in `Edge.hpp` and `Edge.cpp`)

You have also been provided with a fully functioning Edge class with 3 class attributes:

- `Vertex first, Vertex second`: the Vertex objects associated with this edge
- `weight`: the weight of this edge

The Edge class has 7 member functions:

- All of the comparison operators have been overloaded to compare based on edge weight.
- `printEdge`: this function returns a string that prints the Edge in a readable fashion (though you will probably just want to directly print the info yourself as you need it).

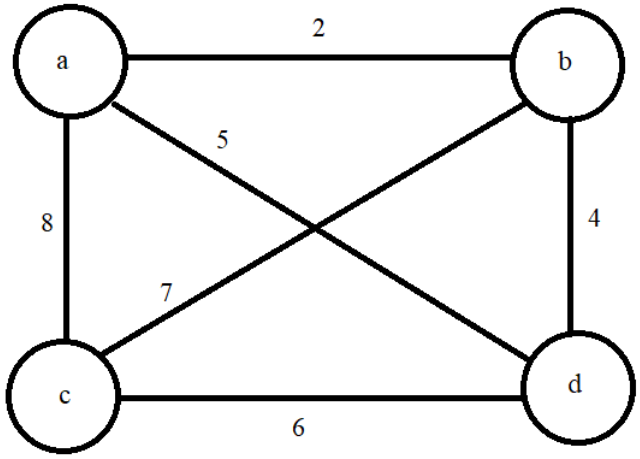
3.3 `getMap`, `getDist`, and `testMaps`

You have also been provided with a number of functions in `runTests.hpp` and `runTests.cpp` that create the Maps and test your code:

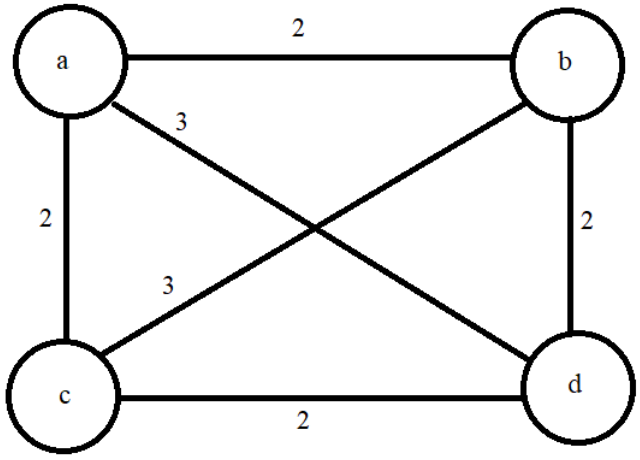
- `getMap`: this function takes in an integer 0-8 choosing the scenario to test, along with pointers to an empty adjacency list, adjacency matrix, and edge list. It creates the map for the associated scenario, then creates and fills the graph. It lastly outputs a string with info about the optimal tour for the scenario.

  The scenarios are:

  0. A small 4 vertex graph.



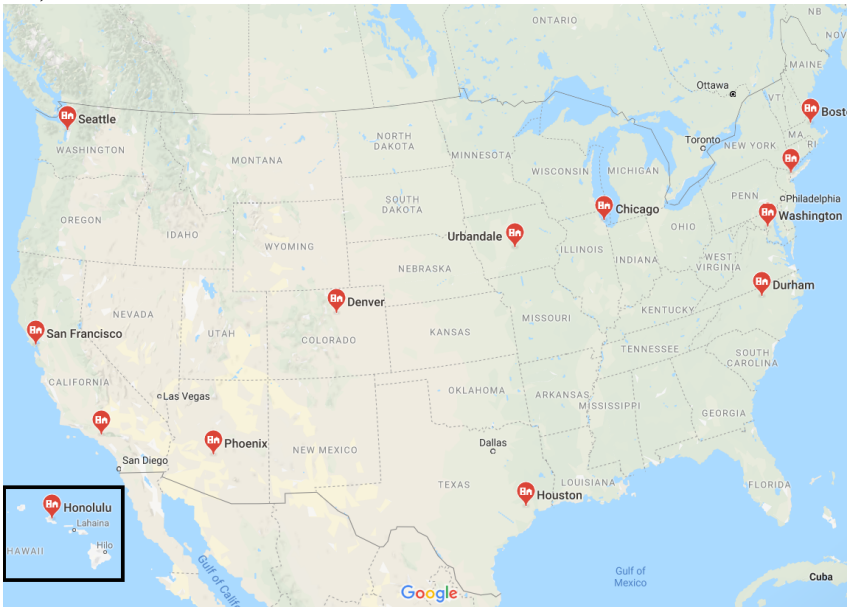
  1. Another small 4 vertex graph.



  2. A group of 7 cities in the US:
     New York City, Urbandale (my hometown), Chicago, Durham (Duke), Los Angeles (LA), Seattle, Washington DC
  3. A group of 9 cities in Europe.
     London, Paris, Madrid, Rome, Berlin, Istanbul, Moscow, Athens, Copenhagen

4. A larger group of 14 cities in the US.

   New York City, Urbandale (my hometown), Chicago, Durham (Duke), Los Angeles (LA), Seattle, Washington DC, Houston, Phoenix, Denver, San Francisco, Honolulu, Boston, Cleveland



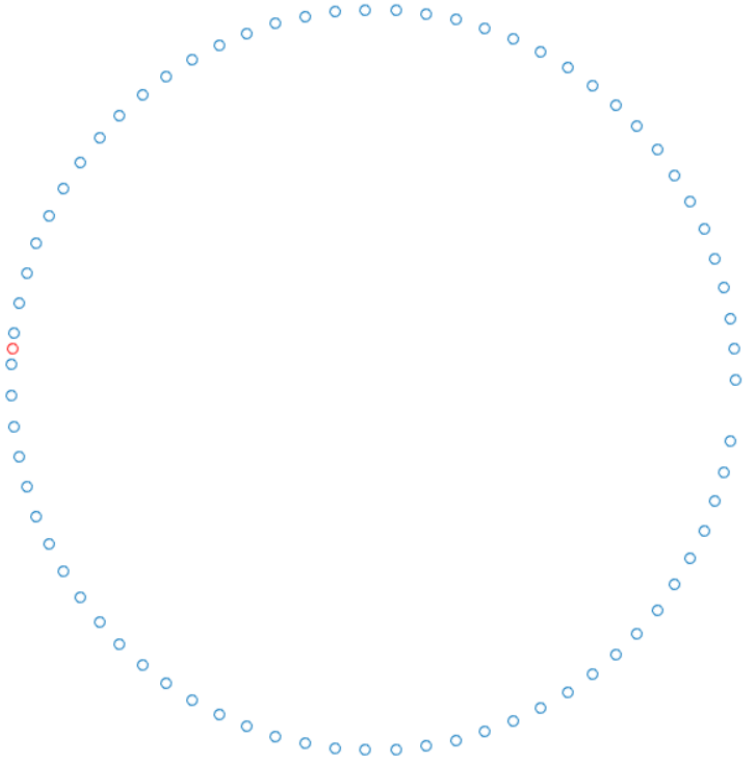
5. A larger group of 12 cities in Europe.

   London, Paris, Madrid, Rome, Berlin, Istanbul, Moscow, Athens, Copenhagen, Dublin, Warsaw, Kiev (Kyiv)

6. A larger group of 14 cities in Europe.

   London, Paris, Madrid, Rome, Berlin, Istanbul, Moscow, Athens, Copenhagen, Dublin, Warsaw, Kiev (Kyiv), St. Petersburg (Petrograd), Stockholm

7. This map consists of 75 points laid out evenly around a great circle. However, one point (marked in red) was moved to a location on the opposite side of the sphere leaving a gap in its place. This guarantees that there is unique MST for this Map (the tree follows the circle, but skips the larger gap left by the displaced vertex). For this map, we will have the TSP tour start at a vertex next to the gap. This means that when DFS is run to find the TSP tour, the approximation will find the optimal solution (the circumference of the Earth: 40030.173592 km).



8. The same great circle as the previous map. Here, however, the starting vertex of the tour will be the vertex at the misplaced point. When DFS is run now, we will follow the circle until we hit the gap, return to a vertex next to the start, then follow the other side of the circle, returning at last to the start when we again reach the gap. This approximation will be nearly as bad as possible, almost double the MST weight.

- `getDist`: this function takes in two sets of latitude and longitude coordinates (in degrees, with N and W as positive coordinates), and returns the great circle distance between them.

- `testMaps`: this function takes in a verbosity flag (to enable extensive printing if set to `True`). It will then test the results on all of the provided maps and will print messages indicating what failed in the case that a test was not passed. It uses the provided functions `getMSTWeight` and `getTourWeight` to calculate the weight of the MST and tour that your code found.

3.4 `prim`

You will need to implement the function `prim` in the file `prim.cpp`. This function will take as input the adjList and adjMat, and will output the vector of Edges in the MST. It must also ensure that at the end of execution, every vertex has been assigned the proper `mstNeighbors` values that encode the same MST edges that were returned.

3.4.1 Priority Queue

In the file `prim.cpp`, you have also been provided with placeholder functions `getMin` and `isEmpty`. These are for you to fill in if you choose to implement your own priority queue. You may choose one of three options for the priority queue:

- Since all of the graphs in this project are complete, it is most efficient to implement the priority queue as an unsorted array of cost values, and perform a linear search to find the vertex with the minimum cost. You have been provided with a vector to use in your `prim` function if you choose this representation. Note that with this approach the index of the vertex in the cost vector is its label. In order to maintain this property, you wont want to delete the min from the vector! Instead, you can use the visited values to skip any vertices that have already been visited (i.e., do not consider those visited vertices when searching for the min, and the `isEmpty` check will be true only once all vertices have been visited).

- The previous implementation has a small inefficiency because it doesn't actually remove values from the queue, resulting in a slower runtime (by a constant factor of roughly 2). If you wanted to improve on this implementation, you would need to track the vertex label associated with each cost, so you could safely remove the entry from the queue and still be able to associated the cost values with their respective vertices. You would also need to track where in the priority queue the vertex is located (so you can rapidly access and change the associated cost value). This alteration is not necessary, and is only a minor improvement for this project.

- There is a priority queue in the standard C++ library that you are welcome to use. Note that containers in C++ do not like to allow you to alter their contents, so changing the cost value of something in the queue will actually require re-inserting with the updated value, and using the visited values to ignore the older version when it eventually gets popped (you should ignore a popped vertex that is already visited). You will again need to track the vertex label associated with each cost in order to use this implementation.

3.5 `kruskal`

You will need to implement the function `kruskal` in the file `kruskal.cpp`. This function will take as input the adjList and (already sorted) edgeList, and will output the vector of Edges in the MST. It must also ensure that at the end of execution, every vertex has been assigned the proper `mstNeighbors` values that encode the same MST edges that were returned.

3.5.1 Disjoint Sets

To do this, you will need to implement and call the operations for disjoint sets (noticing that you have been provided with vectors to store the `pi` and `rank` values for each vertex). These functions are `find` and `union_by_rank`, and their placeholder declarations are already present for you in the file.

When you implement these disjoint set operations, it will be important for you to keep track of the type of the variables involved. Note that `find` and `union_by_rank` both expect the vertices to be represented as the integer labels instead of Vertex objects. Also remember that the `pi` pointers are represented as integer labels as well.

Important Requirement: You should use path compression for your implementation.

3.6 `tsp`

Your final task will be to implement a function that can trace the TSP tour using depth first search on the MST. Specifically, this function will be given the adjList and starting Vertex (label 0), and must return the tour as a vector of vertex labels.

You could implement this DFS using `prev` values to store the tour. But you would have to then extract the actual tour from those `prev` values afterwards. Note, however, that if you build the tour vector by appending (`push_back`) the label of each new vertex visited as you visit it, you will not need the `prev` values.

A note on implementation: you can use the stack in the standard C++ library, or you may also use recursion if you would like.

When consulting the list of neighbors for a vertex, it is important to remember that we are only interested in the edges of the MST. Each vertex has a `mstNeighbors` attribute that represents the list of neighbors in the MST. You should use `mstNeighbors` rather than the `neighbors` vector in this `tsp` function.

Finally, remember that the tour must end at the start vertex to complete the cycle.

4 Compilation and Testing

You have been provided with a functioning `Makefile`, and can compile with the command:

```
make
```

You can run the code with the command:

```
./runTests
```

You can set the verbosity flag to enable more comprehensive printing with the `-v` flag:

```
./runTests -v
```

5 Submission

You **must** submit your `prim.hpp`, `prim.cpp`, `kruskal.hpp`, `kruskal.cpp`, `tsp.hpp`, and `tsp.cpp` code online on Gradescope. You and your partner should only submit one version of your completed project, associate both partners with the submission, and indicate clearly the names of both partners at the top of all submitted files. Attribute help in the header of your `.cpp`.

6 Pair Programming

You are required to work in assigned pairs for this project.

- Your group should submit only one version of your final code. Have one partner upload the code and report through Gradescope and associate both partners with the submission. Make sure that both partner's names are clearly indicated at the top of all submitted files.
- When work is being done on this project, both partners are required to be physically present at the machine in question, with one partner typing ('driving') and the other partner watching ('navigating'). (You may work together remotely by sharing screens.)
- You should split your time equally between driving and navigating to ensure that both partners spend time coding.
- You are allowed to discuss this project with other students in the class, but the code you submit must be written by you and your partner alone.

7 Readable Code

You are expected to produce high quality code for this project, so the code you turn in must be well commented and readable. A reasonable user ought to be able to read through your provided code and be able to understand what it is you have done, and how your functions work.

The guidelines for style in this project:

- Your program file should have a header stating the name of the program, the author(s), and the date.
- All functions need a comment stating: the name of the function, what the function does, what the inputs are, and what the outputs are.

- Every major block of code should have a comment explaining what the block of code is doing. This need not be every line, and it is left to your discretion to determine how frequently you place these comments. However, you should have enough comments to clearly explain the behavior of your code.
- Please limit yourself to 80 characters in a line of code.