Math 590 Project 4: MST/TSP Fall 2019

Due: Friday, December 6

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 `project4.py` file.

Contents

1 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 *completely connected* 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 a 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 behind this approach is based on one important observation: the minimum spanning tree of a graph will always have less cost than a tour of the graph. 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 cost more, i.e., $|\sigma| > |M|$ where $|M|$ is the cost of the minimum spanning tree.

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 and once during the rewind. So this traversal will cost $2|M|$. Note that it is not a proper tour because we visit each vertex twice.

However, since every city is connected to every other city, we can traverse our cities in the order that DFS returns while skipping 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.

To compare the runtime of this approximation, I used brute force to compute the optimal solution for several of the smaller examples and recorded the time:

Map #	Description	Runtime
2	7 US cities	0.005 s
3	9 EU cities	0.092 s
5	12 EU cities	113.95 s

As you can see, only 12 cities took almost 2 minutes to run. I left the program for a 14 city set running for over an hour before killing it without reaching a solution. Meanwhile, the approximation code for finding the MST and the TSP tour combined run in less than a microsecond for each of these examples, and takes much less than a second for even the 75 city test cases. Now, these times represent only 3 data points. However, since we know the theoretical results (Prim and Kruskal are polynomial time while brute force is factorial), we can argue that we are seeing a speedup that we were expecting.

2 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, Edge, Priority Queue, and Map classes.

2.1 Vertex Class (in `p4map.py`)

The first of the fully functioning classes is the Vertex class. This class has 9 class attributes:

- rank: the rank (label) of the given vertex
- neigh: the list of the neighboring vertices
- mstN: the neighbor list for the MST
- visited: the visited flag for DFS
- cost: the cost of the edge into the tree
- prev: the previous vertex in the path
- pi: the parent vertex in the disjoint set
- height: the height in the disjoint set
- city: the string label of the city associated with the vertex

Along with these are three implemented member functions:

- `__init__`: this is the constructor function for the Vertex class. It requires an input rank for the vertex, and sets all of the attributes to have reasonable starting values. You will create a new Vertex with a call: `v = Vertex(rank)`.
- `__repr__`: this function is called whenever a Vertex is printed, i.e. when the call `print(v)` is made. It simply prints the rank of the vertex.
- `isEqual`: this takes in a second Vertex as an input, and compares the rank of the two vertices, returning True if they are equal rank (i.e., if they had the same label). This function can be called using: `v.isEqual(u)`.

The Vertex class also has an overloaded `less-than` (<) operator for use in the priority queue.

2.2 Edge Class (in `p4map.py`)

You have also been provided with a fully functioning Edge class with 2 class attributes:

- vertices: a 2 element list of 2 Vertex objects associated with this edge
- weight: the weight of this edge

The Edge class has 2 member functions:

- `__init__`: this is the constructor for the Edge class. It takes in two Vertex objects and a weight. A new Edge object can be created with the call `e = Edge(v1,v2,w)`.
- `__repr__`: this function is called when a Edge object is printed. It prints the vertex city names and the edge weight.

The Edge class has overloaded all of its comparison operators to compare based on the weight attribute. This allows for sorting a list of edges.

2.3 PriorityQueue Class (in `p4priorityQueue.py`)

This provided PriorityQueue class will be used as the priority queue for Prim's algorithm. Because the Traveling Salesman Problem guarantees that the given graph is fully connected, implementing the priority queue as an unsorted list will provide the best runtime. The reason we are not using a built-in priority queue is because Python's `heapq` does not easily allow for a decrease-key operation, which is necessary for Prim's algorithm.

The fully functioning PriorityQueue class has 1 class attribute:

- array: the array storing the values in the queue

The PriorityQueue class has 5 member functions:

- `__init__`: this is the constructor for the PriorityQueue class. It takes an array of objects to insert into the queue. A new PriorityQueue object can be created with the call `Q = PriorityQueue(array)`.
- `__repr__`: this function is called when a PriorityQueue object is printed. It prints the queue.
- `isEmpty`: this function returns True if the PriorityQueue is empty and False otherwise.
- `insert`: this function will insert a new element into the PriorityQueue.
- `deleteMin`: this function will find the minimum element from the queue, remove it from the queue, and return it.

Note that our PriorityQueue does not need a decrease-key function. The keys are only ever accessed as we are scanning the array for the minimum element. This means that if a vertex has had its cost updated, then the decrease-key operation has already occurred.

2.4 Map Class (in `p4map.py`)

You have also been provided with a fully functioning Map class with 9 class attributes:

- adjMat: the adjacency matrix for the map
- cities: the list of city names
- adjList: the adjacency list for the map (a list of Vertex objects)
- edgeList: the list of Edge objects for the map
- start: the starting vertex of the TSP tour
- MSTalg: either `prim` or `kruskal`
- mst: the list of edges in the MST
- tour: the TSP tour (a list of vertex ranks)
- optTour: a string that will display the optimal tour (when applicable)

The Map class has 8 member functions:

- `__init__`: this is the constructor for the Map class. It takes a map number and the MSTalg string. A new Map object can be created with the call `m = Map(mapNum,MSTalg)`.
- `__repr__`: this function is called when a Map object is printed. It prints the edges in the MST, the weight of the MST, the TSP tour, and the weight of the TSP tour.
- `printList`: this function prints the adjacency list.
- `printMat`: this function prints the adjacency matrix.
- `printEdges`: this function will print all of the edges in the graph.
- `getMST`: this function calls your written code for Prim's or Kruskal's algorithm, and then ensures that the Map class attributes are updated accordingly.
- `getTSP`: this function will call your TSP code to acquire the TSP tour for the given Map and MST.
- `clearMap`: this function resets the Map to its initial state.

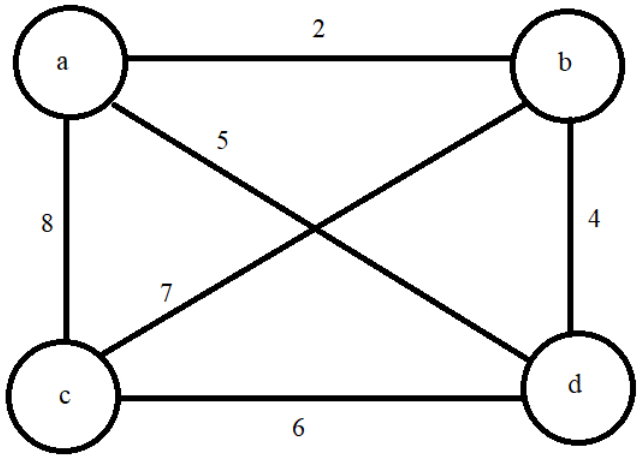
2.5 `getMap`, `getDist`, and `testMaps`

You have also been provided with three functions that create the Maps and test your code:

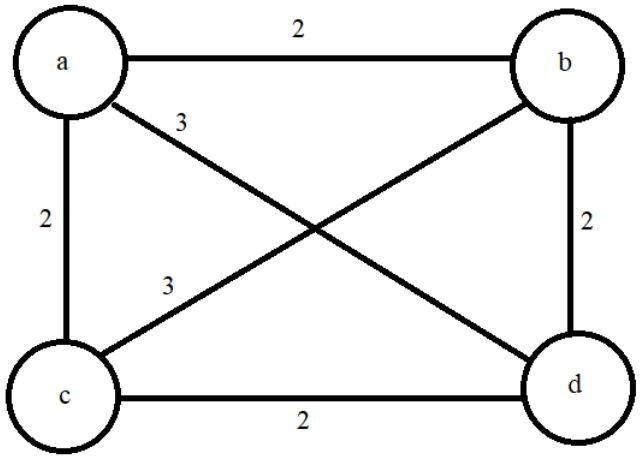
- `getMap` (in `p4map.py`): this function takes in an integer 0-8 and outputs the adjacency matrix, city list, and optimal tour info for the given map.

  The scenarios are:

  0. A small 4 vertex graph.



  1. Another small 4 vertex graph.



  2. A group of 7 cities in the US:

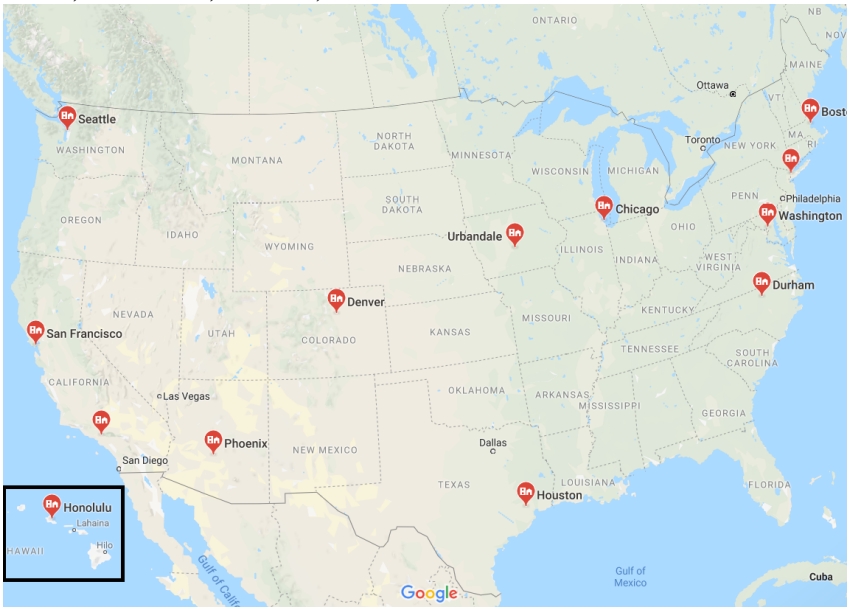
     New York City, Urbandale (my hometown in the state of Iowa), Chicago, Durham (Duke), Los Angeles (LA), Seattle, Washington DC

  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 in the state of Iowa), Chicago, Durham (Duke), Los Angeles (LA), Seattle, Washington DC, Houston, Phoenix, Denver, San Francisco, Honolulu, Boston, Cleveland



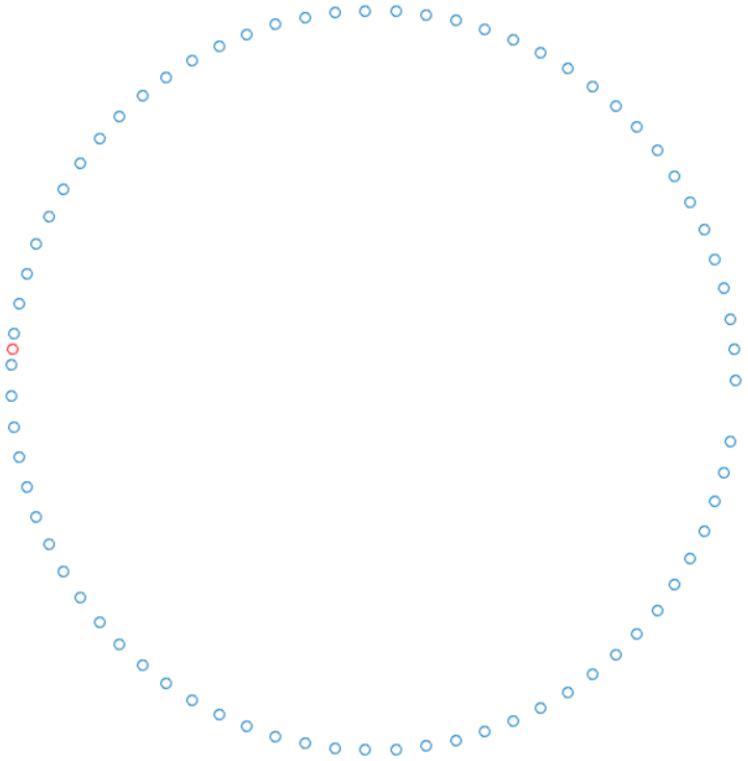
5. A larger group of 12 cities in Europe.

   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 (a twig following the circle that skips the gap). 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 (in p4map.py)`: 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 (in p4tests.py)`: this function takes in which MST algorithm you want to test (`prim` or `kruskal`), and 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.

3 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 (along with the disjoint set functions).
3. Implement a DFS function to find the tour, once you have found the MST.

3.1 `prim`

You will need to implement the function `prim`. This function will take as input the adjList and adjMat of the Map class, and it has no outputs. It must instead ensure that at the end of execution, every vertex has been assigned the proper `vertex.prev` values.

Several things to note:

- You should start by setting the initial `cost`, `prev`, and `visited` values for each vertex in the adjacency list.
- You should use the `visited` attributes for the vertices during execution to make sure you don't try to update the `prev` value of a vertex you have already visited.

3.2 `kruskal`

You will need to implement the function `kruskal`. This function will take as input the adjList and (already sorted) edgeList of the Map class, and it will return a list of edges that are in the MST.

To do this, you will need to implement and call the three operations for disjoint sets (noticing that each Vertex object has a `pi` and `height` attribute for this purpose). These functions are `makeset`, `find`, and `union`, and their declarations are further down in the file.

When you implement these disjoint set operations, it will be important to use the `u.isEqual(v)` method for comparing two vertices.

You should use path compression for your implementation.

It is strongly suggested that you ensure your MST algorithms are working correctly before you continue.

3.3 `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 Map's adjList and start attributes, and must return the tour array of vertex ranks.

You will want to use the `vertex.visited` attribute during your DFS, which will mean that you want to first iterate through the list of vertices as flag them all as unvisited.

Since the Map class is already using the `vertex.prev` values for the MST, you will want to leave those values unchanged. Note, however, that if you build the tour list by appending the rank of each new vertex visited, you will not need the `prev` values.

Recall that DFS uses a stack to track the next unvisited vertex. In python, lists work **exactly** like stacks. If you define a list `s = []`, then you can use `s.append` as the push operation, and `s.pop()` as the pop operation.

When finding 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 `vertex.mstN` attribute that represents the list of neighbors in the MST. You should use `mstN` rather than the `neigh` list.

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

Note: the `project4.py` file contains a main function that runs the testing code with verbosity set to false (for minimal printing). If you would like to turn verbosity on, simply set `verb = True` in this function. Your main function will not be graded, so feel free to edit it as you wish.

4 Submission

You **must** submit your `project4.py` code online on Sakai. If you worked with a partner, only submit one version of your completed project and indicate clearly the names and NetIDs of both partners.

5 Pair Programming

You are allowed to work in pairs for this project. If you elect to work with a partner:

- You should submit only one version of your final code. Have one partner upload the code on their Sakai site. Make sure that both partner's names are clearly indicated at the top of the code.
- 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 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.

6 Style Points

Part of your grade for this project will be 'style points'. The idea here is that the code you turn in must be well commented and readable. A reasonable user aught 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. This means that a grader should be able to read over your code and tell that your algorithms are implemented correctly. The guidelines for these 'style points':

- 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. In python, you can use the symbol \ to indicate a line break/continuation.