CSE 332: Data Structures and Parallelism

Exercise 9 - Spec

Exercise 9 Spec (25su)

The objectives of this exercise are

- **Implement a minimum-spanning-tree algorithm**
- **Use minimum spanning trees combined with breadth-first-search to solve a clustering problem**

Overview

This exercise consists of the following parts:

1. Use given data in order to construct a graph using an adjacency list representation.
2. Implement Prim's algorithm for finding minimum spanning trees.
3. Use the result of Prim's algorithm to solve a clustering problem by deleting edges from the MST, then checking which nodes are connected.

Motivating Application: Unsupervised Learning

There are two primary categories of machine learning tactics. Supervised learning involves using labeled data to train a model that can later apply labels to new data. For example, suppose I had a bunch of photos of Labradors (a dog breed) that I wanted to separate into black labs, yellow labs, and chocolate labs (different varieties of Labradors different only by coat color). If I manually pre-labelled these photos with the correct variety then I could use them to train a supervised learning algorithm to label future photos with the correct variety of Labrador.

An unsupervised model, on the other hand, does not require any pre-labelling of the training data. Instead, unsupervised models attempt to discern patterns in the training data by looking only at the data itself. For example, we might be able to plot the photos of Labradors as points in some cartesian space (as a simple example, we might have a 3-dimensional point per picture which represents the average RGB values of the image). With these points, an unsupervised model might identify clusters of points that are similar to one another and conclude that points in the same cluster must be the same Labrador variety. The advantage to these "clustering" algorithms is that we do not need to take the effort to pre-label, one disadvantage is that it does not tell us which cluster represents which variety.

Problem Statement

The most commonly used clustering algorithm is called k-means clustering. Given a collection of points and a number of clusters k, the k-means clustering algorithm will split those points into k collections such that points which share a cluster are those which are nearest to the same reference point. For this assignment we will be doing a different form of clustering that I will call k-margin clustering. For this algorithm, we will be given a collection of items and an integer k. We will then split the items into k collections in such a way to maximize the “gap” between the closest two clusters. We define the “gap” size between two clusters to be the closest pair of items between them. In other words, we want to split our collection into k subsets such that we have maximized the closest pair of points across those subsets.

This task may seem daunting at first, but minimum spanning trees will help! To begin with, we will represent our collection of items as a 2-d array, such that cell `i,j` of this array represents the distance from item `i` to item `j`. Effectively, this 2-d array can be considered as an undirected, weighted, complete graph. So we now have a graph where all of the items are nodes and the weights of the edges between nodes represent their distance from each other. From here, we will calculate a minimum spanning tree of this graph. This will be helpful because:

1. If we consider any cluster in the graph, the edge which connects that cluster to its closest neighboring cluster must be an edge in a minimum spanning tree of the graph. Suppose that the cost of the clustering is the weight of edge (x, y). This follows from the cut theorem of MSTs. We will define a cut in the graph such that our cluster is one side of the cut and all other nodes are on the other side. In this case, any edge going from this cluster to another cluster will cross the cut. The closest pair will then be the lightest edge which crosses the cut, and therefore is part of a minimum spanning tree!
2. Being a spanning tree, a minimum spanning tree is connected and acyclic and contains $n - 1$ edges. If any edge is removed then the graph is no longer connected, instead it will have two separate components. If any two edges are removed then it will have three separate components. So if we remove k-1 edges then we will have k separate components.

Combining these two observations above, we get a clustering algorithm.

- Our input will be an n x n array of doubles, representing the pairwise distances of all our items (which we’ll just consider to be ints 0 through n-1, so the indices are the items) and an int k for the number of clusters
- We will construct a graph using this n x n array.
- Next we’ll construct a minimum spanning tree on that graph
- Then we remove the k-1 heaviest edges from the MST(the weight of the last edge removed will be the distance between the closest pair of clusters)
- Finally we identify which items are in which cluster by checking which items are connected to each other using the remaining edges of the MST (e.g. by using a breadth-first search).

Your task for this assignment is to implement that algorithm above.

Input Format: For this assignment, input will be encoded in txt files. Supposing that each test consists of n items, the files will contain n+2 lines as follows:

- The first line contains the value of k, i.e. the number of clusters to break the items into
- The next line contains the value n (the number of items)
- The remaining n lines contain a space-separated list of doubles which indicate the distances between each pair of items

For example, consider a file with its contents as shown below.

```
3
5
0 18 21 23 5
18 0 54 30 31
21 54 0 15 32
23 30 15 0 15
5 31 32 15 0
```

This file indicates that we will be splitting 5 items into k clusters. When doing so, item 0 is distance 18 from item 1, item 2 is distance 15 away from item 3, etc.

In this case the optimal clustering would be for the first cluster to contain items 0 and 4, the second to contain items 2 and 3, and the third cluster to contain item 1. To see the cost of this clustering we look at the closest pair of items for each pair of clusters, and then save the smallest. So in this case the cost of the clustering would be 15. The following explains why:

- The distance between the first cluster and the second is 15 because the closest pair of points that crosses these clusters is items 4 and 3 which have distance 15.
- The distance between the first cluster and the third is 18 because the closest pair of points that crosses these clusters is items 0 and 1 which have distance 18.
- The distance between the second cluster and the third is 30 because the closest pair of points that crosses these clusters is items 1 and 3 which have distance 30.
- The overall cost of the clustering is the smallest of these distances, and so it is 15.

Implementation Guidelines

Your implementations in this assignment must follow these guidelines

- You may not add any import statements beyond those that are already present (except for where expressly permitted in this spec). This course is about learning the mechanics of various data structures so that we know how to use them effectively, choose among them, and modify them as needed. Java has built-in implementations of many data structures that we will discuss, in general you should not use them.
- Do not have any package declarations in the code that you submit. The Gradescope autograder may not be able to run your code if it does.
- Remove all print statements from your code before submitting. These could interfere with the Gradescope autograder (mostly because printing consumes a lot of computing time).
- Your code will be evaluated by the gradescope autograder to assess correctness. It will be evaluated by a human to verify running time. Please write your code to be readable. This means adding comments to your methods and removing unused code (including “commented out” code).

For this assignment, there **is partial credit for passing only a few test cases**. For example, if you failed on any test case on `Cluster Cost`, you will receive partial credits corresponding to that test case. We’ve provided all 10 files that we will be testing on. Each file is worth 5 points (2 for Cluster Cost and 3 for Cluster). Therefore, there are 50 points on autograder, plus 20 points on manual inspection. Meanwhile, you will be able to see the details on the testcases you failed.

Provided Code

Several java classes have been provided for you in this zip file. Here is a description of each.

- `Client`
  - This class contains the main method, which does the following:
    - Reads the input file named in the field testFileName
    - Parses the file and converts all the information into a 2-d array along with an integer variable k.
    - Calls the constructor that you will implement, providing the array and k as input
    - Prints the cost of the clustering found.
  - *Do not submit this file*
- `WeightedEdge`
  - These objects will be used to represent an edge in the graph (if you choose to use an adjacency list representation, which I think is easier). Its primary role is just to encapsulate 3 fields:
    - `source`: the source node of the edge
    - `destination`: the destination node of the edge
    - `weight`: the weight of the edge
  - You are not required to use this class, but I found it very helpful!
  - *Do not submit this file*
- `Clusterer`
  - This is the file that you will be modifying for this assignment. Your goal is to implement the constructor, which must calculate the clusters for the given 2-d array of distances and integer k. To guide you through this:
    - In part 1 you will convert the 2-d array into an adjacency list representation (you're welcome to use the array itself as an adjacency matrix if you prefer, but I find the adjacency list easier to work with)
    - In part 2 you will implement Prim's algorithm to compute a MST for this graph
    - In part 3 you will remove edges from the minimum spanning tree, then check which nodes are connected to each other to identify the clusters.
  - ***You will submit this file to Gradescope***
- Various text files (test cases that we will be running on)
  - `specExample.txt`
    - The file described above, The cost of the clustering should be 15.0
  - `2clusters500points.txt`
    - The cost of this clustering should be 1703083.0
  - `3clusters10points.txt`
    - The cost of this clustering should be 119.0
  - `5clusters20points.txt`
    - The cost of this clustering should be 620.0
  - `5clusters100points.txt`
    - The cost of this clustering should be 37821.0

    - `10clusters10points.txt`
        - The cost of this clustering should be 22.0
    - `13clusters50points.txt`
        - The cost of this clustering should be 4248.0
    - `20clusters1000points.txt`
        - The cost of this clustering should be 4249168.0
    - `11clusters121points.txt`
        - The cost of this clustering should be 39360.0
    - `35clusters35points.txt`
        - The cost of this clustering should be 57.0

Not Provided Code that you might decide to use:

- `BinaryMinHeap`
    - You implemented this file as part of Exercise 2. It implements a priority queue, which is one of the data structures you need for Prim's Algorithm. See the notes at the bottom of Part 2.
    - ***If you wish to use this code, you will submit this file to Gradescope.***
- `MyPriorityQueue`
    - This was a provided file in Exercise 2 as an interface for the `BinaryMinHeap` and `BinaryMaxHeap` you implemented.
    - If you decide to use `BinaryMinHeap` include this code in your local directory.
    - *Do not submit this file* (regardless of whether you're using your own priority queue or Java's).
- `Pair`
    - This was a provided file in Exercise 2 as the object you will be using in the BinaryMinHeap.
    - You may wish to use this file so that you can use `BinaryMinHeap`.
    - *Do not submit this file* (regardless of whether you're using your own priority queue or Java's).

Part 1: Making the graph

For this part you will build a graph out of the contents of the test files. The main method in Client reads the text file, then provides its contents as arguments to your constructor. These arguments are:

- `distances`: a 2-d array of doubles with the property that cell `i,j` contains the distance between items `i` and `j`. You may assume that all distances are positive, that cell `i,j` matches cell `j,i`, and that every cell `i,i` is 0.
- `k`: the number of clusters to break our data into

There are several ways to implement the constructor, but the way I recommend is to build an adjacency list representation of the graph by making a list of lists of `WeightedEdge` objects. In this case we have a complete graph, meaning that an adjacency list does not have any space benefit over an adjacency matrix, nor will it have any time benefit. It's also not going to be asymptotically worse by either metric, though. In my opinion, an adjacency list is easier to use because it allows for more readable code. If you prefer to use an adjacency matrix, though, I support your decision!

Part 2: Prim's algorithm

For this part you will implement Prim's algorithm for finding a minimum spanning tree. I recommend following the pseudocode presented in class/section. You're welcome to use any list or priority queue data structures that you would like from java.util. I recommend that you use this algorithm to produce a second adjacency list to represent just the minimum spanning tree. That is, you will produce a new graph with the same nodes as the original, but there will be only the V-1 edges included in the minimum spanning tree. Keep in mind that this graph should be undirected.

Note that the Prim's algorithm shown in section and lecture uses the function PriorityQueue.updatePriority(node) which **<u>is not</u>** a java.util function. We recommend you use one of the 3 following ways to get around this issue:

1. Add edges to the priority queue instead of nodes; but note in this case, when you remove an edge from the priority queue, you'll need to check if that edge still connects not-yet-connected vertices (i.e., that it doesn't create a cycle).
2. Allow duplicates of vertices to be added to the priority queue (this is actually functionally equivalent to 1); but note in this case, when you remove a vertex, you'll need to make sure you haven't already processed that vertex.
3. Import/use the Priority queue you implemented in Exercise 2. You implemented updatePriority, so you can keep your code more similar to the pseudocode. In this case, you'll need to include file `BinaryMinHeap.java` when you upload to Gradescope. Of course, any bugs in your Exercise 2 code could lead to errors in this project. If you decide to use this approach, we will re-run the exercise 2 tests to do a basic check of your priority queue, but be on the lookout for subtle/uncaught bugs in that code that have an impact on this algorithm. If you decide to upload `BinaryMinHeap.java` and it fails the exercise 2 tests, then the MST tests **will not** be run. **We strongly recommend that you**

upload the `BinaryMinHeap.java` to Gradescope for a sanity check even before you start implementing the actual logic in `Clusterer.java`.

Your implementation of Prim's algorithm must obey the O(E log V) bound described in lecture. Properly implemented, any of the above options can meet this bound because Theta(log V) = Theta(log E) (since E <= V^2 and log (V^2) = 2 log V by log rules).
Note that calling `remove(Object o)` on the Java.util PriorityQueue is unlikely to let you meet this bound. It runs in O(n) time when it has n elements (unlike your priority queue, which can do that operation faster due to maintaining the itemToIndex dictionary).

Part 3: Clustering

Now we will use the minimum spanning tree to produce the clusters. The first step is to identify the k-1 largest edges used in the minimum spanning tree. Then, remove them! When doing these steps, remember that every edge appears twice in an adjacency list representation of an undirected graph. Use the weight of the lowest-weight edge you removed as the value of the `cost` field of the object.

Once you've removed these k-1 edges you can identify which integers belong to which cluster. To do this, implement a breadth first search. *Do not implement depth first search*. You can start your breadth-first search from an arbitrary node, then all nodes visited from that one will belong to the same cluster. Then repeat a breadth first search on all nodes not yet visited by the previous searches. You can then assign the list of lists of connected nodes as the `cluster` field.

This is the last thing to implement for this assignment! Once you've done this, verify correctness by trying out each of the txt files provided to make sure the cost is correct. Because there will be several clusterings of the same cost, the easiest way to verify your cluster assignment is to check every item with every item in all the other clusters, making sure that the distance never exceeds the cost. We don't provide the code to do this, but the autograder will be checking this.

The big-O running time of the clustering portion of the code (excluding the Prim's call) should be at worst O(E log V). That is, the overall running time should be dominated by the time to call Prim's.
The implementation we describe here can be done in time O(V log V) (since there are only V-1 edges in a minimum spanning tree), but we'll only check that you hit O(E log V) or better.