IMPLEMENTING KRUSKAL

KARL MUTCH

UCB INTRODUCTION TO WEB ALGORITHMS, 2010.

GOAL:

This weeks' project was chosen as a way for me to become familiar with Minimum-Spanning Tree algorithms.

INTENDED RESULTS:

The goal is to implement a Kruskal class and to test using the existing MSTSingleLinkAlgorithm implementation along side the existing prim's based MST class implementation as a way of validating the results.

WEEK 3 : TASKS

The first tasks involved creating a class Kruskal within the existing iweb2.ch4.clustering.hierarchical package.

The second tasks calls for a test harness to be added to the MSTSingleLinkAlgorithm so that both Prim's and Kruskal's results can be compared to ensure their outputs are equivalent.

WEEK 2 : DIARY

First a brief summary of the pseudo code that is involved in this algorithm might help during the diary.

```
function Kruskal(G = <N, A>: graph; length: $A \rightarrow R^+$): set of edges
  Define an elementary cluster C(v) ← {v}.
  Initialize a priority queue Q to contain all edges in G, using the weights as keys.
  Define a forest T ← Ø       //T will ultimately contain the edges of the MST
   // n is total number of vertices
  while T has fewer than n-1 edges do
    // edge u,v is the minimum weighted route from u to v
    (u,v) ← Q.removeMin()
    // prevent cycles in T. add u,v only if T does not already contain a path between u and v.
    // the vertices has been added to the tree.
    Let C(v) be the cluster containing v, and let C(u) be the cluster containing u.
    if C(v) ≠ C(u) then
      Add edge (v,u) to T.
      Merge C(v) and C(u) into one cluster, that is, union C(v) and C(u).
  return tree T
```

When implementing Kruskal some implementation choices were made to speed up the implementation.

PriorityQueue

The algorithm describes a priority queue into which edges are placed and retrieved in sorted order. The priority queue has the property that items are actually removed from the structure as they are processed.

One important property of the traditional algorithm is that the time complexity depends upon a single comparison sort being used once giving it O(*E log E*) performance which is a guarantee of the java utils Collections package. So while PriorityQueue's do have reduced time I decided to stick with the algorithm's description faithfully.

I broke this out in this implementation to illustrate this aspect of the algorithm.

The PriorityQueue, a min-priority-queue, however does have some very useful properties for these types of greedy algorithms namely that they can be used to implement data structures such as Binary heaps. There is also a thread safe version PriorityBlockedQueue that would prove useful in other situations.

DisjointSet

A disjoint set implementation is included that tracks which vertices are in which component. The implementation done in this example uses a forest approach and the Union is done by combining trees. Path compression, and union by rank optimizations are not yet used and represent an item for future work.

Edge

In order to support natural ordering for Edge's this class inherits a comparable interface and implements this for the sorting step of the algorithm.

Marshaling

In order to implement this algorithm processing was introduced to convert Adjacency Matrices to and from Adjacency Lists. Not doing this distorts the nature of the algorithm around the different data structure.

The test harness implemented is done within the MSTSingleLinkAlgorithm class and the cluster method. This method was enhanced by adding various checks and diagnostic printing to allow the algorithm to be tested in-situ.

Here is the code changed in the cluster method.

```java
// Compute the clusters for a data set previously loaded and
// then return the Dendrogram for the results, c.f. http://en.wikipedia.org/wiki/Dendrogram
public Dendrogram cluster() {

    // First run the Prim's method for calculating the MST
    m = (new MST()).buildMST(a);

    // Now run the Kruskal method for the same
    double [][] kruskalResults = (new Kruskal()).buildMST(a);

    // Do a very basic dimension sanity test for the returned matrix
    if (m.length != kruskalResults.length) {
        throw new RuntimeException("Mismatched length");
    }

    // iterate the results checking each element in turn and printing diagnostics
    // in the event an error is uncovered.
    boolean mismatched = false;
    for (int i = 0 ; i != m.length ; ++i ) {
        for (int j = 0 ; j != m.length ; ++j ) {
            if (m[i][j] != kruskalResults[i][j]) {
                System.out.println("Mismatched at " + i + "," + j + " " + m[i][j] + " " +
kruskalResults[i][j]);
```

```
                mismatched = true;
            }
        }
    }

    // Continue with existing logic

    Dendrogram dnd = new Dendrogram("Distance");
```

The above code supplies enough information for us to validate the implementation.

The algorithm implementation is next. It includes all of the supporting classes for Edges and Disjoint sets etc as inner classes. This might seem a little unruly but it does allow you to cut and paste the entire thing with no dependencies.

```
package iweb2.ch4.clustering.hierarchical;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;


/**
 * This class uses Kruskals's algorithm to build a Minimal Spanning Tree (MST).
 *
 */
public class Kruskal {

    // Inner class for storing Edges for our algorithm
    public class Edge implements Comparable<Edge>
    {
        public int from;
        public int to;
        public double cost;

        public Edge(int from, int to, double cost)
        {
            this.from = from;
            this.to = to;
            this.cost = cost;
        }

        // This method is used by the sort to determine natural ordering
        // based upon a function of the cost
        public int compareTo(Edge other)
        {

            if (cost < other.cost)
                return(-1);

            if (cost == other.cost && from == other.from && to == other.to)
                return(0);

            if (cost == other.cost)
                return(-1);

            if (cost > other.cost)
                return(1);

            return(0);
        }
    }

    // Inner class for handling the notion of Disjoint sets supporting
    // a find and a union method
    public class DisjointSet<E> {
```

```java
         // Somewhere to store the forest while we work.
        private ArrayList<ArrayList<E>> set = new ArrayList<ArrayList<E>>();

        /**
         * Modifier to allow arrays to be supplied to our set
         *
         * @param items An array containing items for the forest
         */
        public void createSubsets(E[] items){
            createSubsets(Arrays.asList(items));
        }

        /**
         * Modifier to allow collections of items to be inserted into
         * our forest.
         *
         * @param items Items to be inserted into the forest, 1 tree per item
         */
        public void createSubsets(Collection<E> items){
            for (E item : items)
            {
                ArrayList<E> subset = new ArrayList<E>();
                subset.add(item);
                set.add(subset);
            }
        }

        /**
         * Disjoint set union method
         *
         * @param setA First set to be merged to
         * @param setB Second set to be merged into the first
         */
        public void union(int setA, int setB){
            set.get(setA).addAll(set.get(setB));
            set.remove(setB);
        }

        /**
         * Accessor method to locate a specified item.  This method could optimize paths
         * but this is not yet implement.
         *
         * @param item The item to be located in the component of vertices
         *
         * @return -1 if not found otherwise the item is returned
         */
        public int find(E item)
        {
            for (int i = 0; i < set.size(); ++i)
            {
                if(set.get(i).contains(item))
                {
                    return i;
                }
            }
            return -1;        // Indicates not found condition
        }
    }

    /**
     * The main implementation loop for the algorithm.
     *
     * @param nodes An array containing the list of known nodes, helper.
     * @param edges An adjacency list of the edges in an unordered collection
     *
     * @return An adjacency list representation of the resulting Minimum-Spanning Tree
     */
    public ArrayList<Edge> getMST(Integer [] nodes, ArrayList<Edge> edges)
    {
        // Comparison sort complying with the naive description of Kruskal's algo
         java.util.Collections.sort(edges);

         // The output edges data structure
         ArrayList<Edge> adjacencyList = new ArrayList<Edge>();

         // Instantiate a Disjoint set to hold the forest
         DisjointSet<Integer> nodeset = new DisjointSet<Integer>();
         // makeSet step
         nodeset.createSubsets(nodes);
         // Iterate edges that were input
         for(Edge e : edges)
         {
             // Check for cycles
             if(nodeset.find(e.from) != nodeset.find(e.to))
             {
                 // Process new edge
                 nodeset.union(nodeset.find(e.from), nodeset.find(e.to));
                 adjacencyList.add(e);
             }
         }
         // Return the adjacency list output from the algorithm
         return (adjacencyList);
    }

    public double[][] buildMST(double[][] adjM)
    {
         // Convert the adjacency matrix into an adjacency list

         // Define an array list that will contain the edges we will process
         ArrayList<Edge> adjacencyList = new ArrayList<Edge>();

         // Define a sorted unique list of known nodes.  This has the property
         // of maintaining a unique collection of all node identifiers without
         // local code to handle duplicates etc.  It also pre-sorts the results
         // to make the Kruskals implementation run a little quicker without
         // violating its naive implementation.  It is in effect a rather
         // limited form of a priority queue.
         Set<Integer> uniqNodes = new TreeSet<Integer>();

         // Iterate the Adjacency matrix and flatten it into a List
         for (int i = 0 ; i != adjM.length ; ++i ) {
             if (adjM.length != adjM[i].length) {
                 throw new RuntimeException("An adjacency matrix by definition should be 
regular");
             }
             for (int j = 0 ; j != adjM[i].length ; ++j) {
                 if (0 > adjM[i][j]) {
                     // We have detected nodes that are not connected
                     continue;
                 }

                 // Add the extant edge to our adjacency list
                 adjacencyList.add(new Edge(i, j, adjM[i][j]));

                 // Add nodes to the node helper list
                 uniqNodes.add(i);
                 uniqNodes.add(j);
             }
         }

         // Supply the adjacency list to the kruskal class and have it generate
         // the MST.  We do the toArray to flatten the set to a vector
         adjacencyList = getMST(uniqNodes.toArray(new Integer[0]), adjacencyList);

         // Convert back into an adjacency matrix by first initializing an Empty graph
         double[][] adjacencyMatrix = new double[adjM.length][adjM.length];
         for (double[] row : adjacencyMatrix) {
             Arrays.fill(row, -1);
         }

         // Populate the adjacency matrix from the adjacency list
        // by iterating the adjacency list.
        Iterator<Edge> anEdge = adjacencyList.iterator();
        while (anEdge.hasNext()) {
            // for each edge populate the matrix on both sides of the pivot used
            // for matrix inversion
            Edge edge = anEdge.next();
            adjacencyMatrix[edge.from][edge.to] = edge.cost;
            adjacencyMatrix[edge.to][edge.from] = edge.cost;
        }

        // Now return the results in a format used by the iWeb2 suite
        return adjacencyMatrix;
    }
};
```

When running this code the absence of output indicates success. That is, both the Prims and the Kruskal produce the same adjacency matrix. For example:

```
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh % import *;
bsh % SFDataset ds = SFData.createDataset();
From file: C:/Devel/iWeb2/data/ch04/clusteringSF.dat
Using attribute names: [Age, IncomeRange, Education, Skills, Social, isPaid]
Loaded 20 data points.
bsh % DataPoint[] dps = ds.getData();
bsh % double[][] adjMatrix = ds.getAdjacencyMatrix();
bsh % MSTSingleLinkAlgorithm sla2 = new MSTSingleLinkAlgorithm(dps,adjMatrix);
bsh % Dendrogram dendroSLA2 = sla2.cluster();
Setting cluster level: 2
Setting cluster level: 3
Setting cluster level: 4
Setting cluster level: 4
Setting cluster level: 5
Setting cluster level: 6
Setting cluster level: 6
Setting cluster level: 6
Setting cluster level: 7
bsh %
```

Any errors would appear here

LESSONS LEARNED

In this example we made use of DisjointSet forests surprisingly they were first described in 1964 but it not until 1983 that Tarjan produced the path compression method, Tarjan, R. E. 1983.
Data structures and network algorithms, CBMS-NSF Regional Conference Series in Applied Mathematics. The point here being that even with seemingly simple algorithms there exist opportunities for optimization. For further information the following might be useful,
http://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf

More sophisticated algorithms do exist to greatly reduce the time complexity of this function, c.f. Chazelle, B. 2000. A minimum spanning tree algorithm with inverse-Ackermann type complexity. J. ACM 47, 6 (Nov. 2000), 1028–1047, and, Pettie, S. and Ramachandran, V. 2002. An optimal minimum spanning tree algorithm. J. ACM 49, 1 (Jan. 2002), 16–34.

When I examined the code for the Prims implementation it becomes apparent that the decision about the PriorityQueue implementation does matter for that algorithm because it has a tendency to bias depending upon the graph complexity. As the number of vetices increase over the number of edges it is better to use the ArrayList style of implementation and in the sparse direction the binary heap style would be better.

FUTURE WORK

Implement the path compression and the union optimizations.