Spanning Trees

Daniel Woldegiorgis

Magnus Herweyer

Last Updated: May 27 2019

1 Introduction

A tree is an acyclic connected Graph, where every edge is a bridge. While trees have many interesting properties, in this paper, we are going to be discussing spanning trees. A Spanning Tree of G is a subgraph T of G, where T is a tree that contains every vertex of G. Every connected graph contains a spanning tree. If we associate weights with the edges of our graphs, we can begin to ask more specific questions about Spanning Trees, such as finding one with minimum or maximum weight. Whilst there are popular algorithms such as Kruskal and Prim that produce Minimum Spanning Trees, we will be explaining and giving an implementation of lesser known Boruvka's Algorithm in this paper. We will also be giving an implementation that produces the total number of spanning trees given any graph using the Matrix Tree Theorem.

While not as common, we also discuss the ways of finding the maximum weight spanning tree in a graph. For ease of testing these algorithms, we propose using the following online IDEs: `https://repl.it/languages/python3` and `https://repl.it/languages/java`.

2 Boruvka's Algorithm

A pseudocode for Boruvka's algorithm is provided below. Wikipedia Url: `https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm`

```
Input: A graph G whose edges have distinct weights
Initialize a forest F to be a set of one-vertex trees, one for each vertex
of the graph.
While F has more than one component:
    Find the connected components of F and label each vertex of G by its
    component
    Initialize the cheapest edge for each component to "None"
    For each edge uv of G:
        If u and v have different component labels:
            If uv is cheaper than the cheapest edge for the component of u:
                Set uv as the cheapest edge for the component of u
            If uv is cheaper than the cheapest edge for the component of v:
                Set uv as the cheapest edge for the component of v
        For each component whose cheapest edge is not "None":
            Add its cheapest edge to F
Output: F is the minimum spanning forest of G.
```

As in the pseudocode, our implementation also requires that the weights associated with each edge be distinct. However, one can easily circumnavigate this by having a consistent selection criteria to resolve such conflicts. The time complexity of this algorithm is O(E log V), because we are logarithmically consuming the number of components before the outer loop terminates (O(V)) and since we have E edges, we perform this at most E times as we continue to add an edge to F. Therefore, all in all, this algorithm takes O(E log V) time, which is the same time as Kruskal's and Prim's Algorithm.

3 Boruvka Implementation

Below we have a Java Implementation of Boruvka's algorithm that follows the pseudocode. For convenience, we have placed the class that holds the algorithm first.

Main.java

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.PriorityQueue;

class Main {
  public static void main(String[] args) {
    Graph g = new Graph(6);
        g.insertEdge(0, 1, 1);
        g.insertEdge(0, 4, 2);
        g.insertEdge(0, 5, 4);
        g.insertEdge(1, 4, 3);
        g.insertEdge(1, 5, 4);
        g.insertEdge(2, 3, 5);
        g.insertEdge(2, 4, 4);
        g.insertEdge(3, 4, 7);
        g.insertEdge(4, 5, 4);
        System.out.println("G:");
        g.prettyPrint();
        System.out.println();
        System.out.println("Spanning Tree:");
        Graph minSpanTree = Boruvka(g);
        minSpanTree.prettyPrint();
        minSpanTree.printTotalCost();

/*      uncomment to find maximum spanning
        Graph negated = Negate(g);
        Graph negMaxSpanTree = Boruvka(negated);
        Graph maxSpanTree = Negate(negMaxSpanTree);
        maxSpanTree.prettyPrint();
        */
  }

  public static Graph Negate(Graph g){
    int v = g.get_num_vertices();
    Graph negated = new Graph(v);
    for(int i = 0; i < v; i++){
      Iterator<Edge> it = g.neighbors(i).iterator();
      while(it.hasNext()){
        Edge e = it.next();
        float wt = e.getWeight();
        wt = wt * -1;
        e.setWeight(wt);
        negated.insertEdge(e);
      }//all edges now have negative weight
    }
    return negated;
  }

  public static Graph Boruvka(Graph g) {
        int v = g.get_num_vertices();

        Graph minSpanTree = new Graph(v);
        Edge curr = new Edge(-1,-1,Integer.MAX_VALUE);

        while (!minSpanTree.isConnected()) {
            //for each vertex
            for (int i = 0; i < v; i++) {

                if (minSpanTree.totalReachableVertices(i) < v) {

                    Iterator<Edge> it = g.neighbors(i).iterator();
                    while(it.hasNext()){
                        Edge e = it.next();
                      //find cheapest edge E from V to vertex outside of the considered Co
                        if(!minSpanTree.hasEdge(e) && !minSpanTree.reachable(i, e.getTail()
                            minSpanTree.insertEdge(e);
                            if(e.getWeight() < curr.getWeight()){
                              curr = e;
                            }
                        }
                    }
                }
            }
            //add the cheapest edge to F
            minSpanTree.insertEdge(curr);
        }
        return minSpanTree;
    }
}
```

Graph.java

```
import java.util.Iterator;
import java.util.PriorityQueue;

public class Graph {
    private int num_vertices;
    private PriorityQueue<Edge>[] graph;

    public int get_num_vertices() {
        return num_vertices;
    }

    public Graph(int num_vertices) {
        this.num_vertices = num_vertices;
        graph = new PriorityQueue[num_vertices];
        for (int i = 0; i < num_vertices; i++)
            graph[i] = new PriorityQueue<Edge>();
    }

    public void insertEdge(int i, int j, float weight) {
        if (!hasEdge(new Edge(i, j, weight))) {
            graph[i].add(new Edge(i, j, weight));
            graph[j].add(new Edge(j, i, weight));
        }
    }

    public void insertEdge(Edge e) {
        if (!hasEdge(e)) {
            graph[e.getHead()].add(e);
            graph[e.getTail()].add(e.reverse());
        }
    }
    public Edge getCheapest(){
      PriorityQueue<Edge> hello = graph[0];
      Edge e = new Edge(hello.peek().getHead(),
      hello.peek().getTail(),hello.peek().getWeight());
      return e;
    }

    public void removeEdge(int i, int j) {
        Iterator<Edge> it = graph[i].iterator();
        Edge other = new Edge(i, j, 0);
        while (it.hasNext()) {
            if (it.next().isSame(other)) {
                it.remove();
```

```java
                break;
            }
        }

        Iterator<Edge> it2 = graph[j].iterator();
        Edge other2 = new Edge(j, i, 0);
        while (it2.hasNext()) {
            if (it2.next().isSame(other2)) {
                it2.remove();
                break;
            }
        }
    }

    public boolean hasEdge(Edge e) {
        Iterator<Edge> it = graph[e.getHead()].iterator();
        while (it.hasNext()) {
            if (it.next().isSame(e)) {
                return true;
            }
        }
        return false;
    }

    public PriorityQueue<Edge> neighbors(int vertex) {
        return graph[vertex];
    }

    public boolean isConnected() {
       boolean visited[] = new boolean[num_vertices];
        int i;
        for (i = 0; i < num_vertices; i++) {
            visited[i] = false;
        }
        for (i = 0; i < num_vertices; i++) {
            if (neighbors(i).size() == 0)
            return false;
        }
        DepthFirstSearch(0, visited);
        for (i = 0; i < num_vertices; i++)
            if (visited[i] == false)
                return false;
        return true;
    }

    public void DepthFirstSearch(int sourceVertex, boolean visited[]) {
        visited[sourceVertex] = true;
        Iterator<Edge> it = neighbors(sourceVertex).iterator();
        while (it.hasNext()) {
            Edge nextVertex = it.next();
            if (!visited[nextVertex.getTail()])
                DepthFirstSearch(nextVertex.getTail(), visited);
        }
    }

    public int totalReachableVertices(int sourceVertex) {
        boolean visited[] = new boolean[num_vertices];
        for (int i = 0; i < num_vertices; i++) {
            visited[i] = false;
        }
        DepthFirstSearch(sourceVertex, visited);
        int count = 0;
        for (int i = 0; i < num_vertices; i++) {
            if(visited[i] == true)
                count++;
        }
        return count;
    }

    public boolean reachable(int sourceVertex, int destVertex){
        boolean visited[] = new boolean[num_vertices];
        for (int i = 0; i < num_vertices; i++) {
            visited[i] = false;
        }
        DepthFirstSearch(sourceVertex, visited);
        return visited[destVertex];
    }

    public void printTotalCost(){
      float cost = 0;
      for (int i=0; i < num_vertices; i++){
        PriorityQueue<Edge> edges = neighbors(i);
        Iterator<Edge> it = edges.iterator();
        Edge current;
        for(int j=0; j < edges.size();j++){
          //System.out.print(it.next() + " ");
            current = it.next();
          if(current.getTail()>=i){
            cost+=current.getWeight();
          }
        }
      }
      System.out.println("The Total cost is: "+ cost);
    }

    //Note: adjacency matrix based graph so prints edges twice
    public void prettyPrint() {
        for (int i = 0; i < num_vertices; i++) {
            PriorityQueue<Edge> edges = neighbors(i);
            Iterator<Edge> it = edges.iterator();
            System.out.print(i + ": ");
            for (int j = 0; j < edges.size(); j++) {
                System.out.print(it.next() + " ");
            }
            System.out.println();
        }
    }
}
```

Edge.java

```
public class Edge implements Comparable{
    private int head;
    private int tail;
    private float weight;

    public Edge(int head, int tail, float weight) {
        this.head = head;
        this.tail = tail;
        this.weight = weight;
    }

    public int getHead() {
        return head;
    }

    public int getTail() {
        return tail;
    }

    public float getWeight() {
        return weight;
    }

    public void setWeight(float wt){
        this.weight = wt;
    }

    public Edge reverse(){
        Edge e = new Edge(tail, head, weight);
        return e;
    }

    public boolean isSame(Edge other) {
        if (this.head == other.head) {
            if (this.tail == other.tail) {
                return true;
            }
        }
        return false;
    }

    public int compareTo(Object o) {
        Edge other = (Edge) o;
        return Double.compare(this.weight, other.weight);
    }

    public String toString() {
        return "[" + head + " , " + tail +" , " + " " + weight + "]";
    }
}
```

4 Maximum Spanning Tree

To find the Maximum Spanning Tree, we can simply negate the weight of each edge, find a minimum spanning tree, and revert the weights of the edges in the graph back to their original weights. http://mathworld.wolfram.com/MaximumSpanningTree.html

```
public static Graph Negate(Graph g){
    int v = g.get_num_vertices();
    Graph negated = new Graph(v);
    for(int i = 0; i < v; i++){
      Iterator<Edge> it = g.neighbors(i).iterator();
      while(it.hasNext()){
        Edge e = it.next();
        float wt = e.getWeight();
        wt = wt * -1;
        e.setWeight(wt);
        negated.insertEdge(e);
      }//all edges now have negative weight
    }
    return negated;
    }
```

We can use the Negate method within the main method as follows:

```
Graph negated = Negate(g);
Graph negMaxSpanTree = Boruvka(negated);
Graph maxSpanningTree = Negate(negMaxSpanTree);
maxSpanningTree.prettyPrint();
```

maxSpanningTree is now a spanning tree with maximum weight.

5 Number of Spanning Trees

If G is a connected graph, we know that it must contain a spanning tree. We also know that if G is a tree, then it contains exactly one spanning tree, namely G itself. However, if G is connected but is not a tree, then G contains more than one spanning tree. But knowing how many can be tricky. We have divided this question into 2 different cases, one where G is complete(K_n) and one where G is not complete. We can see that if G is complete and $V(G) = \{v_1, v_2, \ldots, v_n\}$ then finding the number of spanning trees is equivalent to finding the number of distinct trees with vertex set $\{v_1, v_2, \ldots, v_n\}$. So by the theorem below from Chartrand and Zhang, we can easily compute the number of spanning trees in constant time (provided we know the number of vertices, which can otherwise be obtained with one pass over the data structure counting up the total vertex count) by simply computeing n^{n-2}.

Theorem 1 (Cayley Tree Formula)**.** *The number of distinct trees of order n with a specified vertex set is* n^{n-2}.

Now, let's consider the case where G is not complete. The next method was discovered by Gustav Kirchhoff.

Theorem 2 (Matrix Tree Theorem)**.** *Let G be a graph with* $V(G) = \{v_1, v_2, \ldots, v_n\}$*, let* $A = [a_{ij}]$ *be the adjacency matrix of G and let* $C = [c_{ij}]$ *be the* nxn *matrix, where*

$$c_{ij} = \begin{cases} deg\ v_i & if \quad i = j \\ -a_{ij} & if \quad i \neq j \end{cases}$$

Then the number of spanning trees of G is the value of any cofactor of C.

We provide an implementation that uses the Matrix Tree Theorem to output the number of spanning trees given a graph G. This is also stated as Theorem 4.15 in Chartrand and Zhang.

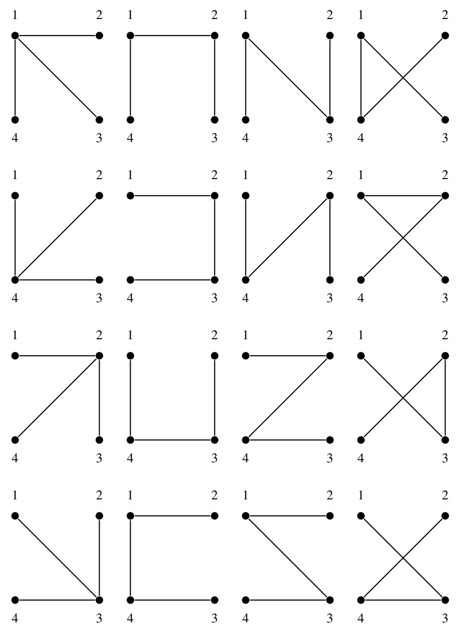
6 Number of Spanning Trees Implementation

```
import numpy as np
import networkx as nx
from sympy import Matrix
```

```
def getLaplacianMatrix(m):
    G = nx.from_numpy_matrix(m)
    G = nx.laplacian_matrix(G).toarray()
    return G

def getCofactorMatrix(arr):
    Co = Matrix(arr)
    Final = Co.adjugate().T
    return Final

def totalSpanningTrees(m):
    G = getLaplacianMatrix(m)
    Co = getCofactorMatrix(G)
    N = Co[0]
    return N

#Complete graph on 4 vertices => 16 (output) = 4^2(by Cayley Tree Formula)
k_4 = np.matrix('1, 1, 1, 1;'
              '1, 1, 1, 1;'
              '1, 1, 1, 1;'
              '1, 1, 1, 1')

#K_4 minus an edge  => 8 correct
k_4_e = np.matrix('1, 1, 1, 1;'
              '1, 1, 0, 1;'
              '1, 0, 1, 1;'
              '1, 1, 1, 1')

#random graphh =>     8 correct
rand = np.matrix('1, 1, 1, 1;'
              '1, 1, 0, 0;'
              '0, 1, 1, 1;'
              '0, 0, 1, 1')
#   => stands for "output"
print(totalSpanningTrees(k_4))  #    =>     16 = 4^2 (by Cayley Tree Formula)
print(totalSpanningTrees(k_4_e))#    =>     16 correct
print(totalSpanningTrees(rand)) #     =>      8 correct
```

Confirmation

a. Given K_4, there are 16 different spanning trees as shown below.



b. For K_4 - e, we will have the 8 remaining spanning trees shown below. (e is the right edge (23) and spanning trees missing this edge were left blank)