**Graph algorithms - Trees**

**Definition and properties**

**Definition:** A *tree* is an undirected graph that is connected and has no cycles

 We understand by *cycle* a closed walk with no repeating vertices (except that the first and the last vertex are the same) and no repeating edges. This means that, if there is an edge between vertices 1 and 2, the walk (1, 2, 1) is not a cycle because it uses the same edge twice (once in each direction).

 The smalles tree that fits the definition consists in a single isolated vertex.

 There is a big difference between *non-rooted trees* considered here and the *rooted trees* used especially for data structures. Any tree becomes a rooted tree by distinguishing any of its vertices as root and directing all edges away from the root. Viceversa, any rooted tree becomes a non-rooted tree if we forget the distinguished vertex and the parent-child direction of edges.

 For data structures, we most often distinguish an order among the children of a vertex. Thus, there are rooted tree with no order among children (we simply have a root and all edges directed away from it), and rooted tree with order on children (where, in addition, we distinguish an order among the children). For binary trees, in addition, we sometimes distinguish between a node with only a left child and a node with only a right child (this is the case for binary search trees, for example). All these kind of trees are distinct. Bottom line: trees studied here are non-rooted and there is no order among the neighbours of any given node.

Equivalent properties for an undirected graph:

1. The graph is a tree;
2. The graph is connected and has at most *n-1* edges (where *n* is the number of vertices);
3. The graph has no cycles and has at least *n-1* edges;
4. The graph is connected and minimal (if we remove any edge, it becomes non-connected);
5. The graph has no cycles and is maximal (if we add any edge, it closes a cycle);
6. For any pair of vertices, there is a unique path connecting them.

**The minimum spanning tree problem**

Given a graph with non-negative costs, find a tree with the same vertices and a subset of the edges of the original graph (a *spanning tree*) of minimum total cost.

Example: input graph:

A diagram of a triangle with numbers and a triangle

Description automatically generated

There are two well-known algorithms for solving this problem: Kruskal's algorithm and Prim's algorithm.Kruskal's algorithm

**Idea**

The idea is to start with a graph with all the vertices and no edges, and then to add edges that do not close cycles. This way, as the algorithm progresses, the graph will consist in small trees (it will be what is called a *forest* - a graph with no cycles, meaning that its connected components are trees), and those trees are joined together to form fewer and larger trees, until we have a single tree spanning all the vertices. In doing all the above, we use the edges in increasing order of their cost.

**The basic algorithm**

Input:

G : undirected graph with costs

Output:

edges : a collection of edges, forming a minimum cost spanning tree

Algorithm:

e0,...,em-1 = list of edges sorted increasing by cost

edges = ∅

for i from 0 to m-1 do

if edges ∪ {ei} has no cycles then

edges.add(ei)

end if

end for

|  |  |
| --- | --- |
| **Edge** | **Cost** |
| 1-2\* | 1 |
| 2-6\* | 1 |
| 4-5\* | 1 |
| 1-6 | 2 |
| 1-3\* | 2 |
| 3-6 | 2 |
| 1-4\* | 3 |
| 3-5 | 3 |
| 3-4 | 4 |
| 5-6 | 5 |

A diagram of a number

Description automatically generated with medium confidence   A diagram of a triangle

Description automatically generated   A diagram of a triangle

Description automatically generated   A diagram of a triangle

Description automatically generated   A diagram of a triangle with numbers and lines

Description automatically generated

**Issue with the basic algorithm**

The difficult part here is how to test the existence of cycles. There is a much easier way: to keep track of the connected components of edges, and to notice that a cycle is formed when adding a new edge if and only if the endpoints of the edge are in the same component.

Keeping track of the connected components is an interesting problem in itself.

Ideas:

* Each component is kept as a rooted tree (independent of the tree of the original graph);
* Each component has a *representative* vertex that is the root of the tree; each vertex in the component has a pointer to its parent;
* Therefore, to test if two vertices are in the same component, we go from each of them up to the representative of its component, and we verify if we reach the same vertex;
* When joining two components, we place the representative of one component as a child of the representative of the other. This way, all vertices in the first component are moved to the second.
* There are 2 optimizations to prevent the height of the rooted tree to increase:
  + When joining two components of different heights, the representative of the shortest is set as child of the representative of the tallest (never vice-versa);
  + When retrieving the representative, we *compress the path*, that is, we place all the vertices along the path to the representative as direct children of the representative.

**Proof of correctness**

The proof is a clasical proof for a greedy algorithm: we compare the Kruskal's solution with the optimal solution for the problem, find the first difference, and modify the optimal solution, without loosing the optimality, so that to match better the Kruskal's solution. By repeating the above step, we turn the optimal solution into Kruskal's solution without loosing the optimality, thus proving that Kruskal's solution is optimal.

**Prim's algorithm**

**Idea**

Prim's algorithm is similar to Kruskal's algorithm; however, instead of starting with lots of trees and joining them together, Prim's algorithm starts with a single tree consisting in a single vertex, and then grows it until it covers all the vertices. At each step, an edge is added, connecting an exterior vertex to the tree. Among all the edges connecting a vertex outside the tree with one in the tree, it is choosen the edge of smallest cost.

**The algorithm**

Input:

G : directed graph with costs

Output:

edges : a collection of edges, forming a minimum cost spanning tree

Algorithm:

PriorityQueue q

Dictionary prev

Dictionary dist

edges = ∅

choose s in X arbitrarily

vertices = {s}

for x in N(s) do

dist[x] = cost(x, s)

prev[x] = s

q.enqueue(x, d[x]) // second argument is priority

while not q.isEmpty() do

x = q.dequeue() // dequeues the element with minimum value of priority

if x ∉ vertices then

edges.add({x, prev[x]})

vertices.add(x)

for y in N(x) do

if y not in dist.keys() or cost(x,y) < dist[y] then

dist[y] = cost(x, y)

q.enqueue(y, dist[y])

prev[y] = x

end if

end for

end if

end while

A diagram of a diagram

Description automatically generated with medium confidence   A diagram of a connection

Description automatically generated   A diagram of a triangle with numbers and lines

Description automatically generated   A diagram of a triangle with numbers and lines

Description automatically generated   A diagram of a triangle with numbers and lines

Description automatically generated