Design and Analysis of Algorithms

Lecture 06 – Minimum Spanning Trees and Shortest Paths

Minimum Spanning Tree

Consider a fully connected graph where the edges represents some kind of weight (or length, cost, duration, etc.) of connecting two vertices.

The most common example would be infrastructure networks (road, water, telecoms, etc) where the edges represent connections between locations (houses, regions, etc.)

The original MST algorithm (Boruvka, 1926) was developed to minimize the distance covered in electrical distribution network of Moravia (then part of Czechoslovakia).

Another interesting example is that of a contagion networks (diffusion of innovation, supply chain disruptions, consecutive market failures, inflation, foreign exchange rates, re-tweets, etc.) where the vertices represents agents that interact.

Many decision making models can be modeled into communication networks, and given that there is some kind of communication cost that is represented in the model, they can be developed into the graphs we are interested.

Minimum Spanning Tree

Any graph would have a number of trees that spans all its vertices.

How about one or more trees that span all vertices, while using edges with a minimum aggregate weight? For the obvious road network example, this means traveling the smallest distance.

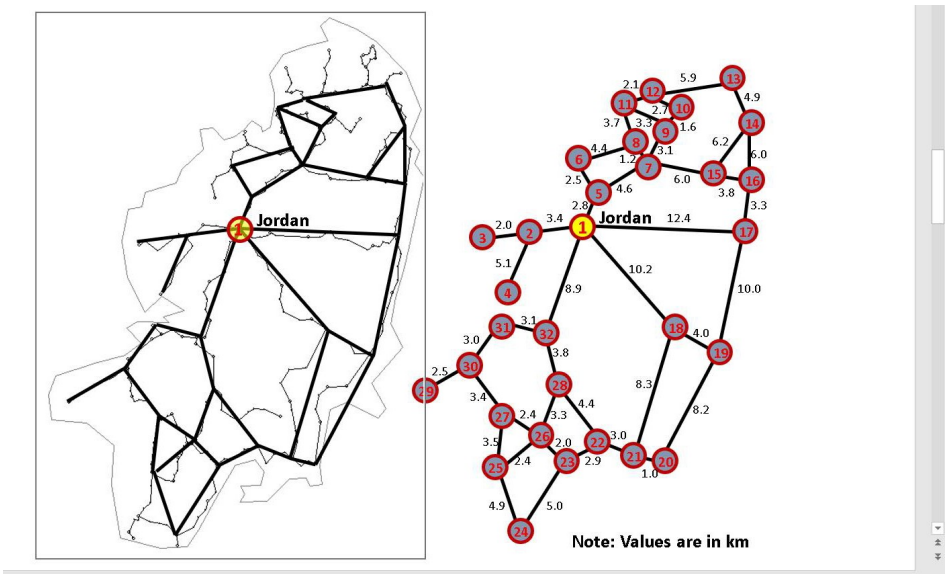
These trees are minimum spanning trees (MST).

There may be multiple MSTs for a graph.

Discussion. If all edge weights are equal then any spanning tree is an MST. Why?

Discussion. If each edge has a distinct weight then there will be only one, unique minimum spanning tree. Why?

For a disjoint graph, we will consider multiple MSTs forming a minimum spanning forest (MSF)



Minimum Spanning Tree

A number of points to consider

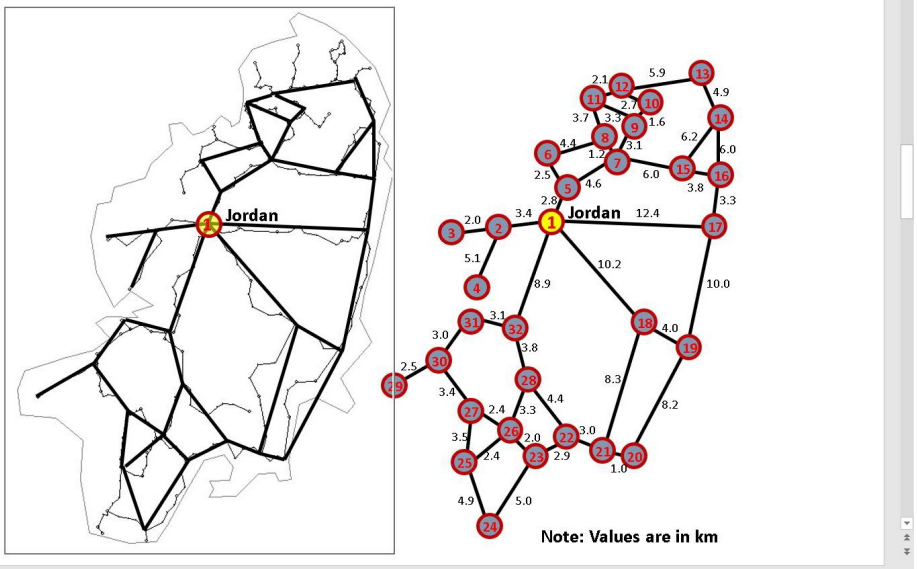
In many graphs, the weights between nodes will not satisfy the triangle inequality. Therefore we cannot make assumptions based on it.

Example. Road networks with the distance defined as duration.

In many graphs weights will depend on directionality.

Example. Telecoms networks with asymmetrical capacity such as ADSL connections, road networks with one way streets, foreign exchange networks with agents offering different prices for buying and selling.

In theory edge weights can be negative. We cannot always assume non-negative weights. In cases where physical activities are modeled, we usually have non-negative and mostly positive edge weights.



Minimum Spanning Tree

In short

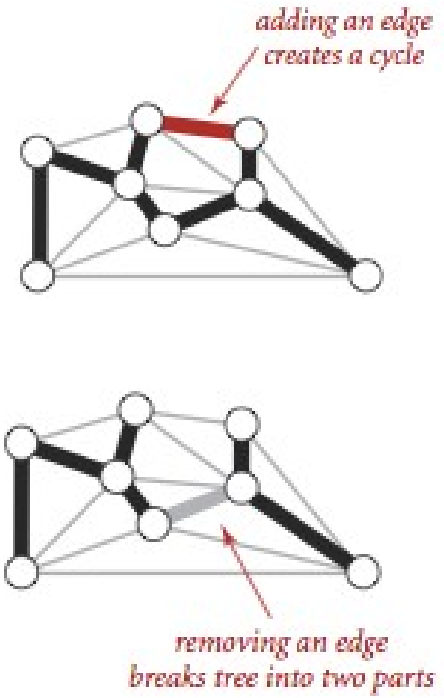
An edge-weighted graph is a graph where we associate weights or costs with each edge.

A minimum spanning tree (MST) of an edge-weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree.

Recall that

Adding an edge that connects any two vertices in a tree creates a unique cycle.

Removing an edge from a tree breaks it into two separate sub-trees.



Minimum Spanning Tree

An MST involves evaluating cycles and cuts.

For any cycle C in the graph, if the weight of an edge e of C is larger than the individual weights of all other edges of C, then this edge cannot belong to an MST.

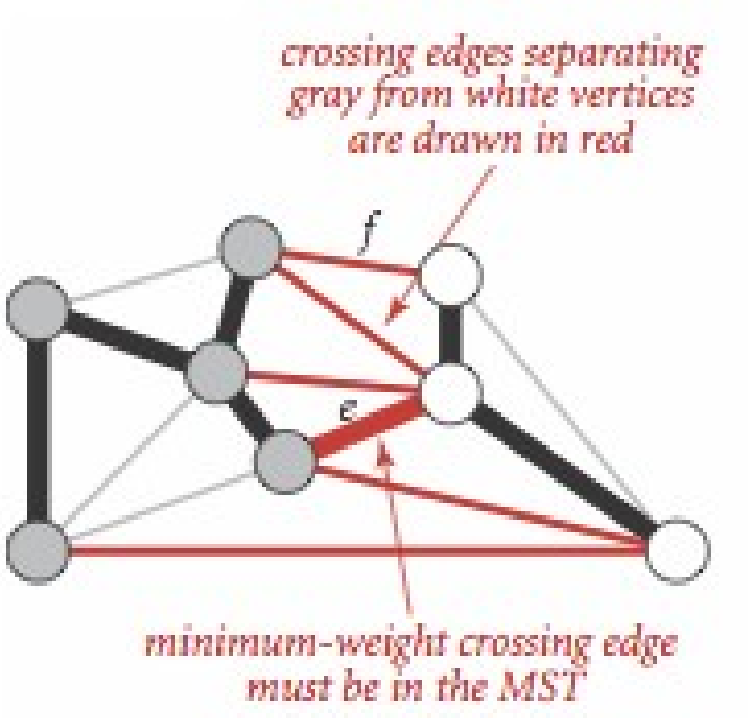
For any cut C of the graph, if the weight of an edge e in the cut-set of C is strictly smaller than the weights of all other edges of the cut-set of C, then this edge belongs to all MSTs of the graph.

The cut property is the basis for the basic algorithms we will cover.

All these algorithms are variations of the greedy algorithm.

They are based on proof by contradiction.

If we didn't make the "greedy" choice, then we will find that we should have made that choice.



Minimum Spanning Tree

Greedy MST algorithm (Prims's Algorithm)

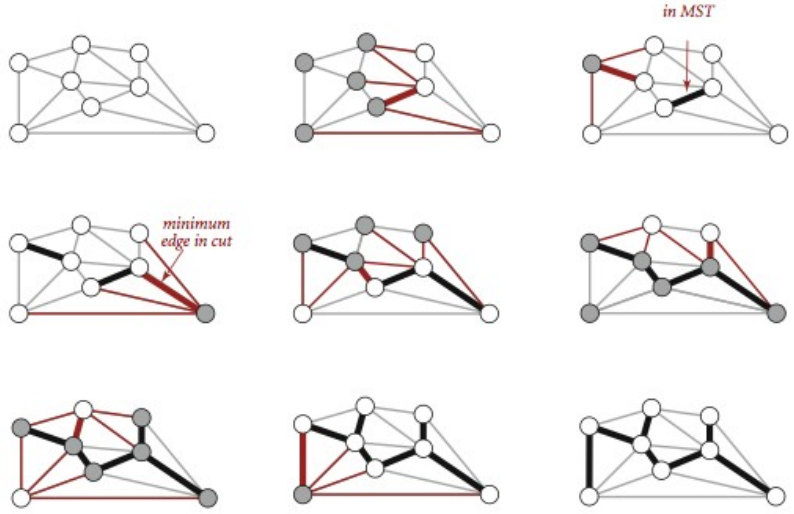
Given a fully-connected graph with v vertices, starting with all edges colored gray, find a cut with no black edges, color its minimum-weight edge black, and continue until v-1 edges have been colored black.

This means

Start with an arbitrary subset of edges that form a cut, find the minimum valued edge and select it as an edge in the MST.

Redo this with the remaining edges.

You need to find v-1 edges.



Minimum Spanning Tree

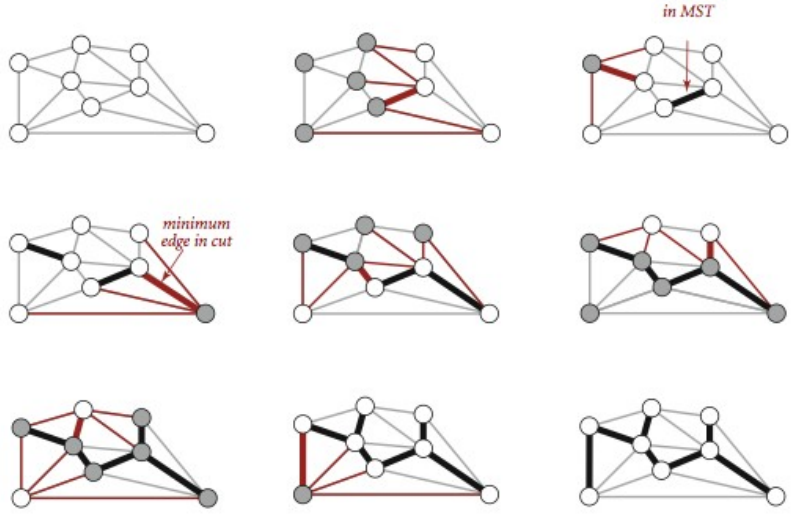
Greedy MST algorithm (Generic Discussion)

What is the complexity of this algorithm?

Assuming n vertices, and e edges.

- Note that, in a bi-directional graph with n vertices we end up with a maximum of n^2 edges so e <= n^2.
- Sorting the edges according to cost is a simple O(eloge) task,
- However, in terms of the number of vertices this could be equivalent to O(n^2 x logn^2), n^2 not due to algorithmic complexity but due to possibly much larger number of edges.
- If there is no connection between two vertices, then we can assume there is an edge with an infinite weight/cost.

Typically the number of edges is similar to the number of vertices. But let's always consider the worst case of n^2.



Minimum Spanning Tree

Greedy MST algorithm (Generic Discussion)

Then we generate a cut.

A cut is generated by selecting two subsets of the vertices (forming grey and white disjoint graphs).

Discussion. How to select these subsets? Is there any complexity involved?

We get a list of edges connecting these two subsets. To do that,

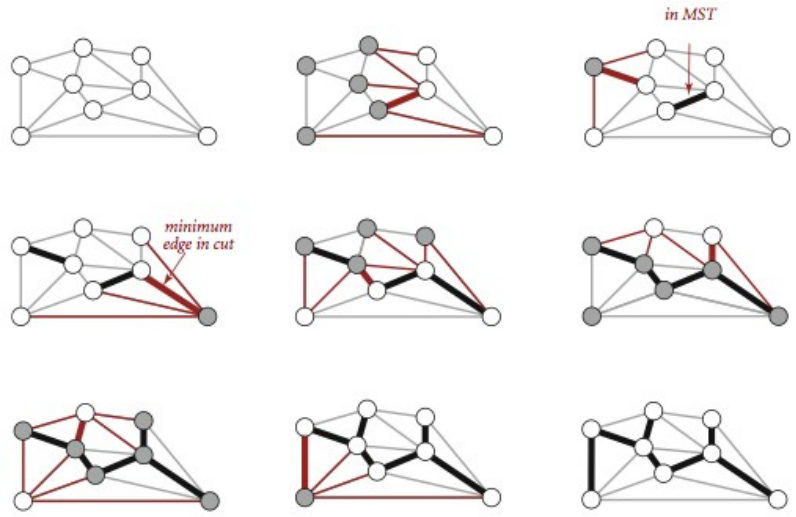
For each vertex in the grey graph, we find the edges that connects it to any vertex in the white sub-graph and add them to a temporary list.

Discussion. For n vertices, the maximum size of this temporary list is n-1. Why? What kind of graph topology would generate exactly n-1 entries?

From the cut generated, we select the edge for an MST.

Find the minimum valued edge in the temporary list. Easy. O(n).

Discussion. How many times do we generate a cut? Does it depend on our method of generating the cut?



Minimum Spanning Tree

Greedy MST algorithm (Kruskal's Algorithm)

For a graph with n vertices and e edges.

Create an empty set of edges to represent the MST.

Step 1. Sort all the edges in non-decreasing order of their weight.

This is o(eloge) as discussed. Here e could be as much as n^2.

Step 2 . Until we have n-1 edges in the MST.

Pick the smallest edge.

Check if it forms a cycle with the spanning tree formed so far.

Kruskal uses the Union-Find algorithm for this. This is O(n).

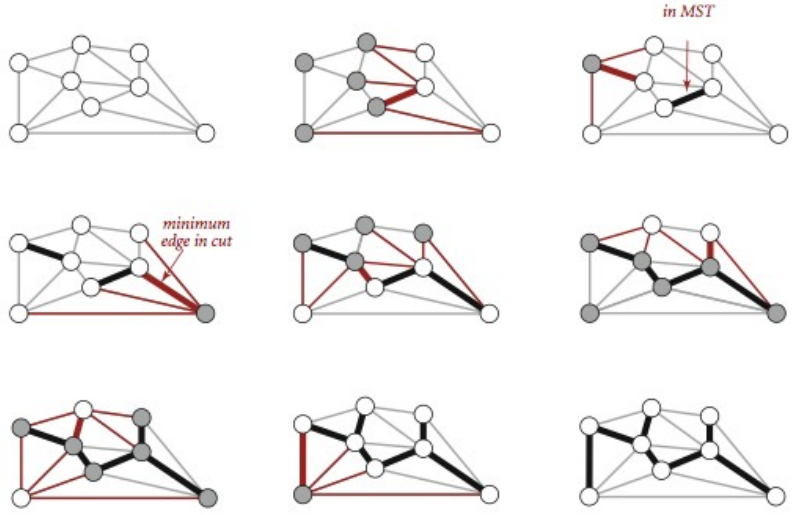
If cycle is not formed, include this edge. Else, discard it.

Step 2 is o(n^2)

So Kruskal's algorithm is o(eloge + n^2) which is equivalent to

O(eloge) if the number of edges is much larger than the number of vertices, or

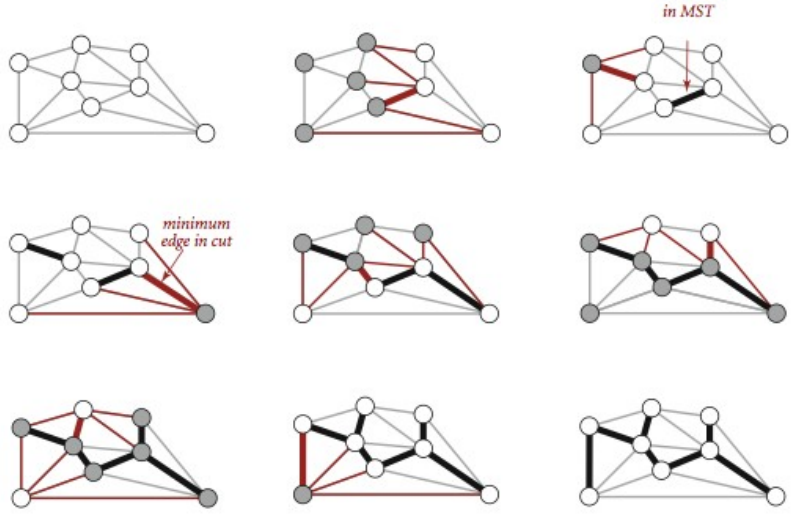
O(n^2) if the number of edges is comparable to the number of vertices.



Minimum Spanning Tree

Greedy MST algorithm (Prim's Algorithm)

- For a graph with n vertices and e edges.
- Create an empty set of edges to represent the MST.
- Create a list of key values to be assigned to each vertex.
  - Assign a key value to all vertices in the input graph.
  - Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.



Minimum Spanning Tree

Greedy MST algorithm (Prim's Algorithm)

Until the set of edges for the MST includes all vertices.

Step 1. Pick a vertex u such that it is not in the MST, and its value is minimum.

Step 2. Add u to the MST.

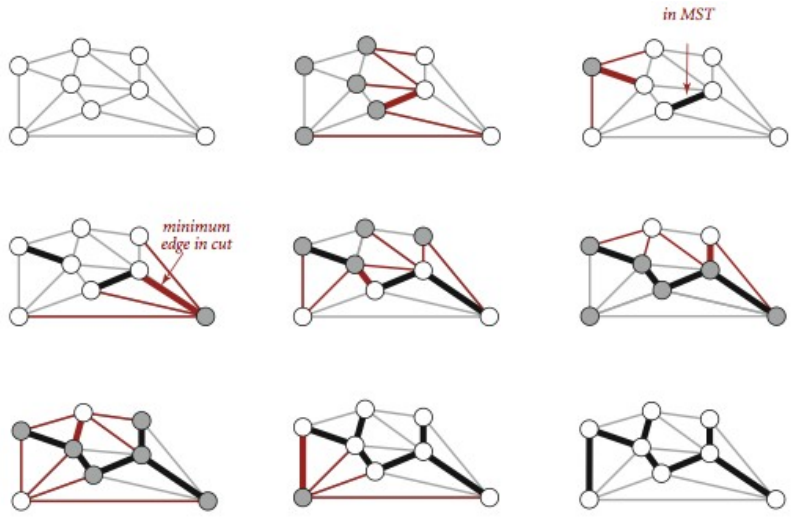
This means finding the minimum weight edge connecting u to the existing vertices in the MST, and selecting that edge as part of MST.

Step 3. Update the key values of all vertices adjacent to u.

To update the key values, iterate over all adjacent vertices. For each vertex v that is adjacent to u,

If the weight of edge u-v is less then the key value of v, then update the key value as the weight of the edge u-v.

Note that this key value was initialized as INFINITE, then maybe updated to a lesser value a few times.



Minimum Spanning Tree

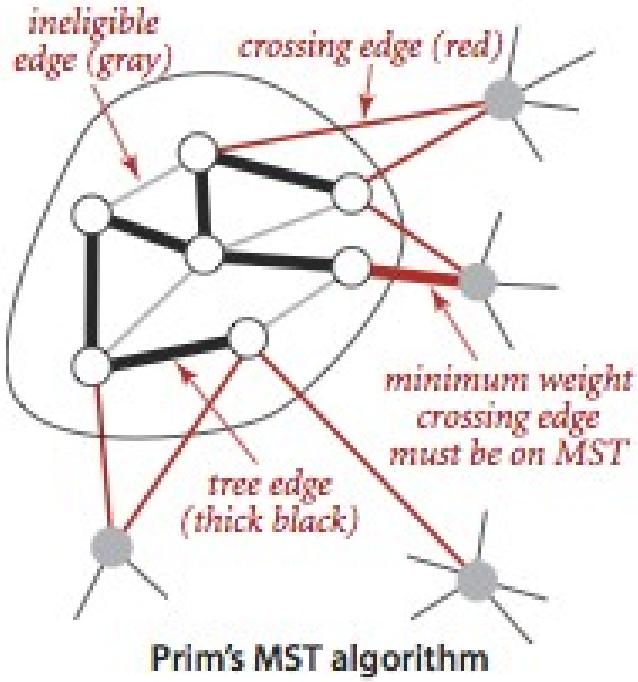
Greedy MST algorithm (Prim's Algorithm)

The idea is to have the key value as the weight of connecting the vertex v to the current MST.

- Initially all key values but one are INFINITE.
- Therefore that vertex with zero key value is the initial vertex added to the MST.
- From that point on, each time we update the MST, we also update the minimum distance of each vertex to the updated MST.
- Because the update the minimum distances, some edges become **ineligible**.

Prim's algorithm uses more data structures, and its complexity is O(elogn).

- When the number of edges is much larger, this is comparable to O(eloge). **Discussion.** Why? How much larger can O(eloge) be than O(elogn). Is this complexity difference relevant to Big-O notation?
- When the number of edges is comparable to the number of vertices this is comparable to O(nlogn) hence better than O(n^2).



Prim's MST algorithm

Minimum Spanning Tree

Greedy MST algorithm (Prim's Algorithm)

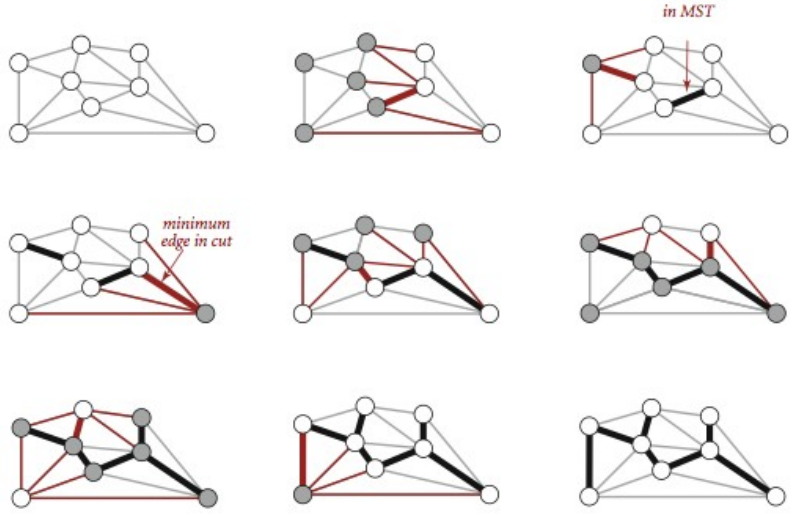
Additional data structures for Prim's algorithm.

- An array for the key values of each vertex.
- A boolean array to keep track of vertices already in the MST.
- A set of edges to construct the MST.
  - As each vertex has to be in the MST, we could simply organize this as an array parent[] where parent[v] would be the index of the parent node where vertex v connects to. Here the root node needs no parent, so setting -1 as the root node's parent's index is acceptable.

For a Java implementation using these three arrays as data structures, please review – https://bit.ly/3Dj5X2m

Note that data structure(s) to represent the graph would allow representation of vertices and edges.

- We particularly like data structures with the following capability.
  - We need to be able to add vertices, individually or in bulk.
  - We need to be able to add an edge, specifying the two vertices.
  - It is easier for many tasks to know the number of edges and vertices.
  - We need to be able to get all adjacent vertices to a given vertex (or be able to iterate over them).



Minimum Spanning Tree

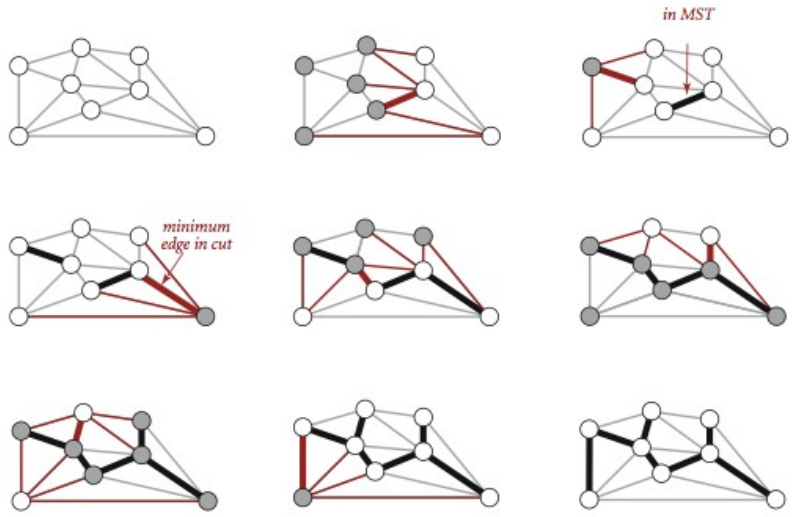
Greedy MST algorithm (Prim's Algorithm)

Note that data structure(s) to represent the graph would already allow representation of vertices and edges.

Recall, that we particularly liked data structures with the following capability.

> We need to be able to get all adjacent vertices to a given vertex (or be able to iterate over them.)

This capability becomes handy in implementing Prim's algorithm.



Minimum Spanning Tree

Data Structure for an Edge-Weighted Graph

An Edge class which implements

- Getting the weight
- Getting the pair of vertexes
- Comparison to another edge
- Equality to another edge

Minimum Spanning Tree

Data Structure for an Edge-Weighted Graph

A Graph class which uses the Edge class, and implements

- Getting the number of vertices
- Getting the number of edges
- Adding an edge (note that the edge can tell the vertexes involves)
- Getting a list of edges connecting to a given vertex
- Getting a list of vertices connected to a given vertex (this can be implemented using the edges)
- Getting a list of all edges

Minimum Spanning Tree

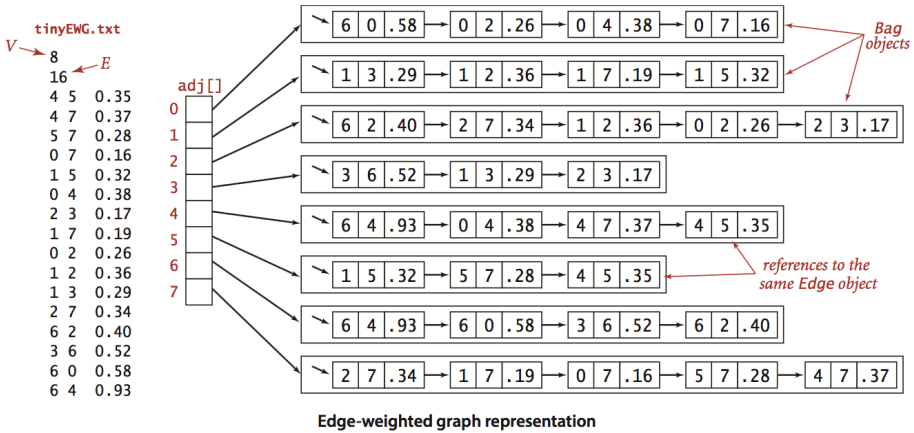
Data Structure for an Edge-Weighted Graph

For efficiency in both memory and computation, we prefer a map.

- The initial array is the adjacency array, which has the index as the index of each vertex and links to a bag (set) of edges connected to the selected vertex.
- For each edge, there are always two entries in two separate sets. Note that in Java, these sets are storing references.

Therefore we have v (or n) references to sets, and 2e references to edges.

The edges could be stored in an array for memory efficient storage.



Edge-weighted graph representation

Minimum Spanning Tree

There are two ways to define the density of a graph.

Old metric – Density = (number of edges) / (number of vertices) = e/v.

New metric – Density = (number of edges) / (number of possible edges).

For undirected graphs, the number of possible edges is v(v-1)/2. For directed graphs it is v(v-1).

The new metric allows the density be always within [0,1].

When to consider a graph as very large?

A very large graph is a graph for which brute force MST algorithms would be infeasible.

A brute force algorithm finds a solution by trying **all** possible answers and picking the best one.

In MST, the brute force approach would generate all spanning trees of the original graph, and then compare them based on total weight (cost, distance, etc.)

Most brute force MST algorithms are based on decision trees. A decision tree representation of a graph would have binary nodes which represent the comparison of weights between two edges in the actual graph.

Given a decision tree that represents a spanning tree of a sub-graph, it is very easy to validate that it actually represents an MST of the whole graph.

Therefore these algorithms generate (parallel or sequentially) a very large number of decision trees to represent all STs for all sub-graphs, and then validate (easily) each decision tree.

Use r = ceiling of ($\log_2 \log_2 \log_2 v$). This metric increases very slowly, so a large value of r is exceptional.

Discussion. Would it matter if we used logarithm in base 10?

Minimum Spanning Tree

Examples.

Undirected graph with 5.000 edges and 1.000 vertices.

Density = 5.000 / 1000 x 999 ~ 0.01. (1 percent, not very dense).

Undirected graph with 100.000 edges and 1.000 vertices.

Density = 100.000 / 1000 x 999 ~ 0.10. (10 percent, considerably dense).

Shortest Path

If our graph is an edge-weighted bi-directional graph (ie. digraph) then there is at least one shortest path, defined as

A directed path from vertex s to vertex t with the property that no other such path has a lower weight.

There are two applications of shortest paths.

Compute shortest path from one vertex (source node) to the other (destination mode).

Compute at the same time the shortest path from one vertex to all other vertices.

Dijkstra's algorithm

The original Dijkstra solution (1956) is focusing on the first type of problem in a road network.

Note that we will be focusing on the second type.

Non-negative weights.

Dijkstra's algorithm

Initialization

Mark all nodes unvisited.

Create a set of all the unvisited nodes called the unvisited set (Q).

Select an arbitrary node as the initial node to visit.

Assign to every node a tentative distance value: set it to zero for the initial node and to infinity for all other nodes.

The tentative distance of a node v represents the length of the shortest path **discovered so far** between the node v and the starting node.

Dijkstra's algorithm

While the set on unvisited nodes (Q) is non-empty.

Select u as the current node to visit based on u having the minimum distance to source discovered so far.

Optional. Set previously selected node as the parent of this node. (To construct a path).

Mark u as visited (removing it from Q)

For each unvisited neighbor v of u (which means that the neighbors are in Q)

Calculate an alternate distance from initial point to v, that passes through u so that

alternate distance = discovered distance to u + distance between u and v (ie. edge weight).

If the alternate distance to v is shorter than the previously discovered distance to v, then update the discovered distance. (This is called **relaxation**).

Note: It is not always necessary to visit the destination node.

Dijkstra's algorithm

Exercise

We want to find the minimum distance from node 0 to node 4.

In which order would we visit the nodes, starting from 0?

1-7-6-5..

At this point we know **a distance** to node 4. Then we visit

2-8-3

At this point we know **the minimum distance** to node 4.

We never visited node 4.