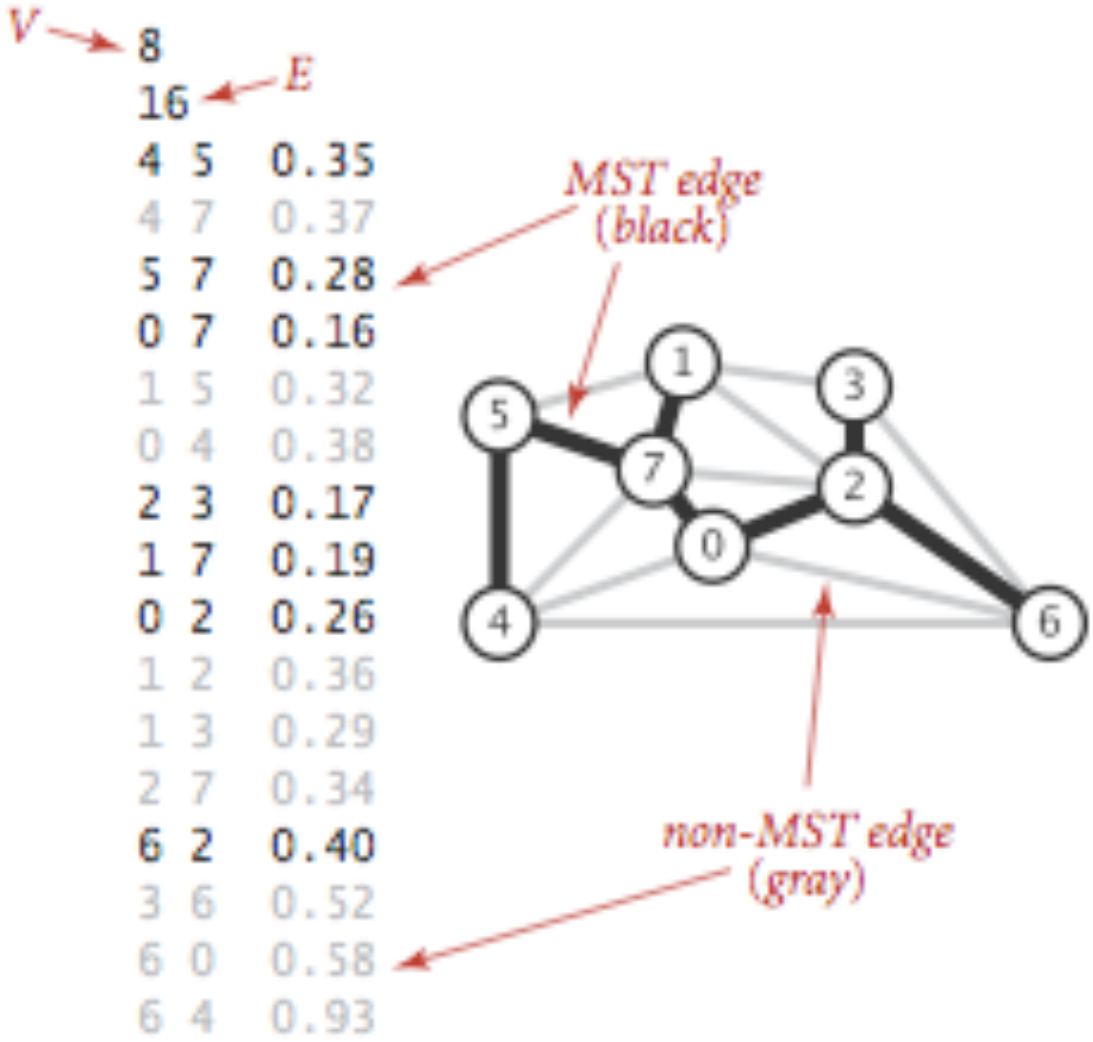
ID1020: Minimal Spanning Trees

kap. 4.3 från Algorithms 4th Edition, Sedgewick.

Minimal Spanning Tree

Minimum spanning tree. 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.

MST



An edge-weighted graph and its MST

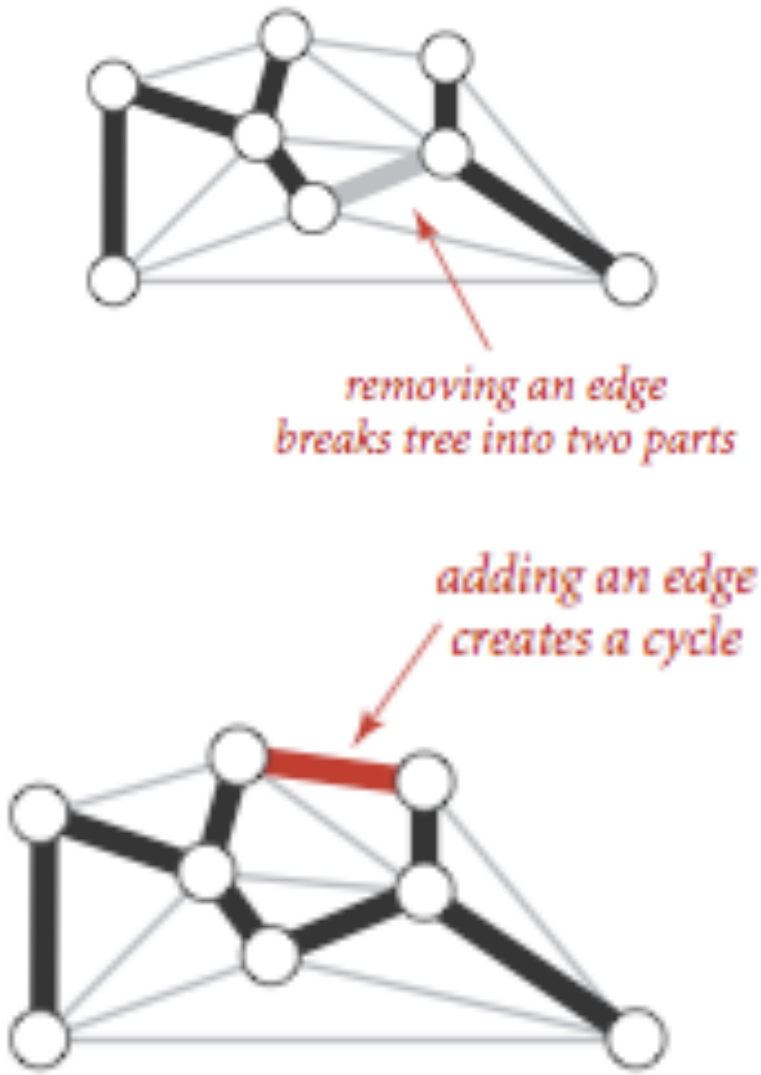
Some assumptions

- The graph is connected. The spanning-tree condition in our definition implies that the graph must be connected for an MST to exist. If a graph is not connected, we can adapt our algorithms to compute the MSTs of each of its connected components, collectively known as a minimum spanning forest.
- The edge weights are not necessarily distances. Geometric intuition is sometimes beneficial, but the edge weights can be arbitrary.
- The edge weights may be zero or negative. If the edge weights are all positive, it suffices to define the MST as the subgraph with minimal total weight that connects all the vertices.
- The edge weights are all different. If edges can have equal weights, the minimum spanning tree may not be unique. Making this assumption simplifies some of our proofs, but all of our our algorithms work properly even in the presence of equal weights.

Tree properties

Underlying principles. We recall two of the defining properties of a tree:

- Adding an edge that connects two vertices in a tree creates a unique cycle.
- Removing an edge from a tree breaks it into two separate subtrees.

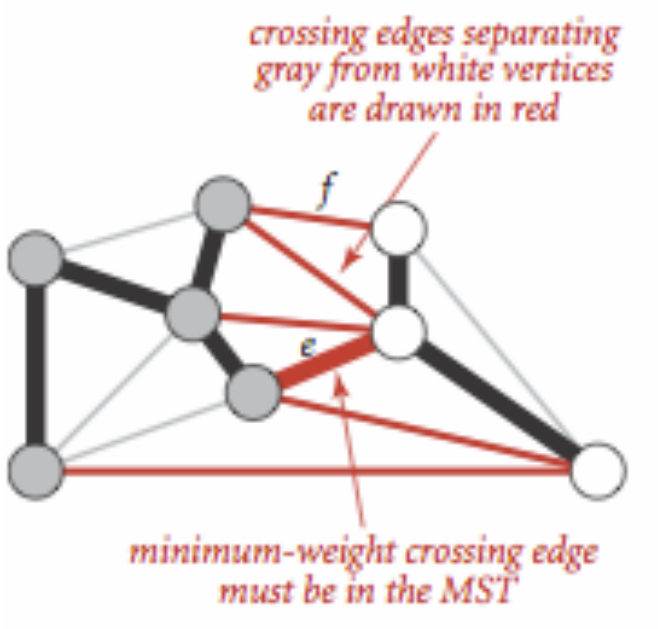


Cuts in graphs

A cut of a graph is a partition of its vertices into two disjoint sets. A crossing edge is an edge that connects a vertex in one set with a vertex in the other. For simplicity, we assume all edge weights are distinct. Under this assumption, the MST is unique. Define cut and cycle. The following properties lead to a number of MST algorithms.

Cuts

Proposition. (Cut property) Given any cut in an edge-weighted graph (with all edge weights distinct), the crossing edge of minimum weight is in the MST of the graph.

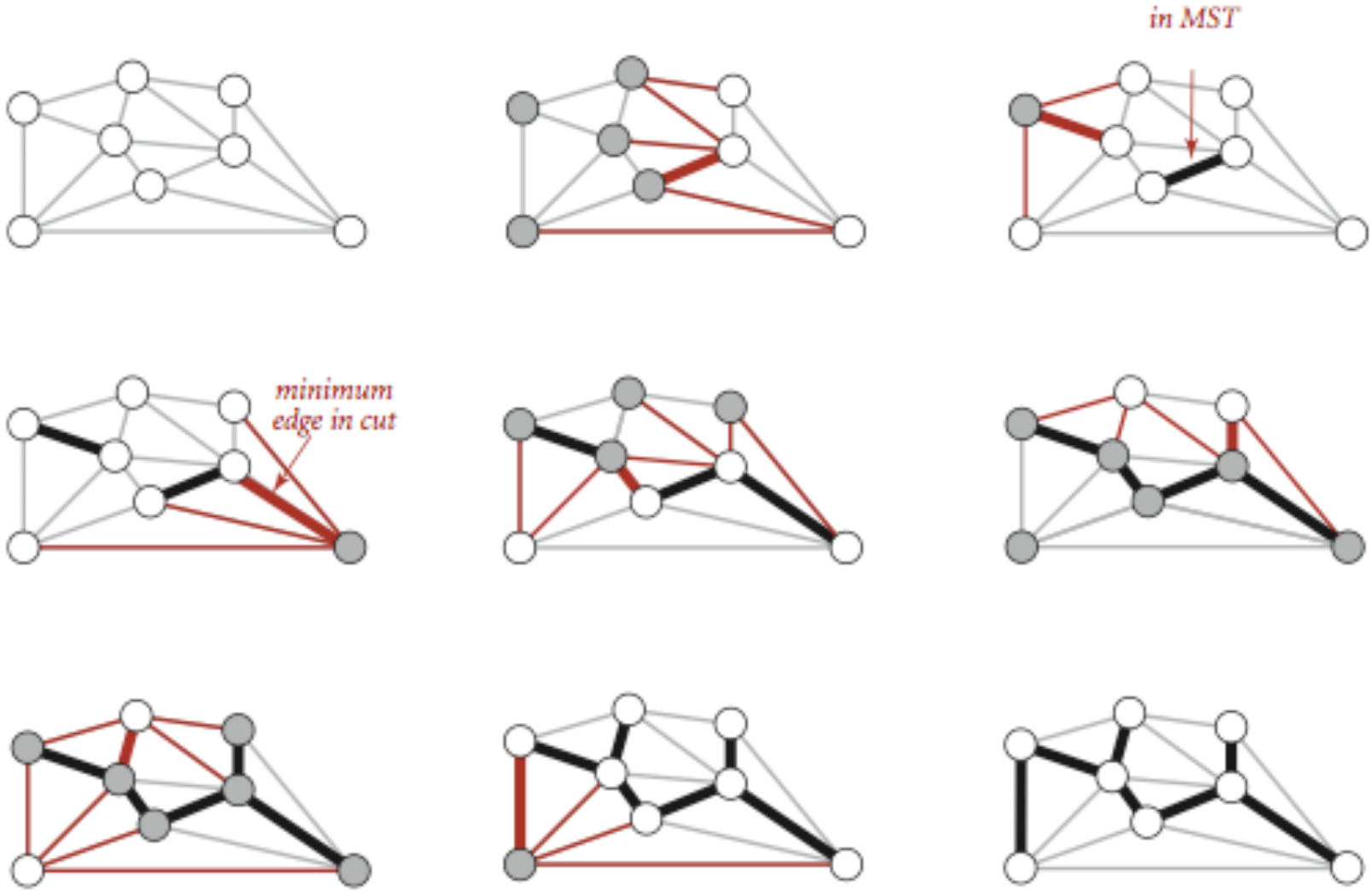


The cut property is the basis for the algorithms that we consider for the MST problem. Specifically, they are special cases of the greedy algorithm.

A *greedy* MST-algorithm

Proposition. (Greedy MST algorithm) The following method colors black all edges in the the MST of any connected edge-weighted 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.

The algorithm



API

Edge-weighted graph data type. We represent the weighted edges using the following API:

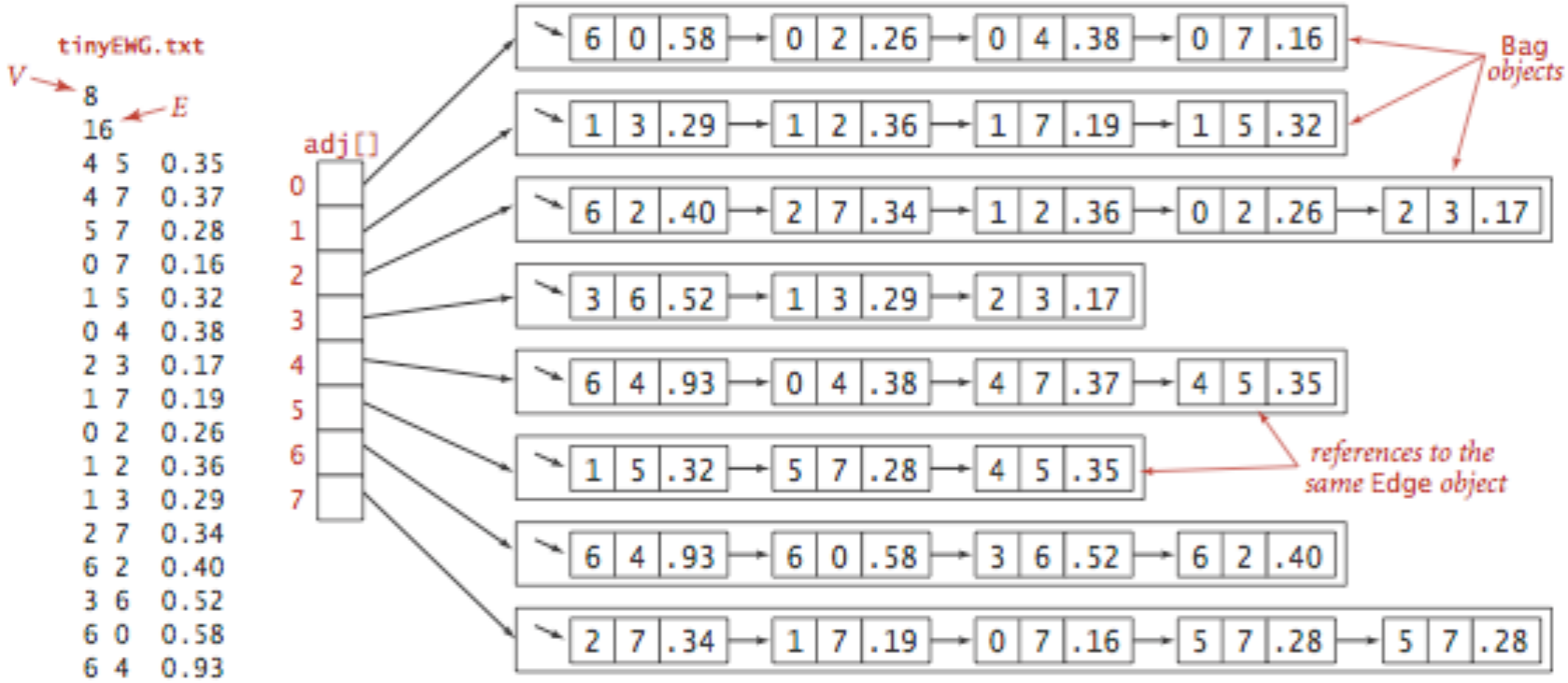
public class Edge implements Comparable<Edge>		
	Edge(int v, int w, double weight)	*initializing constructor*
double	weight()	*weight of this edge*
int	either()	*either of this edge's vertices*
int	other(int v)	*the other vertex*
int	compareTo(Edge that)	*compare this edge to* e
String	toString()	*string representation*

The either() and other() methods are useful for accessing the edge's vertices; the compareTo() method compares edges by weight. Edge.java is a straightforward implementation.

We represent edge-weighted graphs using the following API:

public class	EdgeWeightedGraph	
	EdgeWeightedGraph(int V)	*create an empty V-vertex graph*
	EdgeWeightedGraph(In in)	*read graph from input stream*
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(Edge e)	*add edge e to this graph*
Iterable<Edge>	adj(int v)	*edges incident to v*
Iterable<Edge>	edges()	*all of this graph's edges*
String	toString()	*string representation*

We allow parallel edges and self-loops. EdgeWeightedGraph.java implements the API using the adjacency-lists representation.



Edge-weighted graph representation

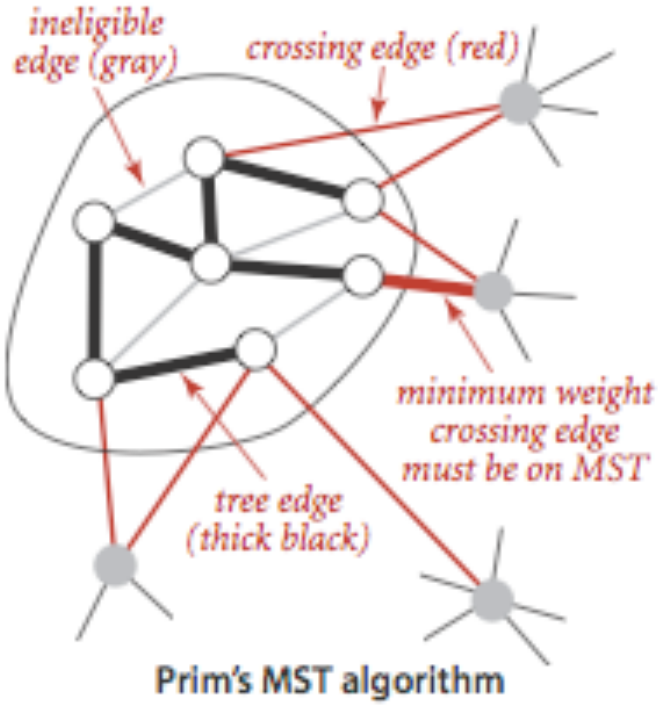
MST API. We use the following API for computing an MST of an edge-weighted graph:

public class	MST	
	MST(EdgeWeightedGraph G)	*constructor*
Iterable<Edge>	edges()	*iterator for MST edges*
double	weight()	*weight of MST*

API for MST implementations

Prim’s algoritm

Prim's algorithm. Prim's algorithm works by attaching a new edge to a single growing tree at each step: Start with any vertex as a single-vertex tree; then add V-1 edges to it, always taking next (coloring black) the minimum-weight edge that connects a vertex on the tree to a vertex not yet on the tree (a crossing edge for the cut defined by tree vertices).

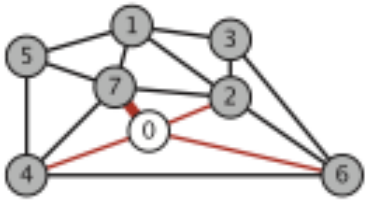


Prim’s MST algorithm

An implementation

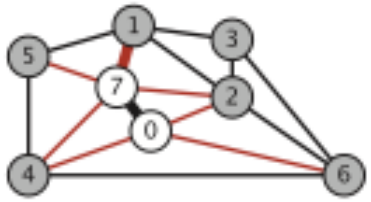
The one-sentence description of Prim's algorithm leaves unanswered a key question: How do we (efficiently) find the crossing edge of minimal weight?

- Lazy implementation. We use a priority queue to hold the crossing edges and find one of minimal weight. Each time that we add an edge to the tree, we also add a vertex to the tree. To maintain the set of crossing edges, we need to add to the priority queue all edges from that vertex to any non-tree vertex. But we must do more: any edge connecting the vertex just added to a tree vertex that is already on the priority queue now becomes ineligible (it is no longer a crossing edge because it connects two tree vertices). The lazy implementation leaves such edges on the priority queue, deferring the ineligibility test to when we remove them. LazyPrimMST.java is an implementation of this lazy approach. It relies on the MinPQ.java priority queue.

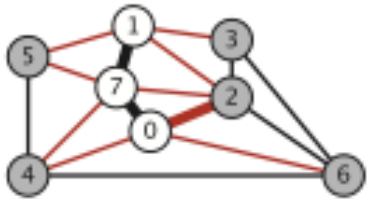


0-7 0.16
0-2 0.26
0-4 0.38
6-0 0.58

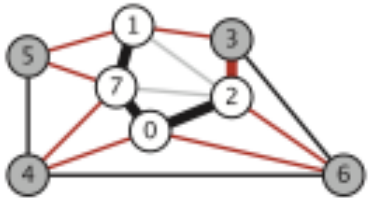
edges with exactly one endpoint in T (sorted by weight)



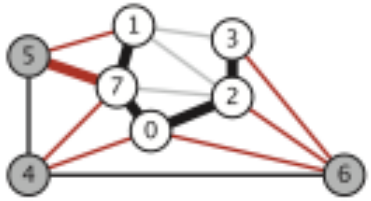
1-7 0.19
0-2 0.26
5-7 0.28
2-7 0.34
4-7 0.37
0-4 0.38
6-0 0.58



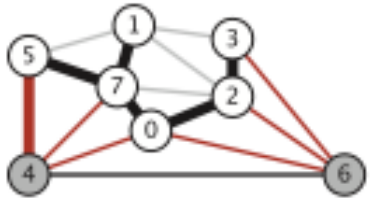
0-2 0.26
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
1-2 0.36
4-7 0.37
0-4 0.38
0-6 0.58



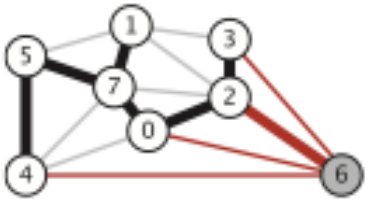
2-3 0.17
5-7 0.28
1-3 0.29
1-5 0.32
4-7 0.37
0-4 0.38
6-2 0.40
6-0 0.58



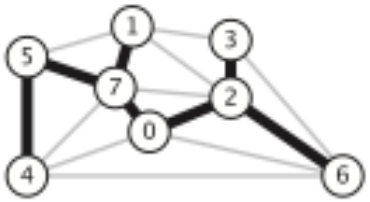
5-7 0.28
1-5 0.32
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58



4-5 0.35
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58

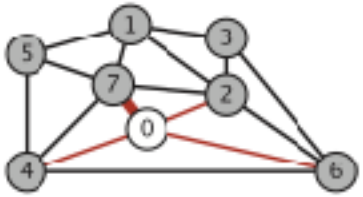


6-2 0.40
3-6 0.52
6-0 0.58
6-4 0.93



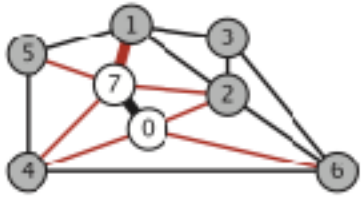
Another implementation

Eager implementation. To improve the lazy implementation of Prim's algorithm, we might try to delete ineligible edges from the priority queue, so that the priority queue contains only the crossing edges. But we can eliminate even more edges. The key is to note that our only interest is in the minimal edge from each non-tree vertex to a tree vertex. When we add a vertex v to the tree, the only possible change with respect to each non-tree vertex w is that adding v brings w closer than before to the tree. In short, we do not need to keep on the priority queue all of the edges from w to vertices tree—we just need to keep track of the minimum-weight edge and check whether the addition of v to the tree necessitates that we update that minimum (because of an edge v-w that has lower weight), which we can do as we process each edge in s adjacency list. In other words, we maintain on the priority queue just one edge for each non-tree vertex: the shortest edge that connects it to the tree.

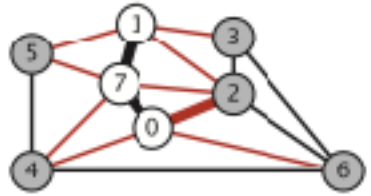


* 0-7 0.16
* 0-2 0.26
* 0-4 0.38
* 6-0 0.58

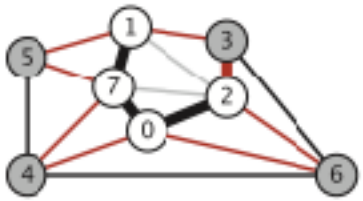
** marks new priority queue entry*



* 1-7 0.19
0-2 0.26
* 5-7 0.28
* 2-7 0.34
* 4-7 0.37
0-4 0.38
6-0 0.58

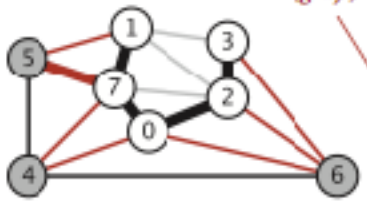


0-2 0.26
5-7 0.28
* 1-3 0.29
* 1-5 0.32
2-7 0.34
* 1-2 0.36
4-7 0.37
0-4 0.38
0-6 0.58

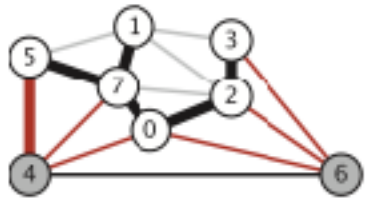


* 2-3 0.17
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
1-2 0.36
4-7 0.37
0-4 0.38
* 6-2 0.40
6-0 0.58

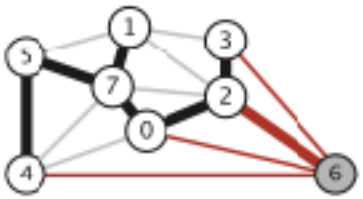
obsolete edges (gray)



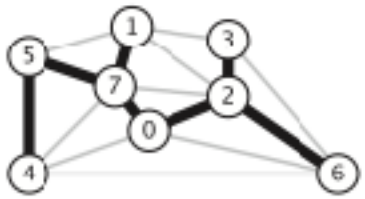
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
* 3-6 0.52
6-0 0.58



1-3 0.29
1-5 0.32
2-7 0.34
* 4-5 0.35
1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58



1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58
* 6-4 0.93



3-6 0.52
6-0 0.58
6-4 0.93

Proposition. Prim's algorithm computes the MST of any connected edge-weighted graph. The lazy version of Prim's algorithm uses space proportional to E and time proportional to E log E (in the worst case) to compute the MST of a connected edge-weighted graph with E edges and V vertices; the eager version uses space proportional to V and time proportional to E log V (in the worst case).

Pseudocode for Prim

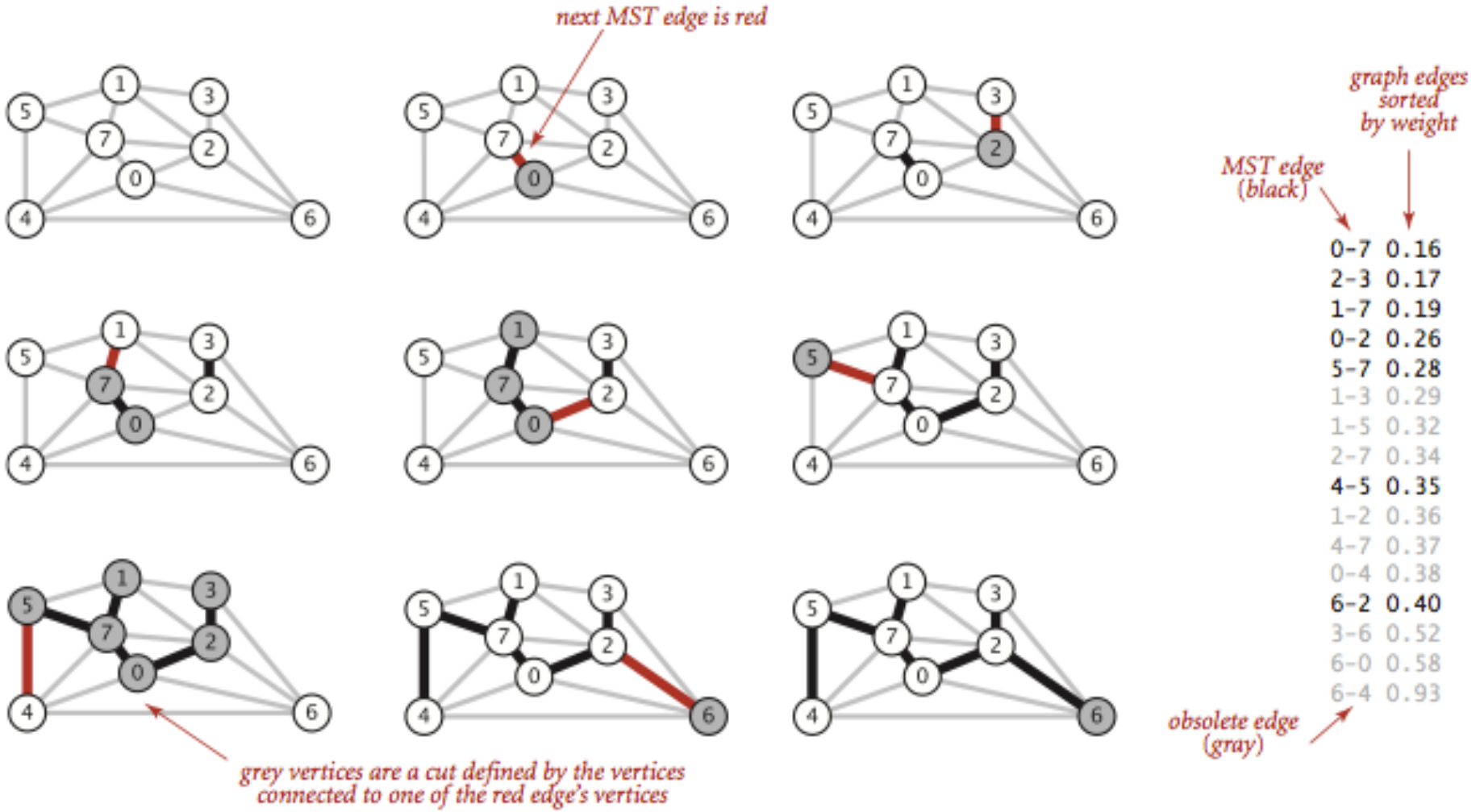
```
Prim(V, E, w, s)
foreach v∈V
    key[v]←∞
key[s] ← 0

Q ← MakeHeap(V,key)
π[s] ← Null
while Q≠∅
        u ← HeapExtractMin(Q)
        foreach neighbor v to u
                if v ∈ Q and w(u, v) < key[v]
                        π[v] ← u
                        key[v] ← w(u, v)
                        Adjust the heap at v
```

Time complexity: O(|V | log |V |) + O(|E| log |V |) = O(|E| log |V |)

Kruskal’s algorithm

Kruskal's algorithm. Kruskal's algorithm processes the edges in order of their weight values (smallest to largest), taking for the MST (coloring black) each edge that does not form a cycle with edges previously added, stopping after adding V-1 edges. The black edges form a forest of trees that evolves gradually into a single tree, the MST.


next MST edge is red
graph edges sorted by weight
MST edge (black)
0-7 0.16
2-3 0.17
1-7 0.19
0-2 0.26
5-7 0.28
1-3 0.29
1-5 0.32
2-7 0.34
4-5 0.35
1-2 0.36
4-7 0.37
0-4 0.38
6-2 0.40
3-6 0.52
6-0 0.58
6-4 0.93
obsolete edge (gray)
grey vertices are a cut defined by the vertices connected to one of the red edge's vertices

To implement Kruskal's algorithm, we use a priority queue to consider the edges in order by weight, a union-find data structure to identify those that cause cycles, and a queue to collect the MST edges. Program KruskalMST.java implements Kruskal's algorithm along these lines. It uses the helper MinPQ.java, UF.java, and Queue.java data types.

Proposition. Kruskal's algorithm computes the MST of any connected edge-weighted graph with E edges and V vertices using extra space proportional to E and time proportional to E log E (in the worst case).

Pseudocode for Kruskal

The idea is to build a spanning forrest. We add edges and make sure that we do not get any cycles.

Time complexity: O(|E| log |E|) (Sorting); FindSet and MakeUnion takes O(log |V |).

```
Kruskal(V, E, w)
A←∅
foreach v ∈ V
    MakeSet(v)

Sort E in increasing weight order

foreach (u, v) ∈ E (in increasing weight order)
if FindSet(u)/= Find Set(v)
    A ← A ∪ {(u, v)}
    MakeUnion(u, v)
return A
```

Correctness

Proof of correctness: We use the following lemma: Assume that all edges have different weight. A cut S in G is a partitioning $V = S \cup (V - S)$.

Theorem.

Let S, V −S be a cut (none of S or V −S is empty). Let e = (u,v) be an edge from S till V-S, u $\in$ S and v $\in$/ S and such that e : s weight is minimal. Then e must be in every MST in G.

Proof: Let T be a MST that does not contain (u, v). There is a path from u to v in T . The path contains some edge (x, y) going from S to V − S. Since (u, v) has least weight of all such edges we see that T + {(u, v)} − {(x, y)} is a MST with lower weight, which is impossible.

Kruskal: Let (u,v) be an edge chosen at any stage in Kruskal. Let S be all nodes that can be reached from u be paths using edges already chosen. Then u $\in$ S and v $\notin$ S. We can see that (u,v) must be an edge of minimal weight cross the cut. The theorem says that this edge must be in all MST:s. Then the choice of (u,v) can not be wrong.

Prim: Let S be the nodes chosen at a certain stage. Let (u,v) be the edge chosen in the next step.Then the theorem says that this edge must be in all MST:s. Then the choice of (u,v) can not be wrong.

Greedy Algorithms

Prim and Kruskal's algorithms are examples of something called greedy algorithms. They are algorithms that at some stages of the algorithms do very simple choices that in the short term seem good.

In Prim's and Kruskal's algorithms the greedy choice is to "Take the lightest edge"

We look at a few other simple examples.

Greedy Algorithms

This type of algorithms work by the following principle: Make every situation that is locally optimal, that is, best for the moment. Some examples we have already seen is Dijkstra's algorithm. There, we extend a set S with the corners marked with the lowest value.
For many problems is a natural greedy algorithm, but often it does not provide an optimal solution. Greedy algorithms are usually fast so they can be used to quickly obtain an approximate solution.
We will now study the resulting problems linked to activity planning

Example: Job Planning

We have n number of activities with given start times and end times s_i f_i. This means that the activity must be performed during the time $[s_i, f_i]$ (full intervall needed). How many people are required to carry out all the activities of a person can not be occupied by more than one task at a time?
Observation: Let us assume that there is a time x such that x is in d ranges of various activities and d is the maximum in this regard. Then at least d people are required to perform the job.

Claim: The activities are possible with the d people.
We use a simple greedy algorithm to solve the problem. We d persons P1, P2, ..., Pd. We sort intervals so that $s1 \leq s2 \leq \cdots \leq sn$. Take the first activity and let P1 perform the activity. In the future, we do the following: When the time s for a new event pops up, we let the first person Pj that is not busy do it.
It is easy to show that the d people together capable of performing the activities. The greedy feature of the algorithm is to just go through the specificities activity in chronological order and assign them to people without doing any more forward-looking planning (that can not pay off).

Job Planning for one person

We have n number activities a1, a2, ..., an taking place during the time interval of the form $[s_i, f_i)$. No ranges may overlap. (They are defined as half open. That means eg. that [2,4) and [4,5) do not overlap.) We want to select the maximum number of tasks to perform. How do you choose a maximal number of activities that do not overlap?

A greedy solution

Surprisingly, it turns out that it is the end times for the activities which are important. Sort so that f1 ≤ f2 ≤ ... ≤ fn. We want to choose a set A of activities that we carry out.

$A \leftarrow \{a_1\}$

$i \leftarrow 1$

for $j \leftarrow 2$ to n

  if $s_j \geq f_i$

    $A \leftarrow A \cup \{a_j\}$

    $i \leftarrow j$

return A

The greedy in this algorithm is that we continually choose the idle interval that ends first. It is quite easy to show that the algorithm provides the optimal solution.

When does a greedy algorithm work?

A greedy algorithm is suitable in situations where you can perform the following reasoning:

1. The problem can be divided into a first greedy choice which leaves us with a new partial problem of the same kind.

2. It is possible to show that if there is an optimal solution L to the problem, then it is always possible to modify L to a new optimal solution L’ having the same first choice as our choice. ("The first choice can not be wrong”)

3. We can use induction to demonstrate that our algorithm gives a correct results. The optimal solution to subproblem is always possible to combine with the greedy choice. So the optimal solution is equivalent to a sequence of greedy choices.