Lecture given at the International Summer School
Modern Computational Science 2012 – Optimization
University of Oldenburg, Germany, August 20-31, 2012

Minimum weight spanning trees of weighted scale free networks

Oliver Melchert
Institute of Physics
Faculty of Mathematics and Science
Carl von Ossietzky Universität Oldenburg
D-26111 Oldenburg
Germany

Abstract. In this lecture we will consider the minimum weight spanning tree (MST) problem, i.e., one of the simplest and most vital combinatorial optimization problems. We will discuss a particular greedy algorithm that allows to compute a MST for undirected weighted graphs, namely *Kruskal's* algorithm, and we will study the structure of MSTs obtained for weighted scale free random graphs. This is meant to clarify whether the structure of MSTs is sensitive to correlations between edge weights and topology of the underlying scale free graphs.

The lecture is supplemented by a set of `Python` scripts that allow you to reproduce figures 4 and 5 shown in the course of the lecture. The supplementary material can be obtained from the MCS homepage (see Ref. [1]).

Contents

1 Recap: Nodes, Edges, and Graphs

Nodes: A *node set* V is a collection of elements i, termed *nodes* (also called *sites* or *vertices*). The number of nodes in a node set is subsequently referred to as $N = |V|$.

Edges: An *edge* (or *arc*) e_{ij} consists of a pair of nodes $i, j \in V$. Edges can either be directed or undirected. In the former case the corresponding node pair is ordered, i.e., $e_{ij} = (i, j)$ with $(i, j) \neq (j, i)$, while in the latter case the node pair is unordered, i.e., $e_{ij} = \{i, j\}$ with $\{i, j\} \equiv \{j, i\}$. In the following, if not stated otherwise, the term edge will always refer to an *undirected edge*, i.e., $e_{ij} = \{i, j\}$. The edge e_{ij} is said to be *incident* with the nodes i and j and it *joins* both nodes. Two distinct nodes are said to be *adjacent*, if they are incident with the same edge. Similarly, two distinct edges are adjacent, if they have a node in common.

Consequently, an *edge set* E is a collection of elements e_{ij} with $i, j \in V$. In what follows, E is not allowed to contain *self-edges*, i.e., edges of type e_{ii}, or multiple parallel edges. The number of edges in an edge set is referred to as $M = |E|$.

Graphs: A *graph* $G = (V, E)$ is a tuple that consists of a node set V and an edge set E. Depending on the characteristics of the edge set, a graph can either be directed or undirected (see Fig. 1(a),(b)). In the following, if not stated otherwise, the term graph will always refer to an *undirected graph*.

Within a graph, a particular node can have several adjacent nodes, given by the set $\mathrm{adj}(i) = \{j \mid e_{ij} \in E\}$. In this context, the *degree* $d(i) = |\mathrm{adj}(i)|$ of node i measures the number of its neighbors. Regarding directed graphs, one might distinguish between the *in-* and *out*-degree of a node, e.g. node 3 of the directed graph in Fig. 1(b) has $d_{\mathrm{out}}(3) = 1$ and $d_{\mathrm{in}}(3) = 2$ while the corresponding node in the undirected graph (Fig. 1(a)) simply has $d(3) = 2$.

Further, *mappings* can be used to relate additional information to the graph. Therefore, nodes as well as edges can be addressed. E.g., a *weight function* $\omega \colon E \to \mathbb{R}$ can be used in order to assign a certain weight $\omega_{ij} \equiv \omega(e_{ij})$ to each edge $e_{ij} \in E$, see Fig. 2. Such a weight might be interpreted as distance between the two nodes or as a cost to get from one node to the other. The triple $G = (V, E, \omega)$ is then called a *weighted graph*.

A *subgraph* $G_{\mathrm{sub}} = (V_{\mathrm{sub}}, E_{\mathrm{sub}})$ is obtained from a graph G by deleting a (possibly empty) subset of its nodes and edges. That means, for a subgraph it holds that $V_{\mathrm{sub}} \subseteq V$ and $E_{\mathrm{sub}} \subseteq E$.

A *cut* $(C, V \setminus C)$ on an undirected graph $G = (V, E)$ is a partition of its nodeset. An edge $\{i, j\} \in E$ *crosses* the cut if $i \in C$ and $j \in V \setminus C$ (or vice versa). A cut

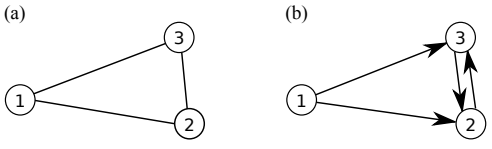

Figure 1: Directed and undirected example graphs. (a) undirected graph with node set $V = \{1, 2, 3\}$ and edge set $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$, (b) directed graph with node set $V = \{1, 2, 3\}$ and edge set $E = \{(1, 2), (1, 3), (2, 3), (3, 2)\}$.

respects a subset $E_{\text{sub}} \subseteq E$ if there is no edge in E_{sub} that crosses the cut. An edge $\{i, j\}$ is called *candidate* regarding the cut $(C, V \setminus C)$ if it crosses the cut and has minimum weight amongst all edges that cross the cut. As an example consider the cut $(C = \{0, 1, 2\}, V \setminus C = \{3\})$ (the nodes in the set C are colored in grey) in Fig. 2(a). Therein, the edges $\{1, 3\}$ and $\{2, 3\}$ cross the cut. Amongst those two, $\{2, 3\}$ signifies the candidate edge since it has the smaller weight.

2 Minimum weight spanning trees (MSTs)

Given an undirected, connected and weighted graph $G = (V, E, \omega)$ where $N = |V|$ and $M = |E|$ signify the number of nodes and edges of G, respectively, and where ω assigns a weight to each edge. Compute a "minimum weight spanning tree" T [2–4].

Minimum weight spanning tree (MST): A MST T is a connected, loopless subgraph of G, consisting of $(N-1)$ edges, connecting all N nodes, thereby minimizing the sum of the edge weights. Note that there are several ways to represent a MST. Here, we will represent it by means of the set T of edges from which it is build up. For the small undirected graph shown in Fig. 2(b), the MST (indicated by bold black lines) reads $T = \{\{0, 1\}, \{1, 2\}, \{2, 3\}\}$ and its corresponding weight is $\omega_T = \sum_{\{i,j\} \in T} \omega_{ij} = 3$. Depending on the precise topology of G and the distribution of edgeweights, T is not necessarily unique.

Relevance of the MST problem: The minimum weight spanning tree (MST) problem is one of the simplest and most vital combinatorial optimization problems. As such, it arises in a vast number of applications. Either as a "standalone" problem or as subtask of a more intricate problem. The generic problem that is solved by an MST algorithm reads: "Connect a set of points using a minimum-weight set of edges". As such, broadcasting problems on networks typically relate to finding the MST on an appropriate weighted graph. E.g., consider a network where the edgeweights signify time delays (or transmission costs) between the respective incident nodes. Then, an "efficient" or "optimal" broadcasting of a message is made sure if the message is transmitted along the MST edges. The latter connects all nodes by using as few edges as possible, thereby minimizing the total time delay (or transmission cost). Further, MSTs find application in the context of single linkage cluster analyses [5].

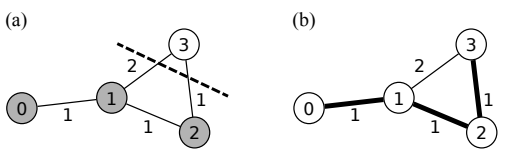

Figure 2: (a) The dashed line illustrates the cut $(\{0, 1, 2\}, \{3\})$ (for more details, see text). (b) example of a MST (bold black lines) for a small undirected weighted graph.

3 Computing MSTs via a "greedy" strategy

Greedy problem solving strategy: At any time (during the solution procedure for a given problem) where a decision is required, a *greedy* problem solving strategy makes the locally optimal choice. Further, unlike in *backtracking* approaches, once a decision is made it is not revised later on. Albeit such an approach might fail to find a globally optimal solution for your particular optimization problem, an optimal solution for the MST problem can very well be obtained via efficient greedy algorithms. A generic greedy approach to compute a MST for a given network G reads:

algorithm MST_greedy(G)

1: $T = \{\}$
2: **while** T is not a MST **do**
3: pick feasible edge $e \in E \setminus T$
4: $T \leftarrow T \cup e$
5: **end while**
6: return MST T

Cut-property: In the above pseudocode the tricky part is to find proper selection criteria for the edges that are added to the MST. In this regard, a feasible edge has to satisfy the *cut property*: Consider a weighted undirected graph $G = (V, E, \omega)$. Let $T' \subseteq E$ so that there is a MST $T \supset T'$. Further, let $(C, V \setminus C)$ be an arbitrary cut that respects T' and let $\{i, j\}$ be a candidate regarding the cut. Then $\{i, j\}$ is a feasible edge that can be used in order to extend T'. E.g., regarding the cut illustrated in Fig. 2(a), the edge $\{2, 3\}$ is a feasible edge that can be used to amend (and also complete) the MST.

4 Kruskal's algorithm

Idea behind the algorithm: The algorithm due to Kruskal [2] is a particular greedy algorithm that, for an instance of an undirected, connected and weighted graph, obtains a MST. The subsequent three steps summarize the solution strategy of Kruskal's algorithm:

Step 1: In the beginning, each node forms a tree. Hence, there is a forest of N single-node trees. Further, an empty list T is initialized that will keep all those edges that are part of the MST.

Step 2: Sort the edges in order of increasing weight. Visit the edges in order of increasing weight and apply the following edge-selection criterion:

Step 3: If the currently considered edge $\{i, j\}$ has both endnodes in different trees, add the edge to T and merge the respective trees. If $\{i, j\}$ has both endnodes in the same tree, discard the edge. Proceed to the next edge or stop when there are no more edges to process.

After M iterations of step 3 (i.e., as soon as all edges are processed), T comprises the edges that comprise the MST.

Efficient implementation: Albeit the pseudocode and the description above seem to be very simple, an efficient implementation of Kruskal's algorithm is more tricky than it appears at first sight. In this regard, step 2 can be accomplished by using merge-sort yielding a sorted edge list in time $O(M \log M)$. Further, in step 1 single-node trees (one tree for each node in the graph) need to be initialized, and in step 3 it must be possible to efficiently determine the tree to which a node belongs. Finally, two selected trees need to be merged to a single tree, occasionally. These latter three tasks can be handled by using a *union-find* data structure that features the following three operations:

Operation 1: `make_set(i)`: Generates a tree consisting of the node i, only

Operation 2: `find(i)`: Yields the "name" of the tree to which node i belongs

Operation 3: `union(a,b)`: Merges trees with names a and b to a new tree with name a

For a connected graph, the total time for the union-find operations (i.e., N `make_set` operations and $O(M)$ `find` and `union` operations) can be approximated by $O(M \log M)$ (note that this also depends on the precise implementation of the union-find data structure [3]). The full (worst case) running time of Kruskal's algorithm amounts to $O(M \log N)$. It summarizes the time spent to sort the edges, perform M find and $N-1$ union operations. A `python` [6] implementation of Kruskal's algorithm is contained in the supplementary material [1]. Generically, `python` uses *Timsort* [7], a hybrid sorting algorithm based on *merge sort* and *insertion sort* [3]. Further, the implementation of the union-find data structure contained in the supplementary material uses *union-by-size*: if two trees are merged by means of a call to `union`, the smaller tree (in terms of the number of nodes contained in the tree) is added to the larger tree. A particular implementation might read (see [3]):

```
def mstKruskal(G):
        """Kruskals minimum spanning tree algorithm

        algorithm for computing a minimum spanning
        tree (MST) T=(V,E') for a connected, undirected
        and weighted graph G=(V,E,w) as explained in
        'Introduction to Algorithms',
        Cormen, Leiserson, Rivest, Stein,
        Chapter 23.2 on 'The algorithms of Kruskal and Prim'

        Input:
        G       - weighted graph data structure

        Returns: (T,wgt)
```

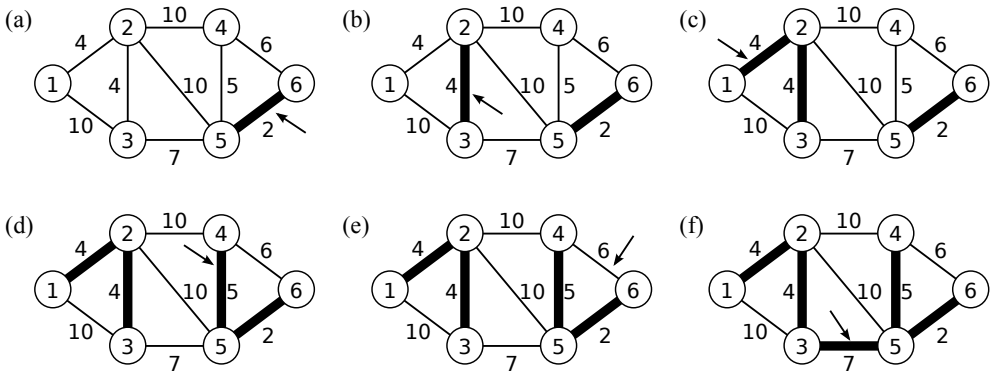

Figure 3: Exemplary application of Kruskal's MST algorithm on a small example graph G consisting of 6 nodes and 9 edges. The bold black edges belong to the forest that is grown in order to construct an MST of G. In the course of the algorithm each edge is considered once (within the subfigures (a–f), a small arrow indicates the edge currently considered). For each edge it is decided whether it can be used to extend the forest constructed so far in order to yield an MST. For more details on the steps (a) through (f), see text.

```
T       - minimum spanning tree stored as edge list
wgt     - weight of minimum weight spanning tree
"""
uf = unionFind_cls()    # union find data structure
T=[]                    # list to store MST edges

# list of edges sorted in order of increasing weight
K = sorted(G.E,cmp=lambda e1,e2: cmp(G.wgt(e1),G.wgt(e2)))

# initialize forrest of sinlge-node trees
for i in G.V:
        uf.makeSet(i)

# construct MST
for (v,w) in K:
        if uf.find(v)!=uf.find(w):
                uf.union(uf.find(v),uf.find(w))
                T.append((v,w))

return T, sum(map(lambda e: G.wgt(e),T))
```

Example graph: In order to illustrate Kruskal's MST algorithm, consider the small graph $G = (V, E, \omega)$ consisting of $N = 6$ nodes and $M = 9$ edges, shown in Fig. 3(a–f). In a first step, the graph edges are sorted in order of increasing weight, i.e., the

set K reads:

$$K = \{\{5,6\},\{2,3\},\{1,2\},\{4,5\},\{4,6\},\{3,5\},\{1,3\},\{2,5\},\{2,4\}\}$$

Then, as a second step, the function `make_set` is used to initialize a forest of trees. In the very beginning, each tree consists of a single node, only. In the third step, the edge selection procedure is carried out until all edges are processed. In detail, the following steps (illustrated in Fig. 3(a–f)) are carried out:

Step (a): Edge $\{5,6\}$ is considered. It can be used to merge two distinct trees in the forest and is hence added to T.

Step (b): Edge $\{2,3\}$ is considered. It can be used to merge two distinct trees in the forest and is hence added to T.

Step (c): Edge $\{1,2\}$ is considered. It can be used to merge two distinct trees in the forest and is hence added to T. Note that $\{2,3\}$ and $\{1,2\}$ had the same weight. However, no matter which of the two edges is picked first does not alter the structure of the final MST.

Step (d): Edge $\{4,5\}$ is considered and added to T.

Step (e): Edge $\{4,6\}$ is considered. It does not connect two distinct trees. Adding it to the forest would introduce a cycle. Hence, the edge is not added to T and the algorithm proceeds to the next edge in K.

Step (f): Edge $\{3,5\}$ is considered and added to T.

All remaining edges would also lead to a cycle (as in step (e) above) and are hence not added to T. Finally, after the algorithm terminates, the edgeset T comprises a MST with weight $\omega_T = 22$.

5 MSTs of weighted scale free networks

The subsequent simulations, reported in Ref. [8], were carried out to clarify whether the structure of MSTs change as a function of the correlations between edge weights and network topology. (Here, as an exercise, I re-implemented the model studied in Ref. [8] and re-performed their simulations and analysis. The resulting code is available as supplementary material at [1]. The figures presented below summarize the results of the simulations performed via the code in the supplementary material.)

Simulation setup: For the numerical experiments scale free (SF) random networks were considered. These networks (also referred to as heterogeneous random graphs) are characterized by a power-law degree distribution $p_k \sim k^{-\gamma}$. The construction of such SF networks via preferential attachment [9] yields an exponent $\gamma = 3$. In particular, SF networks containing $N = 10000$ nodes were used (the number of edges

connecting a newly created node to existing nodes during the preferential attachment procedure was set to $m = 2$).

Further, two qualitatively different weight distributions were considered (note that the authors of Ref. [8] considered more weight distributions, however, the two weight distributions considered here suffice in order to answer the question that kicked off the study):

Disorder type 1:

$$\omega_{ij} = k_i k_j, \tag{1}$$

where the weight associated to edge $\{i, j\}$ is large if the degrees of its endnodes tend to be large, and,

Disorder type 2:

$$\omega_{ij} = 1/k_i k_j, \tag{2}$$

where the weight related to an edge $\{i, j\}$ is large if the degrees of its endnodes tend to be small.

For SF networks respecting the above two weight distributions, MSTs are computed using Kruskal's algorithm (in the original article they use a different MST algorithm due to Prim [3]) and the characteristics of the MSTs is studied.

Results: Considering the degree distribution of the nodes regarding the MSTs, the results indicate that the topology of the MSTs falls into two distinct classes:

1. Disorder type 1 (Eq. (1)) yields MST with exponential degree distribution. The MST avoids edges with large weight, preferentially using edges that connect to low degree nodes (see Figs. 5(a) and 4(a)).

2. Disorder type 2 (Eq. (2)) yields MST with power law degree distribution. Edges with lowest weight are connected to the hubs of G. The MST uses these edges extensively (see Figs. 5(b) and 4(b)).

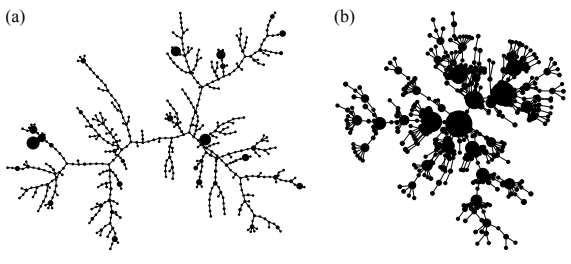

Figure 4: MST of weighted scale free graphs ($N = 500$, $m = 2$). The size of the nodes reflect their degree in G. (a) Disorder type (i): most hubs are located on the outer branches. (b) Disorder type (ii): hubs of G are at the center of the MST, intermediate degree nodes (in G) are located on the branches of the MST.

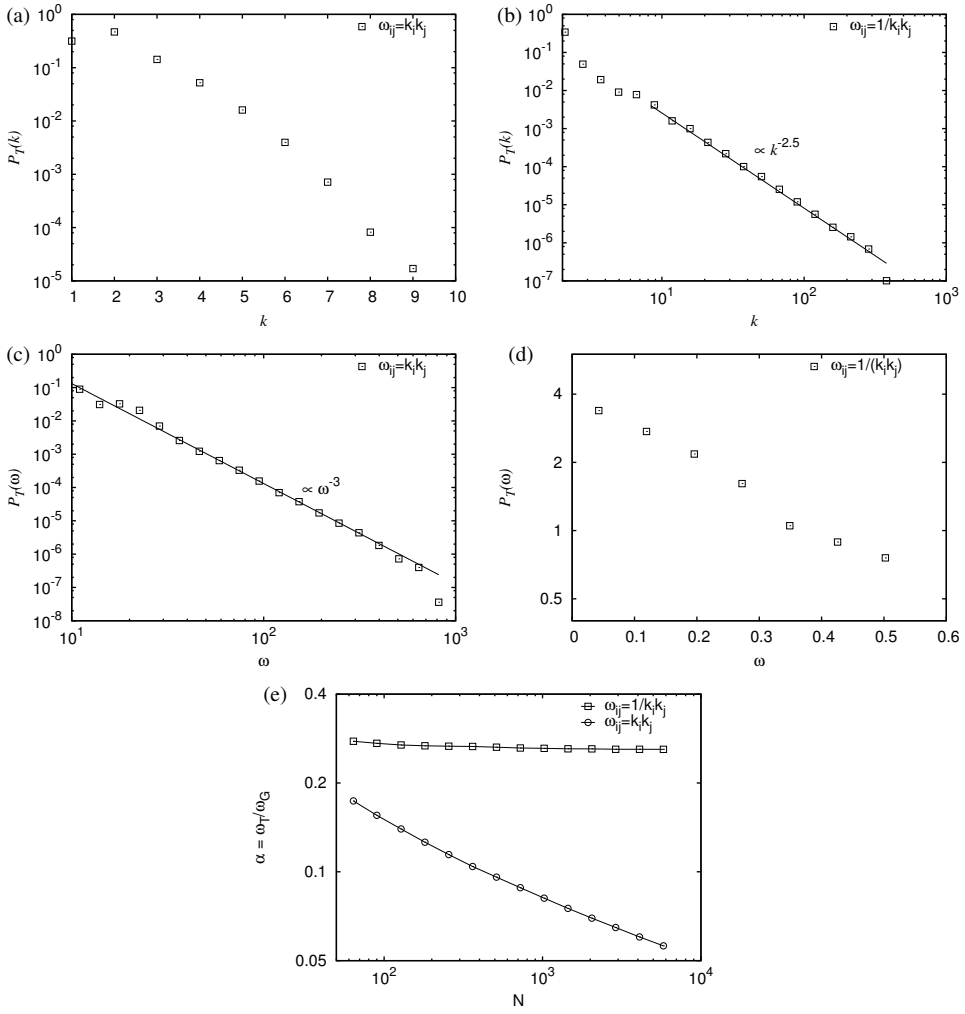

Figure 5: Results for the MSTs computed for weighted scale free graphs ($N = 10000$, $m = 2$). (a) degree distribution for weight assignment $\omega_{ij} = k_i k_j$, (b) degree distribution for weight assignment $\omega_{ij} = 1/k_i k_j$, (c) weight distribution for weight assignment $\omega_{ij} = k_i k_j$, (d) weight distribution for weight assignment $\omega_{ij} = 1/k_i k_j$, (e) MST efficiency.

Considering the weight distribution of the edges that comprise the MSTs, the results show that

1. Disorder type 1 yields MSTs with a power law weight distribution (see Fig. 5(c)).

2. Disorder type 2 yields MSTs with an exponential weight distribution (see Fig. 5(d)).

A measure that tells how efficient a MST connects the nodes of a graph G is given by the MST efficiency $\alpha = \omega_T/\omega_G$. Where the numerator signifies the weight $\omega_T = \sum_{\{i,j\}\in T} \omega_{ij}$ of the MST, and where the denominator specifies the weight of the graph $\omega_G = \sum_{\{i,j\}\in E} \omega_{ij}$. The numerical results regarding the MST efficiency are shown in Fig. 5(e). As evident from the figure, disorder type 1 exhibits a power law decrease of the efficiency with increasing graph size. In contrast to this, the MST efficiency for disorder type 2 saturates at a finite value of α. This suggests that MSTs for disorder type 1 are more efficient than those obtained for disorder type 2.

References

[1] O. Melchert. The supplementary material can be downloaded from the site `http://www.mcs.uni-oldenburg.de/`.

[2] J. B. Kruskal. *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.* Proceedings of the American Mathematical Society, 7 (1956) 48.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd edition.* MIT Press, 2001.

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentics Hall, 1993.

[5] J. C. Gower, and G. J. S. Ross. *Minimum spanning trees and single linkage cluster analysis.* Applied Statistics, 18 (1969) 54.

[6] Python is a high-level general-purpose programming language that can be applied to many different classes of problems, see `http://www.python.org/`.

[7] For more information on the Timsort algorithm, see `http://en.wikipedia.org/wiki/Timsort`.

[8] P. J. McDonald, E. Almaas, and A.-L. Barabási. *Minimum spanning trees of weighted scale-free networks.* Europhys. Lett., 72 (2005) 308.

[9] V. Batagelj, and U. Brandes. *Efficient generation of large random networks.* Phys. Rev. E, 71 (2005) 036113.