Chapter 4

Algorithms in edge-weighted graphs

Recall that an *edge-weighted graph* is a pair (G,w) where $G=(V,E)$ is a graph and $w : E \to \mathbb{R}$ is a *weight function*. Edge-weighted graphs appear as a model for numerous problems where places (cities, computers,...) are linked with links of different weights (distance, cost, throughput,...). Note that a graph can be viewed as an edge-weighted graph where all edges have weight 1.

Let (G,w) be an edge-weighted graph. For any subgraph H of G, the *weight* of H, denoted by $w(H)$, is the sum of all the weights of the edges of H. In particular, if P is a path, $w(P)$ is called the *length* of P. The *distance* between two vertices u and v, denoted by $dist_{G,w}(u,v)$, is the length of a shortest (with minimum length) (u,v)-path.

Observe that $dist_{G,w}$ is a distance: it is *symmetric*, that is, $dist_{G,w}(u,v) = dist_{G,w}(v,u)$, and it satisfies the *triangle inequality*: for any three vertices x, y and z, $dist_{G,w}(x,z) \leq dist_{G,w}(x,y) + dist_{G,w}(y,z)$.

4.1 Computing shortest paths

Given an edge-weighted graph (G,w), one of the main problems is the computation of $dist_G(u,v)$ and finding a shortest (u,v)-path. We have seen in Subsection 2.1.1, that if all the edges have same weight then one can compute a shortest (u,v)-path by running a breadth-first search from u. Unfortunately, this approach fails for general edge-weighted graphs. See Exercise 4.1. We now describe algorithms to solve this problem in general. For this purpose, we solve the following more general problem.

Problem 4.1 (Shortest-paths tree)**.**
Instance: an edge-weighted graph (G,w) and a vertex r.
Find: a subtree T of G such that $\forall x \in V(G), dist_{G,w}(r,x) = dist_{T,w}(r,x)$.

Such a tree is called a *shortest-paths tree*.

4.1.1 Dijkstra's Algorithm

Dijkstra's Algorithm is based on the following principle. Let $S \subset V(G)$ containing r and let $\bar{S} = V(G) \setminus S$. If $P = (r, s_1, \dots, s_k, \bar{s})$ is a shortest path from r to $\bar{S}$, then $s_k \in S$ and P is a shortest path from r to $\bar{s}$. Hence,

$$dist(r, \bar{s}) = dist(r, s_k) + w(s_k \bar{s})$$

and the distance from r to $\bar{S}$ is given by the following formula

$$dist(r, \bar{S}) = \min_{u \in S, v \in \bar{S}} \{dist(r, u) + w(uv)\}$$

To avoid to many calculations during the algorithm, each vertex $v \in V(G)$ is associated to a function $d'(v)$ which is an upper bound on $dist(r, v)$, and to a vertex $p(v)$ which is the potential parent of v in the tree. At each step, we have:

$$\begin{aligned} d'(v) &= dist(r, v) \text{ if } v \in V(T_i) \\ d'(v) &= \min_{u \in V(T_{i-1})} \{dist(r, u) + w(uv)\} \text{ if } v \in \overline{V(T_i)} \end{aligned}$$

Algorithm 4.1 (Dijkstra)**.**

1. Initialize $d'(r) := 0$ and $d'(v) := +\infty$ if $v \neq r$. T_0 is the tree consisting of the single vertex r, $u_0 := r$ and $i := 0$.
2. For any $v \in \overline{V(T_i)}$, if $d'(u_i) + w(u_i v) \leq d'(v)$, then $d'(v) := d'(u_i) + w(u_i v)$ and $p(v) := u_i$.
3. Compute $\min\{d'(v) \mid v \in \overline{V(T_i)}\}$. Let u_{i+1} a vertex for which this minimum is reached. Let T_{i+1} be the tree obtained by adding the vertex u_{i+1} and the edge $p(u_{i+1})u_{i+1}$.
4. If $i = |V| - 1$, return T_i, else $i := i + 1$ and go to Step 2.

Remark 4.2. The algorithm does not work if some weights are negative.

Complexity of Dijkstra's Algorithm: To every vertex is associated a temporary label corresponding to $(d'(v), p(v))$. They are depicted in Figure 4.1. We do

- at most $|E|$ updates of the labels;
- $|V|$ searches for the vertex v for which $d'(v)$ minimum and as many removal of labels.

The complexity depends on the choice of the data structure for storing the labels: if it is a list, the complexity is $O(|E||V| + |V|^2)$. But it can be improved using better data structures. For example, a data structure known as *heap* is commonly used for sorting elements and their

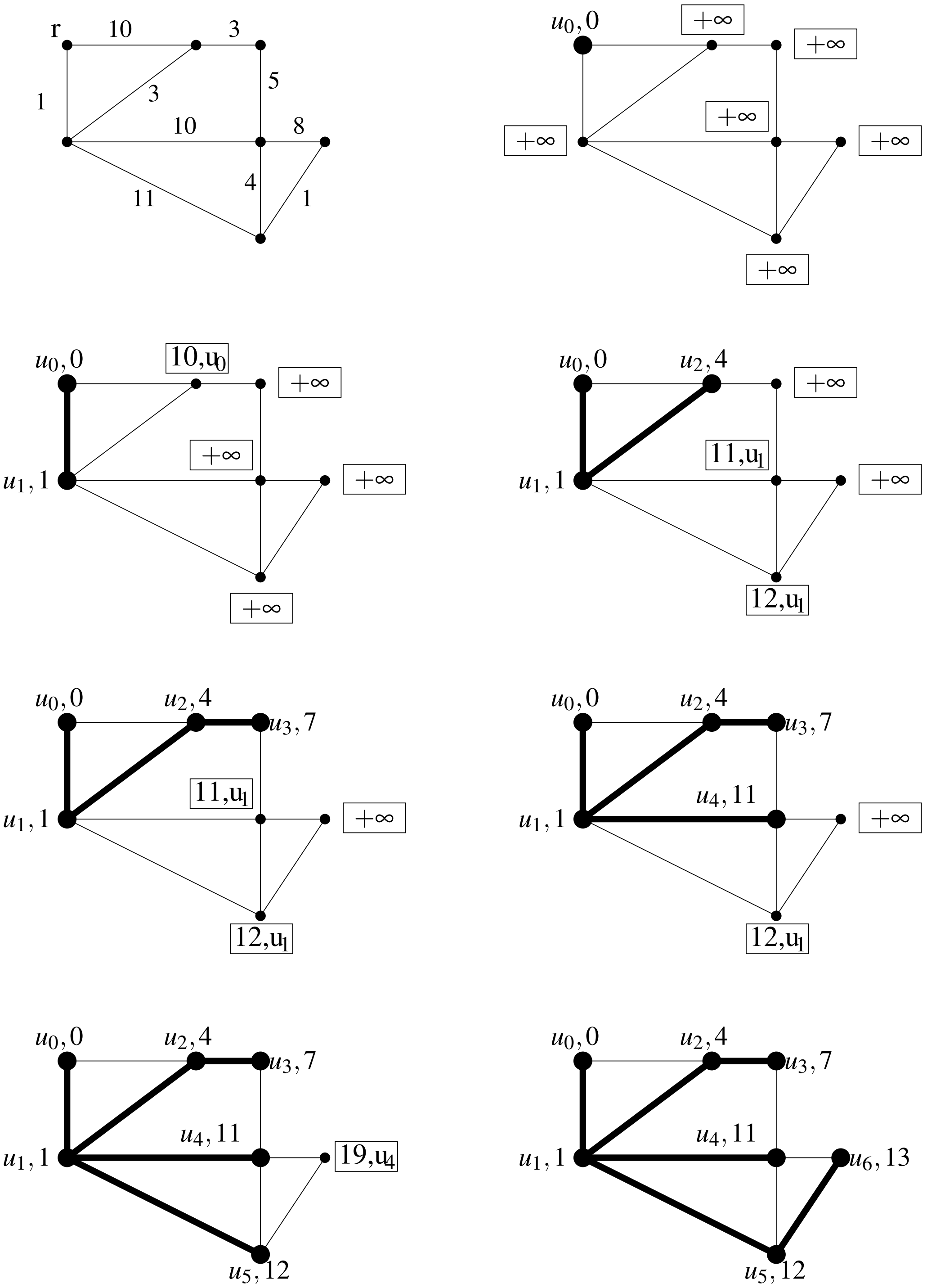


Figure 4.1: A run of Dijkstra's Algorithm on the edge-weighted graph depicted top left. At each step, bold vertices and edges are those of T_i. To each vertex t of T_i is associated its name and the value $d'(t) = dist(r,t)$. Next to each vertex v not in $V(T_i)$ is a box containing the value $d'(v)$ and $p(v)$ if $d'(v) \neq +\infty$.

associated values, called *keys* (such as edges and their weights). A heap is a rooted binary tree T **should we define it?** whose vertices are in one-to-one correspondence with the elements in question (in our case, vertices or edges). The defining property of a heap is that the key of the element located at vertex v of T is required to be at least as small as the keys of the elements located at vertices of the subtree of T rooted at v. This condition implies, in particular, that the key of the element at the root of T is one of smallest value; this element can thus be accessed instantly. Moreover, heaps can be reconstituted rapidly following small modifications such as the addition of an element, the removal of an element, or a change in the value of a key. A *priority queue* is simply a heap equipped with procedures for performing such readjustments rapidly.

Using a priority queue, the complexity of Dijkstra's Algorithm is $O(|E|\log|V| + |V|\log|V|)$.

It should be evident that data structures play a vital role in the efficiency of algorithms. For further information on this topic, we refer the reader to [4, 1, 5, 3].

4.1.2 Bellmann-Ford Algorithm

The algorithm performs n iterations, and gives a label $h(v)$ to any vertex. At iteration i, $h(v)$ is the minimum weight of a path using at most i edges between r and v.

Note that, it always exists a shortest walk using at most $|V(G)|-1$ edges between r and v (otherwise the walk would contain a cycle of negative weight).

Algorithm 4.2 (Bellmann-Ford)**.**

```
1. Initialization : $h(r) := 0$, $h(v) := +\infty, \forall v \neq r$.
2. For $i = 0$ to $|V(G)|-1$ do :
        for all $v \in V(G), h(v) := \min(h(v), \min\{h(u)+w(uv) \mid uv \in E(G)\})$.
3. Return $d(r,v) = h(r,v)$.
```

Complexity of Bellmann-Ford's Algorithm: Each iteration costs $O(|E|)$ (all edges are considered), so the total complexity is $O(|E||V|)$.

The algorithm works even if some edges have negative weight. It can also detect cycles with negative weight. There is such a cycle if and only if, after the $|V|^{\text{th}}$ iteration, the labels h may decrease. Finally, if during an iteration, no $h(v)$ decreases, then $h(v) = d(r,v)$. It is possible to improve the algorithm by continuing the iteration only if $h(v)$ becomes $\min\{h(u)+w(uv) \mid uv \in E(G)\}$ for at least one vertex. The algorithm run in time $O(L|E|)$ where L is the maximum number of edges in a shortest path.

4.2 Minimum-weight spanning tree

Another important problem is the following: given a connected edge-weighted graph, what is the connected spanning subgraph with minimum weight? If all weights are non-negative, since any connected graph has a spanning tree (Corollary 1.10), the problem consists of finding a spanning tree with minimum weight.

Problem 4.3 (Minimum-Weight Spanning Tree)**.**
Instance: a connected edge-weighted graph (G,w).
Find: a spanning tree T of G with minimum weight, i.e. for which $\sum_{e \in T} w(e)$ is minimum.

For $S \subset V(G)$, an edge $e = xy$ is *S-transversal*, if $x \in S$ and $y \notin S$. The algorithms to find a minimum-weight spanning tree are based on the fact that a transversal edge with minimum weight is contained in a minimum-weight spanning tree.

Lemma 4.4. *Let (G,w) be an edge-weighted graph and let $S \subset V$. If $e = s\bar{s}$ is an S-transversal edge with minimum weight, then there is a minimum-weight spanning tree containing e.*

Proof. Let T be a tree that does not contains e. There is a path P between s and $\bar{s}$ in T. At least one edge of P, say e', is S-transversal. Hence, the tree $T' = (T \setminus e') \cup \{e\}$ has weight $w(T') = w(T) + w(e) - w(e') \leq w(T)$ since $w(e) \leq w(e')$. Therefore, if T is a minimum spanning tree, then so does T' and $w(e) = w(e')$. □

In particular, Lemma 4.4 implies that if e is an edge of minimum weight, i.e., $w(e) = \min_{f \in E(G)} w(f) = w_{min}$, then there is a minimum-weight spanning tree containing e.

4.2.1 Jarník-Prim Algorithm

The idea is to grow up the tree T with minimum weight by adding, at each step, a $V(T)$-transversal edge with minimum weight. At each step, E_T is the set of the $V(T)$-transversal edges.

Algorithm 4.3 (Jarník-Prim)**.**

1. Initialize the tree T to any vertex x and E_T is the set of edges incident to x.
2. While $V(T) \neq V(G)$:
   Find an edge $e \in E_T$ with minimum weight. Add e and its end not in T to T. Let E_y be the set of edges incident to y. Remplace E_T by $(E_T \triangle E_y)$.

Complexity of Jarník-Prim Algorithm: During the execution, at most $|E(G)|$ edges are added in E_T, and at most $|E(G)|$ edges are removed. Indeed, an edge e is removed when both its endvertices are in $V(T)$. Since $V(T)$ grows up, e will not be added anymore to E_T. $|V(G)|$ selections of the edge of E_T with minimum weight must be performed. To performs such an algorithm we need a data structure allows the insertion, the removal and the selection of the minimum-weight element efficiently. Using a priority queue, the total complexity of Jarník-Prim Algorithm is $O(|E|\log|E|)$.

4.2.2 Boruvka-Kruskal Algorithm

Boruvka-Kruskal Algorithm is close to Jarník-Prim Algorithm and its correctness also comes from Lemma 4.4. The idea is to start from a spanning forest and to make its number of connected components decreases until a tree is obtained. Initially, the forest has no edges and, at each step, an edge with minimum weight that links two components is added.

For this purpose, we need a fast mechanism allowing to test whether or not u and v are in the same component. A way to do so consists in associating to each connected component the list of all the vertices it contains. To every vertex u is associated a vertex $p(u)$ in the same component. This vertex $p(u)$ is a *representative* of this component. It points to the set $C_{p(u)}$ of vertices of this component and to the size $size(p(u))$ corresponding to the size it.

Algorithm 4.4 (Kruskal)**.**

1. Initalize $T : V(T) := V(G), E(T) := \emptyset$. Order the edges in increasing order of the weights and place them in a stack L; For all $u \in V(G)$, do $p(u) := C_u$ and $size(C_u) := 1$.
2. If $L = \emptyset$, terminate. Else, pull the edge $e = uv$ with minimum weight;
3. If $p(u) = p(v)$ (the vertices are in the same component), then go to 2. Else $p(u) \neq p(v)$, add e in T.
4. If $size(p(u)) \geq size(p(v))$, then $C_{p(u)} := C_{p(u)} \cup C_{p(v)}$, $size(p(u)) := size(p(u)) + size(p(v))$, and for any $w \in C_{p(v)}$, $p(w) := p(u)$.
   Else ($size(p(u)) < size(p(v))$), $C_{p(v)} := C_{p(v)} \cup C_{p(u)}$, $size(p(v)) := size(p(u)) + size(p(v))$, and for any $w \in C_{p(u)}$, $p(w) := p(v)$.
5. Go to 2.

Complexity of Boruvka-Kruskal Algorithm Ordering the edges takes time $O(|E(G)|\log|E(G)|)$. Then, each edge is considered only once and deciding whether the edge must be added to the tree or not takes a constant number of operations.

Now, let us consider the operations used to update the data structure when an edge is inserted in the tree.

We do the union of to sets $C_{p(u)}$ and $C_{p(v)}$. If this sets are represented as lists with a pointor to its last element, it takes a constant time. Such unions are done $|V(G)|-1$ times.

We also have to update the values of some $p(w)$. Let $x \in V(G)$ and let us estimate the number of updates of $p(x)$ during the execution of the algorithm. Observe that when $p(x)$ is updated, the component of x becomes at least twice bigger. Since, at the end, x belongs to a component of size $|V(G)|$, then $p(x)$ is updated at most $\log_2(|V(G)|)$ times. In total, there are at most $|V(G)|\log_2|V(G)|$ such updates.

Since $|V(G)| \leq |E(G)|+1$, the total time complexity is $O(|E(G)|\log|E(G)|)$.

4.2.3 Application to the Travelling Salesman Problem

Rosenkrantz, Sterns and Lewis considered the special case of the Travelling Salesman Problem (3.14) in which the weights satisfy the *triangle inequality*: $w(xy)+w(yz) \geq w(xz)$, for any three vertices x, y and z.

Problem 4.5 (Metric Travelling Salesman)**.**
Instance: an edge-weighted complete graph (G,w) whose weights satisfy the triangle inequality.
Find: a hamiltonian cycle C of G of minimum weight, i.e. such that $\sum_{e\in E(C)} w(e)$ is minimum.

This problem is $\mathcal{NP}$-hard (see Exercise 4.11) but a polynomial-time 2-approximation algorithm using minimum-weight spanning tree exists.

Theorem 4.6 (Rosenkrantz, Sterns and Lewis)**.** *The Metric Travelling Salesman Problem admits a polynomial-time* 2*-approximation algorithm.*

Proof. Applying Jarník-Prim or Boruvka-Kruskal algorithm, we first find a minimum-weight spanning tree T of G. Suppose that C is a minimum-weight hamiltonian cycle. By deleting any edge of C we obtain a hamiltonian path P of G. Because P is a spanning tree, $w(T) \leq w(P) \leq w(C)$.

We now duplicate each edge of T, thereby obtaining a connected eulerian multigraph H with $V(H)=V(G)$ and $w(H)=2w(T)$. The idea is to transform H into a hamiltonian cycle of G, and to do so without increasing its weight. More precisely, we construct a sequence $H_0,H_1,\dots,H_{n-2}$ of connected eulerian multigraphs, each with vertex set $V(G)$, such that $H_0=H$, H_{n-2} is a hamiltonian cycle of G, and $w(H_{i+1}) \leq w(H_i)$, $0 \leq i \leq n-3$. We do so by reducing the number of edges, one at a time, as follows.

Let C_i be an eulerian tour of H_i, where $i < n-2$. The multigraph H_i has $2(n-2)-i > n$ edges, and thus a vertex v has degree at least 4. Let xe_1ve_2y be a segment of the tour C_i; it will follow by induction that $x \neq y$. We replace the edges e_1 and e_2 of C_i by a new edge e of weight $w(xy)$ linking x and y, thereby bypassing v and modifying C_i to an eulerian tour C_{i+1} of $H_{i+1}=(H_i\setminus\{e_1,e_2\})\cup\{e\}$. By the triangle inequality, we have $w(H_{i+1})=w(H_i)-w(e_1)-w(e_2)+w(e) \leq w(H_i)$. The final graph H_{n-2}, being a connected eulerian graph on n vertices and n edges, is a hamiltonian cycle of G. Furthermore, $w(H_{n-2}) \leq w(H_0)=2w(T) \leq 2w(C)$. □

A $\frac{3}{2}$-approximation algorithm for the Metric Travelling Salesman Problem was found by Christofides [2].

4.3 Algorithms in edge-weighted digraphs

Computing shortest paths in directed graphs can be done in much the same way as in undirected graphs by growing arborescences rather than trees. Dijkstra's Algorithm and Bellman-Ford Algorithm translates naturally.

The Minimum-Weight Spanning Tree Problem is equivalent to finding the minimum-weight spanning connected subgraph. The corresponding problem in digraph, namely, finding a connected subdigraph with minimum weight in a connected digraph is much more complex. Actually, this problem is $\mathcal{NP}$-hard even when all edges have same weight because it contains the Directed Hamiltonian Cycle Problem as special case. One can easily describe a polynomial-time 2-approximation algorithm. (See Exercise 4.12). Vetta [6] found a polynomial-time $\frac{3}{2}$-approximation algorithm.

4.4 Exercices

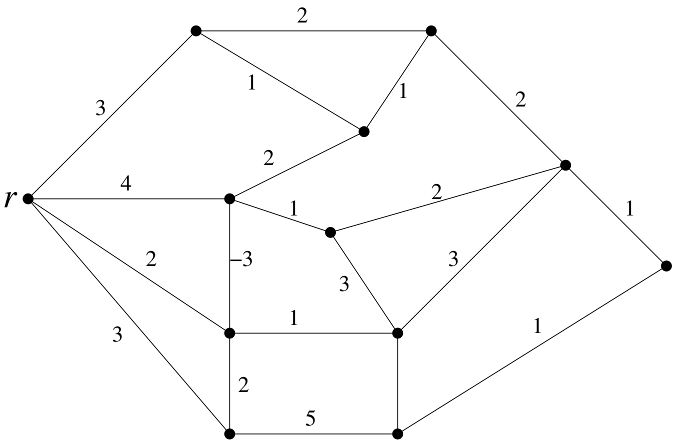
Exercise 4.1. Show a edge-weighted graph G having a vertex u such that no breadth first seach tree from u is a shortest-paths tree.

Exercise 4.2.

Consider the graph depicted in Figure 4.2.

1) Apply Dijkstra's and Bellmann-Ford algorithms for finding a shortest-paths tree from r.

2) Apply the algorithms for finding a minimum-weight spanning tree.



Exercise 4.3. Let (G,w) be a connected edge-weighted graph.
1) Prove that if w is a constant function then every shortest-paths tree is a minimum-weight

spanning tree.
2) Exhibit a connected edge-weighted graph in which there is a shortest-paths tree which is not a minimum-weight spanning tree.

Exercise 4.4. Four imprudent walkers are caught in the storm and nights. To reach the hut, they have to cross a canyon over a fragile rope bridge which can resist the weight of at most two persons. In addition, crossing the bridge requires to carry a torch to avoid to step into a hole. Unfortunately, the walkers have a unique torch and the canyon is too large to throw the torch across it. Due to dizziness and tiredness, the four walkers can cross the bridge in 1, 2, 5 and 10 minutes. When two walkers cross the bridge, they both need the torch and thus cross the bridge at the slowest of the two speeds.

With the help of a graph, find the minimum time for the walkers to cross the bridge.

Exercise 4.5. Let T be a minimum-weight spanning tree of an edge-weighted graph (G,w) and T' another spanning tree of G (not necessarily of minimum weight). Show that T' can be transformed into T by successively exchanging an edge of T' by an edge of T so that at each step the obtained graph is a tree and so that the weight of the tree never increases.

Exercise 4.6. Little Attila proposed the following algorithm to solve the Minimum-Weight Spanning Tree Problem: he considers the edges successively in decreasing order with respect to their weight and suppress the ones that are in a cycle of the remaining graph. Does this algorithm give an optimal solution to the problem? Justify your answer.

Exercise 4.7. Let (G,w) be an edge-weighted graph. For all $t \geq 1$, a *t-spanner* of (G,w) is a spanning edge-weighted graph (H,w) of (G,w) such that, for any two vertices u,v, $dist_{H,w}(u,v) \leq t \times dist_{G,w}(u,v)$.

1) Show that (G,w) is the unique 1-spanner of (G,w).

2) Let $k \geq 1$. Prove that the following algorithm returns a $(2k-1)$-spanner of (G,w).

1. Initalise H : $V(H) := V(G)$, $E(H) := \emptyset$. Place the edges in a stack in increasing order with respect to their weight. The minimum weight edge will be on top of the stack.
2. If L is empty then return H. Else remove the edge uv from the top of the stack;
3. If in H there is no (u,v)-path with at most $2k-1$ edges, add e to H.
4. Go to 2.

3) Show that the spanner returned by the above algorithm contains a minimum-weight spanning tree. (*One could show that at each step the connected components of H and the forest computed by Boruvka-Kruskal Algorithm are the same.*)

Exercise 4.8.
We would like to determine a spanning tree with weight close to the minimum. Therefore we study the following question: What is the complexity of the Minimum-Weight Spanning Tree Problem when all the edge-weights belong to a fixed set of size s. (One could consider first the case when the edges have the same weight or weight in $\{1,2\}$.

We assume that the edges have integral weights in $[1,M]$. We replace an edge with weight in $[2^i, 2^{i+1}-1]$ by an edge of weight 2^i. (We *sample* the weight.) Prove that if we compute a minimum-weight spanning tree with the simplified weight then we obtain a tree with weight at most twice the minimum for the original weight.

What happens if we increase the number of sample weights?

Exercise 4.9. 1) Let G be 2-connected edge-weighted graph. (See Chapter 5 for the definition of 2-connectivity.) Show that all the spanning trees have minimum weight if and only if all the edges have the same weight.

2) Give an example of a connected edge-weighted graph for which all the spanning tree have the same weight but whose edges do not all have the same weight.

Exercise 4.10. The *diameter* of an edge-weighted graph (G,w) is the maximum distance between two vertices: $diam(G) := \max\{dist_{G,w}(u,v) \mid u \in V(G), v \in V(G)\}$.
Show that the following algorithm computes the diameter of an edge-weighted tree T.

1. Pick a vertex x of T.
2. Find a vertex y whose distance to x is maximum (*using Dijkstra's Algorithm for example*).
3. Find a vertex z whose distance to y is maximum.
4. Return $dist_{T,w}(y,z)$.

Exercise 4.11. Show that the Metric Travelling Salesman Problem is $\mathcal{NP}$-hard.

Exercise 4.12.

1) Let D be a strongly connected digraph on n vertices. A spanning subdigraph of D is *strong-minimal* if it is strongly connected and every spanning proper subdigraph is not strongly connected.

a) Show that in the handle decomposition of a strong-minimal spanning subdigraph all the handles have length at least 2.

b) Deduce that a strong-minimal spanning subdigraph of D has at most $2n-2$ arcs.

2) Describe a polynomial-time 2-approximation for the following problem:
Instance: a strongly connected digraph D.
Find: a strongly connected spanning subdigraph with minimum number of arcs.

Bibliography

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley, Reading, MA, 1983.

[2] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Management Sciences Research Report* 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms.* Second Edition. MIT Press, Cambridge, MA, 2001.

[4] D. E. Knuth. *The Art of Computer Programming. Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1969. Second printing.

[5] R. E. Tarjan. *Data Structures and Networks Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 44, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1983.

[6] A. Vetta. Approximating the minimum strongly connected subgraph via a matching lower bound. *Proceedings of the twelfth annual ACM-SIAM symposium on discrete algorithms (SODA 2001)*, pages 417–426, 2001.