Algorithmic Discrete Mathematics
3. Exercise Sheet

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Mathematics
Andreas Paffenholz
Silke Horn

SS 2013
15/16 May 2013

Groupwork

Exercise G1

A *spanning forest* for a graph $G = (V, E)$ with $c(G)$ connected components is a forest $T = (V, F)$ with $F \subseteq E$ and $|F| = |V| - c(G)$. In particular, a spanning forest of a connected graph is a spanning tree.

Generalize the breadth-first-search algorithm so that it computes a spanning forest of a not necessarily connected graph. Also determine the running time.

Solution: We can adapt the algorithm as follows:

Algorithm 1: Breadth-First-Search (BFS) for not necessarily connected graphs

```
Input: graph G = (V, E), V = {1, ..., n} given as adjacency list
Output: predecessor function pred : V → V ∪ {0}
foreach v ∈ V do
    pred(v) ← 0
    seen(v) ← 0
Q ← ∅
foreach r ∈ V do
    if seen(r) = 0 then
        push_back(Q, r)
        seen(r) ← 1
        while Q ≠ ∅ do
            u ← pop_front(Q)
            foreach v ∈ Adj(u) do
                if seen(v) = 0 then
                    seen(v) ← 1
                    pred(v) ← u
                    push_back(Q, v)
```

The predecessor function defines the spanning forest $T = (V, \{\{v, \text{pred}(v)\} \mid v \in V, \text{pred}(v) \neq 0\})$.

The algorithm performs BFS as presented in the lecture on each connected component of the graph. Hence the running time remains $\mathcal{O}(m + n)$.

Exercise G2

Reconsider the depth-first-search algorithm presented in the lecture:

Algorithm 2: Depth-First-Search (DFS)

```
Input: graph G = (V, E), V = {1, ..., n} given as adjacency list
Output: predecessor function pred : V → V ∪ {0}
foreach v ∈ V do
    pred(v) ← 0
    seen(v) ← 0
foreach v ∈ V do
    if seen(v) = 0 then
        DFSvisit(G, v)
```

Function DFSvisit(G,r)

Input: graph $G=(V,E)$ given as adjacency list, root node $r \in V$

$\mathsf{seen}(r) \leftarrow 1$
foreach $v \in \mathrm{Adj}(r)$ **do**
  if $\mathsf{seen}(v)=0$ **then**
    $\mathsf{pred}(v)=r$
    DFSvisit(G,v)

Show that the algorithm correctly computes a spanning forest and determine its running time.

Solution: Let $T=(W,F)$ for $W=\{v \in V \mid \mathsf{seen}(v)=1\}$ and $F=\{\{v,\mathsf{pred}(v)\} \mid \mathsf{pred}(v) \neq 0\}$. We want to show that T is a spanning forest of G. It suffices to consider the case where G is connected. In this case the outer loop in line 4 of DFS calls DFSvisit only once. If G is disconnected it does so once for every connected component of G.

- T is acyclic: This is true since the function DFSvisit is called only once on each node.
- T is connected: Let $r \in V$ be the root with which DFSvisit is called the first time in line 6 of DFS and let $v \in V$ be any vertex. Suppose that v is not connected to r in T. Let $(r=v_0,v_1,\dots,v_k=v)$ be a path in G and let v_j be the first node not in T. Then $\mathsf{seen}(v_j)=0$. But $\mathsf{seen}(v_{j-1})=1$ and at some point the function DFSvisit has been called for all unseen nodes adjacent to v_{j-1}. So $W=V$ and T is connected.

Lastly, we show that the running time of the algorithm is $\mathcal{O}(m+n)$. This is true since in the worst case we have to consider every node and every edge once.

Exercise G3
Recall Kruskal's algorithm:

Algorithm 3: Kruskal's Algorithm

Input: graph $G=(V,E)$, weight function $w: E \to \mathbb{R}$
Output: Minimal spanning tree $T=(V,F)$ of G

1 $F \leftarrow \emptyset$
2 $L \leftarrow E$
3 Sort the edges in L increasingly by weight
4 **while** $L \neq \emptyset$ **do**
5   $e \leftarrow \mathsf{pop\_front}(L)$
6   **if** $(V, F \cup \{e\})$ *is acyclic* **then**
7     $F \leftarrow F \cup \{e\}$

The goal of this exercise is to show that the loop in lines 4–7 can be implemented so that it runs in time $\mathcal{O}(m \log n)$.

To this end, we have to verify whether inserting the edge e in step 6 encloses a cycle. We will at each step keep track of the connected components of the forest. We define a function $\mathsf{find}: V \to V$ that maps a vertex to some unique representative of its connected component. It then suffices to check whether the two endpoints of $e=\{u,v\}$ are in the same component, *i.e.*, e encloses a cycle if and only if $\mathsf{find}(u)=\mathsf{find}(v)$.

If we add e to the forest, we have to form the union of the two connected components containing u and v. To this end, we need a function union that forms the union.

(a) Describe an easy $\mathcal{O}(n)$ implementation of find and union.

(b) We can do faster if we arrange the elements of each connected component in a rooted tree with the representative in the root. Describe the details of such an implementation and show that $\mathsf{find}(v)$ and $\mathsf{union}(u,v)$ run in $\mathcal{O}(\log n)$ time.

(c) Conclude that the loop in lines 4–7 runs in time $\mathcal{O}(m \log n)$.

(d) Can you think of even more improvements?

Solution:

(a) We can store the representatives for the connected components in an array A with one entry for each $v \in V$. Then $\mathsf{find}(v)$ simply returns $A(v)$ (in constant time.) In the beginning $A(v)=v$ for every $v \in V$. Then $\mathsf{union}(u,v)$ goes through A and sets $A(w)$ to $\mathsf{find}(v)$ for every w with $A(w)=\mathsf{find}(u)$. This needs time $\mathcal{O}(n)$.

(b) We start with n isolated nodes, each one being its own representative. Now $\mathsf{union}(u,v)$ attaches the smaller tree as a new child to the root of the other one. (To make this efficient we should store the size of each tree in the root.) Then the depth of each tree is at most $\mathcal{O}(\log n)$. Moreover, $\mathsf{find}(v)$ returns the root of the tree containing v. Both operations run in $\mathcal{O}(\log n)$ time.

(c) The loop goes over all edges of G. For each edge we have to check whether it encloses a cycle. This yields a running time of $\mathcal{O}(m \log n)$.

(d) In the find-function we can attach every node that we pass directly to the root, accelerating subsequent function calls.

Exercise G4
Let $G = (V, E)$ be a d-*regular* graph on n vertices, *i.e.*, each vertex $v \in V$ has degree d. Show that the total number of triangles in G and $\overline{G}$ equals $\binom{n}{3} - \frac{n}{2} d(n-d-1)$. (Recall that the complementary graph $\overline{G}$ of G is defined as $\overline{G} = (V, \binom{V}{2} \setminus E)$.)

Solution: There are $\binom{n}{3}$ triples of vertices. We now have to count those that do not form a triangle. Let $v \in V$ be an arbitrary vertex. A pair x, y of neighbours of v does not form a triangle if one of the edges vx and vy is in G and the other in $\overline{G}$. (Note that this is not an "if and only if" here since we ignored the third edge xy.) For this, there are $d(n-d-1)$ possible choices. If we count this at every vertex, we count every non-triangle twice (since every non-triangle has two vertices with one edge in G and one in $\overline{G}$).

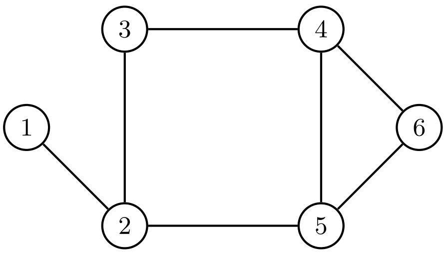
Homework

Exercise H1 (5 points)
Perform

(a) the BFS algorithm and

(b) the DFS algorithm

on the following graph with root node $s = 1$:

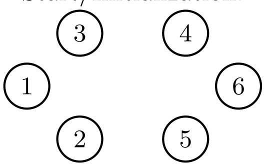


Always go through the vertices in the adjacency list in increasing order. Determine the values of pred, seen and L (only for BFS) in each step. Moreover, give the spanning tree that is constructed.

Solution:

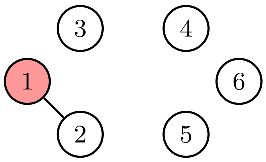
(a) BFS with root node 1 (red nodes are those in the queue, colored, *i.e.*, red or blue, nodes are those with seen$(v) = 1$):

Start/Initialization:



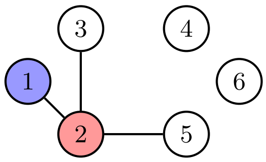
$L = [0, \infty, \infty, \infty, \infty, \infty]$
pred $= [0, 0, 0, 0, 0, 0]$

1. Iteration:



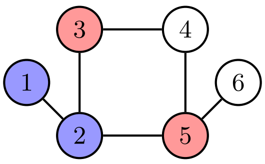
$L = [0, \infty, \infty, \infty, \infty, \infty]$
pred $= [0, 0, 0, 0, 0, 0]$

2. Iteration:



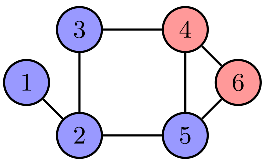
$L = [0, 1, \infty, \infty, \infty, \infty]$
pred $= [0, 1, 0, 0, 0, 0]$

3. Iteration:



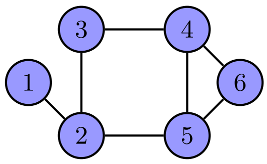
$L = [0, 1, 2, \infty, 2, \infty]$
pred $= [0, 1, 2, 0, 2, 0]$

4. Iteration:



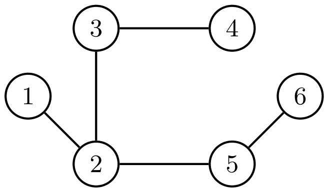
$L = [0, 1, 2, 3, 2, 3]$
pred $= [0, 1, 2, 3, 2, 5]$

5. Iteration:



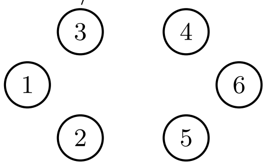
$Q = \emptyset$, STOP

The constructed spanning tree looks as follows:



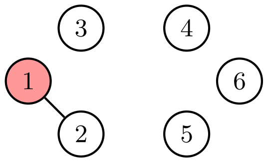
(b) DFS with root node 1 (red nodes are those with $\mathsf{seen}(v) = 1$):

Start/Initialization:



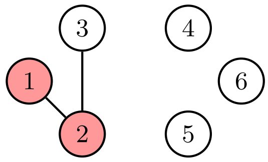
$\mathsf{pred} = (0,0,0,0,0,0)$

1. Iteration:



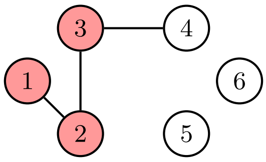
$\mathsf{pred} = (0,0,0,0,0,0)$

2. Iteration:



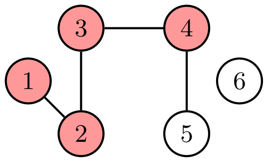
$\mathsf{pred} = (0,1,0,0,0,0)$

3. Iteration:



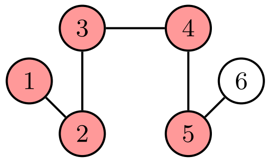
$\mathsf{pred} = (0,1,2,0,0,0)$

4. Iteration:



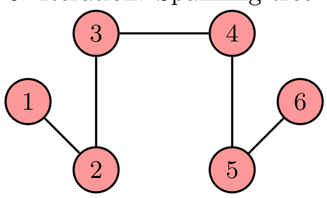
$\mathsf{pred} = (0,1,2,3,0,0)$

5. Iteration:



$\mathsf{pred} = (0,1,2,3,4,0)$

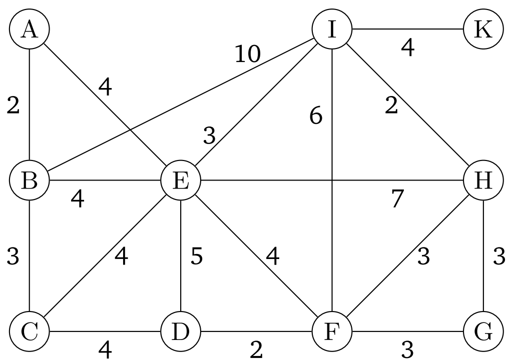
6. Iteration: Spanning tree:



$\mathsf{pred} = (0,1,2,3,4,5)$

Exercise H2 (5 points)

(a) Perform Kruskal's algorithm on the following graph:



You can use the template on the website for the drawings of the graph.

(b) Prove: If the weights of the edges are pairwise distinct then the minimal spanning tree is unique.

Solution:

(a) First we have to sort the edges:

e_k	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}
$\{i_k, j_k\}$	{AB}	{DF}	{HI}	{BC}	{FG}	{FH}	{GH}	{EI}	{AE}	{BE}
c_{e_k}	2	2	2	3	3	3	3	3	4	4

e_k	e_{11}	e_{12}	e_{13}	e_{14}	e_{15}	e_{16}	e_{17}	e_{18}
$\{i_k, j_k\}$	{CE}	{CD}	{EF}	{IK}	{DE}	{FI}	{EH}	{BI}
c_{e_k}	4	4	4	4	5	6	7	10

Now we go through the edges and insert them into the tree unless they form a cycle. See Figure 1.

(b) If the weights are pairwise distinct then the ordering of the edges is unique and hence Kruskal's algorithm always returns the same minimal spanning tree. (And we know that every minimal spanning tree can be computed by Kruskal's algorithm.)

Exercise H3 (5 points)
Let T be a minimal spanning tree in a graph $G = (V, E)$.

(a) Let $\{i, j\} \in E$ such that $G - \{i, j\}$ is still connected. Describe an algorithm that finds a minimal spanning tree in the graph $G_1 = (V, E \setminus \{\{i, j\}\})$ obtained by deleting the edge $\{i, j\}$.

(b) Let $\{k, \ell\} \notin E$. Describe an algorithm that finds a minimal spanning tree in the new graph $G_2 = (V, E \cup \{\{k, \ell\}\})$ obtained by adding the edge $\{k, \ell\}$.

In both cases show that your algorithm is correct and determine its running time.

Solution:

(a)

Algorithm 4: Modify Spanning Tree 1

Input: graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, minimal spanning tree $T = (V, F)$, deleted edge $\{i, j\} \in E$
Output: minimal spanning tree of G_1
if $\{i, j\} \notin F$ **then**
    return T
$A, B \leftarrow$ connected components of $T - \{i, j\}$`// can be obtained by BFS(`$T - \{i, j\}, i$`) and BFS(`$T - \{i, j\}, j$`)`
$e \leftarrow \operatorname{argmin}\{w(e') \mid e' \in E \setminus F, e' \cap A, e' \cap B \neq \emptyset\}$
$F' \leftarrow F \setminus \{\{i, j\}\} \cup \{e\}$
return (V, F')

If the deleted edge was not contained in the original tree T, we return T.
Otherwise, the algorithm determines the two connected components of $T - \{i, j\}$. This can be done in $\mathcal{O}(n)$ using BFS on $T - \{i, j\}$. Then it goes through all edges in E that connect these two components. This runs in $\mathcal{O}(m)$. (The check whether the edge connects the two components can be done in constant time if we construct – during BFS – an array that knows for each vertex whether it is contained in A or B.)
In total this yields a running time of $\mathcal{O}(m)$.

(b)

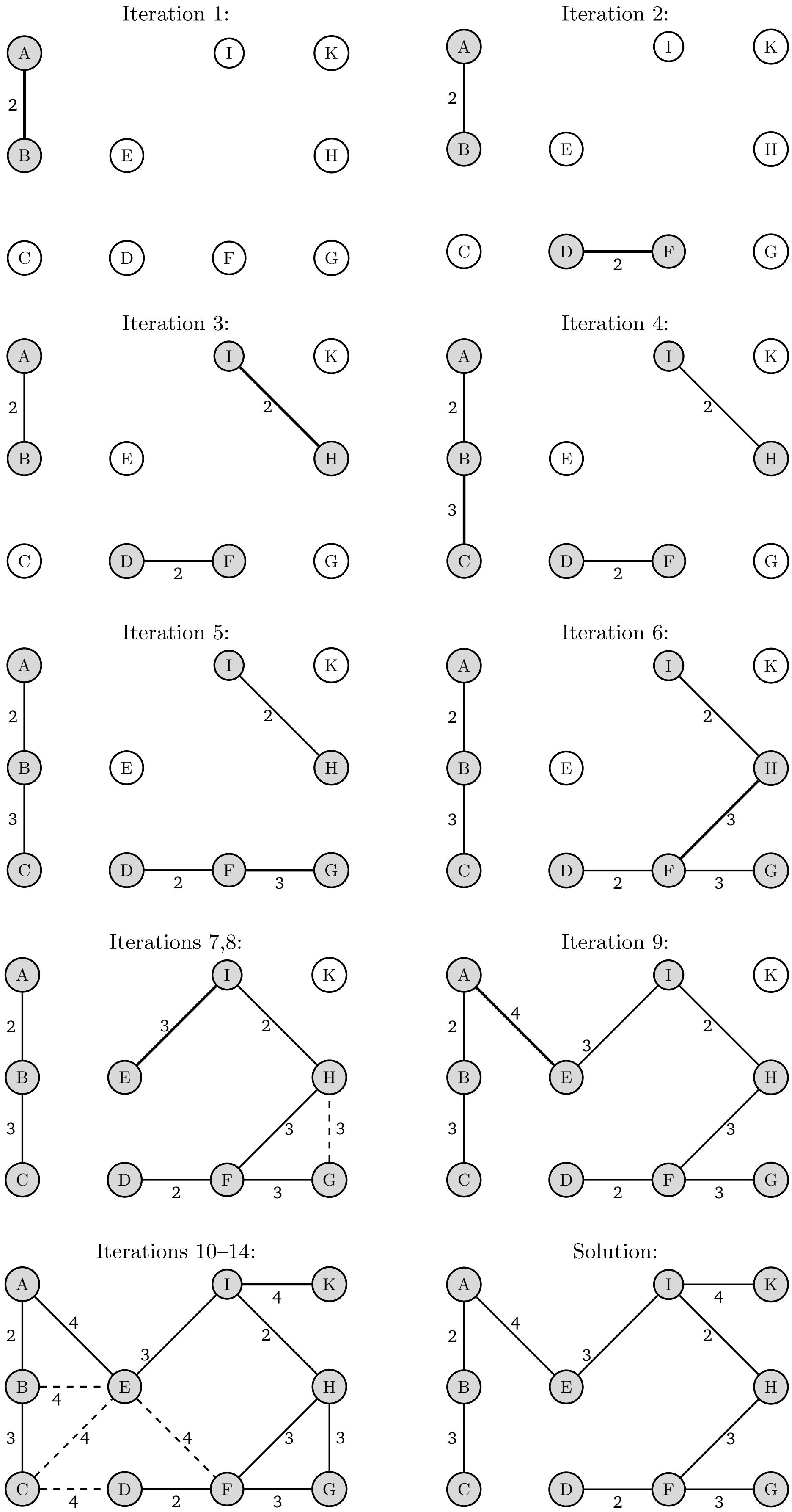
Algorithm 5: Modify Spanning Tree 2

Input: graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, (rooted) minimal spanning tree $T = (V, F)$, deleted edge $\{k, \ell\} \notin E$
Output: minimal spanning tree of G_2
$C \leftarrow$ unique cycle in $T + \{k, \ell\}$
$e \leftarrow \operatorname{argmax}\{w(e') \mid e' \in C\}$
$F' \leftarrow F \cup \{\{k, \ell\}\} - \{e\}$
return (V, F')

The algorithm first determines the unique cycle in $T + \{k, \ell\}$. This can be done by using BFS to find the unique path from k to ℓ in T. This runs in time $\mathcal{O}(n)$.
We then search for the edge with maximal weight in this cycle and delete it. Again this can be done in $\mathcal{O}(n)$.
This yields a total running time of $\mathcal{O}(n)$.

Exercise H4 (5 points)
A *tournament* $T = (V, A)$ is a directed graph in which there is exactly one edge between any two vertices. Show that in every tournament there is a vertex v such that there is a path of length ≤ 2 from v to any other vertex.

Figure 1: Kruskal's algorithm.



(No change in iterations 15–18)

Weight of the spanning tree: 26

Solution: Any vertex v with maximal outdegree does it. We will denote $N^+(v) = \{w \in V \mid (v,w) \in E\}$.

In fact, let u be one of the vertices with $(v,u) \notin E$, *i.e.,* $v \in N^+(u)$. Then there is some vertex $w \in N^+(v)$ such that $(w,u) \in E$. If this were not the case, then $N^+(u) \supseteq N^+(v) \cup \{v\}$ and hence $|N^+(u)| > |N^+(v)|$, contradicting our choice of v.