Lösungen von Übungsblatt 13

Algorithmen (WS 2018, Ulrike von Luxburg)

Lösungen zu Aufgabe 1 Berechne den minimalen Spannbaum $\widetilde{T}$ im veränderten Graphen wie folgt: *Entferne e aus E'. Dadurch zerfällt der ursprüngliche Baum T in zwei Zusammenhangskomponenten, die einen Schnitt in G bilden. Füge anstatt e die leichteste Schnittkante zu E' hinzu und erhalte $\widetilde{T}$.*

Begründung: Offensichtlich handelt es sich bei dem berechneten Graphen $\widetilde{T}$ um einen Spannbaum von G. Nun verwenden wir die Charakterisierung minimaler Spannbäume aus der Angabe.
Zur Erinnerung: Ein Spannbaum $T = (V, E')$ eines gewichteten Graphen $G = (V, E)$ ist ein minimaler Spannbaum genau dann, wenn für jede Kante $e' \in E'$ gilt: Entfernt man e' aus T, so zerfällt T in zwei Zusammenhangskomponenten Z_1 und Z_2, sodass e' eine Schnittkante bezüglich (Z_1, Z_2) in G mit minimalem Gewicht ist.
Um zu zeigen, dass $\widetilde{T}$ auch ein minimaler Spannbaum ist, müssen wir also jede Kante in $\widetilde{T}$ auf die obige Charakterisierung prüfen. Wir betrachten also eine beliebige Kante e' von $\widetilde{T}$ und prüfen, ob nach Entfernen von e', $\widetilde{T}$ in zwei Zusammenhangskomponenten Z_1 und Z_2 zerfällt, sodass e' eine Schnittkante bezüglich (Z_1, Z_2) in G mit minimalem Gewicht ist.

1. Fall: Im einfachen Fall ist e' die gegen e ausgetauschte Kante (oder e selbst). Laut unserem obigen Vorgehen ist e' die leichteste Schnittkante bezüglich des Schnitts, der entsteht, wenn man e' aus $\widetilde{T}$ entfernt.

2. Fall: Hier ist e' eine andere Kante aus $\widetilde{T}$ und damit auch im ursprünglichen Spannbaum T enthalten. Dieser Fall ist etwas schwieriger, denn wenn wir e' aus $\widetilde{T}$ entfernen, zerfällt der Baum möglicherweise in zwei andere Zusammenhangskomponenten, als wenn wir e' aus T entfernen, da die ausgetauschte Kante die Struktur des Baum geändert hat. Betrachten wir dies genauer.

Wir bezeichnen die zwei Zusammenhangskomponenten, die entstehen, wenn wir e aus T entfernen mit U und V. Klarerweise liegen in diesem 2. Fall die Endknoten von e' entweder beide in U oder beide in V. O.B.d.A. liegen beide in V. Wenn wir nun e' aus $T[V]$ entfernen so zerfällt V wiederum in zwei Zusammenhangskomponenten V_1 und V_2. Bezeichne die beiden Endknoten von e' mit $a \in V_1$ und $b \in V_2$. O.B.d.A. sei der Endknoten von e in V über einen Pfad mit a verbunden.

Dann zerfällt T, wenn wir e' entfernen, in die Zusammenhangskomponenten $U \cup V_1$ und V_2. Für alle Kanten $f \in E$ von G mit einem Endknoten in $U \cup V_1$ und dem anderen in V_2 ist das Gewicht mindestens so groß wie das von e', da T ein minimaler Spannbaum in G ist. $(\star)$

Wenn wir nun e' aus $\widetilde{T}$ entfernen, zerfällt $\widetilde{T}$ ebenfalls in zwei Komponenten. Falls der Endknoten der gegen e (eventuell) ausgetauschten Kante in V ebenfalls in V_1 liegt, zerfällt $\widetilde{T}$ ebenfalls in die Zusammenhangskomponenten $U \cup V_1$ und V_2. Doch dann wissen wir bereits, dass e' eine minimale Schnittkante ist. (Siehe $\star$.)
Falls die ausgetauschte Kante in V_2 endet, zerfällt $\widetilde{T}$ in die Zusammenhangskomponenten V_1 und $U \cup V_2$. Jetzt müssen wir noch zeigen, dass e' nicht schwerer ist als jede Schnittkante $h \in E$ mit einem Endknoten in V_1 und dem anderen in U (für V_2 ist es bereits klar). Wegen $(\star)$ ist das Gewicht der gegen e ausgetauschten Kante mindestens so groß ist wie jenes von e'. Außerdem ist die gegen e ausgetauschte Kante eine leichteste Schnittkante bezüglich U und V. Also haben alle Kanten zwischen V_1 und U ein mindestens so großes Gewicht wie e'.

Lösungen zu Aufgabe 2

(a) A maximum independent set consists of the nodes with label c, d, g, h, i.

(b) First we develop an algorithm to calculate the number of nodes in a maximum independent set. To this end we use variables $v.max$, which will contain the number of nodes of a maximum independent set of the subtree rooted at a node $v \in V$. For our algorithm we observe the following. For a node $v \in V$, there are two possibilities:

(1) Either v is in the independent set. In this case no children of v can be in the independent set, but the children of the children of v ('grandchildren') can. Thus, if the grandchildren

of v are $w_1, \ldots, w_k$, we conclude

$$v.max = 1 + \sum_{i=1}^{k} w_i.max$$

in this case (assuming that we already know the values $w_i.max$ for all $i = 1, \ldots, k$.

(2) Otherwise v is not contained in the independent set. In this case the children of v might be in the independent set. Let $c_1, \ldots, c_l$ be the children of v. Then we have

$$v.max = \sum_{i=1}^{l} c_i.max.$$

To calculate the maximum set of v, we have to use the possibility, where we can obtain the higher value for $v.max$:

$$v.max = \max \left\{ \sum_{i=1}^{l} c_i.max, 1 + \sum_{i=1}^{k} w_k.max \right\}.$$

The Algorithm 14 gives pseudocode to implement this description above.

```
1: function SizeMaxIndependentSet(T)
2:     Initialize v.max to 0 for all nodes in the tree T.
3:     while not all nodes of T are visited in a post-order walk do
4:         if v has no children then
5:             v.max ⟵ 1
6:         else
7:             if v has no grandchildren then
8:                 v.max ⟵ Σ_{c:c is child of v} c.max
9:             else
10:                v.max ⟵ max{ Σ_{c:c is child of v} c.max, 1 + Σ_{w:w is grandchild of v} w.max }
11:            end if
12:        end if
13:    end while
14: end function
```

Finally we determine the nodes contained in a maximum independent set by backtracking in Algorithm 23: Basically we perform a tree walk similar to a pre-order tree walk of T and test which of the two options for $v.max$ mentioned above was used to calculate $v.max$.

```
1: function MaxIndependentSet(T)
2:     initizalize an empty set M
3:     initialize an empty list ℓ and insert the root of T into it
4:     while ℓ is not empty do
5:         v ⟵ first element of ℓ
6:         delete the first element of ℓ
7:         if v has no children then
8:             M ⟵ M ∪ {v}
9:         else
10:            if v has no grandchildren then
11:                M ⟵ M ∪ {c : c is child of v}
12:            else
13:                if v.max = Σ_{c:c is child of v} c.max then
14:                    append all children of v to ℓ
15:                else
16:                    M ⟵ M ∪ {v}
17:                    append all grandchildren of v to ℓ
18:                end if
19:            end if
```

20: **end if**
21: **end while**
22: **return** M
23: **end function**

(c) The correctness of Algorithm 14 follows from the description in part (b) and from the observation that by using a post-order walk of T, we already determined the optimal size of the trees rooted at the children and grandchildren of a node v, when we calculate $v.max$ for the node v.

In Algorithm 23 we test which of the two options for $v.max$ described above was used for calculating a optimal value for $v.max$ (lines 10–18). Thereby we iterate over elements in a list ℓ, starting with $\ell = (r)$ (thereby r is the root of T). Note that ℓ contains the nodes of T, which we potentially want to add to the independent set M and which we can add such without creating a conflict to the definition of independent set regarding the parents of these nodes.

Suppose now we consider the first node v in ℓ. If we obtain that it is optimal to have v itself in the independent set, then we add v to the set M (line 16) and append the grandchildren of v to ℓ. Since in this case the children of v cannot be in M by definition of M, and the grandchildren might be in M and must be tested further (as we do later in the algorithm by appending them to ℓ), our algorithms is correct in this case.

If we obtain that it is optimal to not have v in the independent set, it might be possible that the children of v are in M. Therefore we append these children to ℓ for further testing in another iteration of the while-loop. Again our algorithm is correct in this case.

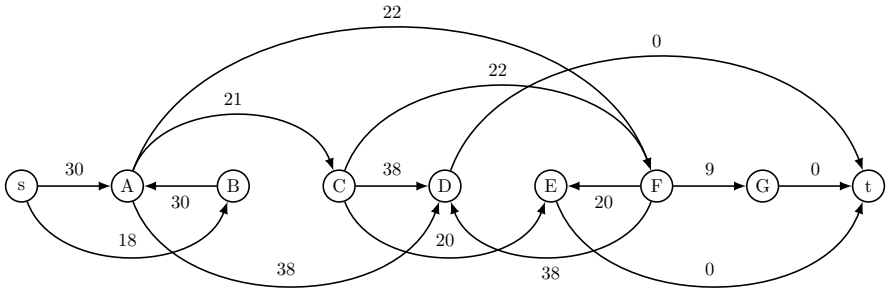
If there exist no children or grandchildren of the current vertex v, we can add v or all children of v respectively to the independent set M (recall that v is a node that we can potentially add to M).

(d) The running time of Algorithm 14 is determined by the while-loop in lines 3–13 and more specifically by the calculation of the sums in line 8 or line 10. Each vertex u of T might be part in the calculation of two different sums: the first sum is the one where the parent of u calculates the sum over all its children; the second sum is the one where the grandparent calculates the sum over all its grandchildren. This implies that the running time of Algorithm 14 is $\mathcal{O}(2n) = \mathcal{O}(n)$ in the worst-case.

The running time of Algorithm 23 mainly depends on the calculation of the sum in line 13. A similar argument as for Algorithm 14 implies that the running time here also is $\mathcal{O}(n)$. Thus, the overall running time is $\mathcal{O}(n)$.

Lösungen zu Aufgabe 3

Wir konstruieren einen Graphen $G = (V \cup \{s, t\}, E)$ mit V der Menge der Mitarbeiter, sowie den Dummy-Knoten s zum Zeitraum $s.start = s.ende = 9$ und t zum Zeitraum $t.start = t.ende = 17$. Eine gerichtete Kante (i, j) liegt vor, wenn $i.ende \in [j.start, j.ende]$, d.h., wenn die Mitarbeiter i und j den Zeitraum von $i.start$ bis $j.ende$ lückenlos abdecken können. Zudem ist (i, j) mit den Kosten von j gewichtet (wobei die Kosten von s und t als 0 angenommen werden). In diesem Graphen entspricht somit jeder Pfad von s nach t einem Schedule von Mitarbeitern, der den Zeitraum 9 Uhr bis 17 Uhr lückenlos abdeckt, wobei die Kosten des Pfades die Gesamtkosten dieses Schedules wiedergeben. Damit ist die gesuchte Lösung ein kürzester Pfad von s nach t in folgendem Graphen G:



Da (s, A, F, G, t) kürzester Pfad von s nach t, ist die Wahl $\{A, F, G\}$ kostenoptimal.