Lösungsstrategien für NP-schwere Probleme der Kombinatorischen Optimierung

— Übungsblatt 5 —

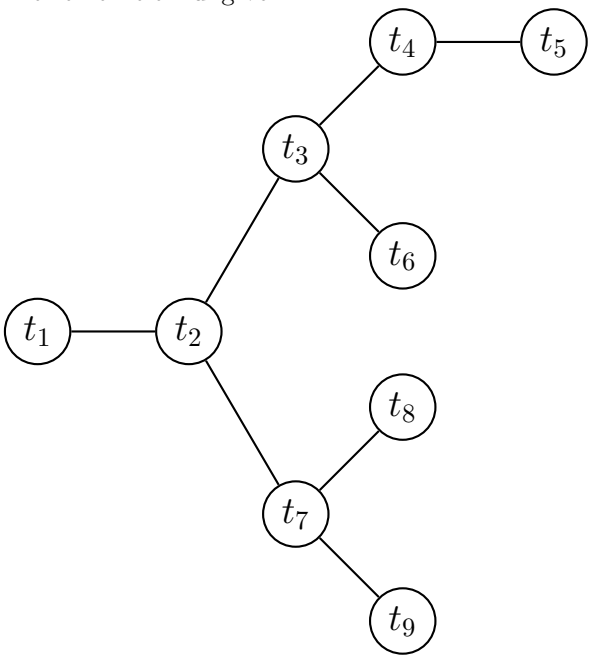
Walter Stieben (4stieben@inf) Tim Reipschläger (4reipsch@inf) Louis Kobras (4kobras@inf) Hauke Stieler (4stieler@inf)

Abgabe am: 23. Mai 2016

Aufgabe 5.1

a)

Hier eine Zeichnung von T:



Hier die zugehörigen $V_{t_1} \dots V_{t_k}$:

i	V_{t_i}
1	$\{v_1, v_2, v_{11}\}$
2	$\{v_2, v_7, v_{11}\}$
3	$\{v_2, v_5, v_7\}$
4	$\{v_2, v_3, v_5\}$
5	$\{v_3, v_4, v_5\}$
6	$\{v_5, v_6, v_7\}$
7	$\{v_7, v_9, v_{11}\}$
8	$\{v_7, v_8, v_9\}$
9	$\{v_9, v_{10}, v_{11}\}$

b)

Wir wollen zu Beginn die folgende Aussage treffen und beweisen:
In jedem triangulierten Kreisgraphen gibt es mindestens zwei Knoten vom Grad 2.

Beweis: Der kleinste triangulierte Kreisgraph ist ein Dreieck und in diesem sind alle drei Knoten vom Grad 2.
Angenommen wir haben einen Kreisgraphen mit n Knoten, wobei $n \geq 4$ ist. Dann kann man diesen Kreisgraphen teilen indem man eine Kante von einem beliebigen Knoten des Kreisgraphen zu einem beliebigen anderen Knoten des Kreisgraphen hinzufügt, wobei der zweite Knoten kein direkter Nachbar des ersten Knotens sein darf. Man erhält auf diese Weise immer 2 Kreisgraphen. Führt man das Prozedere rekursiv mit den beiden gebildeten Kreisgraphen durch, dann wird man schlussendlich nur noch Kreise mit 3 Knoten haben, also Dreiecke. Da ursprünglich ein Kreisgraph vorhanden war und dieser am Ende mit Dreiecken gefüllt ist, bekommen wir auf diese Art einen triangulierten Kreisgraphen. Da wir den ursprünglichen Kreisgraphen mit der ersten Kante in 2 Kreisgraphen geteilt haben und der kleinste Kreis ein Kreis mit 3 Knoten ist (Dreieck) erhalten wir beim Terminieren mindestens 2 Dreiecke, die außen liegen und da sie außen liegen haben sie einen Knoten, der den Grad 2 haben muss.

Algorithm 1 Baumzerlegung für triangulierte Kreisgraphen

```
1: procedure INITIALISIERUNG
2:     Suche einen Knoten u mit Grad 2
3:     Suche die beiden Nachbarn v, w dieses Knotens
4:     Füge {u, v, w} als V_{t_i} zu V_t und einen neuen Knoten t_i zu T hinzu.
5:     REKURSION(u, v, w)
6: end procedure
7: procedure REKURSION(u,v,w)
8:     Suche einen Knoten x, der zu v und w adjazent ist und der nicht u ist.
9:     if Es exisiert ein x then
10:        Füge {v, w, x} als V_{t_i} zu V_t und einen neuen Knoten t_i zu T hinzu.
11:        Füge eine Kante {t_i, t_j} zu T hinzu, wobei t_i der Knoten in T ist, der zum Dreieck zwischen
       u,v,w gehört und t_j der Knoten in T ist, der zum Dreieck zwischen v,w,x gehört.
12:        REKURSION(v,w,x)
13:        REKURSION(w,v,x)
14:    end if
15: end procedure
```

Dieser Algorithmus liefert das korrekte Ergebnis, weil er nach obiger Aussage auf jeden Fall einen Knoten zum initialisieren findet und rekursiv den triangulierten Kreisgraphen abarbeitet, sodass er letztlich eine Baumzerlegung für den gesamten triangulierten Kreisgraphen findet. Er geht dazu von einem beliebigen Startknoten aus und betrachtet deren Nachbarn, fügt das so gefundere Dreieck als ersten Knoten in T ein und die drei Knoten entsprechend zu V_T hinzu. Damit ist die Initialisierung abgeschlossen.
Jetzt folgt ein rekursiver Teil, der zu bisher betrachteten Kanten jeweils guckt, ob diese Teil eines noch nicht betrachteten Dreiecks sind. Findet man so ein Dreieck hat dieses dann ebenfalls zwei neue noch nicht betrachtete Kanten für die man jeweils rekursiv weiterarbeitet. Damit man keine Rückschritte macht, wird immer ein Knoten u übergeben, der derjenige Knoten ist, zu dem wir das Dreieck mit der aktuell betrachteten Kante schon kennen. Immer wenn wir ein neues Dreieck finden, fügen wir einen Knoten zu T hinzu und die drei Knoten entsprechend zu V_T hinzu. Außerem nutzen wir das u, um darüber auf das Dreieck zu schließen, von dem wir in den aktuellen rekursiven Aufruf gekommen sind und ziehen in T eine Kante zwischen dem entsprechenden Knoten zu diesem Dreieck und zwischen dem entsprechenden Knoten zum neu gefundenen Dreieck. Da der Eingabegraph endlich ist und wir keine Rückschritte machen terminiert der Algorithmus auch.

Aufgabe 5.2

c)

Laufzeitanalyse

Die Laufzeit des von `EXPLORE`(Φ, d) lässt sich mittels Rekurrenzgleichung und Substitutionsmethode bestimmen.

Zunächst die Rekurrenzgleichung in Abhängigkeit von d und n. Die Menge C taucht nicht auf, da sie nicht Teil der Eingabe von `EXPLORE`(Φ, d) ist. Dem n wäre also noch ein linearer Faktor hinzuzuzählen, was jedoch für die Laufzeit nicht weiter von Bedeutung ist.

$$T(d,n) = \begin{cases} n & \text{, für } d = 0 \\ n \cdot 3 \cdot T(d-1) & \text{, sonst} \end{cases}$$

Mittels der Sibstitutionsmethode ergibt sich folgende Kette:

$$\begin{aligned} T(d,n) &= n + 3 \cdot T(d-1) \\ &= n + 3 \cdot (n + 3 \cdot T(d-2)) \\ &= n + 3n + 9 \cdot T(d-2) \\ &= 4n + 9 \cdot T(d-2) \\ &= 4n + 9 \cdot (n + 3 \cdot T(d-3)) \\ &= 4n + 9n + 27 \cdot T(d-3) \\ &= 13n + 27 \cdot T(d-3) \\ &= 13n + 27 \cdot (n + 3 \cdot T(d-4)) \\ &= 13n + 27n + 81 \cdot T(d-4) \\ &= 40n + 81 \cdot T(d-4) \qquad (1) \\ &\vdots \\ &= \frac{3^k - 1}{2} + 3^k \cdot T(d-k) \end{aligned}$$

Nach drei Substitutionen lässt sich erkennen, dass der Koeffizient von T stets 3^k (wegen der Folge $3, 9, 27, 81, \dots$) und der vom n stets $\frac{3^k-1}{2}$ sein kann (wegen der Folge $1, 4, 13, 40, \dots$).

> Beweise zu den Gleichungen ersparen wir uns hier der einfachheitshalber.

Der Abbruch des Algorithmus' geschieht bei $k = d$. Dadurch ist $T(d-k) = T(d-d) = T(0) = n$ und wir erhalten die finale Laufzeit in der Größenordnung

$$\mathcal{O}\left(\frac{3^d - 1}{2} + 3^d \cdot n\right).$$

d)

Wenn man $d = \frac{n}{2}$ wählt, so erhält und in die Laufzeit aus 2.a) einsetzt, erhält man folgende neue Laufzeit:

$$\begin{aligned}\frac{3^{n/2}-1}{2} + 3^{n/2} \cdot n &= 3^{n/2} \cdot 0,5 - \left(\frac{1}{2}\right) + 3^{n/2} \cdot n \qquad \text{(Die } \mathcal{O}\text{-Notation ignoriert Konstanten)} \\ &= 3^{n/2} + 3^{n/2} \cdot n \\ &= \left(\sqrt{3}\right)^n + \left(\sqrt{3}\right)^n \cdot n \\ &= (n+1) \cdot \left(\sqrt{3}\right)^n\end{aligned}$$

> Die Notation $n/2$ statt $\frac{n}{2}$ wird zwar von Mathematikern und Theoretikern oftmals abgelehnt, wird hier jedoch zugunsten der Lesbarkeit verwendet.

Somit ist die Laufzeit von `EXPLORE`(Φ, d) mit $d = \frac{n}{2}$ in $\mathcal{O}\left((n+1) \cdot \left(\sqrt{3}\right)^n\right)$.

Um jedoch alle möglichen Belegungen durchzugehen bedarf es $\frac{2^n}{\frac{n}{2}} = \frac{2^{n+1}}{n}$ Aufrufe. Jeder Aufruf von `EXPLORE`(Φ, d) geht $d = \frac{n}{2}$ Belegungen durch, wodurch man nicht volle 2^n mögliche Belegungen benötigt.

Die Anzahl multipliziert mit der Anzahl an Schritten aus vorheriger Rechnung ergibt die Gesamtlaufzeit von $\mathcal{O}\left(\frac{2^{n+1}}{n} \cdot (n+1) \cdot \left(\sqrt{3}\right)^n\right)$.

Somit haben wir eine Laufzeit der Form $\mathcal{O}\left(p(n) \cdot \left(\sqrt{3}\right)^n\right)$ mit $p(n) = \frac{2^{n+1}}{n} \cdot (n+1)$.