ETH Zürich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger, Dr. Johannes Lengler Len
Florian Meier

HS 2018

Algorithmen und Komplexität
Lösungsvorschlag zu Übungsblatt 2

Lösungsvorschlag zu Aufgabe 1

(a) Der ursprüngliche Graph besitzt m Kanten. Weil es $\binom{n}{2}$ mögliche Kanten gibt, enthält der Komplementgraph $\overline{G}$ genau $\binom{n}{2} - m$ Kanten. Ähnlich können wir für die Knotengrade $\deg(v)$ argumentieren: Der Knoten v ist zu jedem der anderen $n-1$ Knoten entweder in G oder in $\overline{G}$ verbunden. Es folgt $\deg_{\overline{G}}(v) = n - 1 - \deg(v)$.

(b) Für den d-dimensionalen Hyperwürfel Q_d gilt:

- Aus der Definition der Knotenmenge $V = \{0,1\}^d$ folgt sofort, dass es $|V| = 2^d$ Knoten gibt.
- Jeder Knoten in Q_d hat Grad d. Mit

$$2|E| = \sum_{v \in V} \deg(v) = |V| \cdot d$$

erhalten wir

$$|E| = |V|d/2 = d2^{d-1}.$$

(c) Es bezeichne $n = |V|$. Falls G keine isolierten Knoten hat, haben alle n Knoten Grade aus der Menge $\{1, \ldots, n-1\}$. Die Abbildung $\deg : V \to \{1, \ldots, n-1\}$ kann aber nicht injektiv sein, da $|V| = n$. Es gibt also zwei Knoten mit gleichem Grad. (Dieses Argument ist unter dem Namen *Pigeonhole-Principle* bekannt: Wenn n Tauben auf $n-1$ Löcher verteilt werden, müssen sich auf alle Fälle zwei Tauben ein Loch teilen.)

Falls G aber einen isolierten Knoten besitzt, gilt $\deg(v) \leq n-2$ für alle $v \in V$. Wieder gibt es nach dem Pigeonhole-Principle keine injektive Zuordnung, und es existieren mindestens zwei Knoten vom gleichen Grad.

(d) Wir partitionieren die Knotenmenge in $V_g = \{v \in V : \deg(v) \text{ gerade}\}$ und $V_u = \{v \in V : \deg(v) \text{ ungerade}\}$. Es gilt $2|E| = \sum_{v \in V} \deg(v) = \sum_{v \in V_g} \deg(v) + \sum_{v \in V_u} \deg(v)$ und daher

$$\sum_{v \in V_u} \deg(v) = 2|E| - \sum_{v \in V_g} \deg(v).$$

Die Summe auf der rechten Seite enthält nur gerade Summanden und ist deshalb gerade. Die Summanden auf der linken Seite sind alle ungerade, müssen aber eine gerade Zahl ergeben. Also muss ihre Anzahl gerade sein.

Lösungsvorschlag zu Aufgabe 2

(a) Wir beweisen die Aussage per Induktion über die Anzahl der Knoten $n = |V|$. Für $n = 1$ (also ein einzelner Knoten ohne Kante) gilt $|E| = |V| - 1$. Nun nehmen wir an, dass die Aussage für n gilt und $G = (V, E)$ ein kreisfreier, zusammenhängender Graph mit $n+1$ Knoten ist. Per Definition ist G ein Baum.

Nach Lemma 1.15 hat G mindestens ein Blatt u. Wegen Lemma 1.16 ist dann $G' := G[V \setminus \{u\}]$ ebenfalls ein Baum. Da G' nur n Knoten hat, können wir die Induktionsannahme anwenden und wissen damit, dass die Aussage $|E'| = |V'| - 1$ für $G' = (V', E')$ gilt. G hat aber genau eine Kante und einen Knoten mehr als G' (nämlich u), also gilt $|E| = |V| - 1$ auch für den ursprünglichen Graphen G.

(b) Sei $G = (V, E)$ ein kreisfreier Graph mit $|E| = |V| - 1$, und seien weiter die Komponenten von G die induzierten Teilgraphen $G[V_1]$, $G[V_2]$, ..., $G[V_m]$. Da G kreisfrei ist, muss auch jede Komponente kreisfrei sein. Aus Teilaufgabe (a) folgt, dass $|E_i| = |V_i| - 1$ für jede Zusammenhangs-Komponente $G[V_i]$ gilt (wobei wir mit E_i die Kantenmenge der Komponente $G[V_i]$ bezeichnen). Alle Komponenten zusammen enthalten jeden Knoten und jede Kante von G genau einmal. Summieren wir über alle m Gleichungen der Form $|E_i| = |V_i| - 1$, erhalten wir $|E| = |V| - m$. Aus der Aufgabenstellung wissen wir dass $|E| = |V| - 1$, und deshalb ist $m = 1$. Es gibt also nur eine Zusammenhangskomponente, somit ist G zusammenhängend.

Lösungsvorschlag zu Aufgabe 3

Für einen Graphen $G = (V, E)$ sei $n = |V|$, $m = |E|$ und die Knoten seine mit $1, \ldots, n$ nummeriert.

1) **Einfache Adjazenzmatrix:**
Der Graph wird in einer $n \times n$-Matrix abgespeichert. An jeder Stelle der Matrix steht eine Eins, falls im Graph eine Kante zwischen den entsprechenden Knoten ist, und eine Null sonst.

$$\text{Adjazenz Matrix } A_G\text{: } (A_G)_{i,j} = \begin{cases} 1, & if\ \{u,v\} \in E \\ 0, & if\ \{u,v\} \notin E \end{cases}$$

Es wird insgesamt $O(n^2)$ Speicherplatz benötigt.

Zu den Operationen:
Um den Grad des Knotens i herauszufinden geht man die i-te Zeile der Matrix durch und zählt die Anzahl Einer in der Zeile, respektive man berechnet einfach die Summe $\sum_{j=1}^{n} (A_G)_{i,j}$ (Laufzeit $\mathcal{O}(n)$). Die Nachbarn des Knotens i sind auch in der i-ten Zeile der Matrix zu finden. Im schlimmstenfall muss auch hier die ganze Zeile durchgegangen werden bis ein Eintrag mit einer Eins gefunden wird (Laufzeit $\mathcal{O}(n)$). Um zu sehen, ob zwei Knoten i und v benachbart sind, wird der Eintrag $(A_G)_{i,j}$ abgefragt, dies ist in konstanter Zeit möglich (Laufzeit $\mathcal{O}(1)$). Möchte man eine Kante $\{u, v\}$ löschen oder hinzuzufügen, so muss man lediglich die Einträge $(A_G)_{i,j}$ und $(A_G)_{j,i}$ der Matrix auf Null beziehungsweise Eins setzen (Laufzeit $\mathcal{O}(1)$). Einen Knoten kann man löschen, indem man die entsprechende Zeile und Spalte der Matrix komplett löscht. Leider bedeutet dies meist dass man die gesammte Matrix kopieren muss (Laufzeit $\mathcal{O}(n^2)$). Wir möchten bemerken: wenn man sich etwas schlau anstellt und dynamische Arrays verwendet, kann man dies in allen praktischen Fällen auf lineare Zeit reduzieren.

2) **Einfache Adjazenzliste**
Der Graph wird durch n Listen gespeichert. Jede Liste beschreibt die Nachbarn eines Knotens. Die Listen sind nur "vorwärts verlinkt", das heisst, um auf das k-te Element der Liste zuzugreifen, muss man die ersten $k - 1$ Elemente durchlaufen.

Um für jeden Knoten eine leere Liste abzuspeichern benötigen wir $\mathcal{O}(n)$ Speicherplatz. Jede weitere Kante benötigt dann konstant viel zusätzlichen Speicherplatz, was insgesamt in $\mathcal{O}(m + n)$ Speicher resultiert.

Zu den Operationen:
Um den Grad eines Knotens u herauszufinden geht man die gesamte Liste des Knoten durch,

da die Länge der Liste genau dem Grad von u entspricht (Laufzeit $\mathcal{O}(\deg(u)) \leq \mathcal{O}(n)$). Einen beliebigen Nachbarn eines Knoten erhält man in konstanter Zeit (Laufzeit $\mathcal{O}(1)$), indem man den ersten Eintrag in der Liste des Knotens ausgibt. Um zu testen ob zwei Knoten u und v benachtbart sind, muss man die Liste von u, bzw. die von v, nach dem Knoten v, bzw. u, absuchen (Laufzeit $\mathcal{O}(\deg(u))$ bzw. $\mathcal{O}(\deg(v))$). Wenn man beide Listen parallel durchgeht und abbricht sobald man das Ende einer der Listen erreicht, kann man dies auf $\mathcal{O}(\min\{\deg(u), \deg(v)\})$ reduzieren. Eine Kante $\{u, v\}$ zu löschen ist etwas schwieriger, weil sowohl die Kante $\{u, v\}$ in der Liste des Knotens u als auch die Gegenkante $\{v, u\}$ in der Liste des des Knotens v gelöscht werden müssen. Dafür muss man schlimmstenfalls die gesamten Listen von u und von v durchgehen (Laufzeit $\mathcal{O}(\deg(u) + \deg(v))$). Eine Kante $\{u, v\}$ kann man hinzufügen, indem man sie am Anfang der Listen von u und v einfügt (Laufzeit $\mathcal{O}(1)$), sofern man weiss, dass die entsprechende Kante noch nicht im Graphen enthalten ist. Falls die hinzugefügte Kante bereits im Graphen enthalten sein könnte, muss man zuerst testen ob die Kante bereits vorhanden ist, was zur gleichen Laufzeit wie in (iii) führt. Um einen Knotens u zu löschen , löscht man die Liste von u und und für jeden Nachbarn v von u den Eintrag der Kante $\{u, v\}$ in der Liste von v. Dies benötigt Laufzeit $\mathcal{O}(\sum_{v \in \Gamma(u)} \deg(v))$. Im schlimmsten Fall ist u mit allen anderen Knoten verbunden, so braucht das Löschen dieser $n - 1$ Kanten die Zeit im schlimmsten Fall $\mathcal{O}(\sum_{v \in V} \deg(v)) = \mathcal{O}(m)$.

3) **Erweiterte Adjazenzliste:**
Der Graph wird in n Listen gespeichert. Jede Liste beschreibt wiederum die Nachbarn eines Knotens. Am Anfang jeder Liste wird zusätzlich der Grad des Knotens gespeichert. Desweitern sind die Listen vorwärts und rückwärts verlinkt und jeder Eintrag besitzt einen Pointer zum korrespondierenden Eintrag in der Liste des benachbarten Knoten.

Um für jeden Knoten eine leere Liste abzuspeichern benötigen wir $\mathcal{O}(n)$ Speicherplatz. Jede weitere Kante benötigt dann konstant viel zusätzlichen Speicherplatz, was insgesamt in $\mathcal{O}(m + n)$ Speicher resultiert.

Zu den Operationen:
Den Grad eines Knotens kann man direkt am Anfang der Liste ablesen (Laufzeit $\mathcal{O}(1)$). Einen Nachbarn ausgeben bzw. überprüfen, ob zwei Knoten benachbart sind, funktioniert wie bei der einfachen Adjazenzliste (Laufzeit $\mathcal{O}(1)$ bzw. $\mathcal{O}(\min\{\deg(u), \deg(v)\})$). Beim Löschen einer Kante kommt es darauf an, in welcher Form man die zu löschende Kante als Input kriegt. Oft kriegt man die zu löschende Kante durch den Algorithmus "direkt" in der Datenstruktur (z.B. finde eine Brücke und lösche diese). In diesem Fall geht das Löschen der Kante in konstanter Zeit (Laufzeit $\mathcal{O}(1)$) folgendermassen. Wir wollen die Kante $\{u, v\}$ löschen und haben Zugriff auf v in der Liste von u. Durch den Pointer zur Gegenkante $\{v, u\}$ finden wir u in der Liste von v in $\mathcal{O}(1)$. Anschliessend löschen wir den Eintrag in beiden Listen und verknüpfen die vorhergehende Elemente der Listen mit den nachfolgenden Elementen in $\mathcal{O}(1)$ (wegen den Doppelpointern können wir auch auf das vorhergehende Element in $\mathcal{O}(1)$ zugreifen). Kriegt man nur die zwei Endkonten der zu löschenden Kante als Input, so muss man die Kante zuerst noch finden, was wie oben erwähnt $\mathcal{O}(\min\{\deg(u), \deg(v)\})$ Zeit benötigt. Eine Kante einzufügen funktioniert gleich wie bei der einfachen Adjazenzliste, ausser dass man wiederum noch den Grad am Anfang der Liste anpassen muss (Laufzeit $\mathcal{O}(1)$), ausser wenn man zuerst testen muss, ob die Kante bereits im Graphen enthalten ist, dann ist die Laufzeit $\min\{\deg(u), \deg(v)\}$. Das Löschen eines Knotens u braucht im schlimmsten Fall nur noch Zeit $\mathcal{O}(\deg(u))$, da die zusätzlichen Pointer zwischen den Listen es uns erlauben, die Kanten $\{u, v\}$ in den Listen der Knoten v direkt zu löschen, ohne jeweils noch die ganze Liste des Knotens v zu durchlaufen.