technische universität dortmund

SoSe 2018

M. Buchin
A. Krivošija, D. Mezlaf
L. Pfahler, A. Rey

DAP2 – Heimübung 11

Ausgabedatum: 15.06.2018 — Abgabedatum: Mo. 25.06.2018 bis 12 Uhr

Abgabe:

Schreiben Sie unbedingt immer Ihren vollständigen Namen, Ihre Matrikelnummer und Ihre Gruppennummer auf Ihre Abgaben! Beweise sind nur dort notwendig, wo explizit danach gefragt wird. Eine Begründung der Antwort wird allerdings *immer* verlangt.

Aufgabe 11.1 (4 Punkte): (Breitensuche: Durchmesser eines Graphen)

Sei $G = (V, E)$ ein ungerichteter Graph. Für $s, t \in V$ bezeichnet $\delta(s, t)$ die Länge des kürzesten Weges in G von s nach t. Der *Durchmesser* d des Graphen G ist nun definiert als das Maximum der Längen kürzester Wege über alle Knotenpaare $s, t \in V$:

$$d := \max\{\delta(s, t) \mid s, t \in V\}$$

Bob erinnert sich aus seiner Zeit als Studierender an die Breitensuche und formuliert folgenden Algorithmus, um den Durchmesser eines zusammenhängenden Graphen G in Laufzeit $\mathcal{O}(|V| + |E|)$ zu ermitteln:

`BerechneDurchmesser`**(Graph** $G = (V, E)$**):**

```
1  s ← beliebiger Knoten aus V
2  "Initialisiere BFS mit Startknoten s"
3  while Q ≠ ∅ do
4      u ← head[Q]
5      foreach v ∈ Adj[u] do
6          if color[v] = weiß then
7              color[v] ← grau
8              d[v] ← d[u] + 1
9              enqueue(Q, v)
10     dequeue(Q)
11     color[u] ← schwarz
12 return max{d[v] | v ∈ V}
```

a) Liefert dieser Algorithmus tatsächlich für jeden Graphen G den korrekten Durchmesser? Falls ja, begründen Sie Ihre Antwort. Falls Nein, konstruieren Sie ein Gegenbeispiel.

b) Wie kann Bobs Algorithmus angepasst werden, um in Laufzeit $\mathcal{O}(|V| \cdot (|V| + |E|))$ den Durchmesser eines Graphen $G = (V, E)$ zu ermitteln?

Lösung:

a) Ein Gegenbeispiel liefert ein einfacher Pfad aus drei Knoten. Wählt der Algorithmus den mittleren der drei Knoten als Startknoten s, liefert er als Ergebnis einen Durchmesser von 1, der korrekte Durchmesser beträge jedoch 2.

b) Anstatt die Breitensuche in einem beliebigen Knoten zu beginnen, starten wir jeweils eine Breitensuche in jedem Knoten und umgeben dieses Vorgehen mit einer Maximumssuche:

```
BerechneDurchmesser(Graph G = (V, E)):
1  max ← −∞
2  foreach s ∈ V do
3      "Initialisiere BFS mit Startknoten s"
4      while Q ≠ ∅ do
5          u ← head[Q]
6          foreach v ∈ Adj[u] do
7              if color[v] = weiß then
8                  color[v] ← grau
9                  d[v] ← d[u] + 1
10                 enqueue(Q, v)
11         dequeue(Q)
12         color[u] ← schwarz
13     max ← max{max, max{d[v] | v ∈ V}}
14 return max
```

Der Algorithmus sucht nun denjenigen Knoten, der einen größten Abstand zu einem anderen Knoten im Graphen hat. Die Variable max, zusammen mit der **foreach**-Schleife realisiert eine normale Maximumssuche über die Maxima der von den Breitensuchen gelieferten größten Distanzen.

Die Laufzeit $\mathcal{O}(|V| \cdot (|V| + |E|))$ ergibt sich aus den $|V|$ Aufrufen der Breitensuche.

Bobs Algorithmus berechnet korrekt das Maximum $d(s) := \{\delta(s,t) \mid t \in V\}$. Wir erweitern den Algorithmus um eine Maximumssuche über alle $d(s)$, unser Algorithmus ermittelt also den Wert

$$\begin{aligned}\max_{s\in V}\{d(s)\} &= \max_{s\in V}\{\max\{\delta(s,t) \mid t \in V\}\} \\ &= \max_{s\in V}\left\{\max_{t\in V}\{\delta(s,t)\}\right\} \\ &= \max_{s,t\in V}\{\delta(s,t)\} = d \ .\end{aligned}$$

Der durch unseren Algorithmus berechnete Wert entspricht also genau dem gesuchten Durchmesser.

Aufgabe 11.2 (6 Punkte + 4 Bonuspunkte): (Breitensuche: Mehr Breitensuche)

In dieser Aufgabe sollen Algorithmen für ungerichtete Graphen konstruiert werden, die sich die Vorgehensweise der Breitensuche zunutze machen. Beide Algorithmen sollen beschrieben und in Pseudocode formuliert werden, und beide Algorithmen sollen bei Eingabe eines Graphen $G = (V, E)$ eine worst-case-Laufzeit von $\mathcal{O}(|V| + |E|)$ haben. Um zu begründen, dass diese Laufzeitschranke eingehalten wird, ist keine vollständige Laufzeitanalyse nötig, sondern lediglich eine Betrachtung der jeweiligen Anpassungen der Breitensuche. Sie dürfen für diese Aufgabe annehmen, dass der gegebene Graph G zusammenhängend ist.

a) Geben Sie einen Algorithmus `istBaum(`G`)` an, der genau dann `TRUE` ausgibt, wenn der eingegebene Graph G ein Baum, also kreisfrei ist. Sonst soll der Algorithmus `FALSE` ausgeben.

Lösung: Der Breitensuche-Algorithmus färbt einen Knoten $v \in V$ *grau* (resp. *schwarz*), sobald er diesen zum ersten Mal "entdeckt" (resp. "abgearbeitet hat"). In beiden Fällen ist klar, dass der Algorithmus einen Pfad vom Startknoten s der Breitensuche zum Knoten v gefunden hat. Wir betrachten nun den Moment, in dem der Breitensuche-Algorithmus die Nachbarn eines Knotens t betrachtet. Stellt sich heraus, dass ein Nachbar v von t bereits grau gefärbt ist, muss dies von einem anderen, früher betrachteten Knoten t' geschehen sein, welcher *nicht* auf dem Pfad von s über t nach v liegt. Es gibt also zwei nichtidentische Pfade von s nach v. Damit ergibt sich ein Kreis, sodass unser Algorithmus `FALSE` ausgeben soll. In Aufgabenteil c) wird dies gezeigt, und da zeigen wir auch, dass unser Algorithmus `TRUE` ausgibt, wenn G kreisfrei ist.

Unser Algorithmus soll nun wie die Breitensuche verfahren und `TRUE` zurückgeben, sobald er einen bereits grau gefärbten Knoten entdeckt (`else`-Zweig in Zeile 8):

`TesteBaum(`**Graph** $G = (V, E)$`):`

```
1  "Initialisiere BFS mit Startknoten s"
2  while Q ≠ ∅ do
3      u ← head[Q]
4      foreach v ∈ Adj[u] do
5          if color[v] = weiß then
6              color[v] ← grau
7              enqueue(Q, v)
8          else if color[v] = grau then
9              return FALSE
10     dequeue(Q)
11     color[u] ← schwarz
12 return TRUE
```

Wir sehen, dass der Breitensuche lediglich eine Verzweigung mit `return`-Anweisung innerhalb der inneren Schleife hinzugefügt werden muss. Jeder Aufruf dieser Zeilen hat eine worst-case-Laufzeit von $\mathcal{O}(1)$, bedingt durch ihre Position innerhalb der inneren Schleife trägt sie also einen Summanden $\mathcal{O}(|E|)$ bei. Die Laufzeit des Algorithmus nimmt gegenüber der Breitensuche also asymptotisch nicht zu.

b) Ein Graph $G = (V, E)$ heißt *bipartit*, wenn die Knotenmenge V in zwei Teilmengen L und R partitioniert[1] werden kann, sodass es keine Kante $(u, v) \in E$ gibt, für die beide Knoten u und v in L oder beide Knoten in R liegen.

Geben Sie einen Algorithmus `istBipartit(`G`)` an, der genau dann `TRUE` ausgibt, wenn der eingegebene Graph G bipartit ist. Sonst soll der Algorithmus `FALSE` ausgeben.

Lösung: Der hier gegebene Algorithmus wird parallel zur Breitensuche versuchen, eine Partitionierung gemäß der Definition für bipartite Graphen zu konstruieren. Dazu verwendet er nicht mehr nur die zwei Farben *grau* und *schwarz*, sondern merkt sich zusätzlich

[1] Die Knotenmenge V in zwei Teilmengen $L \subseteq V$ und $R \subseteq V$ zu partitionieren, bedeutet, dass die Teilmengen L und R die Eigenschaften $L \cap R = \emptyset$ und $L \cup R = V$ erfüllen. Für jeden Knoten $v \in V$ gilt also **genau eine** der Aussagen $v \in L$ oder $v \in R$.

für jeden Knoten, ob dieser der Menge L oder R hinzugefügt wird. Liegt ein Knoten in der Menge L, so muss jeder seiner Nachbarn in der Menge R liegen und umgekehrt. Wir beginnen unsere Breitensuche also damit, den Startknoten s nicht nur schwarz zu färben, sondern zudem der Menge L hinzuzufügen. Nun soll jeder Knoten, der von einem Knoten aus L heraus entdeckt wird, der Menge R hinzugefügt werden und umgekehrt. Soll ein Knoten der Menge L hinzugefugt werden, liegt aber bereits in R, geben wir `false` aus. Wird ein Graph G durch diesen Algorithmus komplett abgearbeitet, bezeugen die so konstruierten Mengen L und R, dass G bipartit ist, da die Breitensuche jede Kante betrachtet und somit einen Konflikt in der Partitionierung durch L und R mit einer `FALSE`-Ausgabe gemeldet hätte. In Aufgabenteil c) zeigen wir, dass dieser Algorithmus keine falsch-negativen Antworten gibt, dass der Algorithmus also für keinen bipartiten Graphen G `FALSE` ausgibt.

`TesteBipartit`(**Graph** $G = (V, E)$)**:**

1 "Initialisiere BFS mit Startknoten s"
2 $L \leftarrow \{s\}$
3 $R \leftarrow \emptyset$
4 **while** $Q \neq \emptyset$ **do**
5 $u \leftarrow$ head$[Q]$
6 **foreach** $v \in Adj[u]$ **do**
7 **if** $u \in L$ **then**
8 **if** $v \in L$ **then**
9 **return** FALSE
10 $R \leftarrow R \cup \{v\}$
11 **else**
12 **if** $v \in R$ **then**
13 **return** FALSE
14 $L \leftarrow L \cup \{v\}$
15 **if** $color[v] = weiß$ **then**
16 color$[v] \leftarrow$ grau
17 enqueue(Q, v)
18 dequeue(Q)
19 color$[u] \leftarrow$ schwarz
20 **return** TRUE

Auch hier wird die Breitensuche um Verzweigungen konstanten Aufwands innerhalb der inneren Schleife erweitert. Der Test, ob ein Knoten $v \in V$ bereits Element einer der Mengen L oder R ist, lässt sich mit einem in V indizierten Array in konstanter Zeit realisieren. Ihr Beitrag zur Gesamtlaufzeit ergibt also einen Summanden $\mathcal{O}(|E|)$, die Gesamtlaufzeit der zugrundeliegenden Breitensuche bleibt also erhalten.

c) **(Bonus)** Beweisen Sie die Korrektheit der von Ihnen gegebenen Algorithmen.

Lösung:

Test: Baum. In der Beschreibung des Algorithmus wird bereits begründet, dass zwei nicht-identische Pfade von s nach v existieren müssen, wenn unser Algorithmus `FALSE` ausgibt. Diese Pfade sind tatsächlich einfache Pfade. Sei nun t der "letzte gemeinsame Knoten" dieser beiden Pfade, für die Pfade $(s, v_1, \ldots, t, t_1, \ldots, v)$ und $(s, v_1, \ldots, t, t_2, \ldots v)$ soll also $t_1 \neq t_2$ gelten. Dann haben wir zwei Pfade $(t, t_1, \ldots, v)$ und $(t, t_2, \ldots v)$, die bis

auf die Knoten v und t keine gemeinsamen Knoten haben. Der gefundene Zyklus ist nun $(t, t_1, \ldots, v, \ldots, t_2, t)$, Wenn unser Algorithmus `FALSE` ausgibt, gibt es also einen Kreis in G.

Anmerkung: Dass die beiden Pfade kein gemeinsames Endstück haben, zeigt man über einen Widerspruchsbeweis: Ein gemeinsamer Knoten s' vor dem Knoten s, der den Anfang eines gemeinsamen Endstücks der beiden Pfade bildet, wäre bereits früher durch die Breitensuche entdeckt worden. Der Knoten s ist jedoch der erste Knoten, der zum Moment des zweiten Entdeckens bereits gefärbt war.

Zu zeigen ist, dass unser Algorithmus `FALSE` ausgibt, wenn es einen Kreis in G gibt. Sei G ein Graph mit einem Kreis C. Sei y derjenige Knoten im Kreis C, der als letztes grau gefärbt wird, und sei u der Knoten, von dem aus y entdeckt wurde. Dann hat y noch keinen schwarz markierten Nachbarn, bevor u schwarz gefärbt wird. Seien x und z die zwei zu y adjazenten Knoten im Kreis, es existiert im Kreis also der Pfad (x, y, z). Die Knoten x und z sind bereits grau, aber noch nicht schwarz gefärbt, da der Knoten y per Annahme als letzter Knoten im Kreis grau gefärbt wurde, und dieser noch keinen schwarz markierten Nachbarn haben konnte, als er noch weiß gefärbt war. Es gilt nun eine der beiden Aussagen $u \neq x$ oder $u \neq z$, da $x \neq z$. Gelte o.B.d.A. $u \neq x$. Dann werden, nachdem u schwarz gefärbt (also abgearbeitet) wird, die Knoten x und y grau gefärbt sein, und die Kante (x, y) existiert, da sie Teil des Kreises ist. Sobald die Breitensuche einen dieser beiden Knoten betrachtet, wird sie den jeweils anderen grau markierten Knoten in der Adjazenzliste finden und `FALSE` ausgeben.

Test: Bipartit. Wir begründen bereits in der Beschreibung, dass unser Algorithmus keine falsch-positiven Antworten gibt. Zu zeigen ist nun also nur noch, dass unser Algorithmus `TRUE` ausgibt, falls der eingegebene Graph G bipartit ist.

Sei nun der Graph $G = (V, E)$ zusammenhängend und bipartit, und sei durch $L^* \subset V$ und $R^* \subset V$ die Partitionierung gegeben. Sei der Startknoten $s \in V$ o.B.d.A. Element der Menge L^* (gilt $s \in R^*$, tauschen wir die Bezeichner der beiden Mengen im folgenden Beweis aus).

Wir zeigen zwei Aussagen:

- Es gibt (bis auf Vertauschung der kompletten Mengen) nur eine Partitionierung der Knoten V in Teilmengen L und R, die bezeugt, dass der gegebene Graph G bipartit ist.

  Beweis durch Widerspruch: Sei L^* und R^* die eingangs erwähnte Partitionierung, und sei durch L und R eine andere partitionierung gegeben. Sei wieder o.B.d.A. $s \in L$. Sei $v \in V$ ein Knoten mit kleinster Distanz $d[v]$ zu s, der in den Partitionierungen unterschiedlich zugeordnet ist, also $v \in L$, aber $v \in R^*$ (oder umgekehrt). Dann gibt es einen zu v adjazenten Knoten u, sodass $d[u] = d[v] - 1$, also $d[u] < d[v]$, und $u \in R$, da u zu v adjazent ist. Es gilt aber auch $u \in L^*$, da $v \in R^*$. Damit ist u ein Knoten kleinerer Distanz zu s, der in L, R anders zugeordnet ist als in L^*, R^*.

- Unser Algorithmus berechnet die Partitionierung L^*, R^*.

  Beweis durch Widerspruch: Sei durch L^* und R^* die Partitionierung gegeben, und sei durch L und R eine andere Partitionierung gegeben. Sei wieder o.B.d.A. $s \in L$ und $s \in L^*$. Sei $v \in V$ der erste Knoten, der durch unseren Algorithmus falsch zugeordnet wird, also $v \in L$, aber $v \in R^*$ (oder umgekehrt). Dann gibt es einen zu v adjazenten Knoten u, der vor v bearbeitet wird. Da v der Menge L zugeordnet

wird, muss $u \in R$ gelten. Es gilt aber auch $u \in L^*$, da $v \in R^*$. Damit ist auch u falsch durch unseren Algorithmus zugeordnet. Da u vor v bearbeitet wird, ist dies ein Widerspruch zur Annahme, dass v der erste falsch zugeordnete Knoten sei.

Alles zusammen gelten also zu jedem Zeitpunkt der Ausführung $L \subseteq L^*$ und $R \subseteq R^*$, unser Algorithmus wird also keinen Konflikt finden und `TRUE` zurückgeben.

deprecated. *Dieser Beweis trifft die gleichen eingehenden Annahmen wir oben: Der Graph G sei bipartit, die Partitionierung durch L^*, R^* gegeben, und $s \in L^*$.* Wir zeigen nun durch Induktion für $1 \leq k \leq |V| + 1$, dass vor der k-ten Iteration der `while`-Schleife $L \subseteq L^*$, $R \subseteq R^*$ und $L \cap R = \emptyset$ gelten, dass $k - 1$ Knoten schwarz gefärbt sind und dass jeder nicht-weiße Knoten v in $L \cup R$ liegt. Wir zeigen also, dass unser Algorithmus genau die Mengen L^* und R^* berechnet und deswegen nicht `FALSE` ausgeben kann.

Anmerkung: Es gibt für einen bipartiten, zusammenhängenden Graphen $G = (V, E)$ nur eine Partitionierung L, R der Knotenmenge V, die die gewünschten Eigenschaften erfüllt.

I.A. Die Menge L wird auf $\{s\}$ gesetzt, und per Annahme gilt $s \in L^*$, es folgt $L \subseteq L^*$. Ferner gilt bislang $R = \emptyset \subseteq R^*$ und damit auch $L \cap R = \emptyset$. Kein weiterer Knoten ist grau oder schwarz gefärbt, also gilt auch der zweite Teil obiger Aussage.

I.V. Sei $0 \leq k < |V|$. Dann gelten obige Aussagen vor der k-ten Iteration.

I.S. Sei $0 \leq k < |V|$. Zu zeigen ist, dass nach der k-ten Iteration die obige Aussage für $k + 1$ gilt. Sei u der Knoten, der in der k-ten Iteration aus der Warteschlage Q gezogen wird. Dieser Knoten wurde beim Einfügen in Q grau gefärbt, ist nach I.V. also in genau einer der Mengen L oder R, o.B.d.A. gelte $u \in L$. Damit gilt wegen $L \subseteq L^*$ auch $u \in L^*$ und $v \in R^*$ für jeden Nachbarn v von u. In Zeile 19 wird der Knoten u schwarz gefärbt, die Anzahl schwarz gefärbter Knoten steigt also um 1.

Wir zeigen jetzt für jeden Nachbarn v von u, dass nach der k-ten Iteration $v \notin L$ und $v \in R$ gelten. Damit ist dann die Korrektheit unserer Schleifeninvariante gezeigt. Wir unterscheiden zwei Fälle.

- Sei v weiß, also noch nicht der Warteschlange hinzugefügt worden. Dies ist die Situation, in der ein Knoten grau gefärbt wird, und auch erst hier wird der Knoten v einer der Mengen L, R hinzugefügt, es wird also nicht `FALSE` ausgegeben. Wegen $u \in L$ wird v der Menge R hinzugefügt, es gelten also weiterhin $L \cap R = \emptyset$ und $R \subseteq R^*$.
- Sei v grau oder schwarz. Dann ist v Element einer der Mengen L oder R. Da $v \notin L^*$ und nach I.V. $L \subseteq L^*$ gelten, folgt $v \notin L$ und damit $v \in R$, unser Algorithmus gibt also auch nicht `FALSE` aus.

Damit haben wir nun gezeigt, dass der Algorithmus nicht `FALSE` ausgibt, wenn der eingegebene Graph G bipartit ist.