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

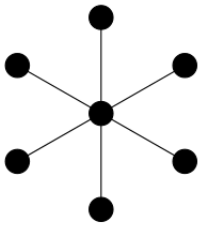
HS 2019

Algorithmen und Komplexität
Lösungsvorschlag zu Übungsblatt 3

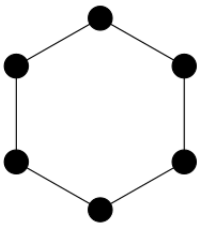
Lösungsvorschlag zu Aufgabe 1

a) Hier gibt es mehrere Möglichkeiten, die einfachste ist wohl der Pfad. Wenn man mit der Breiten- bzw. der Tiefensuche an einem Ende anfängt, gibt es immer nur einen einzigen Knoten, den man als nächstes besuchen kann.

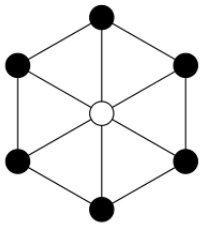
b) Ein Beispiel ist ein sternförmiger Graph (siehe unten). Beginnen wir die Traversierung in der Mitte, dann ist klar, dass jede Breitenordnung eine Tiefenordnung ist und umgekehrt. Wenn wir nicht in der Mitte beginnen, dann ist der nächste Knoten in jedem Fall die Mitte, und wir befinden uns wieder in der zuvorgenannten Situation.



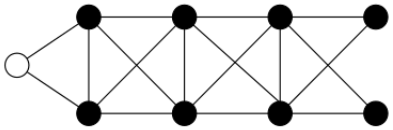
c) Bei der Breitensuche im Kreis sind die ersten beiden besuchten Knoten der linke und der rechte Nachbar des Startknotens. Bei der Tiefensuche hingegen ist der eine Nachbar der erste besuchte Knoten und der andere Nachbar der letzte.



d) Bei der Tiefensuche im Rad, ausgehend vom Zentrum, werden alle äusseren Knoten der Reihe nach besucht. Alle diese Knoten sind direkte Nachbarn des Zentrums, daher können sie auch mit einer Breitensuche in dieser Reihenfolge besucht werden. Bei der Breitensuche können die Knoten jedoch in einer beliebigen Reihenfolge besucht werden, nicht nur in der, welcher die Tiefensuche zwingend vorgibt.



e) Betrachte das untige Bild. Von jedem Knoten mit Abstand i zum Ausgangsknoten ganz links gibt es eine Kante zu jedem Knoten mit Abstand $i+1$ vom Ausgangsknoten. Die Breitensuche besucht zuerst die Knoten mit Abstand 1, dann die mit Abstand 2, usw. Damit gibt es einen für jede Breitenordnung einen Pfad, der die Knoten des Graphen in dieser Reihenfolge durchläuft. Dieser Pfad entspricht gerade einer Tiefenordnung. Es gibt jedoch Tiefenordnungen, die keiner Breitenordnung entsprechen, z.B. zuerst den oberen Kanten folgend nach rechts und danach den unteren Knoten folgend wieder nach links.



Lösungsvorschlag zu Aufgabe 2

a) Ein Graph ist genau dann bipartit, wenn seine Knoten so mit zwei Farben (rot und blau) färbbar sind, dass keine zwei adjazenten Knoten die gleiche Farbe erhalten. Eine solche 2-Färbung nennen wir *zulässig*; die Menge der roten Knoten und die Menge der blauen Knoten ist dann gerade die disjunkte Zerlegung $V_1 \uplus V_2$ der Knotenmenge, die in der Definition von „bipartit" verlangt wird. Beachte, dass so eine Zerlegung in einem zusammenhängenden Graphen eindeutig ist (bis aufs Vertauschen der beiden Mengen). Ist ein Knoten gefärbt müssen alle seine Nachbaren die andere Farbe besitzen und deren Nachbaren wieder die andere Farbe, usw.

Wir nehmen zunächst an, dass der Graph zusammenhängend ist. Wir führen ausgehend von einem beliebigen Startknoten eine Breitensuche durch, und färben den Graphen während der Breitensuche ein. Den Startknoten färben wir o.B.d.A. mit der Farbe rot ein. Von nun an, immer wenn wir während der Breitensuche einen unbesuchten Nachbarn u von einem Knoten v aus besuchen, färben wir in u mit der jeweils anderen Farbe als v. Falls u bereits besucht wurde und die Farben von u und v übereinstimmen geben wir 'nicht bipartit' aus.

Ein Graph ist genau dann bipartit, wenn alle seine Zusammenhangskomponenten bipartit sind. Ist G nicht zusammenhängend, können wir deshalb für jede der Zusammenhangskomponenten einzeln testen, ob sie bipartit ist. Siehe Pseudo-Code für eine genaue Implementierung.

Wir zeigen die Korrektheit des Algorithmus. Stossen wir also im Laufe des Algorithmus auf einen "Widerspruch", d.h. wir besuchen z.B. einen bereits rot eingefärbten Knoten von einem ebenfalls rot gefärbten Knoten aus, dann gibt es keine zulässige 2-Färbung, da wir nie die Wahl gehabt hatten irgendeinen Knoten anders zu färben. Wir können die Breitensuche abbrechen mit der Meldung, dass der Graph nicht bipartit ist. Können wir dagegen die Breitensuche komplett durchführen, ohne dass es ein solches Problem gibt, haben wir eine zulässige 2-Färbung gefunden, da wir im Verlauf der Breitensuche für jede Kante überprüft haben, ob sie zwei Knoten verschiedener Farbe verbindet. Der Graph ist also bipartit.

Die Laufzeit unseres Algorithmus ist $\mathcal{O}(|V| + |E|)$. Die Analyse ist identisch zu der der Breitensuche, beachte dass jedes mal wenn ein Nachbar betrachtet wird lediglich konstant viele Operationen zusätzlich ausgeführt werden.

b) Man überzeugt sich leicht, dass die Bipartition genau dann eindeutig ist (bis auf Vertauschen der Bipartitionsklassen), wenn der Graph zusammenhängend ist.
$\Leftarrow$: Ist der Graph zusammenhängend und legt man für einen Knoten $v \in V$ fest, in welcher Bipartitionsklasse er enthalten sein soll, so folgt für alle übrigen Knoten aus ihrer Distanz (gerade/ungerade) zu v, in welcher Bipartitionsklasse sie enthalten sein müssen. Wir verwenden hier, dass die Distanz zu v für jeden Knoten endlich ist, da G zusammenhängend ist.
$\Rightarrow$: Ist der Graph hingegen nicht zusammenhängend, dann kann man die Bipartition für jede

Algorithm 1 CHECKBIPARTITION(Graph $G = (V, E)$)

```
for all v ∈ V do
  COLOR[v] := nil
end for
Q := new Queue
repeat
  wähle beliebigen Knoten s ∈ V mit COLOR[v] = nil
  COLOR[s] := RED
  Q.Insert(s)
  while not Q.IsEmpty() do
    v := Q.Dequeue()
    for all u ∈ Γ(v) do
      if COLOR[u] = COLOR[v] then
        return "G ist nicht bipartit"
      end if
      if COLOR[v] = RED then
        COLOR[u] := BLUE
      else
        COLOR[u] := RED
      end if
      Q.Insert(u)
    end for
  end while
until COLOR[v] ≠ nil für alle v ∈ V
return "G ist bipartit"
```

Zusammenhangskomponente unabhängig von den anderen Zusammenhangskomponenten wählen und erhält so 'verschiedene' Bipartitionen.

Lösungsvorschlag zu Aufgabe 3

Sei a_v die Anzahl kürzester s-v Pfade in G, d_v die Länge des kürzesten s-v Pfades in G, und sei $D_k \subseteq V$ die Menge der Knoten welche Distanz k zu s haben. Ein kürzester s-v-Pfad, $s, v_1, \ldots, v_{d_v-1}, v$ besteht aus einem Pfad der Länge $d_v - 1$ zu einem Nachbarn von v und dem Knoten v. Wir zeigen unten, dass wir $a[v]$ durch die Summe der $a[u]$, wobei u in D_{d_v-1} und $\Gamma[v]$ ist, berechnen können. Da die Breitensuche zuerst die Knoten D_1, dann die Knoten D_2, usw. besucht, lassen sich die $a[v]$ Werte durch folgende Modifikation der Breitensuche berechnen.

Initialisiere $a[v]$ für alle $v \in V$: $a[v] = 0$ und setze $a[s] = 1$.

Innerhalb der while-Schleife modifizieren wir die for-Schleife folgendermassen:

```
for all v ∈ Γ(v) do
  if d[u] = ∞ then
    d[u] = d[v] + 1
    pred[u] = v
    Q.INSERT(u)
  end if
  if d[u] = d[v] + 1 then
    a[u] = a[u] + a[v]
  end if
end for
```

Korrektheitsbeweis: Es bleibt zu zeigen, dass

$$a_v = \sum_{u \in \Gamma[v] \cap D_{d_v - 1}} a_u \tag{1}$$

gilt. Es ist klar, dass der zweitletzte Knoten in jedem kürzesten s-v-Pfad in $\Gamma[v] \cap D_{d_v-1}$ sein muss (darum gilt $a_v \leq \sum_{u \in \Gamma[v] \cap D_{d_v-1}} a_v$). Gleichzeitig ist auch klar, dass man durch anhängen von v an kürzeste s-u-Pfade zu Knoten in $\Gamma[v] \cap D_{d_v-1}$ lauter verschieden Pfade der Länge d_v erhält (darum gilt $a_v \geq \sum_{u \in \Gamma[v] \cap D_{d_v-1}} a_v$). Daraus folgt Gleichung (1).

Laufzeitanalyse: Analog zur Breitensuche, die Modifikation verändert die Laufzeit lediglich um einen konstanten Faktor.