AD-Übung zum 3. Dezember

Arne Beer, MN 6489196
Merve Yilmaz, MN 6414978
Sascha Schulz, MN 6434677

3. Dezember 2013

1. (a) Der Algorithmus funktioniert nicht mehr. Dies wird anhand dieses Gegenbeispiels deutlich:

```
A = [0,1,4,8,10,13]
value = 1
low = 0
high = 5
// erster Schleifendurchlauf 0 < 5, daher Rumpf ausführen
mid = (0 + 5) / 2 = 2
// A[2] = 4 > value
high = 2 - 1 = 1
// zweiter Schleifendurchlauf 0 < 1, daher Rumpf ausführen
mid = (0 + 1) / 2 = 0
// A[0] = 0 < value
low = 0 + 1 = 1
// dritter Schleifendurchlauf 1 = 1, daher Rumpf nicht ausführen
return not_found
```

Obwohl das Element vorhanden ist, wird zurückgegeben, dass es nicht vorhanden sei. Da es ein Gegenbeispiel gibt, funktioniert der Algorithmus nach der Änderung von `while (low <= high)` zu `while (low < high)` nicht mehr.

(b)

```
binarysearch(a[0..n-1], value) {
low = n - 1
high = 0
while (high <= low) {
    // invariants: value > a[i] for all i < low
                   value < a[i] for all i > high
    mid = (low + high) / 2
    if (a[mid] > value)
       high = mid + 1
       else if (a[mid] < value)
       low = mid - 1
    else
       return mid
   }
    return not_found
}
```

(c) **Formaler Beweis:** Wir müssen beweisen, dass die while-Schleife endet. Angenommen wir befinden uns in Iteration i der while-Schleife.

- Zu Beginn der while-Schleife haben wir `high` $\leq$ `low` (andernfalls hätten wir die while-Schleife nicht betreten).
- Nach dem Ausdruck `mid = (low + high) / 2` gilt `high` $\leq$ `mid` $\leq$ `low`.

- Entweder die Schleife wird durch die Rückgabe von `mid` beendet, womit wir fertig wären.
- Oder sie befindet sich in einer der ersten beiden Fälle des if-Statements. Entweder high wird um mindestens eins erhöht oder low wird um mindestens eins verkleinert, wodurch sich in jedem Schleifendurchlauf die Differenz von `low - high` um mindestens eins verringert.
- Damit gilt `low - high < 0` nach maximal n Iterationen der while-Schleife und die Schleife terminiert.

(d) TODO

2. (a) i. Ein Graph ist 1-färbbar
gdw. jedem Knoten die selbe Farbe zugewiesen werden kann
gdw. es keine zwei Knoten gibt, die sich in Nachbarschaft befinden
gdw. der Graph keine Kanten enthält

ii.
```
IST_2FAERBUNG(G) {
        kanten = E(G)
        valid = true
        farben = new Set()
        foreach kante in kanten {
            knoten1 = kante.knoten1
            knoten2 = kante.knoten2
            farben.add(knoten1.farbe)
            farben.add(knoten2.farbe)
            if (knoten1.farbe == knoten2.farbe) {
                valid = false
            break
            }
    }
return (valid && (farben.getAnzahl() == 2))
}
```

iii. Annahme: n ist die Anzahl der Knoten des Graphen.

Ein Graph ist n-färbbar
gdw. jedem Knoten eine eindeutige Farbe zugeordnet werden kann.
gdw. jeder Knoten einer von n 1-elementigen disjunkten Teilmengen zuordbar ist, wobei innerhalb einer solchen Teilmenge keine Kanten verlaufen.
gdw. kein Knoten existiert, der eine reflexive Kante besitzt
gdw. der Graph schleifenfrei ist.

(b) i. Ein Graph G ist bipartit
gdw. sich die Knoten von G in zwei disjunkte Teilmengen A und B aufteilen, sodass innerhalb einer Teilmenge keine Kanten verlaufen.
gdw. zwei disjunkte Teilmengen A und B existieren, innerhalb denen keine zwei Knoten adjazent sind.
gdw. der Graph 2-färbbar ist.

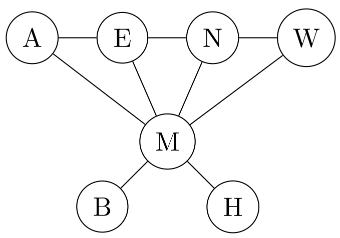
ii.
```
find2coloring(G) {
    for all v in V(G) {
        if v.hasNoColor() {
            v.color = color1
            colorAdjacent(G, v)
        }
    }
}

colorAdjacent(G, v) \
    for all adj in Adj(v) {
        if adj.hasNoColor() {
            adj.color = v.color.counterpart
            colorAdjacent(G, adj)
        }
    }
}
```

iii. Angenommen, der Graph besteht aus n zusammenhängenden Komponenten. Jede Zusammenhangskomponente ist bipartit, d.h. es gibt pro Zusammenhangskomponente 2 Möglichkeiten, diese einzufärben.

Formuliert man dies als Entscheidungsbaum, erhalten wir einen vollständigen binären Baum, bei dem jede Ebene > 0 einer Zusammenhangskomponente zugeordnet ist.

Der Baum besitzt 2^n Blätter, folglich existieren ensprechend viele Pfade und somit Möglichkeiten der 2-Färbung.

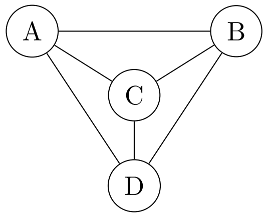
(c) i. Graph zu den aneinandergrenzenden Bezirken:



ii. $Farbe_1$: {M}
$Farbe_2$: {A, N, H, B}
$Farbe_3$: {E, W}

iii. Die Aussage, dass vier Farben minimal sind, besagt nur, dass man es bei einer beliebigen Landkarte schafft diese mit maximal vier Farben zu färben. Es gibt die Obergrenze der nötigen Farben an. Die Regel besagt hingegen nicht, dass immer mindestens vier Farben benötigt werden. Bei einer Landkarte mit nur zwei aneinandergrenzenden Ländern reichen auch zwei Farben. Bei einer Karte mit nur einer zusammenhängenden Fläche ohne angrenzende Flächen reicht sogar eine Farbe.

iv. Ein Land ist von drei Nachbarländern eingeschlossen:



3. Adjazenzliste zu G_1:
$1 \rightarrow 3 \rightarrow 5$
$2 \rightarrow 1 \rightarrow 8$
$3 \rightarrow 4$
4
$5 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 7$
$6 \rightarrow 3 \rightarrow 4 \rightarrow 7$
$7 \rightarrow 2 \rightarrow 6 \rightarrow 8$
$8 \rightarrow 6 \rightarrow 7 \rightarrow 8$

Adjazenzliste zu G_2:
$1 \rightarrow 3 \rightarrow 4 \rightarrow 7$
$2 \rightarrow 3 \rightarrow 5 \rightarrow 6$
$3 \rightarrow 5$
4
$5 \rightarrow 6$
$6 \rightarrow 4$
$7 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6$

(a) Reiheinfolge der Grau-Färbungen bei G_1: 1, 3, 4, 5, 2, 8, 6, 7
Reiheinfolge der Grau-Färbungen bei G_2: 1, 3, 5, 6, 4, 7, 2

(b) Reiheinfolge der Schwarz-Färbungen bei G_1: 4, 3, 1, 7, 6, 8, 2, 5
Reiheinfolge der Schwarz-Färbungen bei G_2: 4, 6, 5, 3, 2, 7, 1

(c) Reihenfolge der besuchten Knoten bei der Breitensuche.
Für G_1: 1, 3, 5, 4, 2, 7, 8, 6
Für G_2: 1, 3, 4, 7, 5, 2, 6

(d) In G_1 existiert keine topologische Sortierung, da Zyklen existieren, ein Gegenbeispiel zum Beweits der Nicht-Existenz: $1 \rightarrow 5 \rightarrow 1$.

Eine topologische Sortierung für G_2: 1, 3, 5, 6, 4, 7, 2

(e) Eine eindeutige topologische Sortierung besteht, wenn ein Hamilton-Kreis für den Graphen existiert. Da der Knoten 1 in G_2 keinerlei eingehende Kante besitzt ist ein solcher Kreis nicht möglich.

(f) ZSK zu G_1: $\{1, 2, 5, 6, 7, 8\}$

ZSK zu G_2: $\emptyset$

4. Grundsätzlich ist für jede der folgenden Teilaufgaben zu beachten:

Ein Modul k ist von einem Modul i abhängig, wenn eine Kante von i nach k existiert. Folglich weißt jedes in Abhängigkeit stehende Modul das Merkmal auf, dass es der eingehende Kantengrade größer als 0 ist.

(a) Pseudo-Code für die Infiltration des Netzwerks:

```
infiltrate(G){
    infiltrated; // List of infiltrated modules
    for all v in V(G) {
        if d_in(v) == 0 {
            infiltrate(v)
            infiltrated.add(v)
        }
    }
    propagateInfiltrationAndDestroy();
}
```

Je nach weiteren Constraints kann in $propagateInfiltrationAndDestroy()$ Tiefen- oder Breitensuche verwendet werden, um dabei jedes Modul statt mit schwarzer Farbe mit der Infiltration zu markieren.

(b) Werden alle Module mit dem eingehnden Kantengrad = 0 eliminiert wird das gesamte Netz eleminiert, da jedes Andere Modul sich in mindestens einer Abhängigkeit zu einem solchen Modul befindet.

Dies kann man sich gut an einer Baumstruktur verdeutlichen. Nur das Wurzelelement erfüllt dieses Kriteritum, während alle Kinderelemente der transitiven Hülle abhängig sind und folglich eliminiert werden. So erlischt jeweils durch die Wurzel der gesamte Baum.

Sollten sich zirkuläte Abhängigkeiten im Geflecht befinden ist die Vorstellung einer Baumstruktur nicht weiter zutreffend. Allerdings benötigt jede zirkuläre Abhängigkeit ein übergeordnetes Modul, welches diese instantiiert - welches dann den selben Gesetzmäßigkeiten folgt.

(c) Die ermittelte Anzahl der Module ist minimal, da ein Modul, welches keine eingehenden Kanten besitzt nicht durch eine Abhängigkeitsbeziehung eliminiert werden kann, folglich ist die direkte Infiltration nötig.