04. Dezember, AD: de Bruyn Kops (6591853), Voroshylov (6590822), Eggert (5690653)

Aufgabe 1

a)

```
1  binsearch (A, value):
2      low = 0
3      high = len(A) - 1
4      # Invarianten:
5      #  value > A[i] for all i < low
6      #  value < A[i] for all i > high
7      while low < high :
8          mid = (low + high) / 2
9          if A[mid] > value :
10             high = mid -1
11         elif A[mid] < value :
12             low = mid + 1
13         else :
14             return mid
15     return not_found
```

Algorithmus wird nicht correct funktionieren.

Beweis:

Sei **A** ein Array der Länge N=1. In diesem Fall werden die Variablen solche Werte haben: low=0, high=0. Die Schleife aber läuft nur dann, wenn $low < high$ ist. In unserem Fall ist schon $low = high$, deshabl springt den Interpreter gerade zur Zeile 15 und macht «return not_found», ohne den einzelnen Wert zu überprüfen. Das also bedeutet dass in diesem Fall Algorithmus nicht correct funktioniert.

b)

```
2      low = 0
3      high = len(A) - 1
4      Invarianten:
5        value > A[i] for all i > high
6        value < A[i] for all i < low
7      while low <= high :
8          mid = (low + high) / 2
9          if A[mid] < value :
10             high = mid -1
11         elif A[mid] > value :
12             low = mid + 1
13         else :
14             return mid
15     return not_found
```

c) Terminierung:

An der Zeile 4 haben wir eine Condition $low \leq high$. Falls das nicht stimmt, fangt die Schleife nicht an zu laufen und die Funktion terminiert. Falls die schleife läuft, haben wir an der Zeile 9 $A[mid] < value$ und an der Zeile 8 $A[mid] > value$ und auch an der Zeile 15 ***return not_found*** . Das bedeutet, dass an der Zeile 15 unsere Schleife terminiert. Falls es nicht passiert hat, ist $low \leq mid \leq high$. In diesem Fall addieren wir entweder 1 zum low oder extrahieren 1 von high. Dessen zufolge haben wir nach dem n Schritte $high - low < 0$ und genau hier wird die Schleife terminieren.

d) Correctheit:

1. Invarianten:
   $value > A[i]\, for\, all\, i < high$
   $value < A[i]\, for\, all\, i > low$
2. Value existiert im Array

   Die Invarianten sind immer «true», das bedeutet, dass ein gesuchtes Element auf keinen Fall sowohl links von *low* als auch rechts von *high* sein kann. Deshalb ist das Elemen immer im unseren Array drin.

3. Critical cases

   Eine Möglichkeit das Element zu verlieren ergibt sich nur dann, wenn $low > high$ ist, bis wir den gesuchten Wert noch nicht gefunden haben. Es ist möglich an der Zeilen 9 und 11.

   Nach der Zeile 8 haben wir insgesamt so eine Konstruktion $low \leq mid \leq high$. Solange $high \geq low + 2$ ist, stimmt immer $low < mid < high$. In diesem Fall stimmt auch $mid - 1 \geq low$ and $mid + 1 \leq high$, das bedeutet dass wir noch $low \leq high$ auch nachdem zeilen 9 und 12 haben. Deshalb läuft auch die Schleife weiter.

   Jetzt möchte ich noch zwei weitere Fälle unterscheiden:

   - $low = high$. Das bedeutet also, dass $low = mid$. Wir aber wissen, dass das Element immer im Array ist, deshalb stimmt auch $A[low] = A[mid] = value$.

Jetzt beenden wir unsere Schleife an der Zeile 14.

- low + 1 = high. In diesem Fall is es auch so dass $low = mid$. Deshalb stimmt auch $A[low] = A[mid] = value$ und wir beenden auch unsere Schleife an der Zeile 14
  Es könnte aber auch so sein, dass $A[low] = A[mid] < value$ oder $A[low] = A[mid] > value$. Dann addieren wir 1 zum *low* weiter und die Schleife terminiert, weil die Aussage $low \leq high$ nicht mehr stimmt. In diesem Fall beenden wir die Funktion an der Zeile 15.

Die Beweise machen klar, dass wenn value im Array ist, endet sich die Funktion immer an der Zeile 14.

04. Dezember, AD: de Bruyn Kops (6591853), Voroshylov (6590822), Eggert (5690653)

Aufgabe 2

a)

(i)

$\underline{\Rightarrow}|$

Ein Graph G = (V,E) ist 1-färbbar.

Dann haben alle Knoten die gleiche Farbe, d.h. $c_1(i) = c_1(j) \forall i, j \in V$.

$\Rightarrow i \nsim j, \forall i, j$ in V, von der Definition von k-färbbar.

$\Rightarrow$ Es gibt keine Kanten zwischen i und j, $\forall i, j \in V$.

$\Rightarrow$ G enthält keine Kanten.

$\underline{\Leftarrow}|$

G enthält keine Kanten.

$\Rightarrow \forall i, j \in V, i \nsim j$.

$\Rightarrow i, j$ müssen nicht verschiedene Farben haben.

$\Rightarrow i, j$ könnten dieselbe Farbe haben.

$\Rightarrow$ G ist 1-färbbar.

$\square$

(ii)

Ein Graph G = (V,E) ist k-färbbar.

Dann $\forall i, j \in V, i \sim j$

$\Rightarrow c_k(i) \neq c_k(j)$.

Wir wählen einer Knoten aus. Dann können wir die Farbe dieses Knotens ändern nach eine Farbe, die es noch nicht in der k-Färbung gibt. Danach haben wir immer noch $i \sim j \Rightarrow c_k(i) \neq c_k(j), \forall i, j \in V$ und unserer Graph ist mit $k + 1$ Farben gefärbt.

Somit ist G auch $(k + 1)$-färbbar.

$\square$

(iii)

Sei n die Zahl der Knoten in Graph G (V,E)

Dann könnten wir jeden Knoten eine eindeutige Farbe geben, sodass für alle $i, j \in V, c_n(i) \neq c_n(j)$. Da wir n Knoten haben, haben wir auch n Farben.

Daher ist G n-färbbar.

$\square$

b)

(i)

$\Rightarrow|$

Sei G bipartit.

Dann $V = V_1 \bigcup V_2$ wobei V_1 und V_2 disjunkt sind, d.h. $V_1 \bigcap V_2 = \varnothing$

sodass $\forall e \in E, e = (i, j)$ wobei $i \in V_1$ und $j \in V_2$.

$\Rightarrow \forall i \in V_1, j \in V_2, i \sim j \Rightarrow c_k(i) \neq c_k(j)$.

Aber da $V_1 \bigcap V_2 = \varnothing$, alle $i \in V_1$ können dieselbe Farbe c_{k1} haben und alle $j \in V_2$ können dieselbe Farbe c_{k2} haben.

$\Rightarrow$ G hat insgesamt zwei Farben, c_{k1} und c_{k2}.

$\Rightarrow$ G ist 2-färbbar.

$\Leftarrow|$

Sei G 2-färbbar.

Dann alle Knoten, die Farbe 1 haben können wir als Teilsatz V_1 nehmen und alle Knoten, die Farbe 2 haben können wir als Teilsatz V_2 nehmen. Da V_1 und V_2 zusammen alle Knoten berücksichtigen, gilt $V = V_1 \bigcup V_2$.

Dann für $i, j \in V_1$ gibt es keine Kante zwischen i und j, und so ähnlich für V_2.

$\Rightarrow V_1 \bigcap V_2 = \varnothing$.

$\Rightarrow$ G ist bipartit.

$\square$

(ii)

Algorithmus der eine 2-Färbung von G findet, wobei G bipartit ist.

FIND2COLORING(G)

```
color(0) = blue                 /* color vertex 0 blue */
for i=0 to |V|-1                /* V is the set of vertices of G */
    for j in adj(i)             /* for all vertices adjacent to vertex i */
        if color(i) = blue      /* color vertex j a different color than vertex i */
            color(j) = red
        else
            color(j) = blue
```

(iii)

Wenn es eindeutige Teilsätze V_1, V_2 gibt, sodass $V = V_1 \bigcup V_2$ und $V_1 \bigcap V_2 = \emptyset$, dann gibt es 2 verschiedene 2-Färbungen für G. In disem Fall sind die 2-Färbungen:

- V_1 hat Farbe 1 und V_2 hat Farbe 2
- V_1 hat Farbe 2 und V_2 hat Farbe 1

Sonst würde es zweimal soviele verschiedene 2-Färbungen als Verknüpfungen von V_1 und V_2 geben.

2. (a) i. Siehe Abbildung 1

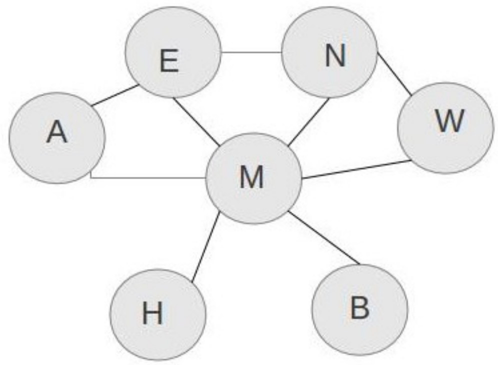

Abbildung 1: Graph zu Aufgabe 2(a)(i)

ii. Siehe Abbildung 2

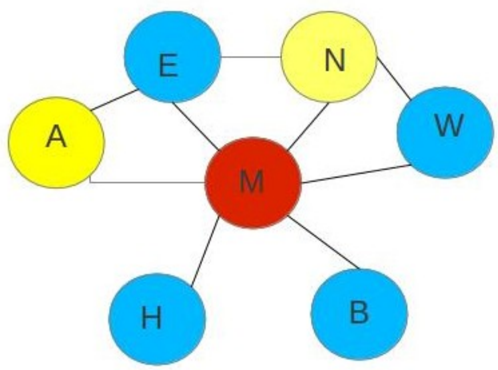

Abbildung 2: Graph zu Aufgabe 2(c)(ii)

iii. Laut des Vier-Farben-Satzes genügen vier Farben um die angrenzenden Länder einer Landkarte unterschiedlich zu färben. Bei der vorliegenden Karte genügen schon 3 unterschiedliche Farben, jedoch besagt der Vier-Farben-Satz lediglich, dass 4 oder weniger Farben benötigt werden.

iv. Siehe Abbildung 3

3. (a) G_1 : 1,5,2,8,7,6,3,4 G_2 : 1,4,7,3,2,5,6

(b) G_1 : 4,3,6,7,8,2,5,1 G_2 : 4,6,5,2,3,7,1

(c) G_1 : 1,(5,3),(2,7,4),(8,6) G_2 : 1,(4,7,3),(6,2),5

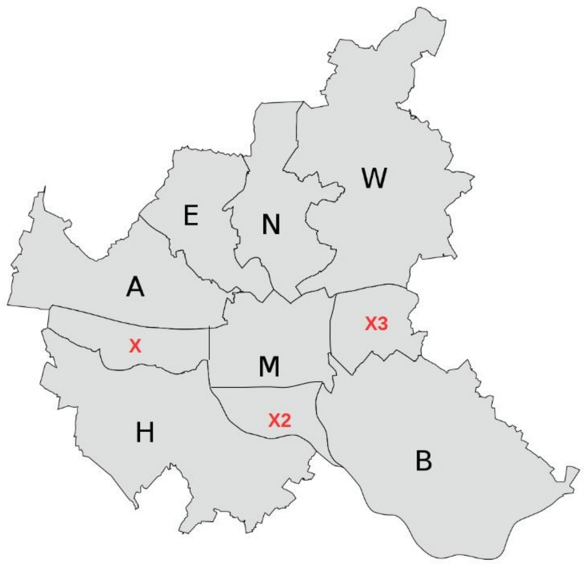

Abbildung 3: Lösung zu Aufgabe 2(c)(iv). Diese Karte ist nicht mehr 3-färbbar.

(d) Für G_1 ist keine topologische Sortierung möglich, da hier mehrere Zyklen existieren, z.B. zwischen den Knoten 1,5,7,8,2. Bei G_2 ist eine topologische Sortierung möglich, und zwar 1,7,2,3,5,6,4.

(e) Die für G_2 gefundene topologische Sortierung ist eindeutig, da nur in der Sortierung von 3 (d) die Bedingung erfüllt werden kann, wonach ein Knoten u links von Knoten v stehen muss wenn eine Kante von u nach v existiert.

(f) G_1 : 1,5,7,2 G_2 : Hier gibt es keine starken Zusammenhangskomponenten.

04. Dezember, AD: de Bruyn Kops (6591853), Voroshylov (6590822), Eggert (5690653)

Aufgabe 4

«High-Level» Pseudocode:

```
eliminieren_mcp(G):

    knoten = Array()
    threads = Array()

    for knote in ermitteln_modules(G, knoten)  :
        knoten.append(knote)

        infiltrieren_module(knote)

        thread = Thread(eliminieren_module(knote));
        threads.append(thread)

    threads_start(threads)
```

Die Hauptidee ist so, dass ich zuerst die längste Pfade ermittle, und dann die erste Knoten der Pfaden nehme um zu infiltrieren und auch zu eliminieren. Natürlich möchte ich auch das ganze MCP so schnell wie möglich ausschalten, und zwar auch so, dass keinen Teil des MCPs was merken könnte. Deshalb mache ich das parallel und benutze «Threads».

Wie ermittle ich die Modulen:

Mit der Hilfe des erweiterten DFS Algorithmus bekomme ich immer die Knoten, die am Anfang des längsten Pfads stehen. Dann vergleiche ich die Anzahl der Childs und nehme den Knote, die höhste Anzahl der Childs haben.

```
ermitteln_modules(G):

    knote = None
    knote_current = None

    for knote in dfs(G, exclude) :

        if knote_current :
            if count(knote.childs) > count(knote_current.childs)
                knote_current = knote
        else :
            knote_current = knote

    return knote_current
```

```
dfs(G, exclude):

    for u in exclude :
        u.color ="black"
        for v in u.childs :
            v.color = "black"

    for knote in G :
        knote.color = "white"

    for knote in G :
        if knote.color is "white" :
            return dfs_visit(G, knote)
```

```
dfs_visit(G, knote):
    knote.color = 'grey'

    if knote.childs :
        knote.depth = knote.depth + 1

        child = None
        child_current = None

        for v in knote.childs :
            if v.color == 'white' :

                child = dfs_visit(G, v)

                if child_current :
                    if child.depth > child_current.depth :
                        child_current = child
                else :
                    child_current = child

    knote.depth += child_current.depth
    knote.color = 'black'

    return knote
```

(a) In diesem Fall suchen wir zuerst nach die längste Modulkette, die auch die höhste Anzahl der Childs haben, und zwar so, bis alle Knoten des Graphs gefunden werden. Dann schalten wir alle Knoten, die auf dem Anfang der Modulketten steht aus. Hierbei kann man verschiedene Fälle unterscheiden:

1. Der Graph hat keine Kanten. In diesem Fall werden alle Knoten infiltriert und parallel aufgeschaltet.
2. Der Graph hat cycles. Im vergleich mit z.b Topologische Sortierung hat der DFS Algorithmus kein Problem damit.
3. Der MCP-Modul hat keine Abhängigkeiten von den anderen Modulen. In diesem Fall wird MCP-Modul allein infiltriert und ausgeschaltet.

(b) Die ermittelte Anzahl der Modulen ist minimal, weil wir immer nach der Modulketten, die miteinander sehr verbunden sind, suchen. Hierbei kann mach folgende Fälle unterscheiden:

1. Der Graph ist vollständig. Hier infiltrieren wir nur einen Modul, weil alle andere schon am anfang als 'black' markiert werden.
2. Der Graph ist connected und acyclisch. Hier brauchen wir warscheinlich ein bisschen mehr Schritte, aber irgendwann finden wir die Wurzel des Baums. Deshalb wird hier auch nur ein Modul infiltriert.
3. Der Graph ist nicht connected. In diesem Fall sind die Modulen nicht verbunden und wir müssen jeder einzelnen Modul allein ausschalten.