TECHNISCHE UNIVERSITÄT BERLIN
Fakultät IV – Elektrotechnik und Informatik
Fachgebiet Neurotechnologie (MAR 4-3)
Prof. Dr. Benjamin Blankertz
Röhr / Schlegel

Algorithmen und Datenstrukturen, SoSe 23

Bitte füllen Sie alle folgenden Felder aus:

Vorname: _____

Nachname: _____

tubIT-Benutzername: _____

Matrikelnummer: _____

Studiengang: _____

Hochschule: _____

Durch meine Unterschrift bestätige ich die Korrektheit obiger Angaben sowie meine Prüfungsfähigkeit und die Anmeldung zur Prüfung!

_____ _____

Ort, Datum Unterschrift

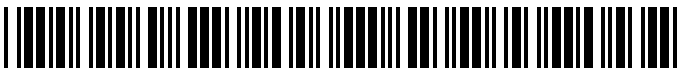
Beachten Sie die folgenden Hinweise!

- Sie brauchen Ihren Namen **nur** auf das Deckblatt zu schreiben. Die restliche Blätter können über die Klausur-ID zugeordnet werden.
- Diese Klausur besteht mit diesem Deckblatt aus den (nummerierten) Seiten **1-11**.
- Am Ende der Klausur befinden sich zwei leere Seiten, die Sie für Notizen verwenden können. Sollten Sie mehr Papier benötigen, können Sie dies von der Aufsicht bekommen. Notieren Sie in diesem Fall die Klausur-ID auf dem Zusatzblatt.
- Notieren Sie Ihre Antworten nur auf dem Blatt (inklusive Rückseite), auf dem die zugehörige Aufgabe steht, da die Aufgaben getrennt korrigiert werden.
- Falls Sie eine Antwort auf ein Zusatzblatt schreiben, markieren Sie dies klar bei der zugehörigen Aufgabe und auf dem Zusatzblatt.
- Geben Sie nur eine Lösung pro Aufgabe ab, streichen Sie alle alternativen Lösungsansätze auf Schmier-/Notizblättern durch.
- Schreiben Sie **nicht** mit roter Farbe, grüner Farbe (Korrekturfarben) oder Bleistift. Diese Lösungen werden nicht bewertet!
- Insgesamt können in der Klausur **100 Punkte** erreicht werden.

Zusatzblätter:

Punktetabelle

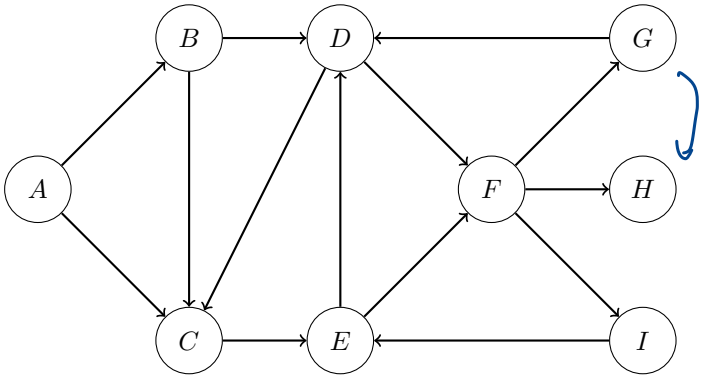
Aufgabe	Punkte		
1			
2			
3			
4			
5			
6			



Aufgabe 1: Tiefensuche ($8 + 2 + 2 + 4 = 16$ Punkte)

Gegeben ist der folgende gerichtete und ungewichtete Graph DG:

A
AB
ABC
ABCE
ABCEDF
ABC EDFG
ABC EDF
ABCEDFH
ABCEDF
ABCEDFI
ABCED



(a) Führen Sie die Tiefensuche auf dem Graphen DG aus. Beginnen Sie die Tiefensuche im **Knoten** A. Notieren Sie alle Knoten in der Reihenfolge, in der sie von der Tiefensuche fertig abgearbeitet werden, d.h. die **Hauptreihenfolge** bzw. **postorder**.

Gehen Sie dabei davon aus, dass bei Wahlmöglichkeit (Tiebreak) die benachbarten Knoten in **alphabetischer Reihenfolge** abgearbeitet werden.

postorder: **G** H I F D E C B A

(b) Ergänzen Sie den oben dargestellten Graphen DG um genau eine gerichtete Kante, sodass Knoten H anstelle von Knoten G als erstes von der Tiefensuche fertig abgearbeitet wird, wenn die Tiefensuche bei Knoten A begonnen wird.

(c) Erklären Sie, wie man die Tiefensuche verwenden kann, um festzustellen, ob in einem gerichteten Graphen ein Knoten v von einem Knoten s aus erreichbar ist.

v ist nicht von s erreichbar, falls v nicht in Postorder/Preorder existiert.

(d) Erklären Sie außerdem, wie man die Tiefensuche verwenden kann, um vom Knoten s aus einen Pfad zu Knoten v zu bestimmen, wenn Knoten v vom Knoten s aus erreichbar ist. Wenn ein solcher Pfad gefunden wurde, hat dieser dann immer minimale Länge? Wenn ja, begründen Sie. Wenn nein, geben Sie ein Gegenbeispiel an.

wir gehen entlangen Preorder von s bis v aus.

Nein, z.B. Preorder: A B C E D F G H I

→ A - B - C ist eine Pfad, aber nicht minimale Länge

A - C ist die Pfad mit minimale Länge

Aufgabe 2: Hashing ($1 + 4 + 10 = 15$ Punkte)

Im Folgenden ist eine noch leere Hashtabelle der Größe 7 gegeben.

Index	0	1	2	3	4	5	6
Key	Alt	Tor		Hut	Eis	Bus	Das

(a) Der Input ist auf die folgenden Wörter beschränkt: Alt, Bus, Das, Eis, Tor, Hut, Mit und Rad. Zur Berechnung der Hashadresse wird die folgende Hashfunktion verwendet: $hash(k) = 3k \mod 7$, wobei k der Hashcode ist. Im folgenden finden Sie die ersten Werte, ergänzen Sie die letzten beiden Werte der Tabelle:

Key	Alt	Bus	Das	Eis	Tor	Hut	Mit	Rad
Hashcode	6	4	2	1	5	8	7	3
Hashadresse	4	5	6	3	1	3	0	2

(b) Fügen Sie die folgenden Wörter in dieser Reihenfolge nacheinander in die obige Hashtabelle ein.

Hut, Bus, Tor, Das, Eis, Alt

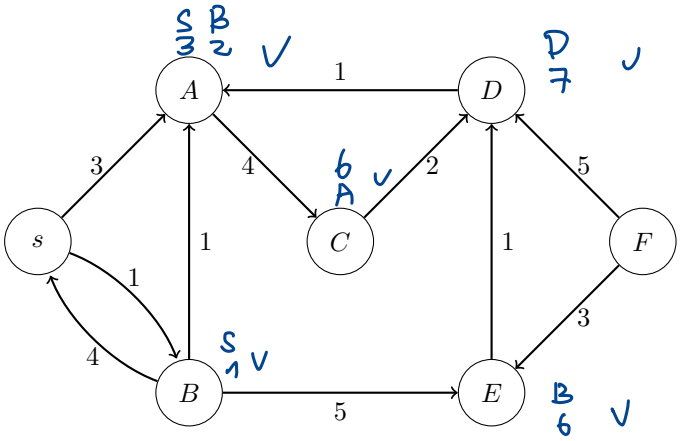
Tragen Sie hierfür als Schlüssel (Key) das Wort ein, das an dieser Stelle eingefügt wird. Kollisionen sollen durch Lineares Sondieren (Linear Probing) mit Inkrement 1 aufgelöst werden.

(c) Überlegen Sie sich, wie man mit Hilfe von Hashtabellen effizient bestimmen kann, welches Wort in einem Buch wie häufig vorkommt.
Hinweis: Man kann mehr als nur den Key in eine HashMap eintragen.
Schreiben Sie eine Methode `fill(String[] buch)`, die eine Hashtabelle `hashmap` so füllt, dass anschließend für jedes in `buch` vorkommende Wort die Häufigkeit des Auftretens aus der Hashtabelle ausgelesen werden kann, ohne weitere Berechnungen durchzuführen.

```
public static HashMap<String, Integer> fill (String[] buch){
    HashMap<String, Integer> hashmap = new HashMap<>();
    for (String wort : buch){
        if (hashmap.containsKey("wort")){
            hashmap.put (wort, hashmap.get(wort) + 1);
        } else {
            hashmap.put(wort, 1);
        }
    }
}
```

Aufgabe 3: SSSP (15 + 4 = 19 Punkte)

Gegeben ist der folgende gerichtete und gewichtete Graph G:



(a) Führen Sie den Dijkstra-Algorithmus auf dem Graphen G aus. Beginnen Sie im Knoten s und notieren Sie die Knoten in der Reihenfolge, in der sie aus der indizierten Vorrangwarteschlange (IndexPQ) entfernt werden, d.h. in der Reihenfolge in der der Dijkstra-Algorithmus die kürzeste Distanz zu diesem Knoten findet. Geben Sie außerdem für jeden Knoten des Graphen, der vom Dijkstra-Algorithmus erfasst wird, die Distanz des kürzesten Pfades vom Startknoten aus zu dem jeweiligen Knoten an.

Gehen Sie dabei davon aus, dass bei Wahlmöglichkeit (Tiebreak) die Knoten in **alphabetischer Reihenfolge** in die Queue hinzugefügt und auch wieder aus der Queue entfernt werden.

Bewertet werden in der unten stehenden Tabelle nur die Spalten **Knoten** und **Distanz zum Startknoten**. Die Spalten *Iteration* und *IndexPQ* müssen nicht ausgefüllt werden, können aber zur Lösung der Aufgabe verwendet werden.

Iteration	Knoten	Distanz zum Startknoten	*IndexPQ*
0	s	0	A:3 B:1
1	B	1	A:2 E6
2	A	2	E:6 C:6
3	E	6	C:6 D:7
4	C	6	D:7
5	D	7	∅

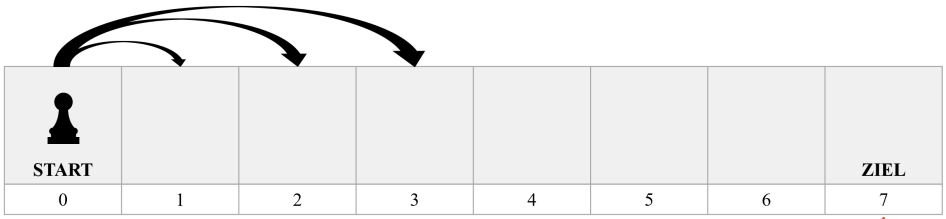
(b) Ändern Sie das Gewicht der Kante (A, C) im Graphen G von 4 zu -4. Kann der Dijkstra Algorithmus den kürzesten Weg zum Knoten D weiterhin effizient bestimmen? Gibt es einen anderen Algorithmus aus der Vorlesung, der in dem geänderten Graphen den kürzesten Weg zum Knoten D effizient finden kann? Begründen Sie.

Nein Bellman-Ford

A → C → D ist eine negative Gewichtet, die nicht geeignet für Dijkstra Algo ist

Aufgabe 4: Java-Programmierung ($4 + 8 + 4 = 16$ Punkte)

In dieser Aufgabe ist die Klasse `Steps` vorgegeben, mithilfe der man die Anzahl der möglichen Zugkombinationen in einem Spiel bestimmen kann. In diesem Spiel kann eine Spielfigur jeden Zug genau 1, 2 oder 3 Felder in Richtung des Ziels ziehen. Die Spielfigur startet auf Feld 0 und das Spiel ist vorbei, wenn die Figur (genau) das Feld $N \in \mathbb{N}$ erreicht hat. Im Folgenden finden Sie ein Beispiel für ein solches Spielfeld für $N = 7$. Es gilt immer: $N \geq 3$.



N=7

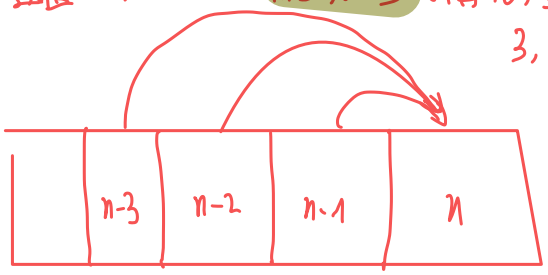
|board| = 7

board_index 0 1 2 3 4 5 6

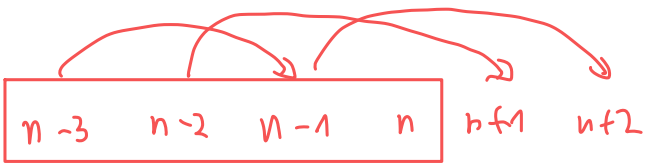
Dieses Problem wird hier mit dynamischer Programmierung gelöst. Dazu wird das Spielfeld `board` als Array repräsentiert und jeder Eintrag `board[n]` des Arrays enthält die Anzahl der **möglichen Zugkombinationen** bis zu dem Feld $n \in \mathbb{N}$. Dabei ist die Reihenfolge der Züge nicht vertauschbar, d.h. zuerst 1 Feld und dann 2 Felder zu ziehen ist eine andere Zugkombination, als erst 2 Felder und dann 1 Feld zu ziehen. Der Vorgabecode enthält keinen Konstruktor und die Methode `countCombinations` zur Bestimmung der Anzahl der möglichen Zugkombinationen ist fehlerhaft.

```
1  public class Steps {
2
3      private int N;
4      private int[] board;
5
6      public Steps(int N) {
7        //TODO
8      }
9
10     public int countCombinations() {
11       //TO BE CORRECTED
12         this.board[0] = 1;
13         this.board[1] = 1;
14         this.board[2] = 4
15         for (int n = 3; n < N; n++) {
16             this.board[n] = this.board[n-1] + this.board[n-2];
17         }
18
19         return this.board[N];
20     }
21
22     public static void main(String[] args) {
23         Steps steps_4 = new Steps(4);
24         System.out.println(steps_4.countCombinations());
25     }
26 }
```

这是n-3, n-2, n-1只移动一步的情况，步长分别为3, 2, 1.



+ this.board[n-3]



这是 n-3, n-2, n-1 移动两步的情况

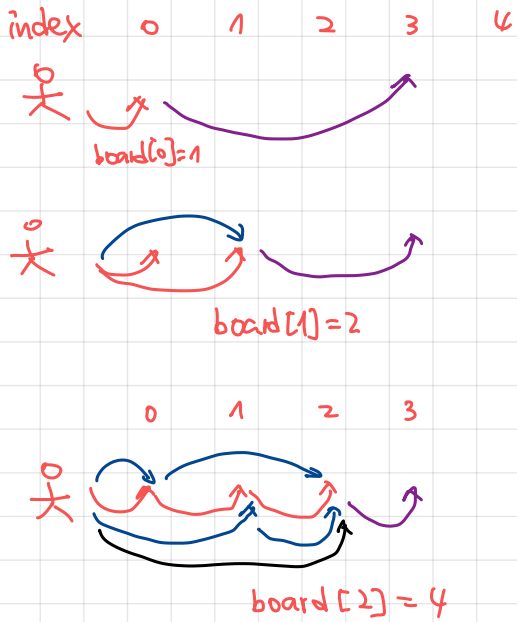
可以观察到，如果在n-3, n-2, n-1存在多步到达 n 的情况，

比如 (n-3) + 2 + 1

我们可以看到 (n-3) 移动两步到达了 (n-1) 的位置，而 n-1 只需1步就到 n 了

→ n-3的多步到达 n 的情况已包含在 (n-2) (n-1) 单步到达 n 的情况

所以，我们只需考虑 n-3, n-2, n-1 一步到 n 的情况

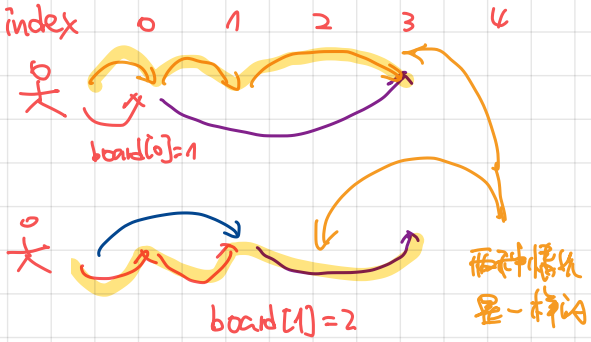


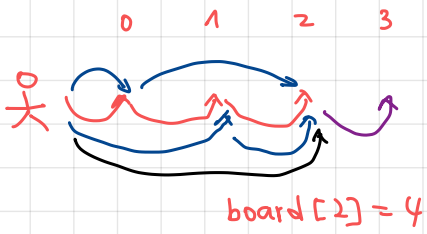
——线为走到3这情

← 首先是board[0]右移3格

← board[1]右移2格

← board[2]右移1格





为什么board[0]只能移动3格，不能移动1,2格或2,1格呢？

→ 这两种情况都board[1]和board[2]包括了

比如左图从board[0]右移1，2格为例

(a) Ergänzen Sie den Konstruktor der Klasse `Steps`. Dieser soll alle Attribute der Klasse korrekt initialisieren. Die Variable N ist der Index des Zielfeldes.

```
1  public Steps(int N) {
2        this.N = N;
3        this.board = new int[N];
4
5
6
7
8
9
10 }
```

(b) Korrigieren und ergänzen Sie die Methode `countCombinations` der Klasse `Steps`, sodass diese die korrekte Anzahl der möglichen Zugkombinationen für das Zielfeld N ausgibt. Nach Ausführen der Methode soll das Array `board` für jeden Index bis inklusive N die Anzahl der möglichen Zugkombinationen bis zu diesem Feld enthalten.

```
1  public int countCombinations() {
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20 }
```

(c) Geben Sie die Laufzeit der Methode `countCombinations` in Abhängigkeit von dem Parameter N in Big-$\mathcal{O}$ Notation an. Geben Sie dabei die kleinste oberere Schranke an und begründen Sie.

O(N)

es gibt eine for-Schleif von 3 bis N

Aufgabe 5: Minimaler Spannbaum $(12 + 6 = 18$ Punkte)

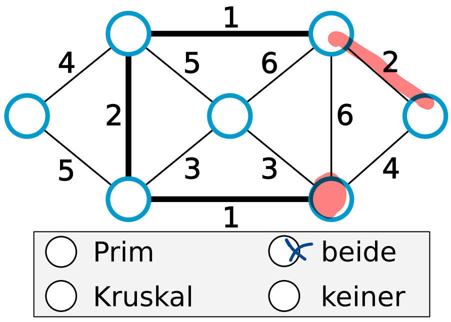
In der folgenden Abbildung sehen Sie sechs mal den gleichen Graphen, wobei jedes mal unterschiedliche Kanten (**breiter**) markiert sind. Hinweis: Lesen Sie zuerst die ganze Aufgabe.

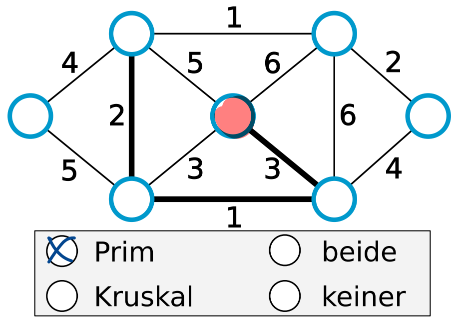
(a) Kreuzen Sie für jeden der sechs Graphen an, ob die markierten Kanten nur durch den Prim Algorithmus, nur durch den Kruskal Algorithmus, durch beide oder durch keinen von beiden ausgewählt worden sein können. Dabei muss der jeweilige Algorithmus nicht bis zum Ende durchgelaufen sein. Es können also auch die Markierungen **während** der Berechnung eines MST abgebildet sein.

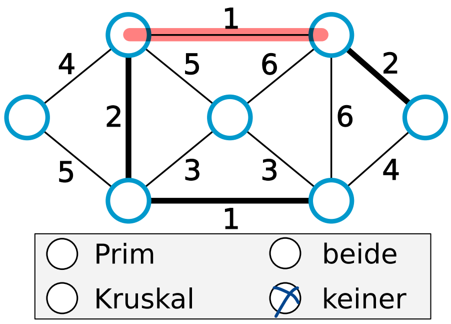
(b) Markieren Sie außerdem in jedem Graphen folgendes:

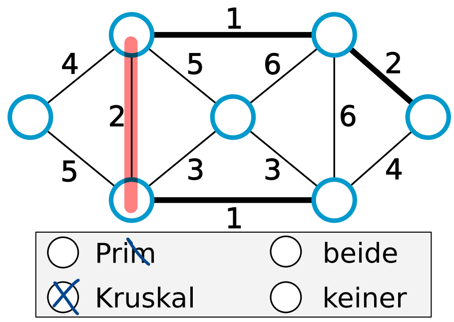
gewählte Algorithmen	Markierung
Kruskal	die Kante, die Kruskal als nächstes auswählen würde
Prim	einen Startknoten für den Prim Algorithmus, der zu der dargestellten Kantenauswahl führt
beide	die Markierungen für *Prim* **und** *Kruskal*, siehe oben
keiner	eine der hervorgehobenen Kanten, die in einer partiellen Lösung von Kruskal nicht ausgewählt worden wäre

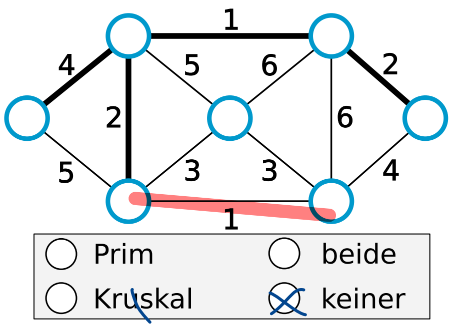
Markieren Sie Kanten, indem Sie das Gewicht umkreisen und Punkte indem Sie sie ausmalen. Wenn Sie eine andere Markierung wählen, z.B. weil Sie etwas verändern wollen, schreiben Sie eine lesbare und eindeutige Legende daneben.

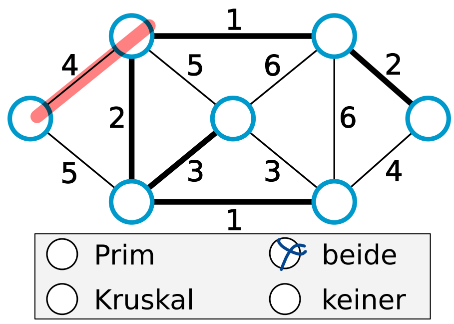








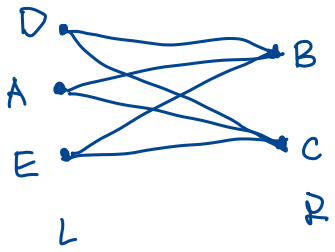




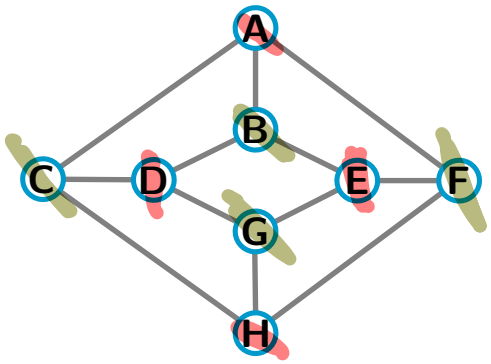
Aufgabe 6: Erkennung bipartiter Graphen ($6 + 4 + 6 = 16$ Punkte)

Ein ungerichteter Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ heißt **bipartiter Graph**, wenn seine Knoten $\boldsymbol{V}$ in zwei disjunkte Teilmengen L und R aufgeteilt werden können, so dass jede Kante in $\boldsymbol{E}$ einen Knoten in L und einen Knoten in R verbindet. (disjunkt heißt: die Knoten von L und R bilden die gesamte Knotenmenge V; L und R haben aber keine gemeinsamen Knoten.).

(a) Skizzieren Sie einen bipartiten Graphen mit **fünf** Knoten, der die folgenden Bedingungen erfüllt: Ein Knoten **A** hat genau zwei Nachbarknoten **B** und **C**. Zwischen **B** und **C** verläuft keine Kante. Jeder von ihnen hat genau zwei weitere Nachbarknoten, die untereinander nicht benachbart sind. Markieren Sie eine Aufteilung der Knoten in L- und R-Knoten gemäß der Definition bipartiter Graphen.



(b) Markieren Sie in dem abgebildeten Graphen alle Knoten mit L bzw. R, so dass dies eine Aufteilung gemäß der Definition bipartiter Graphen darstellt.



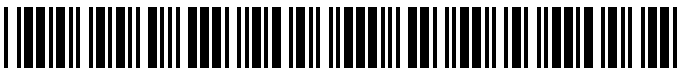
(c) Beschreiben Sie ein Verfahren mit Laufzeit in $\boldsymbol{O}(V + E)$, um festzustellen, ob ein gegebener zusammenhängender Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ ein bipartiter Graph ist. (Es braucht kein Pseudocode zu sein, eine genaue textliche Beschreibung reicht.) Begründen Sie in einem Satz die Laufzeit.

Wir nutzen dfs / bfs um jede Knoten zu durchgehen.
Jeder zwei Knoten färben wir mit unterschiedliche Farben.

G ist nicht bipartiten Graph falls es eine Kante mit 2 gleichen Farben Knoten existiert.
Sonst ist G ein bipartiter Graph.

dfs / bfs hat Zeitkomplexität $O(V+E)$
Jeder Knoten wird einmal gefarbt $O(V)$ und Überprüfung der Farben $O(E)$

insgesamt $O(V+E) + O(V) + O(E) = O(V+E)$



Diese Seite können Sie für Notizen verwenden. Bitte nur im Ausnahmefall für Lösungen verwenden!

Diese Seite können Sie für Notizen verwenden. Bitte nur im Ausnahmefall für Lösungen verwenden!