Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

18. Mai 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 11 FS 16

Lösung 11.1 *Branch-and-Bound.*

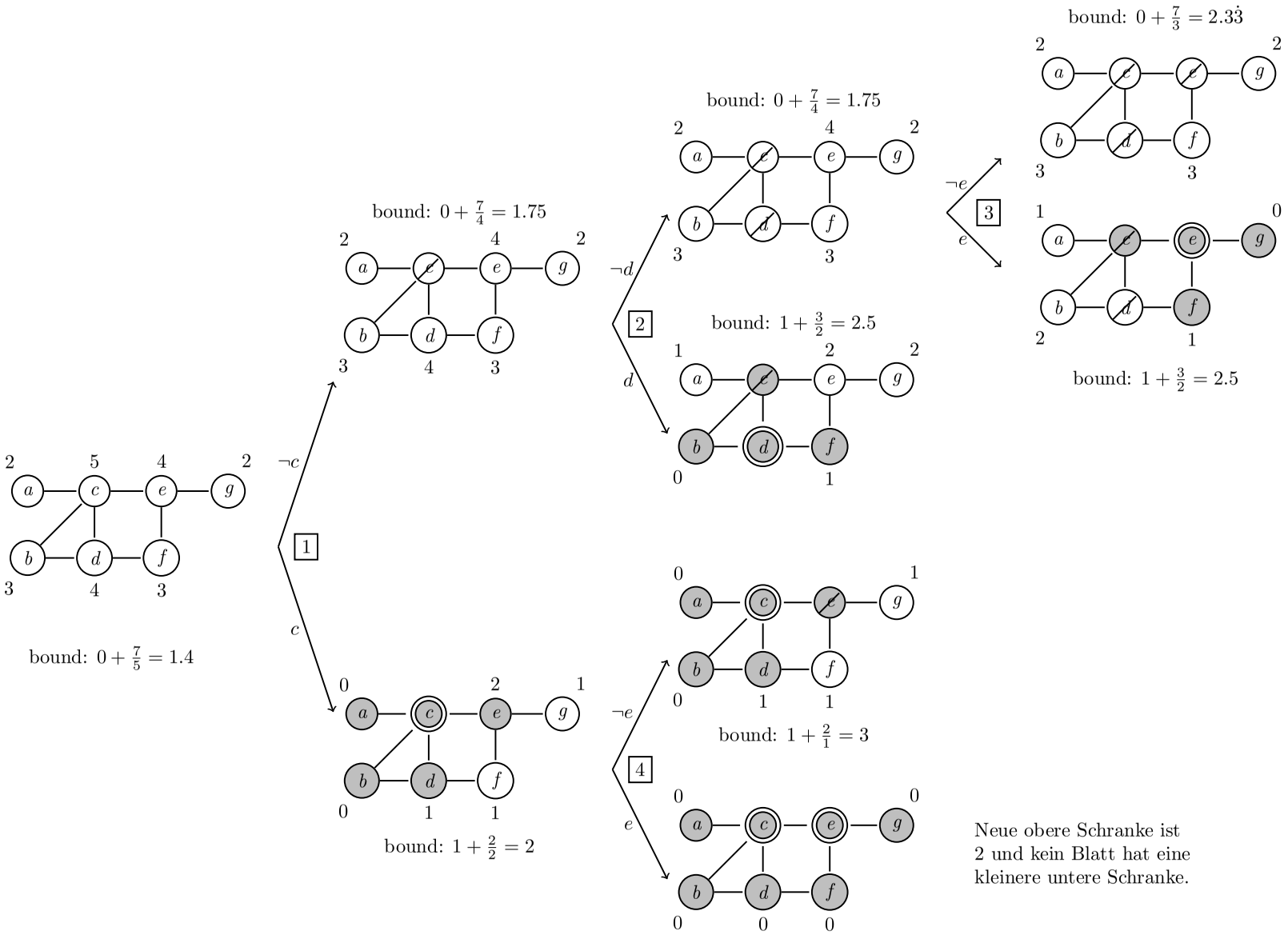
a) Man beachte zunächst, dass in der Vorlesung ausschliesslich gezeigt wurde, wie Branch-und-Bound auf Maximierungsprobleme angewendet werden kann, wir auf diesem Blatt dagegen ein Minimierungsproblem betrachten. Daher muss zu einer Lösung auch keine obere, sondern eine *untere* Schranke angeben werden.

Jede Teillösung wird durch ein Paar (In, Out) mit $In, Out \subseteq V$ und $In \cap Out = \emptyset$ beschrieben. Dabei sei In die Menge all derjenigen Knoten, die definitiv in die dominierende Menge aufgenommen werden sollen, und Out die Menge all derjenigen Knoten, die definitiv nicht in die dominierende Menge aufgenommen werden sollen. Ein Knoten heisst *dominiert*, falls er selbst in In liegt oder einen Nachbarn in In besitzt. Wir definieren δ_v als die Anzahl Knoten, die zusätzlich dominiert werden, wenn wir v auch noch in In aufnehmen (δ_v zählt also auch v mit, falls dieser noch nicht dominiert ist). Natürlich ist $\delta_v = 0$ für alle $v \in In$, da diese Knoten ja bereits Teil der dominierenden Menge sind. Seien ausserdem $\delta_{\max} = \max_{v \in V \setminus Out} \delta_v$ die maximale Anzahl von Knoten, die wir mit einem Knoten aus $V \setminus Out$ dominieren können, und $\bar{D}$ die Menge aller noch nicht dominierten Knoten. Um alle Knoten in $\bar{D}$ zu dominieren, benötigen wir also mindestens $|\bar{D}|/\delta_{\max}$ zusätzliche Knoten. Ausgehend von der gegebenen Teillösung erhalten wir damit $|In| + |\bar{D}|/\delta_{\max}$ als untere Schranke für die Grösse einer dominierenden Menge.

Wir verbessern diese Schranke weiter, indem wir sie auf ∞ setzten, falls ein nicht dominierter Knoten von überhaupt keinem Knoten in $V \setminus Out$ dominiert werden kann. Dann ist unsere Teillösung hoffnungslos und sollte nicht weiter verfolgt werden.

b) Wir wählen jeweils einen Knoten $v \in V \setminus (In \cup Out)$, der möglichst viele neue Knoten dominiert, für den also $\delta_v = \delta_{\max}$ gilt. Wir hoffen, dass diese Heuristik unsere Suche möglichst schnell zum Ziel bringt.

c) Der Entscheidungsbaum sieht wie folgt aus. Die Reihenfolge der Schritte ist jeweils mit Nummern in Kästchen angegeben. Es werden insgesamt 4 Verzweigungen (Branch) durchgeführt, bis eine optimale Lösung mit zwei Knoten gefunden wird. Unsere Lösung ist $In = \{c, e\}$. Wir können abbrechen, sobald wir diese Lösung gefunden haben, da alle verbleibenden unteren Schranken echt grösser als 1 sind, es also keine Lösung mit weniger als 2 Knoten geben kann.



Neue obere Schranke ist 2 und kein Blatt hat eine kleinere untere Schranke.

Lösung 11.2 *Paging-Problem.*

a) Um zu sehen, dass LIFO nicht kompetitiv ist, betrachten wir einen Cache der Grösse $k \geq 2$. Zu Beginn ist der Cache mit den Seiten 1 bis k gefüllt. Wird nun zuerst eine Seite $k+1$ angefragt, so wird die erste Seite aus dem Cache ersetzt. Nach dieser Anfrage befinden sich die Seiten 2 bis $k+1$ im Cache. In den restlichen $n-1$ Abfragen fragen wir abwechselnd die Seite 1 und die Seite $k+1$ an. Bei jeder Anfrage tritt bei LIFO ein Seitenfehler auf. Ein optimaler Algorithmus kommt mit *insgesamt* einem Seitenfehler aus, indem er beim ersten Seitenfehler nicht die erste Seite, sondern eine beliebige andere Seite aus dem Cache ersetzt. Die restlichen Anfragen führen zu keinen weiteren Seitenfehler, da sowohl 1 als auch $k+1$ im Cache sind. Daher ist LIFO nicht kompetitiv.

b) Sei Opt ein optimaler Algorithmus für das Paging-Problem. Da Opt mit demselben Cache startet wie FIFO, wird die erste Phase auch für diesen Algorithmus mit einem Seitenfehler enden. Betrachten wir nun eine beliebige andere Phase P (die letzte Phase ausgenommen) mit genau k Seitenfehlern bei FIFO. Sei s diejenige Seite, die als letzte vor Beginn der Phase zum Cache hinzugefügt wurde. Es genügt zu zeigen, dass k verschiedene Seiten in Phase P angefragt werden. Dann macht Opt auch mindestens einen Seitenfehler da sich der Cache von Opt zu Beginn der Phase von FIFO in höchstens $k-1$ Seiten (s ist sicherlich auch im Cache, da sie gerade angefragt wurde) vom Cache von FIFO unterscheidet.
Wir können durch folgende Argumente zeigen, dass k verschiedene Seiten angefragt werden: Eine Anfrage auf Seite s verursacht während der Phase keine Seitenfehler, da sie erst mit der letzten Anfrage der Phase aus dem Cache entfernt wird. Ausserdem tritt höchstens ein Seitenfehler auf, wenn eine Seite in dieser Phase mehrmals angefragt wird. Daher müssen k verschiedene Seiten angefragt werden, bis s ersetzt wird und die Phase endet.
Also ist FIFO k-kompetitiv.

Lösung 11.3 *Rucksackproblem (**Programmieraufgabe**).*

Der Algorithmus benutzt eine Tabelle $A[\cdot,\cdot]$ mit $n+1$ Zeilen und $W+1$ Spalten. Für $0 \leq i \leq n$ und $0 \leq j \leq W$ repräsentiert der Eintrag $A[i,j]$ den Wert einer optimalen Bepackung, die nur die ersten i Objekte $\{1,\ldots,i\}$ verwenden darf und höchstens Gewicht j besitzt. Nachdem die Tabelle ausgefüllt wurde, enthält der Eintrag $A[n,W]$ genau den Wert einer optimalen Bepackung. Wir beschreiben weiter unten wie die optimale Bepackung selbst ermittelt wird.

Berechnung der Tabelle

Für $i=0$ wird die leere Menge betrachtet. Daher setzen wir $A[0,j]=0$ für jedes j, $0 \leq j \leq W$ (werden keine Objekte benutzt, dann ist der maximale Wert jeder Bepackung 0). Ausserdem setzen wir $A[i,0]=0$ für jedes i, $1 \leq i \leq n$ (wenn das Maximalgewicht 0 beträgt, dann ist der Wert jeder Bepackung erneut 0). Die verbleibenden Einträge können wie folgt berechnet werden:

$$A[i,j] = \begin{cases} A[i-1, j-w_i] + v_i & \text{falls } w_i \leq j \wedge A[i-1,j-w_i]+v_i \geq A[i-1,j] \\ A[i-1,j] & \text{ansonsten} \end{cases} \quad (1)$$

Diese Berechnungs-Vorschrift entspricht den folgenden beiden Fällen:

- Wenn das Objekt i nicht benutzt wird, dann hat die beste Bepackung den Wert $A[i-1,j]$.
- Benutzen wir allerdings Objekt i, so darf das Gewicht w_i nicht das maximale Gewicht j überschreiten. Gilt $w_i \leq j$ und benutzen wir das Objekt i, so ist der maximale Wert einer

Bepackung die Summe des Wertes v_i des neuen Objekts und des Wertes der optimalen Bepackung der ersten $i-1$ Objekte mit einem maximalen Gewicht $j-w_i$.

Da diese Fälle alle Möglichkeiten abdecken, ist $A[i,j]$ als das Maximum dieser beiden Werte definiert. Die Einträge werden nach aufsteigenden i und für gleiche i nach aufsteigendem j berechnet, d.h. in folgender Reihenfolge: $A[1,1], ..., A[1,W], A[2,1], ..., A[2,W], ..., A[n,1], ..., A[n,W]$.

```
public static int[][] computeTable(int n, int[] v, int[] w, int W) {
    int[][] table = new int[n+1][W+1];

    // Initialization
    for(int j=0; j<=W; j++)
        table[0][j] = 0;
    for(int i=1; i<=n; i++)
        table[i][0] = 0;

    // Compute entries of the table
    for(int i=1; i<=n; i++)
        for(int j=1; j<=W; j++) {
            if(j<w[i-1])
                table[i][j]=table[i-1][j];
            else
                table[i][j]=Math.max(table[i-1][j], v[i-1]+table[i-1][j-w[i-1]])
                    ;
        }

    return table;
}
```

Wert der optimalen Bepackung

Nachdem die Tabelle ausgefüllt wurde, enthält der Eintrag $A[n,W]$ genau den Wert einer optimalen Bepackung.

```
public static int computeMaximumValue(int n, int W, int[][] table) {
    return table[n][W];
}
```

Optimale Bepackung durch Rückverfolgung berechnen

Die optimale Bepackung selbst kann durch *Rückverfolgung* ausgehend von $A[n,W]$ ermittelt werden. Wenn $w_n \leq W$ gilt und zusätzlich $A[n,W] = A[n-1, W-w_n] + v_n$ erfüllt ist, dann benutzen wir das Objekt n und fahren mit dem Eintrag $A[n-1, W-w_n]$ fort. Ansonsten benutzen wir das Objekt n nicht und fahren mit $A[n-1,W]$ fort. Dieses Vorgehen wird beendet, wenn alle Zeilen von $A[\cdot,\cdot]$ verarbeitet wurden.

```
public static boolean[] computeOptimalSolution(int n, int[] v, int[] w, int W,
    int[][] table) {
    // Reconstruct the solution
    boolean[] solution = new boolean[n];
    int weight = W;
    for(int i=n; i>=1; i--) {
        solution[i-1] =
            (w[i-1] <= weight) &&
            (table[i][weight] == table[i-1][weight-w[i-1]]+v[i-1]);
```

```
        // Reduce weight limit if object was used
        if(solution[i-1]) weight -= w[i-1];
    }

    return solution;
}
```

Hauptprogramm

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int T = scanner.nextInt();
    for(int t=0; t<T; t++) {
        // Read instance size
        int n = scanner.nextInt();
        // Read weight limit
        int W = scanner.nextInt();
        int[] v = new int[n];
        int[] w = new int[n];
        // Read values of the objects
        for(int i=0; i<n; i++)
            v[i] = scanner.nextInt();
        // Read weights of the objects
        for(int i=0; i<n; i++)
            w[i] = scanner.nextInt();

        // Compute the optimal solution
        int[][] table = computeTable(n, v, w, W);
        int maxvalue = computeMaximumValue(n, W, table);
        boolean[] solution = computeOptimalSolution(n, v, w, W, table);

        // Print the maximum value
        System.out.print(maxvalue);

        // Print the solution
        for(int i=0; i<n; i++)
            if(solution[i])
                System.out.print(" "+(i+1));
        System.out.println();
    }
}
```