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

HS 2019

Algorithmen und Komplexität
Lösungsvorschlag zu Übungsblatt 7

Lösungsvorschlag zu Aufgabe 1

Sei $b_{i,j}$ der maximale Profit, den der Roboter bis zur Position (i, j) sammeln kann, indem er $i-1$ Schritte nach unten und $j-1$ Schritte nach rechts macht. Um zur Position (i, j) zu gelangen muss der Roboter zuvor entweder auf Position $(i-1, j)$ oder $(i, j-1)$ gewesen sein. Es folgt, dass $b_{i,j}$ mit folgender Formel Rekursiv berechnet werden kann:

$$b_{i,j} = a_{i,j} + \max(b_{i-1,j}, b_{i,j-1}) \, .$$

Da zur Berechnung von $b_{i,j}$ lediglich die Werte $b_{i-1,j}$ und $b_{i,j-1}$ benötigt werden, können wir mit zwei For-Schleifen welche über i und j iterieren die gesamte Matrix B berechnen. Für die erste Zeile bzw. Spalte muss die Formel leicht abgeändert werden, da der Roboter von links bzw. oben kommen muss (siehe Pseudo-code).

Um zusätzlich noch den Weg bestehend aus $2n+1$ Feldern auszugeben, speichern wir in einer $(n+1) \times (n+1)$ Matrix C ab ob der Roboter jeweils von oben oder links kam. Falls $b_{i-1,j} \geq b_{i,j-1}$ setze $c_{i,j} = \mathit{oben}$ und sonst $c_{i,j} = \mathit{links}$. Dann können wir in $(n+1, n+1)$ starten und den Weg rückwärts abblaufen und abspeichern, siehe Pseudo-Code.

Korrektheit folgt aus der Korrektheit der Rekursionsformel. Die Laufzeit beträgt $\mathcal{O}(n^2)$, da bei der Berechnung von $b_{i,j}$ für jedes Paar $2 \leq i, j \leq n+1$ konstant viel Zeit benötigt wird (Intialisieren und Ausgabe des Weges braucht $\mathcal{O}(n)$).

Algorithm 1 MARSEXPEDITION (Matrix$A$)

```
{ Initialisierung: }
b_{1,1} = a_{1,1}
for i = 2,...,n+1 do
    b_{i,1} = a_{i,1} + b_{i-1,1}
    c_{i,1} = oben
    b_{1,i} = a_{1,i} + b_{1,i-1}
    c_{1,i} = links
end for

{ Berechnung b_{n+1,n+1}: }
for i = 2,...,n+1 do
    for j = 2,...,n+1 do
        if b_{i-1,j} ≥ b_{i,j-1} then
            b_{i,j} = a_{i,j} + b_{i-1,j}
            c_{i,j} = oben
        else
            b_{i,j} = a_{i,j} + b_{i,j-1}
            c_{i,j} = links
        end if
    end for
end for

{ Bestimmung des Pfades: }
x = n+1
y = n+1
for i = 1...2n+1 do
    w[2n+2-i] = (x,y)
    if c(x,y) = oben then
        y = y-1
    else
        x = x-1
    end if
end for
return Profit: b_{n+1,n+1}, Weg: w[1...2n+1]
```

Lösungsvorschlag zu Aufgabe 2

Wir kürzen monoton steigende Teilsequenz mit MSTS ab. Sei a_i die Länge der längsten MSTS, welche mit x_i endet. Es sei J_i die Menge der Indizes $j < i$ für die $x_j < x_i$ gilt. Falls die Menge J_i leer ist, gilt $a_i = 1$. Ansonsten gilt folgende Rekursionsformel.

$$a_i = 1 + \max_{j \in J_i} a_j \,. \tag{1}$$

Wir beweisen Korrektheit dieser Rekursionsformel. An jede Teilsequenz, welche in x_j mit $j \in J_i$ endet, kann x_i angehängt werden. Daraus folgt $a_i \geq 1 + a_j$ für $j \in J_i$, und somit $a_i \geq 1 + \max_{j \in J_i} a_j$. Betrachte nun das zweitletzte Element x_j der längsten MSTS, welche in x_i endet. j ist sicherlich in J_i. Daraus folgt, dass $a_i \leq 1 + a_j \leq 1 + \max_{j \in J_i} a_j$. Somit folgt (1).

Mithilfe dieser Rekursionsformel können wir a_i für $i = 1, \ldots n$ berechnen (siehe Algorithmus 2). Die Länge der längsten MSTS ist dann gegeben durch $a = \max_{i \in [n]} a_i$, wobei $[n] = \{1, \ldots, n\}$ bezeichnet. Um die längste MSTS auszugeben, speichern wir zusätzlich zur Länge a_i der längsten MSTS, welche in x_i endet, den Vorgänger-Index b_i in dieser MSTS ab. b_i ist gegeben durch

$\arg\max_{j \in J_i} a_j$. Mithilfe der b_i's können wir jeweils zum Vorgänger in der längsten MSTS hüpfen und die MSTS rückwarts durchlaufen (siehe Algorithmus 2): Wenn k der Index des letzten Elements der längsten MSTS ist, so ist b_k, der Index des zweitletzten Elements, b_{b_k} der des drittletzten, usw.

Laufzeitanalyse: Die Ausgabe der Sequenz dauert $\mathcal{O}(n)$. Die Berechnung von a_i und b_i dauert jeweils $\mathcal{O}(n)$. Da dies für $i = 1, \ldots n$ berechnet wird, ergibt sich eine Laufzeit von $\mathcal{O}(n^2)$.

Algorithm 2 LÄNGSTE MONOTON STEIGENDE TEILSEQUENZ

for $i = 1, \ldots, n$ **do**
  { *Bestimmung der Menge* J_i: }
  $J_i = \varnothing$
  for $j = 1, \ldots, i-1$ **do**
    if $x_j < x_i$ **then**
      $J_i = J_i \cup j$
    end if
  end for
  { *Berechnung von* a_i *und* b_i: }
  if $J_i = \varnothing$ **then**
    $a_i = 1$
    $b_i = nil$
  else
    $a_i = 1 + \max_{j \in J_i} a_i$
    $b_i = \arg\max_{j \in J_i} a_i$
  end if
end for

{ *Ausgabe der längsten MSTS* s }
$a = \max_{i \in [n]} a_i$
$k = \arg\max_{i \in [n]} a_i$
$s_a = x_k$
for $i = 1 \ldots a-1$ **do**
  $k = b_k$
  $s_{a-i} = x_k$
end for
return Länge: a, längste MSTS: $s_1, \ldots, s_a$

Lösungsvorschlag zu Aufgabe 3

a) Um einen Algorithmus nach dem Prinzip der dynamischen Programmierung zu entwerfen, definieren wir die Funktion $f(i, b)$ wie folgt. $f(i, b)$ sei der maximale Profit des Rucksackes, wenn das Gewicht höchstens b beträgt und nur die ersten i Objekte zur Verfügung stehen. Für die ersten i Objekte gibt es zwei Möglichkeiten. Entweder das i-te Objekt ist im Rucksack, dann dürfen die restlichen Objekte im Rucksack noch höchstens $b - w_i$ Gewicht haben oder das i-te Objekt ist nicht im Rucksack. Daraus ergibt sich die Rekursion

$$f(i, b) = \max\{f(i-1, b), f(i-1, b-w_i) + p_i\} \ . \tag{2}$$

Der in Algorithm 3 beschriebene Algorithmus benutzt diese Rekursion um eine optimale Lösung zu berechnen.

Zur Korrektheit: Die Korrektheit folgt aus der Korrektheit von (2).

Algorithm 3 KNAPSACK PACKING$(w[1..n], p[1..n], B)$

```
F[0..n, 0..B] := new Array of Integers
for b = 0 to B do
  if w[1] ≤ b then
    F[1, b] := p[1]
  else
    F[1, b] := 0
  end if
end for
for i = 2 to n do
  for b = 0 to B do
    if b ≥ w[i] AND F[i − 1, b − w[i]] + p[i] > F[i − 1, b] then
      F[i, b] := f[i − 1, b − w[i]] + p[i]
    else
      F[i, b] := F[i − 1, b]
    end if
  end for
end for
return F[n, B]
```

Zur Laufzeit: Wie man auf Grund der zwei FOR-Schleifen leicht sehen kann, hat der Algorithmus eine Laufzeit von $O(nB)$. Da wir annehmen können das $B < \sum_{i=1}^{n} w_i \leq n^3$ (sonst können wir alle Objekte in den Rucksack packen) benötigt der Algorithmus nur polynomielle Zeit.

b) Man kann die optimale Lösung speichern, indem man in einem zusätzlichen Feld $L[i, b]$ jeweils speichert, ob für das Maximum $F[i, b]$ das i-te Objekt in den Rucksack gepackt werden muss (also falls $F[i-1, b-w[i]] + p[i] > F[i-1, b]$ dann $L[i, b] = 1$ im Algorithmus 3). Dann kann man aus dem Feld L leicht mit einer Schleife alle Objekte berechnen, die in der optimalen Lösung tatsächlich enthalten sind (Algorithmus 4).

Wir möchten noch erwähnen, dass es in dieser Aufgabe auch möglich ist, für jedes paar (i, b) in $L[i, b]$ direkt die ganze Liste von Objekten zu speichern, die in der Lösung des entsprechend Teilproblems enthalten sind (also $L[i, b] = \{L[i-1, b-w_i], i\}$ oder $L[i, b] = L[i-1, b]$ in Algorithmus 3). Die Laufzeit des Algorithmus erhöht sich dadurch um einen Faktor n (da das kopieren der Listen $O(n)$ Zeit beansprucht), bleibt aber offensichtlich polynomiell in n.

Algorithm 4 KNAPSACK INHALT$(L(), w[1..n], B)$

```
I ← ∅
i ← n
b ← B
while i ≥ 1 do
  if L[i, b] = 1 then
    I ← I ∪ i
    i ← i − 1
    b ← b − w[i]
  else
    i ← i − 1
  end if
end while
return I
```