Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. Fabian Kuhn
P. Bamberger, Y. Maus, P. Schneider

Informatik 2 - Sommersemester 2018
Musterlösung Übungsblatt 9

Abgabe: Montag, 9. Juli, 14:00 Uhr

Aufgabe 1: Währungsarbitrage *(9 Punkte)*

Gegeben sei eine Matrix $(w_{ij}) \in \mathbb{R}^{n\times n}$ mit $w_{ij} > 0$ und $w_{ii} = 1$, welche Tauschkurse zwischen n Währungen darstellt. Der Eintrag w_{ij} gibt dabei an wie viel von Währung j man für eine Einheit von Währung i erhält. Unser Ziel ist es eine Währung $i_1 \in [n]$[1] zu finden für die ein Kreis von Tauschvorgängen $(i_1 \to \ldots \to i_k \to i_1)$ existiert, sodass $w_{i_1,i_2} \cdots w_{i_{k-1},i_k} \cdot w_{i_k,i_1} > 1$ ist. Das bedeutet, dass wir nach dem Tauschen in der gegebenen Reihenfolge mehr von Währung i_1 haben als zuvor, wodurch wir zweifelsfrei unendlich reich werden können.

(a) Angenommen es gäbe keine Kreise wie oben beschrieben. Reduzieren Sie dass Problem eine Tauschfolge $(i_1 \to \ldots \to i_k)$ von i_1 nach i_k mit *maximalem Produkt* $w_{i_1,i_2} \cdots w_{i_{k-1},i_k}$ zu finden, auf das Problem einen Pfad $(i_1 \to \ldots \to i_k)$ mit *minimaler Kantengewichtssumme* auf der Clique $G = (V = [n], E = \{(i,j) : [n] \ni i \neq j \in [n]\})$ mit Gewichtsfunktion $w'(i,j)$ zu finden. *(6 Punkte)*

Hinweise: Definieren Sie die Gewichtsfunktion $w'(i,j) = f(w_{ij})$ auf G als Funktion von w_{ij}, sodass aus dem größten multiplikativen "Pfad" in der Matrix (w_{ij}) der kleinste additive Pfad in G wird (welche Funktion ℓ leistet $\ell(a \cdot b) = \ell(a) + \ell(b)$?). Erklären Sie warum der kürzeste Pfad in G den Gewinn maximiert (bzw. den Verlust minimiert).

(b) Geben Sie einen Algorithmus an, der in $\mathcal{O}(n^3)$ Zeitschritten feststellt, ob ein Kreis $(i_1 \to \ldots \to i_k \to i_1)$ mit $w_{i_1,i_2} \cdots w_{i_{k-1},i_k} \cdot w_{i_k,s} > 1$ existiert. Gehen Sie davon aus, dass Sie auf einen Eintrag der Matrix (w_{ij}) in $\mathcal{O}(1)$ zugreifen können. Begründen Sie die Laufzeit. *(3 Punkte)*

Hinweise: Sie dürfen die obige Reduktion als Blackbox benutzen, falls Sie (a) nicht lösen konnten. Gehen Sie in dem Fall davon aus, dass G einen negativen Kreis hat genau dann wenn es einen Kreis von Tauschvorgängen mit Produkt > 1 in der Matrix (w_{ij}) gibt.

Musterlösung

(a) Wir kennen Algorithmen zum Finden minimaler (additiver) Pfade in einem Graphen. Allerdings muss das Problem in ein Graphenproblem umgewandelt werden, in eine additive Form gebracht werden und von einem Maximierungsproblem in ein Minimierungsproblem umgewandelt werden.

Sei G die Clique über den Knoten $[n]$ mit Gewichtsfunktion $w'(i,j) := -\log(w_{ij})$ für alle $(i,j) \in E$. *(3 Punkte)*

Es gilt, dass $\log(w_{ij}) > -\infty$ ein endlicher Wert ist (aufgrund $w_{ij} > 0$). Allerdings können wir sowohl positive als auch negative Kantengewichte haben. Sei $(i_1 \to \ldots \to i_k)$ ein Pfad von i_1 nach i_k in G und sei $P = (e_1, \ldots, e_{k-1})$ die Menge der zugehörigen Kanten von i_1 nach i_k in G.

[1] $[n] := \{1, \ldots, n\}$

Falls $(i_1 \to \ldots \to i_k)$ das Produkt $\prod_{\ell=1}^{k-1} w_{i_\ell, i_{\ell+1}}$ maximiert, dann ist $\log\big(\prod_{i=1}^{k-1} w_{i_\ell, i_{\ell+1}}\big)$ ebenfalls maximal (da die Logarithmusfunktion auf $\mathbb{R}^+\backslash\{0\}$ streng monoton wachsend ist). Dies wiederum ist genau dann der Fall, wenn $-\log\big(\prod_{i=1}^{k-1} w_{i_\ell, i_{\ell+1}}\big)$ minimal ist. Es gilt die Gleichung

$$-\log\Big(\prod_{i=1}^{k-1} w_{i_\ell, i_{\ell+1}}\Big) = \sum_{i=1}^{k-1} -\log w_{i_\ell, i_{\ell+1}} = \sum_{i=1}^{k-1} w'(e_i). \tag{1}$$

Damit ist $\prod_{i=1}^{k-1} w_{i_\ell, i_{\ell+1}}$ maximal genau dann wenn $\sum_{i=1}^{k-1} w'(e_i)$ minimal ist. Zusammenfassend: Der Pfad der $-\log$(Pfadlänge) in G minimiert, maximiert den Gewinn und umgekehrt. *(3 Punkte)*

(b) Wir reduzieren dass Problem auf ein Graphenproblem mit Graph G und Gewichtsfunktion w' wie oben angegeben. Das geht in Zeit $\mathcal{O}(n^2)$ um die Gewichte zu ermitteln. Dann führen wir den Algorithmus von Bellman-Ford in $\mathcal{O}(|V||E|) = \mathcal{O}(n^3)$ aus um zu ermitteln ob es in G einen negativen Kreis gibt. Ein Kreis mit negativer Summe in G entspricht einem Kreis mit Produkt > 1 bei der Währungsarbitrage (das sieht man anhand von Gleichung 1).

Aufgabe 2: Dynamisches Programmieren: Schachbrett *(8 Punkte)*

Gegeben sei ein Schachbrett mit $n \times n$ Feldern. Wir identifizieren die Felder des Schachbretts mit Tupeln $(x, y) \in [n]^2$, wobei $(1, 1)$ dass Feld ganz links unten, $(n, 1)$ das Feld ganz rechts unten und (n, n) das Feld ganz rechts oben beschreiben soll (die anderen Felder werden entsprechend linear durchnummeriert).
Gegeben sei nun ein Spielstein, der sich initial in der untersten Reihe auf einem Feld $(x, 1)$ für $x \in [n]$ befindet. Wir möchten den Spielstein entlang eines Pfades bis in die oberste Reihe bewegen.
Es gibt aber Einschränkungen bezüglich der Bewegungen des Spielsteines: Sei (x, y) die aktuelle Position des Spielsteins. Dann darf der Spielstein immer nur auf die Felder $(x-1, y+1)$, $(x, y+1)$ oder $(x+1, y+1)$ bewegt werden und auch das nur, falls der Spielstein dann auf dem Schachbrett bleibt.
Zusätzlich sei eine Funktion $v : [n]^2 \times [n]^2 \to \mathbb{R}^+$ gegeben, welche zwei Punkte $p_1, p_2 \in [n]^2$ auf dem Schachbrett auf einen reellen Wert $v(p_1, p_2) \geqslant 0$ abbildet. Wenn die Spielfigur nun von p_1 nach p_2 bewegt wird (gemäß der obigen Einschränkungen), gewinnt man $v(p_1, p_2)$.
Beschreiben Sie einen Algorithmus oder geben Sie Pseudocode an, der nach dem Prinzip des dynamischen Programmierens in $\mathcal{O}(n^2)$ Zeitschritten einen Pfad $(p_1 \to \ldots \to p_n)$ des Spielsteines von einem *beliebigen* Feld p_1 der *untersten* Reihe des Schachbretts zu einem *beliebigen* Feld p_n der *obersten* Reihe findet, welcher den *Gesamtgewinn* $\sum_{i=1}^{n-1} v(p_i, p_{i+1})$ maximiert. Begründen Sie auch kurz die Laufzeit.

Hinweis: Sei $w(x, y)$ der maximale Gewinn den man noch erzielen kann wenn wir von Feld (x, y) aus starten. Stellen Sie zuerst eine rekursive Berechnungsvorschrift für $w(x, y)$ auf. Gehen Sie davon aus, dass $v(p_1, p_2)$ in $\mathcal{O}(1)$ Zeitschritten berechnet werden kann.

Musterlösung

Wie tun wie im Hinweis befohlen und stellen die Rekursionsgleichung auf. Im Basisfall sind wir schon ganz oben auf dem Schachbrett und haben $w(x, n) = 0$ für alle $x \in [n]$. Sei $y < n$ und $1 < x < n$. Dann haben wir

$$\begin{aligned} w(x, y) = \max\Big(&v\big((x, y), (x-1, y+1)\big) + w(x-1, y+1), \\ &v\big((x, y), (x, y+1)\big) + w(x, y+1), \\ &v\big((x, y), (x+1, y+1)\big) + w(x+1, y+1)\Big). \end{aligned} \tag{2}$$

Falls $x = 1, y < n$ ist, können wir nicht schräg links nach oben gehen und erhalten

$$w(1, y) = \max\Big(v\big((x, y), (x, y+1)\big) + w(x, y+1), v\big((x, y), (x+1, y+1)\big) + w(x+1, y+1)\Big). \tag{3}$$

Falls $x = n, y < n$ ist, erhalten wir analog

$$w(n,y) = \max\Big(v\big((x,y),(x-1,y+1)\big) + w(x-1,y+1), v\big((x,y),(x,y+1)\big) + w(x,y+1)\Big). \quad (4)$$

Mit dieser Berechnungsvorschrift können wir nun $w(x,y)$ rekursiv berechnen und dabei mittels dynamischer Programmierung massiv Zeit sparen indem wir bereits berechnete Teillösungen in einem Wörterbuch abspeichern.

Algorithm 1: `w`(x,y)

if $y = n$ **then**
  return 0
if *memo*(x,y) *is not* `Null` **then**
  return `memo`(x,y)
if $1 < x < n$ **then**
  result ← compute `w`(x,y) recursively with Equation 2
else if $x = 1$ **then**
  result ← compute `w`(x,y) recursively with Equation 3
else
  result ← compute `w`(x,y) recursively with Equation 4
`memo`(x,y) ← *result*
return *result*

(4 Punkte)

Den Pfad erhalten wir dann bspw. indem wir alle Werte von $w(x,y)$ vorberechnen und in `memo` speichern und daraus den Pfad rekonstruieren.

Algorithm 2: `computeOptPath`

`memo` ← empty dictionary
for $x \in [n]$ **do**
  `w`$(x,1)$ `// fills memo`
$x_{opt} \leftarrow \arg\max_{x \in [n]} \big(\texttt{memo}(x,1)\big)$
P ← empty sequence of points
P.`append`$(x_{opt}, 1)$
for $y \leftarrow 2$ *to* n **do**
  $x_{opt} \leftarrow \arg\max_{x \in \{x_{opt}-1, x_{opt}, x_{opt}+1\}} \big(\texttt{memo}(x,y)\big)$
  P.`append`$\big\{(x_{opt}, y)\big\}$
return P

(2 Punkte)

Da wir jeden Wert $w(x,y)$ höchstens einmal rekursiv berechnen und ihn danach in `memo` speichern, haben wir höchstens $\mathcal{O}(n^2)$ Rekursionsaufraufe *insgesamt*, also so viele wie es Punkte $(x,y) \in [n]^2$ gibt (asymptotisch). Denn sind einmal alle Werte $w(x,y)$ in `memo` gespeichert, wird bei zukünftigen Aufrufen von `w`(x,y) die Rekursion unterbunden, indem `memo`(x,y) zurückgegeben wird, was dann lediglich $O(1)$ Zeitschritte benötigt. Das Ermitteln des Pfades geht in $\mathcal{O}(n)$. *(2 Punkte)*

Aufgabe 3: Blocksatz-Algorithmus *(13 Punkte)*

In dieser Übung sollen Sie den Blocksatz-Algorithmus aus der Vorlesung implementieren und auf den Text in der Datei `input.txt` anwenden. Eine Zeile soll 80 Zeichen (Buchstaben/ Satzzeichen/ Leerzeichen) enthalten. Ein Zeilenumbruch wird in Python mit "\n" codiert. Der Sonderfall, dass ein Wort mehr Zeichen hat als eine Zeile enthalten darf, muss nicht betrachtet werden.

(a) Implementieren Sie die Funktion `badness`(i, j), welche die badness einer Zeile zurück gibt, die mit dem i-ten Wort beginnt und mit Wort $j - 1$ endet. *(3 Punkte)*

(b) Implementieren Sie den Blocksatz-Algorithmus. Berechnen Sie die optimalen Positionen der Zeilenumbrüche in `input.txt` und die Gesamtbadness. Fügen Sie Ihre Gesamtbadness `erfahrungen.txt` hinzu. *(5 Punkte)*

(c) Damit der ausgebene Text gut aussieht, dürfen sich die Abstände zwischen Wörtern in einer Zeile nicht um mehr als 1 unterscheiden, d.h. wenn x und y die Größen beliebiger Abstände einer Zeile sind, so soll $|x - y| \leqslant 1$ gelten. Implementieren Sie die Funktion `pretty_print`(i, j), welche die Wörter von i bis $j - 1$ in ein Stringobjekt schreibt und Abstände der richtigen Größe zwischen den Wörtern einfügt. Die Methode muss nicht ordnungsgemäß funktionieren, falls die Wörter inklusive der notwendigen Leerzeichen nicht in eine Zeile passen.

Lesen Sie die Datei `input.txt` ein, nutzen Sie Aufgabe a) und b) und schreiben Sie den Text im optimalen Blocksatz (bzgl. Ihrer badness-Funktion) in die Datei `output.txt`. Laden Sie die Datei `output.txt` in den SVN. *(5 Punkte)*

Hinweise: Satzzeichen am Ende eines Wortes zählen zum Wort hinzu. Achten Sie beim Visualisieren Ihres Ergebnisses `output.txt` mit einem Texteditor ihrer Wahl darauf, dass Zeilenumbrüche korrekt angezeigt werden und verwenden Sie nur monospace Schriftarten (alle Zeichen haben die gleiche Breite).

Musterlösung

(b) Die Gesamtbadness beträgt 6746.