Kap. 7.4 Branch-and-Bound
Kap. 7.5 Dynamische Programmierung

Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

25. VO 6. Juli 2006

Überblick

- Wdhlg. (kurz) Branch-and-Bound
  - Anwendung für ATSP

- Dynamische Programmierung
  - Beispiel: 0/1-Rucksackproblem

Motivation

„Warum soll ich heute hier bleiben?"

Dynamische Programmierung taucht immer wieder auf!

„Was gibt es heute Besonderes?"

Überraschungsmoment!

Hier ist es schön kühl!!!

Gerüst für Branch-and-Bound Algorithmen für Maximierungsprobleme

- Berechne eine zulässige Startlösung mit Wert L und eine obere Schranke U für alle möglichen Lösungen.
- Falls $L = U \rightarrow$ STOP: gefundene Lösung ist optimal.

- **Branching:** Partitioniere die Lösungsmenge (in zwei oder mehr Teilprobleme).

- **Search:** Wähle eines der bisher erzeugten Teilprobleme.

- **Bounding:** Berechne für eine dieser Teilmengen T_i je eine untere und obere Schranke L_i und U_i. Sei L der Wert der besten bisher gefundenen Lösung. Falls $U_i \leq L$, braucht man die Teillösungen in der Teilmenge T_i nicht weiter betrachten.

Gerüst für Branch-and-Bound Algorithmen für Minimierungsprobleme

- Berechne eine zulässige Startlösung mit Wert U und eine untere Schranke L für alle möglichen Lösungen.
- Falls $L = U \rightarrow$ STOP: gefundene Lösung ist optimal.

- **Branching:** Partitioniere die Lösungsmenge (in zwei oder mehr Teilprobleme).

- **Search:** Wähle eines der bisher erzeugten Teilprobleme.

- **Bounding:** Berechne für eine dieser Teilmengen T_i je eine untere und obere Schranke L_i und U_i. Sei U der Wert der besten bisher gefundenen Lösung. Falls $U \leq L_i$, braucht man die Teillösungen in der Teilmenge T_i nicht weiter betrachten.

Branch-and-Bound für ATSP

Asymmetrisches Handlungsreisendenproblem, ATSP

- **Gegeben:** Vollständiger **gerichteter Graph** $G=(V,E)$ mit Kantenkosten c_e (geg. durch Distanzmatrix zwischen allen Knotenpaaren)
- **Gesucht:** Tour T (Kreis, der jeden Knoten genau einmal enthält) mit minimalen Kosten $c(T)=\sum c_e$

I.A. gilt hier $c(u,v) \neq c(v,u)$

Gerüst für Branch-and-Bound Algorithmen für ATSP

- **Branching:** Partitioniere die Lösungsmenge, indem jeweils eine Kante (u,v) die ausgewählt wird: 2 neue Teilprobleme:
- T_1: (u,v) wird in Tour aufgenommen; in diesem Fall können auch alle anderen Kanten (u,w) und (w,v) für alle $w \in V$ ausgeschlossen werden.
- T_2: (u,v) wird nicht in Tour aufgenommen.

ATSP-Beispiel

$$D = \begin{pmatrix} \infty & 5 & \textcircled{1} & 2 & 1 & 6 \\ 6 & \infty & 6 & 3 & 7 & \textcircled{2} \\ \textcircled{1} & 4 & \infty & 1 & 2 & 5 \\ 4 & \textcircled{3} & 3 & \infty & 5 & 4 \\ \textcircled{1} & 5 & 1 & 2 & \infty & 5 \\ 6 & \textcircled{2} & 6 & 4 & 5 & \infty \end{pmatrix}$$

○ Zeilenminimumsumme := $1+2+1+3+1+2 = 10$

alternativ: Spaltenminimumsumme := $1+2+1+1+1+2 = 8$

Gerüst für Branch-and-Bound Algorithmen für ATSP

- **Bounding:** Berechne untere Schranke, z.B. durch:
- Für alle Zeilen u der Distanzmatrix berechne jeweils das Zeilenminimum.
- Denn: Zeile u korrespondiert zu den von u ausgehenden Kanten. In jeder Tour muss u genau eine **aus**gehende Kante besitzen: d.h. ein Wert in Zeile u wird auf jeden Fall benötigt.
- Dasselbe kann man für alle Spalten machen: In jeder Tour muss u genau eine **ein**gehende Kante besitzen.
- L := max (Zeilenminsumme, Spaltenminsumme)

Kap. 7.4 Dynamische Programmierung

Dynamische Programmierung

Dynamic Programming, DP

- **Idee:** Zerlege das Problem in kleinere Teil-probleme P_i ähnlich wie bei Divide & Conquer.
- Allerdings: die P_i sind hier abhängig voneinander (im Gegensatz zu Divide & Conquer)
- Dazu: Löse jedes Teilproblem und speichere das Ergebnis E_i so ab dass E_i zur Lösung größerer Probleme verwendet werden kann.

Dynamische Programmierung

Allgemeines Vorgehen:

- Wie ist das Problem sinnvoll einschränkbar bzw. zerlegbar? Definiere den Wert einer optimalen Lösung rekursiv.

- Bestimme den Wert der optimalen Lösung „bottom-up“.

Beispiel: All Pairs Shortest Paths

All-Pairs Shortest Paths (APSP)	
Gegeben:	gerichteter Graph $G = (V,A)$ Gewichtsfunktion $w : A \rightarrow \mathbb{R}_0^+$
Gesucht:	ein kürzester Weg von u nach v für jedes Paar $\mathrm{u}, \mathrm{v} \in \mathrm{V}$

Algorithmus von Floyd-Warshall

Idee: Löse eingeschränkte Teilprobleme:

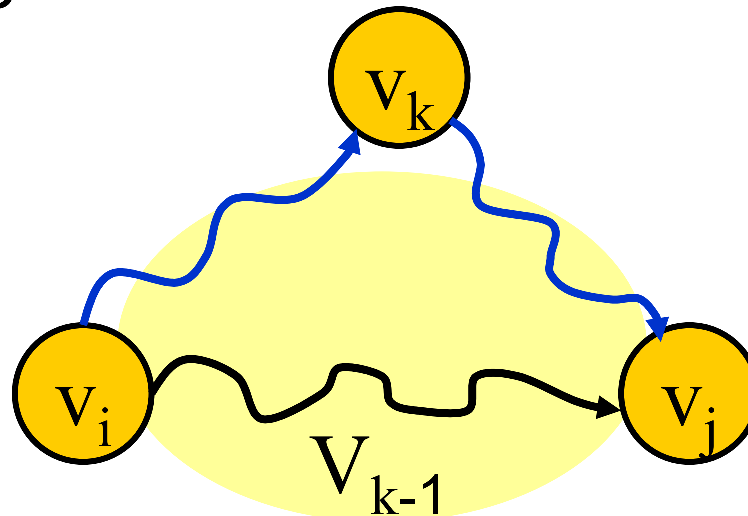
- Sei V_k:={v_1,…,v_k} die Menge der ersten k Knoten
- Finde kürzeste Wege, die nur Knoten aus V_k als Zwischenknoten benutzen dürfen
- Zwischenknoten sind alle Knoten eines Weges außer die beiden Endknoten
- Sei $d_{ij}^{(k)}$ die Länge eines kürzesten Weges von v_i nach v_j, der nur Knoten aus V_k als Zwischenknoten benutzt

Berechnung von $d_{ij}^{(k)}$

Berechnung von $d_{ij}^{(k)}$:

- $d_{ij}^{(0)} = w(v_i,v_j\}$
- Bereits berechnet: $d_{ij}^{(k-1)}$ für alle $1 \leq i,j \leq n$

Für einen kürzesten Weg von v_i nach v_j gibt es zwei Möglichkeiten:



Berechnung von $d_{ij}^{(k)}$

Zwei Möglichkeiten für kürzesten Weg von v_i nach v_j:

- Der Weg p benutzt v_k nicht als Zwischenknoten: dann ist $d_{ij}^{(k)} = d_{ij}^{(k-1)}$
- Der Weg p benutzt v_k als Zwischenknoten: dann setzt sich p aus zwei kürzesten Wegen über v_k zusammen, die jeweils nur Zwischenknoten aus V_{k-1} benutzen: $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

- $d_{ij}^{(k)} := \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
- **Bellmansche Optimalitätsgleichung:** der Wert einer optimalen Lösung eines Problems wird als einfache Funktion der Werte optimaler Lösungen von kleineren Problemen ausgedrückt.

Algorithmus von Floyd-Warshall

ist dynamische Programmierung:

(1) **for** i:=1 **to** n **do**

(2)   **for** j:=1 **to** n **do**

(3) $d_{ij}^{(0)} := w(v_i, v_j\}$

(4) } }

(5) **for** k:=1 **to** n **do**

(6)   **for** i:=1 **to** n **do**

(7)   **for** j:=1 **to** n **do**

(8) $d_{ij}^{(k)} := \min\,(d_{ij}^{(k-1)}\,,\, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

(9) } } }

DP für 0/1-Rucksackproblem

Wie können wir ein Rucksackproblem sinnvoll einschränken?

- Betrachte nur die ersten k der Objekte
- Frage: Was nützt uns die optimale Lösung des eingeschränkten Problems für die Lösung des Problems mit $k+1$ Objekten?
- **Lösung:** Variiere die Anzahl der betrachteten Objekte und die Gewichtsschranke

Dynamische Programmierung für das 0/1-Rucksackproblem

- **Def.:** Es sei $R(i,W)$ für $i \in \{1,\ldots,n\}$ und $W \in \{0,\ldots,K\}$ das eingeschränkte Rucksackproblem mit i Objekten, deren Gewichte $w_1,\ldots,w_i$ und deren Werte $c_1,\ldots.,c_i$ betragen und bei dem das Gewichtslimit W beträgt.
- Wir legen eine Tabelle $T(i,W)$ an, wobei an Position $T(i,W)$ folgende Werte gespspeichert werden:
- $F(i,W)$: der optimale Nutzen für Problem $R(i,W)$
- $D(i,W)$: die dazugehörige optimale Entscheidung über das i-te Objekt

Dynamische Programmierung für das 0/1-Rucksackproblem

- $D(i,W)=1$: wenn es in $R(i,W)$ optimal ist, das i-te Element einzupacken
- $D(i,W)=0$: sonst

- $F(n,K)$ ist dann der Wert einer optimalen Rucksackbepackung

- $F(i,W)$: optimaler Lösungswert für Problem $R(i,W)$
- $D(i,W)$: die dazugehörige optimale Entscheidung über das i-te Objekt

Dynamische Programmierung für das 0/1-Rucksackproblem

Wie können wir den Wert F(i,W) aus bereits bekannten Lösungen der Probleme F(i-1,W´) berechnen?

$$F(i,W) := \max \{ F(i-1,W-w_i)+c_i) , F(i-1,W) \}$$

entweder i-tes Element wird eingepackt: dann muss ich die optimale Lösung betrachten, die in $R(i-1,W-w_i)$ gefunden wurde.

oder i-tes Element wird nicht eingepackt: dann erhält man den gleichen Lösungswert wie bei $R(i-1,W)$.

Dynamische Programmierung für das 0/1-Rucksackproblem

Wie können wir den Wert F(i,W) aus bereits bekannten Lösungen der Probleme F(i-1,W´) berechnen?

$$F(i,W) := \max \{ F(i-1,W-w_i)+c_i) , F(i-1,W) \}$$

Bellmansche Optimalitätsgleichung

Dynamische Programmierung für das 0/1-Rucksackproblem

Wie können wir den Wert F(i,W) aus bereits bekannten Lösungen der Probleme F(i-1,W´) berechnen?

$F(i,W) := \max \{ F(i-1,W-w_i)+c_i) , F(i-1,W) \}$

Randwerte:

- $F(0,W) := 0$ für alle $W:=0,\ldots,K$
- Falls $W<w_i$, dann passt Gegenstand i nicht mehr in den Rucksack für das eingeschränkte Problem auf Gewicht $W \leq K$.

Algorithmus DP für Rucksackproblem

```
(1) for W:=0 to K do F(0,W) := 0   // Initialisierung
(2) for i:=1 to n do {
(3)     for W:=0 to w_i-1 do // Gegenstand i zu groß
(4)         F(i,W) := F(i-1,W)
(5)     for W:=w_i to K do {
(6)         if F(i-1,W-w_i)+c_i > F(i-1,W) then
(7)             F(i,W):= F(i-1,W-w_i)+c_i)
(8)         else F(i,W):= F(i-1,W)
(9) } }
```

Frage: Wie können wir aus dem optimalen Lösungswert die Lösung berechnen?

Wir hatten in $T[i,W]$ gespeichert:

- $D(i,W)$: optimaler Lösungswert für Problem $R(i,W)$
- Wir wissen: $F(n,K)$ enthält den optimalen Lösungswert für unser Originalproblem

Für Problem $R(n,K)$: Starte bei $F(n,K)$:

- Falls $D(n,K)=0$, dann packen wir Gegenstand n nicht ein. Gehe weiter zu Problem $R(n-1,K)$.
- Falls $D(n,K)=1$, dann packen wir Gegenstand n ein. Damit ist das für die ersten $n-1$ Gegenstände erlaubte Gewichtslimit $K-w_n$. Gehe weiter zu Problem $R(n-1,K-w_n)$

Beispiel:
$K=9$
$n=7$

i	1	2	3	4	5	6	7
c_i	6	5	8	9	6	7	3
w_i	2	3	6	7	5	9	4

W \ i	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	6	6	6	6	6	6	6
3	0	6	6	6	6	6	6	6
4	0	6	6	6	6	6	6	6
5	0	6	11	11	11	11	11	11
6	0	6	11	11	11	11	11	11
7	0	6	11	11	11	12	12	12
8	0	6	11	14	14	14	14	14
9	0	6	11	14	15	15	15	15

Algorithmus DP für Rucksackproblem

```
(1) for W:=0 to K do F(0,W) := 0   // Initialisierung
(2) for i:=1 to n do {
(3)     for W:=0 to w_i-1 do // Gegenstand i zu groß
(4)         F(i,W) := F(i-1,W)
(5)     for W:=w_i to K do {
(6)         if F(i-1,W-w_i)+c_i > F(i-1,W) then
(7)             F(i,W):= F(i-1,W-w_i)+c_i)
(8)         else F(i,W):= F(i-1,W)
(9) } }
```

Analyse DP für Rucksackproblem

- Laufzeit der Berechnung der Werte F(i,W) und D(i,W): $O(nK)$
- Laufzeit der Berechnung der Lösung: $O(n)$

- Frage: Ist diese Laufzeit polynomiell?

- Antwort: NEIN, denn die Rechenzeit muss auf die Länge (genauer die Bitlänge) der Eingabe bezogen werden.

Analyse DP für Rucksackproblem

- Antwort: NEIN, denn die Rechenzeit muss auf die Länge (genauer die Bitlänge) der Eingabe bezogen werden.
- Wenn alle Zahlen der Eingabe nicht größer als 2^n sind und $K=2^n$, dann ist die Länge der Eingabe $\Theta(n^2)$, denn: ($2n+1$ Eingabezahlen)*(Kodierungslänge)
- Aber die Laufzeit ist von der Größenordnung $n2^n$ und damit exponentiell in der Inputlänge.

- Wenn aber alle Zahlen der Eingabe in $O(n^2)$ sind, liegt die Eingabelänge im Bereich zwischen $\Omega(n)$ und $O(n \log n)$. Dann ist die Laufzeit in $O(n^3)$ und damit polynomiell.

Analyse DP für Rucksackproblem

- Rechenzeiten, die exponentiell sein können, aber bei polynomiell kleinen Zahlen in der Eingabe polynomiell sind, heißen **pseudopolynomiell.**

- **Theorem:** Das 0/1-Rucksackproblem kann mit Hilfe von Dynamischer Programmierung in pseudopolynomieller Zeit gelöst werden.

Analyse DP für Rucksackproblem

- Das besprochene DP-Verfahren heißt **Dynamische Programmierung durch Gewichte**.
- Denn wir betrachten die Lösungswerte als Funktion der Restkapazitäten im Rucksack.

- Durch Vertauschung der Rollen von Gewicht und Wert kann auch **Dynamische Programmierung durch Werte** durchgeführt werden.
- Dort betrachtet man alle möglichen Lösungswerte und überlegt, wie man diese mit möglichst wenig Gewicht erreichen kann.

Analyse DP für Rucksackproblem

- In der Praxis kann man Rucksackprobleme meist mit Hilfe von DP effizient lösen, obwohl das Problem NP-schwierig ist (da die auftauchenden Zahlen relativ klein sind).

ENDE