Informatik II
Greedy-Algorithmen

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Erinnerung: Dynamische Programmierung

- Zusammenfassung der grundlegenden Idee:
  - Optimale Sub-Struktur: optimale Lösung des Problems besteht aus optimalen Lösungen der Unterprobleme
  - Sich überschneidende Unterprobleme: insgesamt wenige Unterprobleme, viele wiederkehrende Instanzen dieser Unterprobleme
  - Löse bottom-up, erstelle Tabelle mit gelösten Unterproblemen, die zur Lösung größerer benötigt werden
- Variationen:
  - „Tabelle" kann 3-dimensional, dreieckig, ein Baum usw. sein
  - Bottom-Up oder Top-Down

Greedy-Algorithmen

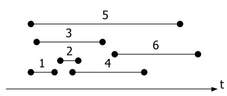
- greedy = gierig
- Grundidee:
  - konstruiere Lösung schrittweise (iterativ)
  - wähle in jedem Schritt die am besten erscheinende Alternative
  - eine Entscheidung wird nie revidiert ("blicke nicht zurück")
  - Hoffnung: eine lokal optimale Lösung führt zu einer global optimalen Lösung
- Dynamische Programmierung kann "Overkill" sein, Greedy-Algorithmen sind oft einfacher zu implementieren
- Greedy-Algorithmen sind nicht immer korrekt/optimal/gut

Activity-Selection-Problem

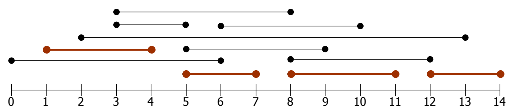
- Beispiel: in einem Vergnügungspark das "Meiste mitnehmen"
  - Eine Karte ermöglicht das Benutzen aller Attraktionen
  - Attraktionen starten und enden zu unterschiedlichen Zeiten
  - Ziel: so viele Attraktionen wie möglich besuchen (eine anderes Ziel wäre, so viel Zeit wie möglich in Attraktionen zu verbringen)

⇒Activity-Selection-Problem

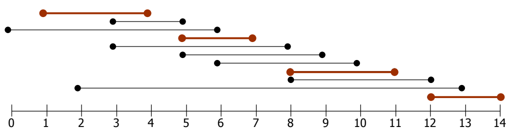
- Formal: sei $S = \{ (s_i, f_i) \mid i = 1 \ldots n \}$ eine Menge von n Aktivitäten
  - s_i / f_i = Startzeit / Endzeit von Aktivität i
  - Aufgabe: finde größte Menge $A \subseteq S$ mit kompatiblen Aktivitäten
  - oBdA sei $f_1 \leq f_2 \leq \ldots \leq f_n$
    - Falls nicht: sortiere Aktivitäten in $O(n \log n)$ oder $O(n)$ gemäß f_i



Beispiel



Wieviel Aktivitäten können ausgeführt werden?



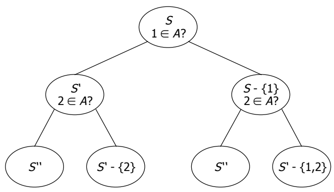
Wie lässt sich Korrektheit beweisen?

Optimale Unterstruktur

- Sei A eine optimale Lösung
- Behauptung:
  Sei k die minimale Aktivität in A (d.h. genau die, die die früheste Endzeit hat), dann ist $A \setminus \{k\}$ eine maximale Lösung für
  $S' = \{i \in S : s_i \geq f_k\}$
  - in Worten: wenn Aktivität $a_1 = (s_k, f_k) \in S$ (richtig) gewählt ist, reduziert sich das Problem darauf, die maximale Lösung für diejenigen Aktivitäten $S' \subseteq S$ zu finden, die zu Aktivität $a_1 \in S$ "passen"
- Beweis:
  - Ann.: wir finden B = maximale Lösung zu S' mit $|B| > |A \setminus \{k\}|$
  - dann ist $B \cup \{k\}$ passend und eine maximale Lösung zu S
  - $|B \cup \{k\}| > |A|$
  - A war nicht maximale Lösung

Wiederkehrende Unterprobleme

- Betrachte rekursiven Algorithmus, der alle verträglichen Untermengen untersucht, um die maximale Menge zu finden
- Beachte die sich wiederholenden Unterprobleme:



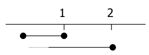
- Dynamische Programmierung? Memoisierung? Ja, aber …

Greedy-Choice-Eigenschaft

- Actitity-Selection-Problem zeigt auch die Greedy-Choice-Eigenschaft:
  - lokal optimale Auswahl ⇒ global optimale Lösung

→Lemma:
wenn S ein, nach der Endzeit sortiertes, Activity-Selection-Problem ist, dann ex. eine optimale Lösung $A \subseteq S$, so daß $\{ (s_1, f_1) \} \in A$

- Beweisskizze:
  - wenn eine optimale Lösung B existiert, die $\{ (s_1, f_1) \}$ nicht enthält, kann die erste Aktivität in B (z.B. (s_2, f_2)) immer durch (s_1, f_1) ersetzt werden
  - gleiche Anzahl an Aktivitäten, also optimal



Greedy-Algorithmus für das ASP

- Eigentlicher Algorithmus ist einfach:
  1. sortiere Aktivitäten nach Endzeit
  2. lege die erste Aktivität fest (also (s_1, f_1)), d.h. diejenige, die am frühesten wieder endet)
  3. entferne alle Aktivitäten aus der sortierten Liste, die vor der Endzeit von f_1 starten (also nicht kompatibel sind)
  - wiederhole, bis keine Aktivitäten mehr übrig sind
- Intuition ist noch einfacher: nimm immer die Aktivität mit der geringsten Dauer, die gerade als nächstes beginnt
- Implementierung:

```
sortiere s & f bzgl f[*]
A = []  # solution array
i = 0   # last finished a.
for k in range( 1, n ):
    if s[k] >= f[i]:
        # "exec" this a. next
        A.append( (s[k],f[k]) )
        i = k
return A
```


Erinnerung: Knapsack-Problem

- 0-1-Knapsack-Problem:
  - ein Dieb muß aus n Gegenständen wählen, wobei der i-te Gegenstand den Wert v_i und das Gewicht w_i besitzt
  - maximiere den Gesamtwert bei vorgegebenem Höchstgewicht W
    - w_i und W sind Ganzzahlen
    - "0-1": jeder Gegenstand muß komplett genommen oder dagelassen werden
- Abwandlung: fraktionales KP (*fractional KP*)
  - Dieb kann Teile von Gegenständen nehmen
  - Z.B.: Goldbarren beim 0-1-Problem und Goldstaub beim fraktionalen Problem

Fraktionales Rucksack-Problem

- Gegeben: n Gegenstände, i-ter Gegenstand hat Gewicht w_i und Wert v_i, Gewichtsschranke W
- Gesucht: $q_1, \ldots, q_n \in [0,1]$ mit $\sum_{i=1}^{n} q_i w_i \leq W$

  und maximalem Wert $\sum_{i=1}^{n} q_i v_i$

```
# Ann.: Array G enthält Instanzen der Klasse Item
# Jedes Item G[i] hat Instanz-Var.s G[i].w und G[i].v
def fract_knapsack( G, w_max ):
    sortiere G absteigend nach G[i].v/G[i].w
    w = 0              # bislang "belegtes" Gesamtgewicht
    for i in range( 0, len(G) ):
        if G[i].w <= w_max - w:
            q[i] = 1
            w += G[i].w
        else:
            q[i] = (w_max - w) / G[i].w
            w = w_max
    return q
```

Aufwand: $O(n) + \underbrace{O(n \log n)}_{\text{Sortieren}}$

Korrektheit

- Hier nur Beweisidee
- Greedy-Algorithmus erzeugt Lösungen der Form

  $(1, \ldots, 1, q_*, 0, \ldots, 0)$
- Ann.: es ex. bessere Lösung $(q'_1, \ldots, q'_m)$; diese hat die Form

  $(1, \ldots, 1, q'_*, \ldots, q'_*, 0, \ldots, 0)$
- Zeige, daß man aus dieser Lösung eine andere Lösung $(q''_1, \ldots, q''_m)$ konstruieren kann, die dasselbe Gewicht hat, aber größeren Wert, und mehr 1-en oder mehr 0-en

- Achtung: Der Greedy-Algorithmus funktioniert nicht für das 0-1-Rucksack-Problem
- Gegenbeispiel: $W = 50$

Gegenstand	Gewicht	Wert	v_i / w_i
1	10	60	6
2	20	100	5
3	30	120	4

  - Greedy: 1 × G1 + 1 × G2
  - optimal: 1 × G2 + 1 × G3

Elemente der Greedy-Strategie

- Greedy-Choice-Eigenschaft: global optimale Lösung kann durch lokal optimale (greedy) Auswahl erreichet werden
- Optimale (Greedy-)Unterstruktur:
  - Eine optimale Lösung eines Problems enthält eine optimale Lösung eines Unterproblems
  - M.a.W.: es ist möglich zu zeigen: wenn eine optimale Lösung A Element s_i enthält, dann ist $A' = A \setminus \{s_i\}$ eine optimale Lösung eines kleineren Problems (ohne s_i und evtl. ohne einige weitere Elemente)
    - Bsp. ASP: $A \setminus \{k\}$ ist optimale Lösung für S'

- Optimale (Greedy-)Unterstruktur (Forts.):
  - Man muß nicht alle möglichen Unterprobleme durchprobieren (wie bei Dyn.Progr.) — es genügt, das "nächstbeste" Element in die Lösung aufzunehmen, wenn man nur das richtige lokale(!) Kriterium hat
    - Bsp. ASP: wähle "vorderstes" f_i
  - Möglicherweise ist eine Vorbehandlung der Eingabe nötig, um die lokale Auswahlfunktion effizient zu machen
    - Bsp. ASP: sortiere Aktivitäten nach Endzeit

Abstrakte Formulierung des Algos

- Die Auswahl-Funktion basiert für gewöhnlich auf der Zielfunktion, sie können identisch sein, oft gibt es unterschiedliche plausible
  - Bsp. ASP:
    - Zielfunktion = Anzahl Aktivitäten (Max gesucht)
    - Auswahl-Funktion = kleinstes $f_i \in S'$

```
# C = Menge aller Kandidaten
# select = Auswahlfunktion
S = []          # Menge mit Lösung
while not solution(S) and C != []:
    x = Element aus C, das select(x) maximiert
    C = C \ x
    if feasible(S,x):     # is x compatible with S?
        S += x
if solution(S):
  return S
else:
  return "keine Lösung"
```

Scheduling in (Betriebs-) Systemen

- Ein einzelner Dienstleister (ein Prozessor, eine Gaspumpe, ein Kassierer in einer Bank, usw.) hat n Kunden zu bedienen
- Die Zeit, die für jeden Kunden benötigt wird, ist vorab bekannt: Kunde i benötigt die Zeit t_i, $1 \le i \le n$
- Ziel:
  - Gesamtverweildauer aller Kunden im System minimieren
  - M.a.W.: Minimierung der Funktion

$$T = \sum_{i=1}^{n} (\text{ Zeit im System für Kunde } i)$$

Beispiel

- Es gibt 3 Kunden mit $t_1 = 5$, $t_2 = 10$, $t_3 = 3$

Reihenfolge			T		
1	2	3	5 + (5 + 10) + (5 + 10 + 3)	= 38	
1	3	2	5 + (5 + 3) + (5 + 3 + 10)	= 31	
2	1	3	10 + (10 + 5) + (10 + 5 + 3)	= 43	
2	3	1	10 + (10 + 3) + (10 + 3 + 5)	= 41	
3	1	2	3 + (3 + 5) + (3 + 5 + 10)	= 29	← optimal
3	2	1	3 + (3 + 10) + (3 + 10 + 5)	= 34	

Algorithmenentwurf

- Betrachte einen Algorithmus, der den optimalen Schedule Schritt für Schritt erstellt
- Angenommen, nach der Bedienung der Kunden $i_1, \ldots, i_m$ ist Kunde j an der Reihe; die Zunahme von T auf dieser Stufe ist

$$t_{i_1} + \ldots + t_{i_m} + t_j$$

- Um diese Zunahme zu minimieren, muß nur t_j minimiert werden
- Das legt einen einfachen Greedy-Algorithmus nahe: füge bei jedem Schritt denjenigen Kunden, der die geringste Zeit benötigt, dem Schedule hinzu
  - "shortest job first"
- Behauptung: Dieser Algorithmus ist immer optimal

Optimalitätsbeweis

- Sei $I = (i_1, \ldots, i_n)$ eine Permutation von $\{1, 2, \ldots, n\}$
- Werden die Kunden in der Reigenfolge I bedient, dann ist die Gesamtverweildauer für alle Kunden zusammen

$$\begin{aligned} T &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_1}) + \cdots \\ &= nt_{i_1} + (n-1)t_{i_2} + (n-2)t_{i_1} + \cdots \\ &= \sum_{k=1}^{n} (n-k+1)t_{i_k} \end{aligned}$$

- Jetzt wird angenommen, daß in I zwei Zahlen a, b gefunden werden können, mit $a < b$ und $t_{i_a} > t_{i_b}$
- Durch Vertauschung dieser beiden Kunden ergibt sich eine neue Bedienungsreigenfolge I', die besser ist, weil

$$T(I) = (n-a+1)t_{i_a} + (n-b+1)t_{i_b} + \sum_{\substack{k=1\\k\neq a,b}}^{n} (n-k+1)t_{i_k}$$

$$T(I') = (n-a+1)t_{i_b} + (n-b+1)t_{i_a} + \sum_{\substack{k=1\\k\neq a,b}}^{n} (n-k+1)t_{i_k}$$

$$\begin{aligned} T(I) - T(I') &= (n-a+1)(t_{i_a} - t_{i_b}) + (n-b+1)(t_{i_b} - t_{i_a}) \\ &= (b-a)(t_{i_a} - t_{i_b}) \\ &> 0 \end{aligned}$$

Bedienreihenfolge	1	...	a	...	b	...	n
bedienter Kunde	i_1	...	i_a	...	i_b	...	i_n
Bedienzeit	t_{i_1}	...	t_{i_a}	...	t_{i_b}	...	t_{i_n}
nach Austausch und von t_{i_a} und t_{i_b}	↓						↓
Bedienzeit	t_{i_1}	...	t_{i_b}	...	t_{i_a}	...	t_{i_n}
bedienter Kunde	i_1	...	i_b	...	i_a	...	i_n