ETH zürich

Übung 9

Datenstrukturen und Algorithmen, D-MATH, ETH Zurich

Programm von heute

Feedback letzte Übung

Repetition Theorie
- Aktivitätenauswahl
- Huffman Codierung
- Rekursive Problemlösestrategien

In-Class-Exercise (praktisch)

Hinweise zu den Aufgaben

1. Feedback letzte Übung

2. Repetition Theorie

Gierige Auswahl

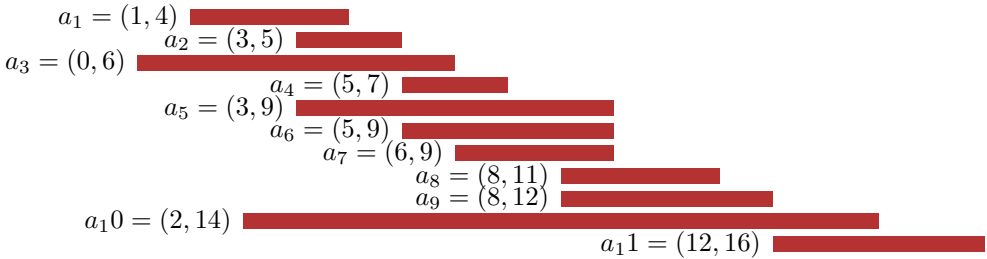
Ein rekursiv lösbares Optimierungsproblem kann mit einem **gierigen (greedy) Algorithmus** gelöst werden, wenn es die folgende Eigenschaften hat:

- Das Problem hat **optimale Substruktur**: die Lösung eines Problems ergibt sich durch Kombination optimaler Teillösungen.
- Es gilt die **greedy choice property**: Die Lösung eines Problems kann konstruiert werden, indem ein lokales Kriterium herangezogen wird, welches nicht von der Lösung der Teilprobleme abhängt.

Beispiele: Gebrochenes Rucksackproblem, Huffman-Coding (s.u.)
Gegenbeispiele: Rucksackproblem. Optimaler binärer Suchbaum.

Aktivitäten Auswahl

Koordination von Aktivitäten, die gemeinsame Resource exklusiv nutzen. Aktivitäten $S = \{a_1, a_2, \ldots, a_n\}$ mit Start und Endzeiten $0 \leq s_i \leq f_i < \infty$, aufsteigend sortiert nach Endzeiten.



Aktivitäten-Auswahl-Problem: Finde maximale Teilmenge (maximal in Anzahl Elementen) kompatibler (nichtüberlappender) Aktivitäten.

Dynamic Programming Ansatz?

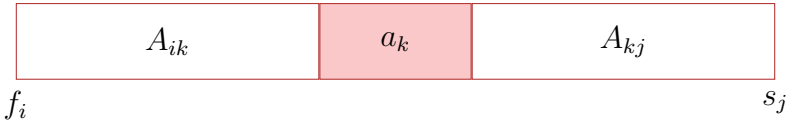
Sei $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.

Dynamic Programming Ansatz?

Sei $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.
Sei A_{ij} eine maximale Teilmenge kompatibler Aktivitäten aus S_{ij}.

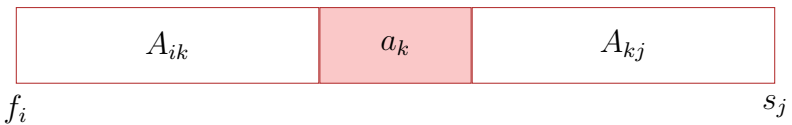
Dynamic Programming Ansatz?

Sei $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.
Sei A_{ij} eine maximale Teilmenge kompatibler Aktivitäten aus S_{ij}.
Sei $a_k \in A_{ij}$ und $A_{ik} = S_{ik} \cap A_{ij}$, $A_{ki} = S_{kj} \cap A_{ij}$, also $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$.



Dynamic Programming Ansatz?

Sei $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.
Sei A_{ij} eine maximale Teilmenge kompatibler Aktivitäten aus S_{ij}.
Sei $a_k \in A_{ij}$ und $A_{ik} = S_{ik} \cap A_{ij}$, $A_{ki} = S_{kj} \cap A_{ij}$, also $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$.



Klar: A_{ik} und A_{kj} müssen maximal sein, sonst wäre $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$ nicht maximal.

Dynamic Programming Ansatz?

Sei $c_{ij} = |A_{ij}|$.
Dann gilt folgende Rekursion

$$c_{ij} = \begin{cases} 0 & \text{falls } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c_{ik} + c_{kj} + 1\} & \text{falls } S_{ij} \neq \emptyset. \end{cases}$$

$\Rightarrow$ Dynamische Programmierung.

Dynamic Programming Ansatz?

Sei $c_{ij} = |A_{ij}|$.
Dann gilt folgende Rekursion

$$c_{ij} = \begin{cases} 0 & \text{falls } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c_{ik} + c_{kj} + 1\} & \text{falls } S_{ij} \neq \emptyset. \end{cases}$$

$\Rightarrow$ Dynamische Programmierung.

Aber es geht noch einfacher.

Greedy

Intuition: Wähle zuerst die Aktivität, die die früheste Endzeit hat (a_1). Das lässt maximal viel Platz für weitere Aktivitäten.

Greedy

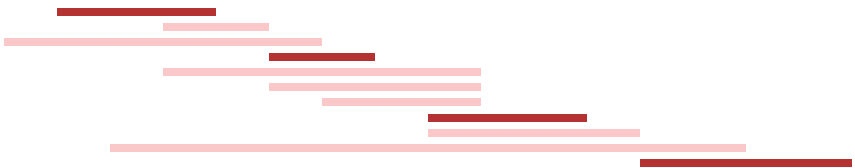
Intuition: Wähle zuerst die Aktivität, die die früheste Endzeit hat (a_1). Das lässt maximal viel Platz für weitere Aktivitäten.

Verbleibendes Teilproblem: Aktivitäten, die starten nachdem a_1 endet. (Es gibt keine Aktivitäten, die vor dem Start von a_1 enden.)

Greedy

Intuition: Wähle zuerst die Aktivität, die die früheste Endzeit hat (a_1). Das lässt maximal viel Platz für weitere Aktivitäten.

Verbleibendes Teilproblem: Aktivitäten, die starten nachdem a_1 endet. (Es gibt keine Aktivitäten, die vor dem Start von a_1 enden.)



Greedy

Theorem 1

Gegeben: Teilproblem S_k, und eine Aktivität a_m aus S_k mit frühester Endzeit. Dann ist a_m in einer maximalen Teilmenge von kompatiblen Aktivitäten aus S_k enthalten.

Sei A_k maximal grosse Teilmenge mit kompatiblen Aktivitäten aus S_k, und a_j eine Aktivität aus A_k mit frühester Endzeit. Wenn $a_j = a_m \Rightarrow$ fertig. Wenn $a_j \neq a_m$, dann betrachte $A'_k = A_k - \{a_j\} \cup \{a_m\}$. Dann besteht A'_k aus kompatiblen Aktivitäten und ist auch maximal, denn $|A'_k| = |A_k|$.

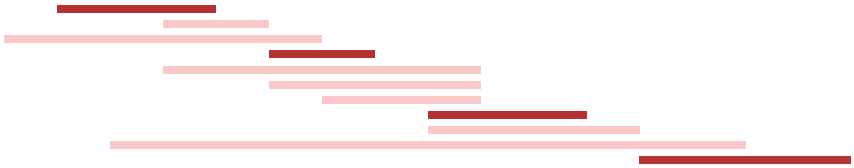
■

Algorithmus RecursiveActivitySelect(s, f, k, n)

Input: Folge von Start und Endzeiten (s_i, f_i), $1 \leq i \leq n$, $s_i < f_i$, $f_i \leq f_{i+1}$ für alle i. $1 \leq k \leq n$

Output: Maximale Menge kompatibler Aktivitäten.

$m \leftarrow k + 1$
while $m \leq n$ and $s_m \leq f_k$ **do**
    $m \leftarrow m + 1$
if $m \leq n$ **then**
    return $\{a_m\} \cup$ RecursiveActivitySelect(s, f, m, n)
else
    return $\emptyset$



Algorithmus IterativeActivitySelect(s, f, n)

Input: Folge von Start und Endzeiten (s_i, f_i), $1 \leq i \leq n$, $s_i < f_i$, $f_i \leq f_{i+1}$ für alle i.

Output: Maximale Menge kompatibler Aktivitäten.

$A \leftarrow \{a_1\}$
$k \leftarrow 1$
for $m \leftarrow 2$ **to** n **do**
    if $s_m \geq f_k$ **then**
        $A \leftarrow A \cup \{a_m\}$
        $k \leftarrow m$
return A

Laufzeit beider Algorithmen:

Algorithmus IterativeActivitySelect(s, f, n)

Input: Folge von Start und Endzeiten (s_i, f_i), $1 \leq i \leq n$, $s_i < f_i$, $f_i \leq f_{i+1}$ für alle i.

Output: Maximale Menge kompatibler Aktivitäten.

$A \leftarrow \{a_1\}$
$k \leftarrow 1$
for $m \leftarrow 2$ **to** n **do**
    if $s_m \geq f_k$ **then**
        $A \leftarrow A \cup \{a_m\}$
        $k \leftarrow m$
return A

Laufzeit beider Algorithmen: $\Theta(n)$

Huffmans Idee

Baum Konstruktion von unten nach oben

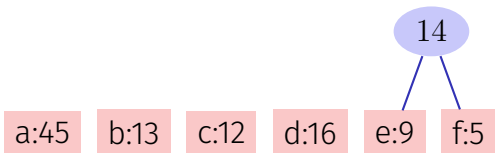
- Starte mit der Menge C der Codewörter

a:45 b:13 c:12 d:16 e:9 f:5

Huffmans Idee

Baum Konstruktion von unten nach oben

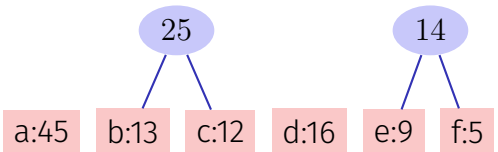
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffmans Idee

Baum Konstruktion von unten nach oben

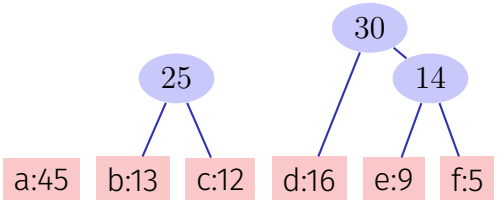
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffmans Idee

Baum Konstruktion von unten nach oben

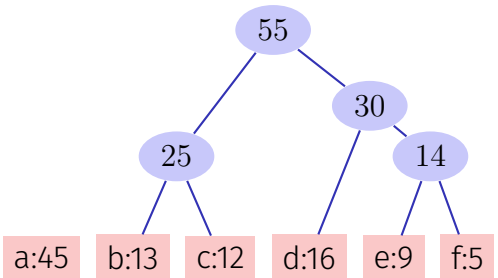
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffmans Idee

Baum Konstruktion von unten nach oben

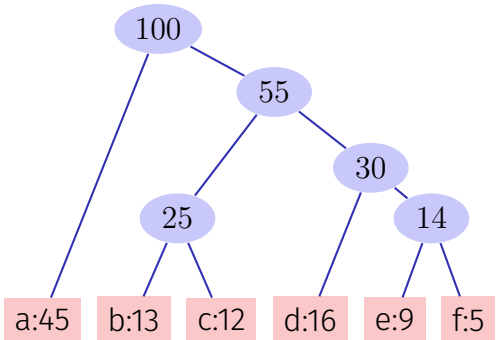
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffmans Idee

Baum Konstruktion von unten nach oben

- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Algorithmus Huffman(C)

Input: Codewörter $c \in C$
Output: Wurzel eines optimalen Codebaums

$n \leftarrow |C|$
$Q \leftarrow C$
for $i = 1$ **to** $n - 1$ **do**
    Alloziere neuen Knoten z
    z.left $\leftarrow$ ExtractMin(Q) // Extrahiere Wort mit minimaler Häufigkeit.
    z.right $\leftarrow$ ExtractMin(Q)
    z.freq $\leftarrow$ z.left.freq + z.right.freq
    Insert(Q, z)
return ExtractMin(Q)

Rekursive Problemlösestrategien

Brute Force Enumeration	Backtracking	Divide and Conquer	Dynamic Programming	Greedy

Rekursive Problemlösestrategien

Brute Force Enumeration	**Backtracking**	**Divide and Conquer**	**Dynamic Programming**	**Greedy**
Rekursive Aufzählbarkeit	Prüfbare Randbedingung, Partielle Validierung	Optimale Substruktur	Optimale Substruktur, Überlappende Teilprobleme	Optimale Substruktur, Gierige Auswahl Eigenschaft

Rekursive Problemlösestrategien

Brute Force Enumeration	**Backtracking**	**Divide and Conquer**	**Dynamic Programming**	**Greedy**
Rekursive Aufzählbarkeit	Prüfbare Randbedingung, Partielle Validierung	Optimale Substruktur	Optimale Substruktur, Überlappende Teilprobleme	Optimale Substruktur, Gierige Auswahl Eigenschaft
DFS, BFS, Alle Permutationen, Baumtraversieren	n Damen, Sudoku, m-Färbung, SAT-Solving, naiver TSP	Binäre Suche, Mergesort, Quicksort, Türme von Hanoi, FFT	Bellman Ford, Warshall, Rod-Cutting, LAT, Editierdistanz, Knapsack Problem DP	Dijkstra, Kruskal, Huffmann Coding

3. In-Class-Exercise (praktisch)

Vervollständigen Sie die DP Implementation zur Berechnung des optimalen Suchbaumes $\longrightarrow$ Code-Expert

4. Hinweise zu den Aufgaben

Huffman Coding

Huffman Code:

Benutze `std::map` (`#include <map>`)

```
std::map<std::string,int> observations;
// simple access to elements
++observations["cat"];
++observations["mouse"];
++observations["mouse"];

// a map is a collection of std::pair
// show all entries
for (auto x:observations){
  std::cout << "observations of " << x.first << ":" << x.second << std::end
}
```

Huffman Code:

Benutze `std::priority_queue` **(**`#include <queue>`**)**

```
struct MyClass {
    int x;
    MyClass(int X): x{X} {};
};

struct compare{
    bool operator() (const MyClass& a, const MyClass& b){
        return a.x < b.x;
    }
};
//...
std::priority_queue<MyClass, std::vector<MyClass>, compare> q;
q.push(MyClass(10));
```

Huffman Code:

Benutze Smart Pointers `std::shared_ptr` (`#include <memory>`)

```
struct List {
    int value;
    std::shared_ptr<List> next;
    List(std::shared_ptr<List> n, int v): value{v}, next{n} {};
};
...
// automatic memory management, we do not need to care
std::shared_ptr<List> l = std::make_shared<List>(nullptr, 10);
l = std::make_shared<List>(l, 20);
while (l != nullptr){ // output: 20 10
  std::cout << l->value << std::endl;
  l = l->next;
}
```

Huffman Node

```
using SharedNode=std::shared_ptr<Node>;
struct Node{
    char value;
    int frequency;
    SharedNode left;
    SharedNode right;

    // constructor for leafs
    Node(char v, int f): value{v}, frequency{f},
      left{nullptr}, right{nullptr} {}
    // constructor for inner nodes
    Node(SharedNode l, SharedNode r): value{0},
      frequency{l->frequency + r->frequency}, left{l}, right{r} {};
};
```