Datenstrukturen und Algorithmen

Übung 8

FS 2018

Programm von heute

1. Feedback letzte Übung
2. Wiederholung Theorie
3. Programmieraufgabe

Stückweise Konstante Approximation

$$H_{\gamma,y} : \mathcal{P} \mapsto \gamma|\mathcal{P}| + \sum_{I \in \mathcal{P}} \sum_{i \in I} (y_i - \mu_I)^2$$

Stückweise Konstante Approximation

$$H_{\gamma,y} : \mathcal{P} \mapsto \gamma|\mathcal{P}| + \sum_{I \in \mathcal{P}} \sum_{i \in I} (y_i - \mu_I)^2$$

- Wie in Übung 1 effizientes berechnen von Durchschnitten: $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i$

Stückweise Konstante Approximation

$$H_{\gamma,y} : \mathcal{P} \mapsto \gamma|\mathcal{P}| + \sum_{I \in \mathcal{P}} \sum_{i \in I} (y_i - \mu_I)^2$$

- Wie in Übung 1 effizientes berechnen von Durchschnitten: $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i \Rightarrow$ prefixsum ✓

Stückweise Konstante Approximation

$$H_{\gamma,y} : \mathcal{P} \mapsto \gamma|\mathcal{P}| + \sum_{I \in \mathcal{P}} \sum_{i \in I} (y_i - \mu_I)^2$$

- Wie in Übung 1 effizientes berechnen von Durchschnitten: $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i \Rightarrow$ prefixsum ✓
- Effizientes Berechnen von $e_{[l,r)} = \sum_{i=l}^{r-1} (y_i - \mu_{[l,r)})^2$

Stückweise Konstante Approximation

$$H_{\gamma,y} : \mathcal{P} \mapsto \gamma|\mathcal{P}| + \sum_{I \in \mathcal{P}} \sum_{i \in I} (y_i - \mu_I)^2$$

- Wie in Übung 1 effizientes berechnen von Durchschnitten:
  $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i \Rightarrow$ prefixsum ✓
- Effizientes Berechnen von $e_{[l,r)} = \sum_{i=l}^{r-1} (y_i - \mu_{[l,r)})^2$
  $\Rightarrow e_{[l,r)} = \sum_{i=l}^{r-1} y_i^2 - \frac{1}{r-l} \left(\sum_{i=l}^{r-1} y_i \right)^2$ ✓

Stückweise Konstante Approximation

$$H_{\gamma,y} : \mathcal{P} \mapsto \gamma|\mathcal{P}| + \sum_{I \in \mathcal{P}} \sum_{i \in I} (y_i - \mu_I)^2$$

- Wie in Übung 1 effizientes berechnen von Durchschnitten: $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i \Rightarrow$ prefixsum ✓
- Effizientes Berechnen von $e_{[l,r)} = \sum_{i=l}^{r-1} (y_i - \mu_{[l,r)})^2$
  $\Rightarrow e_{[l,r)} = \sum_{i=l}^{r-1} y_i^2 - \frac{1}{r-l} \left(\sum_{i=l}^{r-1} y_i \right)^2$ ✓
- **Dynamische Programmierung**: Definition der Tabelle, Berechnung eines Eintrags, Berechnungsreihenfolge, Auslesen der Lösung

Stückweise Konstante Approximation

$$H_{\gamma,y} : \mathcal{P} \mapsto \gamma|\mathcal{P}| + \sum_{I \in \mathcal{P}} \sum_{i \in I} (y_i - \mu_I)^2$$

- Wie in Übung 1 effizientes berechnen von Durchschnitten: $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i \Rightarrow$ prefixsum ✓
- Effizientes Berechnen von $e_{[l,r)} = \sum_{i=l}^{r-1} (y_i - \mu_{[l,r)})^2$
  $\Rightarrow e_{[l,r)} = \sum_{i=l}^{r-1} y_i^2 - \frac{1}{r-l} \left(\sum_{i=l}^{r-1} y_i \right)^2$ ✓
- **Dynamische Programmierung**: Definition der Tabelle, Berechnung eines Eintrags, Berechnungsreihenfolge, Auslesen der Lösung $\Rightarrow$ **?**

Dynamische Programmierung

- **Definition der DP-Tabelle**: zwei Tabellen: B und V mit jeweils $n+1 \times 1$ Einträgen, $B[k]$ beinhaltet Zeiger zum Besten vorherigen Intervall, $V[k]$ beinhaltet entsprechendes Minimum von H_γ.
- **Berechnung eines Eintrags**: um neuen Eintrag in $B[k+1]$ zu berechnen berechne alle H_γ für alle Partitionen von 0 bis $k+1$.
- **Berechnungsreihenfolge**: von links nach rechts
- **Auslesen der Lösung**: konstruiere Intervalle mit $B[n]$ von rechts nach links, Minimum ist gegeben durch $V[n]$

Summen

Gegeben Datenvektor y der Länge $n \in \mathbb{N}$: $(y_i)_{i=1\ldots n} \in \mathbb{R}^n$

Summe $m_n := \sum_{i=1}^{n} y_i \Rightarrow \mu_n = m_n/n$

Summe Quadrate $s_n := \sum_{i=1}^{n} y_i^2$

$$
\begin{aligned}
e_n &:= \sum_{i=1}^{n} (y_i - \mu_n)^2 = \sum_{i=1}^{n} y_i^2 - 2\mu_n y_i + \mu_n^2 \\
&= s_n - 2\mu_n \left(\sum_{i=1}^{n} y_i \right) + n \cdot \mu_n^2 = s_n - 2\mu_n \cdot n\mu_n + n \cdot \mu_n^2 \\
&= s_n - n \cdot \mu_n^2 = s_n - m_n^2/n
\end{aligned}
$$

Statistik

```
// post: return mean of data[from,to)
double mean(unsigned int from, unsigned int to) const{
    assert(from < to && to <= n);
    return getsum(vsum,from,to) / (to-from);
}

// post: return err of constant approximation in interval [from,to)
double err(unsigned int from, unsigned int to) const{
    assert(from < to && to <= n);
    double m = getsum(vsum,from,to);
    return getsum(vssq,from,to) - m*m / (to-from);
}
```

DP – Setup and Base Case

```
double MinimizeH(double gamma,const Statistics& s,
                 std::vector<double>& result){
    int n = s.size();
    // B[k] contains the pointer to the end of the best previous interval
    // i.e. best possible approximation is given by
    // best possible approximation of [0,B[k]), [B[k],k)
    std::vector<int> B(n+1);
    // V(k) contains the corresponsing attainable minimum of H_gamma
    std::vector<double> V(n+1);
    // base case: empty interval
    B[0] = 0;
    V[0] = 0;
```

DP – Construct Table

```
// now consider all combinations of Partition([0, left)) + [left, right)
for (int right=1; right <= n; ++right){
    // interval [0, right)
    int best = 0;
    double min = gamma + s.err(0,right);
    // intervals [left, right), left > 0
    for (int left = 1; left < right; ++left){
        double h = V[left] + gamma + s.err(left,right);
        if (h < min){
            min = h; best = left;
        }
    }
    B[right] = best;
    V[right] = min;
}
```

DP – Reconstruct Solution

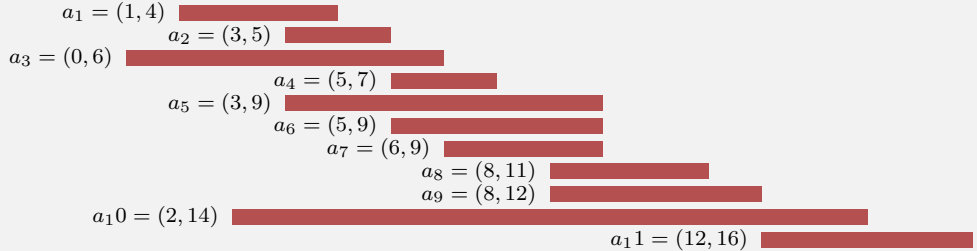
```
    // reconstruct solution
    unsigned int right=n;
    while (right != 0){
        unsigned int left = B[right];
        fill (result,s.mean(left,right), left, right);
        right = left;
    }
    return V[n];

}
```

2. Wiederholung Theorie

Ein anderes Beispiel einer erfolgreichen Greedy Strategie, Graphen

Aktivitäten Auswahl

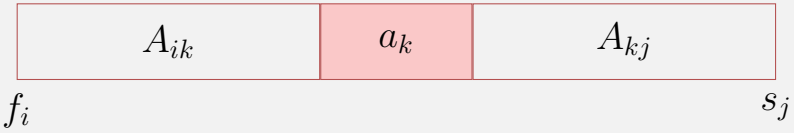
Koordination von Aktivitäten, die gemeinsame Resource exklusiv nutzen. Aktivitäten $S = \{a_1, a_2, \ldots, a_n\}$ mit Start und Endzeiten $0 \leq s_i \leq f_i < \infty$, aufsteigend sortiert nach Endzeiten.



Aktivitäten-Auswahl-Problem: Finde maximale Teilmenge kompatibler (nichtüberlappender) Aktivitäten.

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 ausserdem $a_k \in A_{ij}$ 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 $c_{ij} = c_{ik} + c_{kj} + 1$, also

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

Könnten nun dynamische Programmierung versuchen.

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

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

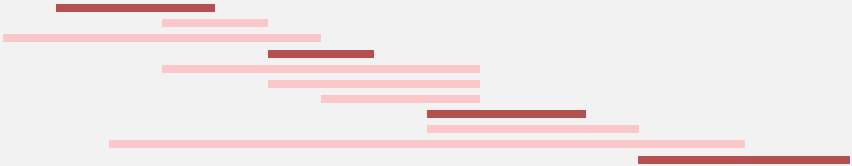
Sei A_k maximal grosse Teilmenge mit kompatiblen Aktivitäten aus S_k und a_j eine Aktivität aus A_k mit frühester Endzeit. Wenn $a_j = a_m$ $\Rightarrow$ fertig. Wenn $a_j \neq a_m$. Dann betrachte $A'_k = A_k - \{a_j\} \cup \{a_m\}$. Dann besteht A'_k aus kompatiblen Aktivitäten und ist auch maximal, 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. $1 \leq k \leq n$

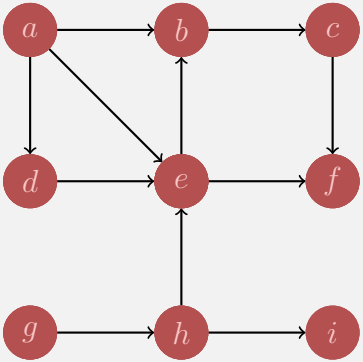
Output : Maximale Menge kompatibler Aktivitäten.

$m \leftarrow k + 1$
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 $\emptyset$



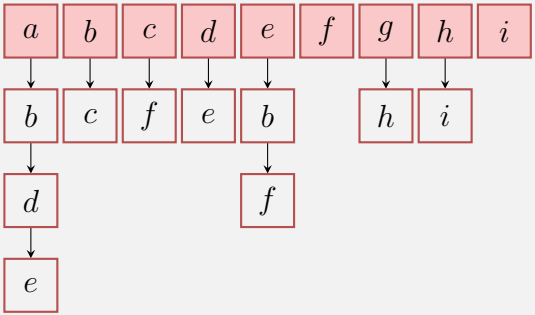
Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



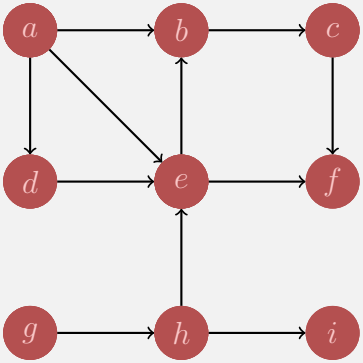
Reihenfolge $a, b, c, f, d, e, g, h, i$

Adjazenzliste

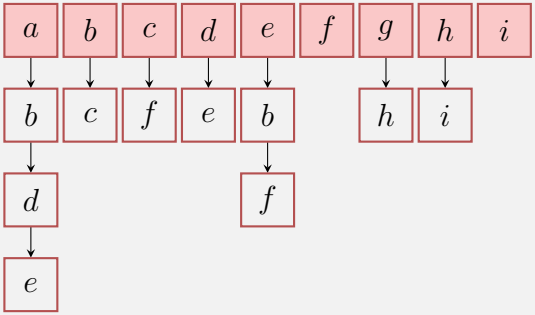


Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.



Adjazenzliste



Reihenfolge $a, b, d, e, c, f, g, h, i$

Iteratives DFS-Visit(G, v)

Input : Graph $G = (V, E)$

Stack $S \leftarrow \emptyset$; push(S, v)
while $S \neq \emptyset$ **do**
    $w \leftarrow$ pop(S)
    if $\neg(w$ besucht$)$ **then**
        Markiere w besucht
        foreach $(w, c) \in E$ **do** // (ggfs umgekehrt einfügen)
            if $\neg(c$ besucht$)$ **then**
                push(S, c)

Stapelgrösse bis zu $|E|$, für jeden Knoten maximal Extraaufwand $\Theta(\deg^+(w) + 1)$. Gesamt: $\mathcal{O}(|V| + |E|)$

Mit Aufruf aus obigem Rahmenprogramm: $\Theta(|V| + |E|)$

Iteratives BFS-Visit(G, v)

Input : Graph $G = (V, E)$

```
Queue Q ← ∅
Markiere v aktiv
enqueue(Q, v)
while Q ≠ ∅ do
    w ← dequeue(Q)
    Markiere w besucht
    foreach (w, c) ∈ E do
        if ¬(c besucht ∨ c aktiv) then
            Markiere c aktiv
            enqueue(Q, c)
```

- Algorithmus kommt mit $\mathcal{O}(|V|)$ Extraplatz aus. (Warum funktioniert dieser simple Trick nicht beim DFS?)
- Gesamtlaufzeit mit Rahmenprogramm: $\Theta(|V| + |E|)$.

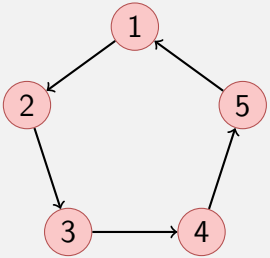
Topologische Sortierung

Topologische Sortierung eines azyklischen gerichteten Graphen $G = (V, E)$: Bijektive Abbildung

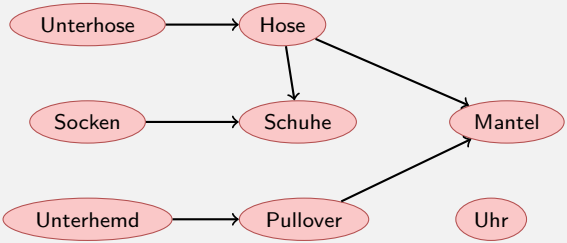
$$\mathrm{ord} : V \to \{1, \ldots, |V|\} \quad | \quad \mathrm{ord}(v) < \mathrm{ord}(w) \; \forall \; (v, w) \in E.$$

Können Wert i auch identifizieren mit v_i. Topologische Sortierung $\widehat{=}$ $\langle v_1, \ldots, v_{|V|} \rangle$.

(Gegen-)Beispiele



Zyklischer Graph: kann nicht topologisch sortiert werden.



Eine mögliche Topologische Sortierung des Graphen:
Unterhemd,Pullover,Unterhose,Uhr,Hose,Mantel,Socken,Schuhe

Beobachtung

Theorem

Ein gerichteter Graph $G = (V, E)$ besitzt genau dann eine topologische Sortierung, wenn er kreisfrei ist

Algorithmus Topological-Sort(G)

Input : Graph $G = (V, E)$.
Output : Topologische Sortierung ord

Stack $S \leftarrow \emptyset$
foreach $v \in V$ **do** $A[v] \leftarrow 0$
foreach $(v, w) \in E$ **do** $A[w] \leftarrow A[w] + 1$ // Eingangsgrade berechnen
foreach $v \in V$ with $A[v] = 0$ **do** push(S, v) // Merke Nodes mit Eingangsgrad 0
$i \leftarrow 1$
while $S \neq \emptyset$ **do**
    $v \leftarrow$ pop(S); $\mathrm{ord}[v] \leftarrow i$; $i \leftarrow i + 1$ // Wähle Knoten mit Eingangsgrad 0
    foreach $(v, w) \in E$ **do** // Verringere Eingangsgrad der Nachfolger
        $A[w] \leftarrow A[w] - 1$
        if $A[w] = 0$ **then** push(S, w)

if $i = |V| + 1$ **then return** ord **else return** "Cycle Detected"

3. Programmieraufgabe

Huffman Coding

Genial Einfacher Algorithmus

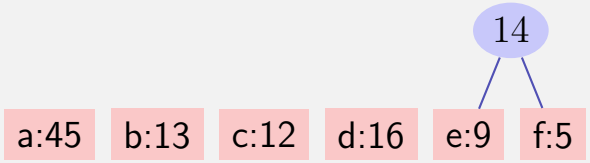
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.

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

Huffman Coding

Genial Einfacher Algorithmus

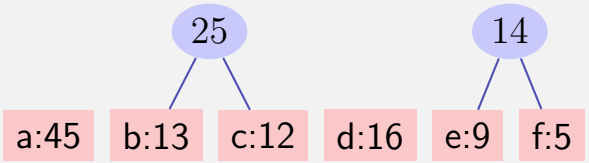
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffman Coding

Genial Einfacher Algorithmus

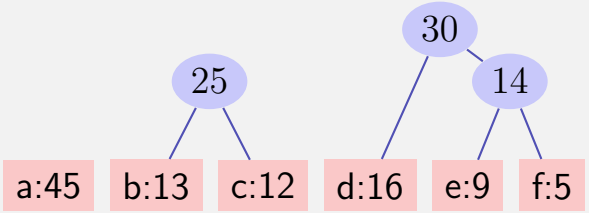
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffman Coding

Genial Einfacher Algorithmus

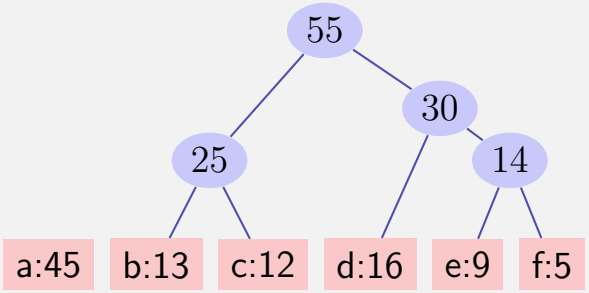
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffman Coding

Genial Einfacher Algorithmus

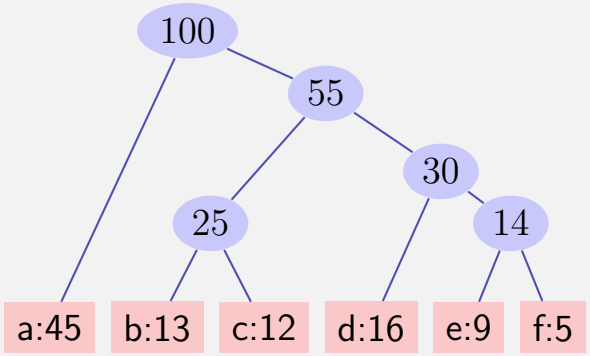
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffman Coding

Genial Einfacher Algorithmus

- 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)

Tipps zur Implementation

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
}
```

Tipps zur Implementation

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));
```

Tipps zur Implementation

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;
}
```

Fragen?