Datenstrukturen und Algorithmen

Exercise 8

FS 2018

Program of today

1. Feedback of last exercise
2. Repetition theory
3. Programming Task

Piecewise Constant Approximation

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

Piecewise Constant Approximation

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

- As in exercise 1 efficient computation of mean: $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i$

Piecewise Constant Approximation

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

- As in exercise 1 efficient computation of mean: $\mu_I = \frac{1}{|I|} \sum_{i\in I} y_i$
  $\Rightarrow$ prefixsum ✓

Piecewise Constant Approximation

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

- As in exercise 1 efficient computation of mean: $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i$
  $\Rightarrow$ prefixsum ✓
- Efficient computing $e_{[l,r)} = \sum_{i=l}^{r-1} (y_i - \mu_{[l,r)})^2$

Piecewise Constant Approximation

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

- As in exercise 1 efficient computation of mean: $\mu_I = \frac{1}{|I|}\sum_{i\in I} y_i$
  $\Rightarrow$ prefixsum ✓
- Efficient computing $e_{[l,r)} = \sum_{i=l}^{r-1}(y_i - \mu_{[l,r)})^2$
  $\Rightarrow e_{[l,r)} = \sum_{i=l}^{r-1} y_i^2 - \frac{1}{r-l}\left(\sum_{i=l}^{r-1} y_i\right)^2$ ✓

Piecewise Constant Approximation

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

- As in exercise 1 efficient computation of mean: $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i$
  $\Rightarrow$ prefixsum ✓
- Efficient computing $e_{[l,r)} = \sum_{i=l}^{r-1} (y_i - \mu_{[l,r)})^2$
  $\Rightarrow e_{[l,r)} = \sum_{i=l}^{r-1} y_i^2 - \frac{1}{r-l} \left(\sum_{i=l}^{r-1} y_i\right)^2$ ✓
- **Dynamic programming**: definition of the table, computation of an entry, calculation order, extracting solution

Piecewise Constant Approximation

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

- As in exercise 1 efficient computation of mean: $\mu_I = \frac{1}{|I|} \sum_{i \in I} y_i$
  $\Rightarrow$ prefixsum ✓
- Efficient computing $e_{[l,r)} = \sum_{i=l}^{r-1} (y_i - \mu_{[l,r)})^2$
  $\Rightarrow e_{[l,r)} = \sum_{i=l}^{r-1} y_i^2 - \frac{1}{r-l} \left(\sum_{i=l}^{r-1} y_i \right)^2$ ✓
- **Dynamic programming**: definition of the table, computation of an entry, calculation order, extracting solution $\Rightarrow$ **?**

Dynamic programming

- **Definition of the DP table**: two tables: B and V with each $n + 1 \times 1$ entries, $B[k]$ contains the pointer to the end of the best previous interval, $V[k]$ contains the corresponding attainable minimum of H_γ.
- **Computation of an entry**: for computing new entry in $B[k+1]$ compute H for all partitions from 0 to $k+1$.
- **Calculation order**: from left to right
- **Extracting the solution**: construct intervals with $B[n]$ going from right to left, Minimum is given by $V[n]$

Sums

Given a data vector of length $n \in \mathbb{N}$: $(y_i)_{i=1\ldots n} \in \mathbb{R}^n$

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

Sum of Squares $s_n := \sum_{i=1}^{n} y_i^2$

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

Statistics

```
// 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

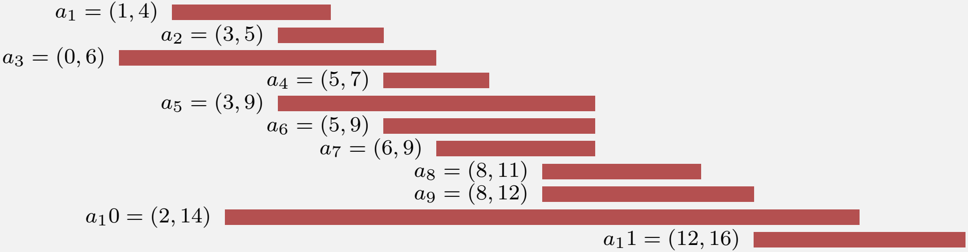
```
    // reconstruct solution
    unsigned int right=n;
    while (right != 0){
        unsigned int left = B[right];
        fill(result, s.mean(left, right), left, right);
        right = left;
    }
    return V[n];

}
```

2. Repetition theory

A Different Example of a Successful Greedy Strategy, Graphs

Activity Selection

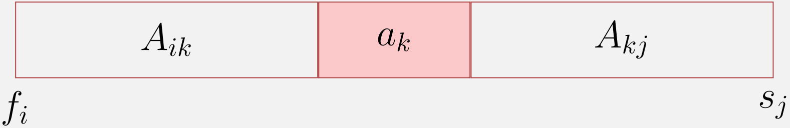
Coordination of activities that use a common resource exclusively.
Activities $S = \{a_1, a_2, \dots, a_n\}$ with start- and finishing times $0 \le s_i \le f_i < \infty$, increasingly sorted by finishing times.



Activity Selection Problem: Find a maximal subset of compatible (non-intersecting) activities.

Dynamic Programming Approach?

Let $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$. Let A_{ij} be a maximal subset of compatible activities from S_{ij}. Moreover, let $a_k \in A_{ij}$ and $A_{ik} = S_{ik} \cap A_{ij}$, $A_{ki} = S_{kj} \cap A_{ij}$, thus $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$.



Straightforward: A_{ik} and A_{kj} must be maximal, otherwise $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$ would not be maximal.

Dynamic Programming Approach?

Let $c_{ij} = |A_{ij}|$. Then the following recursion holds $c_{ij} = c_{ik} + c_{kj} + 1$, therefore

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

Could now try dynamic programming.

Greedy

Intuition: choose the activity that provides the earliest end time (a_1). That leaves maximal space for other activities.

Remaining problem: activities that start after a_1 ends. (There are no activites that can end before a_1 starts.)

Greedy

Theorem

Given: Subproblem S_k, a_m an activity from S_k with earliest end time. Then a_m is contained in a maximal subset of compatible activities from S_k.

Let A_k be a maximal subset with compatible activities from S_K and a_j be an activity from A_k with earliest end time. If $a_j = a_m \Rightarrow$ done. If $a_j \neq a_m$. Then consider $A'_k = A_k - \{a_j\} \cup \{a_m\}$. A'_k conists of compatible activities and is also maximal because $|A'_k| = |A_k|$.

■

Algorithm RecursiveActivitySelect(s, f, k, n)

Input : Sequence of start and end points (s_i, f_i), $1 \leq i \leq n$, $s_i < f_i$, $f_i \leq f_{i+1}$ for all i. $1 \leq k \leq n$

Output : Set of all compatible activitivies.

$m \leftarrow k + 1$

while $m \leq n$ and $s_m \leq f_k$ **do**

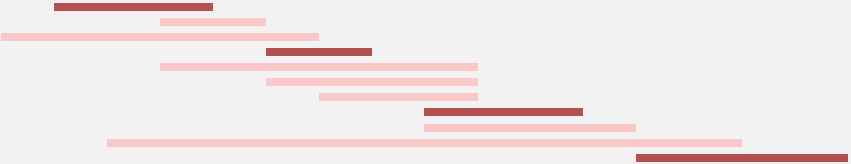
  $m \leftarrow m + 1$

if $m \leq n$ **then**

  return $\{a_m\} \cup \text{RecursiveActivitySelect}(s, f, m, n)$

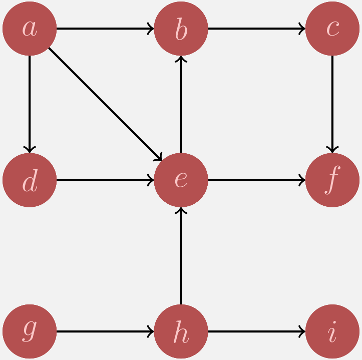
else

  return $\emptyset$



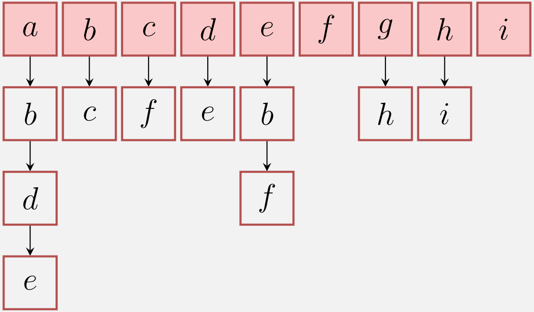
Graph Traversal: Depth First Search

Follow the path into its depth until nothing is left to visit.



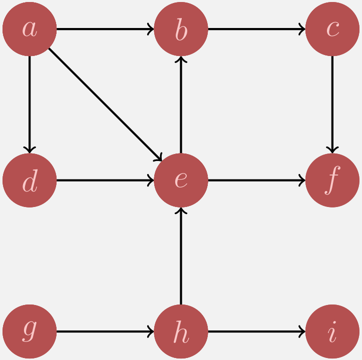
Order $a, b, c, f, d, e, g, h, i$

adjacency list



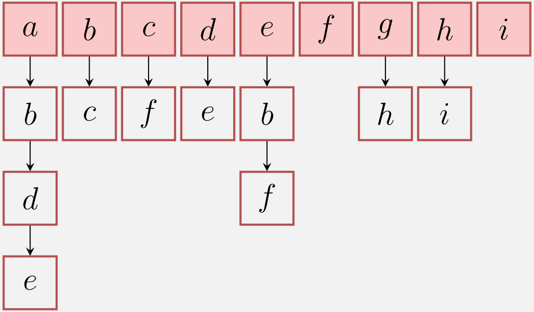
Graph Traversal: Breadth First Search

Follow the path in breadth and only then descend into depth.



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

adjacency list



Iterative 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 visited) **then**
        mark w visited
        foreach $(w, c) \in E$ **do** // (in reverse order, potentially)
            if $\neg$(c visited) **then**
                push(S, c)

Stack size up to $|E|$, for each node an extra of $\Theta(\deg^+(w) + 1)$ operations. Overall: $\mathcal{O}(|V| + |E|)$

Including all calls from the above main program: $\Theta(|V| + |E|)$

Iterative BFS-Visit(G, v)

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

Queue $Q \leftarrow \emptyset$
Mark v as active
enqueue(Q, v)
while $Q \neq \emptyset$ **do**
    $w \leftarrow$ dequeue(Q)
    mark w visited
    foreach $(w, c) \in E$ **do**
        if $\neg(c$ visited $\vee$ c active$)$ **then**
            Mark c as active
            enqueue(Q, c)

- Algorithm requires extra space of $\mathcal{O}(|V|)$. (Why does that simple approach not work with DFS?)
- Running time including main program: $\Theta(|V| + |E|)$.

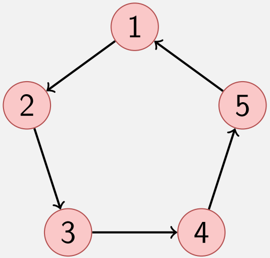
Topological Sorting

Topological Sorting of an acyclic directed graph $G = (V, E)$: Bijective mapping

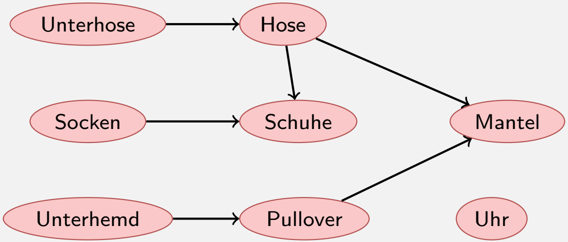
$$\mathrm{ord} : V \to \{1, \ldots, |V|\} \quad | \quad \mathrm{ord}(v) < \mathrm{ord}(w) \ \forall\ (v, w) \in E.$$

We can identify i with v_i. Topological sorting $\ \widehat{=}\ \langle v_1, \ldots, v_{|V|} \rangle$.

(Counter-)Examples



Cyclic graph: cannot be sorted topologically.



A possible toplogical sorting of the graph:
Unterhemd,Pullover,Unterhose,Uhr,Hose,Mantel,Socken,Schuhe

Observation

Theorem

A directed graph $G = (V, E)$ permits a topological sorting if and only if it is acyclic.

Algorithm Topological-Sort(G)

Input : graph $G = (V, E)$.
Output : Topological sorting ord

Stack $S \leftarrow \emptyset$
foreach $v \in V$ **do** $A[v] \leftarrow 0$
foreach $(v, w) \in E$ **do** $A[w] \leftarrow A[w] + 1$ // Compute in-degrees
foreach $v \in V$ with $A[v] = 0$ **do** push(S, v) // Memorize nodes with in-degree 0
$i \leftarrow 1$
while $S \neq \emptyset$ **do**
    $v \leftarrow$ pop(S); $\mathrm{ord}[v] \leftarrow i$; $i \leftarrow i + 1$ // Choose node with in-degree 0
    foreach $(v, w) \in E$ **do** // Decrease in-degree of successors
        $A[w] \leftarrow A[w] - 1$
        if $A[w] = 0$ **then** push(S, w)

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

3. Programming Task

Huffman Coding

Brilliantly Simple Algorithm

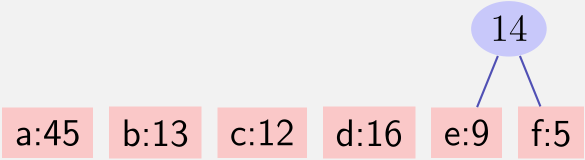
- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.

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

Huffman Coding

Brilliantly Simple Algorithm

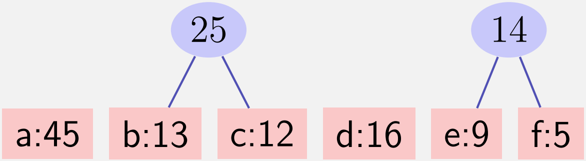
- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Huffman Coding

Brilliantly Simple Algorithm

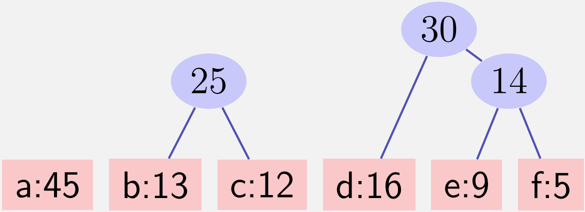
- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Huffman Coding

Brilliantly Simple Algorithm

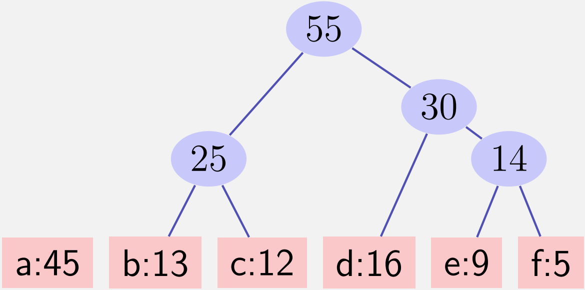
- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Huffman Coding

Brilliantly Simple Algorithm

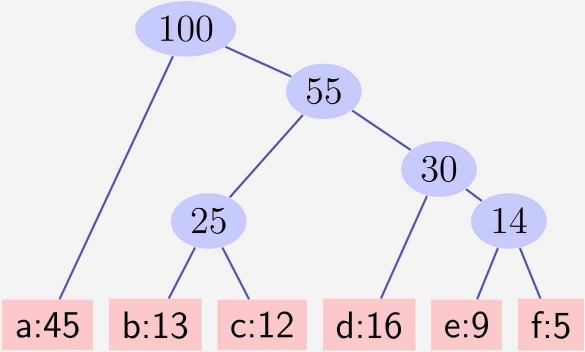
- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Huffman Coding

Brilliantly Simple Algorithm

- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Algorithm Huffman(C)

Input : code words $c \in C$
Output : Root of an optimal code tree

$n \leftarrow |C|$
$Q \leftarrow C$
for $i = 1$ **to** $n - 1$ **do**
  allocate a new node z
  z.left $\leftarrow$ ExtractMin(Q) // extract word with minimal frequency.
  z.right $\leftarrow$ ExtractMin(Q)
  z.freq $\leftarrow$ z.left.freq $+$ z.right.freq
  Insert(Q, z)
return ExtractMin(Q)

Hints for the Implementation

Use `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
}
```

Hints for the Implementation

Use `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));
```

Hints for the Implementation

Use 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;
}
```

Questions?