ETH zürich

Exercise Session 9

Data Structures and Algorithms, D-MATH, ETH Zurich

Program of today

Feedback of last exercise

Repetition Theory

- Activity Selection
- Huffman Coding
- Recursive Problem-Solving Strategies

In-Class-Exercise (practical)

Hints for current tasks

1. Feedback of last exercise

2. Repetition Theory

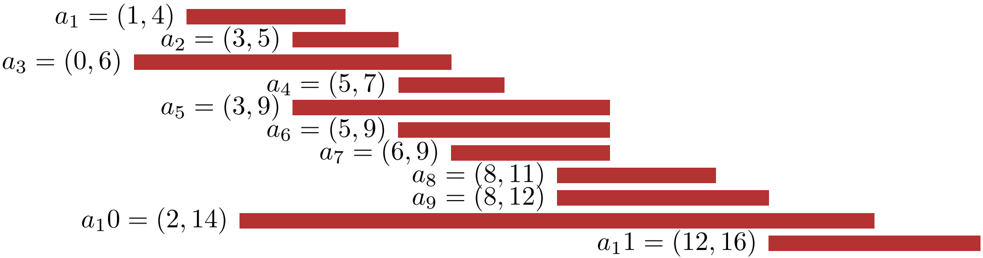
Greedy Choice

A problem with a recursive solution can be solved with a **greedy algorithm** if it has the following properties:

- The problem has **optimal substructure**: the solution of a problem can be constructed with a combination of solutions of sub-problems.
- The problem has the **greedy choice property**: The solution to a problem can be constructed, by using a local property that does not depend on the solution of the sub-problems.

Activity Selection

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



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

Dynamic Programming Approach?

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

Dynamic Programming Approach?

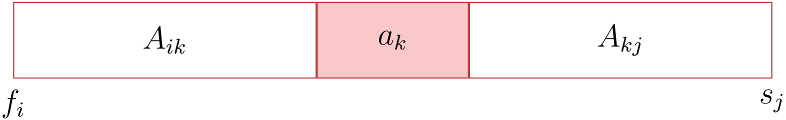
Let $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.
Let A_{ij} be a maximal subset of compatible activities from S_{ij}.

Dynamic Programming Approach?

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

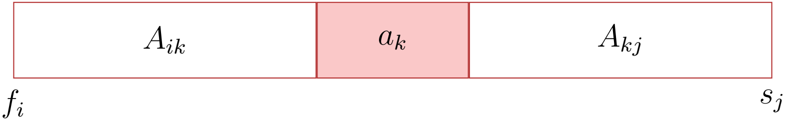
Let A_{ij} be a maximal subset of compatible activities from S_{ij}.

Let $a_k \in A_{ij}$ and $A_{ik} = S_{ik} \cap A_{ij}$, $A_{ki} = S_{kj} \cap A_{ij}$, thus $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$.



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}.
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

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

$\Rightarrow$ Dynamic programming.

Dynamic Programming Approach?

Let $c_{ij} = |A_{ij}|$.
Then the following recursion holds

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

$\Rightarrow$ Dynamic programming.

But there is a simpler alternative.

Greedy

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

Greedy

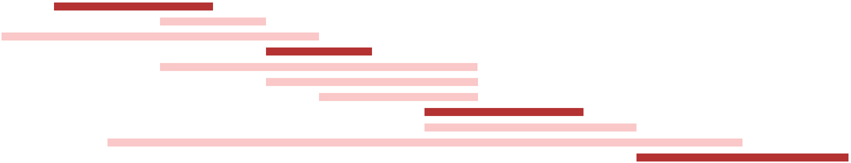
Intuition: choose the activity that provides the earliest end time (a_1). That leaves maximal space for other activities.

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

Greedy

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 1

Given: The set of subproblem S_k, and an activity a_m from S_k with the 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 the 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 consists of compatible activities and is also maximal because $|A'_k| = |A_k|$.

$\blacksquare$

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**

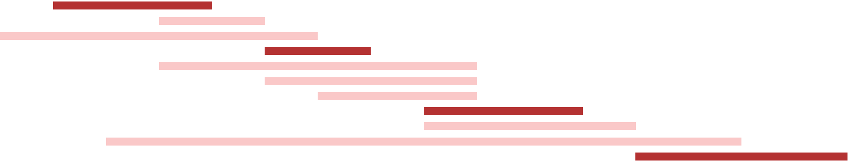
  $m \leftarrow m + 1$

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

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

else

  return $\emptyset$



Algorithm IterativeActivitySelect(s, f, 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.

Output: Maximal set of compatible activities.

$A \leftarrow \{a_1\}$
$k \leftarrow 1$
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

Runtime of both algorithms:

Algorithm IterativeActivitySelect(s, f, 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.

Output: Maximal set of compatible activities.

$A \leftarrow \{a_1\}$
$k \leftarrow 1$
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

Runtime of both algorithms: $\Theta(n)$

Huffman's Idea

Tree construction bottom up

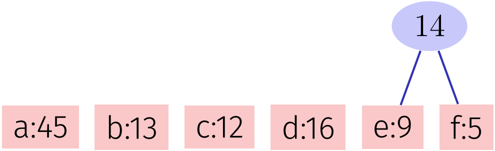
- Start with the set C of code words

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

Huffman's Idea

Tree construction bottom up

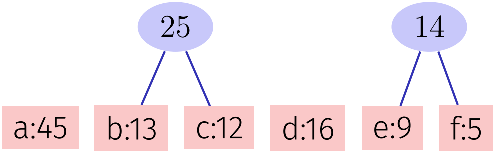
- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Huffman’s Idea

Tree construction bottom up

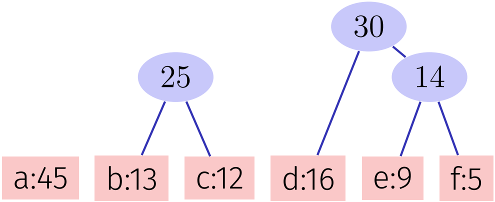
- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Huffman's Idea

Tree construction bottom up

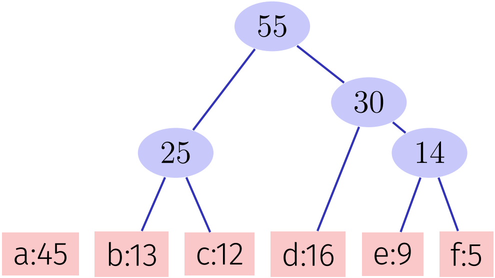
- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Huffman's Idea

Tree construction bottom up

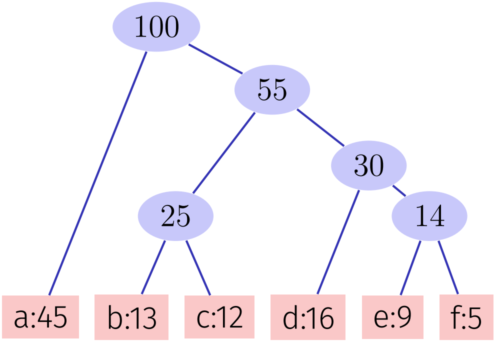
- Start with the set C of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



Huffman's Idea

Tree construction bottom up

- 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)

Recursive Problem-Solving Strategies

Brute Force Enumeration	Backtracking	Divide and Conquer	Dynamic Programming	Greedy

Recursive Problem-Solving Strategies

Brute Force Enumeration	Backtracking	Divide and Conquer	Dynamic Programming	Greedy
Recursive Enumerability	Constraint Satisfaction, Partial Validation	Optimal Substructure	Optimal Substructure, Overlapping Subproblems	Optimal Substructure, Greedy Choice Property

Recursive Problem-Solving Strategies

Brute Force Enumeration	**Backtracking**	**Divide and Conquer**	**Dynamic Programming**	**Greedy**
Recursive Enumerability	Constraint Satisfaction, Partial Validation	Optimal Substructure	Optimal Substructure, Overlapping Subproblems	Optimal Substructure, Greedy Choice Property
DFS, BFS, all Permutations, Tree Traversal	n-Queen, Sudoku, m-Coloring, SAT-Solving, naive TSP	Binary Search, Mergesort, Quicksort, Hanoi Towers, FFT	Bellman Ford, Warshall, Rod-Cutting, LAS, Editing Distance, Knapsack Problem DP	Dijkstra, Kruskal, Huffmann Coding

3. In-Class-Exercise (practical)

Complement the DP implementation to compute an optimal search tree. $\longrightarrow$ CodeExpert

4. Hints for current tasks

Huffman Coding

Huffman Code:

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 << std::enc
}
```

Huffman Code:

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));
```

Huffman Code:

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;
}
```

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} {};
};
```