

23. Greedy Algorithms

Fractional Knapsack Problem, Huffman Coding [Cormen et al, Kap. 16.1, 16.3]

The Fractional Knapsack Problem

set of $n \in \mathbb{N}$ items $\{1, \dots, n\}$ Each item i has value $v_i \in \mathbb{N}$ and weight $w_i \in \mathbb{N}$. The maximum weight is given as $W \in \mathbb{N}$. Input is denoted as $E = (v_i, w_i)_{i=1, \dots, n}$.

Wanted: Fractions $0 \leq q_i \leq 1$ ($1 \leq i \leq n$) that maximise the sum $\sum_{i=1}^n q_i \cdot v_i$ under $\sum_{i=1}^n q_i \cdot w_i \leq W$.

Greedy heuristics

Sort the items decreasingly by value per weight v_i/w_i .

Assumption $v_i/w_i \geq v_{i+1}/w_{i+1}$

Let $j = \max\{0 \leq k \leq n : \sum_{i=1}^k w_i \leq W\}$. Set

■ $q_i = 1$ for all $1 \leq i \leq j$.

■ $q_{j+1} = \frac{W - \sum_{i=1}^j w_i}{w_{j+1}}$.

■ $q_i = 0$ for all $i > j + 1$.

That is fast: $\Theta(n \log n)$ for sorting and $\Theta(n)$ for the computation of the q_i .

Correctness

Assumption: optimal solution (r_i) ($1 \leq i \leq n$).

The knapsack is full: $\sum_i r_i \cdot w_i = \sum_i q_i \cdot w_i = W$.

Consider k : smallest i with $r_i \neq q_i$ Definition of greedy: $q_k > r_k$. Let $x = q_k - r_k > 0$.

Construct a new solution (r'_i) : $r'_i = r_i \forall i < k$. $r'_k = q_k$. Remove weight $\sum_{i=k+1}^n \delta_i = x \cdot w_k$ from items $k+1$ to n . This works because $\sum_{i=k}^n r_i \cdot w_i = \sum_{i=k}^n q_i \cdot w_i$.

Correctness

$$\begin{aligned}\sum_{i=k}^n r'_i v_i &= r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^n (r_i w_i - \delta_i) \frac{v_i}{w_i} \\ &\geq r_k v_k + x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^n r_i w_i \frac{v_i}{w_i} - \delta_i \frac{v_k}{w_k} \\ &= r_k v_k + x w_k \frac{v_k}{w_k} - x w_k \frac{v_k}{w_k} + \sum_{i=k+1}^n r_i w_i \frac{v_i}{w_i} = \sum_{i=k}^n r_i v_i.\end{aligned}$$

Thus (r'_i) is also optimal. Iterative application of this idea generates the solution (q_i) .

Huffman-Codes

Goal: memory-efficient saving of a sequence of characters using a binary code with code words..

Example

File consisting of 100.000 characters from the alphabet $\{a, \dots, f\}$.

	a	b	c	d	e	f
Frequency (Thousands)	45	13	12	16	9	5
Code word with fix length	000	001	010	011	100	101
Code word variable length	0	101	100	111	1101	1100

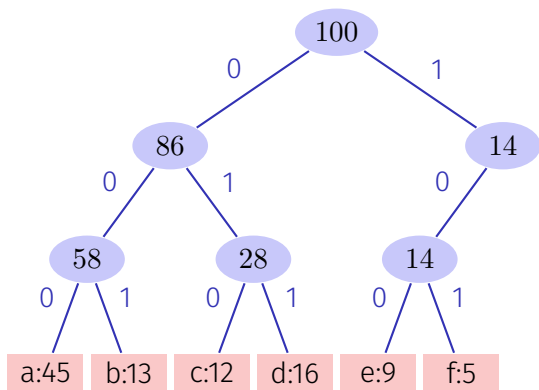
File size (code with fix length): 300.000 bits.

File size (code with variable length): 224.000 bits.

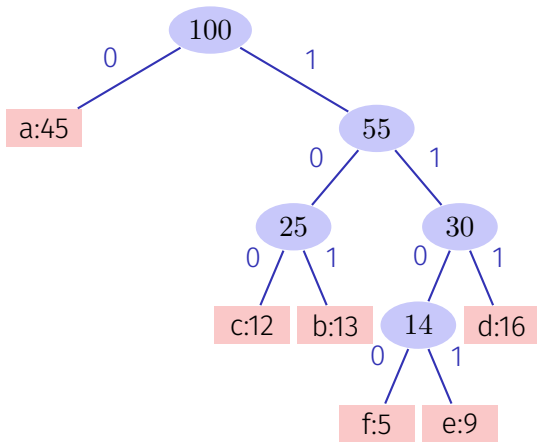
Huffman-Codes

- Consider prefix-codes: no code word can start with a different codeword.
- Prefix codes can, compared with other codes, achieve the optimal **data compression** (without proof here).
- Encoding: concatenation of the code words without stop character (difference to morsing).
 $af fe \rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow 0110011001101$
- Decoding simple because prefixcode
 $0110011001101 \rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow af fe$

Code trees



Code words with fixed length



Code words with variable length

Properties of the Code Trees

- An optimal coding of a file is always represented by a complete binary tree: every inner node has two children.
- Let C be the set of all code words, $f(c)$ the frequency of a codeword c and $d_T(c)$ the depth of a code word in tree T . Define the **cost** of a tree as

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c).$$

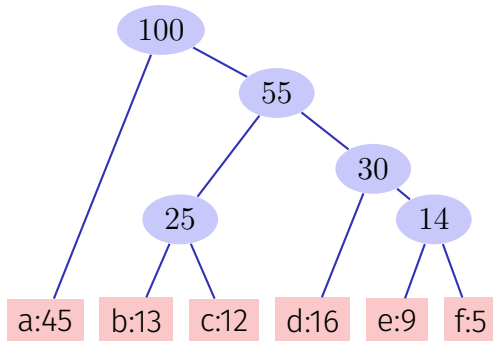
(cost = number bits of the encoded file)

In the following a code tree is called optimal when it minimizes the costs.

Algorithm Idea

Tree construction bottom up

- Start with the set C of code words
- Replace iteratively 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

$z.\text{left} \leftarrow \text{ExtractMin}(Q)$

// extract word with minimal frequency.

$z.\text{right} \leftarrow \text{ExtractMin}(Q)$

$z.\text{freq} \leftarrow z.\text{left}.\text{freq} + z.\text{right}.\text{freq}$

$\text{Insert}(Q, z)$

return $\text{ExtractMin}(Q)$

Analyse

Use a heap: build Heap in $\mathcal{O}(n)$. Extract-Min in $\mathcal{O}(\log n)$ for n Elements.
Yields a runtime of $\mathcal{O}(n \log n)$.

The greedy approach is correct

Theorem 21

Let x, y be two symbols with smallest frequencies in C and let $T'(C')$ be an optimal code tree to the alphabet $C' = C - \{x, y\} + \{z\}$ with a new symbol z with $f(z) = f(x) + f(y)$. Then the tree $T(C)$ that is constructed from $T'(C')$ by replacing the node z by an inner node with children x and y is an optimal code tree for the alphabet C .

Proof

It holds that

$$f(x) \cdot d_T(x) + f(y) \cdot d_T(y) = (f(x) + f(y)) \cdot (d_{T'}(z) + 1) = f(z) \cdot d_{T'}(x) + f(x) + f(y).$$

Thus $B(T') = B(T) - f(x) - f(y)$.

Assumption: T is not optimal. Then there is an optimal tree T'' with $B(T'') < B(T)$. We assume that x and y are brothers in T'' . Let T''' be the tree where the inner node with children x and y is replaced by z . Then it holds that $B(T''') = B(T'') - f(x) - f(y) < B(T) - f(x) - f(y) = B(T')$. Contradiction to the optimality of T' .

The assumption that x and y are brothers in T'' can be justified because a swap of elements with smallest frequency to the lowest level of the tree can at most decrease the value of B .

24. C++ advanced (IV): Exceptions

Some operations that can fail

- Opening files for reading and writing

```
std::ifstream input("myfile.txt");
```

- Parsing

```
int value = std::stoi("12-8");
```

- Memory allocation

```
std::vector<double> data(ManyMillions);
```

- Invalid data

```
int a = b/x; // what if x is zero?
```


Possibilities of Error Handling

- None (inacceptable)
- Global error variable (flags)
- Functions returning Error Codes
- Objects that keep error status
- Exceptions

Global error variables

- Common in older C-Code
- Concurrency is a problem.
- Error handling at good will. Requires extreme discipline, documentation and litters the code with seemingly unrelated checks.

Functions Returning Error Codes

- Every call to a function yields a result.
- Typical for large APIs (e.g. OS level). Often combined with global error code.⁴⁰
- Caller can check the return value of a function in order to check the correct execution.

⁴⁰Global error code thread-safety provided via thread-local storage.

Functions Returning Error Codes

Example

```
#include <errno.h>
...

pf = fopen ("notexisting.txt", "r+");
if (pf == NULL) {
    fprintf(stderr, "Error opening file: %s\n", strerror( errno ));
}
else { // ...
    fclose (pf);
}
```

Error state Stored in Object

- Error state of an object stored internally in the object.

Example

```
int i;  
std::cin >> i;  
if (std::cin.good()){// success, continue  
    ...  
}
```

Exceptions

- Exceptions break the normal control flow
- Exceptions can be thrown (throw) and caught (catch)
- Exceptions can become effective accross function boundaries.

Example: throw exception

```
class MyException{};
```

```
void f(int i){  
    if (i==0) throw MyException();  
    f(i-1);  
}
```

```
int main()  
{  
    f(4);  
    return 0;  
}
```

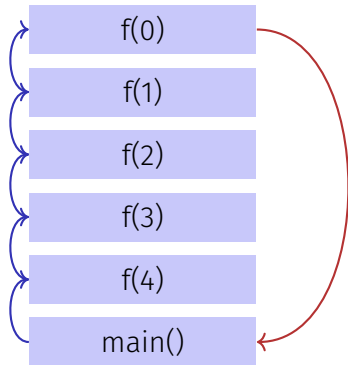
terminate called after throwing an instance of 'MyException'
Aborted

Example: catch exception

```
class MyException{};

void f(int i){
    if (i==0) throw MyException();
    f(i-1);
}

int main(){
    try{
        f(4);
    }
    catch (MyException e){
        std::cout << "exception caught\n";
    }
}
```



exception caught

Resources get closed

```
class MyException{};
struct SomeResource{
    ~SomeResource(){std::cout << "closed resource\n";}
};
void f(int i){
    if (i==0) throw MyException();
    SomeResource x;
    f(i-1);
}
int main(){
    try{f(5);}
    catch (MyException e){
        std::cout << "exception caught\n";
    }
}
```

closed resource
closed resource
closed resource
closed resource
closed resource
exception caught

When Exceptions?

Exceptions are used for **error handling** exclusively.

- Use **throw** only in order to identify an error that violates the post-condition of a function or that makes the continued execution of the code impossible in an other way.
- Use **catch** only when it is clear how to handle the error (potentially re-throwing the exception)
- Do **not** use **throw** in order to show a programming error or a violation of invariants, use **assert** instead.
- Do **not** use exceptions in order to change the control flow. Throw is **not** a better return.

Why Exceptions?

This:

```
int ret = f();  
if (ret == 0) {  
    // ...  
} else {  
    // ...code that handles the error...  
}
```

may look better than this on a first sight:

```
try {  
    f();  
    // ...  
} catch (std::exception& e) {  
    // ...code that handles the error...  
}
```

Why exceptions?

Truth is that toy examples do not necessarily hit the point.

Using return-codes for error handling either pollutes the code with checks or the error handling is not done right in the first place.

That's why

Example 1: Expression evaluation (expression parser from Introduction to programming)

Input: `1 + (3 * 6 / (/ 7))`

Error is deep in the recursion hierarchy. How to produce a meaningful error message (and continue execution)? Would have to pass error code over recursion steps.

Second Example

Value type with guarantee: values in range provided.

```
template <typename T, T min, T max>
class Range{
public:
    Range(){}
    Range (const T& v) : value (v) {
        if (value < min) throw Underflow ();
        if (value > max) throw Overflow ();
    }
    operator const T& () const {return value;}
private:
    T value;
};
```

Error handling in the constructor.

Types of Exceptions, Hierarchical

```
class RangeException {};  
class Overflow : public RangeException {};  
class Underflow : public RangeException {};  
class DivisionByZero: public RangeException {};  
class FormatError: public RangeException {};
```

Operators

```
template <typename T, T min, T max>
Range<T, min, max> operator/ (const Range<T, min, max>& a,
                             const Range<T, min, max>& b){
    if (b == 0) throw DivisionByZero();
    return T (a) * T(b);
}
```

```
template <typename T, T min, T max>
std::istream& operator >> (std::istream& is, Range<T, min, max>& a){
    T value;
    if (!(is >> value)) throw FormatError();
    a = value;
    return is;
}
```

Error handling in the operator.

Error handling (central)

```
Range<int,-10,10> a,b,c;
try{
    std::cin >> a;
    std::cin >> b;
    std::cin >> c;
    a = a / b + 4 * (b - c);
    std::cout << a;
}
catch(FormatError& e){ std::cout << "Format error\n"; }
catch(Underflow& e){ std::cout << "Underflow\n"; }
catch(Overflow& e){ std::cout << "Overflow\n"; }
catch(DivisionByZero& e){ std::cout << "Divison By Zero\n"; }
```