General Methods for Algorithm Design

1. Dynamic Programming
   - Multiplication of matrices
   - Elements of the dynamic programming
   - Optimal triangulation of polygons
   - Longest common subsequence

2. Greedy Methods
   - Scheduling problems
   - The Knapsack Problem
   - Matroids
   - Minimal and maximal spanning trees
   - Huffman codes

Let A be a $p \times q$ matrix and let B be a $q \times r$ matrix. Then $C = A \cdot B$ is a $p \times r$ matrix.

One needs pqr multiplications to compute C:

Algorithm 1 $\mathrm{MM}(A, B)$;

if (# Columns$(A) \neq$ # Rows(B)) **then**
  ERROR
else **for** $i = 1$ **to** # Rows(A) **do**
  for $j = 1$ **to** # Columns(B) **do**
    $C[i, j] := 0$
    for $k = 1$ **to** # Rows(B) **do**
      $C[i, j] := C[i, j] + A[i, k] \cdot B[k, j]$
return C

The product operation $A_1A_2A_3$ is associative and the number of multiplications strictly depends on the order.

Example 1 *Let* A_1 10×100, A_2 100×5 *and* A_3 5×50 *matrices.*

The scheme $((A_1A_2)A_3)$ *requires* $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$ *multiplications.*
The scheme $(A_1(A_2A_3))$ *requires* $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$ *multiplications.*

Problem:
Find an optimal multiplication order for $A_1A_2...A_n$.

For $n = 4$ there are only 5 orders:

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

In general the number of orders is

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & \text{if } n \geq 2. \end{cases}$$

This implies:

$$P(n) = \frac{1}{n}\binom{2n-2}{n-1} = \Omega\left(\frac{4^n}{n^{3/2}}\right).$$

Therefore, "the complete choice" is exponential in time.

a. Optimal Multiplication Order:

Let

$$A_{i..j} = A_i \cdot ... \cdot A_j.$$

There exists k $(i \leq k < j)$, such that the optimal multiplication order for $A_i \cdot ... \cdot A_j$ is or the form:

$$A_i \cdot ... \cdot A_j = A_{i..k} \cdot A_{k+1..j},$$

and the orders for the products for $A_{i..k}$ and $A_{k+1..j}$ are optimal.

b. Recursive Solution:

Let $m[i,j]$ be the minimal number of multiplications in the optimal order for $A_{i..j}$ and A_i be a $p_{i-1} \times p_i$ matrix. One has:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j. \end{cases}$$

Let $p = \langle p_0, ..., p_n \rangle$.

Algorithm 2 $\mathrm{RMC}(p,i,j)$;

if $(i = j)$ **then**
  return 0
$m[i,j] := \infty$
for $k = i$ **to** $j-1$ **do**
  $q := \mathrm{RMC}(p,i,k) + \mathrm{RMC}(p,k+1,j) + p_{i-1}p_k p_j$
  if $(q < m[i,j])$ **then**
    $m[i,j] := q$
return $m[i,j]$

Our goal is to compute $m[1,n] = \mathrm{RMC}(p,1,n)$. The running time:

$$T(n) \ge 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1,$$

or

$$T(n) \ge 2 \cdot \sum_{k=1}^{n-1} T(k) + n,$$

which implies

$$T(n) \ge 2^{n-1}.$$

To show this we apply the induction on n. $T(1) \geq 1 = 2^0$. Assuming $T(i) \geq 2^{i-1}$ for any $i < n$, one has

$$\begin{aligned} T(n) &\geq 2 \cdot \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \cdot \sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \\ &= 2^n + (n-2) \\ &\geq 2^{n-1}. \end{aligned}$$

Why is the running time so large ?

Many computations are repeated several times:

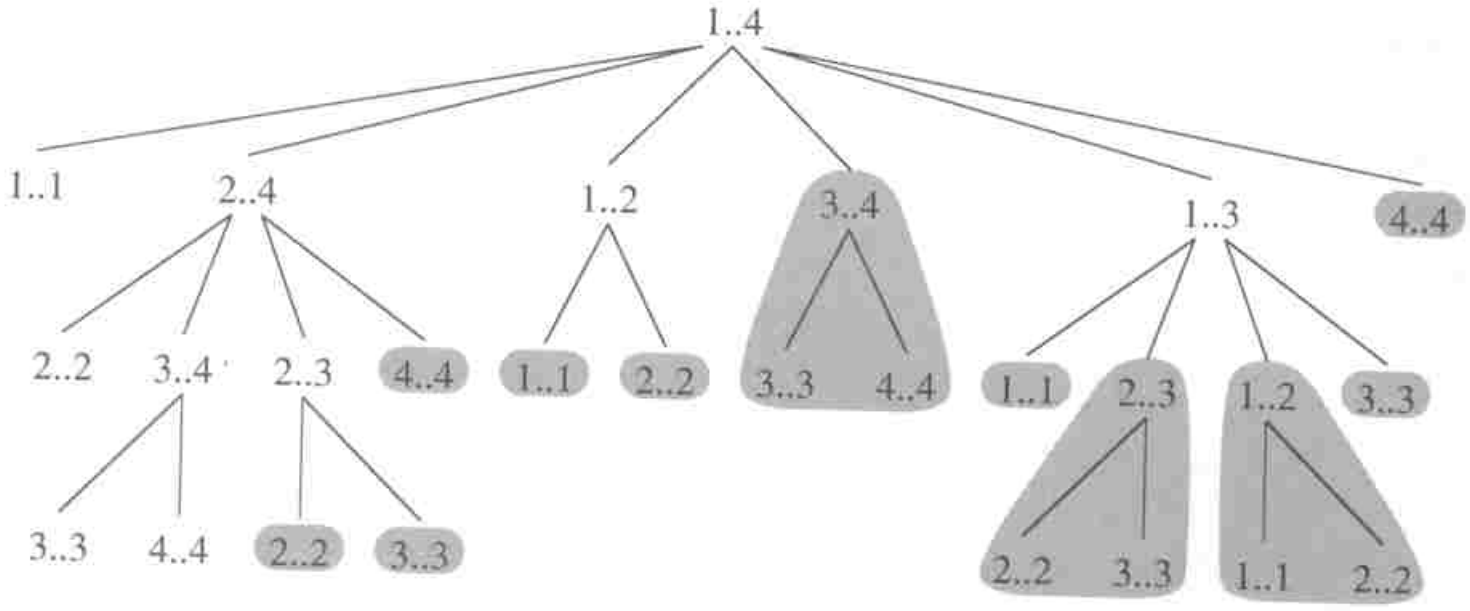

Figure 1: The subproblem structure

c. A Better Algorithm:

The number of different subproblems is $O(n^2)$.

The "bottom-up" principle.

Algorithm 3 $\mathrm{MCO}(p)$;

for $i = 1$ **to** n **do**
  $m[i, i] := 0$

for $l = 2$ **to** n **do**
  for $i = 1$ **to** $n - l + 1$ **do**
    $(j := i + l - 1; \quad m[i, j] := \infty)$

      for $k = i$ **to** $j - 1$ **do**
        $q := m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
        if $(q < m[i, j])$ **then**
          $m[i, j] := q$
          $s[i, j] := k$

return m and s

The running time of MCO is $O(n^3)$. Moreover MCO uses $O(n^2)$ memory to store the values of m and s.

Example 2 *Let* $p = (30, 35, 15, 5, 10, 20, 25)$.

$j \setminus i$	1	2	3	4	5	6
6	**15125**	10500	5375	3500	5000	0
5	11875	$\boxed{7125}$	2500	1000	0	
4	9375	4375	750	0		
3	7875	2625	0			
2	15750	0				
1	0					

Table of $m[i, j]$.

$j \setminus i$	1	2	3	4	5
6	3	3	3	5	5
5	3	$\boxed{3}$	3	4	
4	3	3	3		
3	1	2			
2	1				

Table of $s[i, j]$.

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 & (= 13000) \\ m[2,3] + m[4,5] + p_1 p_3 p_5 & (= 7125) \\ m[2,4] + m[5,5] + p_1 p_4 p_5 & (= 11375) \end{cases} \\
&= 7125
\end{aligned}
$$

d. Construction of optimal solution:

Let $A = \{A_1, A_2, ..., A_n\}$.

$\mathrm{MCM}(A, s, 1, n)$

Algorithm 4 $\mathrm{MCM}(A, s, i, j)$;

if $(i = j)$ **then**
  print "A_i"
else print "("
    $\mathrm{MCM}(A, s, i, s[i, j])$
    $\mathrm{MCM}(A, s, s[i, j] + 1, j)$
    print ")"

Optimal multiplication order for our example:

$$((A_1(A_2A_3))((A_4A_5)A_6)).$$

2. Elements of dynamic programming:

- Structure of the optimal solution
- Overlapping subproblems
- Memorization

Memorized version of RMC

Algorithm 5 $\mathrm{MMC}(p)$;

for $i = 0$ **to** n **do**
  for $j = i$ **to** n **do**
    $m[i, j] := \infty$
return $\mathrm{LC}(p, 1, n)$

Algorithm 6 $\mathrm{LC}(p, i, j)$;

if $(m[i, j] < \infty)$ **then**
  return $m[i, j]$

if $(i = j)$ **then**
  $m[i, j] := 0$
else for $k = i$ **to** $j - 1$ **do**
    $q := \mathrm{LC}(p, i, k) + \mathrm{LC}(p, k + 1, j) + p_{i-1} p_k p_j$
    if $(q < m[i, j])$ **then**
      $m[i, j] := q$

return $m[i, j]$

This "top-down" algorithm computes the solution in $O(n^3)$ time.

3. Triangulation of Polygons

Let P be a simple convex polygon with the node set

$$\{v_0, v_1, ..., v_n\}.$$

Definition 1 *The segments of the form $v_i v_{i+1 \bmod n+1}$ are called edges and the segments of the form $v_i v_j$ which are not edges are called chords of P.*

Definition 2 *A triangulation T of P is a set of chords which partition P into disjoint triangles.*

Assume we are given a weight function $w : \{v_i v_j\} \mapsto R^+$ satisfying $w(v_i v_j) = w(v_j v_i)$ for all $0 \leq i, j \leq n$, $i \neq j$. Furthermore, let $v_i v_j v_k$ be a triangle of the triangulation. Define its weight as

$$w(v_i v_j v_k) = w(v_i v_j) + w(v_i v_k) + w(v_j v_k).$$

The weight of a triangulation T is defined as the sum of weights of all its triangles.

Problem:
Given a polygon P, find a triangulation T of P of minimum weight.

We call such triangulation optimal.

a. The optimal solution structure

Let T be an optimal triangulation of P which contains a triangle $v_0v_kv_n$ for some $1 \leq k < n$. Then T consists of optimal triangulations of polygons $v_0, ..., v_k$ and $v_k, v_{k+1}, ..., v_n$.

b. Splitting into subproblems

Let $w[i, j]$ be the weight of an optimal triangulation of the polygon $v_{i-1}, ..., v_j$.

The number of subproblems is $O(n^2)$.

c. Recursive solution

It holds:

$$w[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq k < j} \{w[i,k] + w[k+1,j] + w(v_{i-1}v_kv_j)\} & \text{if } i < j. \end{cases}$$

Therefore, the optimal triangulation of P can be constructed with (almost) the same algorithm as for the matrix multiplication.

4. Longest Common Substring

Definition 3 *Let $X = (x_1, ..., x_m)$ and $Z = (z_1, ..., z_k)$ be strings. A string Z is called substring of X if $\exists\ i_1, ..., i_k\ (i_1 < \cdots < i_k)$, such that $x_{i_j} = z_j$ for $j = 1, ..., k$.*

Definition 4 *Let X and Y be strings. A string Z is called common substring of X and Y if Z is a substring of X and of Y.*
Z is called longest common substring if its length is maximum.
Denotation: $Z = CS(X, Y)$, resp. $Z = LCS(X, Y)$.

Example 3 *Let*

$$X = (A, B, C, B, D, A, B) \qquad Y = (B, D, C, A, B, A).$$

One has:

$$\begin{aligned} Z &= (B, C, A) = CS(X, Y) \\ Z' &= (B, C, B, A) = LCS(X, Y). \end{aligned}$$

Problem:
Given strings X and Y, compute $LCS(X, Y)$.

a. The structure of $LCS(X, Y)$

For a string $X = (x_1, ..., x_n)$ denote $X_i = (x_1, ..., x_i)$, $i = 1, ..., n$.

Proposition 1 *Let $X = (x_1, ..., x_m)$ and $Y = (y_1, ..., y_n)$ be strings and let $Z = (z_1, ..., z_k) = LCS(X, Y)$. Then:*

1. $x_m = y_n \Longrightarrow z_k = x_m$ *or* $z_k = y_n$, *so* $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$
2. $x_m \neq y_n$, $z_k \neq x_m \Longrightarrow Z = LCS(X_{m-1}, Y)$
3. $x_m \neq y_n$, $z_k \neq y_n \Longrightarrow Z = LCS(X, Y_{n-1})$

Note that if $x_m = y_n$ then WLOG $z_k = x_m$ and $z_k = y_n$.
If $x_m \neq y_n$ then $z_k \neq x_m$ or $z_k \neq y_n$ (or both).

b. Recursive solution

Let $c[i, j]$ be the length of $LCS(X_i, Y_j)$. One has:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j-1],\, c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

The number of subproblems is $\Theta(mn)$.

c. The LCS Algorithm

Algorithm 7 $\mathrm{LCS}(X, Y)$;

for $i = 1$ **to** m **do**
  $c[i, 0] := 0$
for $j = 0$ to n **do**
  $c[0, j] := 0$

for $i = 1$ **to** m **do**
  for $j = 1$ **to** n **do**
    if $(x_i = y_j)$ **then**
      $c[i, j] := c[i - 1, j - 1] + 1$
      $b[i, j] :=$ "$\nwarrow$"
    else if $(c[i - 1, j] \geq c[i, j - 1])$ **then**
      $c[i, j] := c[i - 1, j]$
      $b[i, j] :=$ "$\uparrow$"
    else
      $c[i, j] := c[i, j - 1]$
      $b[i, j] :=$ "$\leftarrow$"

return c and b

The running time of LCS is $O(mn)$.

Example 4 *Let*

$$X = (A, B, C, B, D, A, B) \quad \textit{and} \quad Y = (B, D, C, A, B, A)$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

$$LCS(X, Y) = (B, C, B, A)$$

Construction of LCS

Algorithm call
PRINT-LCS$(b, X, |X|, |Y|)$ or PRINT-LCS$(b, Y, |X|, |Y|)$

Algorithm 8 PRINT-LCS(b, X, i, j);

```
if (i = 0 or j = 0) then
    return

if (b[i, j] = "↖") then
   PRINT-LCS(b, X, i − 1, j − 1)
   print(X_i)
else
    if (b[i, j] = "↑") then
       PRINT-LCS(b, X, i − 1, j)
    else
       PRINT-LCS(b, X, i, j − 1)
```

The running time of PRINT-LCS is $O(m + n)$.

5. Greedy Methods

Let S be a set of n lectures to be given in the same hall. Each lecture V_i is characterized by its starting time s_i and terminating time f_i for $i = 1, ..., n$.

We represent V_i by the interval $[s_i, f_i)$. The intervals (resp. lectures) $[s_i, f_i)$ and $[s_j, f_j)$ are called *compatible* iff

$$f_i \leq s_j \qquad \text{or} \qquad f_j \leq s_i$$

The Scheduling Problem:
Find a maximal set of compatible lectures.

We assume that the lectures are sorted so that

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

Let $s = (s_1, ..., s_n)$ and $f = (f_1, ..., f_n)$.

Algorithm 9 SCHEDULE(s, f);

$A := \{1\}$
$j := 1$
for $i = 2$ **to** n **do**
  if $(s_i \geq f_j)$ **then**
    $A := A \cup \{i\}$
    $j := i$
return A

The running time of the SCHEDULE algorithm is $\Theta(n)$ if the terminating times $\{f_i\}$ are sorted in advance.

Theorem 1 [1] *The* SCHEDULE *algorithm provides a solution to the Schedule problem.*

The greedy method also allows to determine the minimum number of halls to schedule ALL lectures.

NOT ANY greedy method provides an optimal schedule!

Elements of the greedy method:

- Optimal strategy at each step
- Optimal structure of subproblems

6. The knapsack problem

Let $S = (s_1, ..., s_n)$ be a set of n objects. Each object s_i is characterized by a weight w_i and cost p_i, $i = 1, ..., n$. Furthermore, let $W > 0$ be the maximum weight of the objects in the knapsack.

0-1 VERSION:
Find a set of objects $P = \{s_{i_1}, ..., s_{i_k}\} \subseteq S$, such that

$$\sum_{j=1}^{k} w_{i_j} \leq W \qquad \text{and} \qquad \sum_{j=1}^{k} p_{i_j} \text{ is maximized.}$$

FRACTIONAL VERSION:
Find the coefficients c_i s.t. $0 \leq c_i \leq 1$ for $i = 1, ..., n$, such that

$$\sum_{i=1}^{n} c_i w_i \leq W \qquad \text{and} \qquad \sum_{i=1}^{n} c_i p_i \text{ is maximized.}$$

Theorem 2 *There exist a greedy method for solving the fractional version of the knapsack problem.*

For the proof we order the items so that

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$$

and load the knapsack with the the items in this order.

Algorithm 10 KNAPSACK(p, w, W);

Set $c_i = 0$ for $i = 1, \ldots, n$
$i = 1$
while ($W > 0$ && $i \leq n$) **do**
  $c_i = \min\{1, W/w_i\}$
  $W = W - c_i \cdot w_i$
  $i{+}{+}$
return (c[])

We have to show that

- There exists an optimal packing where the first item is selected according to the above algorithm.
- The remaining capacity of the knapsack (if it is any) must be packed in an optimal order.

To show the first assertion, take an optimal packing K and assume that $c_1 < \min\{1, W/w_1\}$. If $c_i > 0$ for some $i \geq 2$, let δ and ϵ be such that

$$(c_1 + \delta)w_1 + (c_i - \epsilon)w_i = c_1 w_1 + c_i w_i,$$

which equates to $\delta w_1 = \epsilon w_i$. We construct a new packing K' with the same weight accordingly. Then $(c_1 + \delta)p_1 + (c_i - \epsilon)p_i - (c_1 p_1 + c_i p_i) \geq 0$, so the new packing is optimal too. Repeating this process will lead to the packing with $c_1 = \min\{1, W/w_1\}$.

The proof of the second assertion is left to the reader.

Solution of 0-1 KNAPSACK with Dynamic Programming

Denote by $K(W, j)$ the maximum cost of packed items for a Knapsack of capacity W and items $1, \ldots, j$. One has

$$K(W, j) = \max\{K(W - w_j, j - 1) + p_j, K(W, j - 1)\}$$

Algorithm 11 KNAPSACK(p, w, W);

Set $K(0, j) = 0$ for $j = 1, \ldots, n$
Set $K(w, 0) = 0$ for all w
for $j = 1$ **to** n **do**
  for $w = 1$ **to** W **do**
    if $(w_j > w)$ **then**
      $K(w, j) = K(w, j - 1)$
    else
      $K(w, j) = \max\{K(w - w_j, j - 1) + p_j, K(w, j - 1)\}$
return $K(W, n)$

The running time is $O(nW)$.

7. Matroids

Under what general conditions do the greedy methods work?

Definition 5 *A matroid is a pair $M = (S, \mathcal{I})$, where $S \neq \emptyset$ is a finite set of elements and $\mathcal{I} \subseteq 2^S$ is a nonempty collection of subsets of S (independent subsets), such that:*
a. $[B \in \mathcal{I}] \wedge [A \subseteq B] \Longrightarrow A \in \mathcal{I}$.
b. $[A, B \in \mathcal{I}] \wedge [|A| < |B|] \Longrightarrow \exists x \in B \setminus A$ *s.t.* $A \cup \{x\} \in \mathcal{I}$.
The element x is called extention of A.

Example 5 *Let S be a finite set and $\mathcal{I}_k$ be the collection of all subsets of S consisting of at most k elements. Then $(S, \mathcal{I}_k)$ is a matroid.*

Example 6 *Let S be the set of columns of an $n \times n$ matrix with real entries and $A \subseteq S$. We put $A \in \mathcal{I} \Longleftrightarrow$ the columns in A are linearly independent. Then $(S, \mathcal{I})$ is a matroid.*

Definition 6 *Let $(S, \mathcal{I})$ be a matroid and let a set $A \in \mathcal{I}$ has no extention. Then the set A is called maximal.*

Proposition 2 *Let A and B be maximal independent subsets of a matroid. Then $|A| = |B|$.*

Proof. If $|A| < |B|$ for some maximal sets A, B, then there must be $x \in B \setminus A$, s.t. $A \cup \{x\}$ is independent, which contradicts the maximality of A. $\square$

Definition 7 *A matroid $M = (S, \mathcal{I})$ is called weighed is there is a function $w : S \mapsto \mathbf{R}^+$ assigning a weight to every $x \in S$.*

For $A \subseteq S$ define the weight of A:

$$w(A) = \sum_{x \in A} w(x).$$

Definition 8 *An independent set $A \in \mathcal{I}$ of a weighed matroid is called optimal if it has maximum weight.*

The Matroid Problem:
Given a weighed matroid M, find its optimal independent set(s).

We assume that the elements of $S = \{x_1, ..., x_n\}$ are sorted so that $w(x_1) \geq w(x_2) \geq \cdots \geq w(x_n)$.

Algorithm 12 GREEDY(M, w);

$A := \emptyset$
for $i = 1$ **to** n **do**
  if $(A \cup \{x_i\} \in \mathcal{I})$ **then**
    $A := A \cup \{x_i\}$
return A

Theorem 3 [1] *The* GREEDY *algorithm constructs an optimal independent set.*

The proof is based on the following 3 Lemmas:

Lemma 1 [1] *Let $M = (S, \mathcal{I})$ be a matroid and assume $\{x\} \in S$ is not an extension of $\{\emptyset\}$. Then x is not an extension of any independent set $A \in \mathcal{I}$.*

Proof. The contrapositive of the statement is: if $x \in S$ is an extension of some $A \in \mathcal{I}$, then $\{x\} \in \mathcal{I}$. This directly follows from $A \cup \{x\} \in \mathcal{I}$ and the hereditary property of $\mathcal{I}$. □

Therefore, to construct an optimal set of M one can reduce S to those elements that appear in some independent set.

Lemma 2 [1] *Let $M = (S, \mathcal{I})$ be a weighed matroid and assume $x \in S$ is an element of maximum weight such that $\{x\} \in \mathcal{I}$. Then there exists an optimal set $A \subseteq \mathcal{I}$ with $x \in A$.*

Proof. Let A be an optimal set. If $x \in A$, we are done. Otherwise, $\exists y \in A \setminus \{x\}$, s.t. $\{x, y\} \in \mathcal{I}$. Repeating this argument several times leads to an maximal set C with $x \in C$ and $|C| = |A|$. Since $C \setminus A = \{x\}$, $w(C) \geq w(A)$. So, C is optimal. □

Hence, we can start with an element x provided by $\textsc{Greedy}$.

Lemma 3 [1] *Let x be the first element chosen by $\textsc{Greedy}$. Then the matroid problem for M can be reduced to a similar problem on the matroid $M' = (S', \mathcal{I}')$ and reduced weight function on S', where*

$$\begin{aligned} S' &= \{y \in S \mid \{x, y\} \in \mathcal{I}\}, \\ \mathcal{I}' &= \{B \subseteq S \setminus \{x\} \mid B \cup \{x\} \in \mathcal{I}\}. \end{aligned}$$

8. Graphs and their Spanning Trees

Definition 9 *A Graph G is a pair (V, E), where V is the set of vertices and*

$$E \subseteq \{(u, v) \mid u, v \in V,\ u \neq v\}$$

is the set of edges.

Definition 10 *A sequence of edges in a Graph G is a finite set of edges of the form*

$$(v_0, v_1),\ (v_1, v_2),\ \cdots (v_{l-1}, v_l),$$

where $l \geq 1$ is the sequence length.

A sequence of edges is called:

- *path, if all edges are distinct.*
- *Simple path, is the vertices $v_0, ..., v_l$ are distinct.*
- *Cycle, if it is a path with $v_0 = v_l$ and $l \geq 2$.*
- *Simple cycle, if it is a path with $v_0 = v_l$ and $l \geq 2$, where the vertices $v_0, ..., v_{l-1}$ are distinct.*

Definition 11 *A graph G is called connected, if for any $u, v \in V$ there exists a path from u to v.*

Definition 12 *A graph is called forest, if it contains no cycle. A connected forest is called tree.*

Definition 13 *A graph $H = (V_H, E_H)$ is called a subgraph of $G = (V_G, E_G)$, if $V_H \subseteq V_G$ and $E_H \subseteq E_G$.*

Definition 14 *A subgraph H of G is called spanning if $V_H = V_G$. If H is a tree, we call it spanning tree.*

Definition 15 *A graph $G = (V, E)$ is called weighed if there is a mapping $w : E \mapsto \mathbf{R}^+$. For a subgraph H of G the weight of H is defined as*

$$w(H) = \sum_{e \in E_H} w(E).$$

The Spanning Tree Problem:
Given a weighed connected graph G, construct its spanning tree of maximum weight.

Let $A \subseteq E_G$. Define a set $\mathcal{I}_G$ as follows:
$A \in \mathcal{I}_G \Longleftrightarrow A$ contains no cycle.

Theorem 4 *Let $G = (V_G, E_G)$ be a graph. Then $(E_G, \mathcal{I}_G)$ is a matroid (graphic matroid).*

Corollary 1 *The Spanning Tree Problem can be solved by a greedy method.*

The greedy method is also applicable to construct a spanning tree of minimum weight.

Proof of Theorem 4

We need to verify two properties of matroids.

For the hereditary property, if $A \subseteq E_G$ contains no cycle and $B \subseteq A$, then B can be obtained from A by dropping some edges. Obviously, this procedure cannot create a cycle in B.

For the exchange property, let $A, B \subseteq E_G$ and $|A| < |B|$. The edges of each A and B partition V_G into connected components. If k_A and k_B is the number of those components, one has

$$\begin{aligned} |V_G| &= |A| + k_A \\ |V_G| &= |B| + k_B. \end{aligned}$$

Since $|A| < |B|$ we get $k_A > k_B$. If there is an edge $e \in B$ connecting two different components of A then we are done, since $e \in B \setminus A$ and the set $A \cup \{e\}$ contains no cycle.

Assume the contrary: for every edge $e = (x, y) \in B$ its endpoints x, y belong to the same component of A. This implies that every component of B can intersect only one component of A. In other words, every component of B is contained in some component of A. Hence, $k_B \geq k_A$ which is a contradiction. $\square$

9. Algorithm of Kruskal

Let M be a set of elements and $A, B \subseteq M$, $A \cap B = \emptyset$.

We will need the following procedures:

$\textsc{Make-Set}(v)$: constructs a new set $\{v\}$ for $v \in M$.

$\textsc{Find-Set}(v)$: finds a set $A \subseteq M$ with $v \in A$.

$\textsc{Union}(u, v)$: computes the union of $A = \textsc{Find-Set}(u)$ and $B = \textsc{Find-Set}(v)$ if $A \cap B = \emptyset$.

Algorithm 13 MST-$\textsc{Kruskal}(G, w)$;

$B := \emptyset$
for all $v \in V(G)$ **do**
  $\textsc{Make-Set}(v)$
Sort the edges E according to the weights w in non-decreasing order.
for all $(u, v) \in E$ **do**
  if ($\textsc{Find-Set}(u) \neq \textsc{Find-Set}(v)$) **then**
    $B := B \cup (u, v)$
    $\textsc{Union}(u, v)$
return B

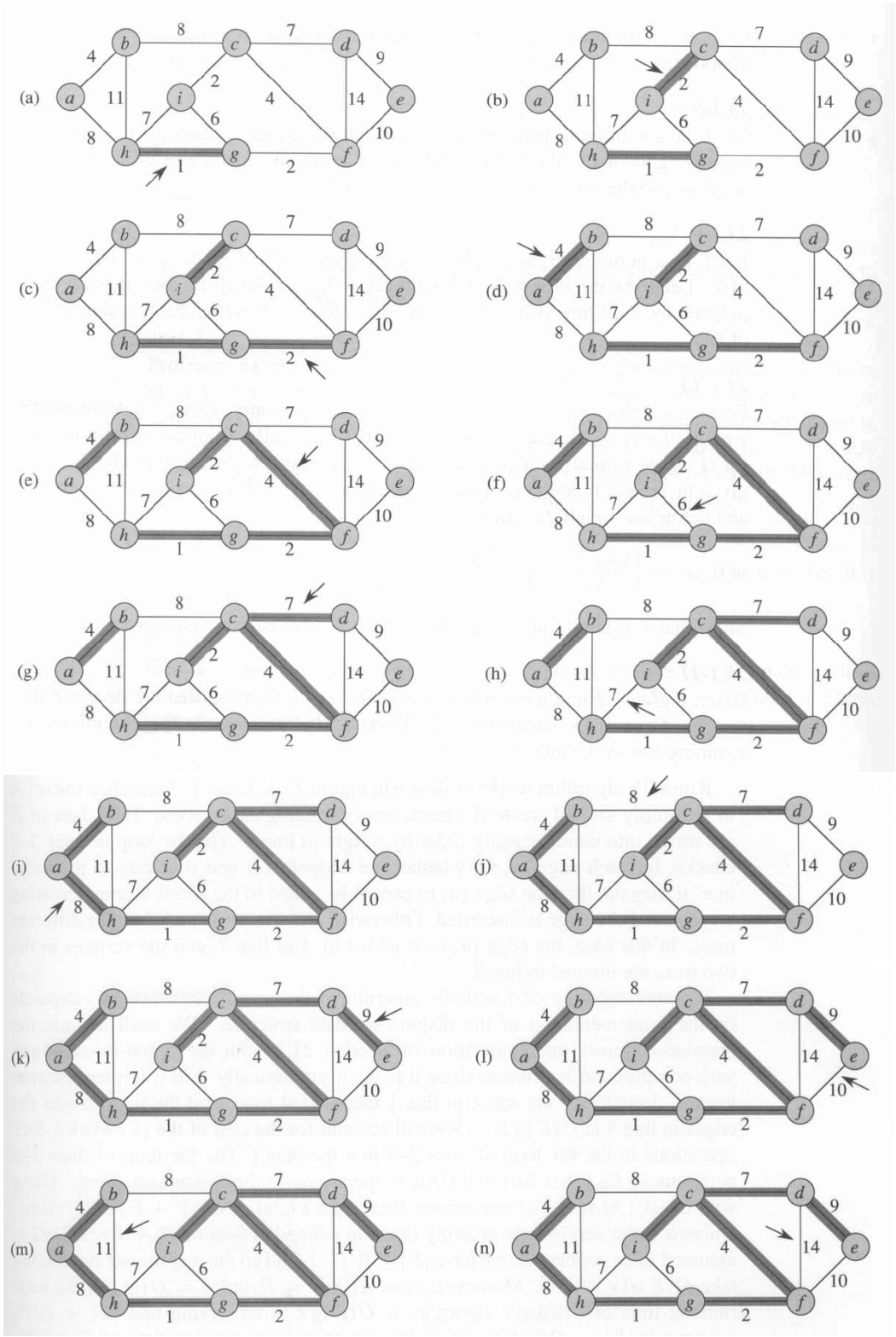

Figure 2: The algorithm of Kruskal

10. Algorithm of Prim

Let $r \in V(G)$. For a tree T with $V(T) \subset V(G)$ and $w \in V(G)$ let $key[w]$ be the min weight of an edge connecting w to T.
We will use the heap procedure $\textsc{Extract-Min}(Q)$.

Algorithm 14 $\textsc{MST-Prim}(G, w, r)$;

$Q := V(G)$ // set of unprocessed vertices
for each $u \in Q$ **do**
$key[u] := \infty$
$key[r] := 0$
$\pi[r] :=$ nil

while $(Q \neq \emptyset)$ **do**
$u := \textsc{Extract-Min}(Q)$
for each $v \in Adj[u]$ **do**
if ($v \in Q$ and $w(u, v) < key[v]$) **then**
$\pi[v] := u$
$key[v] := w(u, v)$

At each step the $\textsc{MST-Prim}$ constructs a tree, which is a subtree of a minimal spanning tree.

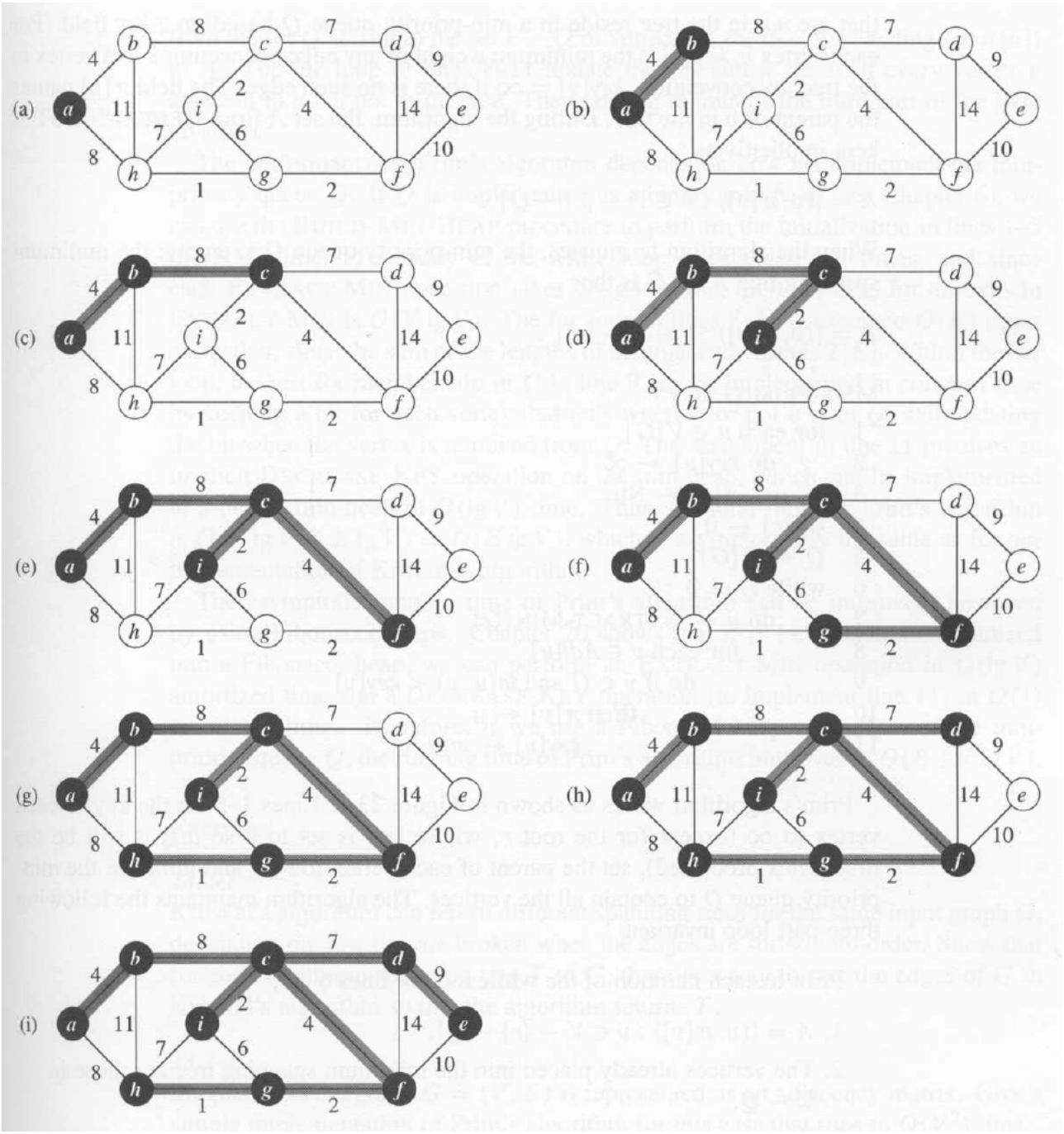

Figure 3: The algorithm of Prim

11. A task-scheduling problem

Instance: a set S of unit-length tasks $a_1, \ldots, a_n$. Each task a_i has a deadline d_i and a non-negative penalty w_i for being completed after the deadline.

Problem: find a schedule for S that minimizes the total penalty.

Example:

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

We call a task early if it finishes before its deadline and late otherwise. Any schedule can be put into the canonical form, where the early tasks preceed the late ones and the early tasks are scheduled in order of increasing deadlines.

We say a set of tasks $A \subseteq S$ is independent if there exists a schedule for these tasks where no task is late. For a set A of tasks denote by $N_t(A)$ the number of tasks in A whose deadline is t or earlier.

Lemma 4 *For any set of tasks the following is equivalent:*

a. *The set A is independent.*

b. *For $t = 0, \ldots, n$, we have $N_t(A) \leq t$.*

c. *If the tasks in A are scheduled in order of monotonically increasing deadlines, then no task is late.*

To construct a schedule with a minimum penalty for the late tasks is equivalent to maximizing the penalty of the early tasks. We use a greedy approach for that.

Theorem 5 *For a set S of the unit-length tasks and the collection $\mathcal{I}$ of all independent sets of tasks, the pair $(S, \mathcal{I})$ is a matroid.*

Proof.

The hereditary property is trivially satisfied, let us check the exchange property. Let A, B be sets of independent tasks, $|A| < |B|$ and denote $k = \max\{t \ : \ N_t(B) \leq N_t(A)\}$ (such a number exists since $N_0(A) = N_0(B) = 0$).

Note that $k < n$ and $N_j(B) > N_j(A)$ for $j = k+1, \ldots, n$. So, B has more tasks with deadline $k+1$ than A. Let $a_i \in B - A$ be one of them. We show that $A' = A \cup \{a_i\}$ is independent.

Indeed, $N_t(A') = N_t(A) \leq t$ for $0 \leq t \leq k$ since A is independent. Also, $N_t(A') \leq N_t(B) \leq t$ for $k < t \leq n$ since B is independent. Hence, A' is independent by Lemma 4(b). □

For the above example the Greedy selects the following tasks by checking at each step the condition $N_t(\cdot) \leq t$:

$$\underbrace{a_1\ a_2\ a_3\ a_4\ a_7}_{\text{early}}\ \underbrace{a_5\ a_6}_{\text{late}}$$

The early tasks are then scheduled according to Lemma 4(c):

$$\underbrace{a_2\ a_4\ a_1\ a_3\ a_7}_{\text{early}}\ \underbrace{a_5\ a_6}_{\text{late}}$$

The penalty of that schedule is $w_5 + w_6 = 50$ and it is minimum.

12. Huffman Code

Let C be a set of symbols where a symbol a_i presents f_i times for $i = 1, ..., n$. We encode the symbols $\{a_i\}$ with binary strings $\{w_i\}$ such that any string w_i is not a prefix of any other string w_i of the encoding. Denote:

$$L(C) = \sum_{i=1}^{n} f_i \cdot |w_i|.$$

Problem:
Given $\{f_i\}$, find a prefix encoding C such that $L(C)$ is minimized (optimal encoding).

For $n = 2$ the problem is trivial. We assume that

$$f_1 \geq f_2 \geq \cdots \geq f_n \qquad \text{and} \qquad n \geq 3$$

Proposition 3 [1] *There exists an optimal prefix encoding such that $|w_{n-1}| = |w_n|$ and the strings w_{n-1} and w_n differ in one bit only.*

Theorem 6 [1] (Optimal Substructure)
Let C be an optimal prefix encoding for $f_1, ..., f_n$
and C' be an optimal prefix encoding for $f_1, ..., f_{n-2}, f_{n-1} + f_n$.
Then:

$$L(C) = L(C') + f_{n-1} + f_n.$$

This Theorem provides a greedy method for constructing an optimal prefix encoding.

Let $f'_i = f_i/|C|$, $i = 1, ..., n$. Then:

$$f'_1 \geq f'_2 \geq \cdots \geq f'_n \qquad \text{and} \qquad \sum_{i=1}^{n} f'_i = 1.$$

The Huffman method consists of the following steps:

1. $\mathbf{while}\ |f'| > 2\ \ \mathbf{do}$
   a. Take two elements f'_{n-1} and f'_n of f' and construct a new sequence $(f'_1, ..., f'_{n-2}, f'_{n-1} + f'_n)$.
   b. Sort the new sequence in non-decreasing order.
2. Encode the sequence (g'_1, g'_2) with 0 and 1.
3. Repeat the procedure 1a. in the reversed direction:
   $\mathbf{while}\ |g'| < n\ \ \mathbf{do}$
   Let a sequence g' be of the form: $(g'_1, ..., f'_{m-1} + f'_m, ..., g'_{m-2})$
   and the corresponding encoding be $a_1, ..., a_i, ..., a_m$,
   where $m = |g'| \leq n - 1$.
   Construct a sequence $(g'_1, ..., g'_{i-1}, g'_{i+1}, ..., g'_m, f'_{m-1}, f'_m)$
   and encode it with $a_1, ..., a_{i-1}, a_{i+1}, ..., a_m, a_i0, a_i1$.

Example 7 *Let* $f' = (0.20, 0.20, 0.19, 0.12, 0.11, 0.09, 0.09)$.

0.20	*0.20*	*0.23*	*0.37*	*0.40*	*0.60*	*0*	*1*	*00*	*01*	*10*	*10*
0.20	*0.20*	*0.20*	*0.23*	*0.37*	*0.40*	*1*	*00*	*01*	*10*	*11*	*11*
0.19	*0.19*	*0.20*	*0.20*	*0.23*			*01*	*10*	*11*	*000*	*000*
0.12	*0.18*	*0.19*	*0.20*					*11*	*000*	*001*	*010*
0.11	*0.12*	*0.18*							*001*	*010*	*011*
0.09	*0.11*									*011*	*0010*
0.09											*0011*