Chapter 1. Minimum spanning trees and shortest paths

Section 1.1. Introduction

Discrete optimization or combinatorial optimization involves problems in which we have to choose the best solution from a finite number of relevant solutions.

A typical example of a discrete optimization problem is the following. We are given a set of elements $\{e_1, \ldots, e_n\}$, and each element has a weight w_i. The problem is to find a k-element subset S of maximum weight $w(S) = \sum_{e_i \in S} w_i$.

An *instance* of this problem is the following set of eight elements with weights, where the problem is to find a 5-element subset of maximum weight.

i	1	2	3	4	5	6	7	8
w_i	3	6	5	8	4	9	2	6

There are $\begin{pmatrix} 8 \\ 5 \end{pmatrix}$ feasible solutions. The *optimal* solution is the set $S = \{2, 3, 4, 6, 8\}$ with weight $w(S) = 6 + 5 + 8 + 9 + 6 = 34$.

The finiteness of the solution set suggests that the brute-force approach of *exhaustive* or *explicit enumeration* will be effective: simply generate all feasible solutions, examine their costs, and select the best one. Such an approach can be very time-consuming; as there is an upper bound on the number of operations that a computer can perform per time period, problems of very limited size only are solvable by explicit enumeration, and we have to search for faster algorithms when we want to start solving discrete optimization problems.

Section 1.2. Graph theory: basic definitions

A (undirected) *graph* G is a pair (V, E), where V is a set of elements called *vertices* or *nodes*. E is a set of 2-element subsets of V called *edges*. A graph $G = (V, E)$ has a graphical representation in which V is drawn as a set of dots and E as a set of line segments each connecting two dots.

Example

$V = \{1, 2, 3, 4, 5\}$
$E = \{\{1,2\}, \{1,4\}, \{1,5\}, \{2,3\}, \{2,4\}, \{3,4\}, \{4,5\}\}$

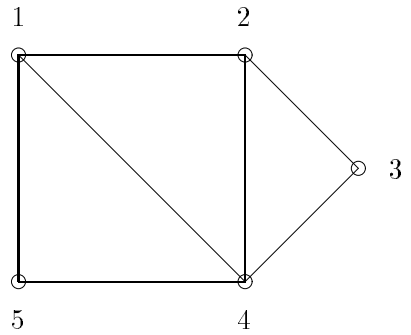


Figure 1.1

A vertex v and an edge e are called *incident* if $v \in e$. Two vertices v and w are called *adjacent* if $\{v, w\} \in E$. The *degree* of a vertex v, denoted by d_v or $d(v)$, is defined as the number of

edges to which v is incident. The total number of vertices is denoted by $|V|$ or n, and the total number of edges by $|E|$ or m. The following proposition is elementary:

Proposition 1.1. $\sum_{v \in V} d_v = 2 \cdot |E|$.

A *walk* is a sequence of vertices and edges $(v_1, \{v_1, v_2\}, v_2, \{v_2, v_3\}, \ldots, v_{K-1}, \{v_{K-1}, v_K\}, v_K)$. Usually, the edges are omitted in the notation of a walk, i.e., it is written down as a sequence of vertices $(v_1, v_2, \ldots, v_K)$. The *length* of a walk is equal to the number of edges in it. Hence, the length of the walk $(v_1, \ldots, v_K)$ is $K - 1$. A walk is called a *path* if the vertices $v_1, \ldots, v_K$ are all different. A *cycle* is a closed walk, i.e., a walk with $v_K = v_1$. A *circuit* is a cycle where $v_1, \ldots, v_{K-1}$ are all different.

In the graph of Figure 1.1,

- (1,3,2,5,3,4) is a walk, but not a path;
- (1,3,2,5) is a path;
- (1,3,2,5,3,4,1) is a cycle, but not a circuit;
- (1,3,2,1) is a circuit.

A graph $G = (V, E)$ is *connected*, if there is a path between each pair of vertices. A graph is *acyclic* if it does not contain any cycles.

Theorem 1.2.

(a) If a graph $G = (V, E)$ is acyclic, then $|E| \leq |V| - 1$,

(b) If a graph $G = (V, E)$ is connected, then $|E| \geq |V| - 1$.

Proof.

We only prove part (a) of the Theorem; part (b) can be proven by a similar reasoning. We use induction on $|V|$. Since the theorem trivially holds for $|V| = 2$, we may state our induction-hypothesis that the theorem is true for each graph on $|V| = n - 1$ vertices. We have to show that the theorem then also holds for each graph on n vertices.

Suppose to the contrary that there exists an acyclic graph $G = (V, E)$ on n vertices with $E \geq |V| = n$. If G contains a vertex with degree equal to zero or one, then we just remove this vertex and the corresponding edge (in case of degree one) and obtain an acyclic graph on $n - 1$ vertices with at least $n - 1$ edges, which contradicts the induction-hypothesis. Hence, we know that each vertex then must have degree at least equal to two. Since the graph is assumed to be acyclic, we can travel from each vertex to another one that we have not visited before. But after having traversed $n - 1$ edges, we have visited each vertex. We are then in a vertex with degree at least two, so we can leave this vertex through an edge that we have not traversed before and that must be incident to a vertex that we have already visited, which implies that G contains a cycle. This contradiction proves part (a) of Theorem 1.2. □

A *tree* $T = (V, E)$ is a connected acyclic graph. It contains a unique path between each pair of vertices. A direct corollary of Theorem 1.2 is that $|E| = |V| - 1$.

One of the earliest applications of graph theory is a characterization of an *Eulerian graph*; an Eulerian graph is a graph that contains a cycle in which each edge is traversed exactly once.

Theorem 1.3. (Euler)
A connected graph $G = (V, E)$ contains an Euler-cycle if and only if the degrees of all vertices are even.

A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V, E')$ with $E' \subseteq E$. An *induced* subgraph of G is a graph $G'' = (V'', E'')$ in which $V'' \subset V$ and E'' consists of all edges in E between two vertices of V''.

Figure 1.2 shows a subgraph and an induced subgraph (induced by the vertices $\{1, 4, 5\}$) of the graph of Figure 1.1, respectively.

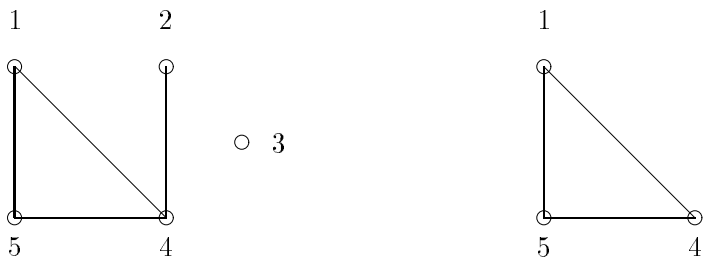


Figure 1.2.

If a graph is not connected, then it can be split into a set of *connected components* $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \ldots, G_K = (V_K, E_K)$, where $V_1, \ldots, V_K$ form a partition of the set of vertices and where each $G_k = (V_k, E_k)$ is the subgraph induced by V_k $(k = 1, \ldots, K)$.

A *directed* graph or digraph $G = (V, A)$ consists of a set of vertices V, and a set of *ordered* 2-element subsets. An example of a directed graph is depicted below.

$V = \{1, 2, 3, 4, 5\}$
$A = \{(1,5), (2,1), (2,4), (3,2),$
$(4,1), (4,3), (5,4)\}$

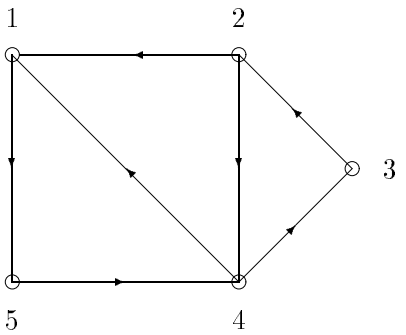


Figure 1.3.

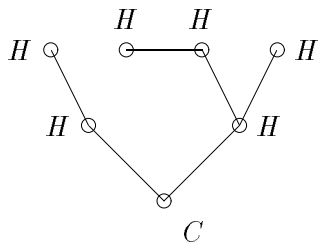
The terminology for digraphs resembles the terminology for (undirected) graphs. For instance, a walk in a digraph is a sequence of vertices and arcs: $(v_1, (v_1, v_2), v_2, (v_2, v_3), \ldots, v_{K-1}, (v_{K-1}, v_K), v_K)$ or shortly $(v_1, v_2, \ldots, v_K)$. Paths, cycles, and circuits are defined analogously. Note that the arcs in a walk have to be traversed in the right directions, i.e., the directions indicated by the arrows.

In the graph of Figure 1.3,

(1,5,4,3,2,4)	is a walk;	(1,5,4,3)	is a path;
(1,5,4,3,2,4,1)	is a cycle;	(1,5,4,1)	is a circuit.

Section 1.3. The minimum spanning tree problem

In small local area telephone networks, houses are connected with a central unit by a cable (coax or fiber). To realize such a network, sleeves must be dug in which these cables are laid. Sleeves may be dug between a pair of houses, and between a house and the central unit. There may be more cables in one sleeve. Since digging sleeves is an expensive process, the total length of the sleeves has to be minimized. A typical solution is drawn below.



This problem can be modeled as a minimization problem in graphs as follows. The central unit and the houses are represented by nodes. Between each pair of nodes v and w, there is an edge $e = \{v, w\}$ with a cost c_e that is equal to the length of a sleeve between v and w. We are looking for a subgraph $G' = (V, E')$ in which there is a path between each vertex and the central unit; we wish to find the subgraph for which the total cost of the edges in E' is minimal.

Formally, this is the problem of finding a minimum spanning tree $T = (V, E')$ in a graph $G = (V, E)$ with respect to a cost function $c : E \to \mathbb{Z}^+$, such that $c(E') = \Sigma_{e \in E'} c_e$ is minimal.

The following algorithm constructs a minimum spanning tree edge by edge in a greedy fashion.

Algorithm MST-1 (Kruskal)

Input: A connected graph $G = (V, E)$, and a cost function on the edges $c : E \to \mathbb{Z}^+$. Let $m = |E|$ be the number of edges.

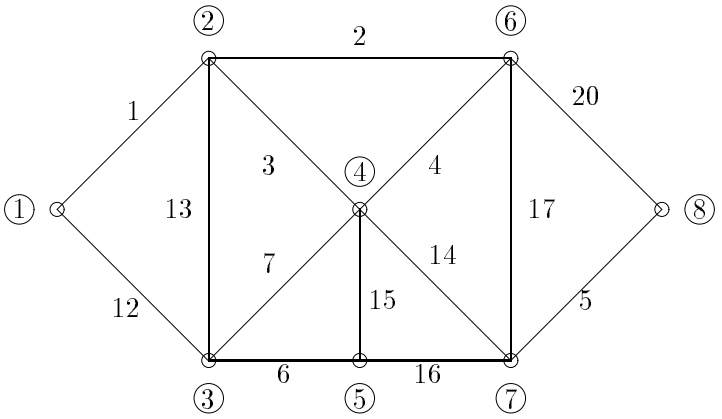
Output: A minimum spanning tree $T = (V, E'), E' \subseteq E$.

STEP 1. Number the edges according to non-increasing cost, that is, $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$. Set $E' \leftarrow \emptyset$. Set $j \leftarrow 1$.

STEP 2. Suppose that the edges $e_1, \ldots, e_{j-1}$ have been considered, and that an acyclic graph (V, E') has been formed. If $E' \cup \{e_j\}$ is acyclic, then e_j is added to E'; otherwise, E' remains unchanged. Set $j \leftarrow j + 1$.

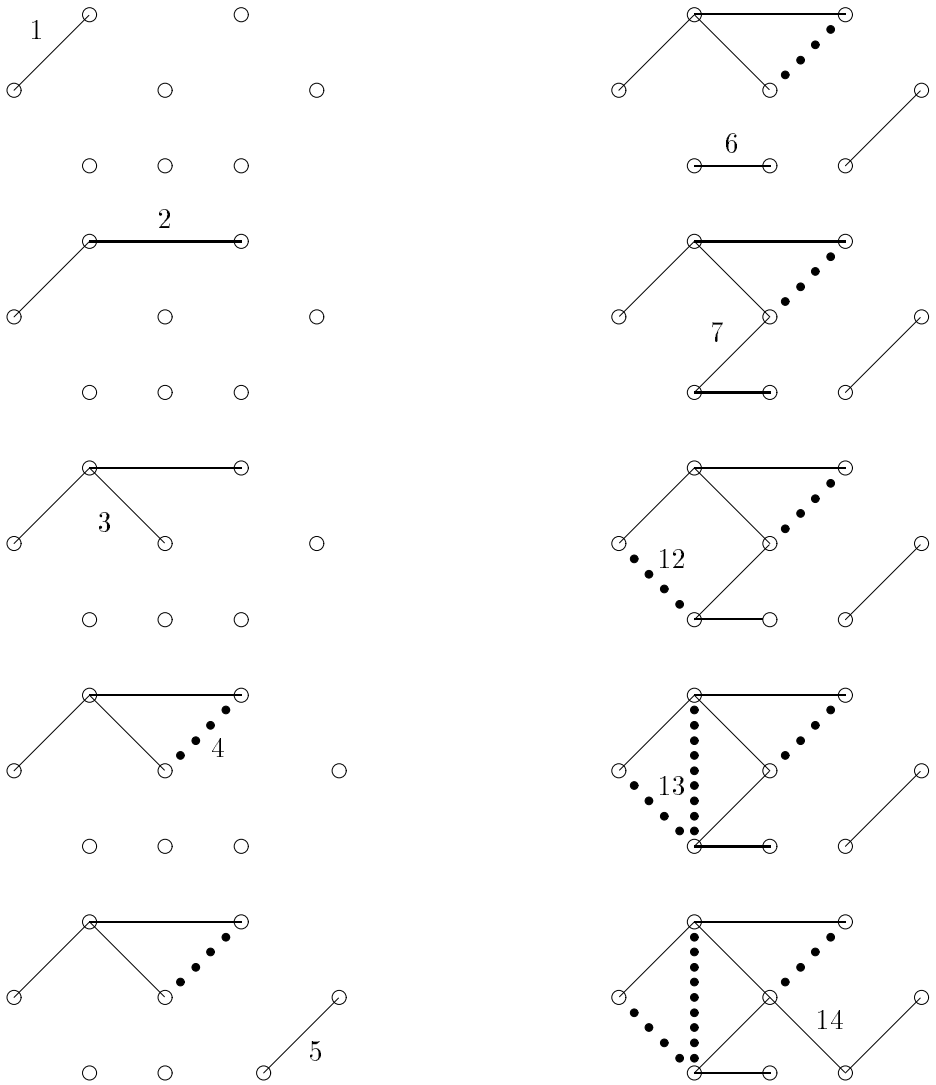
STEP 3. If the graph (V, E') is connected, then stop; otherwise, go to STEP 2.

Example



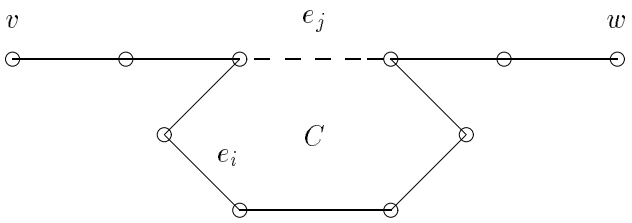
Ordering of the edges

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
e_i	{1,2}	{2,6}	{2,4}	{4,6}	{7,8}	{3,5}	{3,4}	{ 1,3}	{ 2,3}	{4,7}	{4,5}	{ 5,7}	{ 6,7}	{ 6,8}
$c(e_i)$	1	2	3	4	5	6	7	12	13	14	15	16	17	20
$\in E'(Y/N)$	Y	Y	Y	N	Y	Y	Y	N	N	Y				



Theorem 1.4.
Algorithm MST-1 constructs a minimum spanning tree.

Proof.
Let T_1 denote the tree determined by MST-1; renumber the edges such that e_k is the edge that was added to the tree by MST-1 as the k'th edge. Suppose to the contrary that MST-1 does not determine a minimum spanning tree; let T_2 be a minimum spanning tree. Compare T_1 and T_2; let e_i be the first edge from T_1 that is not in T_2. If we add e_i to T_2, then we get a cycle C. As e_i is chosen by MST-1, we know that C contains at least one edge other than $e_1, \ldots, e_{i-1}$; let this be e_j. Since T_2 contains the edges $e_1, \ldots, e_{i-1}$ and e_j, we have that e_j also was available when e_i was added; as e_i was chosen by MST-1, we know that $c(e_i) \leq c(e_j)$. Hence, if we replace e_j with e_i in T_2, then we obtain a spanning tree with cost no more than the cost of T_2. If we proceed our analysis with this new minimum spanning tree T_2, then we end up with the contradiction that the minimum spanning tree T_2, though consisting of the same set of edges as T_1, has length smaller than the length of T_1. This contradiction obviously shows that MST-1 determines a minimum spanning tree.



□

We now present another algorithm for the minimum spanning tree problem. This algorithm also builds a minimum spanning tree edge by edge in a greedy way, but it differs from MST-1 by maintaining one big acyclic connected component and a set of isolated vertices. In each iteration, an edge connecting an isolated vertex to the big component is added to the component.

The algorithm is implemented by use of a basic technique, called labeling. Each isolated vertex v contains a label expressing the cost of connecting v to the big component by means of the cheapest edge. This label is denoted by $l(v)$.

The technique of labeling returns later in this chapter and in Chapter 2. A label may contain any information, like the cost of an edge, a neighboring vertex or, other information with regard to the vertex.

The algorithm makes use of two additional sets V' and E'. The set V' contains all vertices that are in the big component; the set E' contains all edges that are used to form the big component.

Algorithm MST-2 (Prim-Dijkstra)

Input: A connected graph $G = (V, E)$, and a cost function on the edges $c : E \to \mathbb{Z}^+$.

Output: A minimum spanning tree $T = (V, E'), E' \subseteq E$.

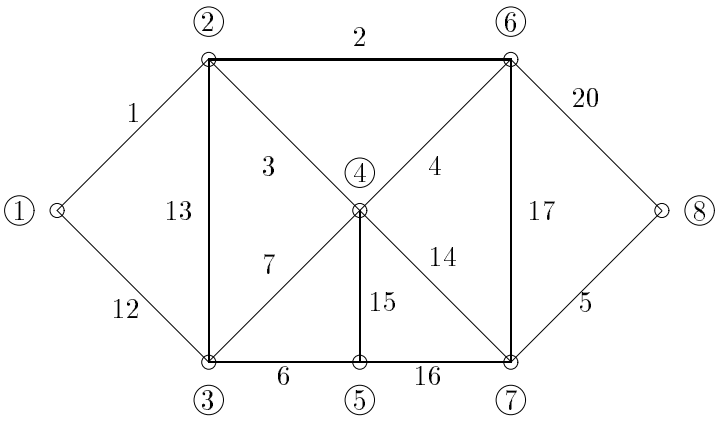
STEP 1. Choose any vertex $v \in V$ as the root; set $V' \leftarrow \{v\}$. For any $w \in V \backslash \{v\}$, set $l(w) \leftarrow c(\{v, w\})$; if the edge $\{v, w\}$ does not exist, then $c(\{v, w\}) \leftarrow \infty$. Set $E' \leftarrow \emptyset$.

STEP 2. Determine a vertex $w \in V \backslash V'$ with minimum label. Let $e = \{v, w\}$ be an edge that connects w with V' at cost $l(w)$. Add $\{v, w\}$ to E' and add w to V'.

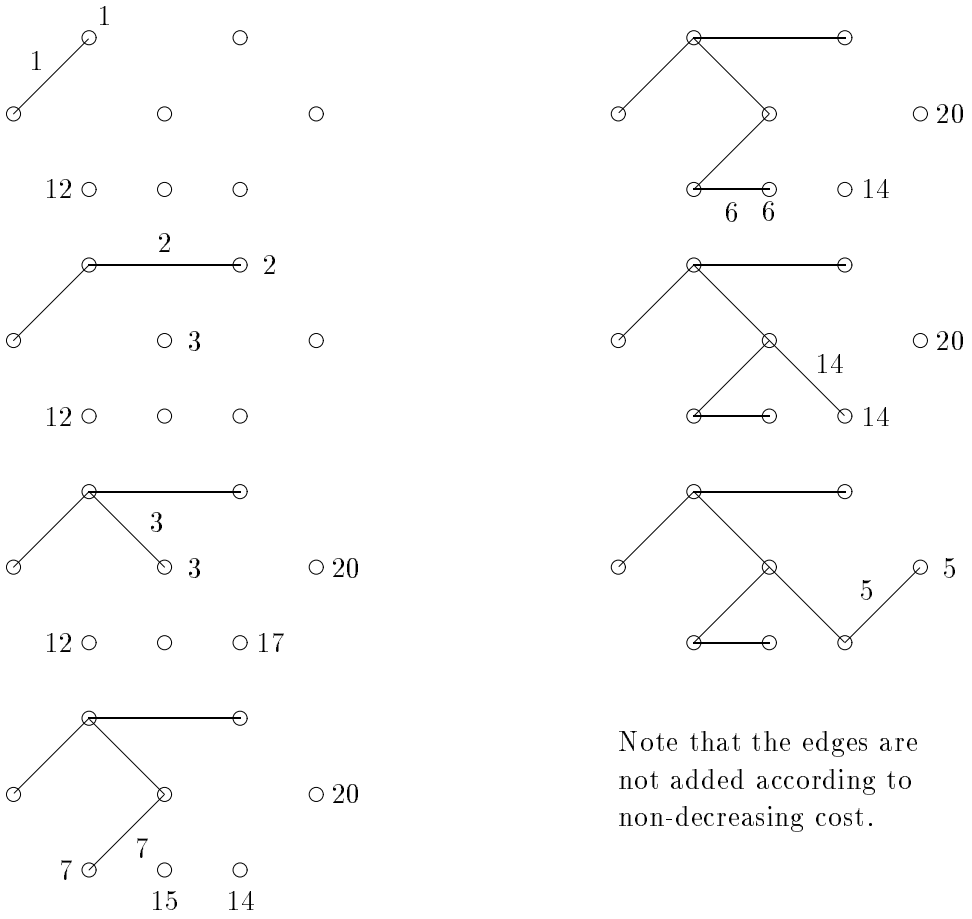
STEP 3. Update the labels $l(w')$ for all vertices $w' \notin V'$ for which the edge $\{w, w'\}$ exists: set $l(w') \leftarrow \min\{l(w'), c(\{w, w'\})\}$.

STEP 4. If $V' = V$, then stop; otherwise, go to STEP 2.

Example



iteration \ vertex	1	2	3	4	5	6	7	8
0		(1)	12	∞	∞	∞	∞	∞
1			12	3	∞	(2)	∞	∞
2			12	(3)	∞		17	20
3			(7)		15		14	20
4					(6)		14	20
5							(14)	20
6								(5)



Note that the edges are not added according to non-decreasing cost.

Algorithm MST-2 lends itself for an analysis of the number of elementary steps performed by the algorithm. Elementary operations are: addition, multiplication, comparison of two numbers, and addition and deletion of elements (here edges or vertices) to and from a set.

MTS-2 performs $n-1$ iterations. In each iteration, an edge (and a vertex) is added to the tree. We distinguish two elementary operations.

The first operation is that we have to determine the minimum label from among the set of labels of the vertices in $V \backslash V'$. This takes n comparisons in the worst case.

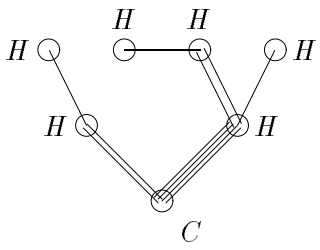
The second one is that, when w moves from $V \backslash V'$ to V', we have to compare the cost of each edge $\{w, w'\}(w' \in V \backslash V')$ with the label of w'. This takes at most $n-1$ comparisons. This operation is also performed at the initialization. Hence, the total number of elementary steps amounts to at most $n + (n-1)(n+n) \leq 2n^2$.

We use the following notation. If there exists a positive integer k such that for two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}$ we have that $f(n) \leq k|g(n)|$ for all $n \geq n_0$ for some $n_0 \in \mathbb{N}$, then we write $f(n) = O(g(n))$.

According to this notation, we have that the number of elementary steps that have to be performed by MST-2 to calculate a minimum spanning tree for a graph $G = (V, E)$ on n vertices amounts to $O(n^2)$; hence, the *running time* $r(n)$ of algorithm MST-2 amounts to is $O(n^2)$.

Section 1.4. The shortest path problem

Reconsider the problem of the previous section, where a telephone company must connect a set of houses to a central unit. If not the cost of digging the sleeves, but the cost of the cables is the most important cost component, then the problem becomes one of minimizing the total length of cable laid in the ground. Therefore, we are then asked to determine the shortest route between the central unit and each house. A typical solution is drawn below.



We can model this problem in terms of graphs in the following way. Consider a graph $G = (V, E)$ and a length function on the edges $c : E \to \mathbb{Z}^+$. Furthermore, there is a special vertex $s \in V$ from which all shortest paths to all other vertices must be determined.

The first algorithm to solve this problem was proposed by Dijkstra. He used the more general setting of the problem in digraphs. This is no serious limitation, however, since a graph can be 'directed' by replacing each edge $\{v, w\}$ with length $c(\{v, w\})$ with two arcs (v, w) and (w, v) that both have length $c(\{v, w\})$.

The idea behind the algorithm is to use labels for all vertices; these labels consist of upper bounds on the lengths of the shortest paths from s to each vertex. Each upper bound is decreased until its value equals the length of a shortest path.

The algorithm makes use of two additional sets V' and A'. The set V' contains all vertices v for which the shortest path from s to v has been determined; the set A' contains all arcs that are used in the shortest paths from s to the vertices v in V'.

Algorithm SP (Dijkstra)

Input: A digraph $G = (V, A)$ and a length function $c : A \to \mathbb{Z}^+$.
A special vertex $s \in V$.

Output: The lengths of all shortest paths from s to the other vertices: $l : V \backslash \{s\} \to \mathbb{Z}^+$.

STEP 1. For any vertex $v \in V \backslash \{s\}$, determine the initial label as the distance from v to s, that is, set $l(v) \leftarrow c((s, v))$; if the arc (s, v) does not exist, then set $l(v) \leftarrow \infty$. Set $V' \leftarrow \{s\}$ and $A' \leftarrow \emptyset$.

STEP 2. Determine a vertex $w \in V \backslash V'$ with minimum label. Let v be a vertex in V' such that $l(w) = l(v) + c((v, w))$, that is, there exists a path of length $l(w)$ from s to w that goes through v. Add (v, w) to A' and add w to V'.

STEP 3. Update the labels $l(w')$ for all vertices $w' \notin V'$ for which there exists an arc (w, w'): set $l(w') \leftarrow \min\{l(w'), l(w) + c((w, w'))\}$.

STEP 4. If $V' = V$, then stop; otherwise, go to STEP 2.

Theorem 1.5.
Algorithm SP calculates the lengths of the shortest paths from s to the other vertices correctly.

Proof.
We prove with induction that:

(a) For $x \in V'$: $l(x)$ is the length of the shortest path from s to x.

(b) For $y \in V \backslash V'$: $l(y) = \min_{x \in V'} \{l(x) + c((x,y)\}$.

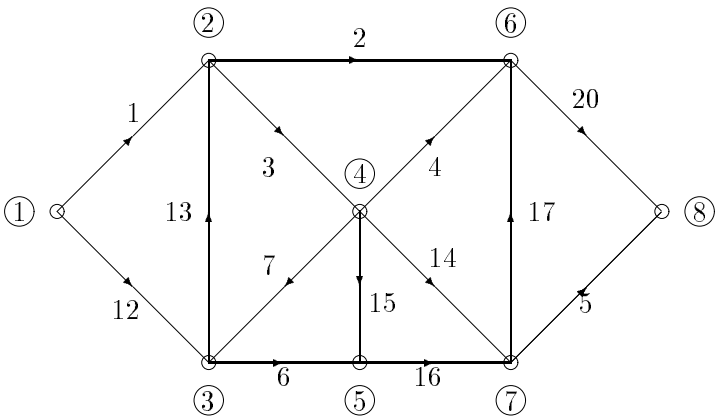
This is certainly true after the initialization. Now consider the iteration in which v is moved from $V \backslash V'$ to V'.
Proof of (a): Consider any path P from s to v. Let y be the first vertex of P in $V \backslash V'$, and let x be its direct predecessor. Hence, $P = (s, \ldots, x, y, \ldots, v)$, where $(s, \ldots, x)$ in V' and $y \notin V'$. Then the length of the path P is:

$$
\begin{aligned}
c(P) = c((s,\ldots,x)) &+ c((x,y)) + c((y,\ldots,v)) \\
&\geq c((s,\ldots,x)) + c((x,y)) && \text{(lengths of arcs are positive)} \\
&\geq l(x) + c((x,y)) && \text{(by hypothesis (a))} \\
&\geq l(y) && \text{(by hypothesis (b))} \\
&\geq l(v) && (v \text{ is chosen, so its label is minimal}).
\end{aligned}
$$

Hence, $l(v)$ is equal to the length of the shortest path from s to v, which implies that property (a) is restored at the end of the iteration.
In the second part of the iteration, the labels of the vertices in $V \backslash V'$ are updated so that (b) holds again, too. □

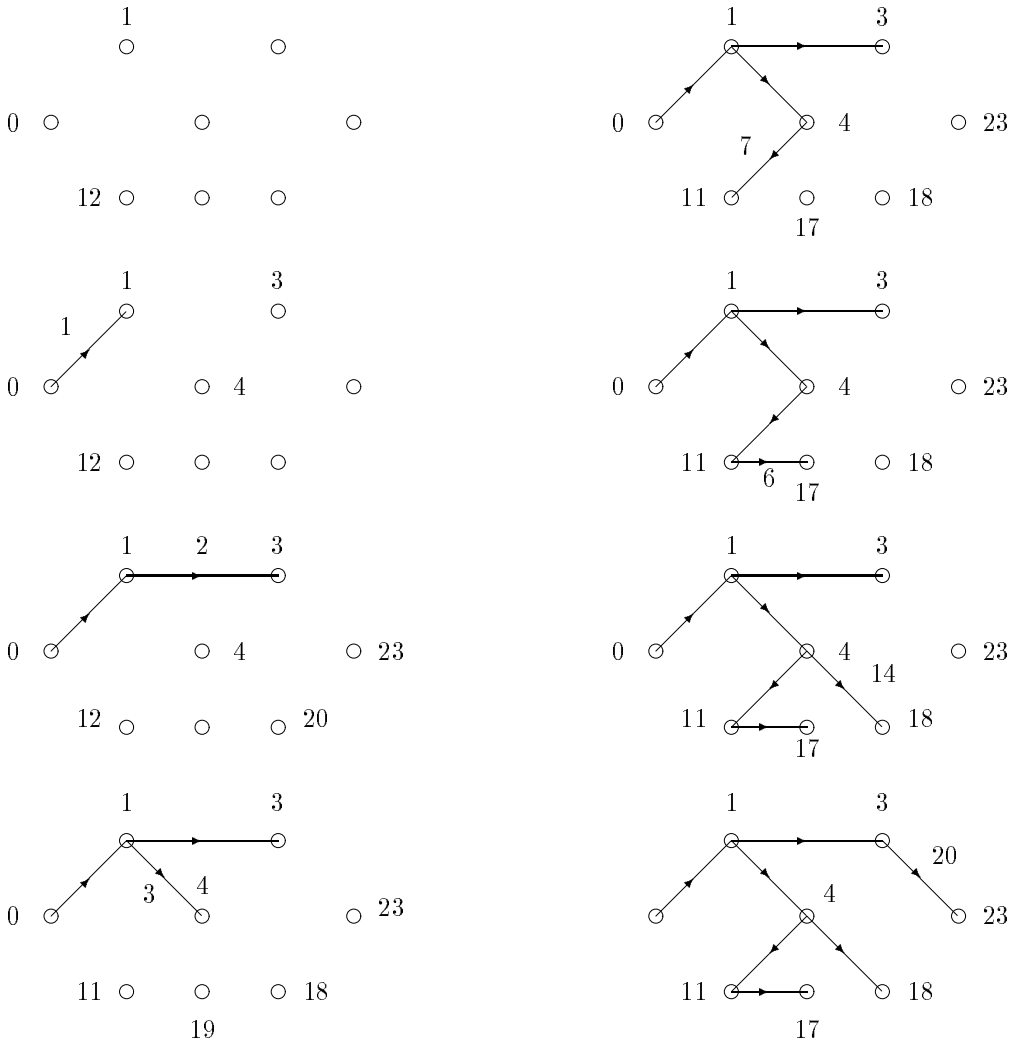
This algorithm resembles MST-2. By means of a proof similar to the one used for MST-2, we can show that Algorithm SP performs $O(n^2)$ elementary steps in case of a digraph on n vertices.
Below, an example of Algorithm SP in action is depicted; vertex 1 serves as s.

Example



Labels during the algorithm

iteration \ vertex	1	2	3	4	5	6	7	8
1		(1)	12	∞	∞	∞	∞	∞
2			12	4	∞	(3)	∞	∞
3			12	(4)	∞		20	23
4			(11)		19		18	23
5					(17)		18	23
6							(18)	23
7								(23)



Chapter 2. Matchings and flows

Section 2.1. Matching problems

The dance-class problem.

A group of boys and girls, with just as many boys as girls, want to dance together; hence, they have to be matched in couples. Each boy prefers to dance with some of the girls, and each girl prefers to dance with some of the boys. A boy and a girl will dance together if and only if both like each other. The question is whether the boys and girls can be combined to dancing pairs such that nobody has to stand aside.

In the corresponding graph problem two sets of vertices V_1 and V_2 are defined that denote the boys and the girls, respectively. An edge between a vertex $v_1 \in V_1$ and $v_2 \in V_2$ is defined if boy v_1 and girl v_2 like to dance with each other. We are asked to find a subset M of the edges such that each vertex is incident to exactly one edge.

A subset M of the edges of a graph $G = (V, E)$ is called a *matching* if each vertex is incident to at most one edge of M. The matching M is called *perfect* or *complete* if each vertex is incident to exactly one edge of M.

The graph by which the dance-class problem is modeled is a so-called *bipartite graph*. A graph $G = (V, E)$ is bipartite if the vertices can be partitioned into two sets V_1 and V_2 ($V_1 \cup V_2 = V, V_1 \cap V_2 = \emptyset$) such that each edge consists of one vertex from V_1 and one vertex from V_2. In general, the number of vertices in V_1 and V_2 is not equal.

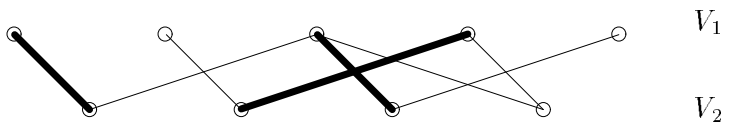


Figure 2.1: A matching in a bipartite graph.

In the following, edges in a matching will be drawn as fat edges. The next theorem provides a characterization of a bipartite graph.

Theorem 2.1

A graph $G = (V, E)$ is bipartite if and only if it does not contain a circuit of odd length.

The problem of finding a maximum cardinality matching or a perfect matching arises frequently in different mathematical and practical situations. Consider the following examples.

- Systems of Distinct Representatives (SDR's). Let S be a set of elements and let $S_1, \ldots, S_K$ be subsets of S. The question is whether there exist K different elements $s_1, \ldots, s_K$ in S such that $s_k \in S_k$ $(k = 1, \ldots, K)$.

Example $S = \{s_1, s_2, s_3, s_4, s_5\}$.

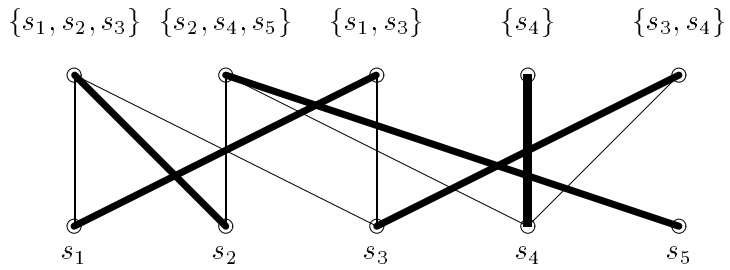


Figure 2.2: A systems of distinct representatives.

- Let $A = (a_{ij})$ be an $n \times n$ matrix with entries that are either zero or one. Let the rows be denoted by $r_1, \ldots, r_n$ and the columns by $c_1, \ldots, c_n$. The question is whether it is possible to combine the rows and columns to pairs such that each row and each column appears in exactly one pair, where each of the chosen pairs (r_i, c_j) must correspond to an entry $a_{ij} = 1$.

  A more practically oriented application is the following:
- A set of jobs in a company must be fulfilled by applicants. Each candidate has the skills to perform a certain subset of the jobs. Assign suitable candidates to as many jobs as possible.

Algorithms for finding maximum matchings try to extend a given matching M to a matching M' with a cardinality that is one higher, i.e., $|M'| = |M| + 1$. These improvement algorithms are based on the concept of alternating and augmenting paths. An *alternating* path in a graph $G = (V, E)$ with respect to a matching M is a path $(v_1, v_2, \ldots, v_K)$ where either $\{v_1, v_2\}, \{v_3, v_4\}, \ldots, \{v_{2k-1}, v_{2k}\}, \ldots$ are in M or $\{v_2, v_3\}, \{v_4, v_5\}, \ldots, \{v_{2k}, v_{2k+1}\}, \ldots$ are in M. Hence, edges from M are alternated with edges from $E \backslash M$. An *augmenting* path is an alternating path in which both endvertices are free, i.e., not incident to an edge in M. Such paths always have odd length, i.e., an odd number of edges and an even number of vertices. A *matched* or *exposed* vertex is a vertex incident to an edge in M.

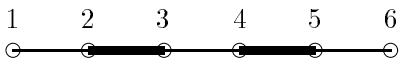


Figure 2.3: An augmenting path P with respect to M.

If there is an augmenting path P in G with respect to a matching M, then the number of edges in the matching can be increased by deleting the edges in $P \cap M$ from M and adding the edges from $P \backslash M$ to M.

Figure 2.4: The alternating path P with respect to $M' = (M \backslash (P \cap M)) \cup P \backslash M$.

Hence, we have that if there exists an augmenting path, then the matching can be increased. The converse is proved in the following theorem.

Theorem 2.2.
Let M be a matching in $G = (V, E)$. M can be extended if and only if there is an augmenting path in G with respect to M.

Proof.
We have shown above that if there exists an augmenting path P, then $M' = (M \cup P \backslash M) \backslash (P \cap M)$ is a matching with cardinality one higher than M.

Now suppose that there exists a matching M' with cardinality one higher than the cardinality of M; we have to indicate an augmenting path with respect to M. Consider the edges in the symmetric difference of M and M', which is defined as $M' \div M = (M' \backslash M) \cup (M \backslash M')$; this set contains the edges that are in M or M', but not in both. Let G' denote the graph $G' = (V, M' \div M)$. Since each vertex is incident to at most one edge in M and M', we have that all vertices in G' have degree 0,1, or 2. Since a vertex with degree equal to 2 is incident to one edge from M and one edge from M', we have that all connected components in G' are either alternating paths or alternating circuits. As the number of edges in M' is one higher than the number of edges in M, we have that there must be a path P containing more edges from M' than from M. This path P is alternating and both endpoints of P are not incident to an edge of M, which implies that P is augmenting with respect to M. □

The algorithm for finding a maximum matching can now be stated as follows.

Algorithm MM
Input: A graph $G = (V, E)$.

Output: A matching $M \subseteq E$.

STEP 1. $M = \emptyset$.

STEP 2. Find an augmenting path P. If no augmenting path is found, then stop.

STEP 3. Increase M to $(M \backslash M \cap P) \cup (P \backslash M)$; go to STEP 2.

A systematic search for augmenting paths is performed by a labeling algorithm. In general graphs it is fairly complicated; therefore, we restrict ourselves to bipartite graphs. The labeling algorithm proceeds as follows: We check for each vertex if it can be connected to an alternating path starting in a free vertex; if so, this vertex receives label Y (yes), otherwise, it receives label N (no). We proceed our labeling until either an alternating path P is discovered or until no more vertices receive label Y that were previously labeled N. The algorithm makes use of two additional sets S and T, which contain the vertices in V and W whose label changed from N to Y.

Algorithm APM

Input: A bipartite graph $G = (V \cup W, E)$ and a matching $M \subseteq E$.

Output: An augmenting path in G with respect to M, if one exists.

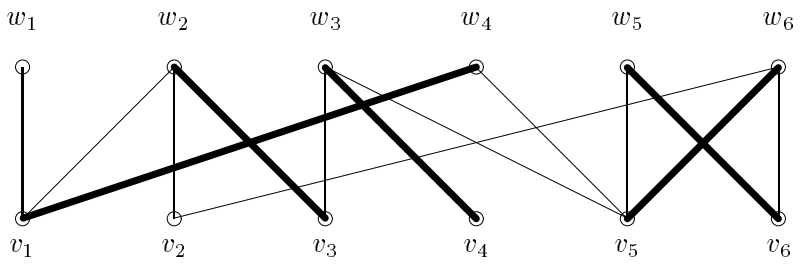
STEP 1. The vertices in V and W are labeled: if a vertex $v \in V$ is unmatched, then it gets label Y; otherwise, it gets label N. All vertices in W get label N. The sets S and T contain all vertices $v \in V$ and $w \in W$ with label Y, respectively; that is, S contains all the vertices in V that are unmatched, whereas $T \leftarrow \emptyset$.

STEP 2. Consider the vertices $v \in S$. All vertices $w \in W$ with label N that are connected to a vertex $v \in S$ through an edge that is not in M obtain label Y and are added to T. Set $S \leftarrow \emptyset$.

STEP 3. Consider the vertices $w \in T$. If any of these vertices w is unmatched, then an augmenting path has been found. Otherwise, all vertices $v \in V$ with label N that are connected to a vertex $w \in T$ through an edge in M obtain label Y and are added to S. Set $T \leftarrow \emptyset$.

STEP 4. If an augmenting path has been found or if $S = \emptyset$, stop. Otherwise, go to STEP 2.

Example.

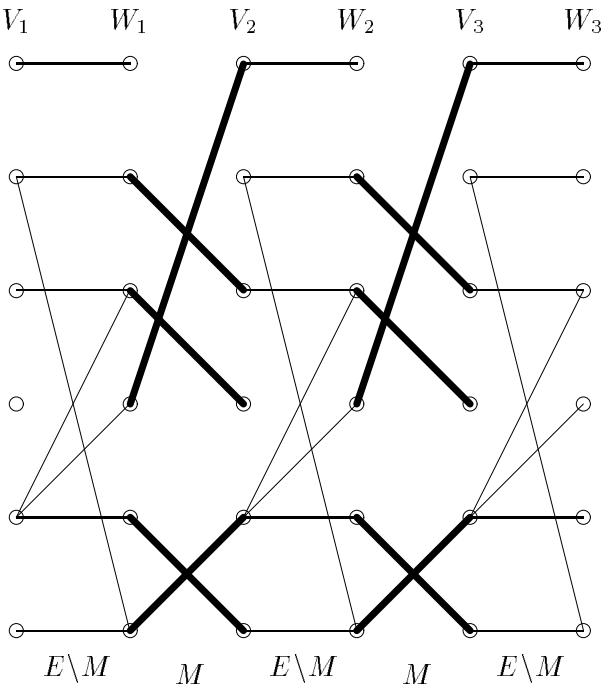


$E \backslash M = \{\{v_1, w_1\}, \{v_1, w_2\}, \{v_2, w_2\}, \{v_2, w_6\}, \{v_3, w_3\}, \{v_5, w_3\}, \{v_5, w_4\}, \{v_5, w_5\}, \{v_6, w_6\}\}$.
$M \quad = \{\{v_1, w_4\}, \{v_3, w_2\}, \{v_4, w_3\}, \{v_5, w_6\}, \{v_6, w_5\}\}$.

To illustrate the working of the labeling algorithm, the vertex set will be copied K times to $V_1, \ldots, V_K, W_1, \ldots, W_K$. The edges between copies V_k and W_k $(k = 1, \ldots, K)$ are those corresponding to $E \backslash M$. The edges between copies W_k, V_{k+1} $(k = 1, \ldots, K-1)$ are those corresponding to M.

In the initialization in Step 1, vertex v_2 gets label Y, whereas all other vertices get the label N: $S \leftarrow \{v_2\}$ and $T \leftarrow \emptyset$. In the first run through Step 2, the vertices w_2 and w_6 receive label Y, $T \leftarrow \{w_2, w_6\}$, and $S \leftarrow \emptyset$. In the first run through Step 3, the vertices v_3 and v_5 receive label Y, $S \leftarrow \{v_3, v_5\}$, and $T \leftarrow \emptyset$. In the second run through Step 2, the vertices w_3, w_4, and w_5 are labeled Y, $T \leftarrow \{w_3, w_4, w_5\}$, and $S \leftarrow \emptyset$. In the second run through Step 3, the vertices v_1, v_4, and v_6 are labeled Y: $S \leftarrow \{v_1, v_4, v_6\}$ and $T \leftarrow \emptyset$. In the third run through Step 2, the vertices w_1, w_2, and w_6 are labeled Y; only for w_1, the label changes. Therefore, $T \leftarrow \{w_1\}$ and $S \leftarrow \emptyset$. In the third run through Step 3, the existence of the augmenting path $\{v_2, w_6\}, \{w_6, v_5\}, \{v_5, w_4\}, \{w_4, v_1\}, \{v_1, w_1\}$ is signaled.

In the above description of algorithm APM, no attention has been paid to storing the augmenting path. This can be taken care of easily, however, by keeping a list for each vertex in the current sets S and T that denotes how this vertex is reached.



The labeling algorithm proceeds by labeling vertices in V_1, then in W_1 and V_1 etcetera. The newly labeled vertices and the corresponding edges are drawn in the following figure.

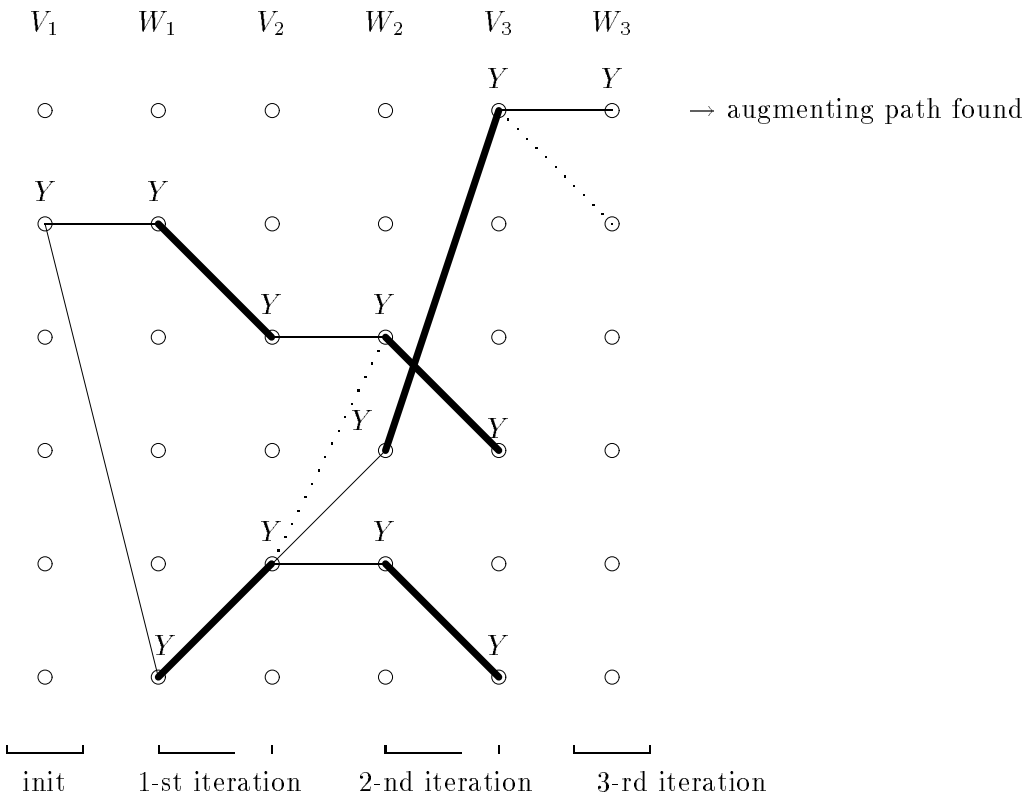
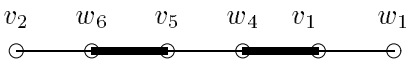


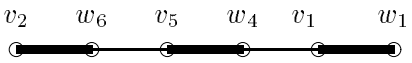
Figure 2.5.

The dotted lines concern edges that connect a vertex that has just been labeled Y to an vertex that already had been labeled Y. Clearly, the tree in Figure 2.5 consists of alternating paths only.

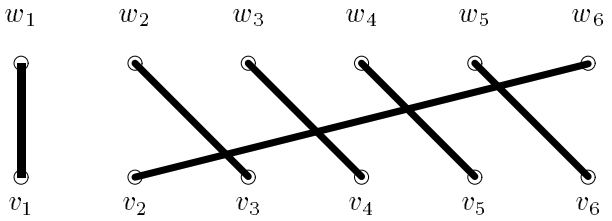
The augmenting path is the following one:



Hence, we can augment the corresponding part of the matching to:



This results in the final (perfect) matching:



Theorem 2.4
Algorithm APM finds an augmenting path, if one exists.

Proof.
Suppose that an augmenting path exists. Let $P = \{v_1, w_1, v_2, w_2, \ldots, w_K)$ be the shortest one. Initially, we have that $v_1 \in S$, and hence, it gets label Y. Now suppose that at the beginning of an iteration $v_1, w_1, \ldots, v_k$ have label Y and $w_k, v_{k+1}, \ldots, w_K$ have label N. Moreover, $v_k \in S$. In the first part of the iteration, w_k gets label Y and it enters T. Note that since P is alternating, we have that $\{v_k, w_k\} \in E \backslash M$. Furthermore, there is no vertex $w_l \in \{w_{k+1}, \ldots, w_K\}$ that has received label Y in the meantime, since this would imply that there exists a shorter augmenting path $(v_1, w_1, v_2, \ldots, v_k, w_l, \ldots, w_K)$. Since P is alternating, we have that $\{w_l, v_{l+1}\} \in M$, and hence, v_{k+1} is labeled Y in the second part of the iteration. Moreover, it enters S. Finally, the unmatched vertex w_K is labeled Y, and the path is detected. □

Algorithm MM consists of repeatedly applying algorithm APM. If the vertex sets V and W both contain n vertices, then APM can be applied at most n times to augment a given matching. In each call of algorithm APM, an edge $e = \{v, w\}$ is examined at most once: namely when w is labeled Y if $e \in M$, and when v is labeled Y if $e \notin M$. Hence, we have that the running time $r(n)$ of algorithm MM on a graph $G = (V \cup W, E)$ with $|V| = |W| = n$ amounts to $O(|V| \cdot |E|)$.

As a corollary of the augmenting path algorithm, we get the following existence theorem of a perfect matching in a bipartite graph.

Theorem 2.5 (Hall)
The graph $G = (V \cup W, E)$ with $|V| = |W|$ contains a perfect matching if and only if for each subset $V' \subseteq V$ we have that the subset $W' \subseteq W$ that contains all vertices adjacent to some vertex in V' is at least as large as V', i.e., $|W'| \geq |V'|$.

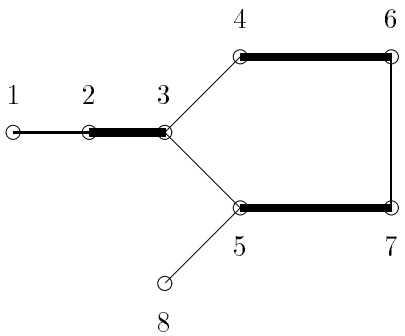
Proof.
If a perfect matching M exists, then we obviously have that for any subset $V' \subseteq V$ the set W' contains the vertices matched to vertices of V' in M, which implies that $|W'| \geq |V'|$.

Now suppose that there is no perfect matching; we will indicate a subset V' for which the condition on W' fails. Let M be a maximum cardinality matching. Run algorithm APM on M, and let V' contain the vertices in V that are labeled Y and W' the vertices in W labeled Y; since the algorithm has stopped, we have that the set W' contains exactly those vertices that are adjacent to at least one vertex in V'. Since M has maximum cardinality, no augmenting path has been found, which implies that none of the vertices $w \in W'$ is free. Since each vertex $w \in W'$ can be matched to a vertex $v \in V'$ only, we have that V' contains $|W'|$ vertices that are connected to a vertex in W' in M and at least one free vertex, as the set S initially is nonempty. Hence, $|V'| > |W'|$. □

Hall presented Theorem 2.5 in the context of SDR's: There exists a system of SDR's if and only if each union of k subsets contains at least k elements.

Theorem 2.5 provides a good characterization of the existence of perfect matchings. If a graph contains a perfect matching, then you can convince somebody by providing a perfect matching. If a graph does not contain a perfect matching, then you can convince somebody by providing a set V' and the corresponding set W' that contains all vertices that are adjacent to at least one of the vertices of V' with $|V'| > |W'|$.

The labeling algorithm APM can not be applied to find matchings in general graphs. It follows immediately from Theorem 2.1 that this must be due to the presence of odd circuits. An example in which algorithm APM fails to find an augmenting path, though there is one, is provided by the following graph.



Since there is no clarity as to which vertices belong to V and which to W, we suppose that $1 \in V$ and $8 \in W$. APM labels the vertices as follows. In the initialization in Step 1, vertex 1 gets label Y, whereas all other vertices get label N: $S \leftarrow \{1\}$ and $T \leftarrow \emptyset$. In the first run through Step 2, vertex 2 receives label Y, $T \leftarrow \{2\}$, and $S \leftarrow \emptyset$. In the first run through Step

3, vertex 3 receives label Y, $S \leftarrow \{3\}$, and $T \leftarrow \emptyset$. In the second run through Step 2, the vertices 4 and 5 are labeled Y, $T \leftarrow \{4,5\}$, and $S \leftarrow \emptyset$. In the second run through Step 3, the vertices 6 and 7 are labeled Y, $S \leftarrow \{6,7\}$, and $T \leftarrow \emptyset$. In the third run through Steps 2 and 3, no vertices are relabeled, which implies that $S \leftarrow \emptyset$, and the algorithm stops. Hence, the augmenting path (1,2,3,4,6,7,5,8) remains undiscovered.

Note that this problem would not have occurred if in the third run through Steps 2 and 3 we would have allowed the vertices 6 and 7 to be included in T; the only reason that they were not included was that they had already been labeled Y, but at the time of their relabeling, they were included in S then. This problem can be solved by specifying the labels of the node according to the distance of the unmatched nodes. A node v gets label O (odd), if an alternating path of odd length from a unmatched node to v is discovered. A node gets label E (even), if an alternating of even length from an unmatched node to v is discovered. Hence, nodes can get two labels: O and E. We leave it as an exercise to the reader to apply this revision of APM to the example.

The maximum cardinality matching problem can be generalized to a weighted version by attaching weights to the edges. Given a graph $G = (V, E)$ and a function $c : E \rightarrow \mathbb{Z}^+$, the problem then becomes: find a matching M of maximum weight; the weight $c(M)$ of the matching M is equal to $\sum_{e \in M} c(e)$.

If $G = (V \cup W, E)$ is bipartite, with $|V| = |W|$, then this problem is known as the assignment problem. Kuhn devised an algorithm to solve the assignment problem in 1955; it is known as the Hungarian method. Edmonds developed an algorithm for the problem on general graphs in 1965, the so-called blossom algorithm; this has been implemented to run in $O(|V|^3)$ time.

Note that the matching of maximum weight does not need to have maximum cardinality. If you want to determine the maximum weight matching from among the matchings with maximum cardinality, then you can still use the same machinery as for finding the matching with maximum weight, however: just add a huge constant to the weight of each edge.

A problem that uses the blossom algorithm as a subprogram is the Chinese postman problem, which is discussed in the next section.

Section 2.2. The Chinese postman problem

Consider the problem that the postman faces each rainy day: he has to deliver mail, and prefers to be home as soon as possible, that is, he wants to minimize the distance that he has to walk. This problem is modeled as a minimization problem in graphs as follows. Consider the streets in which mail has to be delivered as edges, and the intersections of these streets as vertices. The cost of each edge is equal to the length of the corresponding street. The postman wants to find a cycle of minimum length that traverses each edge at least once. Formally, this problem is stated as:

Chinese postman problem

Given a graph $G = (V, E)$ and a cost function $c : E \rightarrow \mathbb{Z}^+$, we are asked to find a minimum-weight set E' of edges to add to G so that the resulting graph $G' = (V, E \cup E')$ contains an *Eulerian cycle*, i.e., a cycle containing each edge of G' exactly once.

As already mentioned in Theorem 1.3, Euler has shown that a necessary and sufficient condition for a multigraph to possess an Eulerian cycle is that the multigraph is connected and that each vertex has *even* degree. Assuming the graph G to be connected, we only have to

take care of the odd-degree vertices.

By Proposition 1.1, the set V' of odd-degree vertices has even cardinality. Since the degree of all vertices in G' has to be even, we know that all vertices in V' have odd degree in the graph (V, E'), whereas all other vertices must have even degree in (V, E'). Moreover, (V, E') should not contain any circuits, since circuits have non-negative cost and can be deleted without changing the status (even/odd) of any vertex. Therefore, we must have that (V, E') consists of a set of paths connecting the odd-degree vertices. Of course, the paths between any pair of odd-degree vertices have minimum length among all paths connecting this pair of vertices.

The following five-phase algorithm solves the Chinese postman problem on a graph $G = (V, E)$ with cost function $c : E \to \mathbb{Z}^+$.

Phase 1: Find the odd-degree vertices V'.
Phase 2: Find the shortest paths between all pairs of vertices in V'.
Phase 3: Construct a complete graph H on V', where the weight of an edge $\{v, w\}$ is equal to the length of the shortest path between the vertices v and w.
Phase 4: Find a perfect matching in H with minimum weight.
Phase 5: For each edge in the matching of H, add the edges of the corresponding path to G.

Example

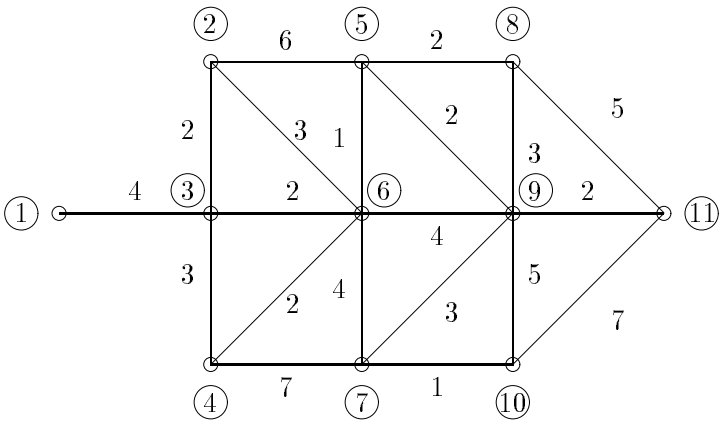
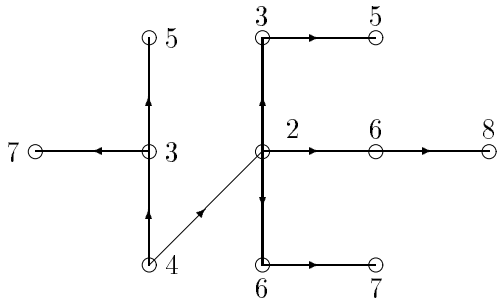


Figure 2.6.

Phase 1: The odd degree vertices are 1,2,4,8,10 and 11.
Phase 2: The shortest paths between all pairs of vertices in $\{1, 2, 4, 8, 10, 11\}$ are derived by applying algorithm SP six times, with root vertex s equal to 1,2,4,8,10 and 11, respectively.
For $s = 4$ we get the following shortest paths; the lengths are denoted by the given labels.

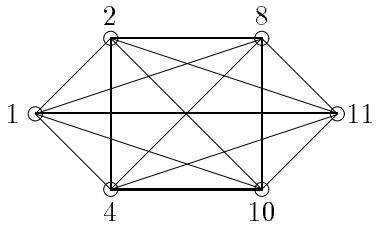


The complete set of shortest path lengths between the odd-degree vertices is given in the following table.

	1	2	4	8	10	11
1						
2	6					
4	7	5				
8	9	6	5			
10	11	8	7	7		
11	11	9	8	5	6	

Figure 2.7.

Phase 3: Find a minimum weight perfect matching in the following graph, the complete graph on the vertices $\{1, 2, 4, 8, 10, 11\}$. The weights of the edges are given in the table in Figure 2.7.



Phase 4:

Matching	Weight	
$\{1,2\},\{4,8\},\{10,11\}$	$6+5+6=17$	$\rightarrow$ minimum
$\{1,2\},\{4,10\},\{8,11\}$	$6+7+5=18$	
$\{1,2\},\{4,11\},\{8,10\}$	$6+8+7=21$	

In any other matching, the weight of the edge containing vertex 1 is at least 7, whereas the other two edges have weight at least 5. Hence, the weight of this matching will amount to at least 17. Therefore, the matching $\{\{1,2\},\{4,8\},\{10,11\}\}$ is optimal. The corresponding shortest paths are:

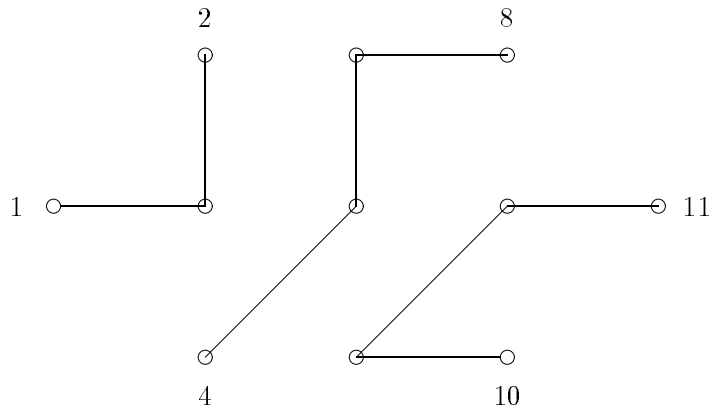
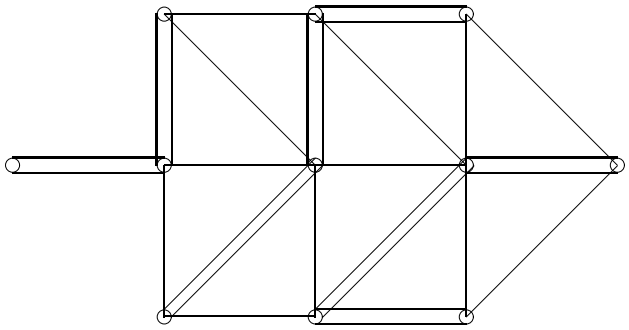


Figure 2.8.

Phase 5: The Eulerian graph contains the edges of the shortest paths of Figure 2.8 twice. Its weight is increased by 17.



The running time of our five-phase algorithm is determined as the maximum of the running times in the phases. Phase 1 takes $O(|V|)$ time, since all vertices are checked in constant time per vertex. Phase 2 takes $O(|V'|^3)$ time, since determining the shortest path from one vertex to all other vertices takes $O(|V'|^2)$ time. Phase 3 takes $O(|V'|^2)$ time, since the number of edges amounts to $|V'|(|V'|-1)/2$. From the previous subsection, it follows that Phase 4 takes $O(|V'|^3)$ time as well, while Phase 5 takes $O(|V'|^2)$ time at most. We do not know, however, what $|V'|$ is going to be; all we know is that $|V'| \leq |V|$. Hence, the running time amounts to $O(|V|^3)$.

Section 2.3. Maximum flows

Some catastrophe has happened in some far away place, and help is needed badly. Fortunately, all that is needed is available, but it is stored in some depot that is quite remote from the disaster area. We are asked to find a way to send as much help to the area as possible. Unfortunately, the time that an army of planes transported all goods to this place lies behind us, but we are allowed to make use of spare capacity in regular flights, trade services, etc. Obviously, in each transshipment point we can dump a lot of goods, but nobody is helped if it is not transported further to the disaster area. Hence, we must add the restriction that everything that is brought into a place that is not the disaster area should be transported out of this place. This problem can be modeled as a minimization problem in graphs. We need to introduce some notation first.

The pair (G, c) in which G is a digraph (V, A) and $c : A \to \mathbb{Z}^+$ is a capacity function on the arcs, is called a *network*. In the network two special vertices, a *source* s and a *sink* t are identified. A *flow* from s to t in (G, c) is a function $x : A \to \mathbb{Z}^+$ on the arcs that satisfies the following two properties.

Flow conservations constraints: For each node $v \in V \backslash \{s, t\}$ the amount of flow leaving v equals the amount of flow entering v:

$$\forall_{v \in V \backslash \{s,t\}} : \sum_{w:(v,w) \in A} x_{(v,w)} = \sum_{w:(w,v) \in A} x_{(w,v)} \ .$$

Capacity constraints: $\forall_{a \in A} : 0 \leq x_a \leq c_a$.

The *flow value* f is the net outflow from s, i.e., the amount of flow leaving s minus the amount of flow entering s. We only consider networks in which no arcs enter s. Hence, the flow value f then becomes:

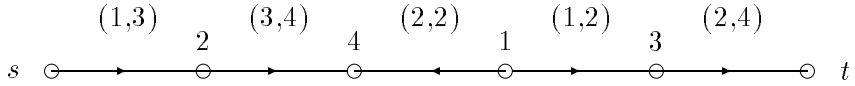
$$f = \sum_{w:(s,w) \in A} x_{(s,w)} - \sum_{w:(w,s) \in A} x_{(w,s)} = \sum_{w:(s,w) \in A} x_{(s,w)} \ .$$

Example

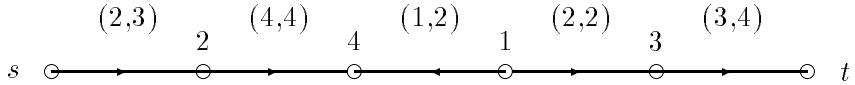
A network (G, c). A feasible flow in the network (G, c)

Figure 2.9.

A method for improving a given flow proceeds by finding paths from s to t on which the flow can be increased. Such paths are called *(flow) augmenting* paths. They differ from ordinary paths in a digraph in the sense that arcs do not need to have the right direction. Take for example the path $(s, 2, 4, 1, 3, t)$ in the graph of Figure 2.9.



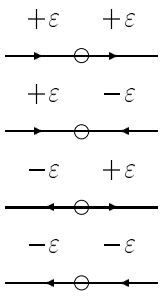
The pair (x_a, c_a) denotes the flow and the capacity of each arc a, respectively. We can increase the flow from s to t by increasing the flow over the *forward* edges (the edges in the right direction from s to t) and decreasing the flow over the *backward* edges (the edges in the opposite direction). Thus, the flow over the arcs $(s, 2), (2, 4), (1, 3)$ and $(3, t)$ is increased by one unit, and the flow over the arc $(1, 4)$ is decreased by one unit.



The flow on all other arcs remains equal. The outflow of s is increased by one unit, whereas the net outflow (outflow minus inflow) of the intermediate nodes is not changed and remains zero. Note that the inflow in t increases by one unit as well.

In general, an augmenting path from s to t contains forward edges, which carry flow below capacity, and backward edges, which carry a positive flow. The flow on forward arcs is increased and the flow on backward arcs is decreased with an amount ε.

The four situations for an intermediate vertex on an augmenting path are:



inflow and outflow increase by ε units.

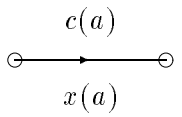
inflow and outflow remain equal.

inflow and outflow remain equal.

inflow and outflow decrease by ε units.

In any of the four cases, the flow conservation constraints remain valid. The value ε is determined by the capacity constraints. For each edge on the path an upper bound on ε can be derived as follows:

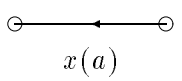
On a forward arc the flow can be



$\varepsilon \leq c(a) - x(a)$

increased at most $c(a) - x(a)$ units

On a backward are the flow can be



$\varepsilon \leq x(a)$

decreased at most $x(a)$ units

The value for ε is determined as the smallest upper bound from among the arcs on the augmenting path.

Augmenting paths are found systematically by the augmenting path algorithm APF; this is a labeling algorithm in which a vertex v is labeled Y whenever there exists an augmenting path from s to v. Algorithm APF makes use of two additional sets S and T. The set S contains all vertices with a label that has been changed from N to Y in the *current* iteration; that is, S contains all vertices v for which an augmenting path from s to v has been discovered in the last run. The set T is introduced only for clarity of exposition; it contains all vertices with a label that has been changed from N to Y in the *previous* iteration.

Algorithm APF

Input: A feasible flow x with value f on a network (G, c).

Output: A path from s to t on which the flow can be augmented.

STEP 1. The vertices are labeled: s gets label Y, whereas all other vertices get label N. Set $S \leftarrow \{s\}$.

STEP 2. Set $T \leftarrow S$ and set $S \leftarrow \emptyset$. Consider the vertices $v \in T$. All vertices $w \in V$ with label N that are connected to a vertex $v \in T$ by an arc (v, w) with $x_{(v,w)} < c_{(v,w)}$ or by an arc (w, v) with $x_{(w,v)} > 0$ get label Y and are added to S.

STEP 3. If t has received label Y, then an augmenting path P has been found; stop. If $S = \emptyset$, then there exists no augmenting path; stop. If none of these two cases applies, then go to STEP 2.

In the above description of algorithm APF, no attention has been paid to storing the augmenting path. This can be taken care of easily, however, by keeping a list for each vertex in the current set S that denotes how this vertex is reached.

Suppose that an augmenting path P has been discovered. Let ε_f denote the minimum value over all forward arcs $a \in P$ of $c(a) - x(a)$, and let ε_b denote the minimum value over all backward arcs $a \in P$ of $x(a)$. We augment the flow by $\varepsilon = \min\{\varepsilon_f, \varepsilon_b\}$ units.

The overall algorithm to determine a maximum flow proceeds by using the augmenting path algorithm until no augmenting path is found anymore.

Algorithm MF

Input: A network (G, c) and a source s and sink t.

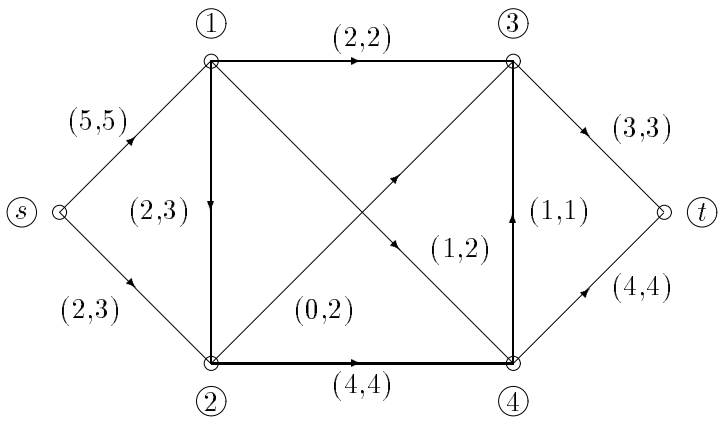
Output: A maximum flow from s to t in (G, c).

STEP 1. Start with any feasible flow, possibly the zero flow.

STEP 2. Find an augmenting path P with algorithm APF. If no augmenting path is found, then stop.

STEP 3. Augment the flow over P by as much as possible; go to STEP 2.

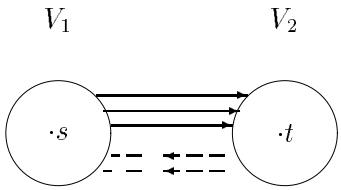
In the example of Figure 2.9, the flow that results after increasing the flow over the path $(s, 2, 4, 1, 3, t)$ by one unit is given in the following network.



In the network above, no augmenting path can be found. Algorithm APF hands labels Y to the vertices s, 2, 1, and 4 and labels N to the vertices 3 and t.

Bounds for the maximum flow value.
Clearly, a given feasible flow provides a lower bound on the maximum flow. To derive an upper bound on the flow, the so-called (s,t)-*cuts* have been introduced. An (s,t)-cut is a partition of the vertex set into two sets V_1 and V_2, where $s \in V_1$ and $t \in V_2$. The *capacity* $c(V_1, V_2)$ of the (s,t)-cut (V_1, V_2) is the total capacity of all arcs leaving V_1 and entering V_2.

$$c(V_1, V_2) = \sum_{\substack{(v,w):\ v \in V_1 \\ w \in V_2}} c_{(v,w)}$$



The capacity of the cut (V_1, V_2) is an upper bound on the value f of any flow x, as is shown below.

$$f = \sum_{(s,v) \in A} x_{(s,v)}$$

$$\overset{(0)}{=} \sum_{(s,v) \in A} x_{(s,v)} + \sum_{v \in V_1 \setminus \{s\}} \left\{ \sum_{w:(v,w) \in A} x_{(v,w)} - \sum_{w:(w,v) \in A} x_{(w,v)} \right\}$$

$$\overset{(1)}{=} \sum_{v \in V_1} \sum_{w:(v,w)\in A} x_{(v,w)} - \sum_{v \in V_1} \sum_{w:(w,v)\in A} x_{(w,v)}$$

$$\overset{(2)}{=} \sum_{\substack{(v,w)\in A:\\ v\in V_1\\ w\in V_2}} x_{(v,w)} - \sum_{\substack{(v,w)\in A:\\ v\in V_2\\ w\in V_1}} x_{(v,w)}$$

$$\overset{(3)}{\leq} \sum_{\substack{(v,w)\in A:\\ v\in V_1\\ w\in V_2}} x_{(v,w)}$$

$$\overset{(4)}{\leq} \sum_{\substack{(v,w)\in A:\\ v\in V_1\\ w\in V_2}} c_{(v,w)} = c(V_1, V_2).$$

(0): The flow conservation constraints.

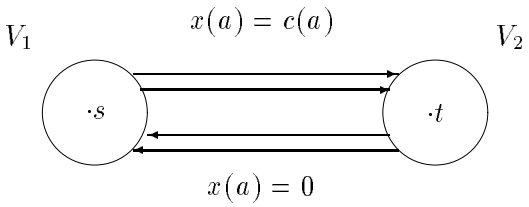
(1): Follows from rearranging the terms. Note that we assumed that no arcs enter s.

(2): Follows from removing the flow through the arcs (v, w) with $v, w \in V_1$; a flow through such an arc contributes $x_{(v,w)}$ to both terms.

(3): Flows are non-negative.

(4): The flow through any arc does not exceed its capacity.

Note that $f = c(V_1, V_2)$ if equality holds for (3) and (4). Inequality (3) becomes an equality if the arcs from V_2 to V_1 do not contain flow. Inequality (4) becomes an equality if all arcs from V_1 to V_2 contain flow at full capacity. This situation is depicted below.



Theorem 2.10. (Ford & Fulkerson) Maxflow = mincut.
If no augmenting path can be found, then there exists an (s, t)-cut with capacity equal to the value of the current flow.

Proof.
Consider the labeling of the vertices after Algorithm APF has stopped. Since no augmenting

path was determined, we have that t is labeled N. Define V_1 as the set of vertices v with label Y. Note that $s \in V_1$ and $t \in V_2$, which implies that this partition forms an (s,t)-cut.

Consider any forward arc in the (s,t)-cut, that is, an arc (v,w) with $v \in V_1$ and $w \in V_2$. Since w is labeled N, we must have that this arc carries flow up to its capacity. Similarly, we know that every backward arc in the (s,t)-cut carries no flow at all. These are exactly the conditions that make (3) and (4) equalities, which was to be proved. □

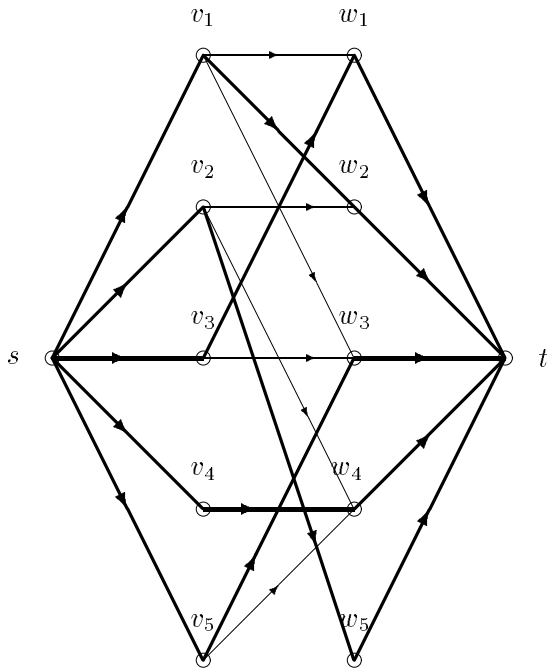
Theorem 2.10 gives a good characterization of maximum flows, i.e., for a given flow one can prove optimality *and* non-optimality by a simple argument: An optimal flow is shown to be optimal by indicating a cut with the same value, whereas a non-optimal flow is shown to be non-optimal by increasing the value of the flow, i.e., by indicating an augmenting path.

Notes

In the literature the 'incremental graph' is often used to find an augmenting path. For a given flow x from s to t in the network (G,c), each arc $a = (v,w)$ with flow $x(a)$ is replaced by two arcs as follows. If $c(a) - x(a) > 0$, then we introduce the arc (v,w) with capacity $c(a) - x(a)$; if $x(a) > 0$, then we introduce the arc (w,v) with capacity $x(a)$. This is depicted below.

In the incremental graph, we then just have to find a path from s to t. The advantage of using the incremental graph in this respect is that the algorithm gets well-organized; the disadvantage is that in the determination of the incremental graph every arc is evaluated, while it needs not to be considered in the determination of the augmenting path.

Maximum matching versus maximum flow.
We can solve the problem of finding a matching with maximum cardinality in case of a bipartite graph as a special case of the maximum flow algorithm, as is illustrated by the following example. The edges of $G(V_1 \cup V_2, E)$ are directed from V_1 to V_2. We add a source s with arcs (s, v) $(v \in V_1)$ and a sink t with arcs (w, t) $(w \in V_2)$. All arcs get capacity 1.



The flow problem corresponding to the system
of distinct representatives of Figure 2.2.

The fat arcs contain a flow of value one. The fat arcs connecting vertices from V_1 to V_2 correspond to edges in G that belong to the matching. Both labeling algorithms APF and APM label the vertices $\{v_3, v_4, v_5, w_2, w_5\}$. The flow can be augmented if the matching can be augmented, and vice versa.

Complexity
If the capacities of the arcs are finite, then so is the algorithm. Each (s, t)-cut has finite capacity, and in each iteration, that is, each time APF is called on, the flow is augmented by an integral number of units. Hence, the number of calls on APF is bounded from above by the capacity of the minimum cut.

The following example shows that it may take a while to find the optimal flow if the augmenting paths are not chosen carefully.

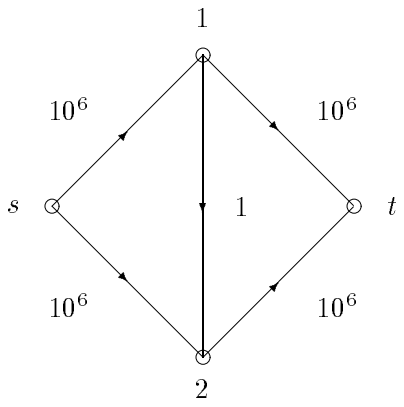


Figure 2.10.

The numbers denote capacities. We can take the paths $(s, 1, 2, t)$ and $(s, 2, 1, t)$ alternately 10^6 times. Each time, the flow is augmented by one unit. We could have taken the paths $(s, 1, t)$ and $(s, 2, t)$ one time each, instead. It seems better to take short paths or paths in which the flow can be augmented by as much as possible.

If capacities are allowed to be arbitrary reals, one can even provide examples where the flow does not even converge to the optimum, if unlucky choices for the augmenting paths are made.

From the example, it is clear that algorithm MF may have to make many calls on subalgorithm APF, if unlucky choices are made. In fact the number of calls can be equal to the capacity of a minimum cut, like in the example. An upper bound on the capacity of a minimum cut is the capacity of the cut $(\{s\}, V \backslash \{s\})$, which amounts to at most $n \cdot c$, where c is the maximum capacity of the arcs. Since the number of elementary steps of algorithm APF is $O(m)$, the total running time is $O(n \cdot m \cdot c)$. This does not seem to be really efficient as the example shows. We will now formalize what we mean with efficient.

First, observe that we cannot just look at the time needed by a computer to run the algorithm, since this is instance-dependent. We will only call an algorithm efficient if it is efficient for *each possible instance*; hence, we have to consider the *running time* of the algorithm.

An algorithm is defined to be *efficient* or *polynomial* if the number of elementary steps $r(n)$, that is, the running time, is bounded by a polynomial in the size of the input $f(n)$. More precisely, if $r(n) = O(p(f(n)))$ where p is a polynomial. This definition depends on the 'size of the input', which is measured in the number of bits needed to encode the instance.

If we return to the maximum flow problem, we see that any instance of this problem consists of a digraph $G = (V, A)$ and a capacity function $c : A \to \mathbb{Z}^+$. We need $O(n)$ bits to encode the vertices V and $O(m)$ bits to encode the arcs A. To encode the capacity c of an arc, we need $O(\log c)$ bits, and hence, we need $O(m \log c)$ bits to encode the entire capacity function. Since $n \leq m \leq m \log c$, the number of bits necessary to encode an instance of the maximum flow problem amounts to $O(m \log c)$. We see that the running time is not polynomial in the input size: c grows exponentially when $\log c$ grows linearly.

Although the maximum flow algorithm is not efficient in the way it is stated above, it can be made to run in polynomial time, if the proper choices for augmenting paths are made.

Chapter 3. Linear Programming

Linear programming is a tool for solving certain optimization problems. In 1947, George Dantzig developed a fast algorithm, the simplex algorithm, for solving linear programming problems. Since the development of the simplex algorithm, linear programming has been used to solve optimization problems in industries as diverse as banking, forestry, petroleum, and trucking.

Section 3.1. The Diet Problem

The next section has been adapted from the book: *Linear Programming* by V. CHVATAL (1983), printed by W.H. Freeman and Company, New York / San Francisco.

We introduce linear programming problems by means of an example. Consider the following diet problem.

Polly wonders how much money she must spend on food in order to get all the energy (2000 kcal), protein (55 g), and calcium (800 mg) that she needs every day. (For iron and vitamins, she will depend on pills. Nutritionists would disapprove, but our example ought to be simple.) She chooses six foods that seem to be cheap sources of the nutrients; her data are collected in Table 1.1.

Food	Serving size	Energy	Protein	Calcium	Price per serving
Oatmeal	28 g	110	4	2	3
Chicken	100 g	205	32	12	24
Eggs	2 large	160	13	54	13
Whole milk	237 cc	160	8	285	9
Cherry pie	170 g	420	4	22	20
Pork with beans	260 g	260	14	80	19

Table 1.1 Nutritive value per serving.

Then she begins to think about her menu. For example, 10 servings of pork with beans would take care of all her needs for only 1.90 per day. On the other hand, 10 servings of pork with beans is a lot of pork with beans; she would not be able to stomach more than 2 servings a day. She decides to impose servings-per-day limits on all six foods:

Oatmeal	at most 4 servings per day.
Chicken	at most 3 servings per day.
Eggs	at most 2 servings per day.
Whole milk	at most 8 servings per day.
Cherry pie	at most 2 servings per day.
Pork with beans	at most 2 servings per day.

Now, another look at her data shows Polly that 8 servings of milk and 2 servings of cherry pie every day will satisfy the requirements nicely and at a cost of only 1.12. In fact, she could cut down a little on the pie or milk or perhaps try a different combination. But so many combinations seem promising that one could go on and on, looking for the best one. Trial and error is not particularly helpful here. To be systematic, we may speculate about some

as of yet unspecified menu consisting of x_1 servings of oatmeal, x_2 servings of chicken, x_3 servings of eggs, and so on. In order to stay below the upper limits, the menu must satisfy:

$$\begin{aligned}
0 \le x_1 &\le 4 \\
0 \le x_2 &\le 3 \\
0 \le x_3 &\le 2 \\
0 \le x_4 &\le 8 \\
0 \le x_5 &\le 2 \\
0 \le x_6 &\le 2.
\end{aligned}$$

And of course, there are the requirements for energy, protein, and calcium; they lead to the inequalities

$$\begin{aligned}
110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 &\ge 2000 \\
4x_1 + 32x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 &\ge 55 \\
2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 &\ge 800.
\end{aligned}$$

If some numbers $x_1, x_2, \ldots, x_6$ satisfy all these inequalities, then they describe a satisfactory menu; such a menu will cost, in cents per day

$$3x_1 + 24x_2 + 13x_3 + 9x_4 + 20x_5 + 19x_6.$$

In designing the most economical menu, Polly wants to find numbers $x_1, x_2, \ldots, x_6$ that satisfy all the constraints and makes the cost as small as possible. As a mathematician would put it

$$\text{minimize } x_1 + 24x_2 + 13x_3 + 9x_4 + 20x_5 + 19x_6$$

subject to

$$\begin{aligned}
110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 &\ge 2000 \\
4x_1 + 32x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 &\ge 55 \\
2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 &\ge 800
\end{aligned}$$

$$\begin{aligned}
0 \le x_1 &\le 4 \\
0 \le x_2 &\le 3 \\
0 \le x_3 &\le 2 \\
0 \le x_4 &\le 8 \\
0 \le x_5 &\le 2 \\
0 \le x_6 &\le 2.
\end{aligned}$$

Problems of this kind are called linear programming problems.

Section 3.2. Easily solvable linear programs

In general, a linear programming problem is the problem of maximizing (or minimizing) a linear function subject to a finite number of linear constraints, that is,

$$\max \sum_{1 \leq j \leq n} c_j x_j$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i = 1, \ldots, N_1)$$

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad (i = N_1 + 1, \ldots, N_2)$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad (i = N_2 + 1, \ldots, N_3).$$

As indicated in the introduction, the simplex method solves linear programming problems fast. There are, however, two types of linear programming problems that can be solved without having to resort to the simplex method.

The first type of easily solvable linear programming problem concerns a linear programming problem in which there is only one constraint that involves more than one variable. Consider for instance the following *continuous knapsack problem.*

$$\max \sum_{1 \leq j \leq n} c_j x_j$$

subject to

$$\sum_{1 \leq j \leq n} a_j x_j \leq b$$

$$0 \leq x_j \leq 1 \quad (j = 1, \ldots, n).$$

The knapsack problem refers to a situation in which there are n different items that can be put into a knapsack with capacity b; each item j has a certain weight a_j associated with it as well as a value c_j. The problem is to determine how much of each item to place in the knapsack in order to maximize the total value subject to the constraint that the total weight should not exceed the capacity b of the knapsack.

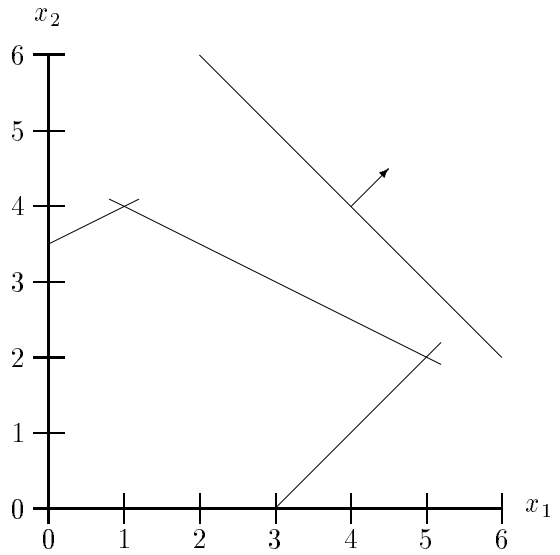
The problem is solved by sorting the items in order of decreasing ratios c_j/a_j and putting them in the knapsack in this order, always taking as much as possible, until the weight limit is reached. The knapsack problem is discussed in full detail in Chapter 4.

The second type of easily solvable linear programming problem concerns a linear programming problem with only two variables. In that case, the problem can be solved graphically. The set of feasible solutions is a polyhedron defined by the linear inequalities. Each feasible solution is on a profit line, that is, a set of feasible solution with the same solution value. There are many such profit lines, all parallel to each other. Using these profit lines it is easy find an optimal solution. Consider for instance the linear program

$$\max\{x_1 + x_2\}$$

subject to

$$\begin{aligned} -x_1 + 2x_2 &\leq 14 \\ x_1 + 2x_2 &\leq 18 \\ x_1 - x_2 &\leq 6 \\ x_1, x_2 &\geq 0. \end{aligned}$$



Section 3.3. Special cases of linear programs

So far, we have not addressed the following questions:

- Does there always exist a solution?
- If the solution space is not empty, does there exist a unique optimal solution?

Infeasible linear programs

It is possible that the feasible region of a linear program, that is, the set of points satisfying all constraints, is empty. Since the optimal solution to a linear program is the best point in the feasible region, such a linear program has no optimal solution. We say that the linear program is infeasible.

Consider for example the following linear program

$$\max\ 4x_1 + 2x_2$$

subject to

$$\begin{aligned} 3x_1 + 2x_2 &\leq 12 \\ x_1 &\geq 3 \\ x_2 &\geq 2. \end{aligned}$$

The feasible region of this linear program is empty: the combination of $x_1 \geq 3$ and $x_2 \geq 2$ implies that $3x_1 + 2x_2 \geq 13$, which is in conflict with the constraint $3x_1 + 2x_2 \leq 12$.

Unbounded linear programs

For a maximization problem, an unbounded linear program occurs if it is possible to find points in the feasible region with arbitrarily large objective function values. Analogously, for a minimization problem, an unbounded linear program occurs if it is possible to find points in the feasible region with arbitrarily small objective function values.

Consider for example the following linear program

$$\max\ 2x_1 - x_2$$

subject to

$$\begin{aligned} x_1 - x_2 &\leq 1 \\ 2x_1 + x_2 &\geq 8 \\ x_1, x_2 &\geq 0. \end{aligned}$$

Starting with a feasible solution, $x_1 = 0$ and $x_2 = 8$ for example, we are always able to increase the objective function by one unit by simply increasing x_1 and x_2 by one unit.

Alternative or multiple optimal solutions
It is possible for a linear program to have multiple or alternative optimal solutions. This is indicated by the fact that the vector of objective function coefficients is a positive multiple of one or more of the vectors of constraint coefficients.

Consider for example the following linear program

$$\max\ 3x_1 + 2x_2$$

subject to

$$\begin{aligned} x_1 + x_2 &\leq 5 \\ 3x_1 + 2x_2 &\leq 12 \\ x_1, x_2 &\geq 0. \end{aligned}$$

Each point (x_1, x_2) with $3x_1 + 2x_2 = 12$ that satisfies the other constraints is an optimal solution for this linear program.

Section 3.4. An example of the simplex algorithm

Consider any feasible bounded linear program with two variables. The constraints of the linear program define a polyhedron that contains all feasible solutions to this linear program. Suppose that the optimal solution to the linear program does not correspond to a vertex of the polyhedron. If the optimal solution lies within the interior of the polyhedron, then we can increase or decrease the value of x_1 without decreasing the value of the objection function until we hit the boundary of the polyhedron, that is, the new optimal solution is on an edge of the polyhedron. Similarly, we can move along this edge without decreasing the value of the objective function until we end in a vertex of the polyhedron. Hence, we have that, in case of a feasible bounded linear problem with two variables, we only have to search over all edges of the polyhedron to find an optimal solution. This can be generalized to the case of a feasible bounded linear program with n variables.

Theorem 3.1.
For any feasible bounded linear program there exists an optimal solution that corresponds to a vertex of the polyhedron that describes the set of feasible solutions. □

The simplex algorithm heavily depends on Theorem 3.1. It proceeds by hopping from one vertex of the polyhedron to another, in such a way that the value of the objective function never decreases. Before describing all ins and outs of the simplex algorithm, we show the iterative step through a simple example. Consider the following linear program

$$\max \; z = 2x_1 - 3x_2 + 4x_3$$

subject to

$$\begin{aligned} x_1 - x_2 + 2x_3 &\leq 7 \\ 2x_1 + x_2 + 3x_3 &\leq 22 \\ x_1 - x_2 + x_3 &\leq 2 \\ x_1, x_2, x_3 &\geq 0. \end{aligned}$$

It is not too hard to deal with the nonnegativity inequalities $x_i \geq 0$ $(i = 1, 2, 3)$; it it is the other inequalities that make the situation unclear. Therefore, we rewrite these inequalities by adding *slack* variables; the inequality $x_1 - x_2 + 2x_3 \leq 7$ is rewritten as $x_4 \geq 0$, where x_4 is defined as $x_4 = 7 - x_1 + x_2 - 2x_3$. In this way, we rewrite the whole linear program as a set of nonnegativity constraints together with a set of equalities. This yields the following linear program

$$\max \; z = 2x_1 - 3x_2 + 4x_3$$

subject to

$$\begin{aligned} x_4 &= 7 - x_1 + x_2 - 2x_3 \\ x_5 &= 22 - 2x_1 - x_2 - 3x_3 \\ x_6 &= 2 - x_1 + x_2 - x_3 \\ x_1, x_2, x_3, x_4, x_5, x_6 &\geq 0. \end{aligned}$$

The essential information in the linear program above consists of the objective function and the set of equalities; we will just write these down, while keeping in mind that z has to be maximized and that the values of the variables should be nonnegative. The compressed version of a linear program is called a *dictionary*. The dictionary in our example reads:

$$\begin{aligned} x_4 &= 7 - x_1 + x_2 - 2x_3 \\ x_5 &= 22 - 2x_1 - x_2 - 3x_3 \\ x_6 &= 2 - x_1 + x_2 - x_3 \\ z &= 2x_1 - 3x_2 + 4x_3. \end{aligned}$$

Almost everything is ready now to put the simplex method to work on this example. All we need now is an initial vertex of the polyhedron. From geometry, we know that a vertex

of a polyhedron that is defined as a set of m equalities in $(n+m)$ variables has the form that there are at least n variables with value equal to zero; the values of the other variables should be positive to make it a feasible solution to the linear program. Hence, we have that $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 0, 0, 7, 22, 2)$ serves as an initial solution. The simplex algorithm hops from one vertex of the polyhedron to another. Since we have just stated that in a vertex of the polyhedron at most m variables have positive value, we should increase the value of one of the zero-valued variables (these variables are called the *non-basic variables*) and decrease the value of one of the variables with positive value (these variables are called the *basic variables*) to zero. The question is of course: which non-basic variable should replace which basic variable?

As to the choice of the non-basic variable, it should be such that the value of the objective function does not decrease. The objective function is stated in terms of non-basic variables only, so we can make sure that the value of the objective function does not decrease by selecting a non-basic variable whose coefficient in the objective function is positive. In our example, this rules out x_2; let us choose x_1. It is easy to compute by how much we should increase x_1: it is the smallest value that makes the value of one of the basic variables equal to zero. Since x_2 and x_3 remain equal to zero, we have that $x_4 = 7 - x_1$, $x_5 = 22 - 2x_1$, and $x_6 = 2 - x_1$. This yields that our new vertex, which we from now on call a *basic feasible solution* will be $(x_1, x_2, x_3, x_4, x_5, x_6) = (2, 0, 0, 5, 18, 0)$; the corresponding value of the objective function amounts to 4.

If we take a closer look at the above step, then we see that we could easily select a non-basic variable whose increase did not decrease the objective function, because the objective function was stated in terms of non-basic variables only. Similarly, we could easily compute the new basic feasible solution, because all basic variables were expressed in terms of non-basic variables only. Hence, in order to conduct the next step, we rewrite the dictionary such that the above conditions are satisfied again. It is easy to express x_1 in terms of the non-basic variables: from the relation $x_6 = 2 - x_1 + x_2 - x_3$ it follows immediately that $x_1 = 2 + x_2 - x_3 - x_6$. Using this relation, we obtain the new dictionary

$$\begin{aligned} x_1 &= 2 + x_2 - x_3 - x_6 \\ x_4 &= 5 - x_3 + x_6 \\ x_5 &= 18 - 3x_2 - x_3 + 2x_6 \\ z &= 4 - x_2 + 2x_3 - 2x_6. \end{aligned}$$

There is only one choice for the non-basic variable that we should increase, since only an increase of x_3 can lead to an increase of the value of the objective function. As to the amount we should increase x_3 with, we have $x_1 = 2 - x_3$, $x_4 = 5 - x_3$, and $x_5 = 18 - x_3$, from which follows that the value of x_3 should become 2 and that x_1 becomes non-basic again. This leads to the basic feasible solution $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 0, 2, 3, 16, 0)$ with objective value 8, and the new dictionary reads

$$\begin{aligned} x_3 &= 2 - x_1 + x_2 - x_6 \\ x_4 &= 3 + x_1 - x_2 + 2x_6 \\ x_5 &= 16 + x_1 - 4x_2 + 3x_6 \\ z &= 8 - 2x_1 + x_2 - 4x_6. \end{aligned}$$

We see that x_2 is the only candidate for being increased. We have that $x_3 = 2 + x_2$, $x_4 = 3 - x_2$, and $x_5 = 16 - 4x_2$, from which follows that the value of x_3 should become 3 and

that x_4 becomes non-basic. This leads to the basic feasible solution $(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 3, 5, 0, 4, 0)$ with objective value 11. The new dictionary is

$$\begin{aligned} x_2 &= 3 + x_1 - x_4 + 2x_6 \\ x_3 &= 5 - x_4 + x_6 \\ x_5 &= 4 - 3x_1 + 4x_4 - 5x_6 \\ z &= 11 - x_1 - x_4 - 2x_6. \end{aligned}$$

Since there are no candidates from among the set of non-basic variables to be increased in value, the simplex algorithm stops. Since all variables must be nonnegative, the last line of the dictionary implies that $z \leq 11$ for each feasible solution. Hence, we may conclude that the current basic feasible solution is optimal. This observation identifies the general rule that we have reached an optimal solution if the coefficients of all non-basic variables in the objective function are smaller than or equal to zero.

Section 3.5. The simplex algorithm

In the previous section, we have seen the simplex algorithm at work on a linear program that was selected to be bounded and feasible (together with some other nice features that will be pointed out later). The simplex algorithm cannot conclude beforehand that some linear program is bounded and feasible; in fact, showing that a linear program is feasible boils down to finding an optimal solution if we add the constraint $z \geq z^*$, where z^* denotes the optimum solution.

Suppose that we apply the simplex algorithm to a linear program that is unbounded; take for example the instance of Section 3.3:

$$\max \; 2x_1 - x_2$$

subject to

$$\begin{aligned} x_1 - x_2 &\leq 1 \\ -2x_1 - x_2 &\leq -8 \\ x_1, x_2 &\geq 0. \end{aligned}$$

We introduce slack variables $x_3 = 1 - x_1 + x_2$ and $x_4 = -8 + 2x_1 + x_2$, which we request to be nonnegative. Starting with the basic feasible solution $(x_1, x_2, x_3, x_4) = (0, 8, 9, 0)$, we obtain the following dictionary:

$$\begin{aligned} x_2 &= 8 - 2x_1 + x_4 \\ x_3 &= 9 - 3x_1 + x_4 \\ z &= -8 + 4x_1 - x_4. \end{aligned}$$

Clearly, x_1 is the non-basic variable that we want to increase as much as possible. As to the value of x_1, we are bounded by $x_2 = 8 - 2x_1 \geq 0$ and $x_3 = 9 - 3x_1 \geq 0$, from which we obtain our new basic feasible solution $(x_1, x_2, x_3, x_4) = (3, 2, 0, 0)$ and our new dictionary

$$x_1 = 3 - \tfrac{1}{3}x_3 + \tfrac{1}{3}x_4$$
$$x_2 = 2 + \tfrac{2}{3}x_3 + \tfrac{1}{3}x_4$$
$$z = 4 - \tfrac{4}{3}x_3 + \tfrac{1}{3}x_4.$$

The non-basic variable whose value we want to increase is x_4. As to the amount with which we are allowed to increase the value of x_4, we have that $x_1 = 3 + \frac{1}{3}x_4 \geq 0$ and $x_2 = 2 + \frac{1}{3}x_4 \geq 0$. Hence, we see that *there is no restriction to the amount with which we can increase the value of the variable* x_4. This is exactly the situation that signals the unboundedness of the linear program under consideration; whenever this happens, we know that the linear program is unbounded, and the simplex algorithm stops.

Now let us have a look at the special case with multiple optimal solutions. Consider the example of Section 3.3:

$$\max\ 3x_1 + 2x_2$$

subject to

$$x_1 + x_2 \leq 5$$
$$3x_1 + 2x_2 \leq 12$$
$$x_1, x_2 \geq 0.$$

We introduce the slack variables $x_3 = 5 - x_1 - x_2$ and $x_4 = 12 - 3x_1 - 2x_2$ that have to be nonnegative. Starting with the basic feasible solution $(x_1, x_2, x_3, x_4) = (0, 0, 5, 12)$, we obtain the dictionary

$$x_3 = 5 - x_1 - x_2$$
$$x_4 = 12 - 3x_1 - 2x_2$$
$$z = 3x_1 + 2x_2.$$

We select x_1 to become basic; this yields the dictionary

$$x_1 = 4 - \tfrac{2}{3}x_2 - \tfrac{1}{3}x_4$$
$$x_3 = 1 - \tfrac{1}{3}x_2 + \tfrac{1}{3}x_4$$
$$z = 12 - x_4.$$

From the last dictionary, it follows that we should not increase the value of x_4, but there is nothing against increasing the value of x_2. We only have to take care that $x_1 = 4 - \frac{2}{3}x_2 \geq 0$ and $x_3 = 1 - \frac{1}{3}x_2 \geq 0$. Hence, each increase of the value of x_2 with an amount at most 3 yields a feasible solution with equal cost. We see that the presence of multiple optimal solutions is signaled by the occurrence of a dictionary in which the coefficient of each of the non-basic variables is nonnegative with at least one coefficient equal to zero. Strictly speaking, this is not enough, since the constraints may prohibit an increase of this non-basic variable. If a

non-basic variable cannot be increased in value because of the constraints, then we call the linear program *degenerate*.
Consider the following example of a degenerate linear program.

$$\max\ x_1 + x_2 - x_3$$

subject to

$$\begin{aligned} x_1 + 2x_2 - 3x_3 &\leq 0 \\ -x_1 + x_2 + 4x_3 &\leq 4 \\ x_1, x_2, x_3 &\geq 0. \end{aligned}$$

We introduce the slack variables $x_4 = -x_1 - 2x_2 + 3x_3$ and $x_5 = 4 + x_1 - x_2 - 4x_3$ that have to be nonnegative. Starting with the basic feasible solution $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 0, 4)$, we obtain the dictionary

$$\begin{aligned} x_4 &= -x_1 - 2x_2 + 3x_3 \\ x_5 &= 4 + x_1 - x_2 - 4x_3 \\ z &= x_1 + x_2 - x_3. \end{aligned}$$

The non-basic variables with positive coefficient in the objective function are x_1 and x_2; for both variables, however, we see that the constraint $x_4 \geq 0$ implies that we cannot increase their value. If we put x_1 in the basis at the expense of x_4, then we obtain the dictionary

$$\begin{aligned} x_1 &= -2x_2 + 3x_3 - x_4 \\ x_5 &= 4 - 3x_2 - x_3 - x_4 \\ z &= -x_2 + 2x_3 - x_4, \end{aligned}$$

which corresponds to the basic feasible solution $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 0, 4)$. On the one hand, you may conclude that this move did not get us any further; we are still in the same vertex. On the other hand, we changed the basis; perhaps we are luckier this time. So let us try another iterative step. There is only one possibility: we have to put x_3 in the basis. The constraint $x_5 \geq 0$ limits the increase to 4, and we put x_3 in the basis at the expense of x_5. Our new dictionary reads

$$\begin{aligned} x_1 &= 12 - 11x_2 - 4x_4 - 3x_5 \\ x_3 &= 4 - 3x_2 - x_4 - x_5 \\ z &= 8 - 7x_2 - 3x_4 - 2x_5, \end{aligned}$$

which corresponds to the optimal basic feasible solution $(x_1, x_2, x_3, x_4, x_5) = (12, 0, 4, 0, 0)$. It follows that the seemingly irrelevant change of basis in the previous iteration got us out of

the vertex $(x_1, x_2, x_3, x_4, x_5) = (0, 0, 0, 0, 4)$ to which we seemed to be stuck.

This example of a degenerate linear program was not that horrible. The question that we should ask is of course: Will we always be so lucky to move out of a vertex by just changing the basis, or is it possible that we move from one basis to another without getting anywhere? Stated a bit more formally, the question is:

- Will the simplex algorithm always find an optimal solution, or can it cycle and get stuck in a non-optimal solution?

Unfortunately, the answer to this question is NO. Some pathological examples have been created in which the simplex algorithm keeps on returning to the same basis, which does not correspond to an optimal solution. There is a simple remedy against cycling, however. Until now, we have not stated any rule with respect to which non-basic variable we should choose to enter the basis, except that its coefficient in the objective function should be nonnegative. Similarly, we have not been concerned by the question which variable should leave the basis, except that it must be a variable whose value drops to zero when the new variable enters the basis. It turns out that if we prescribe that from among the possible variables we should choose the one with smallest index, then the simplex algorithm will never cycle; this is known as Bland's rule.

Theorem 3.2. (Bland)
The simplex algorithm terminates as long as the entering and leaving variables are selected by the smallest subscript rule in each iteration. □

Theorem 3.2 implies that, if we have succeeded in determining an initial basic feasible solution of the polyhedron, the simplex algorithm will find an optimal solution. Hence, it has become time to describe a systematical way to determine an initial basic feasible solution to the linear program or to conclude that the linear program is infeasible.

Consider any linear program that is stated in the standard form

$$\max \sum_{1 \leq j \leq n} c_j x_j$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i = 1, \ldots, m)$$

$$x_1, x_2, \ldots, x_n \geq 0.$$

If all values b_i are nonnegative, then we can use the basic feasible solution in which all variables x_i $(i = 1, \ldots, n)$ have value zero and all slack variables x_{n+i} have value b_i $(i = 1, \ldots, m)$ to initialize the simplex algorithm with. Hence, suppose that not all values b_i are nonnegative; let b_m denote the most negative one. We now create another linear program, called the *auxiliary* problem, to determine whether the above linear program is feasible, and if the answer is yes, to find an initial basic feasible solution.

We relax each constraint $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$, to $\sum_{j=1}^{n} a_{ij}x_j - x_0 \leq b_i$, where x_0 is an auxiliary variable that we want to be nonnegative. This move enlarges the feasible region; we have that the feasible region of the original problem coincides with the set of feasible solutions with $x_0 = 0$. Since we are interested in finding a basic feasible solution that does not contain x_0 in the basis, we choose as our objective to minimize x_0, that is, our objective function will be to maximize $z = -x_0$.

Theorem 3.3.
The original linear program is feasible if and only the objective value of the optimal solution amounts to zero. □

We solve the auxiliary problem through the simplex algorithm. To start the simplex method, we use $(x_0, x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+m-1}, x_{n+m}) = (-b_m, 0, \ldots, 0, b_1 - b_m, \ldots, b_{m-1} - b_m, 0)$ as our initial basic feasible solution; since b_m is minimal from among the set of b_i values, the nonnegativity constraints are satisfied.

Theorem 3.3 provides us with a tool to check whether the original linear program is feasible or not. Consider the case that the original linear program is feasible, that is, we have determined a basic feasible solution with $x_0 = 0$. This is not necessarily a basic feasible solution for the original problem, since x_0 might be a basic variable. To circumvent this problem, we run the simplex algorithm with the additional rule that we choose x_0 to leave the basis whenever this is possible, that is, whenever its value drops to zero, which must happen sooner or later. This optimal basic feasible solution is used in our initialization of the simplex algorithm when solving the original problem.

Section 3.6. Duality

In many practical situations, we may not be interested in finding the optimal solution: for instance, if we just want to know whether there exists a feasible solution with an objective function value that falls within some given interval. To solve such a problem, we want to obtain a quick estimate of the optimal value z^* of the objective function. Consider the example of Section 3.4

$$\max \; z = 2x_1 - 3x_2 + 4x_3$$

subject to

$$\begin{aligned} x_1 - x_2 + 2x_3 &\leq 7 \\ 2x_1 + x_2 + 3x_3 &\leq 22 \\ x_1 - x_2 + x_3 &\leq 2 \\ x_1, x_2, x_3 &\geq 0. \end{aligned}$$

To get a lower bound on z^*, we need only come up with a feasible solution. For example $(x_1, x_2, x_3) = (2, 0, 0)$ provides a lower bound of 4, $(x_1, x_2, x_3) = (0, 0, 2)$ provides a lower bound of 8, and $(x_1, x_2, x_3) = (0, 3, 5)$ provides a lower bound of 11.

To get an upper bound, observe that

$$2x_1 - 3x_2 + 4x_3 \leq 2x_1 - 2x_2 + 4x_3 = 2(x_1 - x_2 + 2x_3) \leq 2 \times 7 = 14.$$

Hence, every feasible solution (x_1, x_2, x_3) has a value less than 14. In particular, this inequality holds for the optimal solution, which implies that $z^* \leq 14$. Similarly,

$$2x_1 - 3x_2 + 4x_3 \leq 3x_1 - 3x_2 + 4x_3 = (x_1 - x_2 + 2x_3) + 2(x_1 - x_2 + x_3) \leq 11.$$

Therefore, we have that $z^* \leq 11$. Note that at this point we have proved that $(x_1, x_2, x_3) = (0, 3, 5)$ is an optimal solution, since the objective function value for this solution is equal to the upper bound 11.

To obtain upper bounds, we have constructed linear combinations of constraints. That is, we multiply the first constraint by some non-negative number y_1, the second by y_2, the third by y_3, after which we add them up (in the first case, we had $(y_1, y_2, y_3) = (2, 0, 0)$; in the second case, we had $(y_1, y_2, y_3) = (1, 0, 2)$). The resulting inequality reads

$$(y_1 + 2y_2 + y_3)x_1 + (-y_1 + y_2 - y_3)x_2 + (2y_1 + 3y_2 + y_3) \leq$$
$$7y_1 + 22y_2 + 2y_3.$$

Of course, each of these multipliers must be nonnegative: otherwise the corresponding inequality would reverse its direction. Now, we want to use this inequality as an upper bound, that is,

$$2x_1 - 3x_2 + 4x_3 \leq (y_1 + 2y_2 + y_3)x_1 + (-y_1 + y_2 - y_3)x_2 + (2y_1 + 3y_2 + y_3) \leq$$
$$7y_1 + 22y_2 + 2y_3,$$

which implies that

$$y_1 + 2y_2 + y_3 \geq 2$$
$$-y_1 + y_2 - y_3 \geq -3$$
$$2y_1 + 3y_2 + y_3 \geq 4.$$

Of course, we want the upper bound to be as small as possible. Thus, we are led to the following problem

$$\min 7y_1 + 22y_2 + 2y_3$$

subject to

$$y_1 + 2y_2 + y_3 \geq 2$$
$$-y_1 + y_2 - y_3 \geq -3$$
$$2y_1 + 3y_2 + y_3 \geq 4$$
$$y_1, y_2, y_3 \geq 0,$$

which is a linear program again. This linear program is called the *dual* of the original linear program.

Summarizing the above, we see that every linear programming problem of the form

$$\max \sum_{1 \leq j \leq n} c_j x_j$$

subject to

$$\sum_{1 \leq j \leq n} a_{ij} x_j \leq b_i \quad \text{for all } i \in \{1, \ldots, m\}$$

$$x_j \geq 0, \quad \text{for all } j \in \{1, \ldots, n\}$$

has an associated linear programming problem

$$\min \sum_{1 \leq i \leq m} b_i y_i$$

subject to

$$\sum_{1 \leq i \leq m} a_{ij} y_i \geq c_j \quad \text{for all } j \in \{1, \ldots, n\}$$

$$y_i \geq 0, \quad \text{for all } i \in \{1, \ldots, m\} \ .$$

In this context, the original problem is called the *primal* problem and the associated problem is called the *dual* problem. As in our example, every feasible solution of the dual yields an upper bound on the optimal value of the primal. To state it more explicitly, for every primal feasible solution $(x_1, \ldots, x_n)$ and for every dual feasible solution $(y_1, \ldots, y_m)$, we have that

$$\sum_{1 \leq j \leq n} c_j x_j \leq \sum_{1 \leq j \leq n} \left(\sum_{1 \leq i \leq m} a_{ij} y_i \right) x_j = \sum_{1 \leq i \leq m} \left(\sum_{1 \leq j \leq n} a_{ij} x_j \right) y_i \leq \sum_{1 \leq i \leq m} b_i y_i.$$

The above inequality is extremely useful: if we happen to stumble upon a primal feasible solution $(x_1^*, \ldots, x_n^*)$ and a dual feasible solution $(y_1^*, \ldots, y_m^*)$ such that

$$\sum_{1 \leq j \leq n} c_j x_j^* = \sum_{1 \leq i \leq n} b_i y_i^*,$$

then we may conclude that both these solutions are optimal.

The question that arises immediately in case of a linear program with finite objective function value is:

- Under which conditions can we guarantee that the optimal solution values of the primal and the dual problem are equal?

Since we are concerned with linear programs with a finite objective function value, we only consider feasible bounded linear programs. For such linear programs, we can determine the optimal solution by means of a dictionary. In case of the general example of Section 3.4, the final dictionary read:

$$\begin{aligned} x_2 &= 3 + x_1 - x_4 + 2x_6 \\ x_3 &= 5 - x_4 - x_6 \\ x_5 &= 4 - 3x_1 + 4x_4 - 5x_6 \\ z &= 11 - x_1 - x_4 - 2x_6, \end{aligned}$$

and we showed that this dictionary corresponded to the optimal solution, because the last line implied that the value 11, which was assumed by the current basic feasible solution, was an upper bound on z. If we look more closely at the last line of the dictionary, then we see that $z \leq 11 - x_4 - 2x_6$. Remembering that x_4 and x_6 were the slack variables defined by $x_4 = 7 - x_1 + x_2 - 2x_3$ and $x_6 = 2 - x_1 + x_2 - x_3$, we can transform the inequality into $z \leq (x_1 - x_2 + 2x_3) + 2(x_1 - x_2 + x_3) \leq 11$, which corresponds to the feasible solution $(y_1, y_2, y_3) = (1, 0, 2)$ of the dual problem. Since the value of this solution of the dual problem is equal to the value of a solution of the primal problem, this solution is optimal. Hence, in case of a feasible bounded linear program, we conclude from the last line of the final dictionary that we can always determine an optimal solution to the dual problem with value that is equal to the value of the optimal solution of the primal problem; this solution is such that y_i is equal to the absolute value of the coefficient of x_{n+i}, that is, the slack variable that corresponds to the ith constraint.

Theorem 3.4.
In case of a feasible bounded linear program, the optimal solutions to the primal and the dual problem yield the same objective function value. □

Theorem 3.4 provides us with a good *certificate of optimality*: given a solution to the primal and dual problem, one can easily check whether these solutions are optimal by just checking if both objective solution values coincide. Another pro of Theorem 3.4 is the following. Obviously, the dual of the dual problem is the primal problem. Hence, instead of solving the primal problem, we can just as well solve the dual problem and read the optimal solution to the primal problem from the last line of the final dictionary. This may be useful if the number of constraints in the primal problem is larger than the number of variables, since empirical experiments have shown that the simplex algorithm runs in time proportional to the number of constraints, whereas the number of variables does not make a real difference.

Section 3.7. More examples of linear programming problems

Transportation problem
The transportation problem arises frequently in planning and distribution of goods and services from supply locations to demand locations. Usually a given quantity of goods is available at each supply location (origin), and we have to transport these goods to meet a specified need at each demand location (destination). With a variety of shipping routes that all have

different costs, the objective is to determine how many units should be shipped from each origin to each destination such that all demands are satisfied and the total cost is minimized. This problem can be formulated as a linear program as follows

$$\min \sum_{1 \le i \le m} \sum_{1 \le j \le n} c_{ij} x_{ij}$$

subject to

$$\sum_{1 \le j \le n} x_{ij} \le s_i \quad \text{(for all } i = 1, \ldots, m),$$

$$\sum_{1 \le i \le m} x_{ij} \ge d_j \quad \text{(for all } j = 1, \ldots, n),$$

$$x_{ij} \ge 0 \quad \text{(for all } i, j),$$

where c_{ij} is the cost of shipping one unit from origin i to destination j, s_i is a limit on the number of units that can be supplied from origin i, and d_j is the number of units of demand at destination j.

Maximum flow problem
The maximum flow problem, which was introduced in Section 2.3, can be formulated as a linear problem as follows. Recall that the maximum flow problem is the problem of transporting the maximum amount of flow from the source s to the sink t through a network with capacities on the arcs; the capacity of arc (i, j) is denoted by c_{ij}.

$$\max \sum_{(s,j) \in A} x_{sj}$$

subject to

$$\sum_{(i,j) \in A} x_{ij} - \sum_{(j,k) \in A} x_{jk} = 0 \quad \text{(for all } j \in V \backslash \{s, t\})$$

$$0 \le x_{ij} \le c_{ij} \quad \text{(for all } (i, j) \in A),$$

where the network is given by a graph $G = (V, A)$ with source s and sink t and c_{ij} is the capacity of an arc.

Section 3.8. 0-1 Linear programming

There exists a large class of combinatorial optimization problems that cannot be formulated as an ordinary linear program; to obtain a correct formulation, we need to introduce binary variables, that is, variables that can assume the values 0 and 1 only. In this section, we investigate the consequences of this binary nature of variables.

The weighted matching problem for bipartite and arbitrary graphs
Consider the following formulation of the weighted matching problem

$$\min \sum_{\{i,j\} \in E} w_{ij} x_{ij}$$

subject to

$$\sum_{j:\{i,j\} \in E} x_{ij} = 1 \qquad \text{(for all } i \in V\text{)},$$

$$x_{ij} \in \{0,1\} \qquad \text{(for all } \{i,j\} \in E\text{)}.$$

We do not know how to solve this problem, since we do not know how to handle the condition that the variables should be binary. If we relax the condition $x_{ij} \in \{0,1\}$ to $0 \leq x_{ij} \leq 1$, however, then we obtain a linear program. If we solve this linear program and find a binary solution, then we have a solution to the original problem as well. It can be shown that for bipartite graphs the linear programming relaxation always produces integral solutions. This does not hold for arbitrary graphs, however. Consider for example the graph in Figure 3.1, that is, a triangle with equal edge weights. All variables in the optimal solution to the linear programming relaxation have value 0.5.

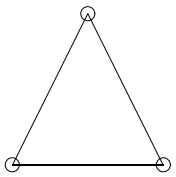


Figure 3.1

Now consider the following inequality

$$\sum_{\{i,j\} \in C} x_{ij} \leq (|C| - 1)/2,$$

where C is an odd cycle in the graph. It is not hard to see that this inequality is satisfied by all integral solutions. It is not satisfied by all solutions to the linear programming relaxation, however, as we can see in our triangle example. This shows that these inequalities are redundant in the 0-1 formulation, but necessary in the linear programming relaxation. It can be shown that if we add all these inequalities to the linear programming relaxation, then the solution to this extended linear programming relaxation will always be integral. Note that there is an exponential number of these inequalities. Therefore, they are handled implicitly in practice.

The 0-1 knapsack problem
The 0-1 knapsack problem is formulated as follows

$$\max \sum_{1 \leq j \leq n} c_j x_j$$

subject to

$$\sum_{1 \leq j \leq n} a_j x_j \leq b$$

$$x_j \in \{0, 1\}.$$

In this case, the linear programming relaxation does not always produce integral solutions, and we do not know the complete set of linear inequalities that need to be added to force solutions to be integral. In Chapter 4, we provide some solution methods for this problem.

Chapter 4. The knapsack problem

Section 4.1. Formulation

We are given a set of n items; each item i ($i = 1, \ldots, n$) has a value $c_i > 0$ and a weight $a_i > 0$. A knapsack with (weight) capacity b has to be filled with items so as to maximize the total value of the items included in the knapsack. Without loss of generality, we assume that all weights a_i and values c_i are integral; due to the integrality of the weights, we can also assume that b is integral.

We formulate the knapsack problem (KS) by using the binary variables x_i ($i = 1, \ldots, n$), where the outcome $x_i = 1$ signals that item i must be included in the knapsack and $x_i = 0$ signals that item i has to be left at home.

$$\text{(KS)} \qquad \max \sum_{i=1}^{n} c_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} a_i x_i \leq b,$$

$$\forall_{i \in \{1,\ldots,n\}} \; x_i \in \{0, 1\}.$$

We suppose that $\sum_{i=1}^{n} a_i > b$ and $a_i \leq b$ $(i = 1, \ldots, n)$ to avoid trivialities.

Although the knapsack problem contains only one non-trivial linear constraint, there is no polynomial algorithm known that solves the problem to optimality. On the other hand, there is no proof that such an algorithm does not exist. There are two alternatives to cope with problems for which no efficient solution methods exist. The first one is to apply an algorithm with *exponential* running time to find an *optimal* solution. To solve the knapsack problem, we can apply the general techniques of *branch and bound* and *dynamic programming*. Both techniques enumerate all possible feasible solutions in a smart way. The second possibility is to apply a *polynomial* algorithm that constructs a good but *possibly non-optimal* solution. Such inexact algorithms are called *approximation algorithms*.

Section 4.2. The continuous knapsack problem

If we no longer demand a solution of the knapsack problem to be integral but allow for fractional values, then we get the *continuous knapsack problem* (CKS); this is a relaxation of the knapsack problem in the sense that the set of feasible solutions for the continuous knapsack problem contains the solution set of knapsack as a subset.

$$\text{(CKS)} \qquad \max \sum_{i=1}^{n} c_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} a_i x_i \leq b,$$

$$\forall_{i \in \{1,\ldots,n\}} \; 0 \leq x_i \leq 1 \; .$$

Consider any instance I_{KS} of the knapsack problem and the corresponding instance of the continuous knapsack problem I_{CKS}; I_{CKS} is called the linear programming relaxation, since the integrality constraints of KS are relaxed to linear constraints. Clearly, for the optimal values of I_{KS} and I_{CKS}, which are denoted by $z(I_{KS})$ and $z(I_{CKS})$, we have that $z(I_{KS}) \leq z(I_{CKS})$. Hence, $z(I_{CKS})$ forms an upper bound on $z(I_{KS})$.

Although the set of feasible solutions of I_{CKS} extends the set of feasible solution of I_{KS}, there is a straightforward greedy algorithm that solves CKS. Renumber the items according to non-increasing $\frac{c_i}{a_i}$ $(i = 1, \ldots, n)$, that is, $\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \ldots \geq \frac{c_n}{a_n}$. For this sequence we have that the most interesting items, that is, those with the highest value per unit of weight are numbered lowest. The greedy algorithm raises the values of the variables in the order $x_1, x_2, \ldots, x_n$.

Greedy algorithm CKS.

Input:	n items with weight $a_i > 0$, value $c_i > 0$, and a capacity $b > 0$.
Output:	An optimal solution $(x_1, \ldots, x_n)$ for the continuous knapsack.

STEP 1. All variables x_i $(i = 1, \ldots, n)$ are set equal to 0; the residual capacity $\bar{b}$ is set equal to b; $j \leftarrow 1$.

STEP 2. If $a_j \leq \bar{b}$, then $x_j \leftarrow 1$; otherwise, $x_j \leftarrow \bar{b}/a_j$. Set $\bar{b} \leftarrow \bar{b} - a_j x_j$; $j \leftarrow j + 1$.

STEP 3. If $\bar{b} > 0$, then go to STEP 2.

Theorem 4.1.
The greedy algorithm gives an optimal solution for CKS.

Proof.
Note that the algorithm fills the knapsack completely, since we assumed that $\sum_{i=1}^{n} a_i > b$ and $c_i > 0$ for all i; this implies that there exists a j, such that $1 = x_1 = \ldots = x_{j-1} > x_j \geq x_{j+1} = \ldots = 0$, where $x_{n+1} = 0$. We show the optimality of this solution by comparing it to any other feasible solution $y_1, \ldots, y_n$ of the continuous knapsack problem. Since all values c_i are positive, this solution can only be optimal if $\sum_{i=1}^{n} a_i y_i = b$. Let k be the smallest index such that $y_k < 1$, and let l be the smallest index with $k < l$ such that $y_l > 0$; note that such an l exists, unless the solution $y_1, \ldots, y_n$ is equal to the solution $x_1, \ldots, x_n$ obtained by the greedy algorithm. We will now increase y_k and decrease y_l, while keeping all other values equal, to obtain a new solution. Let $\varepsilon = \min\{a_k(1 - y_k), a_l y_l\} > 0$. Increase y_k by $\frac{\varepsilon}{a_k}$ and decrease y_l by $\frac{\varepsilon}{a_l}$. It is easily checked that this move yields a feasible solution with value no smaller than the value of the solution $y_1, \ldots, y_n$. Moreover, either y_k has become equal to 1, or y_l has become equal to 0. Repetition of this argument eventually yields the solution $x_1, \ldots, x_n$ obtained by the greedy algorithm. □

Section 4.3. A branch and bound algorithm

The main characteristic of a branch and bound algorithm is that it partitions the solution set into subsets $S_1, \ldots, S_K$, which are described by a partial feasible solution, that is, some of the values of the variables $x_1, \ldots, x_n$ have been fixed. For example, we obtain a partition of the solution set into subsets S_1 and S_2 by *branching* with respect to the value of x_1: we request each solution in S_1 to have $x_1 = 1$ and each solution in S_2 to have $x_1 = 0$. We have that each subset itself boils down to a smaller knapsack problem. An upper bound UB_k on the outcome of the knapsack problem corresponding to S_k can therefore be calculated by means of the greedy algorithm for the continuous knapsack problem. A lower bound LB_k follows from rounding down the fractional variable, if it exists. If no fractional variable exists, then $LB_k = UB_k$.

Let z_F denote the value of the best feasible solution found so far; we have that $z_F \geq \max\{LB_k | k = 1, \ldots, K\}$. We call a pair (S_k, UB_k) *active* if $UB_k > z_F$: the subset S_k may contain a feasible solution with value greater than the value z_F of the current best solution. Hence, it might be wise to pay some extra attention to this subset. The pair (S_k, UB_k) is called *inactive* if $UB_k \leq z_F$: this subset of solutions is of no interest anymore, since it cannot contain a solution with value greater than z_F. Hence, we fathom this subset.

We implement our branch and bound algorithm as follows. We construct a list L that contains all active pairs (S_k, UB_k). The branching part of the algorithm consists of splitting a set S_k into two sets S_k^0 and S_k^1 according to our fixing the value of some variable x_j the value of which has not been fixed before; all solutions in the sets S_k^0 and S_k^1 are requested to have $x_j = 0$ and $x_j = 1$, respectively. Initially, L consists of the original problem, that is, there are no variables whose with fixed value. The solution of the continuous knapsack problem is used as an initial upper bound. We determine z_F by rounding down the value of the fractional variable.

As long as the list L of active pairs is not empty, we perform the following steps.

STEP 1. Choose from among the list of active pairs the one (S_k, UB_k) with maximal upper bound.

STEP 2. Split S_k in S_k^0 and S_k^1 according to the fractional variable x_i in the continuous solution.

STEP 3. Calculate UB_k^0 and UB_k^1 through the greedy algorithm CKS.

STEP 4. Round down the continuous solutions obtained in STEP 3 to obtain feasible solutions for KS; this gives the lower bounds LB_k^0 and LB_k^1.

STEP 5. If $LB_k^0 > z_F$, then set $z_F \leftarrow LB_k^0$; if $LB_k^1 > z_F$, then set $z_F \leftarrow LB_k^1$.

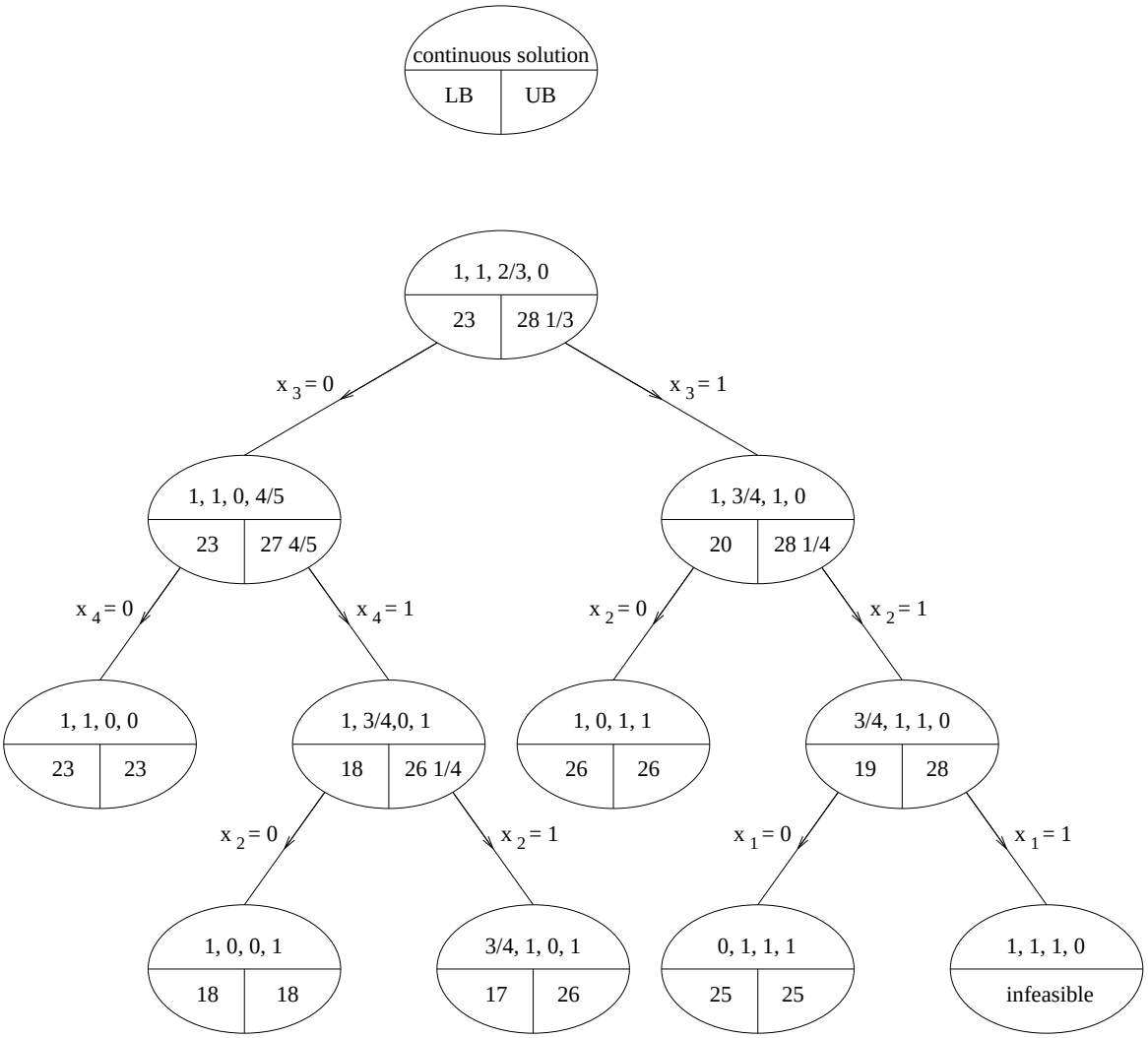
STEP 6. If $UB_k^0 > z_F$, then the pair (S_k^0, UB_k^0) is added to L; if $UB_k^1 > z_F$, then the pair (S_k^1, UB_k^1) is added to L.

The branch and bound process is illustrated by displaying it in the form of a tree, where the active and inactive pairs (S_k, UB_k) form the leaves. Therefore the active sets are usually called nodes.

Example: 4 items; $b = 20$

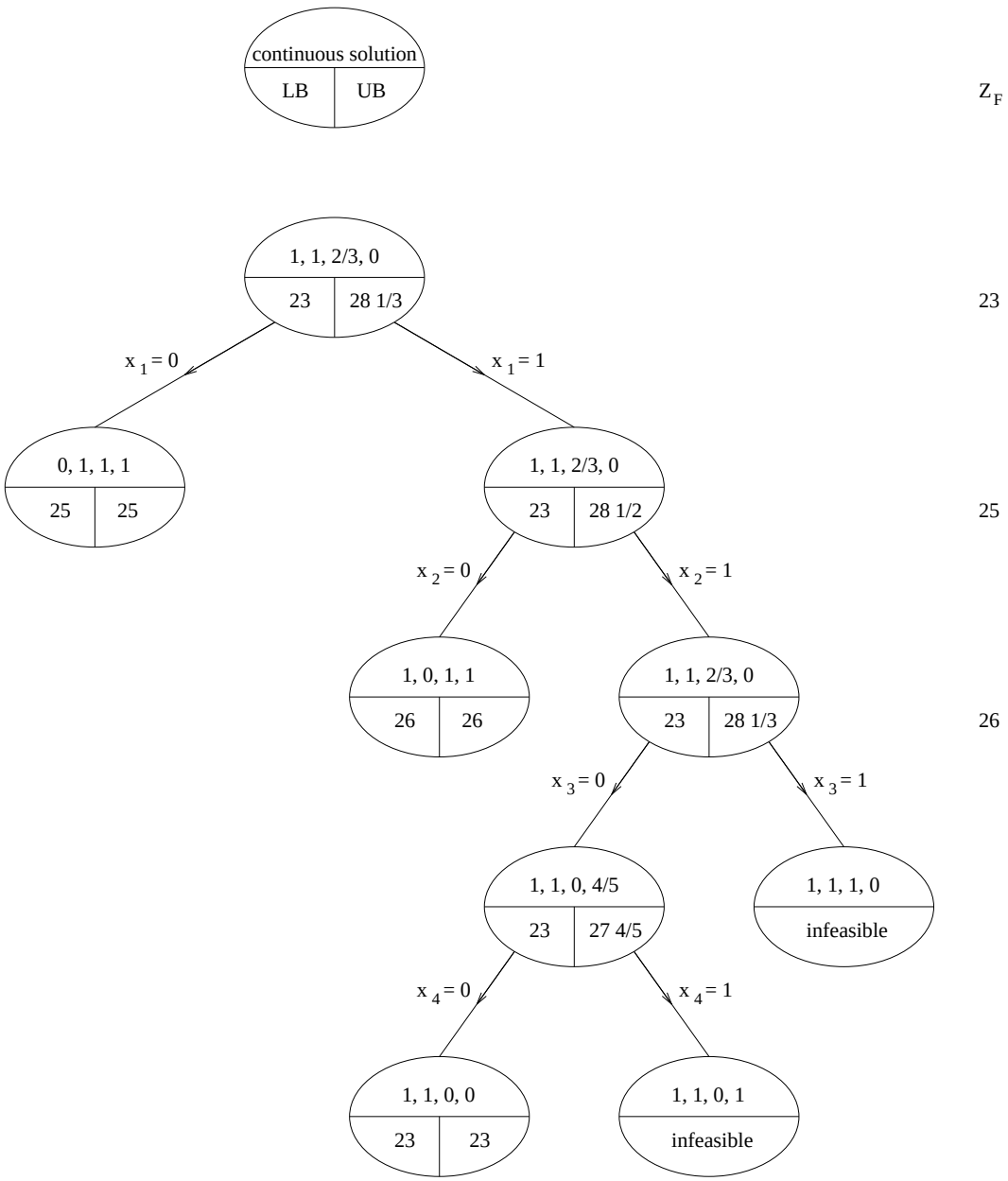
item i	1	2	3	4
c_i	12	11	8	6
a_i	8	8	6	5

Each node in the tree contains the following information.



Note that, since the outcome of an optimal feasible solution for the knapsack problem is integral, we have that the outcome of the knapsack problem amounts to no more than the integral part of the outcome of the continuous knapsack problem. Hence, we can achieve a small improvement by declaring a pair (S_k, UB_k) inactive if $\lfloor UB \rfloor < z_F$. We can even go further by already fathoming a pair (S_k, UB_k) if $\lfloor UB \rfloor = z_F$: we are interested in finding one optimal solution only.

A more significant improvement is obtained by changing the branching rule, that is, the choice of the variable according to which the active subset is split. In the elaboration below, the variable with highest value per unit of weight is chosen, that is, the variable with the lowest index.



Section 4.4. Dynamic Programming

Another method that can be applied to solve the knapsack problem to optimality is *dynamic programming*. The idea behind the dynamic programming algorithm is that, if we know the maximal value $f_k(d)$ that can be gained by filling a knapsack of capacity d with the items $1, \ldots, k$ for all appropriate values of d, then we can readily determine the maximal values $f_{k+1}(d)$ for all appropriate values of d. As to which values of d are appropriate, we have that, since the weights a_i are all integral, only integral values of d are of interest; obviously, there is no use in considering values of d that are greater than b.

The values $f_k(d)$ can be calculated recursively from the values $f_{k-1}(d')$, with $d' = 0, \ldots, d$, as follows:

$$f_k(d) = \max\{f_{k-1}(d), f_{k-1}(d - a_k) + c_k\} \ .$$

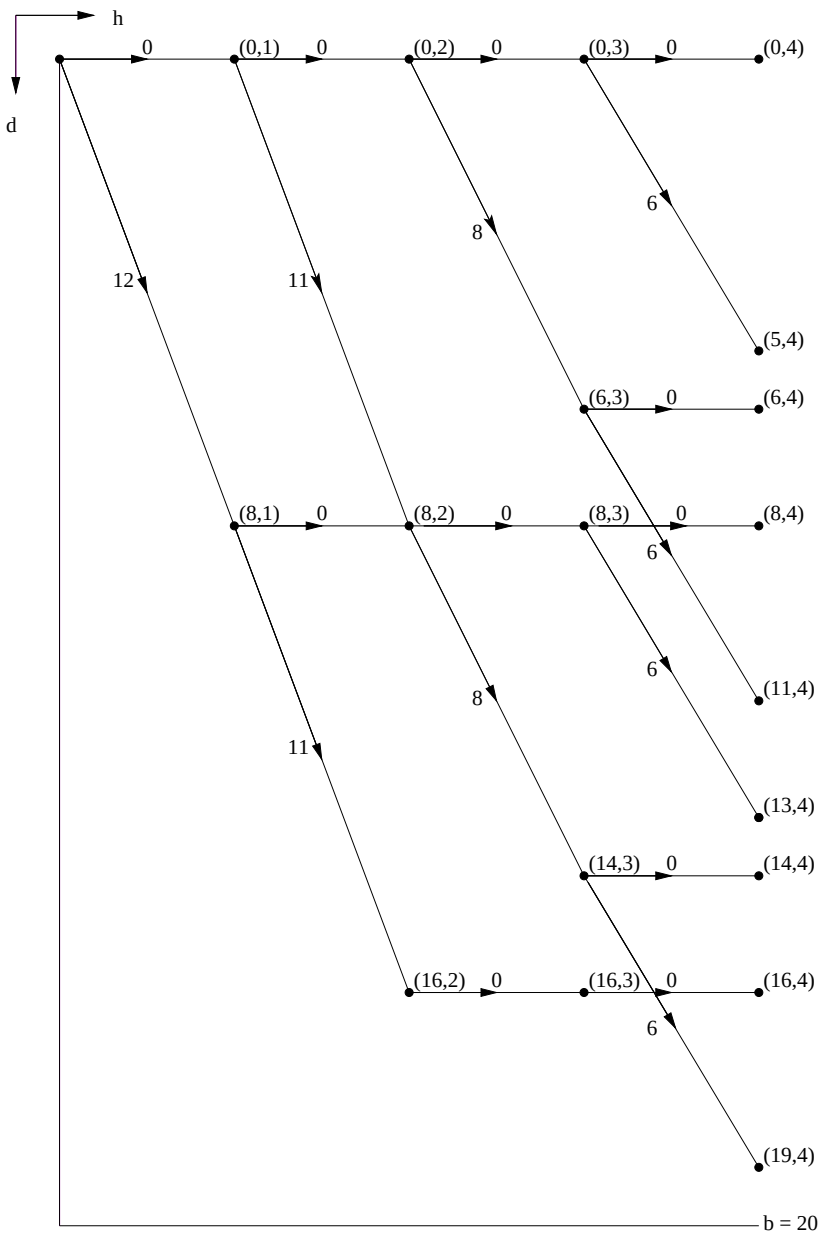
The interpretation is that item k may or may not be part of an optimal filling of the knapsack with capacity d and items $1, \ldots, k$ available. If it is not, then $f_k(d) = f_{k-1}(d)$; if it is, then $f_k(d) = f_{k-1}(d - a_k) + c_k$.

We implement the dynamic programming algorithm as follows. As an initialization, we set $f_0(d) = 0$ for $d = 0, 1, \ldots, b$. In iteration k, we add item k and compute $f_k(d)$ for $d = 0, \ldots, b$ on the basis of our knowledge of $f_{k-1}(d)$ for $d = 0, \ldots, b$. The optimal value of the knapsack problem is then equal to $f_n(b)$, and the corresponding solution is determined by tracing the path back. Since $f_k(d)$ is determined in constant time when $f_{k-1}(d)$ and $f_{k-1}(d - a_k)$ are known, we can compute $f_n(b)$ in $O(nb)$ time. This is not polynomial, because the input of the knapsack problem is a linear function of n and $\log b$ (and of $\log a_i$ and $\log c_i$).

We illustrate the dynamic programming algorithm by treating it as a longest path problem with vertices (k, d) $(k = 0, \ldots, n; d = 0, \ldots, b)$. There is an arc of cost 0 from $(k-1, d)$ to (k, d) and an arc of cost c_k from $(k-1, d-a_k)$ to (k, d). For each k, we need only a part of the values: if $f_k(d+1) = f_k(d)$, then it is not necessary to maintain $f_k(d+1)$. This fact is used in the diagram of the example presented below. For illustrative reasons, many arcs are left out of the diagram.

Example 4 items; $b = 20$

item i	1	2	3	4
c_i	12	11	8	6
a_i	8	8	6	5



The optimal value is $f_4(20) = f_4(19) = 26$.

The question which of the two exact algorithms is to be preferred does not have a unique answer. Empirical analysis suggests that problems with many items are solved fastest by branch bound and problems with few items are best solved by the dynamic programming approach. A possible reason for it is that the dynamic programming algorithm grows linearly with n, whereas branch and bound usually grow sublinearly with n, due to the good lower and upper bounding capability; in case of bad luck, however, we may have to inspect all possible subsets, of which there are $O(2^n)$.

Section 4.5. Approximation algorithms

If we do not insist on finding an optimal solution, we can resort to an approximation algorithm for a quickly determined solution, which unfortunately can be quite bad. It is easy to develop an approximation algorithm; usually, it is much more difficult to estimate the quality of the outcome of the algorithm. We describe three approximation algorithms for the knapsack problem and a method of how to measure the quality of an approximation algorithm.

Approximation algorithm 1.
Round down the solution of the continuous knapsack problem. Denote its value by z_1.

Approximation algorithm 2.
Fill the knapsack with item i with the highest value c_i only. Denote its value by z_2.

An example for which Approximation algorithm 1 performs poorly is the following:

$n = 2; \quad b = 10;$

i	1	2
c_i	2	10
a_i	1	10

Approximation algorithm 1 rounds down the solution $(1, \frac{9}{10})$ to $(1, 0)$ and finds a solution with value $z_1 = 2$, whereas the value $z_{opt} = 10$ is optimal; this yields the relative performance $\frac{z_1}{z_{opt}} = \frac{1}{5}$. This performance can be made arbitrary small in the following way: take the instance $n = 2; c_1 = 2, a_1 = 1, c_2 = a_2 = x; b = x$, where x is a very large integer. Note that Approximation algorithm 2 provides the optimal solution.

An example where Approximation algorithm 2 performs poorly is the following:

$n = 6; \quad b = 6;$

i	1	2	3	4	5	6
c_i	1	1	1	1	1	1
a_i	1	1	1	1	1	1

Approximation algorithm 2 just picks one item and hence $z_2 = 1$, whereas $z_{opt} = 6$; this yields the relative performance $\frac{z_2}{z_{opt}} = \frac{1}{6}$. This performance can also be made arbitrarily small: take the instance with $n = b = x$ and $a_i = c_i = 1$ for $i = 1, \ldots, x$. Approximation algorithm 1 provides the optimal solution.

The question whether there are instances for which both approximation algorithms behave poorly is answered by examining the third approximation algorithm.

Approximation algorithm 3.
Take the best solution of Approximation algorithms 1 and 2, that is, its value, denoted by z_3, is set equal to $z_3 \leftarrow \max\{z_1, z_2\}$.

Consider the following example:

$n = 3; \quad b = 10;$

i	1	2	3
c_i	6	5	5
a_i	6	5	5

For the above example, both Approximation algorithm 1 and Approximation algorithm 2 find a solution of value 6, which implies $z_3 = 6$. Since the optimal solution value is 10, we get the relative performance $\frac{z_3}{z_{opt}} = \frac{6}{10}$. It seems that one cannot do much worse. In fact, there is a hard guarantee on the behavior of Approximation algorithm 3 concerning the relative error.

Theorem 4.2.
Approximation algorithm 3 has worst-case bound 1/2, that is, for all instances of knapsack, Approximation algorithm 3 finds a solution with value at least half the value of the optimal solution, and this bound can be approximated arbitrarily closely.

Proof.
Consider any instance of knapsack. Let the continuous solution be $x_1 = \ldots x_{k-1} = 1, x_k = \varepsilon < 1$, $x_{k+1} = \ldots = x_n = 0$; let Approximation algorithm 2 choose item i with $c_i \geq c_k$. Since $z_{opt} \leq z_{CKS} < c_1 + \ldots + c_{k-1} + c_k \leq c_1 + \ldots + c_{k-1} + c_i = z_1 + z_2 \leq 2z_3$, we have that $z_3 > \frac{1}{2} z_{opt}$.
A type of example with which the worst-case bound of 1/2 can be approximated arbitrarily closely is the following one: Take $n = 3$ with $c_1 = a_1 = x + 1$, $c_2 = a_2 = c_3 = a_3 = x$, and $b = 2x$, where x is some large integer; the instance with $x = 5$ is shown above. The performance of Approximation algorithm 3 amounts to $(x + 1)/2x$, which goes to 1/2 if x goes to infinity. □

Section 4.6. Complexity

For a given problem, the first question is to decide whether it is polynomially solvable or not. Proving that a problem is polynomially solvable can be done by providing a polynomial algorithm that solves the problem to optimality. But what if there is no such algorithm available? Either there is an efficient algorithm that we cannot find, or there is no such algorithm. The question is how to prove that there is no polynomial algorithm available. Unfortunately, we do not know. There is a huge class of problems for which no polynomial optimization algorithms are known, but nobody has been able to prove that such algorithms do not exist. The knapsack problem belongs to this class. But for a subclass of these problems, the following nice result is knows: if we can find a polynomial algorithm that solves one of the problems in this class, then we can solve all problems in this subclass in polynomial time. So, for lack of something better, one is left with showing that the problem under consideration belongs to this subclass. This proof proceeds in the following way. We take an arbitrary instance of a problem that is known to belong to this subclass (so we will call it a *hard* problem) and show how to construct a special instance of the problem under consideration *whose outcome is identical to the outcome of the general instance of the hard problem.* The thought behind it is that we can solve the hard problem by solving the problem under consideration: this must then be hard, too.

To show that the knapsack problem is hard too, we use a well-known simply formulated problem: the partition problem.

Partition
Given a set of t weights $u_1, \ldots, u_t$ with $\sum_{i=1}^{t} u_i = 2u$, the question is: Can these weights be partitioned into two sets S and $\{1, \ldots, t\} \backslash S$ of equal cumulative weight, that is, $\sum_{i \in S} u_i = u$?

Researchers have not been able to come up with an efficient algorithm for partition yet. Therefore, it *seems* unlikely that an efficient algorithm for partition exists. To show that the same holds for the knapsack problem, it suffices to show that partition is a special case of the knapsack problem.

Consider any instance of Partition with t weights, $u_1, \ldots, u_t$; $u = \frac{1}{2}\sum_{i=1}^{t} u_i$.

We construct our special instance of the knapsack problem as follows: For each weight u_i in partition, we introduce an item i with both weight and value equal to u_i; the capacity of the knapsack is set equal to u. This instance is depicted below:

$n \leftarrow t$,
$c_i \leftarrow u_i \ (i = 1, \ldots, n)$,
$a_i \leftarrow u_i \ (i = 1, \ldots, n)$,
$b \leftarrow u$.

The corresponding instances of the knapsack and the partition problem are equivalent in the following sense.

Theorem 4.3.
An instance $(t, u_1, \ldots, u_t, u)$ of partition has a solution if and only if the corresponding instance of knapsack has an optimal solution with value equal to u.

Proof.
Suppose that the instance of partition has a solution; let S be the corresponding solution. For the corresponding instance of knapsack, we choose $x_i = 1$ if $i \in S$ and $x_i = 0$, otherwise. We have that

$$\sum_{i=1}^{n} a_i x_i = \sum_{i \in S} a_i = \sum_{i \in S} u_i = u = b \ , \text{ and}$$

$$\sum_{i=1}^{n} c_i x_i = \sum_{i \in S} c_i = \sum_{i \in S} u_i = u \ .$$

This is optimal, since $u = b \geq \sum_{i=1}^{n} a_i x_i = \sum_{i=1}^{n} c_i x_i$.

On the other hand, suppose that the knapsack problem has an optimal solution with value b. Let S be the set of items i for which $x_i = 1$. Then we have that $u = \sum_{i=1}^{n} c_i x_i = \sum_{i \in S} c_i = \sum_{i \in S} u_i$, which implies that partition has a solution. □

We are not completely done yet. In the previous section on dynamic programming, we showed that it is possible to solve a knapsack problem by solving an instance of the longest path problem in an acyclic network; a problem that is known to be solvable in polynomial time. The instance of the longest path problem, however, was much bigger ($O(nb)$) than the instance of the knapsack problem ($O(n \log b)$). We need the extra condition that the size of the knapsack instance corresponding to the instance of partition must be polynomial in the size of the instance of partition. Now that we have checked that this condition is satisfied, we have that knapsack is at least as hard as partition. Note that the knapsack problem does not have to be just as hard as partition; to show that, we must show that the knapsack problem contains partition as a special case, which can easily be shown.

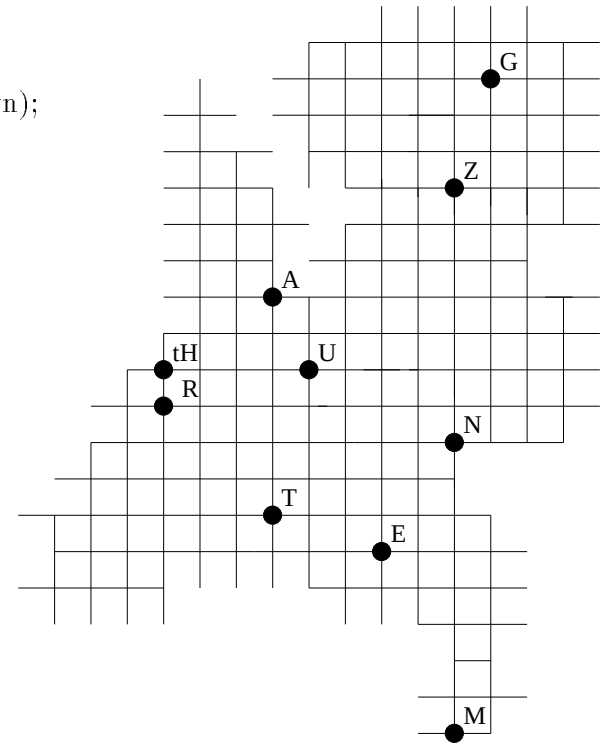
Chapter 5. The traveling salesman problem

Section 5.1. Problem definition

The traveling salesman problem (TSP).

A big company in the Netherlands sends one of its salesmen to the clients in the major cities in the country. A list of these cities together with their distances is given below. Starting and finishing in his hometown, the salesman has to visit these cities all in one day. Since he wants to return home as soon as possible, he must visit the cities in such an order that the total traveling distance is minimal (where we assume that the traveling time depends on the distance only).

Cities:
Amsterdam;
Eindhoven (home town);
Groningen;
the Hague;
Maastricht;
Nijmegen;
Rotterdam;
Tilburg;
Utrecht;
Zwolle.



The distance matrix is the following.

	A	E	G	tH	M	N	R	T	U	Z
A		125	184	57	207	103	77	121	40	103
E			235	127	125	62	101	37	91	144
G				236	309	192	242	233	184	94
tH					217	124	26	102	61	151
M						120	191	119	176	218
N							96	93	62	105
R								76	58	148
T									78	166
U										90
Z										

We have assumed the distance-matrix to be symmetric; therefore, the part above the main diagonal contains all necessary information.

The problem can be modeled as a minimization problem in graphs as follows. We are given a complete graph $G = (V, E)$ and a distance function $d : E \to R^+$. The problem is to find a Hamiltonian circuit of minimum total length that contains each vertex of V exactly once. Such a circuit is described by a permutation $\pi = (\pi(1), \ldots, \pi(n))$ of the vertices, where $\pi(i)$ is the vertex that is on the i-th position assuming that we start with vertex $\pi(1)$. The length of the circuit described by π is therefore

$$\sum_{i=1}^{n} d_{\pi(i),\pi(i+1)} \quad (\text{where } \pi(n+1) = \pi(1)) \; .$$

In the usual terminology for traveling salesman problems, the vertices are called cities and Hamiltonian circuits are called tours.

Variants.
The problem above is called the *symmetric* TSP, because the distance between each pair of cities is independent of the direction in which the salesman travels. This restriction is not always valid; we then obtain the following, more general, form of the TSP.

The asymmetric TSP.
Given a complete digraph $G = (V, A)$ and a length function $d : A \to R^+$, we are asked to find a directed Hamiltonian circuit, that is, a directed circuit that contains each vertex exactly once, of minimum length.

Both the symmetric and the asymmetric TSP are defined on a complete (di)graph. This is not a serious restriction, however, since we can always add arcs or edges with a sufficiently high cost; sufficiently should be read as: if there exists a tour in the original graph, then none of the extra arcs or edges will be part of the tour of minimum length.

The Hamiltonian path problem.
If the salesman does not have to return home, then we get the Hamiltonian path problem: Given a complete graph $G = (V, A)$ and a length function $d : A \to R^+$, we are asked to find a Hamiltonian path, that is, a path that contains each vertex exactly once, of minimum length. There are three variants of this problem, depending on the number of endpoints that are fixed.

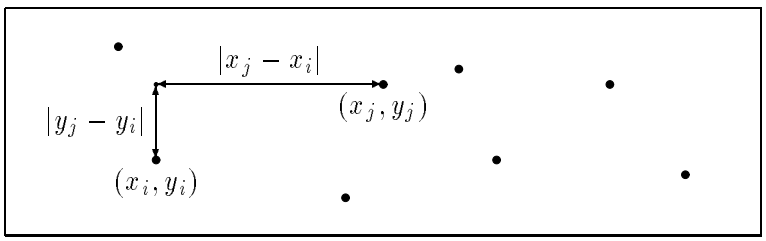
The Hamiltonian path problem is easily converted to a standard TSP problem by adding a dummy vertex with appropriate distance to all other vertices, where the specification of appropriate depends on the number of endpoints that have been fixed. If no endpoints have been fixed, then we are indifferent to which vertices are connected to the dummy vertex (and hence will star as the endpoints of the path). If a vertex has been specified as endpoint, then we have to take care that this vertex is connected to the dummy vertex in the solution of the TSP; this is arranged by setting the distance from this point to the dummy vertex equal to zero, whereas all vertices that are not specified as endpoint get a huge distance to the dummy vertex. On the other hand, it is not hard to convert a TSP problem into a minimum length Hamiltonian path problem.

There are no polynomial algorithms known for the TSP and its variants. It can be shown that the partition problem is a special case, but that is beyond the scope of this manuscript.

Section 5.2. Applications

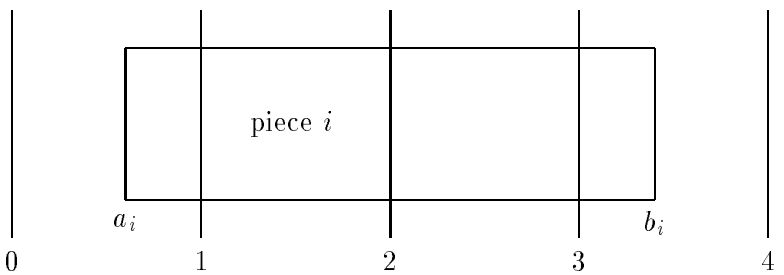
A natural application of the TSP is the Euclidean TSP, in which the cities are given by points in the plane; the distances between pairs of cities are the Euclidean distances. The following application is of a similar type.

Printed circuit boards.
In the process of manufacturing electronic circuit boards, holes have to be drilled in prints of rectangular shape; these holes have to be pairwise connected by conductive material. The head of the drilling equipment can only be moved horizontally and vertically from one hole to another. Usually, an enormous amount of holes has to be drilled in a board. Since moving the head of the drill is a time consuming operation, the total distance that the head has to be moved should be minimized. Moreover, after the drilling process has been completed, the head must return to its starting position. Hence, this problem is equivalent to the problem of finding a Hamiltonian circuit of minimum length in a graph in which the vertices correspond to the holes and the distance between two vertices i and j with coordinates (x_i, y_i) and (x_j, y_j) amounts to $|x_i - x_j| + |y_i - y_j|$.



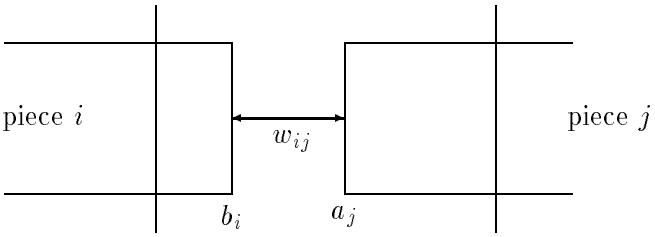
This problem resembles the Euclidean TSP. The only difference between the two problems is the definition of the distances between pairs of points in the plane. For the printed circuit boards the 1-norm is taken, whereas the Euclidean TSP uses the 2-norm.

Wallpaper cutting.
Suppose we are to cut $n - 1$ sheets of wallpaper from a very long roll of wallpaper to cover a wall in a house. The roll contains a printed pattern that repeats itself every meter. Each piece i ($i = 1, \ldots, n$) starts at position a_i and finishes at position b_i with respect to the point where the pattern starts; these positions depend on the pieces, that is, they may be different for distinct pieces, because of the presence of windows, doors, etc., in the wall.



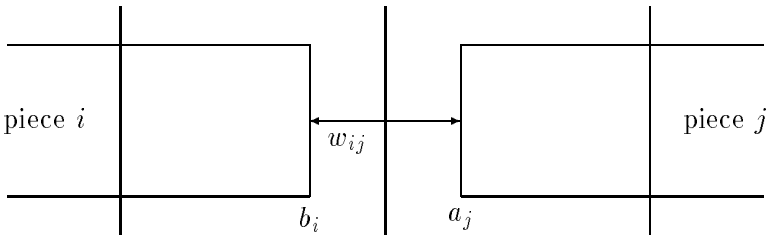
Now suppose that piece j is cut off the roll immediately after piece i. Then the wastage w_{ij} between the pieces depends on whether $a_i \geq b_j$ or not.

Case 1: Let $a_j \geq b_i$.



Hence, the wastage amounts to $w_{ij} = a_j - b_i$.

Case 2: Let $a_j < b_i$.



Hence, the wastage amounts to $w_{ij} = 1 + a_j - b_i$.

We have now characterized the wastage between the pieces. Suppose the roll begins at the zero point on the pattern and after cutting our $n-1$ pieces we have to make an additional cut to let the roll start at the zero point on the pattern again. To take into account the wastage incurred by the first and the last cut, we add a dummy piece n with $a_n = b_n = 0$. This turns the problem into an asymmetric TSP problem with the distance matrix is $W = (w_{ij})$. Surprisingly enough this special case of the TSP can be solved in polynomial time.

Clustering a data array.

Suppose we are given a data array in the form of an $m \times n$ matrix $A = (a_{ij})$ that consists of elements that are either 0 or 1: the entry a_{ij} gets the value 1 if a relation between row i and column j exists. If there is no relation between the row and the column, then $a_{ij} = 0$. We are interested in grouping rows and columns together in such a way that they show similar relations.

For example, consider a number of m marketing techniques and n products. If a marketing technique i works out successfully on a product j, then $a_{ij} = 1$; otherwise, $a_{ij} = 0$. Similar marketing techniques are supposed to be successful on similar products. Therefore, grouping the techniques and the products gives insight in the relations of the marketing techniques and the products.

Consider the following two matrices. The second matrix is constructed from the first one by permuting rows 2 and 5 and columns 2 and 5.

$$\begin{array}{|ccccc|} \hline 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ \hline \end{array} \quad \longrightarrow \quad \begin{array}{|ccccc|} \hline 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

Intuitively, the matrix on the right gives a better idea of which rows and columns have a relation. To formalize this, we introduce *the measure of effectiveness* for each element a_{ij} of the matrix; it is defined as the product of this element with its (at most) four direct neighbors; these are, the upper and lower and the left and right neighbors, if they exist. To ensure their existence, we add artificial rows 0 and $m+1$ and columns 0 and $n+1$ that contain zeroes only to the matrix A. The total measure of effectiveness (TME) is computed by summing the measure of effectiveness over all elements of the matrix except for the artificial rows and columns:

$$TME = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}(a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}).$$

The TME's of the two matrices in the example are 0 and 32, respectively. For arbitrary permutations ρ and σ of the rows $1, \ldots, m$ and columns $1, \ldots, n$, the TME becomes

$$\begin{aligned} TME &= \sum_{i=1}^{m} \sum_{j=1}^{m} a_{\rho(i)\sigma(j)}(a_{\rho(i)\sigma(j-1)} + a_{\rho(i)\sigma(j+1)} + a_{\rho(i-1)\sigma(j)} + a_{\rho(i+1)\sigma(j)}) \\ &= \sum_{i=1}^{m} \sum_{j=1}^{n} [a_{\rho(i)\sigma(j)}a_{\rho(i)\sigma(j-1)} + a_{\rho(i)\sigma(j)}a_{\rho(i)\sigma(j+1)}] \qquad (1) \\ &+ \sum_{i=1}^{m} \sum_{j=1}^{n} [a_{\rho(i)\sigma(j)}a_{\rho(i-1)\sigma(j)} + a_{\rho(i)\sigma(j)}a_{\rho(i+1)\sigma(j)}]. \qquad (2) \end{aligned}$$

The interpretation of (1) is the following. For the row placed on the i-th position, that is, the row denoted by $\rho(i)$, each element is multiplied with its predecessor and its successor with respect to σ. Rewriting this formula yields:

$$\begin{aligned} &\sum_{i=1}^{m} \sum_{j=1}^{n} [a_{\rho(i)\sigma(j)}a_{\rho(i)\sigma(j-1)} + a_{\rho(i)\sigma(j)}a_{\rho(i)\sigma(j+1)}] = \\ &\sum_{k=1}^{m} \sum_{j=1}^{n} [a_{k\sigma(j)}a_{k\sigma(j-1)} + a_{k\sigma(j)}a_{k\sigma(j+1)}] = \\ &\sum_{k=1}^{m} \sum_{j=1}^{n} a_{k\sigma(j)}a_{k\sigma(j-1)} + \sum_{k=1}^{m} \sum_{j=1}^{n} a_{k\sigma(j)}a_{k\sigma(j+1)} = \end{aligned}$$

$$\sum_{k=1}^{m} \sum_{j=0}^{n-1} a_{k\sigma(j+1)} a_{k\sigma(j)} + \sum_{k=1}^{m} \sum_{j=1}^{n} a_{k\sigma(j)} a_{k\sigma(j+1)} =$$

$$2 \sum_{k=1}^{m} \sum_{j=1}^{n} a_{k\sigma(j)} a_{k\sigma(j+1)} = 2 \sum_{j=1}^{n} \left(\sum_{k=1}^{m} a_{k\sigma(j)} a_{k\sigma(j+1)} \right) =$$

$$2 \sum_{j=1}^{n} a_{*\sigma(j)} a_{*\sigma(j+1)},$$

where the first equality holds since the sum is taken over all elements $\rho(i), (i = 1, \ldots, n)$ and the fourth since $a_{k\sigma(0)} = a_{k\sigma(n+1)} = 0$; $a_{*\sigma(j)} a_{*\sigma(j+1)}$ is defined as the *inner product* of the columns $\sigma(j)$ and $\sigma(j+1)$. Hence, we have to minimize the sum over the inner products of columns that are neighbors with respect to the permutation σ. If we define the distance d_{rs} between the columns r and s $(0 \leq r < s \leq n+1)$ as their inner product, which we multiply by -1 to create a minimization problem, that is,

$$d_{rs} = -2 a_{*r} a_{*s} = -2 \sum_{k=1}^{m} a_{kr} a_{ks}$$

then we obtain the problem of finding an undirected Hamiltonian path of minimum length between the columns 0 and $n+1$.

Similarly, (2) can be rewritten as the summation over the inner products of the rows. An analogous analysis leads to the problem of finding a minimum length Hamiltonian path between the rows 0 and $m+1$, where the distances are defined by

$$d_{rs} = -2 a_{r*} a_{s*} = -2 \sum_{j=1}^{n} a_{rj} a_{sj}.$$

We see that the problem of clustering a data array can be decomposed into two Hamiltonion path problems, one defined on the rows and one defined on the columns.

Section 5.3. Integer linear programming formulations

The asymmetric TSP.

Consider the asymmetric TSP with distance matrix d_{ij}. We will show how this problem can be modeled as an integer linear programming problem. To that end, we introduce decision variables x_{ij} $(1 \leq i, j \leq n)$, the outcome of which is interpreted as follows:

$$x_{ij} = \begin{cases} 1, & \text{if the arc from city } i \text{ to city } j \text{ is included in the tour} \\ 0, & \text{otherwise .} \end{cases}$$

Clearly, we need the constraints

$$\forall_{j \in V} \sum_{i=1}^{n} x_{ij} = 1, \tag{1}$$

and

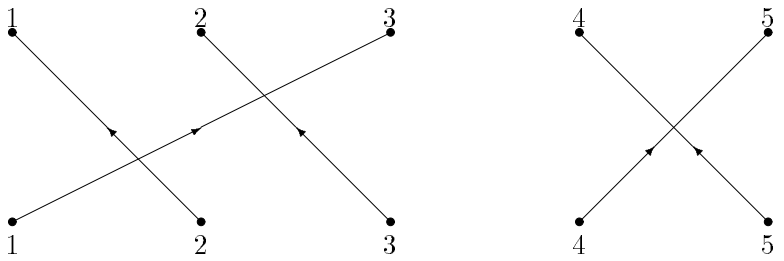
$$\forall_{i \in V} \sum_{j=1}^{n} x_{ij} = 1, \tag{2}$$

since we enter and leave each city exactly once. However, this is not enough. The problem with constraints (1) and (2) and the integrality constraints is a relaxation of the asymmetric TSP, since it allows for solutions that do not correspond to tours. This is illustrated by the following example.

Example.

	1	2	3	4	5
1		1	1	4	4
2	1		1	4	4
3	2	1		4	4
4	3	3	3		1
5	3	3	3	1	

Since the outdegrees and the indegrees of all vertices have to be equal to one, the problem with the integrality constraints and (1) and (2) is a minimum weight bipartite matching problem, with the rows corresponding to a set of vertices V_1 and the columns corresponding to a set of vertices V_2. The optimal solution of the example is simply found. It corresponds to two subtours, one containing the vertices $\{1, 2, 3\}$ and one containing $\{4, 5\}$.



We need constraints to prohibit subtours from occurring. The following restrictions are sufficient to limit the set of feasible solutions to tours:

$$\forall_{S \subset \{1,\ldots,n\}, \emptyset \neq S \neq V} \sum_{i,j \in S} x_{ij} \leq |S| - 1. \tag{3}$$

These restrictions force that each proper subset on k cities contains at most $k - 1$ arcs; subtours clearly do not satisfy this condition. The following theorem shows that adding this extra set of constraints leaves us with a correct formulation.

Theorem 5.1.
Each subset of arcs that satisfies the restrictions (1), (2), (3), and the integrality conditions forms a directed Hamiltonian circuit, and each directed Hamiltonian circuit corresponds to a feasible solution of the above formulation.

Proof.
Consider any integral solution that satisfies (1), (2), and (3). Due to restrictions (1) and (2), we have that each vertex has indegree 1 and outdegree 1. Hence, the graph consists of one or more Eulerian subgraphs. Suppose that there are two or more components, that is, the solution is not connected. Take one of the connected components, say $S \neq V$; this S does not satisfy (3), contrary to our assumptions.

It is easy to check that a Hamiltonian circuit satisfies all constraints. □

Up till now, we have not specified the objective function. Since we are looking for a tour of minimal length, we want that our objective function resembles the total length of the tour. To that end, we choose the objective function, which we want to minimize, equal to

$$\sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij}.$$

The symmetric TSP.
Consider the symmetric TSP with distance matrix d_e. This problem can be modeled as an integer linear programming problem as follows. For each edge $e \in E$, we introduce a decision variable x_e, the outcome of which is interpreted as follows:

$$x_e = \begin{cases} 1 & \text{if edge } e \text{ is in the tour ;} \\ 0 & \text{otherwise .} \end{cases}$$

The degree constraints are the following.

$$\forall_{i \in V} \sum_{e : e \supset i} x_e = 2. \tag{4}$$

The problem concerning the elimination of subtours arises in the symmetric case, too. The same solution, however, also works here, so we add the following constraints:

$$\forall_{S \subset \{1,\ldots,n\}, \emptyset \neq S \neq V} \sum_{e \subset S} x_e \leq |S| - 1. \tag{5}$$

Hence, each subgraph induced by S contains strictly less edges than vertices; we can prove in a similar fashion that the constraints (4) and (5) together with the integrality constraints yield a correct formulation. The objective function, which has to be minimized, is chosen equal to

$$\sum_{e \in E} d_e x_e.$$

Section 5.4. Approximation algorithms

Since no algorithm is known that solves the TSP to optimality in polynomial time, approximation algorithms have been developed to get quick and hopefully good solutions. These algorithms can roughly be divided into two classes: constructive algorithms and improvement algorithms. Constructive algorithms build a tour step by step, whereas improvement algorithms try to improve a given tour by exchanging edges in the tour with edges not in the tour. We provide some examples from both classes of algorithms.

Constructive approximation algorithms.

We discuss two algorithms for the symmetric TSP. In order to be able to say something about the relative performance of the algorithms, we assume that the length function satisfies the *triangle inequality*, that is, we have that $d_{ik} \leq d_{ij} + d_{jk}$ for all i, j, k in V.

The double tree algorithm (TT).

Algorithm TT consists of four phases. In the first three phases, we construct an Euler-cycle that we convert into a Hamilton-circuit in Phase 4.

Phase 1.

Construct a minimum spanning tree with respect to the distance function d.

Phase 2.

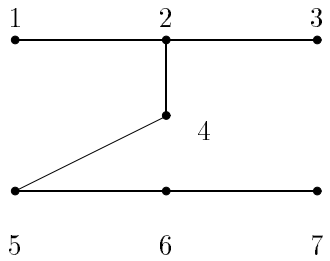
Double all edges in the tree to obtain an Eulerian graph.

Phase 3.

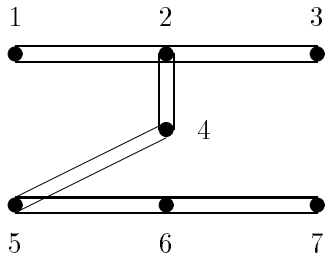
Determine an Euler-cycle in the Eulerian graph determined in Phase 2. Since the Eulerian graph is connected (it contains the minimum spanning tree as a subgraph), the cycle contains each vertex at least once.

Example

	1	2	3	4	5	6	7
1		1	2	2	3	4	5
2			1	1	3	3	4
3				2	3	2	3
4					1	2	3
5						1	2
6							1
7							



A minimum spanning tree



Doubling the tree

We now have to find an Euler-cycle in the graph depicted on the right-hand side. It is easily checked that 3212456765423 is an Euler-cycle.

Phase 4.
Convert the Euler-cycle into a Hamilton-circuit by applying *short-cuts*, as described below.

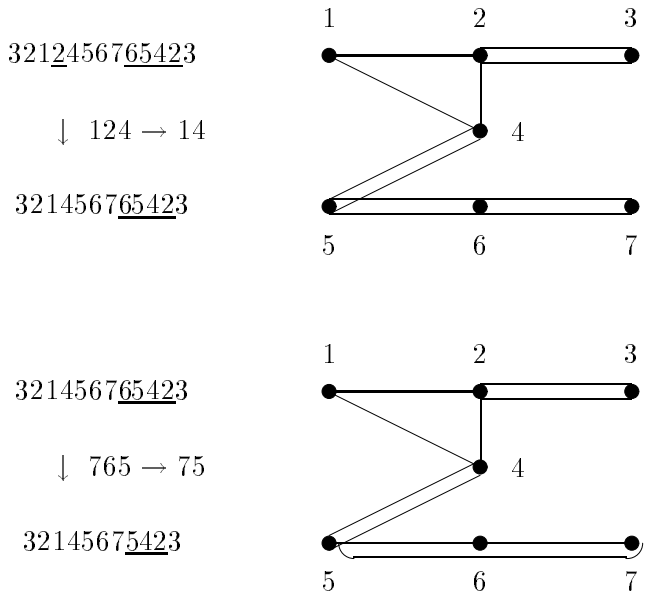
Shrinking a cycle by applying short-cuts.
Let the cities i, j, and k occur in the cycle in this order, that is, the edges $\{i, j\}$ and $\{j, k\}$ follow each other in the cycle. Replace these edges by the edge $\{i, k\}$, as is shown below.

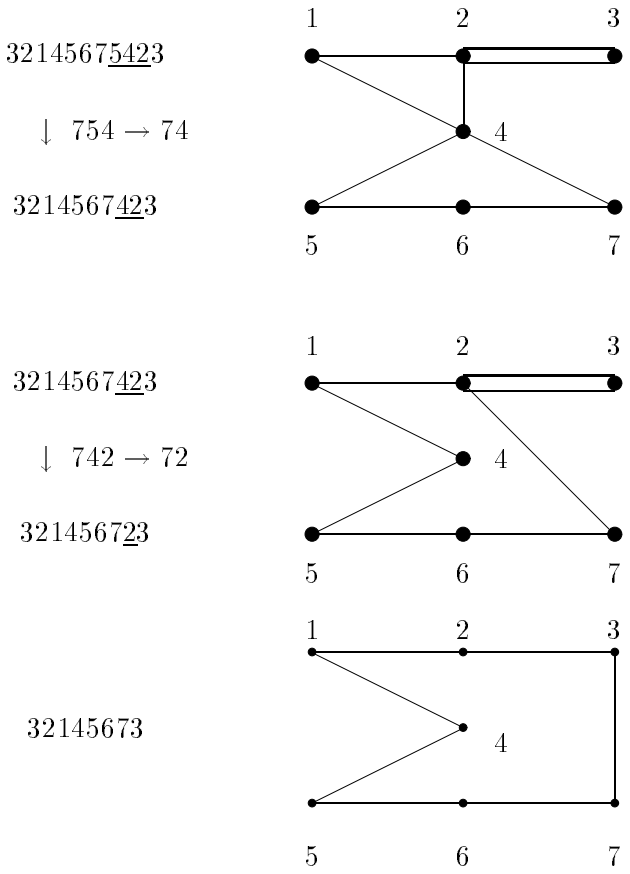
This operation can be performed on any cycle. Its effect is that a new cycle is constructed with one edge less; in this cycle, there is one vertex that is visited one time less in comparison with the previous cycle. Due to the assumption that the length function satisfies the triangle inequality, we have that applying a short-cut does not increase the length of the cycle.

Lemma 5.2 Let $G = (V, E)$ be a complete graph, and let $d : E \to R^+$ be a distance function on the edges, which satisfies the triangle inequality. Let C be a cycle in this graph with total length $d(C)$. If C' is a cycle constructed from C by applying short-cuts, then we have that the total length of C' is no more than $d(C)$.

We apply a short-cut on each second appearance of a vertex j. Therefore, each vertex remains in the cycle at least once. After the short-cut has been applied to all second appearances of the vertices, the cycle contains each vertex exactly once, that is, we have constructed a Hamilton-circuit.

Example continued.





The tour constructed by Algorithm TT

We now show that Algorithm TT will never produce a solution with length more than twice the length of an optimal solution. Hence, we say that Algorithm TT has *worst-case ratio* equal to 2.

Let z_{opt} denote the length of an optimal tour, and let z_{TT} denote the length of the tour constructed by algorithm TT. Finally, let z_T denote the length of a minimum spanning tree. We first prove that $z_T \leq z_{\text{opt}}$; we then complete the proof by showing that $z_{TT} \leq 2 * z_T$.

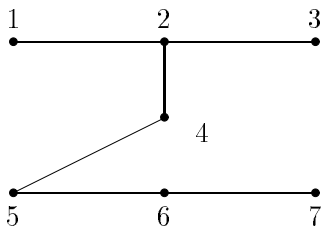
1. Consider any optimal tour. If we delete an arbitrary edge from it, then we obtain a Hamiltonian path. Since the length of any edge is nonnegative because of the triangle inequality, the length of this Hamilton path amounts to no more than z_{opt}. Since a Hamiltonian path is a special case of a spanning tree, we have that its length amounts to at least z_T. Concluding, we have that $z_T \leq z_{\text{opt}}$.

2. The total length of the edges in the Eulerian graph that is constructed by doubling the minimum spanning tree is equal to $2 * z_T$. From Lemma 5.2, it follows that the length z_{TT} of the Hamiltonian circuit that is obtained by applying short-cuts amounts to no more than the length of the Eulerian cycle, which is equal to $2 * z_T$.

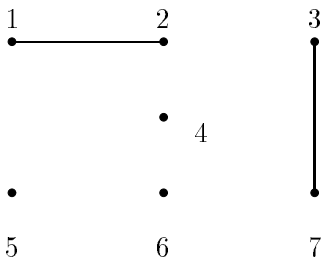
Combining both results, we get $z_{TT} \leq 2 * z_T \leq 2 * z_{\text{opt}}$.

The tree-matching algorithm (TM).
From the analysis above, it follows immediately that, if we want to improve on our worst-case ratio, then we should try to decrease the length of the Eulerian cycle. Recalling that a graph is Eulerian if and only if it is connected with even degree in each point, it seems obvious to start with a minimum spanning tree to make sure that the graph is connected. The only problem left is to take care of the vertices with odd degree; note that the number of vertices with odd degree is even. If we limit our attention to the set V_0 containing the vertices with odd degree, then we see that we can get even degree in each of these vertices by adding a perfect matching M on these vertices with minimum length. The Phases 1, 3, and 4 in Algorithm TM are identical to the corresponding phases in Algorithm TT.

Example continued.

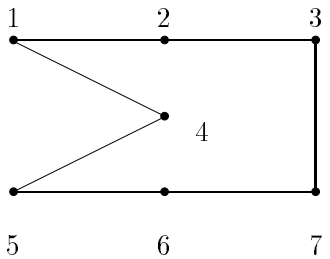


A minimum spanning tree



A perfect matching of minimum lenght

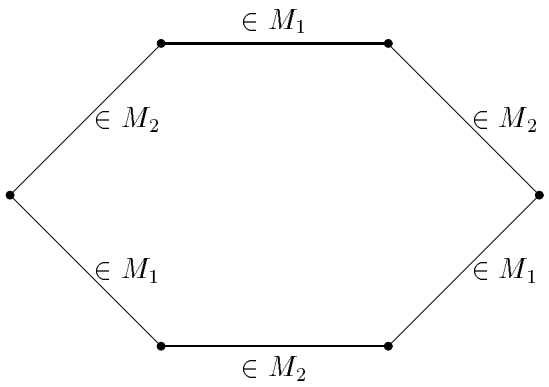
The combination of the tree and the matching depicted above yields an Eulerian graph that contains the Euler-cycle 1237654$\underline{2}$1. Applying the short-cut in which the edges $\{1,2\}$ and $\{2,4\}$ are replaced with the edge $\{1,4\}$ yields the Hamiltonian cycle that is depicted below. Note that for this example algorithm TT and algorithm TM find the same tour; this is a coincidence.



The tour constructed by Algorithm TM

This small change with respect to algorithm TT results in a better worst-case behaviour. For any instance of the TSP (with triangle-inequality), algorithm TM constructs a tour with length no more than 3/2 times the length of an optimal tour. To prove this, it suffices to show that the length z_M of the matching M is no more than $\frac{1}{2}$ times z_{opt}. Namely, then $z_{TM} \leq z_T + z_M \leq z_{\text{opt}} + \frac{1}{2} z_{\text{opt}} = \frac{3}{2} * z_{\text{opt}}$.

The proof makes use of the shrinking algorithm again. Consider any optimal tour with value z_{opt}. Apply short-cuts such that only the vertices in V_0 remain in the cycle; this yields a circuit C on the vertices in V_0 with length no more than z_{opt}. C can be partitioned into two complete matchings M_1 and M_2 on V_0 by 'walking' along the circuit and putting the first edge in M_1, the second edge in M_2, the third edge in M_1, etc., as is shown in the figure below. Note that the circuit contains an even number of edges, because $|V_0|$ is even.



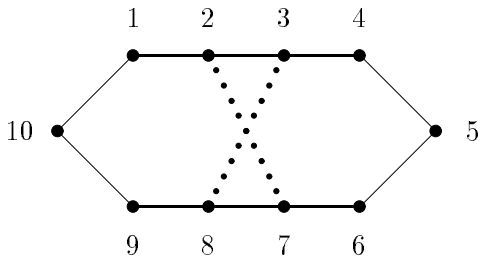
Since M_1 and M_2 are complete matchings on V_0, we have that $d(M) \leq d(M_1)$ and $d(M) \leq d(M_2)$. Hence, we have that $2 * d(M) \leq d(M_1) + d(M_2) = d(C) \leq z_{\text{opt}}$, which was to be proved.

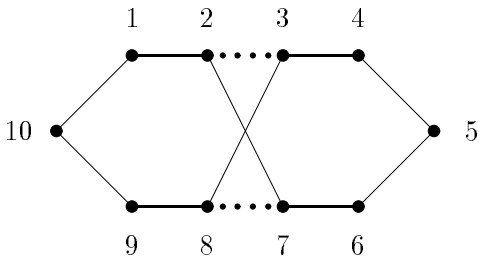
Note that for both algorithms our analysis of the worst-case ratio depends heavily on the assumption that the triangle inequality holds. It can be shown that, in case the triangle inequality fails to hold, the worst-case ratio is no longer fixed, that is, for any constant c an instance of the TSP can be constructed such that the length of the tour constructed by the algorithm is more than c times as long as the length of the optimal tour.

Improvement algorithms.
Suppose that we have a solution, that is, a tour, for a certain instance of the symmetric TSP. It may have been constructed by the previous algorithms or it may be just a random tour. An improvement algorithm tries to replace a subset of the edges of the tour by a new set of edges with smaller total length such that a new, shorter, tour results. We give four variants of improvement algorithms for the TSP.

2-edge-exchange (2-opt).
An iteration of 2-edge-exchange proceeds as follows. Take two edges that are not adjacent, say $\{i_1, j_1\}$ and $\{i_2, j_2\}$. Remove them from the tour. Let the remainder of the tour consist of the paths with endvertices i_1 and i_2 and endvertices j_1 and j_2, respectively. There are two ways to connect these two paths to a tour again: by reinserting the edges $\{i_1, j_1\}$ and $\{i_2, j_2\}$, which yields the old tour, and by inserting the edges $\{i_1, j_2\}$ and $\{i_2, j_1\}$. Choose the pair of edges with smaller cumulative length, that is, change the original tour only if you gain by doing so. Note that the insertion of the edges $\{i_1, i_2\}$ and $\{j_1, j_2\}$ creates a graph consisting of two subtours.

Example

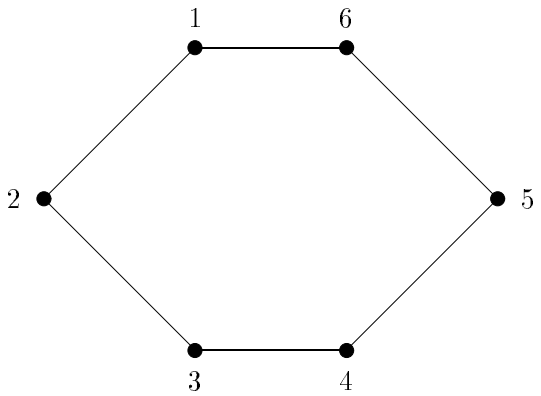




Remove the edges $\{2,3\}$ and $\{7,8\}$. To get a tour again, the edges $\{2,7\}$ and $\{3,8\}$ must be inserted. This is done only if the latter two edges have smaller cumulative length. Note that the addition of the edges $\{2,8\}$ and $\{3,7\}$ would split the tour into two subtours.

The 2-edge-exchange algorithm, or 2-opt for short, proceeds by performing edge exchanges until no pair of edges can be replaced by a better pair. The final tour is called 2-optimal. There is no guarantee at all that this algorithm ends after a polynomial number of steps. In practice, however, even if we start with a random tour, it stops after very few iterations. The typical number of iterations is quadratic in the number of cities n. Note that it takes $O(n^2)$ time to verify whether a tour is 2-optimal.

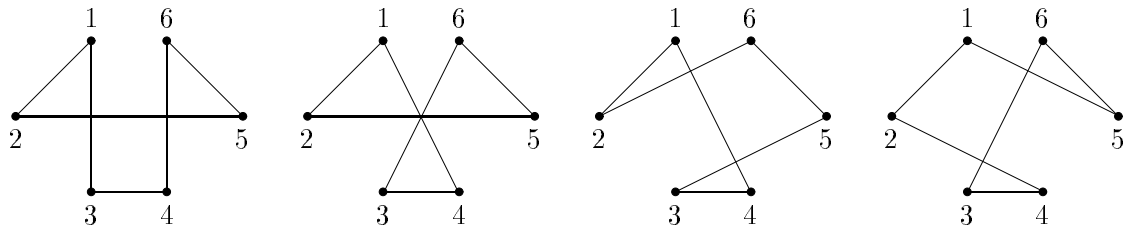
3-edge-exchange (3-opt).
The 2-opt procedure can be generalized by considering larger sets of edges, though the procedure becomes more and more involved then. In practice, we use exchanges of three edges at most at a time. An iteration is described as follows. Take three arbitrary edges, and try to replace them by three new edges, thus creating a new tour. In general, if none of the three edges are adjacent, then there are four candidate triples of edges that may replace the original triple. Consider the tour depicted below.



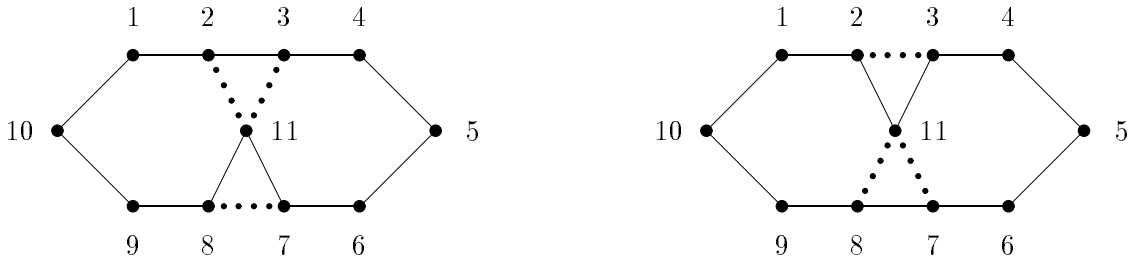
Remove the edges $\{1,6\},\{2,3\},\{4,5\}$. The four triples that can replace these edges are:

(1) $\{1,3\},\{2,5\}$ and $\{4,6\}$;
(2) $\{1,4\},\{2,5\}$ and $\{3,6\}$;
(3) $\{1,4\},\{2,6\}$ and $\{3,5\}$;
(4) $\{1,5\},\{2,4\}$ and $\{3,6\}$.

The corresponding tours are depicted below.



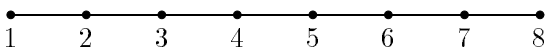
If two of the edges are adjacent, then there is only one way to exchange the edges. This case boils down to the case that one vertex is deleted from the tour and reinserted in another place. If only these types of exchanges are considered, then the algorithm is called vertex-reinsertion, or $2\frac{1}{2}$-opt.



Variable edge-exchange (Lin-Kernighan).
The last exchange algorithm described here is the most sophisticated one. Currently, this algorithm is known as the approximation algorithm that works best for practical TSP problems. It is named after the inventors: Lin and Kernighan. The variable edge-exchange algorithm does not fix the number of edges that are exchanged beforehand, but it determines this number dynamically. We present a simplified version.

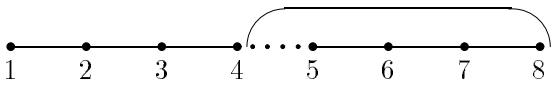
As with all improvement algorithms, an iteration of the Lin-Kernighan algorithm starts with a feasible tour T_0. Suppose that the cities in the tour are numbered in order of their appearance in T_0. Start with deleting an arbitrary edge, say $\{1, n\}$, from T_0; this yields a Hamiltonian path P_0 with endvertices 1 and n. Without loss of generality, vertex 1 is fixed as the head of the path. From among the edges $\{k, n\}$ $(k = 2, \ldots, n-1)$, the one with smallest length is added to the path; suppose this is $\{i, n\}$ (Lin and Kernighan use a somewhat more complicated decision rule). This results in a circuit containing the vertices $\{i, \ldots, n\}$ and a path from 1 to i. The edge $\{i, i+1\}$ is deleted to create a Hamiltonian path P_1; P_1 has 1 and $i+1$ as its endpoints. P_1 can be converted into a tour T_1 by adding the edge $\{1, i+1\}$. The tour T_1 is stored if it has smaller length than T_0. We repeat this process with P_1 instead of P_0 and vertex $i+1$ as the tail of P_1, that is, $i+1$ plays the role that n played in the previous iteration. From among the edges $\{i+1, k\}$ $(k = 2, \ldots, n$, we add the one with smallest length, say edge $\{i+1, j\}$. This results in a circuit and a path again that come together in j. There is only one edge adjacent to j that, after deletion, converts this construction into a *new* Hamiltonian path P_2. P_2 can be converted into a tour T_2 by connecting the endpoints. The tour T_2 is stored if it has smaller length than the tour that is currently best. This process is repeated until an edge that has already been deleted once is selected for addition again.

P_0



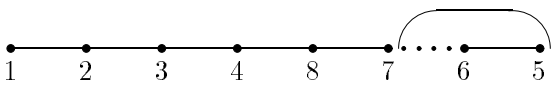
Suppose that the edge $\{4, 8\}$ is selected. To obtain a Hamiltonian path again, we remove the edge $\{4, 5\}$; this yields the Hamiltonian path P_1, which is depicted below. We can turn this path in the Hamiltonian cycle T_1 = 1 2 3 4 8 7 6 5 1.

P_1



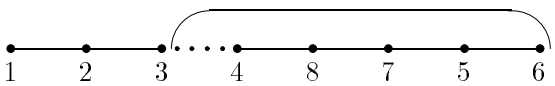
Vertex 5 is the endpoint of the path that is used by the algorithm. Suppose that in this iteration the edge $\{5, 7\}$ is selected; hence, we must remove the ege $\{7, 6\}$. This yields the Hamiltonian path P_2 from which the tour T_2 = 1 2 3 4 8 7 5 6 1 can be formed.

P_2



We can continue in this fashion by adding the edge $\{6, 3\}$, which implies that we remove $\{3, 4\}$. This yields the Hamiltonian path P_3 from which the tour T_3 = 1 2 3 6 5 7 8 4 1 can be formed.

P_3



Although there is no worst-case guarantee for this and the previous improvement algorithms other than the guarantee that we have for the initial tour, improvement algorithms, especially the last one, perform much better than constructive algorithms.

Chapter 6. Machine scheduling

Section 6.1. Problem definition

A company is specialized in drilling holes in hard materials like steel; it has m identical machines available to perform this process. An important client has an assignment for the company that needs to be performed as soon as possible. The assignment consists of drilling holes in a set of n pieces of material; in piece j ($j = 1, \ldots, n$), the client wants p_j holes to be drilled. Since the drilling of the holes is much more time-consuming than all side-activities, we may assume that it takes p_j units of time to process piece j. The planner of the company has the task to plan the use of the machines such that the client gets his processed pieces as quickly as possible; they form one truck-load, so we are not concerned with intermediate completion times: only the completion time of the last piece counts. He has to take into account the side-constraints that the machine can only drill one hole at a time and that the drilling process cannot be stopped before all p_j holes are drilled in piece j.

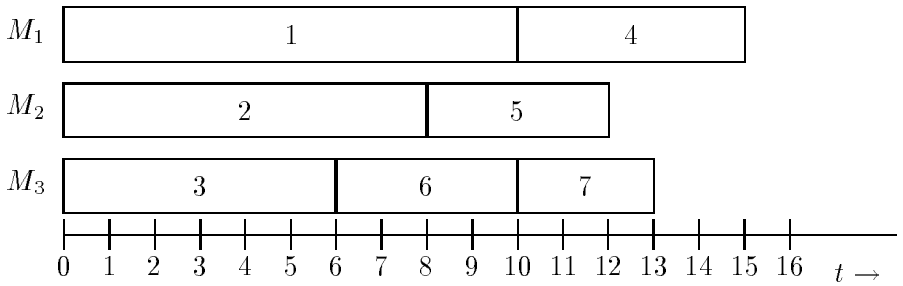
This is a typical problem in machine scheduling. It is modeled as follows. A set of n jobs has to be processed; for the processing of the jobs, there are m identical machines available from time zero onwards that can handle only one job at a time. The processing of job j ($j = 1, \ldots, n$) takes an *uninterrupted* period of p_j units of time, and it has to be executed by a single machine. We are looking for a feasible schedule, that is, an allocation of each job j to a time interval of length p_j on a machine such that no two jobs are processed by the same machine at the same time. Given a feasible schedule, we denote the *completion time* of job j ($j = 1, \ldots, n$) by C_j. The objective is to find a schedule in which the latest job finishes as soon as possible, i.e., we want to minimize the maximum completion time $C_{max} = \max_{j=1,\ldots,n} C_j$.

Example

There are 3 machines available for processing 7 jobs. The processing times are given by the following table.

j	1	2	3	4	5	6	7
p_j	10	8	6	5	4	4	3

A feasible schedule is visualized by a so-called Gantt-chart as follows.



M_1, M_2 and M_3 denote the machines. The jobs 1 and 4 are processed on machine M_1; jobs 2 and 5 are processed on M_2; jobs 3, 6, and 7 are processed on M_3. Machine M_1 finishes after 15 time units, M_2 after 12 time units, and M_3 after 13 time units. Hence, we have that $C_{max} = C_4 = 15$ in this schedule. A better schedule is obtained by swapping jobs 4 and 5 on the first two machines; this schedule finishes after 14 units of time. Since the total processing time of all jobs amounts to 40 and each processing time is integral, we know that the schedule with $C_{max} = 14$ is optimal.

Section 6.2. Minimizing the maximum completion time $C_{\max}$

Complexity

If there is only one machine available, then the problem is trivially solvable, since all jobs must be processed on the same machine. All we have to do is take care that the machine is never idle until all jobs have been processed.

If there are two or more machines available, then the problem becomes hard, since it contains the partition problem as a special case, which we show now. We take an arbitrary instance of partition and convert this instance to an equivalent instance of the 2-machine scheduling problem.

Consider any instance of partition: suppose that it has t integral weights $u_1, \ldots, u_t$ and $U = \frac{1}{2}\sum_{j=1}^{t} u_j$. We construct the following instance of the 2-machine scheduling problem: there are $n = t$ jobs with processing times $p_j = u_j$ $(j = 1, \ldots, t)$.

Theorem 6.1.
There exists a feasible schedule with $C_{\max} \leq U$ if and only if the partition problem has an affirmative answer.

Proof.
First, suppose that the partition problem has an affirmative answer, that is, there is a set $S \subset \{1, \ldots, t\}$ such that $\sum_{j \in S} u_j = U$. From this solution we construct the following solution to the 2-machine scheduling problem. The jobs corresponding to S are executed by machine 1 and the other jobs are executed by machine 2. Hence, we have that $\sum_{j \in S} p_j = \sum_{j \in S} u_j = U$ and $\sum_{j \notin S} p_j = \sum_{j \notin S} u_j = U$. This implies that both machines are finished at time U, and therefore $C_{\max} = U$.

Conversely, suppose that there exists a solution to the machine scheduling problem with $C_{\max} \leq U$. Since the total processing time amounts to $2U$, we have that both machines must finish exactly at time U. Define S as the set containing the indices corresponding to jobs that are executed by machine 1; this set S constitutes a solution to the partition problem, since $\sum_{j \in S} u_j = \sum_{j \in S} p_j = U$. □

Note that the size of the instance of the two-machine problem is polynomially bounded in the size of the instance of the partition problem (in fact, both sizes are equal); hence, we can state that the two-machine problem is as hard as the partition problem.

It is fairly easy to see that the problem with $m = 2$ is a special case of the problem with $m > 2$. For each instance $(n, p_1, \ldots, p_n)$ of the 2-machine problem, we construct an instance of the m-machine problem by adding $m - 2$ jobs of length $\frac{1}{2}\sum_{j=1}^{n} p_j = P$. Clearly, the following theorem holds.

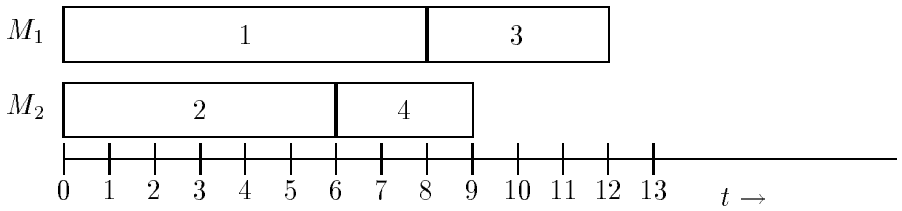
Theorem 6.2.
There is a solution of the 2-machine problem with $C_{\max} = P$ if and only if there is a solution with $C_{\max} = P$ of the m-machine problem (with $m > 2$). □

Since there is little hope for finding a polynomial algorithm to solve the 2-machine problem, we will resort to solving this problem by implicit enumeration by means of dynamic programming and to applying approximation algorithms to find a hopefully good solution in polynomial time.

A dynamic programming algorithm for the 2-machine problem

The idea behind our dynamic programming algorithm is the following. We introduce boolean variables $f_k(t_1, t_2)$ that are to signal whether it is possible to process jobs $1, \ldots, k$ such that the machines are ready at times t_1 and t_2, respectively. Since it makes no sense to leave a machine idle as long as there are unprocessed jobs available, this boils down to the question whether there exists a partition of the first k jobs such that the workload of machine 1 and 2 amounts to no more than t_1 and t_2, respectively.

Example



Hence, $f_4(12, 10) = \text{true}$, $f_4(9, 12) = f_4(12, 9) = \text{true}$, and $f_k(11, 9) = \text{false}$.

The calculation of the values $f_k(t_1, t_2)$ starts with the initialization $f_0(t_1, t_2) = \text{true}$ if $t_1, t_2 \geq 0$ and $f_0(t_1, t_2) = \text{false}$, otherwise. In the k-th iterative step, the values $f_k(t_1, t_2)$ are calculated from the values $f_{k-1}(t_1, t_2)$ by the recursion

$$f_k(t_1, t_2) = \{f_{k-1}(t_1 - p_k, t_2) \vee f_{k-1}(t_1, t_2 - p_k)\}.$$

Finally,

$$C_{\max} = \min\{t | f_n(t, t) = \text{true}\}.$$

The number of variables amounts to $n \times (\sum_{j=1}^{n} p_j)^2$. Since each variable can be calculated by an elementary operation involving only two variables, the number of elementary steps performed by the algorithm is of the same size.

We can implement our dynamic programming algorithm in a smarter way by paying only attention to the boolean variables $f_k(t_1, t_2)$ with $t_1 + t_2 = \sum_{j=1}^{k} p_j$; given these values, we can easily obtain the values of $f_k(t_1, t_2)$ with $t_1 + t_2 > \sum_{j=1}^{k} p_j$ (in fact, we do not even need these values). The calculation of the $f_k(t_1, t_2)$ values proceeds as follows. We initialize by putting $f_0(t_1, t_2) = \text{true}$ if $(t_1, t_2) = (0, 0)$ and $f_0(t_1, t_2) = \text{false}$, otherwise. The recursive step remains the same, and $C_{\max}$ is determined as

$$C_{\max} = \min\{t \geq (\sum_{j=1}^{n} p_j)/2 | f_n(t, \sum_{j=1}^{n} p_j - t) = \text{true}\}.$$

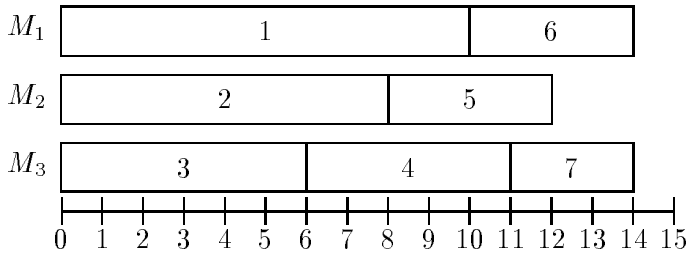
Since the number of variables has decreased to $n \times (\sum_{j=1}^{n} p_j)$, and we still can determine the value of each variable in constant time, the running time of this algorithm is $n \times (\sum_{j=1}^{n} p_j)$.

Approximation algorithms

The most common algorithms used to deal with scheduling and sequencing problems are the so-called *list scheduling* (LS) algorithms. List scheduling algorithms proceed by a two-phase greedy approach to generate solutions. In the first phase, the jobs are *listed*, that is, they are renumbered according to a certain order, for instance, to increasing processing times (SPT) or to decreasing processing times (LPT). In the second phase, the jobs are placed on the machines according to the order in which they occur in the list; the machine that is chosen to put job j on is the one with the smallest workload with respect to the jobs $1, \ldots, j-1$.

Example 7 jobs, 3 machines.

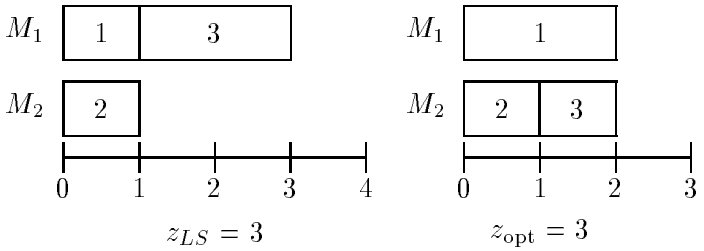
j	1	2	3	4	5	6	7
p_j	10	8	6	5	4	4	3



The question is of course: how bad can the solution get? For an arbitrary numbering of the jobs, the following instance is a bad one.

Example 3 jobs, 2 machines.

j	1	2	3
p_j	1	1	2



Hence, we have that $\frac{z_{LS}}{z_{\text{opt}}} = \frac{3}{2}$, and we know that we run the risk of finding a solution with an objective value that is 50% greater than necessary. The next theorem states that this is as bad as it can get.

Theorem 6.3.

For any ordering of the jobs, the list scheduling algorithm finds a schedule with makespan no more than 3/2 times the length of an optimal schedule.

Proof.

Consider an optimal schedule. Let C^i_{opt} denote the workload of machine i ($i = 1, 2$) in this schedule; without loss of generality, we assume that $z_{\text{opt}} = C^1_{\text{opt}} \geq C^2_{\text{opt}}$. Analogously, C^i_{LS} denotes the workload of machine i ($i = 1, 2$) in the solution found by Algorithm LS; again, we suppose without loss of generality that $z_{LS} = C^1_{LS} \geq C^2_{LS}$.

In our proof, we make use of two obvious lower bounds on z_{opt}; these are $z_{\text{opt}} \geq p_{\max} = \max_{1 \leq j \leq n} p_j$ and $z_{\text{opt}} \geq \frac{1}{2} \sum_{j=1}^{n} p_j$. Now consider the schedule obtained by Algorithm LS; let job i be the last job scheduled on M_1. We have that $p_i \geq C^1_{LS} - C^2_{LS}$; otherwise, machine M_2 would have been idle before job i started, and hence would have been selected by Algorithm

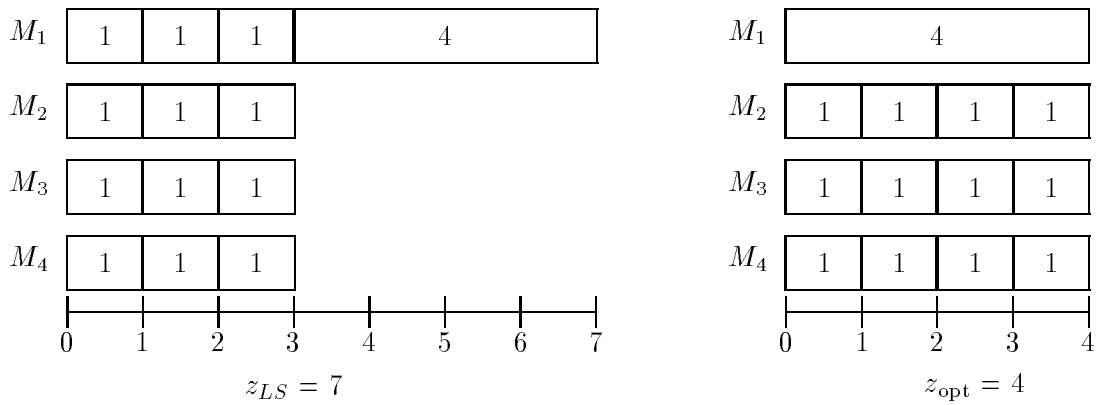
LS for processing job i. This implies that

$$z_{LS} = C^1_{LS} = \tfrac{1}{2}(C^1_{LS} + C^2_{LS}) + \tfrac{1}{2}(C^1_{LS} - C^2_{LS})$$

$$\leq \tfrac{1}{2}\sum_{j=1}^{n} p_j + \frac{1}{2}p_i \leq z_{\text{opt}} + \frac{1}{2}z_{\text{opt}} = \frac{3}{2}z_{\text{opt}}.$$

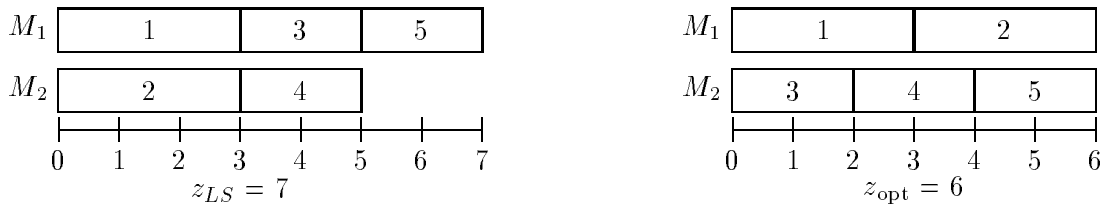
□

It can be shown that in case of m machines Algorithm LS always finds a schedule with makespan no more than $2 - \frac{1}{m}$ times the length of an optimal schedule, independent of the ordering of the jobs. The example below shows that this bound is tight for $m = 4$; it contains $m^2 - m$ jobs of length 1 and one job of length m. Note that in the picture below the length of the job is put inside the box representing it instead of its number.



From the example above, one might get the idea that it is possible to improve the worst-case bound by starting with the jobs with large processing times, that is, by ordering the jobs in the list according to decreasing processing times. A bad instance for the list scheduling algorithm in case the jobs are ordered according to decreasing processing times is the following one.

j	1	2	3	4	5
p_j	3	3	2	2	2



The next theorem shows that this is as bad as it can get in case of two machines.

Theorem 6.4.
The list scheduling algorithm in which the jobs are assigned to the machines in LPT order always finds a schedule with length at most $\frac{7}{6}$ times the length of an optimal schedule.

Proof.

Let z_{opt} denote the makespan of an optimal schedule; let C^i_{LPT} $(i = 1, 2)$ denote the workload of machines 1 and 2. Without loss of generality, let $z_{LPT} = C^1_{LPT} \geq C^2_{LPT}$; let i be the job that is processed last on M_1. We reduce the instance by deleting the jobs $i+1, \ldots, n$: this does not increase z_{opt} and it does not decrease $z_{LPT} = C^1_{LPT}$, and hence, it does not decrease the ratio z_{LPT}/z_{opt}. This implies that if we prove the bound for the reduced instance then it certainly holds for the original instance. Due to the choice of machine 1 to put job i on, we have that $p_i \geq C^1_{LPT} - C^2_{LPT}$. We further make use in our proof of the obvious inequalities $z_{\text{opt}} \geq \frac{1}{2}\sum_{j=1}^{i} p_j$ and $p_i \leq \frac{1}{i}\sum_{j=1}^{i} p_j$ (since i is the smallest job from among the first i jobs).

$$
\begin{aligned}
z_{LPT} = C^1_{LPT} &= \tfrac{1}{2}(C^1_{LPT} + C^2_{LPT}) + \tfrac{1}{2}(C^1_{LPT} - C^2_{LPT}) \\
&\leq \tfrac{1}{2}\sum_{j=1}^{i} p_j + \frac{1}{2}p_i \leq z_{\text{opt}} + \frac{1}{2}(\frac{1}{i}\sum_{j=1}^{i} p_j) \\
&= z_{\text{opt}} + \tfrac{1}{i}(\tfrac{1}{2}\sum_{j=1}^{i} p_j) \leq z_{\text{opt}} + \frac{1}{i}z_{\text{opt}} = (1 + \frac{1}{i}) \times z_{\text{opt}}.
\end{aligned}
$$

Therefore, the theorem holds for $i \geq 6$.

For $i \leq 4$, it is a matter of simple checking to show that the LPT ordering leads to an optimal schedule. We are left with checking the case $i = 5$. We may assume that job 5 finishes last; otherwise, we can delete job 5 from the instance without decreasing the worst-case ratio. Since one of the machines, say M_1, processes at least three jobs, the workload of M_1 amounts to at least $3p_5$, which implies that $p_5 \leq \frac{1}{3}z_{\text{opt}}$. Along the lines used in the proof above, we can prove that the theorem holds for $i = 5$, too. □

Section 6.3. Minimizing the sum of completion times $\sum C_j$

Minimizing the average completion time $\frac{1}{n}\sum_{j=1}^{n} C_j$ is a natural objective when jobs are identified with customers waiting in a queue. Of course, this is equivalent to minimizing $\sum_{j=1}^{n} C_j$, since n is given. If there is only one machine available for processing the jobs, that is, $m = 1$, the problem is solved by a simple sorting of the processing times.

Theorem 6.5.
If there is only one machine available, then $\sum_{j=1}^{n} C_j$ is minimized by scheduling the jobs in order of nondecreasing processing times.

Proof.
We will give two proofs: the first one is meant to just prove the theorem, whereas the second one is meant to provide some insight.

Proof one:
Consider any optimal order; suppose to the contrary that the jobs are not scheduled in order of nondecreasing processing times. Then there are two adjacent jobs, say jobs j and k, such that $p_j > p_k$, whereas job j precedes job k. If we exchange the two jobs, then C_j increases by p_k and C_k decreases by p_j. This interchange decreases the value of the objective function by $p_j - p_k > 0$, since the completion times of all other jobs remain the same. This contradiction proves the theorem.

Proof two:
Suppose that the jobs are numbered according to their appearance on the machine. We then have that

$$\begin{aligned}\sum_{j=1}^{n} C_j &= C_1 + C_2 + ... + C_n \\ &= p_1 + (p_1 + p_2) + \ldots + (p_1 + p_2 + ... + p_n) \\ &= np_1 + (n-1)p_2 + ... + p_n.\end{aligned}$$

We see that the processing time of the job that occupies the kth position in the schedule is counted $(n + 1 - k)$ times. Hence, we can minimize the value of the objective function by scheduling the jobs in order of nondecreasing processing times. □

If there are two or more machines available, then it is easily shown that the processing time of the job that is scheduled on the kth position on machine M_i is counted $(m_i + 1 - k)$ times in the value of the objective function, where m_i denotes the number of jobs that are executed by M_i. This observation leads to the following theorem.

Theorem 6.6.
In any optimal schedule, each machine executes either $\lfloor n/m \rfloor$ or $\lceil n/m \rceil$ jobs.

Proof.
Suppose to the contrary that there exists an optimal schedule in which some machine executes at least two jobs more than some other machine; without loss of generality, we suppose that these machines are M_1 and M_2, which implies that $m_1 \geq m_2 + 2$. Let job i be the job that is executed first by M_1. We move job i from the first position on M_1 to the first position on M_2, and we evaluate the change of the objective function. In the new value of the objective function, p_i is counted only $m_2 + 1$ times instead of m_1 times. Since the number of times that the processing times of all other jobs are counted in the new value of the objective function has not changed, moving job i has decreased the value of the objective function by $(m_1 - m_2 - 1)p_i$; this clearly contradicts the optimality of the original schedule. Therefore, in any optimal schedule each machine executes at most one job more or less than any other machine; this implies the division of the jobs over the machines stated in the theorem. □

Now that we have characterized the number of jobs executed by each machine, we can assign the jobs to the positions on the machines. Since the processing time of any job that is processed last on any machine is counted only once in the value of the objective function, it is optimal to assign the m longest jobs to these positions; it does not matter to which machine they are assigned, as long as they are executed last. Similarly, it is optimal to assign the next m longest jobs to the last but one positions on the machines, and so on.

Section 6.4. Minimizing the sum of the weighted completion times $\sum w_j C_j$

In the previous section, we have considered the problem of minimizing $\sum_{j=1}^{n} C_j$. In general, however, we will not consider each job to be equally important. We can express this difference in importance by attaching a *weight* w_j to job j $(j = 1, \ldots, n)$; the objective is now to minimize $\sum_{j=1}^{n} w_j C_j$. Usually, the weights are assumed to be positive integers.
In case of a single machine, we have that $\sum_{j=1}^{n} w_j C_j$ is minized by scheduling the jobs according to nonincreasing w_j/p_j ratios without unnecessary idle time; this is called Smith's rule.

The correctness of this rule can be proven in a fashion similar to the first proof of Theorem 6.5.

In case of more than one machine, the above result implies that given the assignment of jobs to machines we can determine an optimal schedule for this assignment by processing the jobs in order of nonincreasing w_j/p_j ratio on each machine. Unfortunately, there is no polynomial algorithm known that finds the optimal assignment of jobs to machines. In fact, we can show that partition is a special case of the problem of minimizing the sum of the weighted completion times on two machines.

Consider any instance of partition: suppose it has t integral weights $u_1, \ldots, u_t$ and $U = \frac{1}{2}\sum_{j=1}^{t} u_j$. We construct the following instance of the 2-machine scheduling problem: there are $n = t$ jobs with processing times and weights $p_j = w_j = u_j$ $(j = 1, \ldots, n)$.

Theorem 6.7
The partition problem has an affirmative answer if and only if there is a solution to the corresponding scheduling problem with value no more than $\sum_{i \leq j} u_i u_j - U^2$.

Proof.
First of all, note that the w_j/p_j ratio of all jobs is equal; this implies that the value of the objective function does not depend on the ordering of the jobs on the machines.

Consider any assignment of the jobs to the machines: suppose that the workloads assigned to M_1 and M_2 amount to $U + \delta$ and $U - \delta$, respectively. We show that the value of the objective function for the schedule corresponding to this assignment amounts to $\sum_{i \leq j} u_i u_j - U^2 + \delta^2$.

First, suppose that $\delta = U$, that is, all jobs have been assigned to M_1 and none to M_2. A straightforward calculation shows that the corresponding schedule has $\sum_{j=1}^{n} w_j C_j = \sum_{i \leq j} u_i u_j$. Now suppose that $\delta < U$, that is, M_2 processes a set S of jobs. If we move all jobs in S from M_2 to M_1 to be processed by M_1 immediately after all jobs that are not in S, then we increase the completion time of each job in S by $U + \delta$. We have just computed that the newly obtained schedule has $\sum_{j=1}^{n} w_j C_j = \sum_{i \leq j} u_i u_j$; clearly, the value of the objective function of the original schedule must have been equal to

$$\begin{aligned}
\sum_{j=1}^{n} w_j C_j &= \sum_{i \leq j} u_i u_j - (U + \delta) \sum_{j \in S} w_j \\
&= \sum_{i \leq j} u_i u_j - (U + \delta)(U - \delta) \\
&= (\sum_{i \leq j} u_i u_j - U^2) + \delta^2.
\end{aligned}$$

Proving the theorem has now become straightforward. Suppose that the partition problem has an affirmative answer; let $S \subset \{1, \ldots, t\}$ be such that $\sum_{i \in S} u_i = U$. Then we can determine a schedule with $\sum_{j=1}^{n} w_j C_j = \sum_{i \leq j} u_i u_j - U^2$ by assigning the jobs in S to M_1 and all other jobs to M_2. Conversely, a schedule with $\sum_{j=1}^{n} w_j C_j = \sum_{i \leq j} u_i u_j - U^2$ must be such that the workload on both machines is equal to U, and a subset of the weights that leads to an affirmative answer to partition is readily obtained. □

Since the size of the input of the instance of the problem of minimizing $\sum_{j=1}^{n} w_j C_j$ on two

machines that we constructed is polynomial in the size of the input of the instance of partition, we have that the problem of minimizing $\sum_{j=1}^{n} w_j C_j$ on two machines is at least as hard as the problem partition.