Combinatorial optimisation for telecommunications
Lecture notes

MASCOTTE CNRS-INRIA-UNSA

November 8, 2012

Contents

Chapter 1

Basic concepts

All the definitions given in this section are mostly standard and may be found in several books on graph theory like [1, 2, 3].

1.1 Graphs

A *graph* G is a pair (V,E) of sets satisfying $E \subset [V]^2$, where $[V]^2$ denotes the set of all 2-element subsets of V. We also assume tacitly that $V \cap E = \emptyset$. The elements of V are the *vertices* of the graph G and the elements of E are its *edges*. The vertex set of a graph G is referred to as $V(G)$ and its edge set as $E(G)$. An edge $\{x,y\}$ is usually written as xy. A vertex v is *incident* with an edge e if $v \in e$. The two vertices incident with an edge are its *endvertices*. An edge is said to *join* or *link* its two endvertices. Note that in our definition of graphs, there is no *loops* (edges whose endvertices are equal) nor *multiple edges* (two edges with the same endvertices).

Sometimes we will need to allow multiple edges. So we need the notion of *multigraph* which generalises the one of graph. A *multigraph* G is a pair (V,E) where V is the vertex set and E is a collection of elements of $[V]^2$. In a multigraph G, we say that xy is an edge of *multiplicity* m if there are m edges with endvertices x and y. We write $\mu(x,y)$ for the multiplicity of xy, and write $\mu(G)$ for the maximum of the edges multiplicities in G.

The *complement* of a graph $G = (V,E)$ is the graph $\overline{G}$ with vertex set V and edge set $[V]^2 \setminus E$. A graph is *empty* if it has no edges. A graph is *complete* of for all pair of distinct vertices u,v, $\{u,v\}$ is an edge. The complete graph on n vertices is denoted K_n. Trivially, the complement of an empty graph is a complete graph.

A *subgraph* of a graph G is a graph H such that $V(H) \subset V(G)$ and $E(H) \subset E(G)$. Note that since H is a graph we have $E(H) \subset E(G) \cap [V(H)]^2$. If H contains all the edges of G between vertices of $V(H)$, that is $E(H) = E(G) \cap [V(H)]^2$, then H is the subgraph *induced* by $V(H) = S$. It is denoted $G\langle S\rangle$. The notion of *submultigraph* and *induced submultigraph* are defined similarly. If S is a set of vertices, we denote by $G - S$ the (multi)graph induced by $V(G) \setminus S$. For simplicity, we write $G - v$ rather than $G - \{v\}$. For a collection F of elements of $[V^2]$, we write $G \setminus F = (V(G), E(G) \setminus F)$ and $G \cup F = (V(G), E(G) \cup F)$. As above $G \setminus \{e\}$ and $G \cup \{e\}$ are abbreviated to $G \setminus e$ and $G \cup e$ respectively. If H is a subgraph of G, we say that G is a *supergraph* of H.

Let G be a multigraph. When two vertices are the endvertices of an edge, they are *adjacent* and are *neighbours*. The set of all neighbours of a vertex v in G is the *neighbourhood* of G and is denoted $N_G(v)$, or simply $N(v)$. The *degree* $d_G(v) = d(v)$ of a vertex is the number of edges to which it is incident. If G is a graph, then this is equal to the number of neighbours of v.

Proposition 1.1. *Let $G = (V,E)$ be a multigraph. Then*

$$\sum_{v \in V} d(v) = 2|E|.$$

Proof. By counting *inc* the number of edge-vertex incidence in G. On the one hand, every edge has exactly two endvertices, so $inc = 2|E|$. On the other hand, every vertex v is an ednvertex of $d(v)$ edges, so $inc = \sum_{v \in V} d(v)$. □

The *maximum degree* of G is $\Delta(G) = \max\{d_G(v) \mid v \in V(G)\}$. The *minimum degree* of G is $\delta(G) = \max\{d_G(v) \mid v \in V(G)\}$. If the graph G is clearly understood, we often write Δ and δ instead of $\Delta(G)$ and $\delta(G)$. A graph is *k-regular* if every vertex has degree k. The *average degree* of G is $Ad(G) = \frac{1}{|V(G)|}\sum_{v \in V(G)} d(v) = \frac{2|E(G)|}{|V(G)|}$. The *maximum average degree* of G is $Mad(G) = \max\{Ad(H) \mid H \text{ is a subgraph of } G\}$.

Let G be a graph. A *stable set* or *independent set* in G is a set of pairwise non-adjacent vertices. In other words, a set S is stable if $G\langle S\rangle$ is empty. The *stability number* of G, denoted $\alpha(G)$ is the maximum cardinality of a stable set in G. Conversely, a *clique* in G is a set of pairwise adjacent vertices. In other words, a set S is a clique if $G\langle S\rangle$ is a complete graph. The *clique number* of G, denoted $\omega(G)$ is the maximum cardinality of a stable set in G.

1.2 Digraphs

A *multidigraph* D is a pair $(V(D),E(D))$ of disjoint sets (of *vertices* and *arcs*) together with two maps $tail : E(D) \to V(D)$ and $head : E(D) \to V(D)$ assigning to every arc e a *tail*, $tail(e)$, and a *head*, $head(e)$. The tail and the head of an arc are its *endvertices*. An arc with tail u and head v is denoted by uv and is said to *leave* u and to *enter* v; we say that u *dominates* v and write $u \to v$; we also say that u and v are *adjacent*. Note that a directed multidigraph may have several arcs with same tail and same head. Such arcs are called *multiple arcs*. A *multidigraph* without multiple arcs is a *digraph*. It can be seen as a pair (V,E) with a $E \subset V^2$. An arc whose head and tail are equal is a *loop*. All the digraphs we will consider in this monograph have no loops.

The multigraph G *underlying* a multidigraph D is the multigraph obtained from D by replacing each arc by an edge. Note that the multigraph underlying a digraph may not be a graph: there are edges uv of multiplicity 2 whenever uv and vu are arcs of D. *Subdigraphs* and *submultidigraphs* are defined similarly to subgraphs and submultigraphs.

Let D be a multidigraph. If uv is an arc, we say that u is an *inneighbour* of v and that v is an *outneighbour* of u. The *outneighbourhood* of v in D, is the set $N_D^+(v) = N^+(v)$ of outneighbours of v in G. Similarly, the *inneighbourhood* of v in D, is the set $N_D^-(v) = N^-(v)$ of inneighbours of v in G. The *outdegree* of a vertex v is the number $d_D^+(v) = d^+(v)$ of arcs leaving v and the

indegree of v is the number $d_D^-(v) = d^-(v)$ of arcs entering v. Note that if D is a digraph then $d^+(v) = |N^+(v)|$ and $d^-(v) = |N^-(v)|$. The *degree* of a vertex v is $d(v) = d^-(v) + d^+(v)$. It corresponds to the degree of the vertex in the underlying multigraph.

Proposition 1.2. *Let $D = (V,E)$ be a digraph. Then*

$$\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v) = |E|.$$

The *maximum outdegree* of D is $\Delta^+(D) = \max\{d^+(v), v \in V(D)\}$, the *maximum indegree* of D is $\Delta^-(D) = \max\{d^-(v), v \in V(D)\}$, and the *maximum degree* of D is $\Delta(D) = \max\{d(v), v \in V(D)\}$. When D is clearly understood from the context, we often write Δ^+, Δ^- and Δ instead of $\Delta^+(D)$, $\Delta^-(D)$ and $\Delta(D)$ respectively.

The *converse* of the digraph $D = (V,E)$ is the digraph $\overline{D} = (V(D), \overline{E})$ where $\overline{E} = \{(v,u) \mid (u,v) \in E\}$. A digraph D is *symmetric* if $D = \overline{D}$.

An *orientation* of a graph G is a digraph D obtained by substituting each edge $\{x,y\}$ by exactly one of the two arcs (x,y) and (y,x). An *oriented graph* is an orientation of graph.

1.3 Walks, paths, cycles

Let G be a multigraph. A *walk* in G is a finite (non-empty) sequence $W = v_0 e_1 v_1 e_2 v_2 \dots e_k v_k$ alternating vertices and edges such that, for $1 \leq i \leq k$, v_{i-1} and v_i are the endvertices of e_i. The vertex v_0 is called *start* of W and v_k *terminus* of W. They both are *endvertices* of W. The vertices $v_i, 1 \leq i \leq k-1$ are the *internal vertices*. One says that W *links* v_0 to v_k and that W is a (v_0, v_k)-*walk*.

Let A and B be to set of vertices. An (A,B)-*path* is a path whose start is in A, whose end is in B and whose internal vertices are not in $A \cup B$. We usually abbreviate $(\{a\}, B)$-path to (a,B)-path, $(A, \{b\})$-path to (A,b)-path and $(\{a\}, \{b\})$-path to (a,b)-path.

If $W_1 = u_0 e_1 u_1 e_2 u_2 \dots e_p u_p$ and $W_2 = v_0 f_1 v_1 f_2 v_2 \dots f_q v_q$ are two walks such that $u_p = v_0$, the *concatenation* of W_1 and W_2 is the walk $u_0 e_1 u_1 e_2 u_2 \dots e_p u_p f_1 v_1 f_2 v_2 \dots f_q v_q$. The *concatenation* of k walks $W_1, \dots, W_k$ such that for all $1 \leq i \leq k-1$ the terminus of W_i is the start of W_{i+1} is then defined inductively as the concatenation of W_1 and the concatenation of $W_2, \dots, W_k$.

If G is a graph, then the walk W is entirely determined by the sequence of its vertices. Very often, we will then denote $W = (v_0, v_1, \dots, v_k)$. The *length* of W is k, which is its number of edges (with repetitions). A walk is said to be *even* (resp. *odd*) if its length is even (resp. odd).

A walk whose start and terminus are the same vertex is *closed*. A walk whose edges are all distinct is a *trail*. A closed trail is a *tour*. A walk whose vertices are all distinct is a *path* and a walk whose vertices are all distinct except the start and the terminus is a *cycle*. Observe that a path is necessarily a trail and a cycle is a tour.

A path may also be seen as a non-empty graph $P = (V,E)$ of the form $V = \{x_0, x_1, \dots, x_k\}$ and $E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k\}$ where the vertices x_i are all distinct. Similarly, a cycle may be seen as a non-empty graph $C = (V,E)$ of the form $V = \{x_0, x_1, \dots, x_k\}$ and $E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k, x_kx_0\}$ where the x_i are all distinct.

Proposition 1.3. *Let G be a multigraph.*

(i) There is a (u,v)-walk in G if and only if there is a (u,v)-path.

(ii) An edge is in a closed trail if and only if it is in a cycle.

(iii) There is an odd closed walk, if and only if there is an odd cycle.

Proof. (i) Let $P = v_0e_1v_1e_2v_2\dots e_kv_k$ be a shortest (u,v)-walk. Then P is a path. Indeed suppose for a contradiction that there exists $i < j$ such that $v_i = v_j$. Then $v_0e_1\dots e_iv_ie_{j+1}\dots e_kv_k$ is a (u,v)-walk shorter than P, a contradiction.

(ii) Let $C = ve_1v_1e_2v_2\dots e_kv$ be a shortest closed trail. Then C is a cycle. Indeed suppose for a contradiction that there exists $1 \leq i < j < k$ such that $v_i = v_j$. Then $v_0e_1\dots e_iv_ie_{j+1}\dots e_kv$ is a (u,v)-trail shorter than C, a contradiction.

(iii) Let $C = ve_1v_1e_2v_2\dots e_kv$ be a shortest odd closed walk. Then C is an odd cycle. Indeed suppose for a contradiction that there exists $1 \leq i < j < k$ such that $v_i = v_j$. Then $v_0e_1\dots e_iv_ie_{j+1}\dots e_kv$ and $v_ie_{i+1}\dots e_jv_j$ are shorter closed walks than C. But the sum of the lengths of these two walks is the length of C and so is odd. So, one of the lengths is odd, a contradiction.

□

The *distance* between two vertices u and v in a multigraph is the length of a shortest (u,v)-walk or $+\infty$ if such a walk does not exists. It is denoted $dist_G(u,v)$, or simply $dist(u,v)$ if G is clearly understood from the context. The proof of (i) in the above proposition shows that a shortest (u,v)-walk (if one exists) is a (u,v)-path.

The word "distance" is well chosen because *dist* is a distance in the mathematical sense, that is a binary relation which is *symmetric* (for all $u,v \in V(G)$, $dist(u,v) = dist(v,u)$) and which satisfies the *triangle inequality*: for all $u,v,w \in V(G)$, $dist(u,w) \leq dist(u,v) + dist(v,w)$. See Exercise 1.12.

In multidigraphs, a *directed walk* is a finite (non-empty) sequence $W = v_0e_1v_1e_2v_2\dots e_kv_k$ alternating vertices and arcs such that, for $1 \leq i \leq k$, v_{i-1} is the start and v_i the terminus of e_i. *Directed trail*, *directed tour*, *directed path* and *directed cycle* are then defined similarly to *trail*, *tour*, *path* and *cycle*. Clearly, Proposition 1.3 has its analog for digraphs. Its proof is left in Exercise 1.11.

Proposition 1.4. *Let D be a multidigraph.*

(i) There is a directed (u,v)-walk in D if and only if there is a directed (u,v)-path.

(ii) An edge is in a closed directed trail if and only if it is in a directed cycle.

(iii) There is an odd closed directed walk, if and only if there is an odd directed cycle.

The *distance* between two vertices in a multidigraph is defined analogously to the distance in a mutigraph. However, it is no more a distance in the mathematical sense because it is not symmetric. However it satisfies the triangle inequality. See Exercise 1.12.

1.4 Connectivity and trees

A graph G is *connected* if for any two vertices u, v, there exists a (u,v)-path in G.

Proposition 1.5. *Let G be a graph and x a vertex of G. The graph G is connected if and only if for any vertex u in G, there is a (u,x)-path.*

The *connected components* of a graph are its maximal connected subgraph.

A graph with no cycle is a *forest*. It is also said to be *acyclic*. A connected forest is a *tree*. The *leafs* of a tree T are the vertices of degree at most 1.

Proposition 1.6. *Let G be a graph. If $\delta(G) \geq 2$ then G has a cycle.*

Proof. Let $P = (v_1, v_2, \ldots, v_k)$ be a path of maximal length. Since v_1 has degree 2 it is adjacent to a vertex $w \neq v_2$. The vertex w is in P otherwise $(w, v_1, v_2, \ldots, v_k)$ would be a longer path than P. Thus $w = v_j$ for some $j > 2$ and so $(v_1, v_2, \ldots, v_j, v_1)$ is a cycle. □

Proposition 1.6 implies that every forest has at least one leaf. In fact, it implies that every forest has at least two leaves.

Corollary 1.7. *Every forest on at least two vertices has at least two leaves.*

Proof. By induction on the number of vertices, the result holding trivially for the two forests on two vertices.

Let F be a tree on n vertices, with $n \geq 3$. By Proposition 1.6, F has at least one leaf x. The graph $F - x$ is a forest on $n-1$ vertices. By the induction hypothesis, it has two leaves y_1 and y_2. One of these two vertices, say y, is not adjacent to x since $d(x) \leq 1$. Hence y is also a leaf of F. □

Corollary 1.8. *For every tree T we have $|E(T)| = |V(T)| - 1$.*

Proof. By induction on the number of vertices of T, the result holding trivially if T is the unique tree on one vertex (K_1).

Let T be a tree on at least two vertices. By Corollary 1.7, T has a leaf x. Since T is connected, x has degree at least one, so $d(x) = 1$. Thus, $|E(T-x)| = |E(T)| - 1$. By the induction hypothesis, $|E(T-x)| = |V(T-x)| - 1 = |V(T)| - 2$. Hence, $|E(T)| = |V(T)| - 1$. □

Proposition 1.9. *Let T be a graph. Then the following four statements are equivalent:*

(i) T is a tree;

(ii) for any two vertices u, v of T, there exists a unique (u,v)-path;

(iii) T is connected-minimal, i.e. T is connected and $T \setminus e$ is not connected for all $e \in E(T)$;

(iv) T is acyclic-maximal, i.e. T is acyclic but $T \cup xy$ has a cycle for any pair $\{x,y\}$ of non-adjacent vertices in T.

Proof. (i)⇒(ii): By the contrapositive. Suppose that there exist two distinct (u,v)-paths $P = (p_1, p_2, \ldots, p_k)$ and $Q = (q_1, q_2, \ldots, q_l)$. Let i be the smallest index such that $p_{i+1} \neq q_{i+1}$ and let j be the smallest integer greater than i such that $p_j \in \{q_{i+1}, q_{i+2}, \ldots, q_l\}$. Let j' be the index for which $q_{j'} = p_j$. Then $(p_i, p_{i+1}, \ldots, p_j, q_{j'-1}, q_{j'-2}, \ldots, q_i)$ is a cycle.

(ii)⇒(iii): If there exists a unique path between any two vertices, then T is connected. Let $e = xy$ be an edge. Then (x,y) is the unique (x,y)-path in T. Thus $T \setminus e$ contains no (x,y)-path and so T is not connected. Hence T is connected-minimal.

(iii)⇒(i): By the contrapositive. Suppose that T is not tree. If T is not connected then it is not connected-minimal. Thus we may assume that T is connected and so T contains a cycle C. Let e be an edge of C. Let us show that $T \setminus e$ is connected which implies that T is not connected-minimal. Let x and y be two vertices. Since T is connected there is an (x,y)-path P in T. If P does not contain e then it is also a path in $T - e$. If P contains e then replacing e by $C \setminus e$ in P, we obtain an (x,y)-walk in $T \setminus e$. By Proposition 1.3, there is an (x,y)-path in $T \setminus e$.

(i)⇒(iv): If T is a tree then it is acyclic. Let us show that it is acyclic-maximal. Let x and y be two non-adjacent vertices. Then in T there is an (x,y)-path P since T is connected. The concatenation of P and (y,x) is a cycle in $T \cup xy$.

(iv)⇒(i): By the contrapositive. Suppose that $T$is not a tree. If it is not acyclic then it is not acyclic-maximal. Thus we may assume that T is not connected. So there are two vertices x and y for which there is no (x,y)-path in T. Let us show that $T \cup xy$ is acyclic which implies that T is not acyclic-maximal. Indeed if there were a cycle C, then it must contain xy because T is acyclic. Then $C \setminus xy$ would be an (x,y)-path in T, a contradiction. □

A subgraph H of a graph G is *spanning* if $V(H) = V(G)$.

Corollary 1.10. *A graph G is connected if and only if it has a spanning tree.*

Proof. By induction on the number of edges of G. If G is connected-minimal, then by Proposition 1.9, G is a tree and thus a spanning tree of itself. If G is not connected-minimal, then by definition there is an edge e such that $G \setminus e$ is connected. By the induction hypothesis, $G \setminus e$ has a spanning tree which is also a spanning tree of G. □

1.5 Strong connectivity and handle decomposition

A digraph is *strongly connected* or *strong* if for any two vertices u, v there is a directed (u,v)-path. Observe that swapping u and v implies that there is also a directed (v,u)-path. The *strongly connected components* of a digraph G are its maximum strongly connected subgraphs.

The following proposition follows easily from the definition.

Proposition 1.11. *Let D be a strongly connected digraph. Then every arc is in a directed cycle.*

Proof. Let uv be an arc. Since D is strongly connected then there is a directed (v,u)-path in D. Its concatenation with (u,v) is a directed cycle containing uv. □

Definition 1.12. The *union* of two digraphs D_1 and D_2 is the digraph $D_1 \cup D_2$ defined by $V(D_1 \cup D_2 = V(D_1) \cup V(D_2)$ and $E(D_1 \cup D_2) = E(D_1) \cup E(D_2)$.

Let D be a digraph and H be a subdigraph of D. A *H-handle* is a directed path or cycle (all vertices are distinct except possibly the two endvertices) such that its endvertices are in $V(H)$ and its internal vertices are in $V(D) \setminus V(H)$. A *handle decomposition* of D is a sequence $(C, P_1, \ldots, P_k)$ such that:

- $C = D_0$ is a directed cycle;
- for all $1 \leq i \leq k$, P_i is a D_{i-1}-handle and $D_i = D_{i-1} \cup P_i$;
- $D_k = D$.

The following proposition follows easily from the definitions.

Proposition 1.13. *Let H be a strongly connected subdigraph of D. For any H-handle P, then $H \cup P$ is strongly connected.*

Proof. Left in Exercise 1.26 □

Since every strongly connected digraph contains a directed cycle (Proposition 1.11), an easy induction immediately yields the following.

Corollary 1.14. *Every digraph admitting a handle decomposition is strongly connected.*

The converse is also true: every strongly connected digraph admits a handle decomposition. In addition, it has a handle decomposition starting at any directed cycle.

Theorem 1.15. *Let D be a strongly connected digraph and C a directed cycle in D. Then D has a handle decomposition $(C, P_1, \ldots, P_k)$.*

Proof. Let H be the subdigraph of D that admits a handle decomposition $(C, P_1, \ldots P_k)$ with the maximum number of arcs. Since every arc xy in $E(D) \setminus E(H)$ with both endvertices in $V(H)$ is a H-handle, H is an induced subdigraph of D. Assume for a contradiction that $H \neq D$. Then $V(H) \neq V(D)$. Since D is strongly connected, there is an arc vw with $v \in V(D)$ and $w \in V(D) \setminus V(H)$. Since D is strongly connected, D contains a (w, H)-path P. Then, (v, w, P) is a H-handle in D, contradicting the maximality of H. □

1.6 Eulerian graphs

A trail in a graph G is *eulerian G* if it goes exactly once through every edge of G. A graph is *eulerian* if it has an eulerian tour.

Theorem 1.16 (Euler 1736)**.** *A connected graph is eulerian if and only if all its vertices have even degree.*

Proof. The condition can easily seen to be necessary. Indeed if a vertex appears k times (or $k+1$ if it appears as start and terminus) of an eulerian tour, it is incident to exactly $2k$ edges in the tour and so it has degree $2k$.

Let us now show that the condition is sufficient. The proof follows the lines of the following algorithm.

Algorithm 1.1.

1. Initialise $W := v$ for an arbitrary vertex v.
2. If all the edges of G are in W then return W.
3. If not an edge is not in $W = v_0e_1v_1 \dots e_lv_l$,
4. If an edge incident to v_l, say $e = v_lv_{l+1}$ is not in W, then $W := v_0e_1v_1 \dots e_lv_lev_{l+1}$; go to 2.
5. If not all the edges incident to v_l are in W. Since there is an even number of them, $v_0 = v_l$. Then G has an edge $e \notin W$ incident to a vertex v_i in W, for it is connected. Let $e = v_iu$ be this edge.
   $W := v_ie_{i+1}v_{i+1} \dots e_lv_le_1v_1 \dots e_iv_ieu$; go to 2.

□

1.7 Exercises

Exercise 1.1. Show that K_n, the complete graph on n vertices, has $\binom{n}{2}$ edges.

Exercise 1.2. Build a cubic graph with 11 vertices. (*cubic:* $d(v) = 3$ for all vertex v.)

Exercise 1.3. Show that every graph has two vertices of same degree.

Exercise 1.4. Let G be a graph on at least 4 vertices such that for every vertex v, $G - v$ is regular. Show that G is either a complete graph or an empty graph.

Exercise 1.5. Let n and k be two integers such that $n > k$ and H be a graph on n vertices. Show that if $|E(H)| > (k-1)(n-k/2)$ then H has a subgraph of minimum degree at least k.

Exercise 1.6 (Jealous husbands)**.**
Three jealous husbands and their wives want to cross a river. But they just have a small boat in which at most two persons can fit. None of the husbands would allow his wife to be with another man unless he is present. Draw the graph of all the possible distributions across the river and advice the walkers on the method to cross the river.

Exercise 1.7 (Dog, goat, cabbage)**.**
A man wants to cross a river with his dog, his goat and his (huge) cabbage. Unfortunately, the

man can cross the river with at most one of them. Furthermore, for obvious reasons, the man cannot leave alone on one bank neither the goat and the dog nor the cabbage and the goat. Draw the bipartite graph of all permissible situations. How does the man do to cross the river?

Exercise 1.8. Let u and v be two vertices of a graph G. Show that, if u and v have odd degree and all the other vertices have even degree, then there is a (u,v)-path in G.

Exercise 1.9. Show that in a graph two paths of maximum length have a vertex in common.

Exercise 1.10. Find what is wrong in the following statement: *An edge is in a closed trail if and only if it is in a cycle.*

Exercise 1.11. Show Proposition 1.4.

Exercise 1.12. 1) Show that of G is a multigraph then $dist_G$ is symmetric and satisfies the triangle inequality.

2) Show that of D is a multidigraph then $dist_D$ satisfies the triangle inequality but may be non-symmetric.

Exercise 1.13. Let D be a digraph without directed cycles. Show that D has a vertex with indegree zero.

Exercise 1.14. Let $G=(V,E)$ be a graph. Show the following.
(1) If $|E|\geq|V|$ then G contains a cycle.
(2) If $|E|\geq|V|+4$ then G contains two edge-disjoint cycles.

Exercise 1.15. Let G be a graph of minimum degree at least 3. Show that G contains an even cycle.

Exercise 1.16. Let G be a connected graph. Show that there exists an orientation of G such that the outdegree of every vertex is even if and only if G has an even number of edges.

Exercise 1.17. Let G be a graph. Its *diameter* is the maximum distance between two vertices.

1) Show that if G has a diameter at least 3 then its complement $\overline{G}$ has diameter at most 3.

2) Deduce that every self-complementary graph ($G=\overline{G}$) has diameter at most 3.

3) For $k=1,2,3$, give an example of a self-complementary graph with de diameter k.

Exercise 1.18. Let T be a tree on at least two vertices. Show that if T has no vertex of degree 2 then T has at least $|V(T)|/2+1$ leaves.

Exercise 1.19 (Helly property for trees)**.** Let $T_1,\ldots,T_k$ be subtrees if a tree T. Show that if $T_i\cap T_j\neq\emptyset$ for all i,j, then $\bigcap_{i=1}^{k}T_i\neq\emptyset$.

Exercise 1.20. Is the complement of a non-connected graph always connected?
Is the complement of a connected graph always non-connected?

Exercise 1.21. 1) Show that every connected graph G has a vertex x such that $G-x$ is connected.
2) Does the same hold for strongly connected digraphs?

Exercise 1.22. Let G be a connected graph and e an edge of G. Show that G has a spanning tree containing e.

Exercise 1.23. A graph is *cherry-free* if every vertex has at most one neighbour of degree 1. Prove that a connected cherry-free graph has two adjacent vertices u and v such that $G-\{u,v\}$ is connected.
Hint: Consider a path of maximum length.

Exercise 1.24. Let G be a connected graph and $(V_1,V_2,\ldots,V_n)$ a partition of $V(G)$ such that $G\langle V_i\rangle$ is connected for all $1\leq i\leq n$. Show that there exists two indices i and j such that $G-V_i$ and $G-V_j$ are connected.

Exercise 1.25. The aim of this exercise is to prove that if a graph has n vertices, m edges and k connected components then $n-k\leq m\leq \frac{1}{2}(n-k)(n-k+1)$.
1) Let G be a graph on n vertices with m edges and k connected components.

a) Show that if G is connected then $m\geq n-1$.

b) Deduce that if G has k connected components then $m\geq n-k$.

2) Suppose now that G is a graph on n vertices and k connected components with the maximum number of edges.

a) Show that all the connected components of G are complete graphs.

b) Show that if G has (at least) two connected components then one of them has a unique vertex.

c) Deduce that G has $\frac{1}{2}(n-k)(n-k+1)$ edges.

Exercise 1.26. Prove Proposition 1.13.

Exercise 1.27. Let D be a strongly connected digraph and D' a strongly connected subdigraph of D. Show that any handle decomposition $(C,P_1,\ldots,P_k)$ of D' may be extended into a handle decomposition $(C,P_1,\ldots,P_k,\ldots,P_l)$ of D.

Exercise 1.28. Let D be a strongly connected digraph of minimum outdegree 2. Prove that there exists a vertex v such that $D-v$ is strongly connected.

Exercise 1.29. Show that a graph has an eulerian trail if and only if it has zero or two vertices of odd degree.

Exercise 1.30. Let $G=(V,E)$ be a graph such that every vertex has even degree and $|E|\equiv 0[3]$. Prove that E can be partitionned into $l=\frac{|E|}{3}$ sets $E_1,\ldots,E_l$ such that for all $1\leq i\leq l$, the graph induced by E_i is either a path of length 3 or cycle of length 3.

Bibliography

[1] J. A. Bondy and U. S. R. Murty. *Graph theory*, Series: Graduate Texts in Mathematics, Vol. 244, Springer, 2008.

[2] R. Diestel. *Graph Theory*, 3rd edition. Graduate Texts in Mathematics, 173. Springer-Verlag, Berlin and Heidelberg, 2005.

[3] D. B. West. *Introduction to graph theory*. Prentice Hall, Inc., Upper Saddle River, NJ, 1996.

Chapter 2

Searches in graphs and digraphs

2.1 Searches and connectivity in graphs

Finding the connected component of a vertex v in a graph is not difficult. For this purpose, we need a list of edges to be explored, initially containing all edges, and a list of already explored vertices, initially containing only v. At each step, a new edge ab is explored with a already explored and b is added in the list of explored vertices if it had not been explored yet. When the procedures stops, the set of explored vertices is the connected component of v. This algorithm may be modified to return a spanning tree T of the connected component of v.

Algorithm 2.1 (Search)**.**

1. Mark v and initialize L to the set of all edges incident to v; $V(T) := \{v\}$, $E(T) := \emptyset$.
2. If $L = \emptyset$, then return T; else let $ab \in L$. $L := L \setminus \{ab\}$.
3. If b is not marked, then mark it; $V(T) := V(T) \cup \{b\}$; $E(T) := E(T) \cup \{ab\}$; add all the edges incident to b to L.
4. Go to 2.

There are two well-known searches which correspond to two different orderings of the edges:

- the breadth-first search (BFS) (Algorithm 2.2) explores first the neighbours of v, then the neighbours of its children;
- the depth-first search (DFS) (Algorithm 2.3) explores first all the vertices of a branch pending in v.

The difference between these two approaches is that the vertices are stored either in a queue (FIFO) or in a stack (LIFO). A *queue* is just a list which is updated by either adding a new

element to one end (its *tail*) or removing an element from the other end (its *head*). A *stack* is simply a list, one end of which is identified as its *top*; it may be updated either by adding a new element as its top or else by removing its top element.

The following algorithms also compute the connected component C of v and a spanning tree of C.

Algorithm 2.2 (Breadth-First Search)**.**

1. Mark v, $V(T) := \{v\}$, $E(T) := \emptyset$ and initialize a queue Q to v.
2. If $F = \emptyset$ then return T. Else, remove the the vertex u fromt the head of Q.
3. For every unmarked vertex w adjacent to u, $V(T) := V(T) \cup \{w\}$; $E(T) = E(T) \cup \{uw\}$; add w to the tail of Q and mark w.
4. Go to 2.

Algorithm 2.3 (Depth-First Search)**.**

0. For every vertex, $L(u) := N(u)$.
1. Mark v; $V(T) := \{v\}$; $E(T) := \emptyset$ and initialize a stack P to v.
2. If $P = \emptyset$, then return T. Else, let u be the vertex on top of the stack P.
3. If $L(u) = \emptyset$, then remove u from the top of the stack P and go to 2.
4. Else, remove a vertex w from $L(u)$.
5. If w is marked go to 3. Else, $V(T) := V(T) \cup \{w\}$; $E(T) = E(T) \cup \{uw\}$; add w on top of P and mark w.
6. Go to 2.

A tree obtained by running a breadth-first search is called a *breadth-first search tree* or *BFS-tree*. Similarly, a tree obtained by running a depth-first search is called a *depth-first search tree* or *DFS-tree*. If the search is run from vertex v, this vertex is called the *root* of the search tree.

Observe that in Algorithms 2.1, 2.2 and 2.3 every edge is examined at most twice (once per endvertex). These algorithms can be modified in order to compute all the connected components of a graph so that every edge is examined at most twice. For this purpose, while some vertex does not belong to a connected component (i.e., has not been marked), it is sufficient to compute its connected component.

2.1.1 Distance in graphs

A nice property of a breadth-first search tree is that is can give the distance from the root r to all other vertices. Therefore we need at each vertex a value $l(u)$, called *level* of u, which corresponds to $dist_T(r,u)$ as well as $dist_G(r,u)$ as we will show later. Hence the following algorithm is the following:

Algorithm 2.4 (Distance from r)**.**

1. Mark r, $l(r) := 0$ and initialize a queue Q to v. [[$V(T) := \{v\}$; $E(T) := \emptyset$;]]
2. If $F = \emptyset$ then return l. Else, remove the first vertex u of Q.
3. For every unmarked vertex w adjacent to u, $l(w) := l(u) + 1$; add w to Q and mark w. [[$V(T) := V(T) \cup \{w\}$; $E(T) = E(T) \cup \{uw\}$;]]
4. Go to 2.

Observe that the construction of the tree T (operation between brackets at Step 1 and 3) is practically useless. It just help us to show that the function l has the properties we announced. The first ones justifies our referring to l as the level function.

Theorem 2.1. *Let r be a vertex of a connected graph G and T be a BFS-tree (as constructed by Algorithm 2.4). Then:*

(i) for every vertex v of G, $l(v) = dist_T(r,v)$*;*

(ii) every edge of G joins vertices on the same or consecutive levels of T; that is

$$|l(u) - l(v)| \leq 1, \quad \textit{for all } uv \in E(G).$$

Proof. The proof of (i) is left to the reader in Exercise 2.1. To establish (ii), it suffices to prove that if $uv \in E(G)$ and $l(u) < l(v)$, then $l(u) = l(v) - 1$.

We first establish, by induction on $l(u)$, that if u and v are any two vertices such that $l(u) < l(v)$, then u joined Q before v. This evident if $l(u) = 0$, because u is then the root r of T. Suppose that the assertion is true whenever $l(u) < k$, and consider the case $l(u) = k$, where $k > 0$. Let x be the predecessor of u, that is the vertex which is explored when we add u to Q. Then it follows from line 3 of Algorithm 2.4 that $l(x) = l(u) - 1$. Similarly, if y is the predecessor of v then $l(y) = l(v) - 1$. By induction, x joined Q before y. Therefore u being a neighbour of x, joined Q before v.

Now suppose that $uv \in E(G)$ and $l(u) < l(v)$. If u is the predecessor of v, then $l(u) = l(v) - 1$, again by line 3 of Algorithm 2.4. If not, let y be the predecessor of v. Because v was added to T by the edge yv, and not the edge uv, the vertex y joined Q before u, hence $l(y) \leq l(u)$ by the claim established above. Therefore $l(v) - 1 = ly) \leq l(u) \leq l(v) \leq l(v) - 1$, which implies $l(u) = l(v) - 1$. □

The following theorem shows that Algorithm 2.4 runs correctly.

Theorem 2.2. *Let T be a BFS-tree of a connected graph G, with root r. Then:*

$$dist_T(r,v) = dist_G(r,v), \quad \text{for all } v \in V(G).$$

Proof. Clearly, $dist_T(r,v) \geq dist_G(r,v)$ because T is a subgraph of G.

Let us establish the opposite inequality by induction on the length of a shortest (r,v)-path, the proposition holding trivially when the length is 0.

Let P be a shortest (r,v)-path in G, where $v \neq r$, and let u be the predecessor of v on P. The (r,u)-subpath of P is a shortest (r,u)-path, and $d_G(r,u) = d_G(r,v) - 1$. By induction, $l(u) \leq d_G(r,u)$, and by Theorem 2.1-(ii), $l(v) = l(u) \leq 1$. Therefore

$$dist_T(r,v) = l(v) \leq l(u) + 1 \leq d_G(r,u) + 1 = d_G(r,v).$$

□

2.2 Searches and strong connectivity in directed graphs

One can explore digraphs in much the same way as graphs, but by growing arborescences rather than (rooted trees). An *arboresence* is an orientation of a rooted tree in which all the arcs are directed from the root to the leaves. It can be seen as a digraph in which every vertex has indegree 1 except one called the root which has indegree 0. As with search in graph, search in digraph may be refined by restricting the choice of the arc to be added at each stage. In this way, we obtain directed versions of breadth-first search and depth-first search. We now discuss how search can be applied to find the strongly connected components of a digraph.

To test if a digraph $D = (V,E)$ is strongly connected, one has to check for every pair u,v of vertices if there is a (u,v)-dipath. Checking if such a path exists can be done by performing a search, so running $\binom{|V(D)|}{2}$ searches will do the job. However running a search from a vertex u finds all the vertices v that can be reached from u. So, in fact, one just need to run at most $|V|$ searches (one per vertex) yielding a total time $O(|V||E|)$.

The following proposition will yield an algorithm that test if a digraph is strong by running only two searches.

Proposition 2.3. *Let D be a digraph and v a vertex of D. D is strongly connected if and only if, for every $u \neq v$, there are a (u,v)-path and a (v,u)-path.*

Proof. If D is strongly connected, by definition, for every $u \neq v$, there are a (u,v)-path and a (v,u)-path.

Let us assume now that for every $u \neq v$, there are a (u,v)-path and a (v,u)-path. Let us show that D is strongly connected. Let u and w be two distinct vertices of D. There are a (u,v)-path P and a (v,w) path Q the concatenation of which is a (u,w)-walk. By Proposition 1.3, there is a (u,w)-path. □

We now describe an algorithm that computes the strongly connected components of a vertex v of a digraph. It is based on two searches starting from v, the first one in D and the reverse $\bar{D}$ of D. During the first search, the vertices u reachable from v are marked 1. During the second search the vertices u from which v can be reached marked 2 and included in the strongly connected component of v if they are already marked 1 (See Figure 2.1).

Algorithm 2.5 (Strongly connected component)**.**

1. Search D starting from v marking the vertices with 1.
2. Search $\bar{D}$ starting from v marking the vertices with 2.
3. Return the vertices marked with 1 and 2.

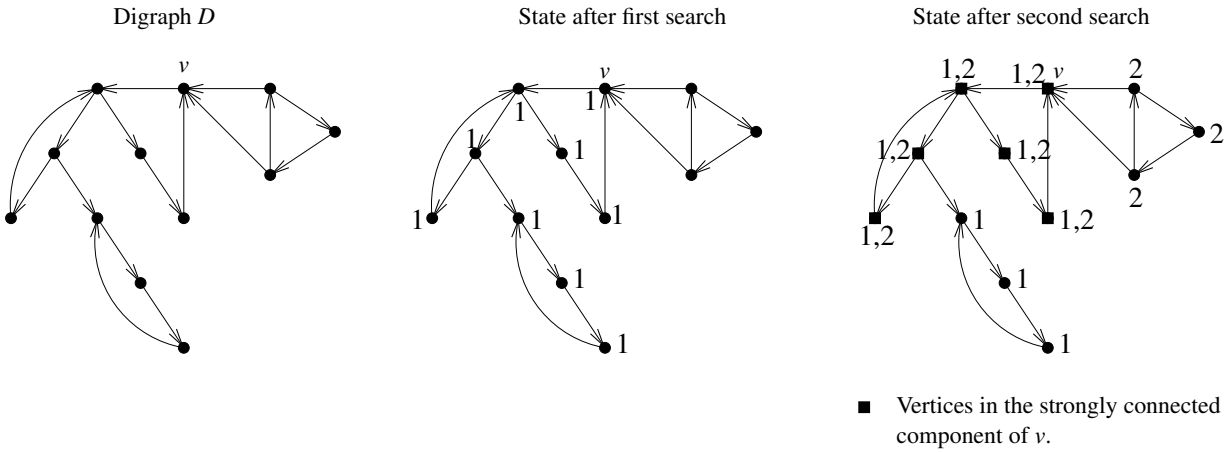


Figure 2.1: Execution of Algorithm 2.5

Contrary to Algorithm 2.1, Algoritm 2.5 does not give all strongly connected components of D in a single search examining each edges twice. Indeed, let us consider the digraph D with $V(D) = \{v_1, v_2, \ldots, v_n\}$ and$E(D) = \{(v_i, v_j) \mid i < j\}$. The components consist of each $\{v_i\}$. Hence, $|V|$ executions of Algoritm 2.5 must be done. Moreover, at each execution, all edges are considered.

2.2.1 Computing strongly connected components in one search

We now describe an algorithm that computes all strongly connected components of a digraph in time $O(|E|)$. It is a modified depth-first search in which two extra values are stored and updated. When a vertex is explored u it becomes *active* and is associated to two values $l(u)$ and $b(u)$. The first one $l(u)$, called *label* of u, corresponds to the order of appearance of u during the

search. It will never change. A vertex w such that $l(w) \geq l(u)$ is called a *successor* of u. A key ingredient of the algorithm is that as long as u is active, there is a directed (u,w)-path to each of its successors w. The second value $b(u)$ corresponds to the smallest label of a vertex reachable from u in the subdigraph induced by the explored arcs. Thus it needs to be updated when new arcs are explored.

Algorithm 2.6 (All Strongly Connected Components)**.**

0. Initialize i to 0.
1. If all vertices are marked, then terminate.
2. Else $i := i+1$.
3. Let u be an unmarked vertex. $l(u) := i$, $b(u) := i$, and u becomes active.
4. If at least one arc leaving u, say (u,v), is not marked, then do
   - 4.1 Mark (u,v).
   - 4.2 If v has already been explored and is active, then update $b(u)$: $b(u) := \min(b(u), b(v))$.
   - 4.3 Else v is a new vertex. v becomes active; $i := i+1$; $l(v) := i$; $b(v) := l(v)$; $u := v$.
   - 4.4 Go to 4.
5. Else, all arcs leaving u are marked, the exploration of u is over:
   - 5.1 If $b(u) = l(u)$ then all active successors of u induce a strongly connected component: return it and all its vertices become inactive; Go to 1.
   - 5.2 Else $b(u) < l(u)$. Let w be the vertex from which u has been explored. Update $b(w)$: $b(w) := \min(b(w), b(u))$; $u := w$; Go to 4.

Correctness of Algorithm 2.6: We will show by induction the following three points.

1) If $l(u) < l(v)$, and if u and v are active, then v is a successor of u;
2) At each step, for any active vertex v, there is a directed path from v to the vertex w with label $l(w) = b(v)$.
3) When the exploration of u terminates (Step 5), all the active vertices of $S(u) = \{v \text{ active } \mid b(u) \leq l(v) \leq l(u)\}$ are in the same strongly connected component as u.
4) $b(u) = l(u)$ if and only if $S(u)$ is a strongly connected component.

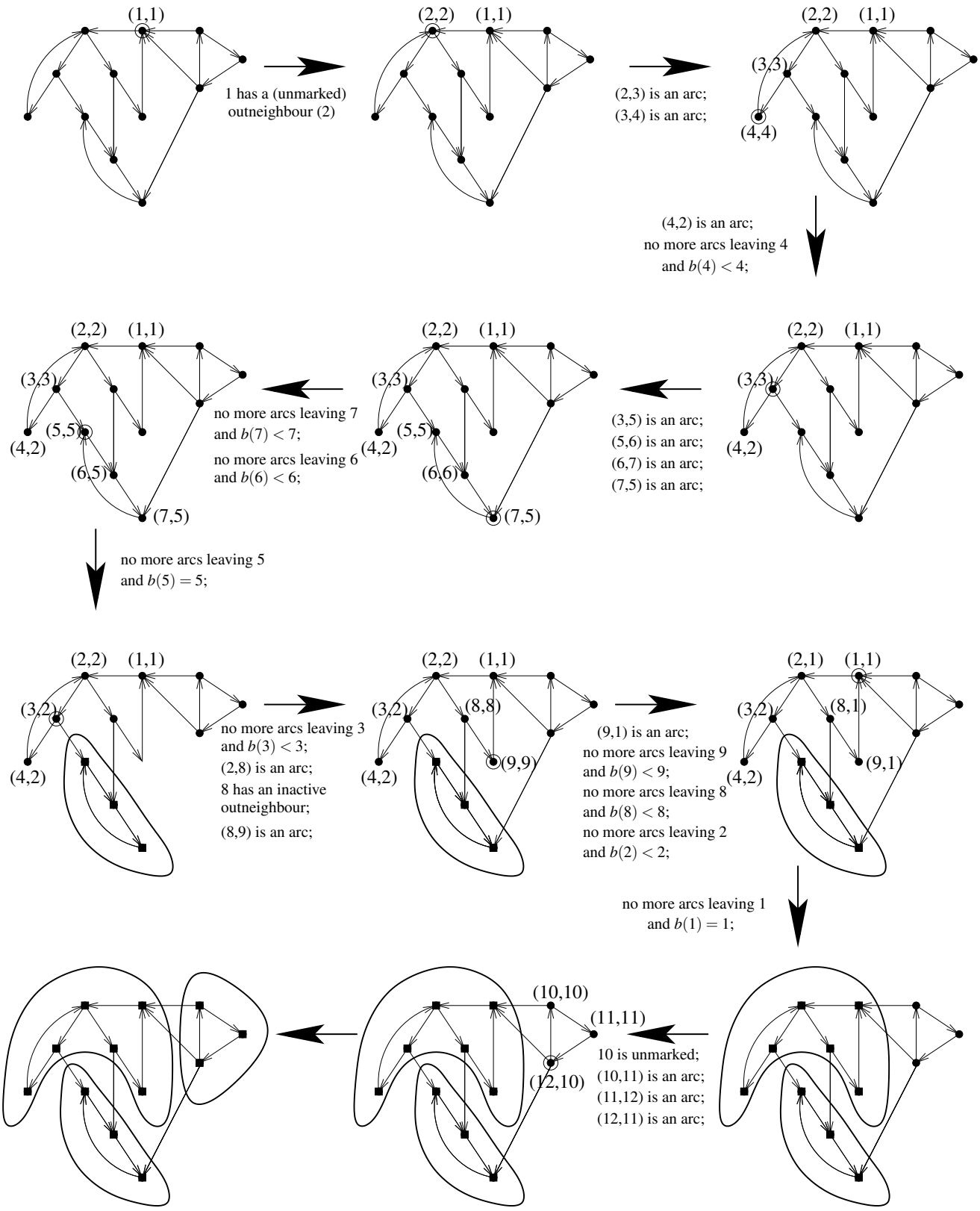


Figure 2.2: A run of Algorithm 2.6. The vertex in a circle is the current vertex u. At each step, $(l(v), b(v))$ is represented close to every active vertex v. Finally, once a vertex becomes inactive, it is depicted by a square.

1) and 2) Left to the reader.

3) From Proposition 2.3, it is sufficient to show that, for every v in $S(u)$, there is a directed (u,v)-path and a directed (v,u)-path. Let v be a vertex in $S(u)$.
Let us assume first that $l(v) < l(u)$. Then, by 1), there is a directed (v,u)-path. Let w be the vertex such that $l(w) = b(u)$. By 2), there is a directed (u,w)-path, and by 1) there is a (w,v)-path. The concatenation of these two paths is a directed (u,v)-walk. By Proposition 1.4 (i), there is a directed (u,v)-path.
Let us now assume that $l(v) > l(u)$. By 1) there is a (u,v)-path. Moreover, $b(v) < l(v)$, since otherwise, the vertex v would have become inactive at Step 5. Let v_1 be the vertex with label $l(v_1) = b(v)$. By 2), there is a directed (v,v_1)-path P_1. If $l(v_1) \leq l(u)$, then, by 1), there is a directed (v_1,u)-path whose concatenation with P_1 is a directed (v,u)-walk. Hence, a directed (v,u)-path exists according Proposition 1.3 (i). If $l(v_1) > l(u)$, then $b(v_1) < l(v_1)$, for otherwise, the vertex v_1 (and also v) would have become inactive at Step 5. Let v_2 be the vertex such that $l(v_2) = b(v_1)$. Using similar arguments, while $l(v_i) > l(u)$, we have $l(v_i) > b(v_i)$ and we denote by v_{i+1} the vertex such that $l(v_{i+1)} = b(v_i)$. Since the label of v_i strictly decreases, the sequence of the v_i's is finite. Let k be the last index of this sequence.Then $l(v_k) \leq l(u)$. By 2), for every $1 \leq i \leq k$, there is a directed (v_{i-1},v_i)-path. Moreover, by 1), there is a directed (v_k,u)-path. The concatenation of all these paths is a directed (v,u)-walk, and then, by Proposition 1.4 (i), there is a directed (u,v)-path.

4) If $b(u) = l(u)$, then the arcs leaving $S(u)$ have their heads inactive (vertices in a distinct strongly connected component, by the induction hypothesis 3). Hence, $S(u)$ is a strongly connected component.
If $b(u) < l(u)$, then the vertex v labelled $b(u)$ is active. By 3), u and v are in the same strongly connected component but $v \notin S(u)$. Hence $S(u)$ is no a strongly connected component.

2.3 Bipartite graphs

A *bipartition* of a graph G is a partition (A,B) of $V(G)$ into two stable sets. Hence, every edge of G has an endvertex in A and the other in B. We often write $G = ((A,B),E)$ for a bipartite graph with bipartition (A,B).

Bipartite graphs satisfy some properties.

Proposition 2.4. *Let $G = ((A,B),E)$ be a bipartite graph.*

(i) for any two vertices u and v, all the (u,v)-walks have the same length parity.

(ii) if G is connected then it has only two bipartitions (A,B) and (B,A).

Proof. (i) Without loss of generality, we may assume that u is in A. Let $(v_0,v_1,\ldots,v_k)$ be a (u,v)-walk (so $v_0 = u$ and $v_k = v$). Since G is bipartite and $V_0 \in A$ then $v_1 \in B$ and so $v_2 \in A$. And so on by induction, if i is even then $v_i \in A$ and if i is odd $v_i \in B$. Thus if $v \in A$ then k is even and if $v \in B$ k is odd.

(ii) Let u be a vertex. Let A_0 (resp. A_1) be the set of vertices at even (resp. odd) distance to v_0 in G. By (i), in any bipartition of G, A_0 is included in the part containing u and A_1 in the other. $A_0 \cup A_1 = V(G)$ since the graph is connected then there are only two possible partitions of G: (A_0, A_1) and (A_1, A_0). □

There are graphs which are not bipartite, for example the odd cycles. Indeed, in the cycle $(v_0, v_1, \ldots, v_{2k}, v_0)$ the path (v_0, v_{2k}) and $(v_0, v_1, \ldots, v_{2k})$ are two (v_0, v_{2k})-paths of different length parity. Hence if a graph is bipartite, it contains no odd cycles. This easy necessary condition to be bipartite is in fact sufficient.

Theorem 2.5. *A graph G is bipartite if and only if it has no odd cycle.*

Proof. Clearly, it suffices to prove it for connected graphs. Let G be a connected graph. If G contains an odd cycle, it is not bipartite.

Conversely, assume that G contains no odd cycle. Let v_0 be a vertex of G. Let A_0 (resp. A_1) be the set of vertices at even (resp. odd) distance to v_0 in G. Let us now show that (A_0, A_1) is a bipartiton of G. Let uv be an edge of G and P_u (resp. P_v) be a shortest (u, v_0)-path (resp. (v, v_0)-path). The concatenation P_u, P_v and (v, u) is a closed walk. By Proposition 1.3, this walk has even length otherwise G would contain an odd cycle. Hence P_u and P_v have different length parity and so uv has an endvertex in each of the A_i, $i = 0, 1$. □

The above proof may be translated into an algorithm which, given a connected graph G, returns either a bipartition if G is bipartite or "G is not bipartite" otherwise. Basicallly, it runs a Breadth-First Search from a vertex and check if there is no edge between vertices of levels of differents parity. Hence instead of marking the vertices with their level number as for the distance (see Subsection 2.1.1), we mark them with the parity of their level and thus we just need two marks, 0 and 1.

Algorithm 2.1 (Finding a bipartition)**.**

1. Pick a vertex x and mark it with $m(x) := 0$; $N := \{x\}$.
2. If N is non-empty, then remove a vertex v of N and do the following.

   For each neighbour w of v do
   - If $m(w) = m(v)$, return "G is not bipartite";
   - Otherwise if w is unmarked, then mark it with $m(v) + 1$ *mod* 2 and put w in N; Go to 2.
3. If N is empty, let $A_i, i = 0, 1$ be the set of vertices marked i and return "G is bipartite with bipartition" (A_0, A_1).

Algorithm 2.1 may be easily modified to return an odd cycle when G is not bipartite. See Exercise 2.10.

2.4 Exercises

Exercise 2.1. Show Theorem 2.1-(i).

Exercise 2.2 (Entriger, Kleitman and Székely)**.**
For a connected graph G, define $\sigma(G) = \sum_{u,v \in V(G)} dist(u,v)$.

1) Let G be a connected graph. For $v \in V(G)$, let T_v be a BFS-tree of G rooted at v. Show that $\sum_{v \in V(G)} \sigma(T_v) = 2(n-1)\sigma(G)$.

2) Deduce that every connected graph G has a spanning tree T such that $\sigma(T) \leq 2(1 - \frac{1}{n})\sigma(G)$.

Exercise 2.3 (Tuza)**.** Let G be a connected graph, let x be a vertex of G, and let T be a spanning tree of G that maximizes the function $\sum_{v \in V(G)} dist_T(x,v)$. Show that T is a DFS-tree of G.

Exercise 2.4 (Chartrand and Kronk)**.** Let G be a connected graph in which every DFS-tree is a path (rooted at the start). Show that G is a cycle, a complete graph, or a bipartite graph in which both parts have the same number of vertices.

Exercise 2.5 (Pósa)**.** A *chord* of a cycle C in a graph G is an edge in $E(G) \setminus E(C)$ both of whose endvertices lie on C. Let G be a graph such that $|E(G)| \geq 2|V(G)| - 3$ and $|V(G)| \geq 4$. Show that G contains a cycle with at least one chord.

Exercise 2.6. Let a be vertex of a connected graph G. Prove that G is bipartite if and only if $dist(a,b) \neq dist(a,c)$ for all edge bc.

Exercise 2.7.
1) Show that every tree is bipartite.
2) Prove that every tree has a leaf in the largest part of its bipartition.

Exercise 2.8. Prove that a bipartite graph G has at most $|V(G)|^2/4$ edges and give a graph attaining this bound.

Exercise 2.9. Show that a graph is bipartite if and only if each of its subgraphs H has a stable set of size at least $|V(H)|/2$.

Exercise 2.10. Give an algorithm that, given a connected graph G, returns either a bipartition if G is bipartite or an odd cycle if G is non-bipartite.

Exercise 2.11. Describe an algorithm based on a breadth-first search for finding a shortest odd cycle in a graph.

Exercise 2.12. Let $G = ((A,B),E)$ be a bipartite graph without isolated vertices such that $d(x) \geq d(b)$ for all $xy \in E$, where $a \in A$ and $b \in B$. Prove that $|A| \leq |B|$, with equality if and only if $d(a) = d(b)$ for all $ab \in E$.

Chapter 3

Algorithms in edge-weighted graphs

Recall that an *edge-weighted graph* is a pair (G,w) where $G=(V,E)$ is a graph and $w : E \to \mathbb{R}$ is a *weight function*. Edge-weighted graphs appear as a model for numerous problems where places (cities, computers,...) are linked with links of different weights (distance, cost, throughput,...). Note that a graph can be viewed as an edge-weighted graph where all edges have weight 1.

Let (G,w) be an edge-weighted graph. For any subgraph H of G, the *weight* of H, denoted by $w(H)$, is the sum of all the weights of the edges of H. In particular, if P is a path, $w(P)$ is called the *length* of P. The *distance* between two vertices u and v, denoted by $dist_{G,w}(u,v)$, is the length of a shortest (with minimum length) (u,v)-path.

Observe that $dist_{G,w}$ is a distance: it is *symmetric*, that is, $dist_{G,w}(u,v) = dist_{G,w}(v,u)$, and it satisfies the *triangle inequality*: for any three vertices x, y and z, $dist_{G,w}(x,z) \leq dist_{G,w}(x,y) + dist_{G,w}(y,z)$.

3.1 Computing shortest paths

Given an edge-weighted graph (G,w), one of the main problems is the computation of $dist_G(u,v)$ and finding a shortest (u,v)-path. We have seen in Subsection 2.1.1, that if all the edges have same weight then one can compute a shortest (u,v)-path by running a breadth-first search from u. Unfortunately, this approach fails for general edge-weighted graphs. See Exercise 3.1. We now describe algorithms to solve this problem in general. For this purpose, we solve the following more general problem.

Problem 3.1 (Shortest-paths tree)**.**
Instance: an edge-weighted graph (G,w) and a vertex r.
Find: a subtree T of G such that $\forall x \in V(G), dist_{G,w}(r,x) = dist_{T,w}(r,x)$.

Such a tree is called a *shortest-paths tree*.

3.1.1 Dijkstra's Algorithm

Dijkstra's Algorithm is based on the following principle. Let $S \subset V(G)$ containing r and let $\bar{S} = V(G) \setminus S$. If $P = (r, s_1, \ldots, s_k, \bar{s})$ is a shortest path from r to $\bar{S}$, then $s_k \in S$ and P is a shortest path from r to $\bar{s}$. Hence,

$$dist(r, \bar{s}) = dist(r, s_k) + w(s_k \bar{s})$$

and the distance from r to $\bar{S}$ is given by the following formula

$$dist(r, \bar{S}) = \min_{u \in S, v \in \bar{S}} \{dist(r, u) + w(uv)\}$$

To avoid to many calculations during the algorithm, each vertex $v \in V(G)$ is associated to a function $d'(v)$ which is an upper bound on $dist(r, v)$, and to a vertex $p(v)$ which is the potential parent of v in the tree. At each step, we have:

$$\begin{aligned} d'(v) &= dist(r, v) \text{ if } v \in V(T_i) \\ d'(v) &= \min_{u \in V(T_{i-1})} \{dist(r, u) + w(uv)\} \text{ if } v \in \overline{V(T_i)} \end{aligned}$$

Algorithm 3.1 (Dijkstra)**.**

1. Initialize $d'(r) := 0$ and $d'(v) := +\infty$ if $v \neq r$. T_0 is the tree consisting of the single vertex r, $u_0 := r$ and $i := 0$.
2. For any $v \in \overline{V(T_i)}$, if $d'(u_i) + w(u_i v) \leq d'(v)$, then $d'(v) := d'(u_i) + w(u_i v)$ and $p(v) := u_i$.
3. Compute $\min\{d'(v) \mid v \in \overline{V(T_i)}\}$. Let u_{i+1} a vertex for which this minimum is reached. Let T_{i+1} be the tree obtained by adding the vertex u_{i+1} and the edge $p(u_{i+1})u_{i+1}$.
4. If $i = |V| - 1$, return T_i, else $i := i + 1$ and go to Step 2.

Remark 3.2. The algorithm does not work if some weights are negative.

Complexity of Dijkstra's Algorithm: To every vertex is associated a temporary label corresponding to $(d'(v), p(v))$. They are depicted in Figure 3.1. We do

- at most $|E|$ updates of the labels;
- $|V|$ searches for the vertex v for which $d'(v)$ minimum and as many removal of labels.

The complexity depends on the choice of the data structure for storing the labels: if it is a list, the complexity is $O(|E||V| + |V|^2)$. But it can be improved using better data structures. For example, a data structure known as *heap* is commonly used for sorting elements and their

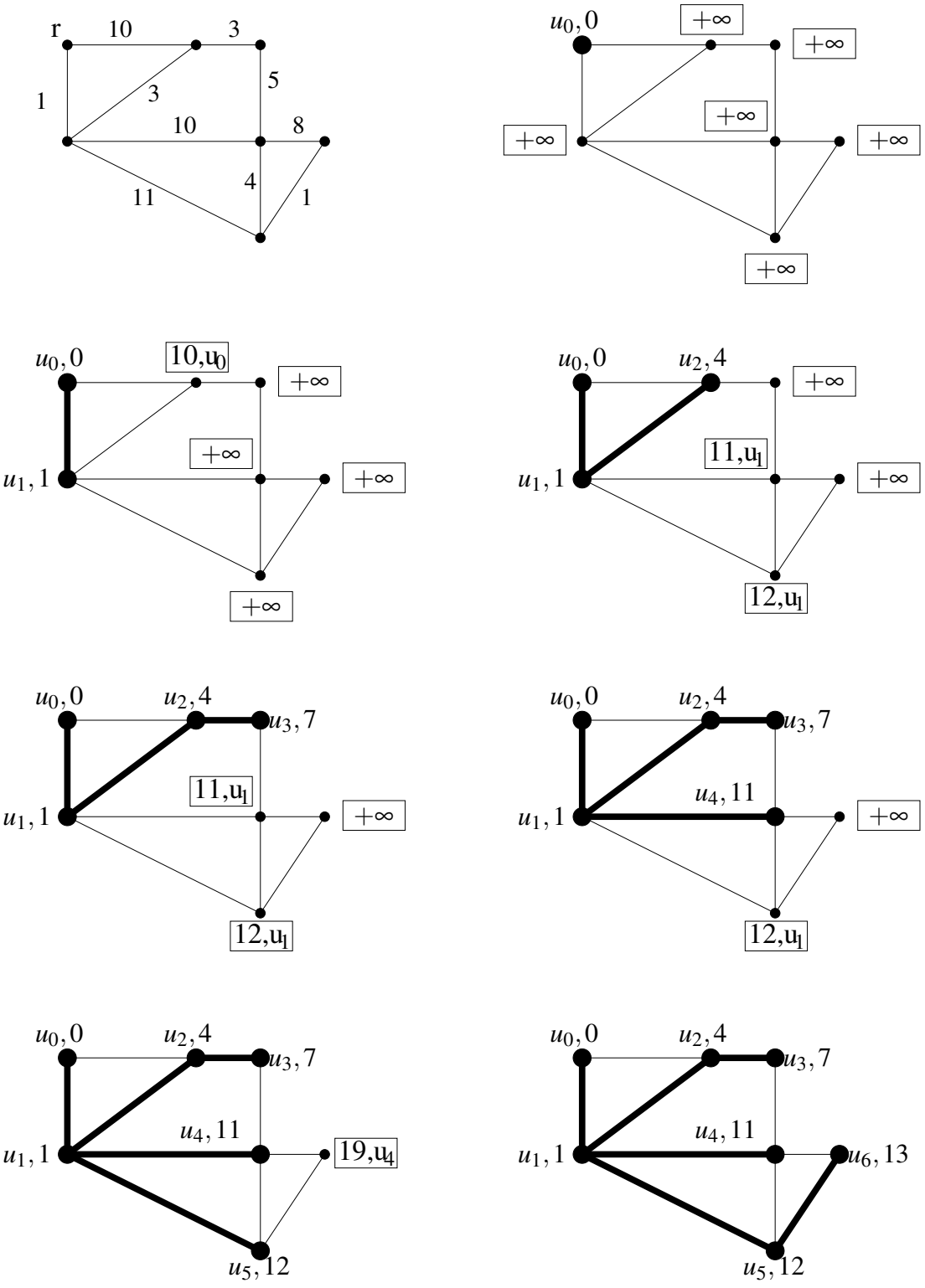


Figure 3.1: A run of Dijkstra's Algorithm on the edge-weighted graph depicted top left. At each step, bold vertices and edges are those of T_i. To each vertex t of T_i is associated its name and the value $d'(t) = dist(r,t)$. Next to each vertex v not in $V(T_i)$ is a box containing the value $d'(v)$ and $p(v)$ if $d'(v) \neq +\infty$.

associated values, called *keys* (such as edges and their weights). A heap is a rooted binary tree T **should we define it?** whose vertices are in one-to-one correspondence with the elements in question (in our case, vertices or edges). The defining property of a heap is that the key of the element located at vertex v of T is required to be at least as small as the keys of the elements located at vertices of the subtree of T rooted at v. This condition implies, in particular, that the key of the element at the root of T is one of smallest value; this element can thus be accessed instantly. Moreover, heaps can be reconstituted rapidly following small modifications such as the addition of an element, the removal of an element, or a change in the value of a key. A *priority queue* is simply a heap equipped with procedures for performing such readjustments rapidly.

Using a priority queue, the complexity of Dijkstra's Algorithm is $O(|E|\log|V|+|V|\log|V|)$.

It should be evident that data structures play a vital role in the efficiency of algorithms. For further information on this topic, we refer the reader to [4, 1, 5, 3].

3.1.2 Bellmann-Ford Algorithm

The algorithm performs n iterations, and gives a label $h(v)$ to any vertex. At iteration i, $h(v)$ is the minimum weight of a path using at most i edges between r and v.

Note that, it always exists a shortest walk using at most $|V(G)|-1$ edges between r and v (otherwise the walk would contain a cycle of negative weight).

Algorithm 3.2 (Bellmann-Ford)**.**

1. Initialization : $h(r) := 0$, $h(v) := +\infty, \forall v \neq r$.
2. For $i = 0$ to $|V(G)|-1$ do :
   for all $v \in V(G), h(v) := \min(h(v), \min\{h(u)+w(uv) \mid uv \in E(G)\})$.
3. Return $d(r,v) = h(r,v)$.

Complexity of Bellmann-Ford's Algorithm: Each iteration costs $O(|E|)$ (all edges are considered), so the total complexity is $O(|E||V|)$.

The algorithm works even if some edges have negative weight. It can also detect cycles with negative weight. There is such a cycle if and only if, after the $|V|^{\text{th}}$ iteration, the labels h may decrease. Finally, if during an iteration, no $h(v)$ decreases, then $h(v) = d(r,v)$. It is possible to improve the algorithm by continuing the iteration only if $h(v)$ becomes $\min\{h(u)+w(uv) \mid uv \in E(G)\}$ for at least one vertex. The algorithm run in time $O(L|E|)$ where L is the maximum number of edges in a shortest path.

3.2 Minimum-weight spanning tree

Another important problem is the following: given a connected edge-weighted graph, what is the connected spanning subgraph with minimum weight? If all weights are non-negative, since any connected graph has a spanning tree (Corollary 1.10), the problem consists of finding a spanning tree with minimum weight.

Problem 3.3 (Minimum-Weight Spanning Tree)**.**
Instance: a connected edge-weighted graph (G,w).
Find: a spanning tree T of G with minimum weight, i.e. for which $\sum_{e\in T} w(e)$ is minimum.

For $S \subset V(G)$, an edge $e = xy$ is *S-transversal*, if $x \in S$ and $y \notin S$. The algorithms to find a minimum-weight spanning tree are based on the fact that a transversal edge with minimum weight is contained in a minimum-weight spanning tree.

Lemma 3.4. *Let (G,w) be an edge-weighted graph and let $S \subset V$. If $e = s\bar{s}$ is an S-transversal edge with minimum weight, then there is a minimum-weight spanning tree containing e.*

Proof. Let T be a tree that does not contains e. There is a path P between s and $\bar{s}$ in T. At least one edge of P, say e', is S-transversal. Hence, the tree $T' = (T \setminus e') \cup \{e\}$ has weight $w(T') = w(T) + w(e) - w(e') \leq w(T)$ since $w(e) \leq w(e')$. Therefore, if T is a minimum spanning tree, then so does T' and $w(e) = w(e')$. □

In particular, Lemma 3.4 implies that if e is an edge of minimum weight, i.e., $w(e) = \min_{f\in E(G)} w(f) = w_{min}$, then there is a minimum-weight spanning tree containing e.

3.2.1 Jarník-Prim Algorithm

The idea is to grow up the tree T with minimum weight by adding, at each step, a $V(T)$-transversal edge with minimum weight. At each step, E_T is the set of the $V(T)$-transversal edges.

Algorithm 3.3 (Jarník-Prim)**.**

1. Initialize the tree T to any vertex x and E_T is the set of edges incident to x.
2. While $V(T) \neq V(G)$:
   Find an edge $e \in E_T$ with minimum weight. Add e and its end not in T to T. Let E_y be the set of edges incident to y. Remplace E_T by $(E_T \triangle E_y)$.

Complexity of Jarník-Prim Algorithm: During the execution, at most $|E(G)|$ edges are added in E_T, and at most $|E(G)|$ edges are removed. Indeed, an edge e is removed when both its endvertices are in $V(T)$. Since $V(T)$ grows up, e will not be added anymore to E_T. $|V(G)|$ selections of the edge of E_T with minimum weight must be performed. To performs such an algorithm we need a data structure allows the insertion, the removal and the selection of the minimum-weight element efficiently. Using a priority queue, the total complexity of Jarník-Prim Algorithm is $O(|E|\log|E|)$.

3.2.2 Boruvka-Kruskal Algorithm

Boruvka-Kruskal Algorithm is close to Jarník-Prim Algorithm and its correctness also comes from Lemma 3.4. The idea is to start from a spanning forest and to make its number of connected components decreases until a tree is obtained. Initially, the forest has no edges and, at each step, an edge with minimum weight that links two components is added.

For this purpose, we need a fast mechanism allowing to test whether or not u and v are in the same component. A way to do so consists in associating to each connected component the list of all the vertices it contains. To every vertex u is associated a vertex $p(u)$ in the same component. This vertex $p(u)$ is a *representative* of this component. It points to the set $C_{p(u)}$ of vertices of this component and to the size $size(p(u))$ corresponding to the size it.

Algorithm 3.4 (Kruskal)**.**

1. Initalize $T : V(T) := V(G)$, $E(T) := \emptyset$. Order the edges in increasing order of the weights and place them in a stack L; For all $u \in V(G)$, do $p(u) := C_u$ and $size(C_u) := 1$.
2. If $L = \emptyset$, terminate. Else, pull the edge $e = uv$ with minimum weight;
3. If $p(u) = p(v)$ (the vertices are in the same component), then go to 2. Else $p(u) \neq p(v)$, add e in T.
4. If $size(p(u)) \geq size(p(v))$, then $C_{p(u)} := C_{p(u)} \cup C_{p(v)}$, $size(p(u)) := size(p(u)) + size(p(v))$, and for any $w \in C_{p(v)}$, $p(w) := p(u)$.
   Else $(size(p(u)) < size(p(v)))$, $C_{p(v)} := C_{p(v)} \cup C_{p(u)}$, $size(p(v)) := size(p(u)) + size(p(v))$, and for any $w \in C_{p(u)}$, $p(w) := p(v)$.
5. Go to 2.

Complexity of Boruvka-Kruskal Algorithm Ordering the edges takes time $O(|E(G)|\log|E(G)|)$. Then, each edge is considered only once and deciding whether the edge must be added to the tree or not takes a constant number of operations.

Now, let us consider the operations used to update the data structure when an edge is inserted in the tree.

We do the union of to sets $C_{p(u)}$ and $C_{p(v)}$. If this sets are represented as lists with a pointor to its last element, it takes a constant time. Such unions are done $|V(G)|-1$ times.

We also have to update the values of some $p(w)$. Let $x \in V(G)$ and let us estimate the number of updates of $p(x)$ during the execution of the algorithm. Observe that when $p(x)$ is updated, the component of x becomes at least twice bigger. Since, at the end, x belongs to a component of size $|V(G)|$, then $p(x)$ is updated at most $\log_2(|V(G)|)$ times. In total, there are at most $|V(G)|\log_2|V(G)|$ such updates.

Since $|V(G)| \leq |E(G)|+1$, the total time complexity is $O(|E(G)|\log|E(G)|)$.

3.2.3 Application to the Travelling Salesman Problem

Rosenkrantz, Sterns and Lewis considered the special case of the Travelling Salesman Problem (**??**) in which the weights satisfy the *triangle inequality*: $w(xy)+w(yz) \geq w(xz)$, for any three vertices x, y and z.

Problem 3.5 (Metric Travelling Salesman)**.**
Instance: an edge-weighted complete graph (G,w) whose weights satisfy the triangle inequality.
Find: a hamiltonian cycle C of G of minimum weight, i.e. such that $\sum_{e\in E(C)} w(e)$ is minimum.

This problem is $\mathcal{NP}$-hard (see Exercise 3.12) but a polynomial-time 2-approximation algorithm using minimum-weight spanning tree exists.

Theorem 3.6 (Rosenkrantz, Sterns and Lewis)**.** *The Metric Travelling Salesman Problem admits a polynomial-time* 2*-approximation algorithm.*

Proof. Applying Jarník-Prim or Boruvka-Kruskal algorithm, we first find a minimum-weight spanning tree T of G. Suppose that C is a minimum-weight hamiltonian cycle. By deleting any edge of C we obtain a hamiltonian path P of G. Because P is a spanning tree, $w(T) \leq w(P) \leq w(C)$.

We now duplicate each edge of T, thereby obtaining a connected eulerian multigraph H with $V(H)=V(G)$ and $w(H)=2w(T)$. The idea is to transform H into a hamiltonian cycle of G, and to do so without increasing its weight. More precisely, we construct a sequence $H_0, H_1, \ldots, H_{n-2}$ of connected eulerian multigraphs, each with vertex set $V(G)$, such that $H_0 = H$, H_{n-2} is a hamiltonian cycle of G, and $w(H_{i+1}) \leq w(H_i)$, $0 \leq i \leq n-3$. We do so by reducing the number of edges, one at a time, as follows.

Let C_i be an eulerian tour of H_i, where $i < n-2$. The multigraph H_i has $2(n-2)-i > n$ edges, and thus a vertex v has degree at least 4. Let xe_1ve_2y be a segment of the tour C_i; it will follow by induction that $x \neq y$. We replace the edges e_1 and e_2 of C_i by a new edge e of weight $w(xy)$ linking x and y, thereby bypassing v and modifying C_i to an eulerian tour C_{i+1} of $H_{i+1} = (H_i \setminus \{e_1,e_2\}) \cup \{e\}$. By the triangle inequality, we have $w(H_{i+1}) = w(H_i) - w(e_1) - w(e_2) + w(e) \leq w(H_i)$. The final graph H_{n-2}, being a connected eulerian graph on n vertices and n edges, is a hamiltonian cycle of G. Furthermore, $w(H_{n-2}) \leq w(H_0) = 2w(T) \leq 2w(C)$. □

A $\frac{3}{2}$-approximation algorithm for the Metric Travelling Salesman Problem was found by Christofides [2].

3.3 Algorithms in edge-weighted digraphs

Computing shortest paths in directed graphs can be done in much the same way as in undirected graphs by growing arborescences rather than trees. Dijkstra's Algorithm and Bellman-Ford Algorithm translates naturally.

The Minimum-Weight Spanning Tree Problem is equivalent to finding the minimum-weight spanning connected subgraph. The corresponding problem in digraph, namely, finding a connected subdigraph with minimum weight in a connected digraph is much more complex. Actually, this problem is $\mathcal{NP}$-hard even when all edges have same weight because it contains the Directed Hamiltonian Cycle Problem as special case. One can easily describe a polynomial-time 2-approximation algorithm. (See Exercise 3.13). Vetta [6] found a polynomial-time $\frac{3}{2}$-approximation algorithm.

3.4 Exercices

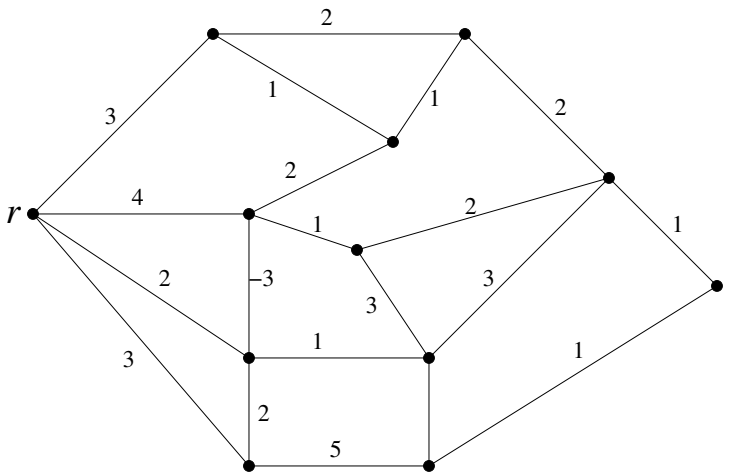
Exercise 3.1. Show a edge-weighted graph G having a vertex u such that no breadth first seach tree from u is a shortest-paths tree.

Exercise 3.2.

Consider the graph depicted in Figure 3.2.

1) Apply Dijkstra's and Bellmann-Ford algorithms for finding a shortest-paths tree from r.

2) Apply the algorithms for finding a minimum-weight spanning tree.



Exercise 3.3. Let (G,w) be a connected edge-weighted graph.
1) Prove that if w is a constant function then every shortest-paths tree is a minimum-weight

spanning tree.
2) Exhibit a connected edge-weighted graph in which there is a shortest-paths tree which is not a minimum-weight spanning tree.

Exercise 3.4. Four imprudent walkers are caught in the storm and night. To reach the hut, they have to cross a canyon over a fragile rope bridge which can resist the weight of at most two persons. In addition, crossing the bridge requires to carry a torch to avoid to step into a hole. Unfortunately, the walkers have a unique torch and the canyon is too large to throw the torch across it. Due to dizziness and tiredness, the four walkers can cross the bridge in 1, 2, 5 and 10 minutes. When two walkers cross the bridge, they both need the torch and thus cross the bridge at the slowest of the two speeds.

With the help of a graph, find the minimum time for the walkers to cross the bridge.

Exercise 3.5. Let T be a minimum-weight spanning tree of an edge-weighted graph (G,w) and T' another spanning tree of G (not necessarily of minimum weight). Show that T' can be transformed into T by successively exchanging an edge of T' by an edge of T so that at each step the obtained graph is a tree and so that the weight of the tree never increases.

Exercise 3.6. Little Attila proposed the following algorithm to solve the Minimum-Weight Spanning Tree Problem: he considers the edges successively in decreasing order with respect to their weight and suppress the ones that are in a cycle of the remaining graph. Does this algorithm give an optimal solution to the problem? Justify your answer.

Exercise 3.7. Let (G,w) be an edge-weighted graph. For all $t \geq 1$, a *t-spanner* of (G,w) is a spanning edge-weighted graph (H,w) of (G,w) such that, for any two vertices u,v, $dist_{H,w}(u,v) \leq t \times dist_{G,w}(u,v)$.

1) Show that (G,w) is the unique 1-spanner of (G,w).
2) Let $k \geq 1$. Prove that the following algorithm returns a $(2k-1)$-spanner of (G,w).

1. Initalise H : $V(H) := V(G)$, $E(H) := \emptyset$. Place the edges in a stack in increasing order with respect to their weight. The minimum weight edge will be on top of the stack.

2. If L is empty then return H. Else remove the edge uv from the top of the stack;

3. If in H there is no (u,v)-path with at most $2k-1$ edges, add e to H.

4. Go to 2.

3) Show that the spanner returned by the above algorithm contains a minimum-weight spanning tree. (*One could show that at each step the connected components of H and the forest computed by Boruvka-Kruskal Algorithm are the same.*)

Exercise 3.8.
We would like to determine a spanning tree with weight close to the minimum. Therefore we study the following question: What is the complexity of the Minimum-Weight Spanning Tree Problem when all the edge-weights belong to a fixed set of size s. (One could consider first the case when the edges have the same weight or weight in $\{1,2\}$.

We assume that the edges have integral weights in $[1,M]$. We replace an edge with weight in $[2^i, 2^{i+1}-1]$ by an edge of weight 2^i. (We *sample* the weight.) Prove that if we compute a minimum-weight spanning tree with the simplified weight then we obtain a tree with weight at most twice the minimum for the original weight.

What happens if we increase the number of sample weights?

Exercise 3.9. 1) Let G be 2-connected edge-weighted graph. (See Chapter **??** for the definition of 2-connectivity.) Show that all the spanning trees have minimum weight if and only if all the edges have the same weight.

2) Give an example of a connected edge-weighted graph for which all the spanning tree have the same weight but whose edges do not all have the same weight.

Exercise 3.10. The *diameter* of an edge-weighted graph (G,w) is the maximum distance between two vertices: $diam(G) := \max\{dist_{G,w}(u,v) \mid u \in V(G), v \in V(G)\}$.
Show that the following algorithm computes the diameter of an edge-weighted tree T.

1. Pick a vertex x of T.
2. Find a vertex y whose distance to x is maximum (*using Dijkstra's Algorithm for example*).
3. Find a vertex z whose distance to y is maximum.
4. Return $dist_{T,w}(y,z)$.

Exercise 3.11. THE POSTMAN PROBLEM
In his job, a postman picks up mail at the post office, delivers it, and then returns to the post office. He must, of course, cover each street at least once. Subject to this condition, he wishes to choose a route entailing as little walking as as possible.

1. Show that the Postman Problem is equivalent to the following graph-theoretic problem.

   Problem 3.7 (MINIMUM-WEIGHT EULERIAN SPANNING SUBGRAPH)**.**
   Instance: an edge-weighted connected graph (G,w) with nonnegative weights.
   Find: by duplicating edges (along with their weights), an eulerian weighted spanning supergraph H of G whose weight $w(H)$ is as small as possible.
   (An Euler tour in H can then be found by applying Algorithm 1.1

2. In the special case where G has just two vertices of odd degree, explain how the above problem can be solved in polynomial time.

Exercise 3.12. Show that the Metric Travelling Salesman Problem is $\mathcal{NP}$-hard.

Exercise 3.13.

1) Let D be a strongly connected digraph on n vertices. A spanning subdigraph of D is *strong-minimal* if it is strongly connected and every spanning proper subdigraph is not strongly connected.

a) Show that in the handle decomposition of a strong-minimal spanning subdigraph all the handles have length at least 2.

b) Deduce that a strong-minimal spanning subdigraph of D has at most $2n-2$ arcs.

2) Describe a polynomial-time 2-approximation for the following problem:
Instance: a strongly connected digraph D.
Find: a strongly connected spanning subdigraph with minimum number of arcs.

Exercise 3.14. An *arborescence* is an orientation of a tree in which every vertex has indegree 1 except one called the root which has indegree 0. The aim of this exercise is to obtain a polynomial-time algorithm for finding a minimum-weight spanning arborescence with prescribed root u in an edge-weighted strong digraph (D,w).

1) Show that if xy is an arc with minimum weight in (D,w) and $y \neq u$, then there is a minimum-weight spanning arborescence with root u containing xy.

2) Deduce a polynomial-time algorithm for finding a minimum-weight spanning arborescence with root u in (D,w).

Bibliography

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley, Reading, MA, 1983.

[2] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Management Sciences Research Report* 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms.* Second Edition. MIT Press, Cambridge, MA, 2001.

[4] D. E. Knuth. *The Art of Computer Programming. Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1969. Second printing.

[5] R. E. Tarjan. *Data Structures and Networks Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 44, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1983.

[6] A. Vetta. Approximating the minimum strongly connected subgraph via a matching lower bound. *Proceedings of the twelfth annual ACM-SIAM symposium on discrete algorithms (SODA 2001)*, pages 417–426, 2001.

Chapter 4

Flow Problems

4.1 Introduction and definitions

Problems related to transport have been investigated since the early fifties. The problem is to route some goods, called *commodities*, from production sites to consumption sites, through a network consisting of communication links inter-connecting the sites (pipe-lines, routes, telecommunication networks). Moreover, each link has a maximum admissible throughput, called the *capacity* of the link.

In general, most of the sites of the network do not produce nor consume anything, they are only used to interconnect links. To each production or consumption site is associated some real number that corresponds to the maximum production or consumption.

The main objective is to maximize the throughput of the traffic.

We first consider networks with directed links, i.e., that can be used in one direction. Formally, a *flow network* is defined as follows.

Definition 4.1 (Flow network)**.** A *flow network* is a four-tuple $(G, pr_{max}, co_{max}, c)$ such that:

- G is a digraph (or a mulitdigraph), the vertices of which represent the sites and the arcs represent the links;
- pr_{max} is a function from $V(G)$ to $\mathbb{R}^+ \cup +\infty$; $pr_{max}(v)$ corresponds to the maximum production possible in v. If v is not a production site, then $pr_{max}(v) = 0$.
- co_{max} is a function from $V(G)$ to $\mathbb{R}^+ \cup +\infty$; $co_{max}(v)$ corresponds to the maximum consumption possible in v. If v is not a consumption site, then $co_{max}(v) = 0$.
- c is a function from $E(G)$ to $\mathbb{R}^+ \cup +\infty$; $c(e)$ corresponds to the capacity of the link e.

Definition 4.2 (Flow)**.** The *flow* is a triple $F = (pr, co, f)$ where

- pr is a function of production such that, for any vertex v, $0 \leq pr(v) \leq pr_{max}(v)$;

- co is a function of consumption such that, for any vertex v, $0 \leq co(v) \leq co_{max}(v)$;

- f is a function over E, called *flow function* that satisfies the following constraints:

Positivity:	$\forall e \in E(G)$,	$f(e) \geq 0$
Capacity constraint:	$\forall e \in E(G)$,	$f(e) \leq c(e)$
Flow conservation:	$\forall v \in V(G)$,	$\sum_{(u,v)\in E(G)} f((u,v)) + pr(v) = \sum_{(v,u)\in E(G)} f((v,u)) + co(v)$

By summing, the $|V(G)|$ equations of flow conservation, we get:

$$\sum_{v\in V(G)} pr(v) = \sum_{v\in V(G)} co(v)$$

In other words, the sum of all produced commodities equals the sum of all consumed commodities. This value corresponds to the amount of routed traffic, it is the *flow value*, denoted by $v(F)$.

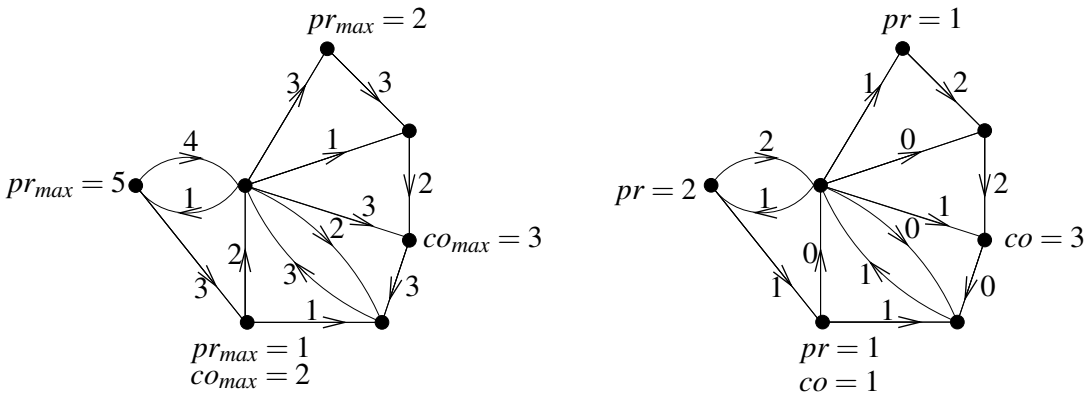


Figure 4.1: Example of flow network (left) and a flow of value 4 in it (right)

Hence, the problem is the following:

Problem 4.3 (Maximum Flow)**.**
Instance: a flow network N.
Find: a flow with maximum value.

A flow with maximum value is said to be a *maximum flow*.

4.2 Reducing to an elementary network

Before studying this problem, we show that it is equivalent to consider an elementary problem where there is a unique production site with infinite maximum production and a unique consumption site with infinite maximum consumption.

Definition 4.4 (Elementary network)**.** A flow network $(G, \infty_s, \infty_t, c)$ is *elementary*:

- s and t are two distinct vertices where s is a *source* (i. e. $d^-(s) = 0$) and t is a *sink* (i.e. $d^+(t) = 0$);
- $\infty_s(s) = +\infty$ and $\infty_s(v) = 0$ for all $v \neq s$;
- $\infty_p(t) = +\infty$ and $\infty_p(v) = 0$ for all $v \neq t$.

We denote $(G, \infty_s, \infty_t, c)$ by (G, s, t, c).

Definition 4.5 (Associated elementary network)**.** If $N = (G, pr_{max}, co_{max}, c)$ is a flow network, its *associated* elementary network is the following elementary network $N = (\bar{G}, s, p, \bar{c})$ obtained from N in the following way:

- add a source s and a sink t;
- link s to all vertices v with non-null production with an arc (s, v) of capacity $pr_{max}(v)$;
- link all vertices v with non-null consumption to t with a link (v,t) of capacity $co_{max}(v)$;

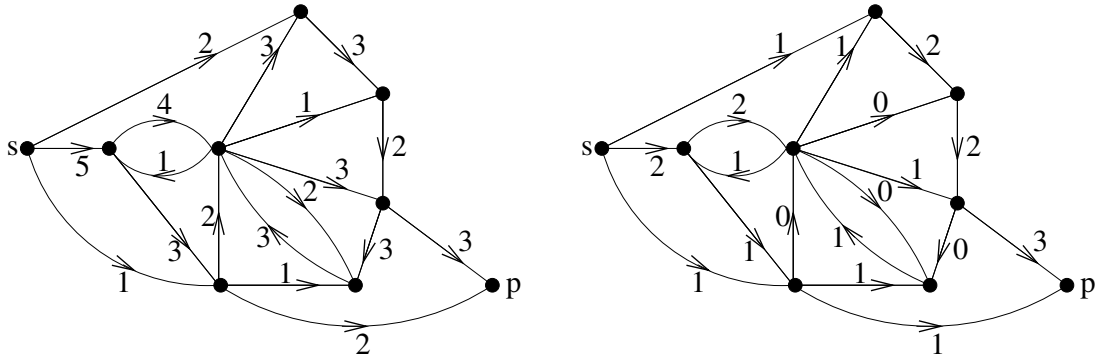


Figure 4.2: Elementary network associated to the flow network depicted in Figure 4.1 (right) and the flow in this network corresponding to the one of Figure 4.1 (left)

It is easy to see that there is a one-to-one correspondence between any flow $F = (pr, co, f)$ of the elementary network and a flow $F' = (pr', co', f')$ of the initial problem defined as follows: for any production site u, $pr'(u) = f((s,u))$; for any consumption site v, $co'(v) = f((v,t))$ and for any arc (x,y), $f'((x,y)) = f((x,y))$.

Clearly, both flows have same value $v(F) = v(F') = pr(s) = co(t)$. Hence:
Finding a maximum flow in a flow network is equivalent to finding a maximum flow solve in its associated elementary network.

In the following, we only consider elementary flow networks.

Note that, in an (elementary) flow network $N = (G, s, t, c)$, the flow conservation constraint can be written:

$$\forall v \in V(G) \setminus \{s,t\}, \sum_{(u,v)\in E(G)} f((u,v)) = \sum_{(v,u)\in E(G)} f((v,u))$$

Moreover, a flow $F = (pr, co, f)$ of an (elementary) flow network is well defined by the flow function f since, by the flow conservation constraint in s and t, we have

$$\begin{aligned} v(F) &= pr(s) = \sum_{u\in V(G),(s,u)\in E(G)} f((s,u)) \\ &= co(t) = \sum_{u\in V(G),(u,t)\in E(G)} f((u,t)) \end{aligned}$$

Therefore, for ease of presentation, we often identify a flow F with its function f, and we note the flow value by $v(f)$.

To simplify the notations in the sequel, for a flow f or a capacity c, and an arc (u,v), we write $f(u,v)$ instead of $f((u,v))$, and $c(u,v)$ instead of $c((u,v))$.

4.3 Cut and upper bound on the maximum flow value

We will show that the value of a maximum flow in a network is limited by the existence of some bottlenecks through which the traffic must go. Roughly, to go from the source to the sink, the flow must cross the border of a set of vertices, its size (the sum of the capacities of the corresponding edges) will limit the value of the flow. Such a border is called a *cut*.

Definition 4.6 (Cut)**.** In a flow network $N = (G, s, t, c)$, an (s,t)-*cut*, or simply a *cut*, is a bipartition $C = (V_s, V_t)$ of the vertices of G such that $s \in V_s$ and $t \in V_t$. The arcs form V_s to V_t (i.e. with tail in V_s and head in V_t) are the *arcs of* C. Their set is denoted by $E(C)$. The *capacity* of the cut C, denoted by $\delta(C)$, is the sum of the capacities of its arcs: $\sum_{e\in E(C)} c(e)$.

Let f be a flow and $C = (V_s, V_t)$ be a cut, $out(f,C)$ denotes the flow on arcs leaving V_s and $in(f,C)$ the flow entering V_s :

$$\begin{aligned} out(f,C) &= \sum_{(u,v)\in E(C),\ u\in V_s, v\in V_t} f(u,v) \\ in(f,C) &= \sum_{(u,v)\in E(G),\ u\in V_t, v\in V_s} f(u,v) \end{aligned}$$

Note that the flow conservation implies that

$$\text{for all cut } C, \quad v(f) = out(f,C) - in(f,C)$$

In particular, the value of the flow is always at most $out(f,C)$. But clearly $out(f,C) \leq \delta(C)$,so

$$v(f) \leq \delta(C). \tag{4.1}$$

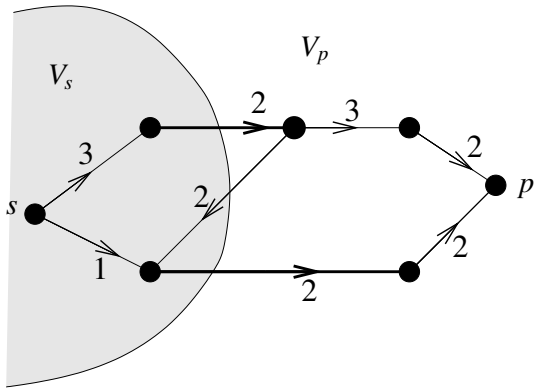


Figure 4.3: A flow network and a cut with capacity 4. Bold arcs are those of the cut

A cut can be viewed as a set of arcs that the flow must cross and whose capacity limits the value of the flow.

Let $v_{max} = max\{v(f), f \text{ a flow}\}$ and $\delta_{min} = min\{\delta(C),\ C \text{ a cut}\}$. Equation 4.1 applied to a maximum flow and a minimum-capacity cut yields

$$v_{max} \leq \delta_{min}. \tag{4.2}$$

In fact, the next theorem,due to Ford and Fulkerson, states that there is equality.

Theorem 4.7 (FORD–FULKERSON THEOREM)**.** *The maximum value of an (s,t)-flow equals the minimum capacity of an (s,t)-cut, i.e.,*

$$v_{max} = \delta_{min}.$$

We often say that *max flow equals min cut*. It is an example of"min-max theorem"

There are many proofs of this theorem, some being non-constructive proof of this theorem. In the next section, we present the original proof which is based on an algorithm computing a maximum flow and a cut with minimum capacity.

4.4 Auxiliary Network and "push" Algorithm

Most of the algorithms for computing maximum flows are based on the following idea: Starting from an existing flow (initially, it may be null), the flow is increased by going from the source to the sink by "pushing" the commodity where it is possible.

The difference between the algorithms mainly consists of the way used to decide where and how to push some flow.

For this purpose, we define an *auxiliary network*. This graph, denoted by $N(f)$, depends on the existing flow f.

Definition 4.8 (Auxiliary network)**.** Given a flow network $N = (G, s, t, c)$ and a flow f, we build the auxiliary network $N(f) = (G(f), s, t, c_f)$ as follows.
For any pair of vertices (u, v), let

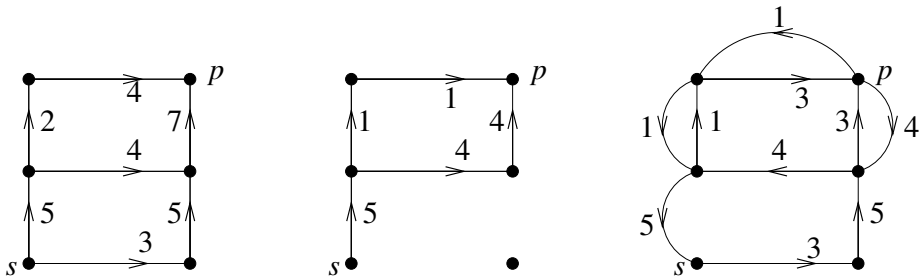
$$c_f(u,v) = c(u,v) - f(u,v) + f(v,u)$$

with $c(u,v)$, $f(u,v)$ and $f(v,u)$ equal to 0 when they are not defined (if (u,v) is not an arc of G). Note that $c_f(u,v) \geq 0$ since $c(u,v) - f(u,v) \geq 0$. Then, $G(f)$ is defined as follows

$$\begin{aligned} V(G(f)) &= V(G) \\ E(G(f)) &= \{(u,v) \mid c_f(u,v) > 0\} \end{aligned}$$

Note that $c_f(u,v) + c_f(v,u) = c(u,v) + c(v,u)$. Intuitively, $c_f(u,v)$ is the sum of the remaining capacity on the arc (u,v), i.e., $c(u,v) - f(u,v)$, plus a virtual capacity $f(v,u)$, that allows to "remove" some flow on the arc (v,u), which corresponds to virtually push some flow along (u,v). See examples in Figures 4.4 and 4.5.

Note that the auxiliary network has no arc with capacity 0. This is important because it ensures that, for any directed path P in $G(f)$, the minimum capacity of the arcs of P is positive.



Flow network Flow Auxiliary network

Figure 4.4: A flow network, a flow and the corresponding auxiliary network

Let P be a directed (s,t)-path in $N(f)$ where the minimum capacity of an arc in P is $\varepsilon > 0$. The flow f' obtained by *pushing* ε *units of flow* along P is defined by:
For any arc $(u,v) \in E(P)$, we push the flow along (u,v), that is, we increase the flow with ε units on (u,v). Two cases may happen:

- If the remaining capacity of (u,v) is sufficient for the ε units of flow, i.e., $f(u,v) + \varepsilon \leq c(u,v)$, then $f'(u,v) = f(u,v) + \varepsilon$ and $f'(v,u) = f(v,u)$.

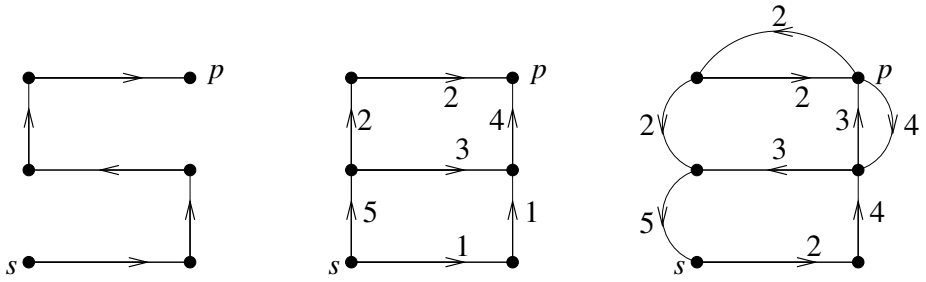
- If the remaining capacity of (u,v) is not sufficient for the ε units of flow, i.e., $f(u,v) + \varepsilon > c(u,v)$, then, by definition of the auxiliary network, we have $f(v,u) \geq f(u,v) + \varepsilon - c(u,v) > 0$. Hence, (v,u) has a flow excess of $f(u,v) + \varepsilon - c(u,v)$ units that we must remove. We set $f'(u,v) = c(u,v)$ and $f'(v,u) = f(v,u) - (f(u,v) + \varepsilon) + c(u,v)$.

If (u,v) and (v,u) do not belong to P, the flow remains unchanged on these arcs, $f'(u,v) = f(u,v)$ and $f'(v,u) = f(v,u)$.

Lemma 4.9. *If f is a flow of value $v(f)$, then f' is a flow of value $v(f)+\varepsilon$.*

Proof. See Exercice 4.5. □

For instance, from the flow depicted in Figure 4.4, by taking the directed (s,t)-path depicted in Figure 4.5 with minimum capacity 1 and pushing the flow, we obtain the new flow and the new auxiliary network depicted in Figure 4.5.



(s,p)-path New flow New auxiliary network

Figure 4.5: Push along a directed (s,t)-path of the auxiliary network in Figure 4.4.

Now, we have some elements of an algorithm:

1. Start with any flow f;
2. Compute the auxiliary network $N(f)$;
3. Find a directed path from s to p in $G(f)$;
4. If such a directed path P exists, then push some flow along P and update f.

Note that, if there is no directed path from the source to the sink in the auxiliary graph, the previous algorithm does nothing. We will see that, in this case, the existing flow is maximum.

Proposition 4.10. *If $G(f)$ does not contain a path from the source to the sink, then the flow f is maximum.*

Proof. Let V_s be the set of all vertices that can be reached from s in $G(f)$. V_s does not contain the sink since there are no (s,t)-paths. Hence, $p \in V_t = V \setminus V_s$. Let C be the cut (V_s, V_t).

Let (u,v) be an arc of C. By definition of V_s, the arc (u,v) is not in $G(f)$. So, by definition of $G(f)$, this means that $c_f(u,v) = 0$. Hence, f is such that $f(u,v) = c(u,v)$ and $f(v,u) = 0$.

We get that $out(f,C) = \delta(C)$ and $in(f,C) = 0$. Informally, there is no flow entering V_s and all arcs of C (that is leaving V_s) are saturated.

But $v(f) = out(f,C) - in(f,C)$, therefore

$$v(f) = \delta(C)$$

The current flow has the same value as the capacity of the cut C. Hence, f is maximum by Equation (4.2). □

Note that this proof also exhibit a cut (V_s, V_t) with same capacity as the maximum value of a flow. Hence it is a minimum-capacity cut. It shows that it is easy to find a minimum-capacity cut (V_s, V_t) from a maximum flow: V_s is the set of the vertices reachable from the source in the auxiliary network and V_t its complement.

Now, we can prove Theorem 4.7.

Proof of Theorem 4.7: Let f be a flow with maximum value in a flow network (G,s,t,c), and let $N(f)$ be the auxiliary network. In $G(f)$, there are no directed (s,t)-paths, otherwise we obtain a flow with greater value by pushing some flow along this path (Lemma 4.9). Hence, by the proof of Proposition 4.10, the cut $C = (V_s, V_t)$, where V_s is the set of vertices v such that there is a directed (s,v)-path in $G(f)$, has capacity $v(f) = v_{max}$. Therefore, $\delta_{min} \leq \delta(C) \leq v_{max}$. □

4.5 Ford-Fulkerson algorithm

We now have the following algorithm:

Algorithm 4.1 (Ford and Fulkerson (1956))**.**

1. Start with null flow $f = 0$;
2. Compute the auxiliary network $N(f)$;
3. Look for a directed path from s to t in $G(f)$;
4. If such a directed path P exists, then push some flow along P, update f and go to 2;
5. Else terminate and return f.

This algorithm is correct in the sense that If the algorithms terminates, then it returns a maximum solution. But

1) does it always terminate?

2) If it terminates, what is its complexity?

The answer to question 1 is somehow yes and no: we will see that the algorithm always terminates if the capacities are rational, but it can take infinite time if capacities are real. Besides, even when the capacities are integers, the running time depends linearly on the value of the maximum flow that may be huge.

Analysis of Ford-Fulkerson Algorithm

Proposition 4.11. *(i) If all capacities are integers, then Algorithm 4.1 terminates after at most v_{max} searches of a directed path.*

(ii) If all the capacities are rational, then Algorithm 4.1 terminates after at most $\mu \cdot v_{max}$ searches of a directed path, with μ the least common multiple of the denominators of the capacities.

Proof. (i) If capacities are integers, at each iteration, the algorithm pushes at least one unit of flow (since, in this case, ε is integral). Each iteration requires the search of a path from s to p in $G(f)$. This can be done in time $O(|E(G)|)$ by any search algorithm (See Chapter 2). Hence, its total running time is at most

$$O(\, v_{max} \cdot |E(G)|).$$

(ii) If capacities are rational, the algorithm terminates since the flow increases of at least $\frac{1}{\mu}$ at each iteration. μ can be very large, but it is fixed. Actually, we can solve the problem by multiplying all capacities by μ and by solving the integral problem obtained: it is proportional the initial one.

□

Remark 4.12. If some capacities are integers, then the Ford-Fulkerson algorithm returns an integral maximum flow.

Proposition 4.13. *If the capacities are real, the algorithm may not terminate. Moreover, the increasing sequence of the flow value may converge to a value a lot smaller than the optimal value.*

Proof. Consider the flow network shown in Figure 4.6 for which the capacities of all edges are infinite (or if the reader prefers, a huge integer M). Let α denote the positive root of $x^3 + x - 1 = 0$. Clearly $1/2 < \alpha < 1$. Let the initial flow f_0 be as shown in Figure 4.6.

We shall prove that by employing a well (or very badly) chosen sequence of four augmenting paths over and over, Algorithm 4.1 will produce an infinite sequence of flows, the values of which are monotone increasing and which converge to a limit not exceeding 16. For any $m \geq 0$, start from the flow f_{4m} and push along the directed path (s,c,d,a,b,t). The pushed amount is α^{4m+1} because of arc (a,b) in the auxiliary network. The resulting flow is f_{4m+1}. Push along the directed path (s,c,b,a,d,t) an amount of α^{4m+2} (because of (c,b)) to produce f_{4m+2}, push along the directed path (s,a,b,c,d,t) an amount of α^{4m+3} (because of (c,d)) to produce

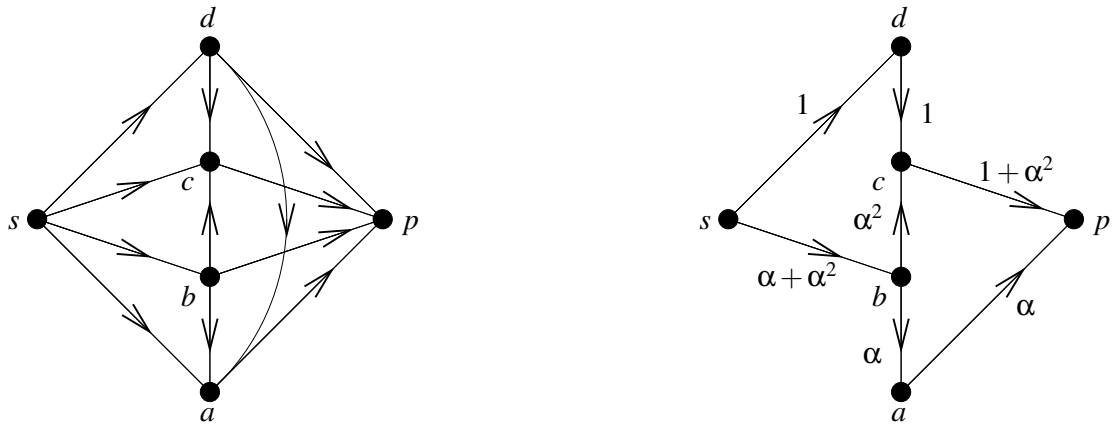


Figure 4.6: A flow network and the initial flow f_0

f_{4m+3} and push along the directed path (s,a,d,c,b,t) an amount of α^{4m+4} (because of (a,d)) to produce f_{4m+4}. See Figure 4.7.

For $k \geq 1$, the augmentation of the value from f_{k-1} to f_k is α^k and hence

$$\begin{aligned}
v(f_k) &= v(f_0) + \alpha + \alpha^2 + \cdots + \alpha^r \\
&= (1 + \alpha + \alpha^2) + \alpha + \alpha^2 + \cdots + \alpha^r \\
&= \frac{1}{\alpha}(\alpha + \alpha^2 + \alpha^3 + \alpha^2 + \alpha^3 + \cdots + \alpha^{r+1}) \\
&= \frac{1}{\alpha}(\alpha + \alpha^2 + (1-\alpha) + \alpha^2 + \alpha^3 + \cdots + \alpha^{r+1}) \\
&= \frac{1}{\alpha}(1 + 2\alpha^2 + \alpha^3 + \cdots + \alpha^{r+1}) \\
&< \frac{1}{\alpha}(1 + \alpha + \alpha^2 + \alpha^3 + \cdots + \alpha^{r+1}) \\
&< \frac{1}{\alpha - \alpha^2} = \frac{1}{\alpha^4} < 16.
\end{aligned}$$

From this construction, it is easy to construct an example of a network where some capacities are irrational such that even starting with a null flow, the sequence of flows obtained by pushing along some directed paths converges to a value less than 16, while the value of a maximum flow is infinite (or arbitrarily large). See Exercise 4.9.

□

Remark 4.14. While the termination is not ensured in case of irrational capacities, Theorem 4.7 ($v_{max} = \delta_{min}$) remains valid. We prove the real case by taking the limit of the rational case. In practice, the rational case is the only important problem, since computers works with a finite precision.

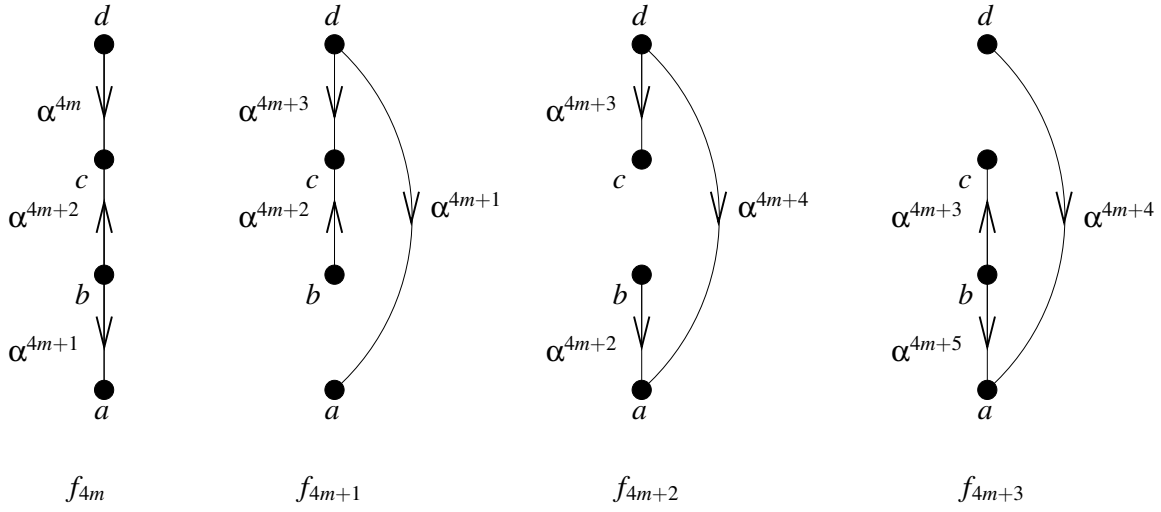


Figure 4.7: The sequence of flows. The arcs leaving s and the arcs entering t are not drawn

The bound of Proposition 4.11 on the number of pushes is not good because it depends on the value of the maximum flow which may be huge. Figure 4.8 shows an example where v_{max} pushes, if they are badly chosen, are necessary to reach a flow with maximum value. Indeed, if

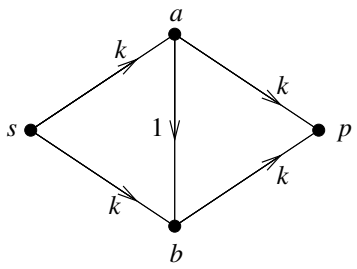


Figure 4.8: Example where $2k = v_{max}$ pushes may be performed.

the pushes are alternatively performed along $P_1 = (s,a,b,t)$ and $P_2 = (s,b,a,t)$, then one unit of flow is pushed at each iteration because the capacity of (a,b) or (b,a) equals 1. Therefore, $2k = v_{max}$ pushes are necessary (k along P_1 and k along P_2).

To improve the previous algorithm and to ensure its quick termination in all cases, the idea is to consider the pushes on some specific directed paths.

4.6 Pushing along shortest paths

Instead of pushing the flow along an arbitrary directed path (s,t)-path in the auxiliary network, a better algorithm consists in pushing along a shortest directed (s,t)-path. Here, by shortest

path, we mean a shortest path in terms of number of arcs, not of total capacity.

Algorithm 4.2 (Edmonds-Karp, 1970)**.**

1. Start with null flow $f = 0$;
2. Compute the auxiliary network $N(f)$;
3. Look for a shortest directed path from s to p in $G(f)$;
4. If such a path P exists, push some flow along P, update f and go to 2;
5. Else terminate and return f.

We will show that such an algorithm performs at most $\frac{|E(G)||V(G)|}{2}$ iterations. That is, its time complexity is at most $\frac{|E(G)||V(G)|}{2}$ times the complexity of finding a shortest path. Note that, contrary to the Ford-Fulkerson Algorithm (4.1), this bound is independent from the capacities.

The proof is based on two simple properties.

1) During the iterations, the distance between s and t in the auxiliary graph cannot decrease.

2) During the iterations, the distance between s and t remains unchanged at most $|E(G)|$ consecutive iterations.

Let f_0 be a flow and let $G_0 = G(f_0)$; We set:

- E' the set of arcs of G_0 that belong to a shortest path from s to t in G_0;
- E'^{-} the arcs obtained by reversing the ones in E' ;
- G_1 the graph obtained from G_0 by adding the arcs in E'^{-}.

Lemma 4.15. *The graph G_1 has the following properties:*

(i) A directed path from s to t with length $dist_{G_0}(s,t)$ does not use any arc in E'^{-}.

(ii) The distance from s to p in G_1 is $dist_{G_0}(s,t)$;

Proof. (i) Let P be a directed (s,t)-path in G_1 that contains some arcs in E'^{-}, and let (v,u) be the last arc of E'^{-} that belongs to P. Let us show that P is not a shortest directed (s,t)-path. Since (u,v) belongs to a shortest directed (s,t)-path, we have $dist_{G_0}(u,p) = dist_{G_0}(v,p) + 1$. Since between u and p, the path P uses only arcs in G_0, its length is $l + 1 + dist_{G_0}(u,p)$ (where l is the length of the subpath of P between s and v). Hence, P has length $l + 2 + dist_{G_0}(v,p)$. The path P' obtained by following P until v and then using a shortest directed (v,p)-path has length $l + dist_{G_0}(v,p) = l + dist_{G_0}(v,p)$. Therefore, P is not a shortest directed (s,t)-path.

(ii) Follows from (i). □

Theorem 4.16. *Algorithm 4.2 performs at most $|E(G)||V(G)|$ iterations. Its time-complexity is $|E(G)||V(G)|$ searches of shortest directed paths, so $O(|E(G)|^2|V(G)|)$.*

Proof. During a push, we add to the auxiliary network the arcs in E'^{-}. By Lemma 4.15, it does not decrease the distance between s and p. The algorithm consists of $|V|$ steps corresponding to the set of all possible distances between s and p. A step consists of a set of iterations when the distance between s and p remains unchanged.

Let us consider successive iterations that let the distance $dist(s,t)$ unchanged. During some iterations, the auxiliary network G_0 changes since some arcs are added and other arcs are removed. However, since $d(s,t)$ remains the same, the paths along which pushes are performed use only arcs of G_0. Since at each iteration, at least one arc of G_0 is removed (we push the maximum possible along the chosen path, so one arc is removed), at most $|E(G)|$ such iterations are performed.

Since $dist(s,t) \leq |V(G)|$, we have at most $|V(G)|$ steps of at most $|E(G)|$ iterations each. At each iteration we mainly have to find a shortest directed (s,t)-path. This can be done in time $O(|E(G)|)$ by any search algorithm (See Chapter 2). Hence the total complexity is $O(|E(G)|^2|V(G)|)$.

□

4.7 Algorithm using a scale factor

If the capacities are integral then Algorithm 4.1 may take time to find the maximum flow becasue of the disparity of the values of the capacities. In the example in Figure 4.8, the pushes may be performed along an arc with capacity 1, while there is a path between the source and the sink with capacity k. This problem may be overcome by using a scale factor. The idea is to try to work with capacities with similar values, to saturate them and then to consider capacities with smaller values. The idea is to start with the greatest capacities, to push some flow in a network that consists only of these links and to try to saturate them as much as possible. Then, the network is replaced by the current auxiliary network to which we add new links with smaller capacities than the one considered yet.

Algorithm 4.3 (Scaling Algorithm)**.**

0. Compute the smallest integer m such that $c(e) < 2^m$ for any arc e.
   For all $e \in E(G)$, set $c(e) = \sum_{j=0}^{m-1} 2^j c_j(e)$ with $c_j(e) \in \{0,1\}$.
1. $k := m-1$; for all $e \in E(G)$, $c'(e) := 0$ and $f(e) := 0$;
2. If $k < 0$, then terminate, else for all $e \in E(G)$, $c'(e) := c'(e) + 2^k c_k(e)$.
3. Increase as much as possible the flow f in (G,s,t,c'); $k := k-1$; go to Step 2.

Clearly, this algorithm computes a maximum flow because, at the end, all capacities are taken into account.

Let us consider the complexity of this algorithm. For this purpose, we need the following proposition the proof of which is left in Exercise 4.10.

Proposition 4.17. *Let $N = (G,s,t,c)$ be a flow network and α a positive real. Let e be an arc of G and N' be the flow network (G,s,t,c') where c' is defined by $c'(e) = c(e) + \alpha$ and $c'(f) = c(f)$ for all $f \neq e$. Then $v_{max}(N') \leq v_{max}(N) + \alpha$.*

Theorem 4.18. *Algorithm 4.3 performs less than $(\log_2(v_{max}) + 1) \cdot |E(G)|$ pushes.*

Proof. Let $c'_i = \sum_{j=i}^{m-1} 2^j c_j$ be the capacity at Step 3 when $k = i$, g_i be the flow added during this step, and f_{i+1} be the total flow before this step. Hence $f_i = f_{i+1} + g_i$.

Let us prove by decreasing induction that at Step 3-(i) Algorithm 4.3 performs at most $|E(G)|$ pushes and that $f_i(e)$ is a multiple of 2^i for every arc e.
The results holds trivially for $i = n$. Suppose now that the results holds for $i+1$. Since the capacity c'_i and the flow f^{i+1} are multiple of 2^i, in the auxiliary network, the capacity of the arcs are multiples of 2^i. Hence finding g_i corresponds to finding a maximum flow g'_i in the network with integral capacities $(G,s,t,(c'_i - f_{i+1})/2^i)$ with $v(g'_i) \leq v(g_i)/2^i$. Moreover, at the end of Step 3-($i+1$), the flow could not be increased anymore, therefore, there are no directed paths with minimum capacity 2^{i+1}. Hence, at Step 3.(i), any directed path in the auxiliary network has minimal capacity 2^i. Hence, each iteration pushes exactly 2^i units of flow at Step 3.(i). Thus $v(g_i) \leq 2^i \cdot |E(G)|$ and so $v(g'_i) \leq 2^i |E(G)|$. By Proposition 4.11, finding g'_i and g_i is done in at most $|E(G)|$ pushes. In addition, by Remark 4.12, the flow g'_i has integral values and so the values of g_i are multiple of 2^i.

Let i_0 be the largest integer such that a push has been done at i_0. If $2^{i_0} \geq v_{max}$, then after pushing once at Step 3-(i_0), we obtained a flow of value 2^{i_0}, which must be a maximum flow.

Otherwise, $i_0 < \log_2(v_{max})$ and for each i, $0 \leq i \leq i_0$, Algorithm 4.3 performs at most $|E(G)|$ pushes. Hence in total, the number of pushes it at most $(i_0+1)|E(G)| \leq (\log_2(v_{max}) + 1) \cdot |E(G)|$. □

4.8 Flows in undirected graphs

Until now, we have considered the problem in directed graphs. However, there are some contexts in which the corresponding network is undirected. For instance, when the links are bidirectional. In this case, each link has one maximum capacity but the flow may circulate in both directions if the sum of the two traffics is at most the capacity of the link.

A *undirected flow network* $N = (G, pr_{max}, co_{max}, c)$ is defined similarily to the directed case. The definition of the flow is a bit modified since two values must be associated to each edge uv: $f(u,v)$ corresponds to the traffic from u to v and $f(v,u)$ corresponds to the traffic from v to u.

Definition 4.19 (Undirected Flow)**.** The *flow* is a three-tuple $F = (pr, co, f)$ where

- pr is a function of production such that, for any vertex v, $0 \leq pr(v) \leq pr_{max}(v)$;
- co is a function of consumption such that, for any vertex v, $0 \leq co(v) \leq co_{max}(v)$;
- f is a function over the ordered pair (u,v) (with u and v adjacent), called *flow function* that satisfies the following constraints:

Positivity:	$\forall e \in E$,	$f(e) \geq 0$
Capacity constraint:	$\forall uv \in E$,	$f((u,v)) + f((v,u)) \leq c(uv)$
Flow conservation:	$\forall v \in V$,	$\sum_{(u,v)\in E} f((u,v)) + pr(v) = \sum_{(v,u)\in E} f((v,u)) + co(v)$

As in the directed case, the *value of the flow F* is

$$v(F) = \sum_{v\in V} pr(v) = \sum_{v\in V} co(v)$$

Let $N = (G, pr_{max}, co_{max}, c)$ be an undirected flow network. The (directed) flow network *associated* to N is $\vec{N} = (\vec{G}, pr_{max}, co_{max}, \vec{c})$ obtained by replacing each edge uv by an arc (u,v) and an arc (v,u) each of which has capacity $c(u,v)$. Formally, $(\vec{G} = (V(G), \bigcup_{uv\in E(G)} \{(u,v),(v,u)\})$ and $\vec{c}((u,v)) = \vec{c}((v,u)) = c(uv)$.

Clearly, a flow function f in N and a flow function $\vec{f}$ in $\vec{N}$ satisfy the same constraints but the capacity constraint. The one in N:

$$\forall uv \in E(G), f((u,v)) + f((v,u)) \leq c(uv)$$

is stronger than the one in $\vec{N}$:

$$\forall uv \in E(G), \vec{f}((u,v)) \leq c(uv) \text{ et } \vec{f}((v,u)) \leq c(uv)$$

Hence, any flow of N is a flow in $\vec{N}$. Hence, the maximum value of a flow in N is at most the maximum value of a flow in $\vec{N}$. In other words, $v_{max}(N) \leq v_{max}(\vec{N})$. We show that they are equal.

Definition 4.20 (simple flow)**.** A flow is *simple* if, for any pair of vertices $\{u,v\}$, we have either $f(u,v) = 0$ or $f(v,u) = 0$. To any flow, there is a corresponding simple flow defined as follows. If $f(u,v) \geq f(v,u) > 0$ then $\tilde{f}(u,v) = f(u,v) - f(v,u)$ and $\tilde{f}(v,u) = 0$. It is easy to see that $v(f) = v(\tilde{f})$ since $f(v,s) = 0$ for any vertex v because $d^-(s) = 0$ and so $f(s,v) = \tilde{f}(s,v)$.

Proposition 4.21. *Let N be an undirected flow network and let $\vec{N}$ be the associated flow network:*

$$v_{max}(N) = v_{max}(\vec{N})$$

Proof. Let $\vec{F} = (pr, co, \vec{f})$ be a flow in $\vec{N}$. Let $F = (pr, co, f)$ be the simple flow of $\vec{F}$. Then $v(F) = v(\vec{F})$. Since F is simple, for any edge $uv \in E(G)$, we have $f(u,v) = 0$ or $f(v,u) = 0$. Besides, the capacity constraint in $\vec{N}$ gives $f(u,v) \leq c(uv)$ ou $f(v,u) \leq c(uv)$. Hence, for any $uv \in E(G)$, $f((u,v)) + f((v,u)) \leq c(uv)$. Then, F is a flow for N.

Therefore, any flow of $\vec{N}$ corresponds to a flow of N with same value. □

This proposition allows us to reduce the undirected problem to the directed case. To find a maximum flow in an undirected network, it is sufficient to solve the problem in the associated directed network and to take the corresponding simple flow.

4.9 Applications of flows

4.9.1 Connectivity in graphs

Menger's Theorem (Theorem **??** is very closely related to Theorem 4.7. Observe that the arc set of an (s,t)-cut in a flow network corresponds to an (s,t)-edge-separator. Hence one can deduce Menger's Theorem from Theorem 4.7. We now do it for Theorem **??**-(ii) for digraphs. In fact the proof of this results we gave in Section **??** was using the push technique (in disguise).

Proof of Theorem **??***-(ii) for digraphs.* Let G be a digraph. The edge-connectivity $\kappa'(s,t)$ is the value of the capacity of a cut in the flow network N obtained by assigning to each arc a capacity of 1, and choosing s as source and t as sink. Indeed, if $C = (V_s, V_t)$ is a cut, then after the removal of the arcs of C, there remain no (s,t)-paths. Hence $\kappa'(s,t) \leq \delta_{min}(N)$. Reciprocally, let E' be an arc-separator of G and $G' = G \setminus E'$. Let V_s be the set of vertices w of G' such that there exists a directed (s,w)-path in G', and $V_t = V \setminus V_u$. By definition of V_s, any arc (x,y) with $x \in V_s$ and $y \in V_t$ is in E'. Hence, $\delta((V_s,V_t)) \leq |E'|$. Therefore, $\delta_{min}(N) \leq \kappa'(s,t)$.

From Theorem 4.7, there is a flow of value $\kappa'(s,t)$ in this network. Moreover, by Remark 4.12, we may assume that this flow is integral. Since the edges have unit capacity, the flow of value $\kappa'(s,t)$ can be decomposed into a set of $\kappa'(s,t)$ pairwise edge-disjoint (s,t)-paths. Hence $\kappa'(s,t) = \Pi'(s,t)$. □

The other cases of Menger's Theorem may also be derived from Theorem 4.7. See Exercise 4.13. Thus Menger's Theorem can be viewed as a particular case of Theorem 4.7. In fact, they are equivalent and it is not too difficult to prove Theorem 4.7 from Menger's Theorem. See Exercise 4.14.

4.9.2 Maximum matching in bipartite graphs

The theorems on matching in bipartite graphs that we proved in Chapter **??** are direct applications of flows. Indeed to every bipartite graph $G = ((A,B),E)$, one can associate the flow network $N_G = (H,s,t,c)$ defined as follows: H is the digraph obtained from G by orienting all edges of G from A to B and adding a source s and a sink t, all arcs (s,a) for $a \in A$ and (b,t) for $b \in B$; the capacity equals 1 for all arcs. See Figure 4.9.

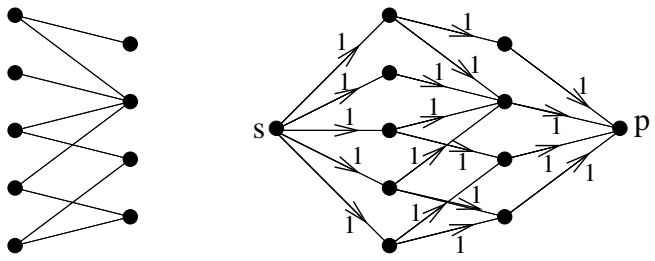


Figure 4.9: A bipartite graph and its associated flow network

There is a one-to-one correspondence between the matchings of G and the integral flows in N_G: to every matching M corresponds the flow f_M with value 1 on the arc set $\bigcup_{(a,b)\in E(G)}\{(s,a),(a,b),(b,t)\}$. Moreover the size of M is equal to the value of the flow f_M. In addition, an M-augmenting path in G corresponds to a directed (s,t)-path in the auxiliary network $N_G(f_M)$. Hence Algorithm **??** is a particular case of Algorithm 4.1.

One can also deduce all the results of Chapter **??** from Theorem 4.7. For example, we now give a proof of Theorem **??** (*"Let $G=(A,B)$ bipartite, there is a matching of size k if and only if $\forall S\subset A, |A|-|S|+|N(S)|\geq k$"*) using flows.

Proof of Theorem **??**. Let us prove that the maximum size μ of a matching in G is equal to the maximum value of a flow in N_G. If M is a matching of size μ in G, then f_M has value μ. Hence $\mu \leq v_{max}$. Reciprocally, from Remark 4.12, there is a maximum flow f with integral values. It is in one-to-one correspondence with a matching M of size $v(f)=v_{max}$. Hence, $\mu \geq v_{max}$.

Let $C=(V_s,V_t)$ be a minimum cut. Let $A_s=A\cap V_s$ and $B_s=B\cap V_s$. If there is a vertex $b\in(B\cap N(A_s))\setminus B_s$, then setting $C'=(V_s\cup\{b\},V_t\setminus\{b\})$, we get $\delta(C')\leq\delta(C)-1+1=\delta(C)$. So, by adding the vertices of $(B\cap N(A_s))\setminus B_s$ in V_s if needed, we may assume that the minimum cut that we consider is such that $N(A_s)\subseteq B_s$.

Let us consider the capacity of C.

$$\begin{aligned}\delta_{min}=\delta(C) &= |\{(s,a)\mid a\in A\setminus A_s\}|+|\{(b,t)\mid b\in B_s\}|+|\{(a,b)\}\mid a\in A_s, b\in B\setminus B_s\}|\\ &= |A|-|A_s|+|B_s|\end{aligned}$$

Since $N(A_s)\subseteq B_s$, it follows that $\delta_{min}\geq |A|-|A_s|+|N(A_s)|$. Moreover, the cut $(A_s\cup N(A_s), V(H)\setminus(A_s\cup N(A_s)))$ has capacity $|A|-|A_s|+|N(A_s|$. So $\delta_{min}=|A|-|A_s|+|N(A_s)|$. We conclude that

$$\delta_{min}=\min\{S\subset A\mid |A|-|S|+N(S)\}$$

Hence, from Theorem 4.7, $\mu=v_{max}=\delta_{min}=\min\{S\subset A\mid |A|-|S|+N(S)\}$. □

4.9.3 Maximum-gain closure

In this problem, we have several jobs. Each job $j\in J$ is associated to a gain $g(j)$. The gain may be negative (if so, it corresponds to a loss). We note J^+ (resp. J^-) the set of jobs with positive gain (resp., negative gain).

Besides, there are several *closure* constraints.That is, the choice of a job may imply the choice of one or several other jobs. We represent this closure relation (e.g., implication relation) by an *implication digraph* D_J: its vertices are the jobs and there is an arc (j_1,j_2) if and only if the choice of j_1 implies the choice of j_2.

The objective is to find a set of *compatible*jobs with maximum gain, that is a subset A of J such that:

- there are no arcs leaving A ($E((A,\overline{A}))=\emptyset$) i.e. A is a *closure* in D_J;
- $\sum_{j\in A} g(j)$ is maximum.

Let us show how this problem can be reduced to a problem of cut with minimum capacity, hence to a maximum flow problem. Let $N = (G, s, t, c)$ be the following flow network (See Figure 4.10:

- $V(G) = T \cup \{s,t\}$;
- for any job j with positive gain, link the source s to the job j with an arc (s, j) with capacity $c(s, j) = g(j)$;
- for any job j with non-positive gain, link j to the sink t with an arc (j,t) with capacity $c(j,t) = -g(j)$;
- if a job j_1 implies a job j_2, we add the arc (j_1, j_2) with infinite capacity in the network.

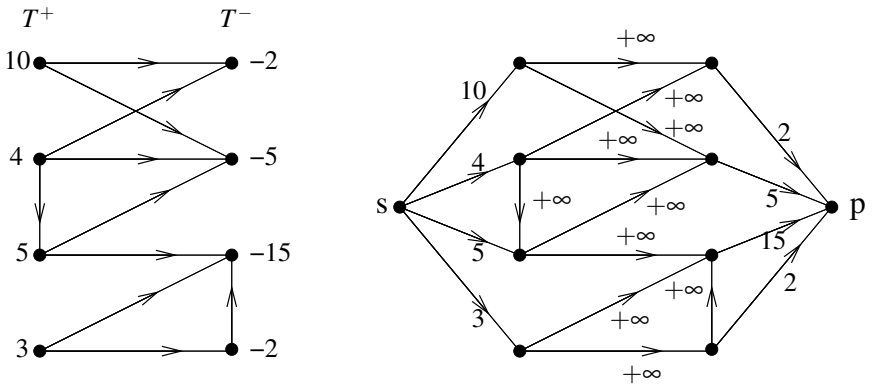


Figure 4.10: An implication digraph and its corresponding flow network

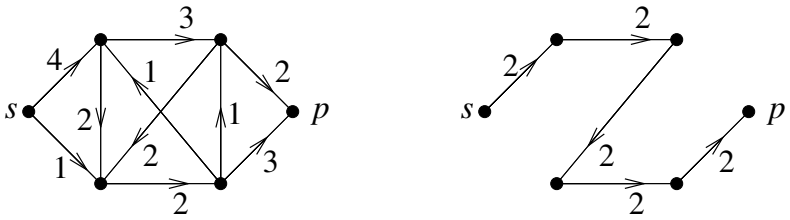
Let us consider a cut $C = (V_s, V_t)$ with finite capacity in the network. We have $S = \{s\} \cup A$ with $A \subset J$. Moreover, since the cut has finite capacity, its arc set contains no arcs of D_J, so A is a set of compatible jobs. Let $A^+ = A \cap J^+$ and $A^- = A \cap J^-$. The capacity of C is:

$$\begin{aligned}
\delta(C) &= \sum_{j \in J^+ \setminus A^+} c(s, j) + \sum_{j \in A^-} c(j,t) \\
&= \sum_{j \in J^+ \setminus A^+} g(j) - \sum_{j \in A^-} g(j) \\
&= \sum_{j \in J^+} g(j) - \sum_{j \in A^+} g(j) - \sum_{j \in A^-} g(j) \\
&= \sum_{j \in J^+} g(j) - \sum_{j \in A} g(j)
\end{aligned}$$

Since $\sum_{j \in J^+} g(j)$, the sum of all positive gains, is a constant, minimizing the capacity of the cut $S = \{s\} \cup A$ is equivalent to maximizing the gain of A.

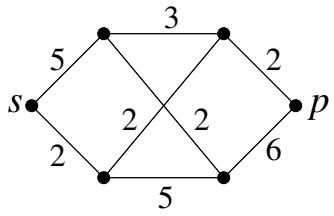
4.10 Exercices

Exercise 4.1. Let N be the flow network and f_0 the (s,t)-flow in N as depicted in the figure below.



1) Start from f_0 and find a maximum (s,t)-flow. Detail the steps of the algorithm.
2) Describe a minimum cut of this network.

Exercise 4.2. Find a maximum (s,t)-flow and a minimum (s,t)-cut in the network depicted below. (Detail the steps of the "push" algorithm.)



Exercise 4.3. There are 3 production sites A,B,C and 5 consumption sites $1,2,3,4,5$; their production and consumption, respectively, are given in the following tables.

A	B	C
5	4	7

1	2	3	4	5
3	4	5	2	1

Finally, each production sites can only serve the consumption sites as summarized in the following table.

A	B	C
13	24	345

The problem is to satisfy the consumption sites. Model the following problem in terms of flows and give a solution to the problem or explain why it could not exist.

Exercise 4.4. Prove the following property: *for all* (s,t)*-cut* $C = (V_s, V_t)$, $v(f) = out(f,C) - in(f,C)$. Why is the hypothesis $s \in V_s$ and $p \in V_t$ important?

Exercise 4.5. Prove Lemma 4.9. Verify that the positivity, the capacity constraint and the flow conservation are satisfied for f'.

Exercise 4.6. The *support* of a flow is the set of arcs on which the flow function is positive. Show that there always exists a maximum flow whose support has no directed cycle.

Exercise 4.7. Let $N = (G,s,t,c)$ be a flow network such that, for all arc e, $c(e)$ is an even integer.
1) Prove that the maximum value of a flow is an even integer.
2) Show that there is a maximum flow f such that, for all arc e, $f(e)$ is an even integer.

Exercise 4.8. Modify Algorithm 4.1 to obtain an algorithm finding a minimum cut in a flow network.

Exercise 4.9. Construct a flow network for which Algorithm 4.1 produces a sequence of flow whose values converges to a finite value when pushing on some sequence of directed paths, while the value of a maximum flow is infinite (or arbitrarily large).

Exercise 4.10. Prove Proposition 4.17.

Exercise 4.11. In this exercise, we study a variant of the algorithm for finding a maximum flow, in which we push the flow along a directed path of maximum residual capacity.

1) Give an algorithm finding a directed path of maximum residual capacity.

2) Show that if we push an amount of x at one step, then we push an amount of at least x at each following step.

Exercise 4.12. Let $N = (G,s,t,c)$ be a flow network. To each vertex $v \in V(G)$, we associate an real $w(v)$. We want to compute a flow f of maximum value satisfying the following extra constraint: $\forall v \in V(G)$ the flow entering v is at most $w(v)$ (i.e. $\sum_{u \in N^-(v)} f(uv) \leq w(v)$). Show how to find such a flow by computing a maximum flow on a network obtained from N by slight modifications.

Exercise 4.13. Deduce Menger's Theorem (**??**) from Theorem 4.7 . (*Hint*: One can use Exercise 4.12 to prove Theorem **??**-(i).

Exercise 4.14. Deduce Theorem 4.7 from Menger's Theorem (**??**).

Exercise 4.15. Several companies send members to a conference; the ith company send m_i members. During the conference, several workshops are organized simulteanously; the ith workshops can receive at most n_j participants. The organizers want to dispatch participants into workshops so that two members of a same company are not in a same workshop. (The workshop do not need to be full.)

a) Show how to use a flow network for testing if the constraints may be satisfied.

b) If there are p companies and q workshops indexed in such a way that $m_1 \geq \cdots \geq m_p$ and $n_1 \leq \cdots \leq n_q$. Show that there exists a dispatching of participants into groupes satisfying the constraints if and only if, for all $0 \leq k \leq p$ and all $0 \leq l \leq q$, we have $k(q-l) + \sum_{j=1}^{l} n_j \geq \sum_{i=1}^{k} m_i$.

Exercise 4.16 (Unsplittable flow)**.** We consider a flow network with one production site s and many consumption sites, say $t_1, t_2, \ldots t_n$. The consumption at site t_i is d_i. We want to route commodities for s to t_i with the following additionnal constraint: the traffic from s to t_i must be routed along a unique directed path (it cannot be split).

Show that this problem is NP-complete.

Chapter 5

Linear programming

The nature of the programmes a computer scientist has to conceive often requires some knowledge in a specific domain of application, for example corporate management, network protocols, sound and video for multimedia streaming,... Linear programming is one of the necessary knowledges to handle optimization problems. These problems come from varied domains as production management, economics, transportation network planning, ... For example, one can mention the composition of train wagons, the electricity production, or the flight planning by airplane companies.

Most of these optimization problems do not admit an optimal solution that can be computed in a reasonable time, that is in polynomial time (See Chapter **??**). However, we know how to efficiently solve some particular problems and to provide an optimal solution (or at least quantify the difference between the provided solution and the optimal value) by using techniques from linear programming.

In fact, in 1947, G.B. Dantzig conceived the Simplex Method to solve military planning problems asked by the US Air Force that were written as a linear programme, that is a system of linear equations. In this course, we introduce the basic concepts of linear programming. We then present the Simplex Method, following the book of V. Chvátal [2]. If you want to read more about linear programming, some good references are [6, 1].

The objective is to show the reader how to model a problem with a linear programme when it is possible, to present him different methods used to solve it or at least provide a good approximation of the solution. To this end, we present the *theory of duality* which provide ways of finding good bounds on specific solutions.

We also discuss the practical side of linear programming: there exist very efficient tools to solve linear programmes, e.g. CPLEX [3] and GLPK [4]. We present the different steps leading to the solution of a practical problem expressed as a linear programme.

5.1 Introduction

A *linear programme* is a problem consisting in maximizing or minimizing a linear function while satisfying a finite set of linear constraints.

Linear programmes can be written under the *standard form*:

$$\begin{array}{lrcll} \text{Maximize} & \sum_{j=1}^{n} c_j x_j & & & \\ \text{Subject to:} & \sum_{j=1}^{n} a_{ij} x_j & \leq & b_i & \text{for all } 1 \leq i \leq m \\ & x_j & \geq & 0 & \text{for all } 1 \leq j \leq n. \end{array} \tag{5.1}$$

All constraints are inequalities (and not equations) and all variables are non-negative. The variables x_j are referred to as *decision variables*. The function that has to be maximized is called the problem *objective function*.

Observe that a constraint of the form $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ may be rewritten as $\sum_{j=1}^{n} (-a_{ij}) x_j \leq -b_i$. Similarly, a minimization problem may be transformed into a maximization problem: minimizing $\sum_{j=1}^{n} c_j x_j$ is equivalent to maximizing $\sum_{j=1}^{n} (-c_j) x_j$. Hence, every maximization or minimization problem subject to linear constraints can be reformulated in the standard form (See Exercices 5.1 and 5.2.).

A n-tuple $(x_1, \ldots, x_n)$ satisfying the constraints of a linear programme is a *feasible solution* of this problem. A solution that maximizes the objective function of the problem is called an *optimal solution*. Beware that a linear programme does not necessarily admits a unique optimal solution. Some problems have several optimal solutions while others have none. The later case may occur for two opposite reasons: either there exist no feasible solutions, or, in a sense, there are too many. The first case is illustrated by the following problem.

$$\begin{array}{lrcrcr} \text{Maximize} & 3x_1 & - & x_2 & & \\ \text{Subject to:} & x_1 & + & x_2 & \leq & 2 \\ & -2x_1 & - & 2x_2 & \leq & -10 \\ & & & x_1, x_2 & \geq & 0 \end{array} \tag{5.2}$$

which has no feasible solution (See Exercise 5.3). Problems of this kind are referred to as *unfeasible*. At the opposite, the problem

$$\begin{array}{lrcrcr} \text{Maximize} & x_1 & - & x_2 & & \\ \text{Subject to:} & -2x_1 & + & x_2 & \leq & -1 \\ & -x_1 & - & 2x_2 & \leq & -2 \\ & & & x_1, x_2 & \geq & 0 \end{array} \tag{5.3}$$

has feasible solutions. But none of them is optimal (See Exercise 5.3). As a matter of fact, for every number M, there exists a feasible solution x_1, x_2 such that $x_1 - x_2 > M$. The problems verifying this property are referred to as *unbounded*. Every linear programme satisfies exactly one the following assertions: either it admits an optimal solution, or it is unfeasible, or it is unbounded.

Geometric interpretation.

The set of points in $\mathbb{R}^n$ at which any single constraint holds with equality is a hyperplane in $\mathbb{R}^n$. Thus each constraint is satisfied by the points of a closed half-space of $\mathbb{R}^n$, and the set of feasible solutions is the intersection of all these half-spaces, a convex polyhedron P.

Because the objective function is linear, its level sets are hyperplanes. Thus, if the maximum value of $\mathbf{cx}$ over P is z^*, the hyperplane $\mathbf{cx} = z^*$ is a supporting hyperplane of P. Hence $\mathbf{cx} = z^*$ contains an extreme point (a corner) of P. It follows that the objective function attains its maximum at one of the extreme points of P.

5.2 The Simplex Method

The authors advise you, in a humanist élan, to skip this section if you are not ready to suffer. In this section, we present the principle of the Simplex Method. We consider here only the most general case and voluntarily omit here the degenerate cases to focus only on the basic principle. A more complete presentation can be found for example in [2].

5.2.1 A first example

We illustrate the Simplex Method on the following example:

$$\begin{array}{lrcrcrcr} \text{Maximize} & 5x_1 & + & 4x_2 & + & 3x_3 & & \\ \text{Subject to:} & & & & & & & \\ & 2x_1 & + & 3x_2 & + & x_3 & \leq & 5 \\ & 4x_1 & + & x_2 & + & 2x_3 & \leq & 11 \\ & 3x_1 & + & 4x_2 & + & 2x_3 & \leq & 8 \\ & & & & & x_1,x_2,x_3 & \geq & 0. \end{array} \tag{5.4}$$

The first step of the Simplex Method is to introduce new variables called *slack variables*. To justify this approach, let us look at the first constraint,

$$2x_1 + 3x_2 + x_3 \leq 5. \tag{5.5}$$

For all feasible solution x_1,x_2,x_3, the value of the left member of (5.5) is at most the value of the right member. But, there often is a gap between these two values. We note this gap x_4. In other words, we define $x_4 = 5 - 2x_1 - 3x_2 - x_3$. With this notation, Equation (5.5) can now be written as $x_4 \geq 0$. Similarly, we introduce the variables x_5 and x_6 for the two other constraints of Problem (5.4). Finally, we use the classic notation z for the objective function $5x_1 + 4x_2 + 3x_3$. To summarize, for all choices of x_1,x_2,x_3 we define x_4,x_5,x_6 and z by the formulas

$$\begin{array}{rcrcrcrcr} x_4 & = & 5 & - & 2x_1 & - & 3x_2 & - & x_3 \\ x_5 & = & 11 & - & 4x_1 & - & x_2 & - & 2x_3 \\ x_6 & = & 8 & - & 3x_1 & - & 4x_2 & - & 2x_3 \\ z & = & & & 5x_1 & + & 4x_2 & + & 3x_3. \end{array} \tag{5.6}$$

With these notations, the problem can be written as:

$$\text{Maximize } z \text{ subject to } x_1,x_2,x_3,x_4,x_5,x_6 \geq 0. \tag{5.7}$$

The new variables that were introduced are referred as *slack variables*, when the initial variables are usually called the *decision variables*. It is important to note that Equation (5.6) define an equivalence between (5.4) and (5.7). More precisely:

- Any feasible solution (x_1,x_2,x_3) of (5.4) can be uniquely extended by (5.6) into a feasible solution $(x_1,x_2,x_3,x_4,x_5,x_6)$ of (5.7).

- Any feasible solution $(x_1,x_2,x_3,x_4,x_5,x_6)$ of (5.7) can be reduced by a simple removal of the slack variables into a feasible solution (x_1,x_2,x_3) of (5.4).
- This relationship between the feasible solutions of (5.4) and the feasible solutions of (5.7) allows to produce the optimal solution of (5.4) from the optimal solutions of (5.7) and *vice versa*.

The Simplex strategy consists in finding the optimal solution (if it exists) by successive improvements. If we have found a feasible solution (x_1,x_2,x_3) of (5.7), then we try to find a new solution $(\bar{x_1},\bar{x_2},\bar{x_3})$ which is better in the sense of the objective function:

$$5\bar{x_1}+4\bar{x_2}+3\bar{x_3} \geq 5x_1+4x_2+3x_3.$$

By repeating this process, we obtain at the end an optimal solution.

To start, we first need a feasible solution. To find one in our example, it is enough to set the decision variables x_1,x_2,x_3 to zero and to evaluate the slack variables x_4,x_5,x_6 using (5.6). Hence, our initial solution,

$$x_1=0,x_2=0,x_3=0,x_4=5,x_5=11,x_6=8 \tag{5.8}$$

gives the result $z=0$.

We now have to look for a new feasible solution which gives a larger value for z. Finding such a solution is not hard. For example, if we keep $x_2=x_3=0$ and increase the value of x_1, then we obtain $z=5x_1\geq 0$. Hence, if we keep $x_2=x_3=0$ and if we set $x_1=1$, then we obtain $z=5$ (and $x_4=3,x_5=7,x_6=5$). A better solution is to keep $x_2=x_3=0$ and to set $x_1=2$; we then obtain $z=10$ (and $x_4=1,x_5=3,x_6=2$). However, if we keep $x_2=x_3=0$ and if we set $x_1=3$, then $z=15$ and $x_4=x_5=x_6=-1$, breaking the constraint $x_i\geq 0$ for all i. The conclusion is that one can not increase x_1 as much as one wants. The question then is: how much can x_1 be raised (when keeping $x_2=x_3=0$) while satisfying the constraints $(x_4,x_5,x_6\geq 0)$?

The condition $x_4=5-2x_1-3x_2-x_3\geq 0$ implies $x_1\leq\frac{5}{2}$. Similarly, $x_5\geq 0$ implies $x_1\leq\frac{11}{4}$ and $x_6\geq 0$ implies $x_1\leq\frac{8}{3}$. The first bound is the strongest one. Increasing x_1 to this bound gives the solution of the next step:

$$x_1=\frac{5}{2},x_2=0,x_3=0,x_4=0,x_5=1,x_6=\frac{1}{2} \tag{5.9}$$

which gives a result $z=\frac{25}{2}$ improving the last value $z=0$ of (5.8).

Now, we have to find a new feasible solution that is better than (5.9). However, this task is not as simple as before. Why? As a matter of fact, we had at disposal the feasible solution of (5.8), but also the system of linear equations (5.6) which led us to a better feasible solution. Thus, we should build a new system of linear equations related to (5.9) in the same way as (5.6) is related to (5.8).

Which properties should have this new system? Note first that (5.6) express the strictly positive variables of (5.8) in function of the null variables. Similarly, the new system has to express the strictly positive variables of (5.9) in function of the null variables of (5.9): x_1,x_5,x_6 (and z) in function of x_2,x_3 and x_4. In particular, the variable x_1, whose value just increased

from zero to a strictly positive value, has to go to the left side of the new system. The variable x_4, which is now null, has to take the opposite move.

To build this new system, we start by putting x_1 on the left side. Using the first equation of (5.6), we write x_1 in function of x_2, x_3, x_4:

$$x_1 = \frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4 \tag{5.10}$$

Then, we express x_5, x_6 and z in function of x_2, x_3, x_4 by substituting the expression of x_1 given by (5.10) in the corresponding lines of (5.6).

$$\begin{aligned}
x_5 &= 11 - 4\left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) - x_2 - 2x_3 \\
&= 1 + 5x_2 + 2x_4, \\
x_6 &= 8 - 3\left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) - 4x_2 - 2x_3 \\
&= \frac{1}{2} + \frac{1}{2}x_2 - \frac{1}{2}x_3 + \frac{3}{2}x_4, \\
z &= 5\left(\frac{5}{2} - \frac{3}{2}x_2 - \frac{1}{2}x_3 - \frac{1}{2}x_4\right) + 4x_2 + 3x_3 \\
&= \frac{25}{2} - \frac{7}{2}x_2 + \frac{1}{2}x_3 - \frac{5}{2}x_4.
\end{aligned}$$

So the new system is

$$\begin{array}{rcrcrcrcrl}
x_1 &=& \frac{5}{2} &-& \frac{3}{2} & x_2 &-& \frac{1}{2} & x_3 &-& \frac{1}{2} & x_4 \\
x_5 &=& 1 &+& 5 & x_2 & & & & + & 2 & x_4 \\
x_6 &=& \frac{1}{2} &+& \frac{1}{2} & x_2 &-& \frac{1}{2} & x_3 &+& \frac{3}{2} & x_4 \\
z &=& \frac{25}{2} &-& \frac{7}{2} & x_2 &+& \frac{1}{2} & x_3 &-& \frac{5}{2} & x_4.
\end{array} \tag{5.11}$$

As done at the first iteration, we now try to increase the value of z by increasing a right variable of the new system, while keeping the other right variables at zero. Note that raising x_2 or x_4 would lower the value of z, against our objective. So we try to increase x_3. How much? The answer is given by (5.11) : with $x_2 = x_4 = 0$, the constraint $x_1 \geq 0$ implies $x_3 \leq 5$, $x_5 \geq 0$ impose no restriction and $x_6 \geq 0$ implies that $x_3 \leq 1$. To conclude $x_3 = 1$ is the best we can do, and the new solution is

$$x_1 = 2, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 0 \tag{5.12}$$

and the value of z increases from 12.5 to 13. As stated, we try to obtain a better solution but also a system of linear equations associated to (5.12). In this new system, the (strictly) positive variables x_2, x_4, x_6 have to appear on the right. To build this new system, we start by handling the new left variable, x_3. Thanks to the third equation of (5.11) we rewrite x_3 and by substitution

in the remaining equations of (5.11) we obtain:

$$\begin{array}{rcrcrcrcr} x_3 & = & 1 & + & x_2 & + & 3x_4 & - & 2x_6 \\ x_1 & = & 2 & - & 2x_2 & - & 2x_4 & + & x_6 \\ x_5 & = & 1 & + & 5x_2 & + & 2x_4 & & \\ z & = & 13 & - & 3x_2 & - & x_4 & - & x_6. \end{array} \tag{5.13}$$

It is now time to do the third iteration. First, we have to find a variable of the right side of (5.13) whose increase would result in an increase of the objective z. But there is no such variable, as any increase of x_2, x_4 or x_6 would lower z. We are stuck. In fact, this deadlock indicates that the last solution is optimal. Why? The answer lies in the last line of (5.13):

$$z = 13 - 3x_2 - x_4 - x_6. \tag{5.14}$$

The last solution (5.12) gives a value $z = 13$; proving that this solution is optimal boils down to prove that any feasible solution satisfies $z \leq 13$. As any feasible solution $x_1, x_2, \ldots, x_6$ satisfies the inequalities $x_2 \geq 0, x_4 \geq 0, x_6 \geq 0$, then $z \leq 13$ directly derives from (5.14).

5.2.2 The dictionaries

More generally, given a problem

$$\begin{array}{lrcll} \text{Maximize} & \sum_{j=1}^{n} c_j x_j & & & \\ \text{Subject to:} & \sum_{j=1}^{n} a_{ij} x_j & \leq & b_i & \text{for all } 1 \leq i \leq m \\ & x_j & \geq & 0 & \text{for all } 1 \leq j \leq n \end{array} \tag{5.15}$$

we first introduce the *slack variables* $x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ and we note the objective function z. That is, we define

$$\begin{array}{rcll} x_{n+i} & = & b_i - \sum_{j=1}^{n} a_{ij} x_j & \text{for all } 1 \leq i \leq m \\ z & = & \sum_{j=1}^{n} c_j x_j & \end{array} \tag{5.16}$$

In the framework of the Simplex Method, each feasible solution $(x_1, x_2, \ldots, x_n)$ of (5.15) is represented by $n+m$ positive or null numbers $x_1, x_2, \ldots, x_{n+m}$, with $x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ defined by (5.16). At each iteration, the Simplex Method goes from one feasible solution $(x_1, x_2, \ldots, x_{n+m})$ to an other feasible solution $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_{n+m})$, which is better in the sense that

$$\sum_{j=1}^{n} c_j \bar{x}_j > \sum_{j=1}^{n} c_j x_j.$$

As we have seen in the example, it is convenient to associate a system of linear equations to each feasible solution. As a matter of fact, it allows to find better solutions in an easy way. The technique is to translate the choices of the values of the variables of the right side of the system into the variables of the left side and in the objective function as well. These systems have been named *dictionaries* by J.E. Strum (1972). Thus, every dictionary associated to (5.15) is a system of equations whose variables $x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ and z are expressed in function of $x_1, x_2, \ldots, x_n$. These $n+m+1$ variables are closely linked and every dictionary express these dependencies.

Property 5.1. *Any feasible solution of the equations of a dictionary is also a feasible solution of (5.16) and* vice versa*.*

For example, for any choice of $x_1, x_2, \ldots, x_6$ and of z, the three following assertions are equivalent:

- $(x_1, x_2, \ldots, x_6, z)$ is a feasible solution of (5.6);
- $(x_1, x_2, \ldots, x_6, z)$ is a feasible solution of (5.11);
- $(x_1, x_2, \ldots, x_6, z)$ is a feasible solution of (5.13).

From this point of view, the three dictionaries (5.6), (5.11) and (5.13) contain the same information on the dependencies between the seven variables. However, each dictionary present this information in a specific way. (5.6) suggests that the values of the variables x_1, x_2 and x_3 can be chosen at will while the values of x_4, x_5, x_6 and z are fixed. In this dictionary, the decision variables x_1, x_2, x_3 act as independent variables while the slack variables x_4, x_5, x_6 are related to each other. In the dictionary (5.13), the independent variables are x_2, x_4, x_6 and the related ones are x_3, x_1, x_5, z.

Property 5.2. *The equations of a dictionary have to express m variables among $x_1, x_2, \ldots, x_{n+m}, z$ in function of the n remaining others.*

Properties 5.1 and 5.2 define what a dictionary is. In addition to these two properties, the dictionaries (5.6),(5.11) and (5.13) have the following property.

Property 5.3. *When putting the right variables to zero, one obtains a feasible solution by evaluating the left variables.*

The dictionaries that have this last property are called *feasible dictionaries*. As a matter of fact, any feasible dictionary describes a feasible solution. However, all feasible solutions cannot be described by a feasible dictionary. For example, no dictionary describe the feasible solution $x_1 = 1$, $x_2 = 0$, $x_3 = 1$, $x_4 = 2$, $x_5 = 5$, $x_6 = 3$ of (5.4). The feasible solutions that can be described by dictionaries are referred as *basic solutions*. The Simplex Method explores only basic solutions and ignores all other ones. But this is valid because if an optimal solution exists, then there is an optimal and basic solution. Indeed, if a feasible solution cannot be improved by the Simplex Method, then increasing any of the n right variables to a positive value never increases the objective function. In such case, the objective function must be written as a linear function of these variables in which all the coefficient are non-positive, and thus the objective function is clearly maximum when all the right variables equal zero. For example, it was the case in (5.14).

5.2.3 Finding an initial solution

In the previous examples, the initialization of the Simplex Method was not a problem. As a matter of fact, we carefully chose problems with all b_i non-negative. This way $x_1 = 0$, $x_2 = 0$,

$\cdots, x_n = 0$ was a feasible solution and the dictionary was easily built. These problems are called *problems with a feasible origin*.

What happens when confronted with a problem with an unfeasible origin? Two difficulties arise. First, a feasible solution can be hard to find. Second, even if we find a feasible solution, a feasible dictionary has then to be built. A way to solve these difficulties is to use another problem called *auxiliary problem*:

$$\begin{array}{lrcll} \text{Minimise} & x_0 & & & \\ \text{Subject to:} & \sum_{j=1}^{n} a_{ij}x_j - x_0 & \leq & b_i & (i = 1,2,\cdots,m) \\ & x_j & \geq & 0 & (j = 0,1,\cdots,n). \end{array}$$

A feasible solution of the auxiliary problem is easily available: it is enough to set $x_j = 0 \forall j \in [1 \ldots n]$ and to give to x_0 a big enough value. It is now easy to see that the original problem has a feasible solution if and only if the auxiliary problem has a feasible solution with $x_0 = 0$. In other words, the original problem has a feasible solution if the optimal value of the auxiliary problem is null. Thus, the idea is to first solve the auxiliary problem. Let see the details on an example.

$$\begin{array}{lrcrcrcr} \text{Maximise} & x_1 & - & x_2 & + & x_3 & & \\ \text{Subject to :} & & & & & & & \\ & 2x_1 & - & x_2 & + & 2x_3 & \leq & 4 \\ & 2x_1 & - & 3x_2 & + & x_3 & \leq & -5 \\ & -x_1 & + & x_2 & - & 2x_3 & \leq & -1 \\ & & & & & x_1,x_2,x_3 & \geq & 0 \end{array}$$

$$\begin{array}{lrcrcrcrcr} \text{Maximise} & -x_0 & & & & & & & & \\ \text{Subject to:} & & & & & & & & & \\ & 2x_1 & - & x_2 & + & 2x_3 & - & x_0 & \leq & 4 \\ & 2x_1 & - & 3x_2 & + & x_3 & - & x_0 & \leq & -5 \\ & -x_1 & + & x_2 & - & 2x_3 & - & x_0 & \leq & -1 \\ & & & & & x_1,x_2,x_3,x_0 & \geq & 0 & & \end{array}$$

We introduce the slack variables. We obtain the dictionary:

$$\begin{array}{rcrcrcrcrcr} x_4 & = & 4 & - & 2x_1 & + & x_2 & - & 2x_3 & + & x_0 \\ x_5 & = & -5 & - & 2x_1 & + & 3x_2 & - & x_3 & + & x_0 \\ x_6 & = & -1 & + & x_1 & - & x_2 & + & 2x_3 & + & x_0 \\ w & = & & & & & & & & - & x_0. \end{array} \tag{5.17}$$

Note that this dictionary is not feasible. However it can be transformed into a feasible one by operating a simple pivot , x_0 entering the basis as x_5 exits it:

$$\begin{array}{rcrcrcrcrcr} x_0 & = & 5 & + & 2x_1 & - & 3x_2 & + & x_3 & + & x_5 \\ x_4 & = & 9 & & & - & 2x_2 & - & x_3 & + & x_5 \\ x_6 & = & 4 & +3 & x_1 & - & 4x_2 & + & 3x_3 & + & x_5 \\ \hline w & = & -5 & - & 2x_1 & + & 3x_2 & - & x_3 & - & x_5. \end{array}$$

More generally, the auxiliary problem can be written as

$$\begin{array}{lrcll} \text{Maximise} & -x_0 & & & \\ \text{Subject to:} & \sum_{j=1}^{n} a_{ij}x_j - x_0 & \leq & b_i & (i=1,2,\cdots,m) \\ & x_j & \geq & 0 & (j=0,1,2,\cdots,n) \end{array}$$

and the associated dictionary is

$$\begin{array}{rcrcrcl} x_{n+i} & = & b_i - & \sum_{j=1}^{n} a_{ij}x_j & + & x_0 & (i=1,2,\cdots,m) \\ w & = & & & - & x_0 & \end{array}$$

This dictionary can be made feasible by pivoting x_0 with the variable the "most unfeasible", that is the exiting variable x_{n+k} is the one with $b_k \leq b_i$ for all i. After the pivot, the variable x_0 has value $-b_k$ and each x_{n+i} has value $b_i - b_k$. All these values are non negative. We are now able to solve the auxiliary problem using the simplex method. Let us go back to our example.

After the first iteration with x_2 entering and x_6 exiting, we get:

$$\begin{array}{rcrcrcrcrcr} x_2 & = & 1 & + & 0.75x_1 & + & 0.75x_3 & + & 0.25x_5 & - & 0.25x_6 \\ x_0 & = & 2 & - & 0.25x_1 & - & 1.25x_3 & + & 0.25x_5 & + & 0.75x_6 \\ x_4 & = & 7 & - & 1.5x_1 & - & 2.5x_3 & + & 0.5x_5 & + & 0.5x_6 \\ \hline w & = & -2 & + & 0.25x_1 & + & 1.25x_3 & - & 0.25x_5 & - & 0.75x_6. \end{array}$$

After the second iteration with x_3 entering and x_0 exiting:

$$\begin{array}{rcrcrcrcrcr} x_3 & = & 1.6 & - & 0.2x_1 & + & 0.2x_5 & + & 0.6x_6 & - & 0.8x_0 \\ x_2 & = & 2.2 & + & 0.6x_1 & + & 0.4x_5 & + & 0.2x_6 & - & 0.6x_0 \\ x_4 & = & 3 & - & x_1 & & & - & x_6 & + & 2x_0 \\ \hline w & = & & & & & & & & - & x_0. \end{array} \tag{5.18}$$

The last dictionary (5.18) is optimal. As the optimal value of the auxiliary problem is null, this dictionary provides a feasible solution of the original problem: $x_1 = 0, x_2 = 2.2, x_3 = 1.6$. Moreover, (5.18) can be easily transformed into a feasible dictionary of the original problem. To obtain the first three lines of the desired dictionary, it is enough to copy the first three lines while removing the terms with x_0:

$$\begin{array}{rcrcrcrcr} x_3 & = & 1.6 & - & 0.2x_1 & + & 0.2x_5 & + & 0.6x_6 \\ x_2 & = & 2.2 & + & 0.6x_1 & + & 0.4x_5 & + & 0.2x_6 \\ x_4 & = & 3 & - & x_1 & & & - & x_6 \end{array} \tag{5.19}$$

To obtain the last line, we express the original objective function

$$z = x_1 - x_2 + x_3 \tag{5.20}$$

in function of the variables outside the basis x_1, x_5, x_6. To do so, we replace the variables of (5.20) by (5.19) and we get:

$$\begin{aligned} z = x_1 - (2.2 + 0.6x_1 + 0.4x_5 + 0.2x_6) + (1.6 - 0.2x_1 + 0.2x_5 + 0.6x_6) \\ z = -0.6 + 0.2x_1 - 0.2x_5 + 0.4x_6 \end{aligned} \tag{5.21}$$

The desired dictionary then is:

$$\begin{array}{rcrcrcrcr} x_3 & = & 1.6 & - & 0.2x_1 & + & 0.2x_5 & + & 0.6x_6 \\ x_2 & = & 2.2 & + & 0.6x_1 & + & 0.4x_5 & + & 0.2x_6 \\ x_4 & = & 3 & - & x_1 & & & - & x_6 \\ \hline z & = & -0.6 & + & 0.2x_1 & - & 0.2x_5 & + & 0.4x_6 \end{array}$$

This strategy is known as the *Simplex Method in two phases*. During the first phase, we set and solve the auxiliary problem. If the optimal value is null, we do the second phase consisting in solving the original problem. Otherwise, the original problem is not feasible.

5.3 Duality of linear programming

Any maximization linear programme has a corresponding minimization problem called the *dual problem*. Any feasible solution of the dual problem gives an upper bound on the optimal value of the initial problem, which is called the *primal*. Reciprocally, any feasible solution of the primal provides a lower bound on the optimal value of the dual problem. Actually, if one of both problems admits an optimal solution, then the other problem does as well and the optimal solutions match each other. This section is devoted to this result also known as the *Duality Theorem*. Another interesting application of the dual problem is that, in some problems, the variables of the dual have some useful interpretation.

5.3.1 Motivations: providing upper bounds on the optimal value

A way to quickly estimate the optimal value of a maximization linear programme simply consists in computing a feasible solution whose value is sufficiently large. For instance, let us consider the following problem formulated in Problem 5.4. The solution $(0,0,1,0)$ gives us a lower bound of 5 for the optimal value z^*. Even better, we get $z^* \geq 22$ by considering the solution $(3,0,2,0)$. Of course, doing so, we have no way to know how close to the optimal value the computed lower bound is.

Problem 5.4.

$$\begin{array}{rcrcl} \text{Maximize} & & 4x_1 + x_2 + 5x_3 + 3x_4 & & \\ \text{Subject to:} & & x_1 - x_2 - x_3 + 3x_4 & \leq & 1 \\ & & 5x_1 + x_2 + 3x_3 + 8x_4 & \leq & 55 \\ & & -x_1 + 2x_2 + 3x_3 - 5x_4 & \leq & 3 \\ & & x_1, x_2, x_3, x_4 & \geq & 0 \end{array}$$

The previous approach provides lower bounds on the optimal value. However, this intuitive method is obviously less efficient than the Simplex Method and this approach provides no clue about the optimality (or not) of the obtained solution. To do so, it is interesting to have upper bounds on the optimal value. This is the main topic of this section.

How to get an upper bound for the optimal value in the previous example? A possible approach is to consider the constraints. For instance, multiplying the second constraint by $\frac{5}{3}$, we get that $z^* \leq \frac{275}{3}$. Indeed, for any $x_1, x_2, x_3, x_4 \geq 0$:

$$\begin{aligned} 4x_1 + x_2 + 5x_3 + 3x_4 &\leq \frac{25}{3}x_1 + \frac{5}{3}x_2 + 5x_3 + \frac{40}{3}x_4 = (5x_1 + x_2 + 3x_3 + 8x_4) \times \frac{5}{3} \\ &\leq 55 \times \frac{5}{3} = \frac{275}{3} \end{aligned}$$

In particular, the above inequality is satisfied by any optimal solution. Therefore, $z^* \leq \frac{275}{3}$. Let us try to improve this bound. For instance, we can add the second constraint to the third one. This gives, for any $x_1, x_2, x_3, x_4 \geq 0$:

$$\begin{aligned} 4x_1 + x_2 + 5x_3 + 3x_4 &\leq 4x_1 + 3x_2 + 6x_3 - 3x_4 \\ &\leq (5x_1 + x_2 + 3x_3 + 8x_4) + (-x_1 + 2x_2 + 3x_3 - 5x_4) \\ &\leq 55 + 3 = 58 \end{aligned}$$

Hence, $z^* \leq 58$.

More formally, we try to upper bound the optimal value by a linear combination of the constraints. Precisely, for all i, let us multiply the i^{th} constraint by $y_i \geq 0$ and then sum the resulting constraints. In the previous two examples, we had $(y_1, y_2, y_3) = (0, \frac{5}{3}, 0)$ and $(y_1, y_2, y_3) = (0, 1, 1)$. More generally, we obtain the following inequality:

$$\begin{aligned} & y_1(x_1 - x_2 - x_3 + 3x_4) + y_2(5x_1 + x_2 + 3x_3 + 8x_4) + y_3(-x_1 + 2x_2 + 3x_3 - 5x_4) \\ = \; & (y_1 - 5y_2 - y_3)x_1 + (-y_1 + y_2 + 2y_3)x_2 + (-y_1 + 3y_2 + 3y_3)x_3 + (3y_1 + 8y_2 - 5y_3)x_4 \\ \leq \; & y_1 + 55y_2 + 3y_3 \end{aligned}$$

For this inequality to provide an upper bound of $4x_1 + x_2 + 5x_3 + 3x_4$, we need to ensure that, for all $x_1, x_2, x_3, x_4 \geq 0$,

$$\begin{aligned} & 4x_1 + x_2 + 5x_3 + 3x_4 \\ \leq \; & (y_1 - 5y_2 - y_3)x_1 + (-y_1 + y_2 + 2y_3)x_2 + (-y_1 + 3y_2 + 3y_3)x_3 + (3y_1 + 8y_2 - 5y_3)x_4. \end{aligned}$$

That is, $y_1 - 5y_2 - y_3 \geq 4$, $-y_1 + y_2 + 2y_3 \geq 1$, $-y_1 + 3y_2 + 3y_3 \geq 5$, and $3y_1 + 8y_2 - 5y_3 \geq 3$.

Combining all inequalities, we obtain the following minimization linear programme:

$$\begin{aligned} \text{Minimize} \quad & y_1 + 55y_2 + 3y_3 \\ \text{Subject to:} \quad & \\ & y_1 - 5y_2 - y_3 \geq 4 \\ & -y_1 + y_2 + 2y_3 \geq 1 \\ & -y_1 + 3y_2 + 3y_3 \geq 5 \\ & 3y_1 + 8y_2 - 5y_3 \geq 3 \\ & y_1, y_2, y_3 \geq 0 \end{aligned}$$

This problem is called the *dual* of the initial maximization problem.

5.3.2 Dual problem

We generalize the example given in Subsection 5.3.1. Consider the following general maximization linear programme:

Problem 5.5.

$$\begin{array}{rll} \text{Maximize} & \sum_{j=1}^{n} c_j x_j & \\ \text{Subject to:} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i & \text{for all } 1 \leq i \leq m \\ & x_j \geq 0 & \text{for all } 1 \leq j \leq n \end{array}$$

Problem 5.5 is called the *primal*. The matricial formulation of this problem is

$$\begin{array}{rl} \text{Maximize} & \mathbf{c}^T \mathbf{x} \\ \text{Subject to:} & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & x \geq \mathbf{0} \end{array}$$

where $\mathbf{x}^T = [x_1, \ldots, x_n]$ and $\mathbf{c}^T = [c_1, \ldots, c_n]$ are vectors in $\mathbb{R}^n$, and $\mathbf{b}^T = [b_1, \ldots, b_m] \in \mathbb{R}^m$, and $\mathbf{A} = [a_{ij}]$ is a matrix in $\mathbb{R}^{m \times n}$.

To find an upper bound on $\mathbf{c}^T\mathbf{x}$, we aim at finding a vector $\mathbf{y}^T = [y_1, \ldots, y_m] \geq 0$ such that, for all feasible solutions $\mathbf{x} \geq 0$ of the initial problem, $\mathbf{c}^T\mathbf{x} \leq \mathbf{y}^T\mathbf{A}\mathbf{x} \leq \mathbf{y}^T\mathbf{b} = \mathbf{b}^T\mathbf{y}$, that is:

$$\begin{array}{rl} \text{Minimize} & \mathbf{b}^T \mathbf{y} \\ \text{Subject to:} & \mathbf{A}^T\mathbf{y} \geq \mathbf{c} \\ & \mathbf{y} \geq \mathbf{0} \end{array}$$

In other words, the *dual* of Problem 5.5 is defined by:

Problem 5.6.

$$\begin{array}{rll} \text{Minimize} & \sum_{i=1}^{m} b_i y_i & \\ \text{Subject to:} & \sum_{i=1}^{m} a_{ij} y_i \geq c_j & \text{for all } 1 \leq j \leq n \\ & y_i \geq 0 & \text{for all } 1 \leq i \leq m \end{array}$$

Notice that the dual of a maximization problem is a minimization problem. Moreover, there is a one-to-one correspondence between the m constraints of the primal $\sum_{j=1\ldots n} a_{ij} x_j \leq b_i$ and the m variables y_i of the dual. Similarly, the n constraints $\sum_{i=1}^{m} a_{ij} y_i \geq c_j$ of the dual correspond one-to-one to the n variables x_i of the primal.

Problem 5.6, which is the dual of Problem 5.5, can be equivalently formulated under the standard form as follows.

$$\begin{array}{rll} \text{Maximize} & \sum_{i=1}^{m} (-b_i) y_i & \\ \text{Subject to:} & \sum_{i=}^{m} (-a_{ij}) y_i \leq -c_j & \text{for all } 1 \leq j \leq n \\ & y_i \geq 0 & \text{for all } 1 \leq i \leq m \end{array} \tag{5.22}$$

Then, the dual of Problem 5.22 has the following formulation which is equivalent to Problem 5.5.

$$\begin{array}{rll} \text{Minimize} & \sum_{j=1}^{n} (-c_j) x_j & \\ \text{Subject to:} & \sum_{j=1}^{n} (-a_{ij}) x_j \geq -b_i & \text{for all } 1 \leq i \leq m \\ & x_j \geq 0 & \text{for all } 1 \leq j \leq n \end{array} \tag{5.23}$$

We deduce the following lemma.

Lemma 5.7. *If D is the dual of a problem P, then the dual of D is P. Informally, the dual of the dual is the primal.*

5.3.3 Duality Theorem

An important aspect of duality is that feasible solutions of the primal and the dual are related.

Lemma 5.8. *Any feasible solution of Problem 5.6 yields an upper bound for Problem 5.5. In other words, the value given by any feasible solution of the dual of a problem is an upper bound for the primal problem.*

Proof. Let $(y_1,\dots,y_m)$ be a feasible solution of the dual and $(x_1,\dots,x_n)$ be a feasible solution of the primal. Then,

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

□

Corollary 5.9. *If $(y_1,\dots,y_m)$ is a feasible solution of the dual of a problem (Problem 5.6) and $(x_1,\dots,x_n)$ is a feasible solution of the corresponding primal (Problem 5.5) such that $\sum_{j=1}^{n} c_j x_j = \sum_{i=1}^{m} b_i y_i$, then both solutions are optimal.*

Corollary 5.9 states that if we find two solutions for the dual and the primal achieving the same value, then this is a certificate of the optimality of these solutions. In particular, in that case (if they are feasible), both the primal and the dual problems have same optimal value.

For instance, we can easily verify that $(0,14,0,5)$ is a feasible solution for Problem 5.4 with value 29. On the other hand, $(11,0,6)$ is a feasible solution for the dual with same value. Hence, the optimal solutions for the primal and for the dual coincide and are equal to 29.

In general, it is not immediate that any linear programme may have such certificate of optimality. In other words, for any feasible linear programme, can we find a solution of the primal problem and a solution of the dual problem that achieve the same value (thus, this value would be optimal)? One of the most important result of the linear programming is the duality theorem that states that it is actually always the case: for any feasible linear programme, the primal and the dual problems have the same optimal solution. This theorem has been proved by D. Gale, H.W. Kuhn and A. W. Tucker [5] and comes from discussions between G.B. Dantzig and J. von Neumann during Fall 1947.

Theorem 5.10 (DUALITY THEOREM)**.** *If the primal problem defined by Problem 5.5 admits an optimal solution $(x_1^*,\dots,x_n^*)$, then the dual problem (Problem 5.6) admits an optimal solution $(y_1^*,\dots,y_m^*)$, and*

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

Proof. The proof consists in showing how a feasible solution $(y_1^*, \ldots, y_m^*)$ of the dual can be obtained thanks to the Simplex Method, so that $z^* = \sum_{i=1}^m b_i y_i^*$ is the optimal value of the primal. The result then follows from Lemma 5.8.

Let us assume that the primal problem has been solved by the Simplex Method. For this purpose, the slack variables have been defined by

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \quad \text{for } 1 \leq i \leq m.$$

Moreover, the last line of the last dictionary computed during the Simplex Method gives the optimal value z^* of the primal in the following way: for any feasible solution $(x_1, \ldots, x_n)$ of the primal we have

$$z = \sum_{j=1}^{n} c_j x_j = z^* + \sum_{i=1}^{n+m} \bar{c}_i x_i.$$

Recall that, for all $i \leq n+m$, $\bar{c}_i$ is non-positive, and that it is null if x_i is one of the basis variables. We set

$$y_i^* = -\bar{c}_{n+i} \qquad \text{for } 1 \leq i \leq m.$$

Then, by definition of the y_i^*'s and the x_{n+i}'s for $1 \leq i \leq m$, we have

$$\begin{aligned} z = \sum_{j=1}^{n} c_j x_j &= z^* + \sum_{i=1}^{n} \bar{c}_i x_i - \sum_{i=1}^{m} y_i^* \left(b_i - \sum_{j=1}^{n} a_{ij} x_j \right) \\ &= \left(z^* - \sum_{i=1}^{m} y_i^* b_i \right) + \sum_{j=1}^{n} \left(\bar{c}_j + \sum_{i=1}^{m} a_{ij} y_i^* \right) x_j. \end{aligned}$$

Since this equation must be true whatever be the affectation of the x_i's and since the $\bar{c}_i$'s are non-positive, this leads to

$$\begin{aligned} z^* &= \sum_{i=1}^{m} y_i^* b_i \quad \text{and} \\ c_j &= \bar{c}_j + \sum_{i=1}^{m} a_{ij} y_i^* \leq \sum_{i=1}^{m} a_{ij} y_i^* \quad \text{for all } 1 \leq j \leq n. \end{aligned}$$

Hence, $(y_1^*, \ldots, y_m^*)$ defined as above is a feasible solution achieving the optimal value of the primal. By Lemma 5.8, this is an optimal solution of the dual. □

5.3.4 Relation between primal and dual

By the Duality Theorem and Lemma 5.7, a linear programme admits a solution if and only if its dual admits a solution. Moreover, according to Lemma 5.8, if a linear programme is unbounded,

then its dual is not feasible. Reciprocally, if a linear programme admits no feasible solution, then its dual is unbounded. Finally, it is possible that both a linear programme and its dual have no feasible solution as shown by the following example.

$$\begin{array}{lrcl} \text{Maximize} & 2x_1 - x_2 & & \\ \text{Subject to:} & x_1 - x_2 & \leq & 1 \\ & -x_1 + x_2 & \leq & -2 \\ & x_1, x_2 & \geq & 0 \end{array}$$

Besides the fact it provides a certificate of optimality, the Duality Theorem has also a practical interest in the application of the Simplex Method. Indeed, the time-complexity of the Simplex Method mainly yields in the number of constraints of the considered linear programme. Hence, when dealing with a linear programme with few variables and many constraints, it will be more efficient to apply the Simplex Method on its dual.

Another interesting application of the Duality Theorem is that it is possible to compute an optimal solution for the dual problem from an optimal solution of the primal. Doing so gives an easy way to test the optimality of a solution. Indeed, if you have a feasible solution of some linear programme, then a solution of the dual problem can be derived (as explained below). Then the initial solution is optimal if and only if the solution obtained for the dual is feasible and leads to the same value.

More formally, the following theorems can be proved

Theorem 5.11 (Complementary Slackness)**.** *Let $(x_1, \ldots, x_n)$ be a feasible solution of Problem 5.5 and $(y_1, \ldots, y_m)$ be a feasible solution of Problem 5.6. These are optimal solutions if and only if*

$$\sum_{i=1}^{m} a_{ij} y_i = c_j, \quad \text{or } x_j = 0, \quad \text{or both} \quad \text{for all } 1 \leq j \leq n, \text{ and}$$
$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad \text{or } y_i = 0, \quad \text{or both} \quad \text{for all } 1 \leq i \leq m.$$

Proof. First, we note that since x and y are feasible $(b_i - \sum_{j=1}^{n} a_{ij} x_j) y_i \geq 0$ and $(\sum_{i=1}^{m} a_{ij} y_i - c_j) x_j \geq 0$. Summing these inequalities over i and j, we obtain

$$\sum_{i=1}^{m} \left(b_i - \sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq 0 \tag{5.24}$$
$$\sum_{j=1}^{n} \left(\sum_{i=1}^{n} a_{ij} y_i - c_j \right) x_j \geq 0 \tag{5.25}$$

Adding Inequalities 5.24 and 5.25 and using the Duality Theorem (Theorem 5.10), we obtain

$$\sum_{i=1}^{m} b_i y_i - \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} x_j y_i + \sum_{j=1}^{n} \sum_{i=1}^{m} a_{ij} y_i x_j - \sum_{j=1}^{n} c_j x_j = \sum_{i=1}^{m} b_i y_i - \sum_{j=1}^{n} c_j x_j = 0.$$

Therefore, Inequalities 5.24 and 5.25 must be equalities. As the variables are positive, we further get that

$$\text{for all } i, \quad \left(b_i - \sum_{j=1}^{n} a_{ij}x_j\right) y_i = 0$$
$$\text{and for all } j, \quad \left(\sum_{i=1}^{m} a_{ij}y_i - c_j\right) x_j = 0.$$

A product is equal to zero if one of its two members is null and we obtain the desired result. □

Theorem 5.12. *A feasible solution $(x_1,\ldots,x_n)$ of Problem 5.5 is optimal if and only if there is a feasible solution $(y_1,\ldots,y_m)$ of Problem 5.6 such that:*

$$\begin{array}{rr} \sum_{i=}^{m} a_{ij}y_i = c_j \quad if & x_j > 0 \\ y_i = 0 \quad if & \sum_{j=1}^{m} a_{ij}x_j < b_i \end{array} \tag{5.26}$$

Note that, if Problem 5.5 admits a non-degenerate solution $(x_1,\ldots,x_n)$, i.e., $x_i > 0$ for any $i \leq n$, then the system of equations in Theorem 5.12 admits a unique solution.

Optimality certificates - Examples. Let see how to apply this theorem on two examples.

Let us first examine the statement that

$$x_1^* = 2,\ x_2^* = 4,\ x_3^* = 0,\ x_4^* = 0,\ x_5^* = 7,\ x_6^* = 0$$

is an optimal solution of the problem

$$\begin{array}{lrcrcrcrcrcrcl} \text{Maximize} & 18x_1 & - & 7x_2 & + & 12x_3 & + & 5x_4 & & & + & 8x_6 & & \\ \text{Subject to:} & 2x_1 & - & 6x_2 & + & 2x_3 & + & 7x_4 & + & 3x_5 & + & 8x_6 & \leq & 1 \\ & -3x_1 & - & x_2 & + & 4x_3 & - & 3x_4 & + & x_5 & + & 2x_6 & \leq & -2 \\ & 8x_1 & - & 3x_2 & + & 5x_3 & - & 2x_4 & & & + & 2x_6 & \leq & 4 \\ & 4x_1 & & & + & 8x_3 & + & 7x_4 & - & x_5 & + & 3x_6 & \leq & 1 \\ & 5x_1 & + & 2x_2 & - & 3x_3 & + & 6x_4 & - & 2x_5 & - & x_6 & \leq & 5 \\ & & & & & & & & & x_1,x_2,\cdots,x_6 & & & \geq & 0 \end{array}$$

In this case, (5.26) says:

$$\begin{array}{rcrcrcrcrcr} 2y_1^* & - & 3y_2^* & + & 8y_3^* & + & 4y_4^* & + & 5y_5^* & = & 18 \\ -6y_1^* & - & y_2^* & - & 3y_3^* & & & + & 2y_5^* & = & -7 \\ 3y_1^* & + & y_2^* & & & - & y_4^* & - & 2y_5^* & = & 0 \\ & & y_2^* & & & & & & & = & 0 \\ & & & & & & & & y_5^* & = & 0 \end{array}$$

As the solution $(\frac{1}{3},0,\frac{5}{3},1,0)$ is a feasible solution of the dual problem (Problem 5.6), the proposed solution is optimal.

Secondly, is

$$x_1^* = 0,\ x_2^* = 2,\ x_3^* = 0,\ x_4^* = 7,\ x_5^* = 0$$

an optimal solution of the following problem?

$$\begin{array}{lrcrcrcrcrcl}
\text{Maximize} & 8x_1 & - & 9x_2 & + & 12x_3 & + & 4x_4 & + & 11x_5 & & \\
\text{Subject to:} & 2x_1 & - & 3x_2 & + & 4x_3 & + & x_4 & + & 3x_5 & \leq & 1 \\
& x_1 & + & 7x_2 & + & 3x_3 & - & 2x_4 & + & x_5 & \leq & 1 \\
& 5x_1 & + & 4x_2 & - & 6x_3 & + & 2x_4 & + & 3x_5 & \leq & 22 \\
& & & & & & & x_1, x_2, \cdots, x_5 & & & \geq & 0
\end{array}$$

Here (5.26) translates into:

$$\begin{array}{rcrcrcl}
-3y_1^* & + & 7y_2^* & + & 4y_3^* & = & -9 \\
y_1^* & - & 2y_2^* & + & 2y_3^* & = & 4 \\
& & y_2^* & & & = & 0
\end{array}$$

As the unique solution of the system $(3.4, 0, 0.3)$ is not a feasible solution of Problem 5.6, the proposed solution is not optimal.

5.3.5 Interpretation of dual variables

As said in the introduction of this section, one of the major interests of the dual programme is that, in some problems, the variables of the dual problem have an interpretation.

A classical example is the *economical interpretation* of the dual variables of the following problem. Consider the problem that consits in maximizing the benefit of a company building some products. Each variable x_j of the primal problem measures the amount of product j that is built, and b_i the amount of resource i (needed to build the products) that is available. Note that, for any $i \leq n, j \leq m$, $a_{i,j}$ represents the number of units of resource i needed per unit of product j. Finally, c_j denotes the benefit (the price) of a unit of product j.

Hence, by checking the units of measure in the constraints $\sum a_{ij} y_i \geq c_j$, the variable y_i must represent a benefit per unit of resource i. Somehow, the variable y_i measures the unitary value of the resource i. This is illustrated by the following theorem the proof of which is omitted.

Theorem 5.13. *If Problem 5.5 admits a non-degenerate optimal solution with value z^*, then there is $\varepsilon > 0$ such that, for any $|t_i| \leq \varepsilon$ $(i = 1, \ldots, m)$, the problem*

$$\begin{array}{lll}
\textit{Maximize} & \sum_{j=}^{n} c_j x_j & \\
\textit{Subject to} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i + t_i & (i = 1, \ldots, m) \\
& x_j \geq 0 & (j = 1, \ldots, n)
\end{array}$$

admits an optimal solution with value $z^* + \sum_{i=1}^{m} y_i^* t_i$, where $(y_1^*, \ldots, y_m^*)$ is the optimal solution of the dual of Problem 5.5.

Theorem 5.13 shows how small variations in the amount of available resources can affect the benefit of the company. For any unit of extra resource i, the benefit increases by y_i^*. Sometimes, y_i^* is called the *marginal cost* of the resource i.

In many networks design problems, a clever interpretation of dual variables may help to achieve more efficient linear programme or to understand the problem better.

5.4 Exercices

5.4.1 General modelling

Exercise 5.1. Which problem(s) among (5.27), (5.28) and (5.29) are under the standard form?

$$\begin{array}{lrcrcl} \text{Maximize} & 3x_1 & - & 5x_2 & & \\ \text{Subject to:} & 4x_1 & + & 5x_2 & \geq & 3 \\ & 6x_1 & - & 6x_2 & = & 7 \\ & x_1 & + & 8x_2 & \leq & 20 \\ & & & x_1, x_2 & \geq & 0 \end{array} \tag{5.27}$$

$$\begin{array}{lrcrcrcrcl} \text{Minimize} & 3x_1 & + & x_2 & + & 4x_3 & + & x_4 & + & 5x_5 & \\ \text{Subject to:} & 9x_1 & + & 2x_2 & + & 6x_3 & + & 5x_4 & + & 3x_5 & \leq 5 \\ & 8x_1 & + & 9x_2 & + & 7x_3 & + & 9x_4 & + & 3x_5 & \leq 2 \\ & & & & & & & x_1, x_2, x_3, x_4 & & & \geq 0 \end{array} \tag{5.28}$$

$$\begin{array}{lrcrcl} \text{Maximize} & 8x_1 & - & 4x_2 & & \\ \text{Subject to:} & 3x_1 & + & x_2 & \leq & 7 \\ & 9x_1 & + & 5x_2 & \leq & -2 \\ & & & x_1, x_2 & \geq & 0 \end{array} \tag{5.29}$$

Exercise 5.2. Put under the standard form:

$$\begin{array}{lrcrcrcrcl} \text{Minimize} & -8x_1 & + & 9x_2 & + & 2x_3 & - & 6x_4 & - & 5x_5 & \\ \text{Subject to:} & 6x_1 & + & 6x_2 & - & 10x_3 & + & 2x_4 & - & 8x_5 & \geq 3 \\ & & & & & & & x_1, x_2, x_3, x_4, x_5 & & & \geq 0 \end{array}$$

Exercise 5.3. Shiow that the linear programme (5.2) has no feasible solutions and that the linear programme (5.3) is unbounded.

Exercise 5.4. Find necessary and sufficient conditions on the numbers s and t for the problem

$$\begin{array}{lrcrcl} \text{Maximize} & x_1 & + & x_2 & & \\ \text{Subject to:} & sx_1 & + & tx_2 & \leq & 1 \\ & & & x_1, x_2 & \geq & 0 \end{array}$$

a) to admit an optimal solution;

b) to be unfeasible;

c) to be unbounded.

Exercise 5.5. Prove or disprove: if the linear programme (5.1) is unbounded, then there exists an index k such that the problem:

$$\begin{array}{lrcll} \text{Maximize} & x_k & & & \\ \text{Subject to:} & \sum_{j=1}^{n} a_{ij} x_j & \leq & b_i & \text{for } 1 \leq i \leq m \\ & x_j & \geq & 0 & \text{for } 1 \leq j \leq n \end{array}$$

is unbounded.

Exercise 5.6. The factory RadioIn builds to types of radios A and B. Every radio is produced by the work of three specialists Pierre, Paul and Jacques. Pierre works at most 24 hours per week. Paul works at most 45 hours per week. Jacques works at most 30 hours per week. The resources necessary to build each type of radio and their selling prices as well are given in the following table:

	Radio A	Radio B
Pierre	1h	2h
Paul	2h	1h
Jacques	1h	3h
Selling prices	15 euros	10 euros

We assume that the company has no problem to sell its production, whichever it is.

a) Model the problem of finding a weekly production plan maximizing the revenue of RadioIn as a linear programme. Write precisely what are the decision variables, the objective function and the constraints.

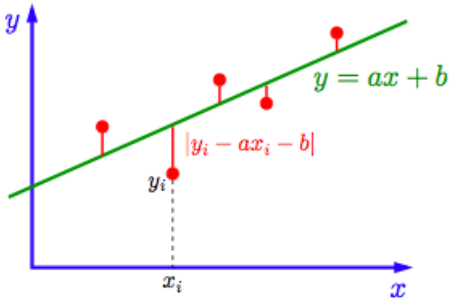
b) Solve the linear programme using the geometric method and give the optimal production plan.

Exercise 5.7. The following table shows the different possible schedule times for the drivers of a bus company. The company wants that at least one driver is present at every hour of the working day (from 9 to 17). The problem is to determine the schedule satisfying this condition with minimum cost.

Time	9 – 11h	9 – 13h	11 – 16h	12 – 15h	13 – 16h	14– 17h	16 – 17h
Cost	18	30	38	14	22	16	9

Formulate an integer linear programme that solves the company decision problem.

Exercise 5.8 (Chebyshev's approximation)**.** Data : m measures of points $(x_i, y_i) \in \mathbb{R}^2$, $i = 1, \ldots, m$.
Objective: Determine a linear approximation $y = ax + b$ minimizing the largest error of approximation. The decision variables of this problem are $a \in \mathbb{R}$ and $b \in \mathbb{R}$. The problem may be



formulated as:

$$\min z = \max_{i=1,\ldots,m} \{|y_i - ax_i - b|\}.$$

It is unfortunately not under the form of a linear programme. Let us try to do some transformations.

Questions:

1. We call MIN-MAX the problem of minimizing the maximum of a set of numbers:

$$\min z \text{ with } z = \max\{c_1 x, ..., c_k x\}.$$

   How to write a MIN-MAX as a linear programme?

2. Can we express the following constraints

$$|x| \leq b$$

   or

$$|x| \geq b$$

   in a linear problem (that is without absolute values)? If yes, how?

3. Rewrite the problem of finding a Chebyshev's linear approximation as a linear programme.

5.4.2 Simplex

Exercise 5.9. Solve with the Simplex Method the following problems:

a.

$$\begin{array}{lrcrcrcl} \text{Maximize} & 3x_1 & + & 3x_2 & + & 4x_3 & & \\ \text{Subject to:} & & & & & & & \\ & x_1 & + & x_2 & + & 2x_3 & \leq & 4 \\ & 2x_1 & & & + & 3x_3 & \leq & 5 \\ & 2x_1 & + & x_2 & + & 3x_3 & \leq & 7 \\ & & & & & x_1, x_2, x_3 & \geq & 0 \end{array}$$

b.

$$\begin{array}{lrcrcrcrcl} \text{Maximize} & 5x_1 & + & 6x_2 & + & 9x_3 & + & 8x_4 & & \\ \text{Subject to:} & & & & & & & & & \\ & x_1 & + & 2x_2 & + & 3x_3 & + & x_4 & \leq & 5 \\ & x_1 & + & x_2 & + & 2x_3 & + & 3x_4 & \leq & 3 \\ & & & & & & & x_1, x_2, x_3, x_4 & \geq & 0 \end{array}$$

c.

$$\begin{array}{lrcrcrcl} \text{Maximize} & 2x_1 & + & x_2 & & & \\ \text{Subject to:} & & & & & & \\ & 2x_1 & + & 3x_2 & \leq & 3 \\ & x_1 & + & 5x_2 & \leq & 1 \\ & 2x_1 & + & x_2 & \leq & 4 \\ & 4x_1 & + & x_2 & \leq & 5 \\ & & & x_1, x_2 & \geq & 0 \end{array}$$

Exercise 5.10. Use the Simplex Method to describe *all* the optimal solutions of the following linear programme:

$$\begin{array}{lrcrcrcrcl}
\text{Maximize} & 2x_1 & + & 3x_2 & + & 5x_3 & + & 4x_4 & & \\
\text{Subject to:} & & & & & & & & & \\
& x_1 & + & 2x_2 & + & 3x_3 & + & x_4 & \leq & 5 \\
& x_1 & + & x_2 & + & 2x_3 & + & 3x_4 & \leq & 3 \\
& & & & & & & x_1,x_2,x_3,x_4 & \geq & 0
\end{array}$$

Exercise 5.11. Solve the following problems using the Simplex Method in two phases.

a.

$$\begin{array}{lrcrcrcl}
\text{Maximise} & 3x_1 & + & x_2 & & & & \\
\text{Subject to:} & & & & & & & \\
& x_1 & - & x_2 & \leq & -1 & & \\
& -x_1 & - & x_2 & \leq & -3 & & \\
& 2x_1 & + & x_2 & \leq & 4 & & \\
& & & & & x_1,x_2 & \geq & 0
\end{array}$$

b.

$$\begin{array}{lrcrcrcl}
\text{Maximise} & 3x_1 & + & x_2 & & & & \\
\text{Subject to:} & & & & & & & \\
& x_1 & - & x_2 & \leq & -1 & & \\
& -x_1 & - & x_2 & \leq & -3 & & \\
& 2x_1 & + & x_2 & \leq & 2 & & \\
& & & & & x_1,x_2 & \geq & 0
\end{array}$$

c.

$$\begin{array}{lrcrcrcl}
\text{Maximise} & 3x_1 & + & x_2 & & & & \\
\text{Subject to:} & & & & & & & \\
& x_1 & - & x_2 & \leq & -1 & & \\
& -x_1 & - & x_2 & \leq & -3 & & \\
& 2x_1 & - & x_2 & \leq & 2 & & \\
& & & & & x_1,x_2 & \geq & 0
\end{array}$$

5.4.3 Duality

Exercise 5.12. Write the dual of the following linear programme.

$$\begin{array}{lrcrcrcl}
\text{Maximize} & 7x_1 & + & x_2 & & & & \\
\text{Subject to:} & & & & & & & \\
& 4x_1 & + & 3x_2 & \leq & 3 & & \\
& x_1 & - & 2x_2 & \leq & 4 & & \\
& -5x_1 & - & 2x_2 & \leq & 3 & & \\
& & & x_1,x_2 & \geq & 0 & &
\end{array}$$

Exercise 5.13. Consider the following linear programme.

$$\begin{array}{lrcrcrcrcr} \text{Minimize} & -2x_1 & - & 3x_2 & - & 2x_3 & - & 3x_4 & & \\ \text{Subject to:} & & & & & & & & & \\ & -2x_1 & - & x_2 & - & 3x_3 & - & 2x_4 & \geq & -8 \\ & 3x_1 & + & 2x_2 & + & 2x_3 & + & x_4 & \leq & 7 \\ & & & & & & & x_1,x_2,x_3,x_4 & \geq & 0 \end{array} \tag{5.30}$$

a) Write the programme (5.30) under the standard form.

b) Write the dual (D) of programme (5.30).

c) Give a graphical solution of the dual programme (D).

d) Carry on the first iteration of the Simplex Method on the linear programme (5.30).

After three iterations, one find that the optimal solution of this programme is $x_1 = 0$, $x_2 = 2$, $x_3 = 0$ and $x_4 = 3$.

e) Verify that the solution of (D) obtained at Question c) is optimal.

Exercise 5.14. Prove that the following linear programme is unbounded.

$$\begin{array}{lrcrcrcr} \text{Maximize} & 3x_1 & - & 4x_2 & + & 3x_3 & & \\ \text{Subject to :} & & & & & & & \\ & -x_1 & + & x_2 & + & x_3 & \leq & -3 \\ & -2x_1 & - & 3x_2 & + & 4x_3 & \leq & -5 \\ & -3x_1 & + & 2x_2 & - & x_3 & \leq & -3 \\ & & & & & x_1,x_2,x_3 & \geq & 0 \end{array}$$

Exercise 5.15. We consider the following linear programme.

$$\begin{array}{lrcrcrcr} \text{Maximize} & x_1 & - & 3x_2 & + & 3x_3 & & \\ \text{Subject to :} & & & & & & & \\ & 2x_1 & - & x_2 & + & x_3 & \leq & 4 \\ & -4x_1 & + & 3x_2 & & & \leq & 2 \\ & 3x_1 & - & 2x_2 & - & x_3 & \leq & 5 \\ & & & & & x_1,x_2,x_3 & \geq & 0 \end{array}$$

Is the solution $x_1^* = 0$, $x_2^* = 0$, $x_3^* = 4$ optimal?

Exercise 5.16. We consider the following linear programme.

$$\begin{array}{lrcrcrcrcrcr} \text{Maximize} & 7x_1 & + & 6x_2 & + & 5x_3 & - & 2x_4 & + & 3x_5 & \\ \text{Subject to:} & & & & & & & & & & \\ & x_1 & + & 3x_2 & + & 5x_3 & - & 2x_4 & + & 2x_5 \leq & 4 \\ & 4x_1 & + & 2x_2 & - & 2x_3 & + & x_4 & + & x_5 \leq & 3 \\ & 2x_1 & + & 4x_2 & + & 4x_3 & - & 2x_4 & + & 5x_5 \leq & 5 \\ & 3x_1 & + & x_2 & + & 2x_3 & - & x_4 & - & 2x_5 \leq & 1 \\ & & & & & & & x_1,x_2,x_3,x_4,x_5 & & \geq & 0. \end{array}$$

Is the solution $x_1^* = 0, x_2^* = \frac{4}{3}, x_3^* = \frac{2}{3}, x_4^* = \frac{5}{3}, x_5^* = 0$, optimal?

Exercise 5.17. 1. Because of the arrival of new models, a salesman wants to sell off quickly its stock composed of eight phones, four hands-free kits and nineteen prepaid cards. Thanks to a market study, he knows that he can propose an offer with a phone and two prepaid cards and that this offer will bring in a profit of seven euros. Similarly, we can prepare a box with a phone, a hands-free kit and three prepaid cards, yielding a profit of nine euros. He is assured to be able to sell any quantity of these two offers within the availability of its stock. What quantity of each offer should the salesman prepare to maximize its net profit?

2. A sales representative of a supermarket chain proposes to buy its stock (the products, not the offers). What unit prices should he negociate for each product (phone, hands-free kits, and prepaid cards)?

Exercise 5.18 (FARKAS' LEMMA)**.** The following two linear programmes are duals of each other.

$$\begin{array}{rlll} \text{Maximize} & \mathbf{0^T x} \text{ subject to} & \mathbf{Ax = 0} & \text{and } \mathbf{x \geq b} \\ \text{Minimize} & -\mathbf{b^T z} \text{ subject to} & \mathbf{A^T y - z = 0} & \text{and } \mathbf{z \geq 0} \end{array}$$

Farkas' Lemma says that exactly one of the two linear systems:

$$\mathbf{Ax = 0},\ \mathbf{x \geq b} \quad \text{and} \quad \mathbf{yA \geq 0},\ \mathbf{yAb} > 0$$

has a solution. Deduce Farkas' Lemma from the Duality Theorem (5.10).

Exercise 5.19. The following two linear programmes are duals of each other.

$$\begin{array}{rlll} \text{Minimize} & \mathbf{0}^T\mathbf{y} \text{ subject to} & \mathbf{A}^T\mathbf{y} \geq \mathbf{c} & \\ \text{Maximize} & \mathbf{c^T x} \text{ subject to} & \mathbf{Ax = 0} & \text{and } \mathbf{x \geq 0} \end{array}$$

A variant of Farkas' Lemma says that exactly one of the two linear systems:

$$\mathbf{A^T y \geq c} \quad \text{and} \quad \mathbf{Ax = 0},\ \mathbf{x \geq 0},\ \mathbf{cx} > 0$$

has a solution. Deduce this variant of Farkas' Lemma from the Duality Theorem (5.10).

Exercise 5.20 (Application of duality to game theory- Minimax principle)**.** In this problem, based on a lecture of Shuchi Chawla, we present an application of linear programming duality in the theory of games. In particular, we will prove the Minimax Theorem using duality.

Let us first give some definition. A *two-players zero-sum game* is a protocol defined as follows: two players choose strategies in turn; given two strategies x and y, we have a *valuation function* $f(x,y)$ which tells us what the payoff for Player one is. Since it is a zero sum game, the payoff for the Player two is exactly $-f(x,y)$. We can view such a game as a matrix of payoffs for one of the players. As an example take the game of Rock-Paper-Scissors, where the payoff

is one for the winning party or 0 if there is a tie. The matrix of winnings for player one will then be the following:

$$A = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix}$$

Where A_{ij} corresponds to the payoff for player one if player one picks the i-th element and player two the j-th element of the sequence (Rock, Paper, Scissors). We will henceforth refer to player number two as the column player and player number one as the row player. If the row player goes first, he obviously wants to minimize the possible gain of the column player.

What is the payoff of the row player? If the row player plays first, he knows that the column player will choose the minimum of the line he will choose. So he has to choose the line with the maximal minimum value. That is its payoff is

$$\max_i \min_j A_{ij}.$$

Similarly, what is the payoff of the column player if he plays first? If the column player plays first, the column player knows that the row player will choose the maximum of the column that will be chosen. So the column player has to choose the column with minimal maximum value. Hence, the payoff of the row player in this case is

$$\min_j \max_i A_{ij}.$$

Compare the payoffs. It is clear that

$$\max_i \min_j A_{ij} \leq \min_j \max_i A_{ij}.$$

The minimax theorem states that if we allow the players to choose probability distributions instead of a given column or row, then the payoff is the same no matter which player starts. More formally:

Theorem 5.14 (Minimax theorem)**.** *If x and y are probability vectors, then*

$$\max_{\mathbf{y}}(\min_{\mathbf{x}} \mathbf{y}^T \mathbf{A}\mathbf{x}) = \min_{\mathbf{x}}(\max_{\mathbf{y}}(\mathbf{y}^T \mathbf{A}\mathbf{x})).$$

Let us prove the theorem.

1. Formulate the problem of maximizing its payoff as a linear programme.
2. Formulate the second problem of minimizing its loss as a linear programme.
3. Prove that the second problem is a dual of the first problem.
4. Conclude.

Exercise 5.21. Prove the following proposition.

Proposition 5.15. *The dual problem of the problem*

$$\textit{Maximize } \mathbf{c}^T\mathbf{x} \textit{ subject to } \mathbf{Ax} \leq \mathbf{a} \textit{ and } \mathbf{Bx} = \mathbf{b} \textit{ and } \mathbf{x} \geq \mathbf{0}$$

is the problem

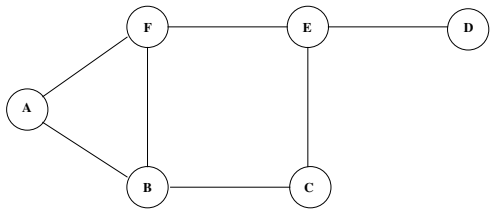
$$\textit{Minimize } \mathbf{a}^T\mathbf{y} + \mathbf{b}^T\mathbf{z} \textit{ subject to } \mathbf{A}^T\mathbf{y} + \mathbf{B}^T\mathbf{z} \geq \mathbf{c} \textit{ and } \mathbf{y} \geq \mathbf{0}.$$

5.4.4 Modelling Combinatorial Problems via (integer) linear programming

Lots of combinatorial problems may be formulated as linear programmes.

Exercise 5.22 (VERTEX COVER)**.** A *vertex cover* in a graph $G = (V,E)$ is a set K of vertices such that each edge e of E is incident to at least one vertex of K. The VERTEX COVER problem is to find a vertex cover of minimum cardinality in a given graph.

1. Express VERTEX COVER for the *following graph* as an integer linear programme:

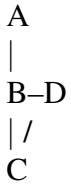


2. Express VERTEX COVER for a *general graph* as a linear programme.

Exercise 5.23 (EDGE COVER)**.** An *edge cover* of a graph $G = (V,E)$ is a set of edges $F \subseteq E$ such that every vertex $v \in V$ is incident to at least one edge of F. The EDGE COVER problem is to find an edge cover of minimum cardinality in a given graph.

Adapt the integer linear programme modelling VERTEX COVER to obtain an integer linear programming formulation of EDGE COVER.

Exercise 5.24. Consider the graph



What does the following linear programme do?

$$\begin{array}{ll} \text{Minimize} & x_A + x_B + x_C + x_D \\ \text{Subject to:} & \\ & x_A + x_B \geq 1 \\ & x_B + x_D \geq 1 \\ & x_B + x_C \geq 1 \\ & x_C + x_D \geq 1 \\ & x_A \geq 0, x_B \geq 0, x_C \geq 0, x_D \geq 0 \end{array}$$

Exercise 5.25 (Maximum cardinality matching problem (Polynomial $<$ flows or augmenting paths))**.** Let $G = (V,E)$ be a graph. Recall that a *matching* $M \subseteq E$ is a set of edges such that every vertex of V is incident to at most one edge of M. The MAXIMUM MATCHING problem is to find a matching M of maximum size. Express MAXIMUM MATCHING as a integer linear programme.

Exercise 5.26 (MAXIMUM CLIQUE)**.** Recall that a *clique* of a graph $G = (V,E)$ is a subset C of V, such that every two vertices in V are joined by an edge of E. The MAXIMUM CLIQUE problem consist of finding the largest cardinality of a clique.

Express MAXIMUM CLIQUE as an integer linear programme.

Exercise 5.27 (Resource assignment)**.** A university class has to go from Marseille to Paris using buses. There are some strong inimities inside the group and two people that dislike each other cannot share the same bus. What is the minimum number of buses needed to transport the whole group? Write a LP that solve the problem. (We suppose that a bus does not have a limitation on the number of places.)

Exercise 5.28 (French newspaper enigma)**.** What is the maximum size of a set of integers between 1 and 100 such that for any pair (a,b), the difference a-b is not a square ?

1. Model this problem as a graph problem.

2. Write a linear programme to solve it.

Exercise 5.29 (MAXIMUM STABLE SET)**.** Recall that a *stable set* of a graph $G = (V,E)$ is a subset S of paorwise non-adjacent vertices. The MAXIMUM STABLE SET problem consist in finding the largest cardinality of a stable set. Give a linear programming formulation of this problem.

Exercise 5.30 (MINIMUM SET COVER)**.** Let $U = \{1,...,n\}$ be a set and $\mathcal{S} = \{S_1, \ldots, S_m\}$ a set of subsets of $\mathcal{U}$. An *$\mathcal{S}$-cover* of U is a subset of $\mathcal{T}$ of $\mathcal{S}$ such that $\bigcup_{T \in \mathcal{T}} T = \mathcal{U}$. The MINIMUM SET COVER problem consists in, given a set U $\mathcal{S}$, finding an $\mathcal{S}$-cover of U of minimum cardinality.

Formulate MINIMUM SET COVER as an integer linear programme.

(The associate decision problem k-SET COVER, which consists in deciding wether U has an $\mathcal{S}$-cover of cardinality at most k is $\mathcal{NP}$-complete.

Exercise 5.31 (Instance of MAXIMUM SET PACKING)**.** Suppose you are at a convention of foreign ambassadors, each of which speaks English and other various languages.

- French ambassador: French, Russian
- US ambassador:
- Brazilian ambassador: Portuguese, Spanish
- Chinese ambassador: Chinese, Russian
- Senegalese ambassador: Wolof, French, Spanish

You want to make an announcement to a group of them, but because you do not trust them, you do not want them to be able to speak among themselves without you being able to understand them (you only speak English). To ensure this, you will choose a group such that no two ambassadors speak the same language, other than English. On the other hand you also want to give your announcement to as many ambassadors as possible.

Write a linear programme giving the maximum number of ambassadors at which you will be able to give the message.

Exercise 5.32 (MAXIMUM SET PACKING)**.** Given a finite set S and a list $\mathcal{L}$ of subsets of S. The MAXIMUM SET PACKING problem consists in finding the maximum number of pairwise disjoint sets in a given list $\mathcal{L}$. Give a linear programming formulation of this problem.

5.4.5 Modelling flows and shortest paths.

Recall that a *flow network* is a four-tuple $N = (D,s,t,c)$ where

- $D = (V,A)$.
- c is a capacity function from A to $\mathbb{R}^+ \cup \infty$. For an arc $a \in A$, $c(a)$ represents its capacity, that is the maximum amount of flow it can carry.
- s and t are two distinct vertices: s is the source of the flow and t the sink.

A *flow* is a function f from A to $\mathbb{R}^+$ which respects the flow conservation constraints and the capacity constraints. See Chapter 4

Exercise 5.33 (MAXIMUM FLOW)**.** Write a linear programming formulation of the MAXIMUM FLOW problem.

Exercise 5.34 (MULTICOMMODITY FLOW)**.** Consider a flow network $\mathcal{N} = (D,s,t,c)$. Consider a set of demands given by the matrix $\mathcal{D} = (d_{ij} \in \mathbb{R}; i, j \in V, i \neq j)$, where d_{ij} is the amount of flow that has to be sent from node i to node j. The multicommodity flow problem is to determine if all demands can be routed simultaneously on the network. This problem models a telecom network and is one of the fundamental problem of the networking research field.

Write a linear program that solves the multicommodity flow problem.

Exercise 5.35 (Shortest (s,t)-path)**.** Let $D = (V,A,l)$ be a an arc-weighted digraph with l is a length function from A to $\mathbb{R}^+$. For $a \in A$, l(a) is the length of arc a. Let s and t two distinguished vertices.

Write a linear programme that finds the length of a shortest path between s and t.

Exercise 5.36 (Eccentricy and diameter)**.** The *distance* between two vertices in a graph is the number of edges in a shortest path connecting them. The *eccentricity* ε of a vertex v is the greatest distance between v and any other vertex. It can be thought of as how far a node is from the node most distant from it in the graph. The *diameter* of a graph is the maximum eccentricity of any vertex in the graph. That is, it is the greatest distance between any pair of vertices.

1. Write a linear programme to compute the eccentricity of a given vertex.

2. Write a linear programme which computes the diameter of a graph.

Exercise 5.37 (MINIMUM (s,t)-CUT)**.** Recall that in a flow network $N = (G,s,t,c)$ an (s,t)-*cut* is a bipartition $C = (V_s,V_t)$ of the vertices of G such that $s \in V_s$ and $t \in V_t$. The capacity of the cut C, denoted by $\delta(C)$, is the sum of the capacities of the out-arcs of V_s (i.e., the arcs (u,v) with $u \in V_s$ and $v \in V_t$).

Write a linear programme that finds the minimum capacity of an (s,t)-cut.
Hint: Use variables to know in which partition is each vertex and additional variables to know which edges are in the cut.

Bibliography

[1] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. Wiley-Interscience, 1998.

[2] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, New York, 1983.

[3] *ILOG CPLEX optimization software*. `http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/`.

[4] *GNU Linear Programming Kit*. `http://www.gnu.org/software/glpk/`.

[5] D. Gale, H. W. Kuhn, and A. W. Tucker. Linear Programming and the Theory of Games. In *T. C. Koopmans (ed.), Activity Analysis of Production and Allocation*, New York, Wiley, 1951.

[6] M. Sakarovitch. *Optimisation Combinatoire: Graphes et Programmation Linéaire*. Hermann, Paris, 1984.