Supplementary Notes to Networks and Integer Programming

Jesper Larsen Jens Clausen

Kongens Lyngby 2009

Technical University of Denmark
Department of Management Engineering
DK-2800 Kongens Lyngby, Denmark
www.man.dtu.dk

ISSN 0909-3192

Preface

This is a draft version. This volume is a supplement to Wolseys "Integer Programming" for the course *Networks and Integer Programming* (42113) at the Department of Management Engineering at the Technical University of Denmark.

Whereas Wolsey does an excellent job of describing the intricacies and challenges of general integer program and the techniques necessary to master the field, he basically ignores the field on network optimization. We do not claim to master this in total but we address the most important problems in network optimization in relation to Wolseys text.

We have included two supplementary readings in form of a chapter on duality and branch and bound. Finally an appendix contains a small introduction to the interactive part of CPLEX.

Kgs. Lyngby, February 2009

Jesper Larsen Jens Clausen

Contents

Chapter 1

Introduction

1.1 Graphs and Networks

Network models are build from two major building blocks: **edges** (sometimes called arcs) and **vertices** (sometimes called nodes). Edges are lines connecting two vertices. A **graph** is a structure that is composed of vertices and edges. A **directed graph** (sometime denoted digraph) is a graph in which the edges have been assigned an orientation – often shown by arrowheads on the lines. Finally, a **network** is a (directed) graph in which the edges have an associated **flow**. Table 1.1 gives a number of examples of the use of graphs.

Vertices	Edges	Flow
cities	roads	vehicles
switching centers	telephone lines	telephone calls
pipe junctions	pipes	water
rail junctions	railway lines	trains

Table 1.1: Simple examples of networks

A graph $G = (V(G), E(G))$ consists of a finite set of vertices $V(G)$ and a set of edges $E(G)$ – for short denoted $G = (V, E)$. Only where it is necessary we will use the "extended" form to describe a graph. E is a subset of $\{(v, w) : v, w \in V\}$.

The number of vertices resp. edges in G is denoted n resp. m: $|V(G)| = n$ and $|E(G)| = m$.

Figure 1.1 shows a picture of a graph consisting of 9 vertices and 16 edges. Instead of a graphical representation is can be stated as $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $E = \{(1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (2, 7), (2, 8), (3, 4), (3, 6), (4, 5), (4, 8), (5, 7), (5, 9), (6, 7), (6, 8), (8, 9)\}$.

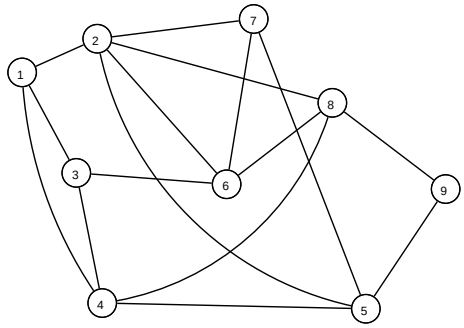


Figure 1.1: Graphical illustrations of graphs

An edge $(i, j) \in E$ is **incident with** the vertices i and j. i and j are called **neighbors** (or **adjacent**). In a **complete** (or **fully connected**) graph all possible pairs of vertices are connected with an edge, i.e. $E = \{(i, j) : i, j \in V\}$. We define $V^+(i)$ and $V^-(i)$ to be the set of edges out of i resp. into i, that is, $V^+(i) = \{(i, j) \in E : i \in V\}$ and $V^-(i) = \{(j, i) \in E : j \in V\}$.

A **subgraph** H of G has $V(H) \subset V(G)$ and $E(H) \subset E(G)$, where $E(H) \subset \{(i, j) : i, j \in V(H)\}$. When $A \subset E$, $G \setminus A$ is given by $V(G \setminus A) = V(G)$ and $E(G \setminus A) = E \setminus A$. When $B \subset V$, $G \setminus B$ (or $G[V \setminus B]$) is given by $V(G \setminus B) = V(G) \setminus B$ and $E(G \setminus B) = E \setminus \{(i, j) : i \in B \vee j \in B\}$. $G \setminus B$ is also called the **subgraph induced by** $V \setminus B$. The subgraph H of G is **spanning** if $V(H) = V(G)$. An example of a subgraph in Figure 1.1 can be $V(H) = \{1, 2, 3, 4\}$ then the edges in the subgraphs are the edges of G where both endpoints are in $V(H)$ i.e. $E(H) = \{(1, 2), (1, 3), (1, 4), (3, 4)\}$.

A **path** P in G is a sequence $v_0, e_1, v_1, e_2, ..., e_k, v_k$, in which each v_i is a vertex, each e_i an edge, and where $e_i = (v_{i-1}, v_i), I = 1, ..., k$. P is called a path from v_0 to v_k or a (v_0, v_k)-path. The path is said to be **closed** if $v_0 = v_k$, **edge-simple**, if all e_i are different, and **simple**, if all v_i are different. A **chain** is a simple path. If we look at our graph in Figure 1.1 then $1, 4, 5, 9, 8$ denotes a path, as all vertices on the path are distinct it is a simple path (or chain). Furthermore the path $1, 3, 6, 2, 7, 6, 2, 1$ is a closed path.

A **circuit** or **cycle** C is a path, which is closed and where $v_0, ..., v_{k-1}$ are all different. A **dicircuit** is a circuit, in which all edges are forward. An example of a circuit in Figure 1.1 is $1, 3, 6, 2, 1$ or $2, 6, 8, 2$.

A graph G is **connected**, if a path from i to j exists for all pairs (i, j) of vertices. If G is connected, $v \in V$, and $G \setminus v$ is not connected, v is called a **cut-vertex**. A circuit-free graph G is called a **forest**; if G is also connected G is a **tree**.

Let us return to the concept of orientations on the edges. Formally a directed graph or digraph, $G = (V(G), E(G))$ consists of a finite set of vertices $V(G)$ and a set of edges $E(G)$ - often denoted $G = (V, E)$. E is a subset of $\{(i, j) : i, j \in V\}$. Each edge has a **start-vertex** (**tail** - $t(i, j) = i$) and an **end-vertex** (**head** - $h(i, j) = j$). The number of vertices resp. edges in G is denoted n resp. m : $|V(G)| = n$ and $|E(G)| = m$.

A edge $(i, j) \in E$ is **incident with** the vertices i and j, and j is called a **neighbor** to i and they are called **adjacent**.

In a **complete** (or **fully connected**) graph all edges are "present", i.e $E = \{(i, j) : i, j \in V\}$. Any digraph has a **underlying graph**, which is found by ignoring the direction of the edges. When graph-terminology is used for digraphs, these relates to the underlying graph.

A **path** P in the digraph G is a sequence $v_0, e_1, v_1, e_2, ..., e_k, v_k$, in which each v_i is a vertex, each e_i an edge, and where e_i is either (v_{i-1}, v_i) or (v_i, v_{i-1}), $I = 1, ..., k$. If e_i is equal to (v_{i-1}, v_i), e_i is called **forward**, otherwise e_i is **backward**. P is called a **directed** path, if all edges are forward.

A graph G can be strongly connected, which means that any vertex is reachable from any other vertex. When talking about directed graphs it means that directions of the edges should be obeyed.

1.2 Algorithms

1.2.1 Concepts

An algorithm is basically a recipe that in a deterministic way describes how to accomplish a given task.

show some pseudo code, explain and give a couple of small examples.

1.2.2 Depth-first search

1.2.3 Breadth-first search

Chapter 2

Teaching Duality in Linear Programming – The Multiplier Approach

Duality in LP is often introduced through the relation between LP problems modelling different aspects of a planning problem. Though providing a good motivation for the study of duality this approach does not support the general understanding of the interplay between the primal and the dual problem with respect to the variables and constraints.

This paper describes the multiplier approach to teaching duality: Replace the primal LP-problem P with a relaxed problem by including in the objective function the violation of each primal constraint multiplied by an associated multiplier. The relaxed problem is trivial to solve, but the solution provides only a bound for the solution of the primal problem. The new problem is hence to choose the multipliers so that this bound is optimized. This is the dual problem of P.

LP duality is described similarly in the work by A. M. Geoffrion on Lagrangean Relaxation for Integer Programming. However, we suggest here that the approach is used not only in the technical parts of a method for integer programming, but as a general tool in teaching LP.

2.1 Introduction

Duality is one of the most fundamental concepts in connection with linear programming and provides the basis for better understanding of LP models and their results and for algorithm construction in linear and in integer programming. Duality in LP is in most textbooks as e.g. [2, 7, 9] introduced using examples building upon the relationship between the primal and the dual problem seen from an economical perspective. The primal problem may e.g. be a blending problem:

Determine the contents of each of a set of available ingredients (e.g. fruit or grain) in a final blend. Each ingredient contains varying amounts of vitamins and minerals, and the final blend has to satisfy certain requirements regarding the total contents of minerals and vitamins. The costs of units of the ingredients are given, and the goal is to minimize the unit cost of the blend.

The dual problem here turns out to be a problem of prices: What is the maximum price one is willing to pay for "artificial" vitamins and minerals to substitute the natural ones in the blend?

After such an introduction the general formulations and results are presented including the statement and proof of the duality theorem:

Theorem 2.1 (Duality Theorem) *If feasible solutions to both the primal and the dual problem in a pair of dual LP problems exist, then there is an optimum solution to both systems and the optimal values are equal.*

Also accompanying theorems on unboundedness and infeasibility, and the Complementary Slackness Theorem are presented in most textbooks.

While giving a good motivation for studying dual problems this approach has an obvious shortcoming when it comes to explaining duality in general, i.e. in situations, where no natural interpretation of the dual problem in terms of primal parameters exists.

General descriptions of duality are often handled by means of symmetrical dual forms as introduced by von Neumann. Duality is introduced by stating that two LP-problems are dual problems *by definition*. The classical duality theorems are then introduced and proved. The dual of a given LP-problem can then be found by transforming this to a problem of one of the two symmetrical types and deriving it's dual through the definition. Though perfectly clear from a formal point of view this approach does not provide any understanding the interplay

between signs of variables in one problem and type of constraints in the dual problem. In [1, Appendix II], a presentation of general duality trying to provide this understanding is given, but the presentation is rather complicated.

In the following another approach used by the author when reviewing duality in courses on combinatorial optimization is suggested. The motivating idea is that of problem relaxation: If a problem is difficult to solve, then find a family of easy problems each resembling the original one in the sense that the solution provides information in terms of bounds on the solution of our original problem. Now find that problem among the easy ones, which provides the strongest bounds.

In the LP case we relax the primal problem into one with only non-negativity constraints by including in the objective function the violation of each primal constraint multiplied by an associated multiplier. For each choice of multipliers respecting sign conditions derived from the primal constraints, the optimal value of the relaxed problem is a lower bound (in case of primal minimization) on the optimal primal value. The dual problem now turns out to be the problem of maximizing this lower bound.

The advantages of this approach are 1) that the dual problem of any given LP problem can be derived in a natural way without problem transformations and definitions 2) that the primal/dual relationship between variables and constraints and the signs/types of these becomes very clear for all pairs of primal/dual problems and 3) that *Lagrangean Relaxation* in integer programming now becomes a natural extension as described in [3, 4]. A similar approach is sketched in [8, 5].

The reader is assumed to have a good knowledge of basic linear programming. Hence, concepts as Simplex tableau, basic variables, reduced costs etc. will be used without introduction. Also standard transformations between different forms of LP problems are assumed to be known.

The paper is organized as follows: Section 2 contains an example of the approach sketched, Section 3 presents the general formulation of duality through multipliers and the proof of the Duality Theorem, and Section 4 discusses the pros and cons of the approach presented. The main contribution of the paper is not theoretical but pedagogical: the derivation of the dual of a given problem can be presented without problem transformations and definitions, which are hard to motivate to people with no prior knowledge of duality.

Fruit type	F1	F2	F3	F4
Preservative R	2	3	0	5
Preservative Q	3	0	2	4
Cost pr. ton	13	6	4	12

Table 2.1: Contents of R and Q and price for different types of fruit

2.2 A blending example

The following example is adapted from [6]. The Pasta Basta company wants to evaluate an ecological production versus a traditional one. One of their products, the Pasta Basta lasagne, has to contain certain preservatives, R and Q, in order to ensure durability. Artificially produced counterparts R' and Q' are usually used in the production – these are bought from a chemical company PresChem, but are undesirable from the ecological point of view. R and Q can alternatively be extracted from fresh fruit, and there are four types of fruit each with their particular content (number of units) of R and Q in one ton of the fruit. These contents and the cost of buying the fruit are specified in Table 2.1.

Pasta Basta has a market corresponding to daily needs of 7 units of R and 2 units of Q. If the complete production is based on ecologically produced preservatives, which types of fruit and which amounts should be bought in order to supply the necessary preservatives in the cheapest way?

The problem is obviously an LP-problem:

$$
(P)\qquad
\begin{array}{lrrrrl}
\min & 13x_1 & +6x_2 & +4x_3 & +12x_4 & \\
\text{s.t.} & 2x_1 & +3x_2 & & +5x_4 & = 7 \\
 & 3x_1 & & +2x_3 & +4x_4 & = 2 \\
 & & x_1, x_2, & x_3, x_4 \geq 0 & &
\end{array}
$$

With x_2 and x_3 as initial basic variables the Simplex method solves the problem with the tableau of Table 2.2 as result.

Natural questions when one knows one solution method for a given type of problem are: Is there an easier way to solve such problems? Which LP problems are trivial to solve? Regarding the latter, it is obvious that LP minimization problems with no constraints but non-negativity of $x_1, \cdots, x_n$ are trivial to solve:

$$\begin{array}{ll} \min & c_1x_1 + \cdots + c_nx_n \\ \text{s.t.} & x_1, \cdots, x_n \geq 0 \end{array}$$

If at least one cost coefficient is negative the value of the objective function is unbounded from below (in the following termed "equal to $-\infty$") with the corresponding variable unbounded (termed "equal to ∞") and all other variables equal to 0, otherwise it is 0 with all variables equal to 0.

The blending problem P of 2.2 is not of the form just described. However, such a problem may easily be constructed from P: Measure the violation of each of the original constraints by the difference between the right-hand and the left-hand side:

$$\begin{array}{l} 7 - (2x_1 + 3x_2 + 5x_4) \\ 2 - (3x_1 + 2x_3 + 4x_4) \end{array}$$

Multiply these by penalty factors y_1 and y_2 and add them to the objective function:

$$(PR(y_1, y_2)) \qquad \min_{x_1, \cdots, x_4 \geq 0} \left\{ \begin{array}{rl} & 13x_1 + 6x_2 + 4x_3 + 12x_4 \\ + & y_1(7 - 2x_1 - 3x_2 - 5x_4) \\ + & y_2(2 - 3x_1 - 2x_3 - 4x_4) \end{array} \right\}$$

We have now constructed a family of relaxed problems, one for each value of y_1, y_2, which are easy to solve. None of these seem to be the problem we actually want to solve, but the solution of each $PR(y_1, y_2)$ gives some information regarding the solution of P. It is a lower bound for any y_1, y_2, and the maximum of these lower bound turns out to be equal to the optimal value of P. The idea of replacing a difficult problem by an easier one, for which the optimal solution provides a lower bound for the optimal solution of the original problem, is also the key to understanding Branch-and-Bound methods in integer programming.

	x_1	x_2	x_3	x_4	
Red. Costs	15/2	0	3	0	-15
x_2	-7/12	1	-5/6	0	3/2
x_4	3/4	0	1/2	1	1/2

Table 2.2: Optimal Simplex tableau for problem LP

Let opt(P) and opt($PR(.)$) denote the optimal values of P and $PR(.)$ resp. Now observe the following points:

1. $\forall y_1, y_2 \in \mathcal{R}$: opt$(PR(y_1, y_2) \leq$ opt(P)
2. $\max_{y_1, y_2 \in \mathcal{R}}$ (opt$(PR(y_1, y_2))) \leq$ opt(P)

1) states that opt$(PR(y_1, y_2)$ is a lower bound for opt(P) for any choice of y_1, y_2 and follows from the fact that for any set of values for $x_1, \cdots, x_4$ satisfying the constraints of P, the values of P and PR are equal since the terms originating in the violation of the constraints vanish. Hence opt$(PR(y_1, y_2))$ is found by minimizing over a set containing all values of feasible solutions to P implying 1). Since 1) holds for all pairs y_1, y_2 it must also hold for the pair giving the maximum value of opt$(PR(.))$, which is 2). In the next section we will prove that

$$\max_{y_1, y_2 \in \mathcal{R}} (opt(PR(y_1, y_2))) = opt(P)$$

The best bound for our LP problem P is thus obtained by finding optimal multipliers for the relaxed problem. We have here tacitly assumed that P has an optimal solution, i.e. it is neither infeasible nor unbounded - we return to that case in Section 2.3.

Turning back to the relaxed problem the claim was that it is easily solvable for any *given* y_1, y_2 . We just collect terms to find the coefficients of $x_1, \cdots, x_4$ in $PR(y_1, y_2))$:

$$(PR(y_1, y_2)) \qquad \min_{x_1, \cdots, x_4 \geq 0} \left\{ \begin{array}{rrrrr} & (13 & -2y_1 & -3y_2) & x_1 \\ + & (6 & -3y_1 &) & x_2 \\ + & (4 & & -2y_2) & x_3 \\ + & (12 & -5y_1 & -4y_2) & x_4 \\ + & & 7y_1 & +2y_2 & \end{array} \right\}$$

Since y_1, y_2 are fixed the term $7y_1 + 2y_2$ in the objective function is a constant. If any coefficient of a variable is less than 0 the value of PR is $-\infty$. The lower bound for opt(P) provided by such a pair of y-values is of no value. Hence, we concentrate on y-values for which this does not happen. These pairs are exactly those assuring that each coefficient for $x_1, \cdots, x_4$ is non-negative:

$$\begin{array}{llllllll}
(13 & -2y_1 & -3y_2) & \geq 0 & \Leftrightarrow & 2y_1 & +3y_2 & \leq & 13 \\
(6 & -3y_1 &) & \geq 0 & \Leftrightarrow & 3y_1 & & \leq & 6 \\
(4 & & -2y_2) & \geq 0 & \Leftrightarrow & & 2y_2 & \leq & 4 \\
(12 & -5y_1 & -4y_2) & \geq 0 & \Leftrightarrow & 5y_1 & +4y_2 & \leq & 12
\end{array}$$

If these constraints all hold the optimal solution to $PR(y_1, y_2)$ has $x_1, \cdots, x_4$ all equal to 0 with a value of $7y_1 + 2y_2$ which, since y_1, y_2 are finite, is larger than $-\infty$. Since we want to maximize the lower bound $7y_1 + 2y_2$ on the objective function value of P, we have to solve the following problem to find the optimal multipliers:

$$(DP) \qquad \begin{array}{lllll}
\max & 7y_1 & +2y_2 & & \\
\text{s.t.} & 2y_1 & +3y_2 & \leq & 13 \\
& 3y_1 & & \leq & 6 \\
& & 2y_2 & \leq & 4 \\
& 5y_1 & +4y_2 & \leq & 12
\end{array}$$

$$y_1, y_2 \in \mathcal{R}$$

The problem DP resulting from our reformulation is exactly the dual problem of P. It is again a linear programming problem, so nothing is gained with respect to ease of solution – we have no reason to believe that DP is any easier to solve than P. However, the above example indicates that linear programming problems appear in pairs defined on the *same* data with one being a minimization and the other a maximization problem, with variables of one problem corresponding to constraints of the other, and with the type of constraints determining the signs of the corresponding dual variables. Using the multiplier approach we have derived the dual problem DP of our original problem P, and we have through 1) and 2) proved the so-called Weak Duality Theorem – that opt(P) is greater than or equal to opt(DP).

In the next section we will discuss the construction in general, the proof of the Duality Theorem as stated in the introduction, and the question of unboundedness/infeasibility of the primal problem. We end this section by deriving DP as frequently done in textbooks on linear programming.

The company PresChem selling the artificially produced counterparts R' and Q' to Pasta Basta at prices r and q is considering to increase these as much as possible well knowing that many consumers of Pasta Basta lasagne do not care about ecology but about prices. These customers want as cheap a product as

possible, and Pasta Basta must also produce a cheaper product to maintain its market share.

If the complete production of lasagne is based on P' and Q', the profit of PresChem is $7r + 2q$. Of course r and q cannot be so large that it is cheaper for Pasta Basta to extract the necessary amount of R and Q from fruit. For example, at the cost of 13, Pasta Basta can extract 2 units of R and 3 units of Q from one ton of F1. Hence

$$2r + 3q \leq 13$$

The other three types of fruit give rise to similar constraints. The prices r and q are normally regarded to be non-negative, but the very unlikely possibility exists that it may pay off to offer Pasta Basta money for each unit of one preservative used in the production provided that the price of the other is large enough. Therefore the prices are allowed to take also negative values. The optimization problem of PresChem is thus exactly DP.

2.3 General formulation of dual LP problems

2.3.1 Proof of the Duality Theorem

The typical formulation of an LP problem with n nonnegative variables and m equality constraints is

$$\begin{array}{lrcl} \min & cx & & \\ & Ax & = & b \\ & x & \geq & 0 \end{array}$$

where c is an $1 \times n$ matrix, A is an $m \times n$ matrix and b is an $n \times 1$ matrix of reals. The process just described can be depicted as follows:

$$\left.\begin{array}{lrcl} \min & cx & & \\ & Ax & = & b \\ & x & \geq & 0 \end{array}\right\} \quad \mapsto$$

$$\max_{y \in \mathcal{R}^m} \{\min_{x \in \mathcal{R}^n_+} \{cx + y(b - Ax)\}\} \quad \mapsto$$

$$\max_{y \in \mathcal{R}^m} \{\min_{x \in \mathcal{R}^n_+} \{(c - yA)x + yb\}\} \quad \mapsto$$

$$\left\{\begin{array}{lrcl} \max & yb & & \\ & yA & \leq & c \\ & y & free & \end{array}\right.$$

The proof of the Duality Theorem proceeds in the traditional way: We find a set of multipliers which satisfy the dual constraints and gives a dual objective function value equal to the optimal primal value.

Assuming that P and DP have feasible solutions implies that P can be neither infeasible nor unbounded. Hence an optimal basis $\mathcal{B}$ and a corresponding optimal basic solution $x_{\mathcal{B}}$ for P exists. The vector $y_{\mathcal{B}} = c_{\mathcal{B}}\mathcal{B}^{-1}$ is called the **dual solution** or the set of **Simplex multipliers** corresponding to $\mathcal{B}$. The vector satisfies that if the reduced costs of the Simplex tableau is calculated using $y_{\mathcal{B}}$ as π in the general formula

$$\bar{c} = c - \pi A$$

then $\bar{c}_i$ equals 0 for all basic variables and $\bar{c}_j$ is non-negative for all non-basic variables. Hence,

$$y_{\mathcal{B}} A \leq c$$

holds showing that $y_{\mathcal{B}}$ is a feasible dual solution. The value of this solution is $c_{\mathcal{B}}\mathcal{B}^{-1}b$, which is exactly the same as the primal objective value obtained by assigning to the basic variables $x_{\mathcal{B}}$ the values defined by the updated right-hand side $\mathcal{B}^{-1}b$ multiplied by the vector of basic costs $c_{\mathcal{B}}$.

The case in which the problem P has no optimal solution is for all types of primal and dual problems dealt with as follows. Consider first the situation where the objective function is unbounded on the feasible region of the problem. Then any set of multipliers must give rise to a dual solution with value $-\infty$

(resp. $+\infty$ for a maximization problem) since this is the only "lower bound" ("upper bound") allowing for an unbounded primal objective function. Hence, no set of multipliers satisfy the dual constraints, and the dual feasible set is empty. If maximizing (resp. minimizing) over an empty set returns the value $-\infty$ (resp.$+\infty$), the desired relation between the primal and dual problem with respect to objective function value holds – the optimum values are equal.

Finally, if no primal solution exist we minimize over an empty set - an operation returning the value $+\infty$. In this case the dual problem is either unbounded or infeasible.

2.3.2 Other types of dual problem pairs

Other possibilities of combinations of constraint types and variable signs of course exist. One frequently occurring type of LP problem is a maximization problem in non-negative variables with less than or equal constraints. The construction of the dual problem is outlined below:

$$\left.\begin{array}{lrcl} \max & cx & & \\ & Ax & \leq & b \\ & x & \geq & 0 \end{array}\right\} \quad \mapsto$$

$$\min_{y\in\mathcal{R}^m_+}\{\max_{x\in\mathcal{R}^n_+}\{cx + y(b - Ax)\}\} \quad \mapsto$$

$$\min_{y\in\mathcal{R}^m_+}\{\max_{x\in\mathcal{R}^n_+}\{(c - yA)x + yb\}\} \quad \mapsto$$

$$\left\{\begin{array}{lrcl} \min & yb & & \\ & yA & \geq & c \\ & y & \geq & 0 \end{array}\right.$$

Note here that the multipliers are restricted to being non-negative, thereby ensuring that for any feasible solution, $\hat{x_1}, \cdots, \hat{x_n}$, to the original problem, the relaxed objective function will have a value greater than or equal to that of the original objective function since $b - A\hat{x}$ and hence $y(b - A\hat{x})$ will be non-negative. Therefore the relaxed objective function will be pointwise larger than or equal to the original one on the feasible set of the primal problem, which ensures that an upper bound results for all choices of multipliers. The set of multipliers minimizing this bound must now be determined.

Showing that a set of multipliers exists such that the optimal value of the relaxed problem equals the optimal value of the original problem is slightly more

complicated than in the previous case. The reason is that the value of the relaxed objective function no longer is equal to the value of the original one for each feasible point, it is larger than or equal to this.

A standard way is to formulate an LP problem P' equivalent to the given problem P by adding a **slack variable** to each of the inequalities thereby obtaining a problem with equality constraints:

$$\left.\begin{array}{lrcl} \max & cx & & \\ & Ax & \leq & b \\ & x & \geq & 0 \end{array}\right\} \quad = \quad \left\{\begin{array}{lrcl} \max & cx & +0s & \\ & Ax & +Is & = \quad b \\ & & x, s & \geq \quad 0 \end{array}\right.$$

Note now that if we derive the dual problem for P' using multipliers we end up with the dual problem of P: Due to the equality constraints, the multipliers are now allowed to take both positive and negative values. The constraints on the multipliers imposed by the identity matrix corresponding to the slack variables are, however,

$$yI \geq 0$$

i.e. exactly the non-negativity constraints imposed on the multipliers by the inequality constraints of P. The proof just given now applies for P' and DP', and the non-negativity of the optimal multipliers $y_{\mathcal{B}}$ are ensured through the sign of the reduced costs in optimum since these now satisfy

$$\bar{c} = (c\ 0) - y_{\mathcal{B}}(A\ I) \leq 0 \quad \Leftrightarrow \quad y_{\mathcal{B}}A \geq c \quad \wedge \quad y_{\mathcal{B}} \geq 0$$

Since P' and P are equivalent the theorem holds for P and DP as well.

The interplay between the types of primal constraints and the signs of the dual variables is one of the issues of duality, which often creates severe difficulties in the teaching situation. Using the common approach to teaching duality, often no explanation of the interplay is provided. We have previously illustrated this interplay in a number of situations. For the sake of completeness we now state all cases corresponding to a primal minimization problem – the case of primal maximization can be dealt with likewise.

First note that the relaxed primal problems provide *lower bounds*, which we want to *maximize*. Hence the relaxed objective function should be pointwise *less than or equal* to the original one on the feasible set, and the dual problem

is a *maximization problem*. Regarding the signs of the dual variables we get the following for the three possible types of primal constraints ($\mathbf{A}_{i.}$ denotes the i'th row of the matrix $\mathbf{A}$):

$\mathbf{A}_{i.}x \leq b_i$ For a feasible x, $b_i - A_{i.}x$ is larger than or equal to 0, and $y_i(b_i - A_{i.}x)$ should be non-positive. Hence, y_i should be non-positive as well.

$\mathbf{A}_{i.}x \geq b_i$ For a feasible x, $b_i - A_{i.}x$ is less than or equal to 0, and $y_i(b_i - A_{i.}x)$ should be non-positive. Hence, y_i should be non-negative.

$\mathbf{A}_{i.}x = b_i$ For a feasible x, $b_i - A_{i.}x$ is equal to 0, and $y_i(b_i - A_{i.}x)$ should be non-positive. Hence, no sign constraints should be imposed on y_i.

Regarding the types of the dual constraints, which we previously have not explicitly discussed, these are determined by the sign of the coefficients to the variables in the relaxed primal problem in combination with the sign of the variables themselves. The coefficient of x_j is $(c - yA)_j$. Again we have three cases:

$\mathbf{x}_j \geq 0$ To avoid unboundedness of the relaxed problem $(c - yA)_j$ must be greater than or equal to 0, i.e. the j'th dual constraint will be $(yA)_j \leq c_j$.

$\mathbf{x}_j \leq 0$ In order not to allow unboundedness of the relaxed problem $(c - yA)_j$ must be less than or equal to 0, i.e. the j'th dual constraint will be $(yA)_j \geq c_j$.

$\mathbf{x}_j$ *free* In order not to allow unboundedness of the relaxed problem $(c - yA)_j$ must be equal to 0 since no sign constraints on x_j are present, i.e. the j'th dual constraint will be $(yA)_j = c_j$.

2.3.3 The Dual Problem for Equivalent Primal Problems

In the previous section it was pointed out that the two equivalent problems

$$\left.\begin{array}{lrcl} \max & cx & & \\ & Ax & \leq & b \\ & x & \geq & 0 \end{array}\right\} = \left\{\begin{array}{lrcl} \max & cx + 0s & & \\ & Ax + Is & = & b \\ & x, s & \geq & 0 \end{array}\right.$$

give rise to exactly the same dual problem. This is true in general. Suppose P is any given minimization problem in variables, which may be non-negative, non-positive or free. Let P' be a minimization problem in standard form, i.e a

problem in non-negative variables with equality constraints, constructed from P by means of addition of slack variables to $\leq$-constraints, subtraction of surplus variables from $\geq$-constraints, and change of variables. Then the dual problems of P and P' are equal.

We have commented upon the addition of slack variables to $\leq$-constraints in the preceding section. The subtraction of slack variables are dealt with similarly. A constraint

$$a_{i1}x_1 + \cdots + a_{in}x_n \geq b_i \Leftrightarrow (b_i - a_{i1}x_1 - \cdots - a_{in}x_n) \leq 0$$

gives rise to a multiplier, which must be non-negative in order for the relaxed objective function to provide a lower bound for the original one on the feasible set. If a slack variable is subtracted from the left-hand side of the inequality constraint to obtain an equation

$$a_{i1}x_1 + \cdots + a_{in}x_n - s_i = b_i \Leftrightarrow (b_i - a_{i1}x_1 - \cdots - a_{in}x_n) + s_i = 0$$

the multiplier must now be allowed to vary over $\mathcal{R}$. A new constraint in the dual problem, however, is introduced by the column of the slack variable, cf. Section 2.2:

$$-y_i \leq 0 \Leftrightarrow y_i \geq 0,$$

thereby reintroducing the sign constraint for y_i.

If a non-positive variable x_j is substituted by $x_j^{'}$ of opposite sign, all signs in the corresponding column of the Simplex tableau change. For minimization purposes however, a positive sign of the coefficient of a non-positive variable is beneficial, whereas a negative sign of the coefficient of a non-negative variable is preferred. The sign change of the column in combination with the change in preferred sign of the objective function coefficient leaves the dual constraint unchanged.

Finally, if a free variable x_j is substituted by the difference between two non-negative variables $x_j^{'}$ and $x_j^{''}$ two equal columns of opposite sign are introduced. These give rise to two dual constraints, which when taken together result in the same dual equality constraint as obtained directly.

The proof of the Duality Theorem for all types of dual pairs P and DP of LP problems may hence be given as follows: Transform P into a standard problem P' in the well known fashion. P' also has DP as its dual problem. Since the Duality Theorem holds for P' and DP as shown previously and P' is equivalent to P, the theorem also holds for P and DP.

2.4 Discussion: Pros and Cons of the Approach

The main advantages of teaching duality based on multipliers are in my opinion

- the independence of the problem modeled by the primal model and the introduction of the dual problem, i.e. that no story has go with the dual problem,
- the possibility to avoid problem transformation and "duality by definition" in the introduction of general duality in linear programming,
- the clarification of the interplay between the sign of variables and the type of the corresponding constraints in the dual pair of problems,
- the early introduction of the idea of getting information about the optimum of an optimization problem through bounding using the solution of an easier problem,
- the possibility of introducing partial dualization by including only some constraint violations in the objective function, and
- the resemblance with duality in non-linear programming, cf. [3].

The only disadvantage in my view is one listed also as an advantage:

- the independence of the introduction of the dual problem and the problem modelled by the primal model

since this may make the initial motivation weaker. I do not advocate that duality should be taught based solely on the multiplier approach, but rather that it is used as a supplement to the traditional presentation (or vice versa). In my experience, it offers a valuable supplement, which can be used to avoid the situation of frustrated students searching for an intuitive interpretation of the dual problem in cases, where such an interpretation is not natural. The decision on whether to give the traditional presentation of duality or the multiplier approach first of course depends on the particular audience.

Bibliography

[1] R. E. Bellman and S. E. Dreyfus, Applied Dynamic Programming, Princeton University Press, 1962.

[2] G. B. Dantzig, Linear Programming and Extensions, Princeton University Press, 1963.

[3] A. M. Geoffrion, Duality in Non-Linear Programming: A Simplified Applications-Oriented Approach, SIAM Review 13, 1 1 - 37, 1971.

[4] A. M. Geoffrion, Lagrangean Relaxation for Integer Programming, Math. Prog. Study **2** 82 - 114, 1974.

[5] M. X. Goemans, Linear Programming, Course Notes, 1994, available from http://theory.lcs.mit.edu/ goemans.

[6] J. Krarup and C. Vanderhoeft, Belgium's Best Beer and Other Stories, VUB Press (to appear).

[7] K. G. Murty, Linear and Combinatorial Programming, John Wiley, 1976.

[8] H. P. Williams, Model Solving in Mathematical Programming, John Wiley, 1993.

[9] W. L. Winston, Operations Research - Applications and Algorithms, Int. Thomson Publishing, 1994.

Part I

Network Optimization

Chapter 3

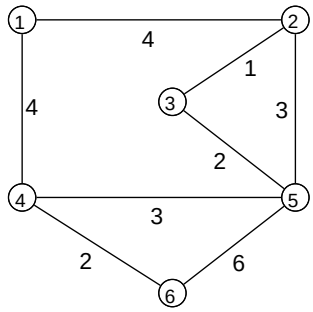
The Minimum Spanning Tree Problem

The minimum spanning tree problem is one of the oldest and most basic graph problems in computer science and Operations Research. Its history dates back to Boruvka's algorithm developed in 1926 and still today it is an actively researched problem. Boruvka, a Czech mathematician was investigating techniques to secure an efficient electrical coverage of Bohemia.

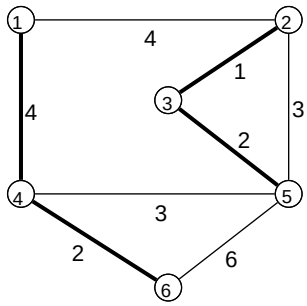
Let an undirected graph $G = (V, E)$ be given. Each of the edges are assigned a cost (or weight or length) w, that is, we have a cost function $w : E \to R$. The network will be denoted $G = (V, E, w)$. We will generally assume the cost to be non-negative, but as we will see it is more a matter of convenience.

Let us recall that a tree is a connected graph with no cycles, and that a spanning tree is a subgraph in G that is a tree and includes all vertices of G i.e. V.

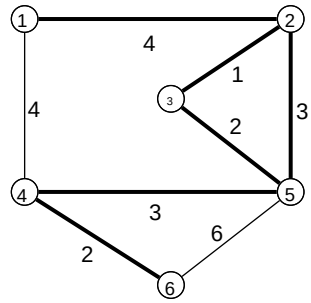
The minimum spanning tree (MST) problem calls for finding a spanning tree whose total cost is minimum. The total cost is measured as the sum of the costs of all the edges in the tree. Figure 3.1 shows an MST in a graph G. Note that a network can have many MST's.



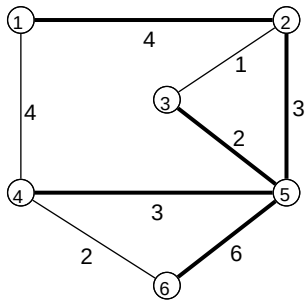
(a) A network



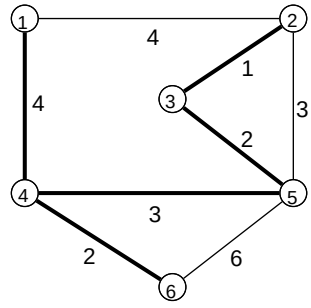
(b) Not a spanning tree as $\langle 1, 4, 6 \rangle$ and $\langle 2, 3, 5 \rangle$ are not connected



(c) Not a spanning tree as the subgraph contains a cycle $\langle 2, 3, 5, 2 \rangle$



(d) A spanning tree with a value of 18



(e) A minimum spanning tree with a value of 12

Figure 3.1: Examples illustrating the concept of spanning and minimum spanning tree.

We will denoted edges in the spanning tree as *tree edges* and those edges not part of the spanning tree as *nontree edges*.

The minimum spanning tree problem arises in a wide number of applications both as a "stand alone" problem but also as subproblem in more complex problems.

As an example consider a telecommunication company planning to lay cables to a new neighborhood. In its operations it is constrained to only lay cables along certain connections represented by the edges. At the same time a network that reaches all houses must be build (ie. a spanning tree). For each edge a cost of digging down the cables have been computed. The MST gives us the cheapest solution that connects all houses.

At some few occasions you might meet the maximum spanning tree problem (or as Wolsey calls it the "maximum weight problem"). Solving the maximum spanning tree problem, where we wish to maximize the total cost of the constituent edges, can be solved by multiplying all costs with -1 and the solve the minimum spanning problem.

3.1 Optimality conditions

A central concept in understanding and proving the validity of the algorithms for the MST is the concept of a *cut* in an undirected graph. A cut $\mathcal{C} = \{X, X'\}$ is a partition of the set of vertices into two subsets. For $X \subset V$, denote the set of arcs crossing the cut $\delta X = \{(u, v) \in E : u \in X, v \in V \setminus X\}$. The notion of a cut is shown in Figure 3.2. Edges that have one endpoint in the set X and the other in the endpoint X' is said to *cross the cut*.

We say that a cut **respects** a set A of edges if no edge in A crosses the cut.

Let us use the definition of a cut to establish a mathematical model for the problem.

We define a binary variable x_{ij} for each edge (i, j). Let

$$x_{ij} = \begin{cases} 1 & \text{if } (i,j) \text{ is in the subgraph} \\ 0 & \text{otherwise} \end{cases}$$

For a spanning tree of n vertices we need $n - 1$ edges. Furthermore, for each possible cut $\mathcal{C}$ at least one of the edges crossing the cut must be in the solution

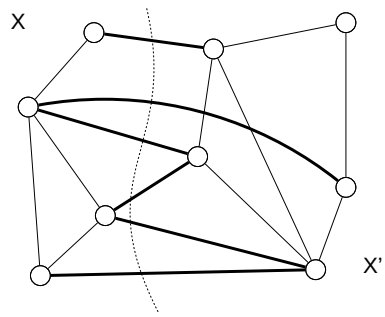


Figure 3.2: The cut C defined by the partitioning $X \subset V$ and $X' = V \setminus X$ is shown for a graph G. The edges crossing the cut $\{X, X'\}$ are shown in bold.

– otherwise the subgraph defined by the solution is not connected. Therefore we get the following model:

The Cut-set formulation of the MST

$$\begin{aligned} \min \quad & \sum_{e \in E} w_e x_e \\ s.t. \quad & \sum_{e \in E} x_e = n - 1 \\ & \sum_{e \in \delta(S)} x_e \geq 1, \qquad \emptyset \subset S \subset V \\ & x_e \in \{0, 1\} \end{aligned}$$

Sub tour elimination formulation of the MST

$$\begin{aligned} \min \quad & \sum_{e \in E} w_e x_e \\ s.t. \quad & \sum_{e \in E} x_e = n - 1 \\ & \sum_{e \in \gamma(S)} x_e \leq |S| - 1, \quad \emptyset \subset S \subset V \\ & x_e \in \{0, 1\} \end{aligned}$$

where $\gamma(S) = \{(i, j) : i \in S, j \in S\}$.

Note that with the definition of a cut if we delete any tree edge (i, j) from a

spanning tree it will partition the vertex set V into two subsets. Furthermore if we insert an edge (i, j) into a spanning tree we will create exactly one cycle. Finally, for any pair of vertices i and j, the path from i to j in the spanning tree will be uniquely defined (see Figure 3.3).

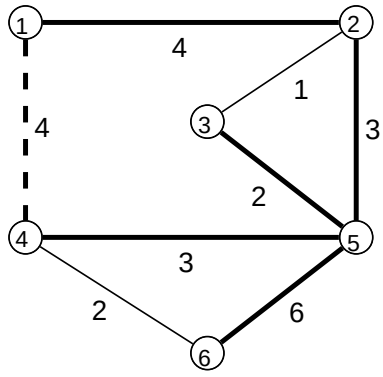


Figure 3.3: Adding the edge $(1, 4)$ to the spanning tree forms a *unique* cycle $\langle 1, 4, 5, 2, 1\rangle$

In this chapter we will look at two algorithms for solving the minimum spanning tree problem. Although different they both use the greedy principle in building an optimal solution. The greedy principle advocates making the choice that is best at the moment, and although it in general is not guaranteed to provide optimal solutions it does in this case. In many textbooks on algorithms you will also see the algorithms for the minimum spanning tree problem being used as it is a classical application of the greedy principle.

The greedy strategy shared by both our algorithms can be described by the following "generic" algorithm, which grows the minimum spanning tree one edge at a time. The algorithm maintains a set A with the following condition:

> Prior to each iteration, A is a subset of edges of some minimum spanning tree.

At each iteration we determine an edge (i, j) that can be added to A without breaking the condition stated above. So that if A is a subset of edges of some minimum spanning tree before the iteration then $A \cup \{(i, j)\}$ is a subset of edges of some minimum spanning tree before the next iteration.

An edge with this property will be denoted a **safe** edge. So we get the following selection rule for a "generic" algorithm.

Algorithm 1: The Selection rule

1 select a cut in G that does not contain any selected edges
2 determine a safe edge of the cut-set and select it
3 if there are more than one safe edge select a random one of these

So how do we use the selection rule?

- The selection rule is used $n-1$ times as there are $n-1$ edges in a tree with n vertices. Each iteration will select exactly one edge.
- Initially none of the edges in the graph are selected. During the process one edge at a time will be selected and when the methods stops the selected edges form a MST for the given graph G.

Now the interesting question is, of course, how do find these safe edges? First, a safe edge must exist. The condition defines that there exists a spanning tree T such that $A \subseteq T$. As A must be a proper subset of T there must be an edge $(i,j) \in T$ such that $(i,j) \notin A$. The edge (i,j) is therefore a safe edge.

In order to come up with a rule for recognizing safe edges we need to define a **light** edge. A light edge crossing a cut is an edge with the minimum weight of any edge crossing the cut. Note that there can be more than one light edge crossing a cut. Now we can state our rule for recognizing safe edges.

Theorem 3.1 *Let $G = (V, E, w)$ be a connected, undirected graph (V, E) with a weight function $w : E \rightarrow R$. Let A be a subset of E that is part of some minimum spanning tree for G. Let $\{S, \bar{S}\}$ be any cut of G that respects A, and let (i,j) be a light edge crossing the cut $\{S, \bar{S}\}$. Then, edge (i,j) is safe for A.*

Proof: Let T be a minimum spanning tree that includes A. If T contains the light edge (i,j) we are done, so lets assume that this is not the case.

Now the general idea of the proof is to construct another minimum spanning tree T' that includes $A \cup \{(i,j)\}$ by using a cut-and-paste technique. This will then prove that (i,j) is a safe edge.

If we look at the graph $T \cup \{(i,j)\}$ it contains a cycle (see Figure 3.4). Let us call this cycle p. Since i and j are on opposite sides of the cut $\{S, \bar{S}\}$, there is at least one edge in T on the path p that also crosses the cut. Let (k,l) be such an edge. As the cut respects A (k,l) cannot be in A. Furthermore as (k,l) is

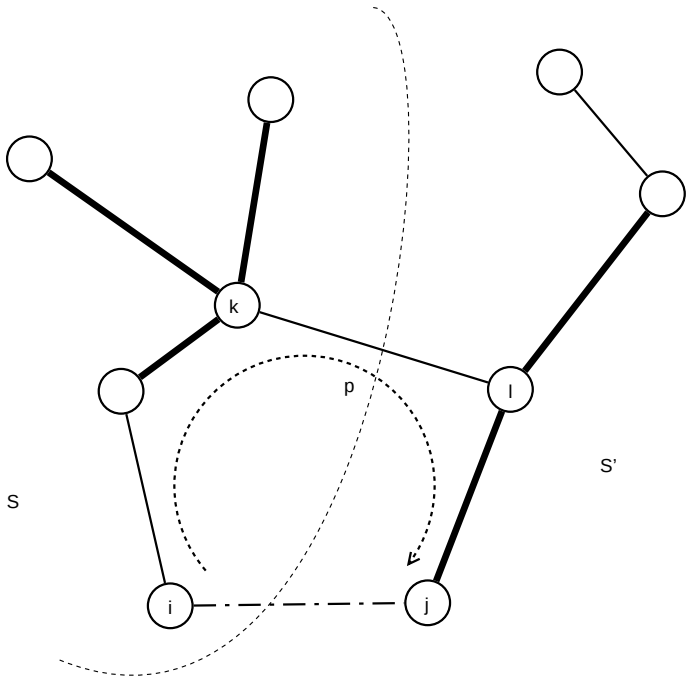


Figure 3.4: The edges in the minimum spanning tree T is shown, but not the edges in G. The edges already in A are drawn extra fat, and (i, j) is a light edge crossing the cut $\{S, S' = \bar{S}\}$. The edge (k, l) is an edge on the unique path p from i to j in T. A minimum spanning tree T' that contains (i, j) is formed by removing the edge (k, l) from T and adding (i, j).

on the unique path in T from i to j T would decompose into two components if we remove (k, l).

Adding (i, j) reconnects the two components thus forming a new spanning tree $T' = T \setminus \{(k, l)\} \cup \{(i, j)\}$. The question is whether T' is a spanning tree?

Since (i, j) is a light edge (that was one of the initial reasons for picking it) crossing the cut $\{S, \bar{S}\}$ and (k, l) also crosses the cut

$$w_{ij} \leq w_{kl}$$

And therefore

$$\begin{aligned} w_{T'} &= w_T - w_{kl} + w_{ij} \\ &\leq w_T \end{aligned}$$

and as T is a minimum spanning tree we have $w_T \leq w_{T'}$. So T' must also be a minimum spanning tree.

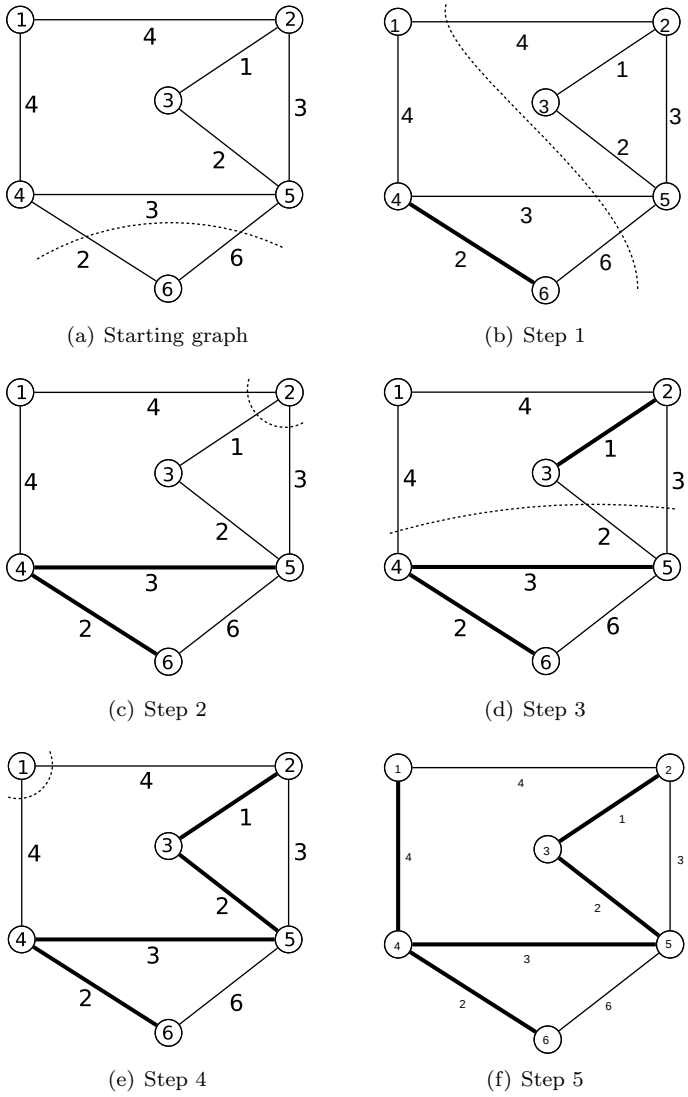
What we miss is just to prove that (i, j) is actually a safe edge for A. As $A \subseteq T$ and $(k, l) \notin A$ then $A \subseteq T'$, so $A \cup \{(i, j)\} \subseteq T'$. Consequently, since T' is a minimum spanning tree, (i, j) is safe for A.△

Now we can look at how an execution of the generic algorithm would look like. The selection rule tells us to pick a cut respecting A and then choose an edge of minimum weight in this cut.

Figure 3.5 illustrates the execution of the generic algorithm. Initially no edges are selected. We can consider any cut we would like. Let us use the cut $X = \{6\}$ and $X' = V \setminus X$ (a). The cheapest edge crossing the cut is $(4, 6)$. It is is added to the solution. We can now pick any cut not containing the edge $(4, 8)$. We consider the cut $X = \{1, 4, 6\}$ (b). Cheapest edge crossing the cut is $(4, 5)$ which is added to the solution. Next we consider the cut $X = \{2\}$ (c). Here the cheapest edge crossing the cut is $(2, 3)$ which is added to the solution. Now let us consider the cut $X = \{1, 2, 3\}$ (d). The edge $(3, 5)$ is now identified as the cheapest and is added to the solution. The next cut we consider is $X = \{1\}$ (e). This will add $(1, 4)$ to the solution, which has become a spanning tree. We can not find any cuts that does not contain edges in the tree crossing the cut (f).

More conceptually A defines a graph $G_A = (V, A)$. G_A is a forest, and each of the connected components of G_A is a tree. When the algorithm begins A is empty and the forests contains n trees, one for each vertex. Moreover, any safe edge (i, j) for A connects distinct components of G_A, since $A \cup \{(i, j)\}$ must be acyclic.

The two algorithms in section 3.2 and 3.3 use the following corollary to our theorem.



(a) Starting graph

(b) Step 1

(c) Step 2

(d) Step 3

(e) Step 4

(f) Step 5

Figure 3.5: The execution of the generic algorithm on our sample graph.

Corollary 3.2 *Let $G = (V, E, w)$ be a connected, undirected graph with weight function $w : E \to R$. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C = (V_C, E_C)$ be a component in the forest $G_A = (V, A)$. If (i, j) is a light edge connecting C to some other component in G_A, then (i, j) is safe for A.*

Proof: The cut $\{V_C, \bar{V}_C\}$ respects A, and (i, j) is a light edge for this cut. Therefore (i, j) is safe for A. $\triangle$

3.2 Kruskal's Algorithm

The condition of theorem 3.1 is a natural basis for the first of the algorithms for the MST that will be presented.

We build the minimum spanning tree from scratch by adding one edge at a time. First we sort all edges in nondecreasing order of their cost. They are then stored in a list called $\mathcal{L}$. Furthermore, we define a set F that will contain the edges that have been chose to be part of the minimum spanning tree being constructed. Initially F will therefore be empty. Now we examine the edges one at a time in the order they appear in $\mathcal{L}$ and check whether adding each edge to the current set F will create a cycle. If it creates a cycle then one of the edges on the cycle cannot be part or the minimum spanning tree and as we are checking the edges in nondecreasing order, the currently considered is the one with the largest cost. We therefore discard it. Otherwise the edge is added to F. We terminate when $|F| = n - 1$. At termination, the edges in F constitute a minimum spanning tree T. This algorithm is known as *Kruskal's algorithm*. Pseudo-code for the algorithm is presented in Algorithm 2.

The argument for correctness follows from that if an edge is added, it is the first edge added across some cut, and due to the non-decreasing weight property, it is a minimum weight edge across that cut. That is, it is a light edge but then it is also a safe edge.

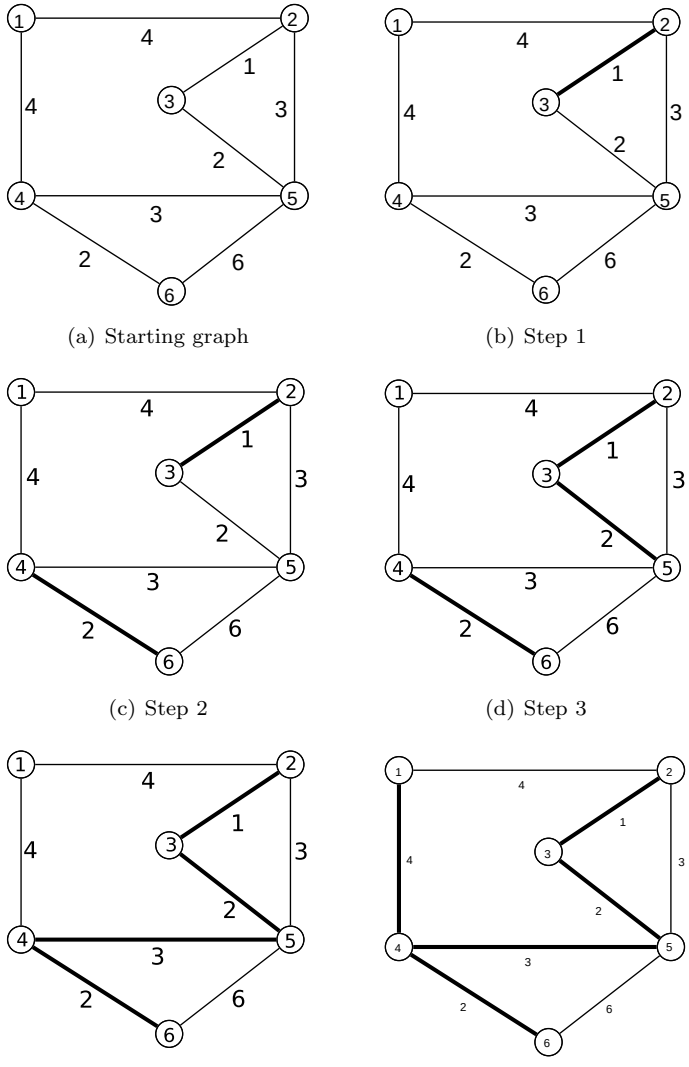
Algorithm 2: Kruskal's algorithm

Data: Input parameters
Result: The set F containing the edges in the MST

```
1 F ← ∅
2 L ← edges sorted by nondecreasing weight
3 for each (u, v) ∈ L do
4     if a path from u to v in F does not exist then
5         F ← F ∪ (u, v)
```

Running Kruskal's algorithm on the example of Figure 3.1 results in the steps depicted in Figure 3.6. First the edges are sorted in non-decreasing order: $(2,3)$, $(4,6)$, $(3,5)$, $(2,5)$, $(4,5)$, $(1,4)$, $(1,2)$, $(5,6)$. First the edge $(2,3)$ is selected (b). Then edge $(4,6)$ is selected (c), and afterwards the edge $(3,5)$ is selected (d). The edge $(2,5)$ is discarded as it would create a cycle $(2 \to 3 \to 5 \to 2)$ (e). Instead $(4,5)$ is selected $(1,4)$ is selected (f). Had the order between $(1,4)$ and $(1,2)$ been different we would have chosen differently. As 5 edges are selected

for a graph with 6 vertices we are done.



(a) Starting graph

(b) Step 1

(c) Step 2

(d) Step 3

(e) Step 4

(f) Step 5

Figure 3.6: The execution of Kruskal's algorithm on our sample graph.

The time complexity of Kruskal's algorithm consists of the time complexity of first sorting the edges in non-decreasing order and then checking whether we produce a cycle or not. Sorting can most efficiently be done in $O(m \log m) = O(m \log n^2) = O(m \log n)$ which is achieved by eg. Merge Sort (see e.g. [1]).

The time to detect a cycle depends on the method used for this step. A naive implementation would look like this. The set F is a any stage of the execution of the algorithm a forest. In step 3 in the execution of our example F contains three trees, namely $\langle 1 \rangle, \langle 2, 3, 5 \rangle$ and $\langle 4, 6 \rangle$. The subsets defining the trees in the forest will be denoted $F_1, F_2, \ldots, F_f$. These can be stored as singly linked lists. When we want to examine whether inserting (k, l) will create a cycle or not, we scan through these linked lists and check whether both the vertices k and l belongs to the same linked list. If that is the case adding (k, l) would produce a cycle. Otherwise we add edge (k, l) and thereby merge the two lists containing k and l into one list. This data structure requires $O(n)$ time for each edge we examine. So the overall time complexity will be $O(mn)$

A better performance can be achieved using more advanced data structures. In this way we can get a time complexity of $O(m + n \log n)$ plus the time it takes to sort the edges, all in all $O(m \log n)$.

3.3 Prim's algorithm

The idea of the Prim's algorithm is to grow the minimum spanning tree from a single vertex. We arbitrarily select a vertex as the starting point. Then we iteratively add one edge at a time. We maintain a tree spanning a subset of vertices S and add a nearest neighbour to S. This is done by finding an edge (i, j) of minimum cost with $i \in S$ and $j \in \bar{S}$, that is, finding an edge of minimum cost crossing the cut $\{S, \bar{S}\}$. The algorithm terminates when $S = V$.

An initial look at the time complexity before we state the algorithm in pseudo-code suggests that we select a minimum cost edge $n - 1$ times, and in order to find the minimum cost edge in each iteration we must search through all the m edges, giving in total the cost $O(mn)$. Hence, the bottleneck becomes the identification of the minimum cost edge.

If we for each vertex adds two pieces of information we can improve the time complexity:

1. l_i represents the minimum cost of an edge connecting i and some vertices in S, and

2. p_i that identifies the other endpoint of a minimum cost edge incident with vertex i.

If we maintain these values, we can easily find the minimum cost of an edge in the cut. We simply compute $\arg\min\{l_i : i \in \bar{S}\}$. The label p_i identifies the edge (p_i, i) as the minimum cost edge over the cut $\{S, \bar{S}\}$, and therefore it should be added in the current iteration.

In order to maintain updated information in the labels we must update the values of a label as it is moved from $\bar{S}$ to S. This reduces the time complexity of the identification from $O(m)$ to $O(n)$. Thus the total time complexity for Prim's algorithm will be $O(n^2)$ (see algorithm 3).

Algorithm 3: Prim's algorithm

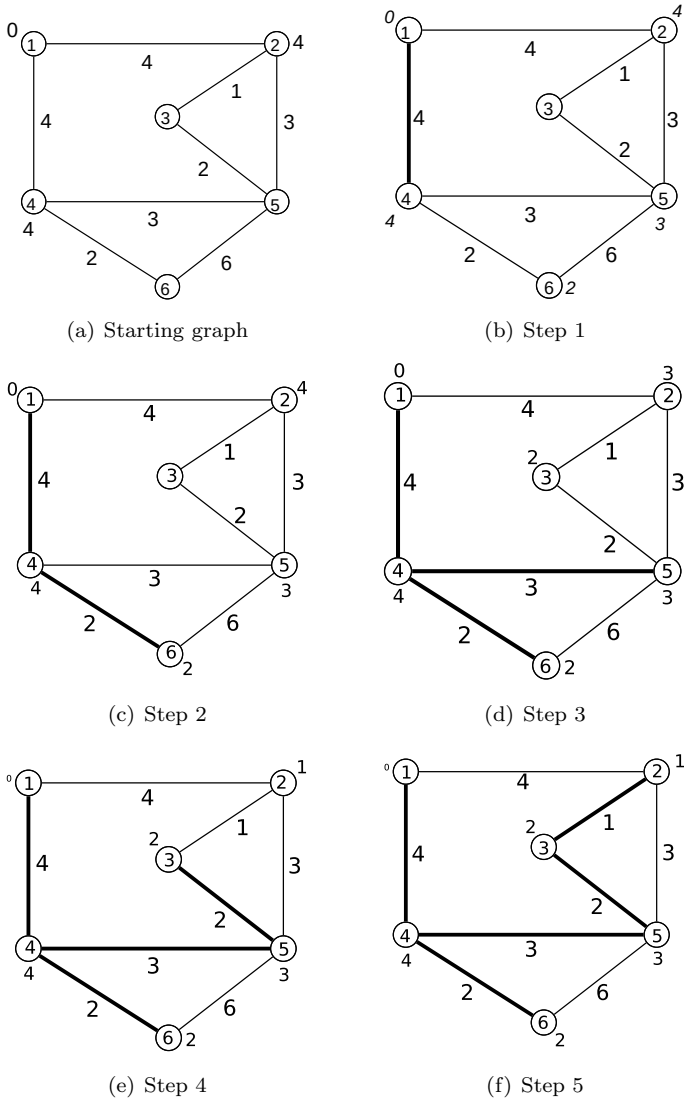
Data: A graph $G = (V, E)$ with n vertices and m edges. A source vertex is denoted r. For each edge $(i, j) \in E$ an edge weight w_{ij} is given.
Result: An n-vector p the predecessor vertex for i in the minimum spanning tree.

```
1  Q ← V
2  p_r ← 0, l_r ← 0
3  p_i ← NIL, l_i ← ∞ for i ∈ V \ {r}
4  while Q ≠ ∅ do
5      i ← arg min_{j∈Q}{l_j}
6      Q ← Q \ {i}
7      for (i, j) ∈ E where j ∈ Q do
8          if w_ij < l_j then
9              p_j ← i
10             l_j ← w_ij
```

Time complexity depends on the exact data structures, however, the basic implementations with list representation results in a running time of $O(n^2)$

A more advanced implementation make use of the *heap* data structure. The "classical" binary heap would give us a time complexity of $O(m \log n)$. Further information can be found in section 3.4.

Figure 3.7 show the execution of Prim's algorithm on our example graph. Initially we choose 1 to be the vertex we expand from. Each vertex will get a label of $+\infty$ except vertex 1 that will get the value 0. So initially vertex 1 is included in S and the labels of vertex 4 are updated to 4 (due to the edge $(1, 4)$) and the label of vertex 2 is also updated to 4 (due to the edge $(1, 2)$) (a). Now vertex 4



(a) Starting graph

(b) Step 1

(c) Step 2

(d) Step 3

(e) Step 4

(f) Step 5

Figure 3.7: The execution of Prims's algorithm on our sample graph. We have choosen vertex 1 to be the initial vertex in the tree. A vertex without a "distance" symbolizes a distance of $+\infty$.

is selected for inclusion into S. The choice between 2 and 4 is actually arbitraty as both vertices have a label of value 4 (the lowest value of all labels) (b). The selection of vertex 4 updates the values of vertex 5 and 6 to 3 resp. 2. In the next step we select vertex 6 and enter it into set S thereby adding the edge $(4, 6)$ to our solution (c). Next we select vertex 5 which leads to an update of the label of vertex 3 but also vertex 2 (the value is lowered from 4 to 3) (d). We then select vertex 3 and update the value of label 2 (e) before finally selecting vertex 2. Now $S = V$ and the algorithm terminates.

3.4 Supplementary Notes

For more information of sorting and also data structures for an efficient implementation of the algorithms described in this chapter see [1].

For a description of how the better time complexity can be established for Kruskal's algorithm we refer to [2].

There exist different versions of heap data structures with different theoretical and practical properties (Table 3.1 shows some worst case running times for Prim's algorithm depending on the selected data structure). An initial discussion on the subject related to MST can be found in [2].

heap type	**running time**
Binary heap	$O(m \log n)$
d-heap	$O(m \log_d n))$ with $d = \max\{2, \frac{m}{n}\}$
Fibonacci heap	$O(m + n \log n)$
Johnson's data structure	$O(m \log \log C)$

Table 3.1: Running time of Prim's algorithm depending on the implemented data structure

Further information on the heap data structure can be found in [1].

3.5 Exercises

1. Construct the minimum spanning tree for the graph in Figure 3.8. Try to use both Kruskals algorithm and Prims algorithm.

2. **An alternative algorithm:** Consider the following algorithm for finding a MST H in a connected graph $G = (V, E, w)$: At each step, consider for

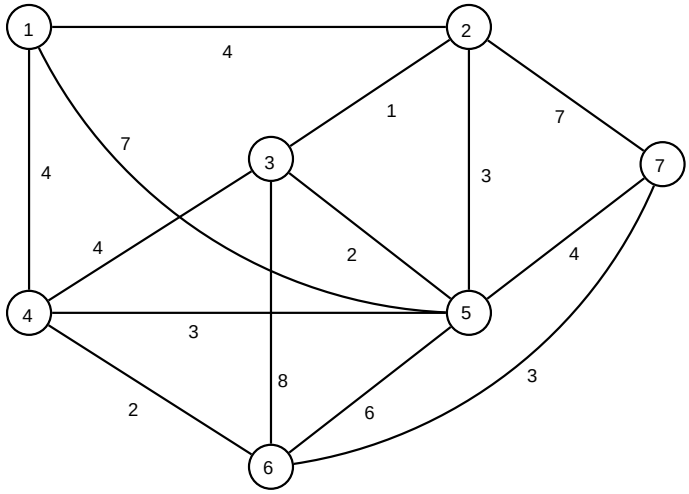


Figure 3.8: Find the minimum spanning tree for the graph using both Kruskals and Prims algorithm

each edge e whether the graph remains connected if it is removed or not. Out of the edges that can be removed without disconnecting the graph choose the one with the maximum edge cost and delete e from H. Stop if no edge can be removed without disconnecting the graph.

Show that the algorithm finds an MST of G.

Apply the algorithm to the example of exercise 1.

3. **The MiniMax Spanning Tree problem - MMST:** Suppose that rather than identifying a MST wrt. the *sum* of the edge costs, we want to minimize the *maximum cost* of any edge in the tree. The problem is called the MiniMax Spanning Tree problem.

   Prove that every MST is also an MMST. Does the converse hold?

4. **The Second-best Minimum Spanning Tree - SBMST:** Let $G = (V, E, w)$ be an undirected, connected graph with cost function w. Suppose that all edge costs are distinct, and assume that $|E| \geq |V|$.

   A **Second-best Minimum Spanning Tree** is defined as follows: Let $\mathcal{T}$ be the set of all spanning trees of graph G, and let T' be a MST of G. Then a second-best minimum spanning tree is a spanning tree $T'' \in \mathcal{T}$ such that $w(T'') = \min_{T \in \mathcal{T} - \{T'\}}\{w(T)\}$.

   (a) Let T be a MST of G. Prove that there exists edges $(i, j) \in T$ and $(k, l) \notin T$ such that $T' = T - \{(i, j)\} \cup \{(k, l)\}$ is a second-best minimum spanning tree of G.

Hints: (1) Assume that T' differs from T by more than two edges, and derive a contradiction. (2) If we take an edge $(k, l) \notin T$ and insert it in T we make a cycle. What can we say about the cost of the (k, l) edge in relation to the other edges?

(b) Give an algorithm to compute the second-best minimum spanning tree of G. What is the time complexity of the algorithm?

5. **Cycle detection for Kruskals algorithm**. Given an undirected graph. Describe an $O(m + n)$ time algorithm that detects whether there exists a cycle in the graph.

6. **The end-node problem**. *([3])*. Suppose that you are given a graph $G = (V, E)$ and weights w on the edges. In addition you are also give a particular vertex $v \in V$. You are asked to find the minimum spanning tree such that v is *not* an end-node. Explain briefly how to modifya minimum spaning tree algorithm to solve the same problem efficiently.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. The MIT Press and McGraw-Hill, 2001.

[2] Ravinda K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice Hall, 1993.

[3] George B. Dantzig, Mukund N. Thapa. *Linear Programming*. Springer, 1997.

Chapter 4

The Shortest Path Problem

One of the classical problems in Computer Science and Operations Research is the problem of finding the shortest path between two points. The problem can in a natural way be extended to the problem of finding the shortest path from one point to all other points.

There are many applications of the shortest path problem. It is used in route planning services at different webpages (in Denmark examples are `www.krak.dk` and `www.dgs.dk`) and in GPS units. It is also used in `www.rejseplanen.dk`, which generates itineraries based on public transportation in Denmark (here shortest is with respect to time).

Given a lengthed directed graph $G = (V, E)$. Each of the edges are assigned a weight (or length or cost) w, which can be positive as well as negative. The overall network is then denoted as $G = (V, E, w)$. Finally a starting point, denoted the *root* or *source* r is given. We now want to find the shortest paths from r to all other vertices in $V \setminus \{r\}$. We will later in section 8.1 look at the situation where we want to calculate the shortest path between any given pair of vertices. Let us for convenience assume that all lengths are integer.

Note that if we have a negative length cycle in our graph then it does not make sense to ask for the shortest path in the graph (at least not for all vertices). Consider Figure 4.1. This graph contains a negative length cycle ($\langle 2, 5, 3, 2 \rangle$).

Although the shortest paths from $r = 1$ to $7, 8$ and 9 are well defined it means that the shortest paths for the vertices $2, 3, 4, 5, 6$ can be made arbitrary short by using the negative length cycle several times. Therefore the shortest paths from 1 to each of the vertices are undefined.

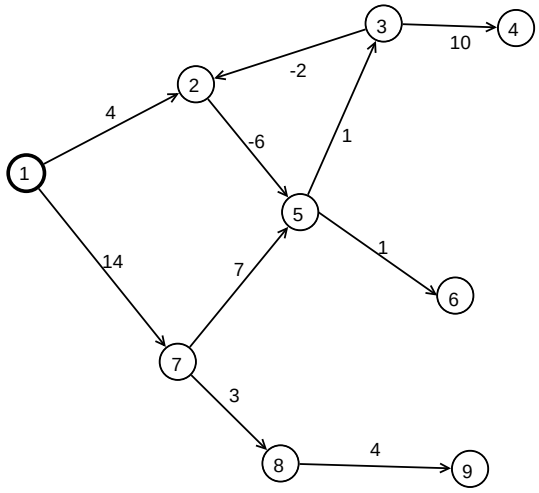


Figure 4.1: An example of a graph $G = (V, E, w)$ with root node 1 with a negative length cycle $\langle 2, 5, 3, 2 \rangle$

Like with the minimum spanning tree problem efficient algorithms exist for the problem, but before we indulge in these methods let us try to establish a mathematical model for the problem.

Let vertex r be the root. For vertex r $n-1$ paths have to leave r. For any other vertex v, the number of paths entering the vertex must be exactly 1 larger than the number of paths leaving the vertex, as the path from 1 to v ends here. Let x_e denote the *number* of paths using each edge $e \in E$. This gives the following mathematical model:

$$\min \quad \sum_{e \in E} w_e x_e \tag{4.1}$$

$$s.t. \quad \sum_{e \in V^-(r)} x_e - \sum_{e \in V^+(r)} x_e = -(n-1) \tag{4.2}$$

$$\sum_{e \in V^-(i)} x_e - \sum_{e \in V^+(i)} x_e = 1 \qquad i \in V \setminus r \tag{4.3}$$

$$x_e \in \mathcal{Z}_+ \qquad e \in E \tag{4.4}$$

Let $P = \langle v_1, v_2, \ldots, v_k \rangle$ be a path from v_1 to v_k. A subpath P_{ij} of P is then defined as $P_{ij} = \langle p_i, p_{i+1}, \ldots, p_j \rangle$. Now if P is a shortest path from v_1 to v_k then any subpath P_{ij} of P will be a shortest path from v_i to v_j. If not it would contradict the assumption that P is a shortest path.

Furthermore the subgraph defined by the shortest paths from r to all other vertices is a tree rooted at r. Therefore the problem of finding the shortest paths from r to all other nodes is sometimes refereed to as the Shortest Path Tree Problem. This means that the predecessor the each vertex is uniquely defined. Furthermore, if G is assumed to be strongly connected the tree is a spanning tree.

In order to come up with an algorithm for the shortest path problem we define *potentials* and *feasible potentials*.

Consider a vector of size n, that is, one entry for each vertex in the network $y = y_1, \ldots, y_n$. This is called a **potential**. Furthermore, if y satisfies that

1. $y_r = 0$ and
2. $y_i + w_{ij} \geq y_j$ for all $(i, j) \in E$

then y is called a **feasible** potential.

Not surprisingly, y_j can be interpreted as the length of a path from r to j. $y_i + w_{ij}$ is also the length of a path from r to j. It is a path from r to i and then finally using the arc (i, j). So if y_j is equal to the length of a shortest path from r to j then $y_j \leq y_i + w_{ij}$. If the potential is feasible for a shortest path problem then we have an optimal solution.

4.1 The Bellman-Ford Algorithm

One way to use the definition of potentials in an algorithm is to come up with an initial setting of the potentials and then to check if there exists an edge (i, j) where $y_i + w_{ij} < y_j$. If that is not the case the potential is feasible and the solution is therefore optimal. On the other hand if there is an edge (i, j) where $y_i + w_{ij} < y_j$ then potential is not feasible and the solution is therefore not optimal – there exists a path from r to j via i that is shorter than the one we currently have. This can be repaired by setting y_j equal to $y_i + w_{ij}$. Now this error is fixed and we can again check if the potential is now feasible. We

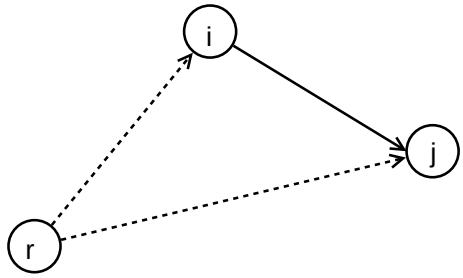


Figure 4.2: As the potential in node i reflects the possible length of a path from r to i and so for node j, the sum $y_i + w_{ij}$ also represents the length of a path from node r to j and therefore it makes sense to compare the two values. The dashed lines represents a path composed of at least one arc.

keep doing this until the potential is feasible. This is Fords algorithm for the shortest path problem as shown more formally in algorithm 4. The operation of checking and updating y_j is in many text books known as "correcting" or "relaxing" (i, j).

As the potentials always have to be an upper bound on the length of the shortest path we can use $y_r = 0$ and $y_i = +\infty$ for $i \in V \setminus \{r\}$ as initial values.

Algorithm 4: Ford's algorithm

Data: A distance matrix W for a digraph $G = (V, E)$. If the edge $(i, j) \in E$ the w_{ij} equals the distance from i to j, otherwise $w_{ij} = +\infty$

Result: Two n-vectors, y and p, containing the length of the shortest path from 1 to i resp. the predecessor vertex for i on the path for each vertex in $\{1, ..., n\}$

```
1 y_r ← 0, y_i ← ∞ for all other i
2 p_r ← 0, p_i ← NIL for all other i
3 while an edge exists (i, j) ∈ E such that y_j > y_i + w_ij do
4 |   y_j ← y_i + w_ij
5 |   p_j ← i
```

Note that no particular sequence is required. This is a problem if the instance contains a negative length circuit. It will not be possible to detect the negative length cycle and therefore the algorithm will never terminate, so for Fords algorithm we need to be aware of *negative length circuits*, as these may lead to an infinite computation. A simple solution that quickly resolves the deficiency of

Fords algorithm is to use the same sequence for the edges in each iteration. We can be even less restrictive. The extension from Ford's algorithm to Bellman-Ford's algorithm is to go through all edges and relax the necessary ones before we go through the edges again.

Algorithm 5: Bellman-Ford Algorithm

Data: A distance matrix W for a digraph $G = (V, E)$ with n vertices. If the edge $(i, j) \in E$ the w_{ij} equals the distance from i to j, otherwise $w_{ij} = +\infty$

Result: Two n-vectors, y and p, containing the length of the shortest path from r to i resp. the predecessor vertex for i

```
1 y_r ← 0; y_i ← ∞ for i ∈ V \ {r}
2 p_r ← 0; p_i ← NIL for i ∈ V \ {r}
3 k ← 0
4 while k < n and ¬(y feasible) do
5 |   k ← k + 1
6 |   foreach (i, j) ∈ E do                    /* correct (i, j) */
7 |   |   if y_j > y_i + w_ij then
8 |   |   |   y_j ← y_i + w_ij
9 |   |   |   p_j ← i
```

Note that in step 6 in the algorithm we run through all edges in the iteration.

The time complexity can be calculated fairly easy by looking at the worst case performance of the individual steps of the algorithm. The initialization (1-2) is $O(n)$. Next the outer loop in step 3 is executed $n-1$ times each time forcing the inner loop in step 4 to consider each edge once. Each individual inspection and prospective update can be executed in constance time. So step 4 takes $O(m)$ and this is done $n-1$ times so all in all we get $O(nm)$.

Proposition 4.1 *(Correctness of Bellman-Ford's Algorithm) Given a connected, directed graph $G = (V, E, w)$ with a cost function $w : E \to R$ and a root r The Bellman-Ford's algorithm produces a shortest path tree T.*

Proof: The proof is based on induction. The induction hypothesis is: In iteration k, if the shortest path P from r to i has $\leq k$ edges, then y_i is the length of P.

For the base case $k = 0$ the induction hypothesis is trivially fulfilled as $y_r = 0 =$ shortest path from r to r.

For the inductive step, we now assume that for any shortest path from r to i with less than k edges y_i is the length of the path, ie. the length of the shortest path. In the k'th iteration we will perform a correct on all edges.

Given that the shortest path from r to l consist of k edges $P_l = \langle r, v_1, v_2, \ldots, v_{k-1}, l \rangle$ due to our induction hypothesis we know the shortest path from r to $v_1, v_2, \ldots, v_{k-1}$. In the k'th iteration we perform a correct operation on all edges we specifically also correct the edge (v_{k-1}, l). This will establish the shortest path from r to l and therefore y_{v_i} = shortest path from r to l.

If all distances are non-negative, a shortest path containing at most $(n-1)$ edges exists for each $i \in V$. If negative edge lengths are present, the algorithm still works. If a *negative length circuit* exists, this can be discovered by an extra iteration in the main loop. If any y values are changed in the n'th iteration it indicates that at least one shorter path has been established. But with as the graph contains n vertices a simple path can at most contain $n-1$ edges, and therefore no changes should be made in the n'th iteration. $\triangle$

4.2 Shortest Path in Acyclic graphs

Before we continue with the general case let us make a small detour and look at the special case of an acyclic graph. In case the graph is acyclic we can solve the Shortest Path problem faster. Recall that an acyclic graph is a directed graph without (directed) cycles.

In order to be able to reduce the running time we need to sort the vertices that we will check in a specific order. A **topological sorting** of the vertices is a numbering $N : V \mapsto \{1, ..., n\}$ such that for any $(v, w) \in V \ : N(v) < N(w)$.

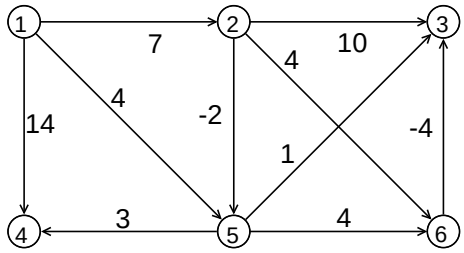


Figure 4.3: An acyclic graph

The topological sorting of the vertices of the graph in Figure 4.3 is $1, 2, 5, 4, 6, 3$.

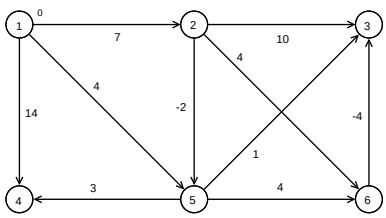
As vertex 1 is the only vertex with only outgoing edges it will consequently be the first edge in the sorted sequence of vertices. If we look at vertex 5, it is placed after vertices 1 and 2 as it has two incoming edge from those two vertices.

Algorithm 6: Shortest path for acyclic graph

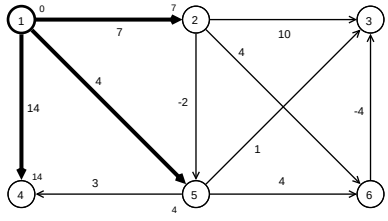
Data: A **topological sorting** of $v_1, ..., v_n$

```
1 y_1 ← 0, y_v ← ∞
2 p_1 ← 0, p_v ← Nil for all other v
3 for i ← 1 to n − 1 do
4 |   foreach j ∈ V^+(v_i) with y_j > y_{v_i} + w_{v_i j} do
5 |   |   y_j ← y_{v_i} + w_{v_i j}
6 |   |   p_j ← v_i
```

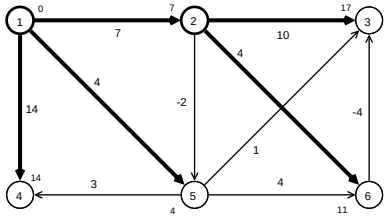
Let us assume we have done the topological sorting. In algorithm 6 each edge is considered *only once* in the main loop due to the topological sorting. Hence, the time complexity is $O(m)$.



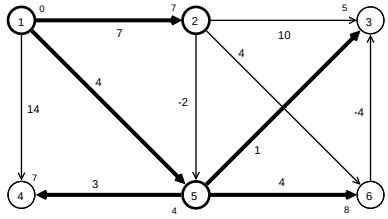
(a) Init



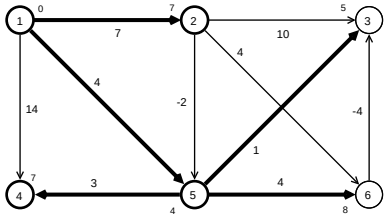
(b) Step 1



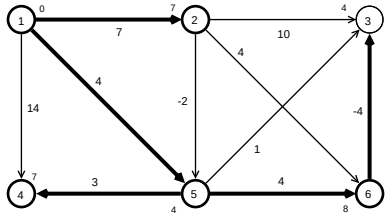
(c) Step 2



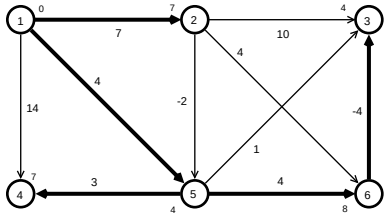
(d) Step 3



(e) Step 4



(f) Step 5



(g) Step 6

Figure 4.4: The execution of the acyclic algorithm on our sample graph.

A topological sorting of the vertices in an acyclic graph can be achieved by the algorithm below. Note that if the source vertex is not equal to the first vertex in the topological sorting it means that some vertex cannot be reached from the source vertex and will remained labeled with distance $+\infty$.

Algorithm 7: Topological sorting

Data: An acyclic digraph $G = (V, E)$
Result: A numbering N of the vertices in V so for each edge $(i, j) \in E$ it holds that $N_v < N_w$

```
1  start with all edges being unmarked
2  N_v ← 0 for all v ∈ V
3  i ← 1
4  while i ≤ n do
5  |  find v with all incoming edges marked
6  |  if no such v exists then
7  |  |  STOP
8  |  N_v ← i
9  |  i ← i + 1
10 |  mark all (v, w) ∈ E
```

Now with the time complexity of $O(m)$ the overall time complexity of finding the shortest path in an acyclic graph becomes $O(m)$.

4.3 Dijkstra's Algorithm

Now let us return to a general (lengthed) directed graph. We can improve the time complexity of the Bellman-Ford algorithm if we assume that all edge lengths are non-negative. This assumption makes sense in many situations; for example, if the lengths are road distances, travel times, or travelling costs. Even though we could still use Bellman-Ford's algorithm Dijkstra's algorithm – named after its inventor, the Dutch computer scientist Edgar Dijkstra – utilizes the fact that the edges have non-negative lengths to get a better running time.

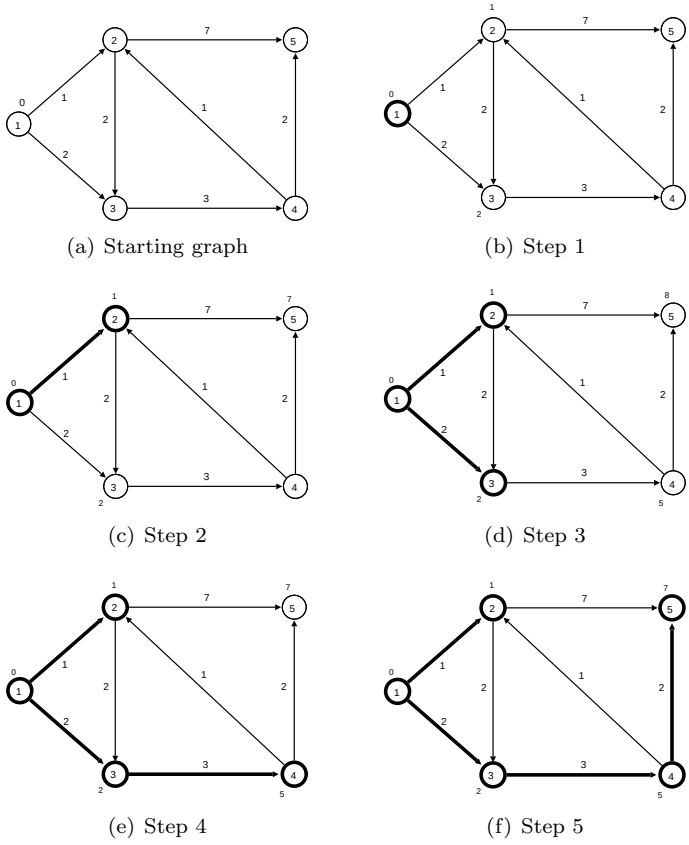
Algorithm 8: Dijkstra's algorithm

Data: A digraph $G = (V, E)$ with n vertices and m edges. A source vertex is denoted r. For each edge $(i, j) \in E$ an edge weight w_{ij} is given. **Note:** $w_{ij} \geq 0$.

Result: Two n-vectors, y and p, containing the length of the shortest path from r to i resp. the predecessor vertex for i on the path for each vertex in $\{1, ..., n\}$.

```
1  S ← {r}, U ← ∅
2  p_r ← 0, y_r ← 0
3  p_i ← NIL, y_i ← ∞ for i ∈ V \ {r}
4  while S ≠ ∅ do
5      i ← arg min_{j∈S}{y_j}
6      for j ∈ V \ U ∧ (i, j) ∈ E do
7          if y_j > y_i + w_ij then
8              y_j ← y_i + w_ij
9              p_j ← i
10     S ← S \ {i}; U ← U ∪ {i};
```

An example of the execution of Dijkstras algorithm is shown in Figure 4.5. Initially all potentials are set to $+\infty$ except the potential for the source vertex that is set to 0. First the source vertex will be selected and we relax all outgoing edges of vertex 1, that is, the edges $(1, 2)$ and $(1, 3)$. This results in updating y_2 and y_3 to 1 resp. 2. In the second step vertex 2 is selected, thereby determining edge $(1, 2)$ as part of the shortest path solution. All outgoing edges are inspected and we relax $(2, 7)$ but not $(2, 3)$ as it is more expensive to get to vertex 3 via vertex 2. In step 3 vertex 3 is selected making edge $(1, 3)$ path of the solution. In step 4 vertex 4 is chosen. This adds $(3, 4)$ to the shortest path tree and finally in step 5 we add vertex 5 and thereby $(4, 5)$ to the solution.



(a) Starting graph (b) Step 1

(c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5

Figure 4.5: The execution of Dijkstra's algorithm on a small example directed graph. Vertex 1 is source vertex. A vertex without a "distance" symbolizes a distance of ∞

Note that this algorithm is very similar to Prim's algorithm for the MST (see section 3.3). The only difference to Prim's algorithm is in fact the update step in the inner loop, and this step takes – like in the MST algorithm – $O(1)$ time. Hence the complexity of the algorithm is $O(n^2)$ if a list representation of the y vector is used, and a complexity of $O(mlogn)$ can be obtained if the `heap` data structure is used instead.

Theorem 4.2 *(Correctness of Dijkstras Algorithm) Given a connected, directed graph $G = (V, E, w)$ with length function $w : E \rightarrow R_+$ and a root node r. The Dijkstra algorithm will produce a shortest path tree T rooted at r.*

Proof: The proof is made by induction. The induction hypothesis is that before an iteration it holds that for each vertex i in U, the shortest path from the source vertex r to i has been found and is of length y_i, and for each vertex i not in U, y_i is the shortest path from r to i with all vertices except i belonging to U.

As $U = \emptyset$ this is obviously true initially. Also after the first iteration where $U = \{r\}$ the assumption is true. (check for yourself)

Let v be the element with the smallest y value selected in the inner loop of iteration k, that is, $y_v \leq y_u$ for all $u \in S$. Now y_v is the length of a path Q from r to v passing only through vertices in U (see Figure 4.6).

Suppose that this is not the shortest path from r to v – then another path R from r to v is shorter.

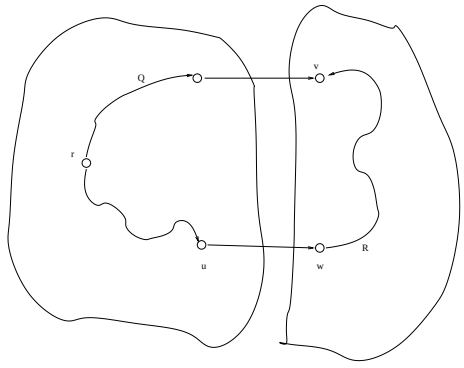


Figure 4.6: Proof of the correctness of the Dijkstra algorithm

Consider path R. R starts in r, which is in U. Since v is not in P, R has at least one edge from a vertex in P to a vertex not in P. Let (u, w) be the first

edge of this type. The vertex w is a candidate for vertex choice in the inner loop in the current iteration, but was discarded as v was picked. Hence $y_w \geq y_v$. All edge lengths are assumed to be non-negative, therefore the length of the part of R from w to v is non-negative, and hence the total length of R is at least the length of Q. This is a contradiction – hence Q is a shortest path from r to v. Furthermore, the update step in the inner loop ensures that after the current iteration it again holds for u not in P (which is now the "old" P augmented with v) that y_u is the shortest path from from r to u with all vertices except u belonging to P. $\triangle$

4.4 Relation to Duality Theory

From our approach solving the Shortest Path Problem using potentials it is not evident that there is a connection to our mathematical model (4.1) - (4.4). It would seem like the model and our solution approaches are independent ways of looking at the problem. In fact, they have a great deal in common, and in this section we will elaborate more on that.

A classical result (dating back to the mid 50's) is that the vertex-edge incidence matrix of a directed graph is *totally unimodular*. This means that the determinant of each square submatrix is equal to $0, 1$ or -1. Since right-hand sides are integers it implies that all positive variables in any basic feasible solution to such problems automatically take integer values.

This means that the integer requirement in our model (4.4) can be relaxed to

$$x_e \geq 0 \quad e \in E. \tag{4.5}$$

Now our shortest path problem is actually an LP problem.

Let us now construct the dual. Let y_i be the dual variable for constraint i. The dual to the shortest path problem then becomes:

$$\begin{aligned} \max \quad & -(n-1)y_r + \textstyle\sum_{i \neq r} y_i && && (4.6)\\ \text{s.t.} \quad & y_j - y_i \leq w_{ij} && (i,j) \in E && (4.7)\\ & y_i \text{ free} && && (4.8) \end{aligned}$$

Let us look at the primal and dual problem of the instance in Figure 4.7 with $r = 1$. The dual variables $y_1, y_2, \ldots, y_5$ are associated with the five vertices in the graph. The primal and dual problems are conveniently set up in Table 4.1.

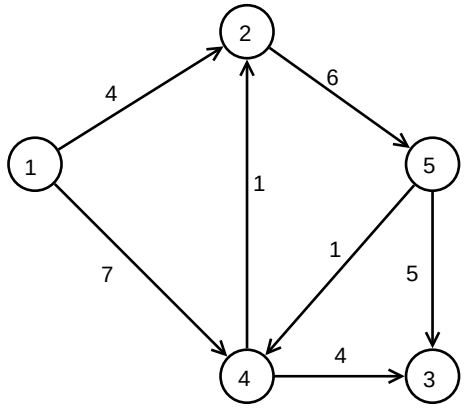


Figure 4.7: An example

Edges	$(1,2)$	$(1,4)$	$(2,5)$	$(4,2)$	$(4,3)$	$(5,3)$	$(5,4)$	rel	RHS
Variables	x_{12}	x_{14}	x_{25}	x_{42}	x_{43}	x_{53}	x_{54}		
y_1	-1	-1						$=$	-4
y_2	1		-1	1				$=$	1
y_3					1	1		$=$	1
y_4		1		-1	-1		1	$=$	1
y_5			1			-1	-1	$=$	1
relation	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$		
w_{ij}	4	7	6	1	4	5	1		

Table 4.1: Derivation of primal and dual LP problem

For this instance the dual objective can be read from the RHS column and the columns of the primal (one for each edge) constitute the constraints, so we get:

$$\begin{aligned}
\max \quad & -4y_1 + y_2 + y_3 + y_4 + y_5 && (4.9)\\
\text{s.t.} \quad & y_2 - y_1 \leq 4 && (4.10)\\
& y_4 - y_1 \leq 7 && (4.11)\\
& \vdots \; \vdots && (4.12)\\
& y_4 - y_5 \leq 1 && (4.13)\\
& y_1, y_2, \ldots, y_5 \text{ free} && (4.14)
\end{aligned}$$

It can easily be shown that the dual problem is determined up to an arbitrary constant. We can therefore fix one of the dual variables, and here we naturally choose to fix $y_r = 0$.

Central in LP theory is that all simplex-type algorithms for LP can be viewed as interplay between primal feasibility, dual feasibility and the set of complementary slackness conditions (CSC). In our case CSC will be:

$$(y_i + w_{ij} - y_j)x_{ij} = 0 \quad (i,j) \in E \tag{4.15}$$

CSC must be fullfilled by a pair of optimal solutions for the primal resp. dual problem. The predecessor variables p can be seen as representing the primal variables x in our model. If $p_j = i$ this implies that $x_{ij} > 0$ in the model.

Initially our algorithm start by setting p_i equal to NIL, basically stating that all x_{ij}'s are zero. So the CSC holds initially. Now in an iteration of any one of our methods for solving the shortest path problem we find an edge to correct, that is, we find a invalid constraint in our dual model. Then we set y_j equal to $y_i + w_{ij}$ *and* change the predecessor, which corresponds to setting x_{ij} to a non-negative value. CSC was fullfilled before and after the operations it still holds.

So in essence, what our algorithm do is to *maintain* primal feasibility and CSC, and trying to obtain dual feasibility. When dual feasibility is obtained we have primal and dual feasibility, and CSC. Now fundamental LP theory tells us that theses solutions are optimal as well.

4.5 Applications of the Shortest Path Problem

Now we will look at some interesting applications of the shortest path problem. However note that we may not be able to use Dijkstra's algorithm in solutions to some of these applications as the graphs involved may have negative edge lengths.

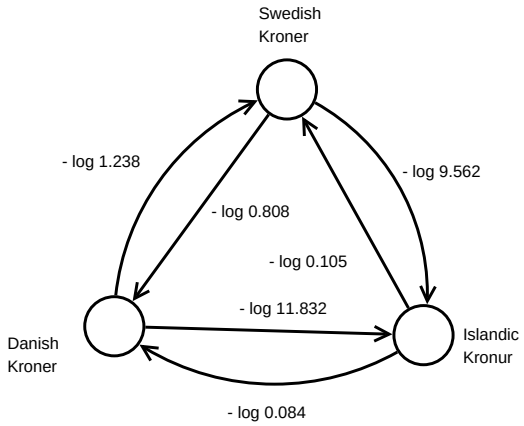


Figure 4.8: A graph to solve the currency conversion problem. Currencies are taken from `www.xe.com` at 11:17 on February 16th 2007.

The currency conversion problem

Consider a directed graph $G = (V, E)$ where each vertex denotes a currency. We want to model the possibilities of exchanging currency in order to establish the best way to go from one currency to another.

If 1 unit of a currency u can buy r_{uv} units of another currency v we model it by an edge of weight $-\log r_{uv}$ from u to v (see Figure 4.8).

If there is a path $P = \langle c_1, c_2, \ldots, c_l \rangle$ from vertex c_1 to c_l, the weight of the path is given by

$$\begin{aligned} w(P) &= \sum_{i=1}^{l-1} w_{c_i c_{i+1}} \\ &= \sum_{i=1}^{l-1} -\log r_{c_i c_{i+1}} \\ &= -\log \prod_{i=1}^{l-1} r_{c_i c_{i+1}} \end{aligned}$$

So if we convert currencies along the path P, we can convert n units of currency x to ne^{-w_P} units of currency y. Thus, the best way to convert x to y is to do it along the shortest path in this graph. Also, if there is a negative cycle in the graph, we have found a way to increase our fortune by simply making a sequence of currency exchanges.

Difference constraints

While we in general rely on the simplex method or an interior point method to solve a general linear programming problem, there are special cases that can be treated more efficient in another way.

In this section, we investigate a special case of linear programming that can be reduced to finding shortest paths from a single source. The single-source shortest path problem can then be solve using the Bellman-Ford algorithm, thereby also solving the linear programming problem.

Now sometimes we don't really care about the objective function; we just wish to find an *feasible solution*, that is, any vector x that satisfies the constraints ($Ax \leq b$), or to determine that no feasible solution exists.

In a *system of difference constraints* each row of A contains exactly one 1, one -1 and the rest 0's. Thus, each of the constraints can be written on the form $x_j - x_i \leq b_k$. An example could be:

$$\begin{array}{rcl} x_1 - x_2 & \leq & -3 \\ x_1 - x_3 & \leq & -2 \\ x_2 - x_3 & \leq & -1 \\ x_3 - x_4 & \leq & -2 \\ x_2 - x_4 & \leq & -4 \end{array}$$

This system of three constraints has a feasible solution, namely $x_1 = 0, x_2 = 3, x_3 = 5, x_4 = 7$.

Systems of difference constraints occur in many different applications. For example, the variables x_i may be times at which events are to occur. Each constraint can be viewed as stating that there must be at least/most certain amount of time (b_k) between two events.

Another feasible solution to the system of difference constraints above is $x_1 = 5, x_2 = 8, x_3 = 10, x_4 = 12$. In fact, we have:

Theorem 4.3 *Let $x = (x_1, x_2, \ldots, x_n)$ be a feasible solution to a system of difference constraints, and let d be any constant. Then $x + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$ is a feasible solution to the system of difference constraints as well.*

Proof: Look at the general for of a constraint:

$$x_j - x_i \leq b_k$$

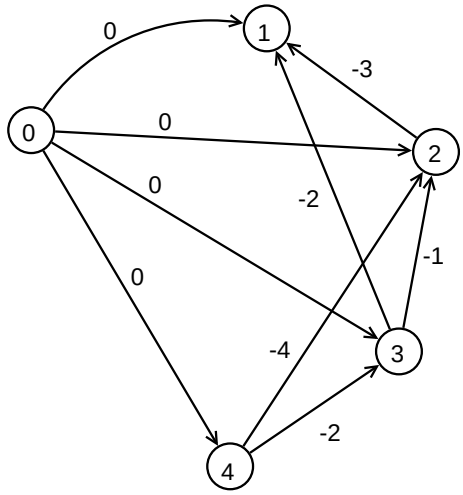


Figure 4.9: The graph representing our system of difference constraints.

If we insert $x_j + d$ and $x_i + d$ instead of x_j and x_i we get

$$(x_j + d) - (x_i + d) = x_j - x_i \leq b_k$$

So if x satisfies the constraints so does $x + d$.△

The system of constraints can be viewed from a graph theoretic angle. If A is an $m \times n$ matrix then we represent it by a graph with n vertices and m directed edges. Vertex v_i will correspond to variable x_i, and each edge corresponds to one of the m inequalities involving two unknowns. Finally, we add a vertex v_0 and edges from v_0 to every other vertex to guarantee that every other vertex is reachable. More formally we get a set of vertices

$$V = \{v_0, v_1, \ldots, v_n\}$$

and

$$E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ is a constraint}\} \cup \{(v_0, v_1), (v_0, v_2), \ldots (v_0, v_n)\}$$

Edges of the type (v_0, v_i) will get the weight 0, while the edges (v_i, v_j) get a weight of b_k. The graph of our example is shown in Figure 4.9

Theorem 4.4 *Given a system $Ax \leq b$ of difference constraints. Let G be the corresponding constraint graph constructed as described above vertex 0 being the root. If G contains no negative weight cycles, then*

$$x_1 = y_1, x_2 = y_2, \ldots, x_n = y_n$$

is a feasible solution to the system. Here y_i is the length of the shortest path from 0 to i.

Proof: Consider any edge $(i, j) \in E$. By the triangle inequality, $y_j \leq y_i + w_{ij}$, or equivalently $y_j - y_i \leq w_{ij}$. Thus letting $x_i = y_i$ and $x_j = y_j$ satisfies the difference constraint $x_j - x_i \leq w_{ij}$ that corresponds to edge (i, j). $\triangle$

Theorem 4.5 *Given a system $Ax \leq b$ of difference constraints. Let G be the corresponding constraint graph constructed as described above vertex 0 being the root. If G contains a negative weight cycle then there is no feasible solution for the system.*

Proof: Suppose the negative weight cycle is $\langle 1, 2, \ldots, k, 1\rangle$. Then

$$\begin{array}{rcl} x_2 - x_1 & \leq & w_{12} \\ x_3 - x_2 & \leq & w_{23} \\ & \vdots & \\ x_k - x_{k-1} & \leq & w_{k-1,k} \\ x_1 - x_k & \leq & w_{k1} \end{array}$$

Adding left hand sides together and right hand sides together results in $0 \leq$ weight of cycle, that is, $0 < 0$. That is obviously a contradiction. $\triangle$

A system of difference constraints with m constraints and n unknowns produces a graph with $n + 1$ nodes and $n + m$ edges. Therefore Bellman-Ford will run in $O(n + 1)(n + m)) = O(n^2 + nm)$. In [1] it is established that you can achieve a time complexity of $O(nm)$.

Using the Bellman-Ford algorithm we can now calculate the following solution $x_1 = -7, x_2 = -4, x_3 = -2, x_4 = 0$. By adding 7 to all variables we get the first solution we saw.

Swapping applications in a daily airline fleet assignment

Brug Talluris artikel som har brug for korteste vej som endnu et eksempel på brugen af korteste vej. [6] er nummer 832 i mit paper-bibliotek.

4.6 Supplementary Notes

A nice exposition to the relationship between the shortest path problem and LP theory can be found in [3]. The section on the subject in these notes are based on that text.

More information about totally unimodular (TU) matrixes can be found in [4]. Besides TU there exists other classes of matrices, Balanced and Perfect, that also have the same integer property. More information on those can be found in [5].

4.7 Exercises

1. Use Dijkstra's algorithm to find the solution the shortest paths from vertex 1 to all other vertices in the graph in Figure 4.10.

   Afterwards verify the result by constructing the shortest path integer programming model for the graph. Solve it in your MIP solver. What happens if we relax the integrality constraint and solve it using an LP solver?

   Also verify the potentials by establishing the dual problem and solve it in your LP solver.

2. Given two vertices i and j. Is the path between i and j in a minimum spanning tree necessarily a shortest path between the two vertices? Give a proof or counterexample.

3. You are employed in a consultancy company dealing mainly with routing problems. The company has been contacted by a customer, who is planning to produce a traffic information system. The goal of the system is to be able to recommend the best path between two destinations in a major city taking into account both travel distance and time. The queries posed to the system will be of the type "what is the shortest path (in kilometers) from Herlev to Kastrup, for which the transportation time does not exceed 25 minutes?" You have been assigned the responsibility of the optimization component of the system.

   As a soft start on the project, you find different books in which the usual single source shortest path problem is considered.

   (a) Apply Dijkstra's algorithm to the digraph in Figure 4.11 and show the resulting values of $y_1, \ldots, y_4$ and the corresponding shortest path tree.

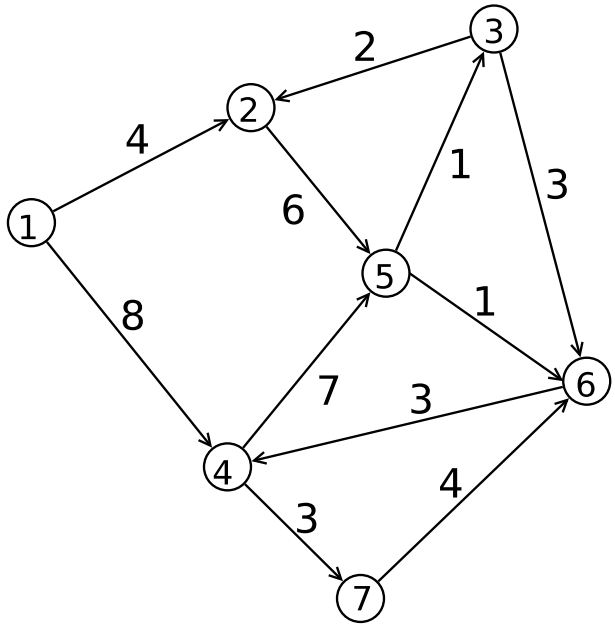


Figure 4.10: Find the shortest paths from the source vertex 1 to all other vertices

(b) The Shortest Path Problem can be formulated as a transshipment problem, in which the source has capacity 4 and each other vertex is a demand vertex with demand 1. State the problem corresponding to the digraph in Figure 4.11 and compute the vertex potentials with the shortest path tree found in question (a) as basis tree. Using the potentials, show that the distances determined in question (a) are not the correct shortest distances. Then find the correct distances (and correct shortest path tree) using i) Bellman-Ford algorithm for shortest paths; ii) the transshipment algorithm as described in chapter 7.

(c) For the sake of completeness, you decide also to look at all-to-all shortest path algorithms. You apply the algorithm of Floyd-Warshall. Show the first iteration of the algorithm for the graph of Figure 4.11.

(d) As indicated one can use Dijkstra's algorithm repeatedly to solve the all-pairs-shortest path problem – given that the graph under consideration has non-negative edge costs.

In case G does not satisfy this requirement, it is, however, possible to change the cost of the edges so all costs become non-negative *and* the shortest paths with respect to the new edge costs are the same as

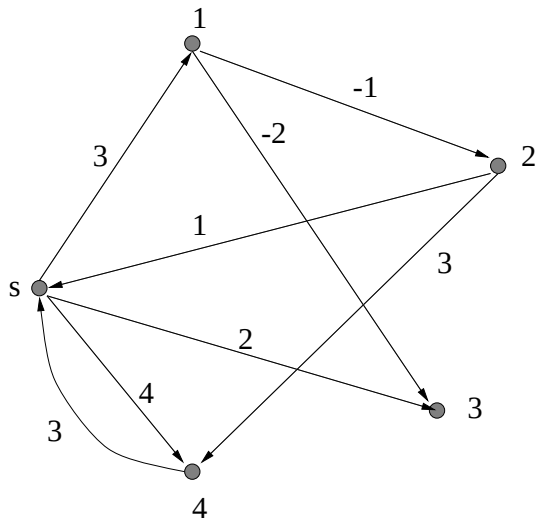


Figure 4.11: Digraph with source vertex s

those for the old edge costs (although the actual lengths of the paths change). The method is the following:

An artificial vertex K is added to G, and for each vertex v in G, an edge of cost 0 from K to v is added, cf. Figure 4.12, which illustrates the construction for the digraph of Figure 4.11.

The shortest path from K to each vertex $i \in V$ is now calculated – this is denoted π_i. The new edge lengths w'_{ij} are now defined by $w'_{ij} = w_{ij} + \pi_i - \pi_j$.

Which shortest path algorithm would you recommend for finding the shortest paths from K to the other vertices, and why? Show the application of this algorithm on the digraph of Figure 4.12.

(e) Explain why it holds that w'_{ij} is non-negative for all $i, j \in V$, such that Dijkstra's algorithm may now be applied to G with the modified edge costs.

Present an argument showing that for all $s, t \in V$, the shortest s-t path with respect to the original edge costs w_{ij} , $v, w \in V$ is also the shortest s-t path with respect to the modified edge costs w'_{ij}.

(f) You are now in possession of two all-to-all shortest path algorithms: The Floyd-Warshall algorithm, and the repeated application of Dijkstra's algorithm once per vertex of the given graph. What is the computational complexity for each of the methods? For which graphs would you recommend Floyd-Warshall's algorithm, and for which would you use |V| applications of Dijkstra's algorithm?

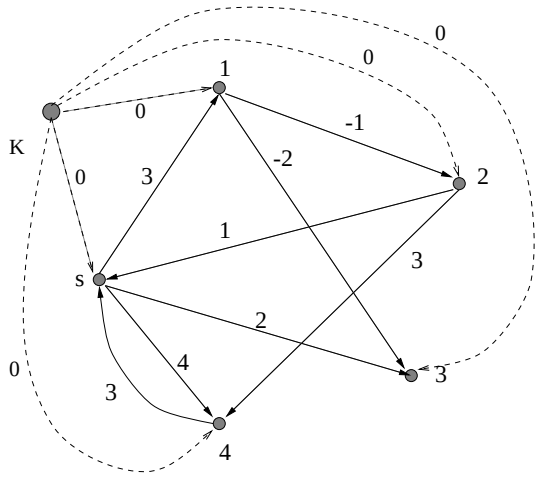


Figure 4.12: Digraph with additional vertex K and with 0-length edges added.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. The MIT Press and McGraw-Hill, 2001.

[2] Ravinda K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice Hall, 1993.

[3] Jakob Krarup and Malene N. Rørbech, *LP formulations of the shortest path tree problem*, 4OR 2:259–274 (2004).

[4] Laurence A. Wolsey. *Integer Programming*. Wiley Interscience, 1998.

[5] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley Interscience, 1998.

[6] Kalyan T. Talluri, *Swapping Applications in a Daily Airline Fleet Assignment*, Transportation Science 30(3):237–248.

Chapter 5

Project Planning

The challenging task of managing large-scale projects can be supported by two Operations Research techniques called PERT (Program Evaluation and Review Technique) and CPM (Critical Path Method). These techniques can assist the project manager in breaking down the overall project into smaller parts (called activities or tasks), coordinate and plan the different activities, develop a realistic schedule, and finally monitor the progress of the project.

The methods were developed independently of each other in the late 50's and the original versions of PERT and CPM had some important differences, but are now often used interchangeably and are combined into one acronym: PERT/CPM. In fact PERT was developed in 1958 to help measure and control the progress of the Polaris Fleet Ballistic Missile program for the US Navy. Around the same time the American chemical company DuPont was using CPM to manage its construction and repair of factories.

Today project management software based on these methods are widely available in many software packages such as eg. Microsoft Projects.

PERT/CPM has been used for many different projects including:

1. Construction of a new building or road.

2. Movie productions.
3. Construction of IT-systems.
4. Ship building.
5. Research and development of a new product.

The role of the project manager in project management is one of great responsibility. In the job one needs to direct and supervise the project from the beginning to the end. Some of the roles are:

1. The project manager must define the project, reduce the project to a set of manageable tasks, obtain appropriate and necessary resources.
2. The project manager must define the final goal for the project and must motivate the workers to complete the project on time.
3. A project manager must have technical skills. This relates to financial planning, contract management, and managing innovation and problem solving within the project.
4. No project ever goes 100% as planned. It is the responsibility of the project manager to adapt to changes such as reallocation of resources, redefining activities etc.

In order to introduce the techniques we will look at an example. As project manager at *for GoodStuff Enterprises* we have the responsibility for the development of a new series of advanced intelligent toys for kids called *MasterBlaster*. Based on a preliminary idea top management has given us green light to a more thorough feasibility study. As the toy should be ready before the Christmas sale we have been asked to investigate if we can finish the project within 30 weeks.

The tasks that needs to be carried out during the project is broken down into a set of individual "atomic" tasks called *activities*. For each activity we need to know the duration of the activity and its immediate predecessors. For the MasterBlaster project the activities and their data can be seen in Table 5.1.

5.1 The Project Network

As project managers we first and foremost would like to get a visualization of the project that reveal independencies and the overall "flow" of the project. As the saying goes one picture is worth a thousand words.

Activity	Description	Immediate predecessor	Duration (weeks)
A	Product design	–	10
B	Market research	–	4
C	Production analysis	A	8
D	Product model	A	6
E	Marketing material	A	6
F	Cost analysis	C	5
G	Product testing	D	4
H	Sales training	B, E	7
I	Pricing	F, H	2
J	Project report	F, G, I	3

Table 5.1: Activities in project MasterBlaster

The first issue we will look at is how to visualize the project flow, that is, how do we visualize the connection of the different tasks in the project and their relative position in the overall plan.

Project networks consist of a number of nodes and a number of arcs. Since the first inventions of the project management methods PERT and CPM there have been two alternatives for presenting project networks:

AOA (Activity-on-arc): Each activity is represented as a an *arc*. A node is used to separate an activity from each of its immediate predecessors.

AON (Activity-on-node: Each activity is represented by a *node*. The arcs are used to show the precedence relationships.

AON have generally been regarded as considerably easier to construct than AOA. AON is also seen as easier to revise than AOA when there are changes in the network. Furthermore AON are easier to understand than AOA for inexperienced users. Let us look at the differences on a small example. In Table 5.2 we have a breakdown of a tiny project into activities (as duration is not important for visualizing the project it is omitted here).

First we build a flow network as an AON network. In an AON network nodes are equal to activities and arcs will represent precedence relations. Note that this will result in a *acyclic* network (why?). In order to start the process two "artificial" activities "St" (for start) and "Fi" (for finish) are added. St is the first activity of the project and Fi is the final activity of the project.

We start of by drawing the nodes St, A, B, C, D, and Fi. Now for each prede-

Activity	**Immediate predecessor**
A	–
B	–
C	A, B
D	B

Table 5.2: A breakdown of activities for a tiny project

cessor relation we draw an arc, that is, we draw an arc (A, C), a (B, C) arc and a (B, D) arc. As A and B do not have any predecessors we introduce an (St, A) arc and an (St, B) arc. Finally we introduce a (C, Fi) arc and a (D, Fi) arc as they do not occur as predecessors to any activity. We have now constructed an AON network representation of our small example shown in Figure 5.1

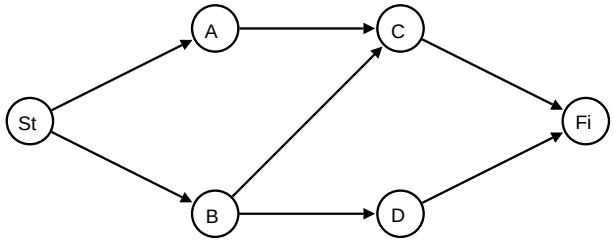


Figure 5.1: An AON network for our little example

Where the activities are represented by nodes in the AON network they are represented by arcs in an AOA network. The nodes merely describes an "state change" or a checkpoint. In the small example we need to be able to show that whereas activity C is preceded by A and B, D is only preceded by B. The only way to do this is by creating a "dummy" activity d, and we then get the network as shown in Figure 5.2

Although the AOA network is more compact than the AON network the problem with the AOA network is the use of dummy activities. The larger and more complex the project becomes the more dummy activities will have to be introduced to construct the AOA network, and these dummy activities will blur the picture making it harder to understand.

We will therefore focus on AON networks in the following as they are more intuitive and easy to construct and comprehend.

Let us construct the project network for our MasterBlaster project. We begin by making the start node (denoted St). This node represents an activity of zero

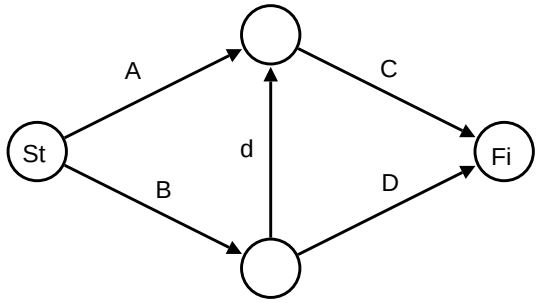


Figure 5.2: An AOA network for out little example

duration that starts the project. All nodes in the project that do not have an immediate predecessor will have the start node as their immediate predecessor. After the St node the nodes A and B, an arc from St to A, and one from St to B are generated. This is shown in Figure 5.3 where the duration of the activities are indicated next to the nodes.

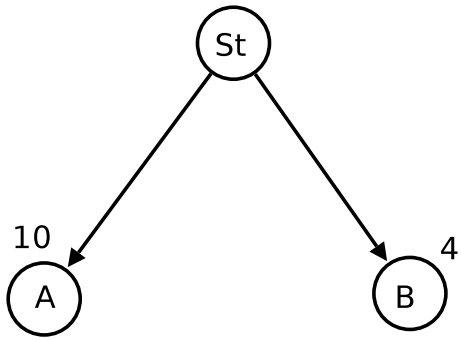


Figure 5.3: Start building the project network by making the start node and connecting nodes that does not have an immediate predecessor to the start node. In this case it is node A and B.

Then we make the nodes for activities C, D and E and making an arc from A to each of these nodes. This way we continue to build the AON network. After having made node J and connected it to activities F and I, we conclude the construction of the project network by making the finish node, Fi. All activities that are not immediate predecessor to an activity is connected to Fi. Just as the start node the finish node represents an imaginary activity of duration zero. For our MasterBlaster project the project network is shown in Figure 5.4.

In relation to our project a number of questions are quite natural:

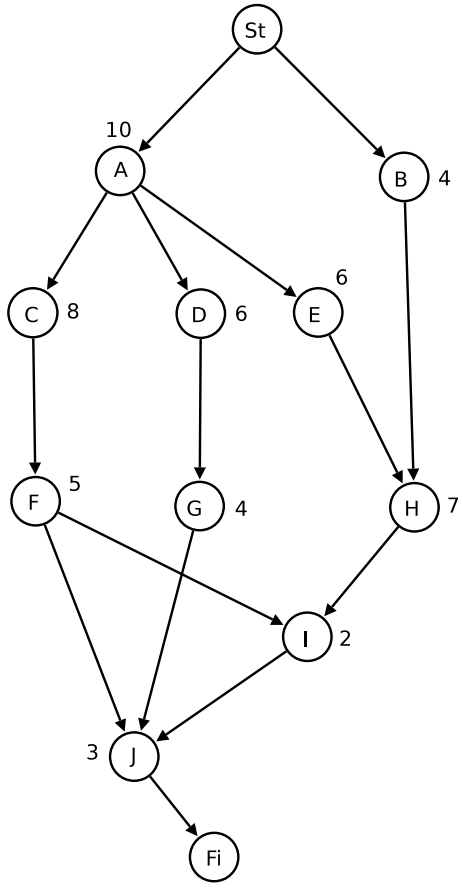


Figure 5.4: A AON network of our MasterBlaster project. The duration of the individual activities are given next to each of the nodes.

- What is the total time required to complete the project if no delays occur?
- When do the individual activities need to start and finish (at the latest) to meet this project completion time?
- When can the individual activities start and finish (at the earliest) if no delays occur?
- Which activities are critical in the sense that any delay must be avoided to prevent delaying project completion?
- For other activities, how much delay can be tolerated without delaying the project completion?

5.2 The Critical Path

A path through a project network starts at the St node, visits a number of activities, and ends at the Fi node. Such a path cannot contain a cycle. An example of a path is $\langle St, A, C, F, J, Fi \rangle$. The MasterBlaster project contains a total of 5 paths which are enumerated below:

$$\begin{array}{l} \langle St, A, C, F, J, Fi, \rangle \\ \langle St, A, C, F, I, J, Fi \rangle \\ \langle St, A, D, G, J, Fi \rangle \\ \langle St, A, E, H, I, J, Fi \rangle \\ \langle St, B, H, I, J, Fi \rangle \end{array}$$

To each of the paths we assign a *length*, which is the total duration of all activities on the path if they where executed with no breaks in between. The length of a path is calculated by adding the durations of the activities on the path together. For the first path $\langle St, A, C, F, J, Fi \rangle$ we get $10 + 8 + 5 + 3 = 26$. Below we have listed all paths and their length.

$\langle St, A, C, F, J, Fi \rangle$	26
$\langle St, A, C, F, I, J, Fi \rangle$	28
$\langle St, A, D, G, J, Fi \rangle$	23
$\langle St, A, E, H, I, J, Fi \rangle$	28
$\langle St, B, H, I, J, Fi \rangle$	16

The estimated project duration equals the *length* of the *longest path* through the project network. The length of a path identifies a minimum time required to complete the project. Therefore we cannot complete a project faster than the length of the longest path.

This longest path is also called the *critical path*. Note that there can be more than one critical path if several paths have the same maximum length. If we run a project we must be especially careful about keeping the time on the activities on a critical path. For the other activities there will be some slack, making it possible to catch up on delays without forcing a delay in the entire project.

In the case of our MasterBlaster project there are in fact two critical paths, namely: $\langle St, A, C, F, I, J, Fi \rangle$ and $\langle St, A, E, H, I, J, Fi \rangle$ with a length of 28 weeks. Thus the project can be completed in 28 weeks, two weeks less than the limit set by top management.

5.3 Finding earliest start and finish times

As good as it is to know the project duration it is necessary in order to control and manage a project to know the scheduling of the individual activities. The PERT/CPM scheduling procedure begins by determining start and finish time for the activities. They need to start and end maintaining the calculated duration of the project. No delays means that

1. actual duration equals estimated duration, and
2. each activity begins as soon as all its immediate predecessors are finished.

For each activity i in our project we define:

- ES_i as the earliest start for activity i, and
- EF_i to be Earliest finish for activity i

Let us denote the duration of activity i as d_i. Now clearly EF_i can be computed as $\mathrm{ES}_i + d_i$.

ES and EF is computed in a top down manner for the project graph. Initially we assign 0 to $\mathrm{ES}_{\mathrm{st}}$ and $\mathrm{EF}_{\mathrm{st}}$. Now we can set ES_A and ES_B to 0 as they follow immediately after the start activity. EF_A must then be 10, and EF_B 4. In the same way earliest start and finish can easily be computed for C, D and E based on an earliest start of 10 (as this is the earliest finish of their immediate predecessor, activity A). Activities F and G can be computed now that we know the earliest finish of their immediate predecessors.

When we need to compute ES and EF for activity H we need to apply the *Earliest Start Time Rule*. It states that the earliest start time of an activity is equal to the *largest* of the earliest finish times of its immediate predecessors, or

$$\mathrm{ES}_i = \max\{\mathrm{EF}_j : j \text{ is an immediate predecessor of } i\}$$

Applying this rule for activity H means that $\mathrm{ES}_H = \max\{\mathrm{EF}_B, \mathrm{EF}_E\} = \max\{4, 16\} = 16$ and then $\mathrm{EF}_H = 23$. We continue this way from top to bottom computing ES and EF for activities. The values for the project can be seen in Figure 5.6

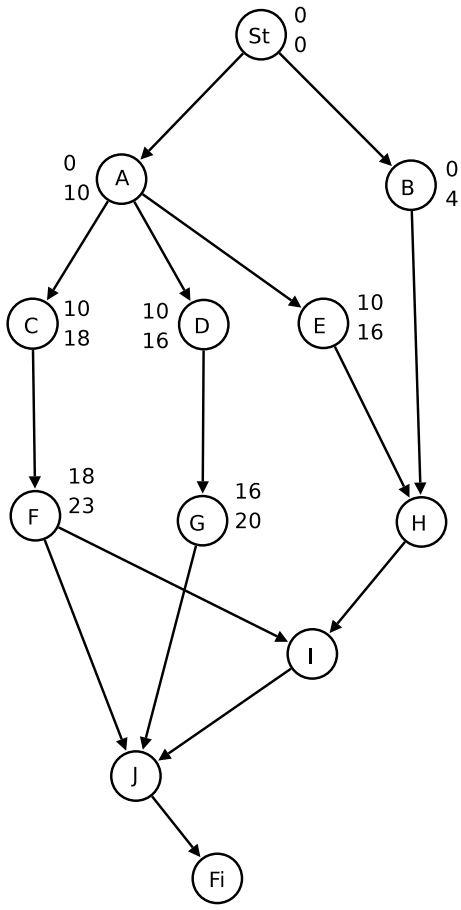


Figure 5.5: A AON network of our MasterBlaster project with earliest start and finish times.

5.4 Finding latest start and finish times

The *latest start time for an activity* i is the latest possible time that the activity can start without delaying the completion of the project. We define:

- LS_i to be the latest start time for activity i, and
- LF_i to be latest finish time for for activity i.

Obviously $\mathrm{LS}_i = \mathrm{LF}_i - d_i$.

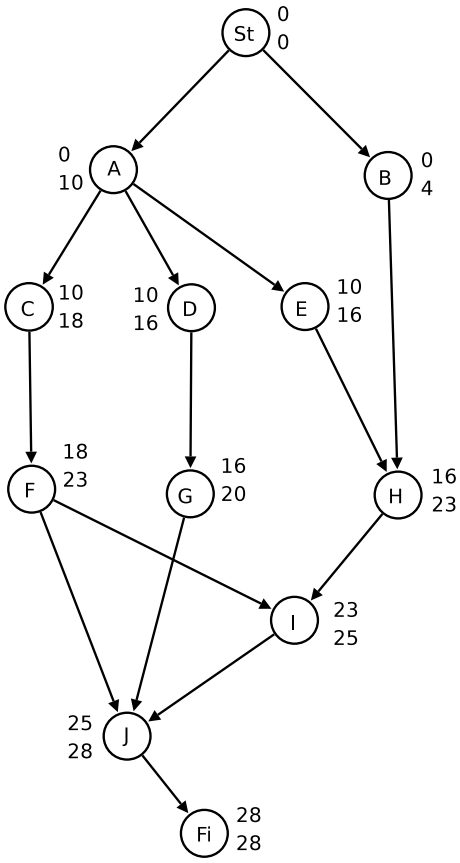


Figure 5.6: A AON network of our MasterBlaster project with earliest start and finish times.

We want to compute LS and LF for each of the activities in our problem. To do so we use a bottom up approach given that we have already established the completion time of our project. This automatically becomes then latest finish time of the finish node. Initially $\mathrm{LS}_{\mathrm{fi}}$ and $\mathrm{LF}_{\mathrm{fi}}$ is set to 28.

Now LS and LF for activity J is computed. Latest finish time must be equal to the latest start time of the successor which is the finish activity. So we get $\mathrm{LS}_J = 28$ and $\mathrm{LF}_J = 25$. Again LS and LF for activity I can easily be calculated as it has a unique successor. So we get $\mathrm{LS}_I = 25$ and $\mathrm{LF}_I = 23$. For activity F we need to take both activity I and J into consideration. In order to calculate the latest finish time we need to use the *Latest Finish Time Rule*, which states that the latest finish time of an activity is equal to the *smallest* of the latest

start times of its immediate successors, or

$$\mathrm{LF}_i = \min\{\mathrm{LS}_j : j \text{ is an immediate successor of } i\}$$

because activity i has to finish before any of its successors can start.

With this in mind we can now compute $\mathrm{LF}_F = \min\{\mathrm{LS}_I, \mathrm{LS}_J\} = \min\{23, 25\} = 23$ and LS_F will therefore be 18. In the same way the remaining latest start and finish times can be calculated. For the MasterBlaster project we get the numbers in Figure 5.7.

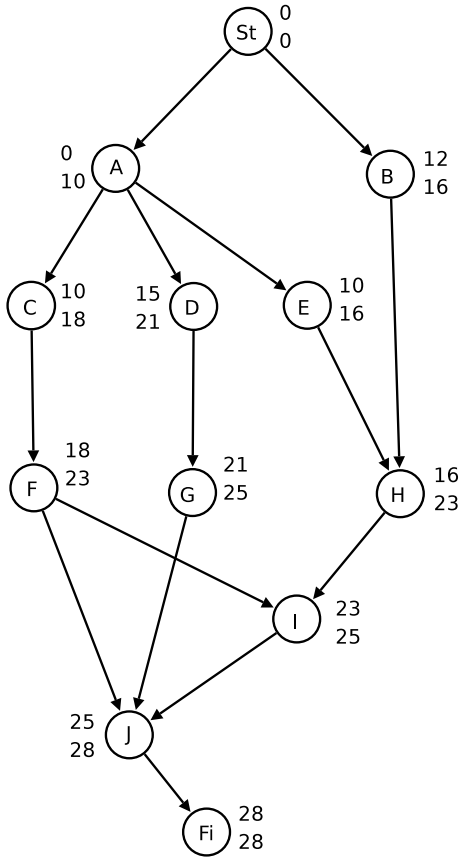


Figure 5.7: A AON network of the MasterBlaster project with latest start and finish times.

In Figure 5.8 the latest and earliest start and finish times are presented in one figure. These quite simple measures are very important in handling a project.

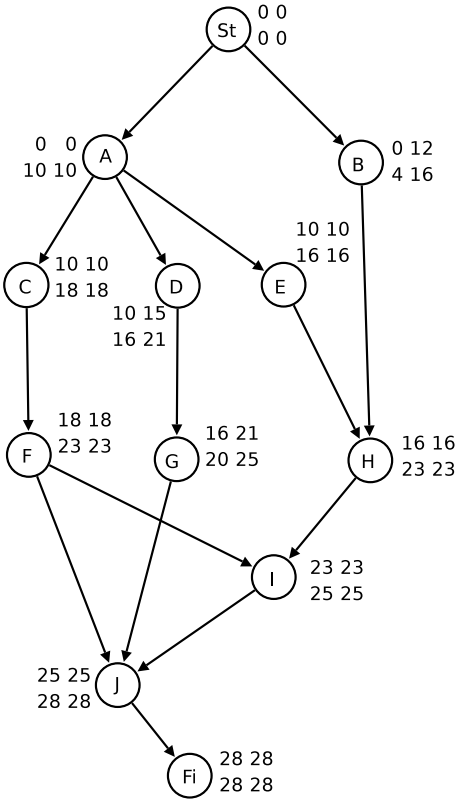


Figure 5.8: A AON network of the MasterBlaster project with earliest and latest start and finish times.

When we eg. look at activity G we can see that the project can start as early as week 16 but even if we start as late as in week 21 we can still finish the project on time. On the contrary activity E has to be managed well as there is no room for delays. It will cause a knock-on effect and thereby delay the entire project it activity E gets delayed.

We can in fact for each of the activities compute the *slack*. The slack for an activity is defined as the difference between its latest finish time and its earliest finish time. For our projects the slacks are given in Table 5.3

An activity with a positive slack has some flexibility: it can be delayed (up to a certain point) and not cause a delay of the entire project.

Slack	Activities
0	A, C, E, F, H, I, J
pos.	B, D, G

Table 5.3: The slacks for the MasterBlaster project. The main issue is to know whether the slack is zero or positive.

Note that each activity with zero slack is on a critical path through the project network. Any delay for an activity along this path will delay project completion. Therefore through the calculation of the slacks the project manager now has a clear identification of activities that needs his utmost attention.

5.5 Considering Time-Cost trade-offs

Once we have identified the critical path and the timing on the activity the next question is if we can shorten the project. This is often done in order to finish within a certain deadline. In many building projects there are incentives made by the developer to encourage the builder to finish before time.

Crashing an activity refers to taking (costly) measures to reduce the duration of an activity below its normal time. This could be to use extra manpower or using more expensive but also more powerful equipment. Crashing a project refers to crashing a number of activities in order to reduce the duration of the project below its normal time. Doing so we are targeting a specific time limit and we would like to reach this time limit in the least cost way.

In the most basic approach we assume that the crashing cost is *linear*, that is, for each activity the cost for each unit of reduction in time is constant. The situation is shown in Figure 5.9.

Top management reviews our project plan and comes up with a offer. They think 28 weeks will make the product available very late in comparison with the main competitors. They offer an incentive of 40000 Euros if the project can be finish in 25 weeks or earlier.

Quickly we evaluate every activity in our project to estimate the unit cost of crashing each activity and by how much we can crash each of the activities. The numbers for our project is shown in Figure 5.4.

Now based on these figures we can look at the problem of finding the least cost

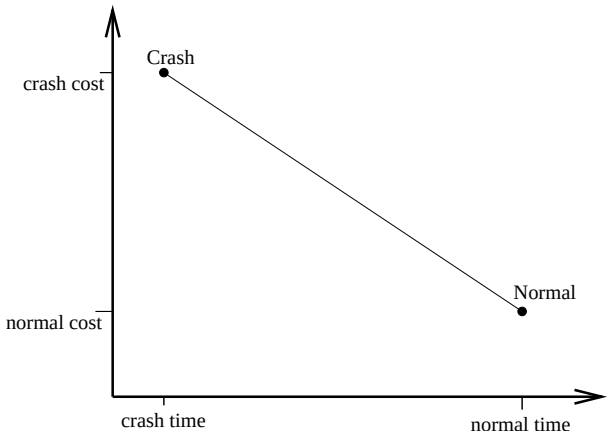


Figure 5.9: Crashing an activity

way of reducing the project duration from the current 28 weeks to 25 weeks.

One way is the *Marginal Cost Approach*. The Marginal Cost Approach was developed in conjunction with the early attempts of project management. Here we use the unit crash cost to determine a way of reducing the project duration by 1 week at a time. The easiest way to do this is by constructing a table like the one shown in Table 5.5.

As the table clearly demonstrates, a weakness of the method is that even with a few routes it can be quite difficult to keep an overview (and to fit it on one piece of paper!). And many projects will have significantly more paths than 5.

In the Marginal Cost Approach we repeatedly ask the question "What is the cheapest activity to crash in a critical path?". Initially there are two critical paths in the MasterBlaster project, which are $\langle St, A, C, F, I, J, Fi\rangle$ and $\langle St, A, E, H, I, J, Fi\rangle$. We can run through Table 5.4 and find the cheapest activity to crash which is activity H. We now crash activity H by one week and our updated table will look like Table 5.6.

As a consequence of crashing activity H by one week all paths that contains activity H are reduced by one in length.

The length of the critical path is still 28 weeks but there is not only one critical path in the project namely $\langle St, A, C, F, I, J, Fi\rangle$. So we have not yet reached the 25 weeks and therefore perform yet another iteration. We look at the critical path(s). We find the cheapest activity that can be crashed, which is activity F,

Activity	Duration (normal)	Duration (crashing)	Unit Crashing cost
A	10	7	8000
B	4	3	4000
C	8	7	9000
D	6	4	16000
E	6	3	11000
F	5	3	6000
G	4	3	7000
H	7	3	5000
I	2	2	—
J	3	2	9000

Table 5.4: Crashing activities in project MasterBlaster.

Activity to crash	Crash cost	Path length				
		ACFJ	ACFIJ	ADGJ	AEHIJ	BHIJ
		26	28	23	28	16

Table 5.5: The initial table for starting the marginal cost approach

and crash it by one week we then get the situation in Table 5.7.

As we have still not reached the required 25 weeks we continue. The final result can be seen in Table 5.8

So overall it costs us 35000 Euros to get the project length down to 25 weeks. Given that the overall reward is 40000 Euros the "surplus" is 5000 Euros. As the project must be considered more unstable after the crashing we should definitely think twice before going for the bonus. The wise project manager will probably resist the temptation as the risk is to high and the payoff to low.

Another way of determining the minimal crashing cost is to state the problem as an LP model and solve it using an LP solver.

Activity to crash	Crash cost	Path length				
		ACFJ	ACFIJ	ADGJ	AEHIJ	BHIJ
		26	28	23	28	16
H	5000	26	28	23	27	15

Table 5.6: Crashing activity H by one week and update the paths accordingly.

Activity to crash	**Crash cost**	**Path length**				
		ACFJ	ACFIJ	ADGJ	AEHIJ	BHIJ
		26	28	23	28	16
H	5000	26	28	23	27	15
F	6000	25	27	23	27	15

Table 5.7: Crashing activity D by one week for the second time and update the paths accordingly.

Activity to crash	**Crash cost**	**Path length**				
		ACFJ	ACFIJ	ADGJ	AEHIJ	BHIJ
		26	28	23	28	16
H	5000	26	28	23	27	15
F	6000	25	27	23	27	15
H	5000	25	27	23	26	14
F	6000	24	26	23	26	14
H	5000	24	26	23	25	13
A	8000	23	25	22	24	13

Table 5.8: After crashing activities H (three times), F (twice) and A (once).

We need to present the problem as an LP problem with objective function, constraints and variable definitions.

The decisions that we need to take are by how much we crash an activity and when the activity should start. So for each activity we define two variables x_i and y_i, where x_i is the reduction in the duration of the activity and y_i is the starting time of the activity.

How will our objective function look like? The objective function is to minimize the total cost of crashing activities: $\min 8000x_A + 4000x_B + \ldots + 9000x_J$.

With respect to the constraints we have to impose the constraint that the project must be finished in less than or equal to a certain number of weeks. A variable, y_{Fi}, is introduced and the project duration constraint $y_{Fi} \leq 25$ is added to the model.

Furthermore we need to add constraints that correspond to the fact that the predecessor of an activity needs to be finished before the successor can start. Since the start time of each activity is directly related to the start time and duration of each of its immediate predecessors we get:

start time of this activity $\geq$ (start time - duration) for this immediate predecessor

In other words if we look at the relationship between activity A and C we get $y_C \geq y_A + (10 - x_A)$. For each arc in the AON project network we get exactly one of these constraints.

Finally we just need to make sure that we do not crash more than actually permitted so we get a series of upper bounds on the x-variables, that is, $x_A \leq 7, x_B \leq 1, \ldots, x_J \leq 1$.

For our project planning problem we arrive at the following model:

$$
\begin{array}{ll}
\min & 8000x_A + 4000x_B + 9000x_C + 16000x_D + 11000x_E + \\
& 6000x_F + 7000x_G + 5000x_H + 9000x_J \\
s.t. & y_C \geq y_A + (10 - x_A) \\
& y_D \geq y_A + (10 - x_A) \\
& y_E \geq y_A + (10 - x_A) \\
& \quad \ldots \\
& y_{Fi} \geq y_J + (3 - x_J) \\
& y_{Fi} \leq 25 \\
& x_A \leq 3 \\
& \quad \ldots \\
& x_J \leq 1
\end{array}
$$

This model can now be entered into CPLEX or another LP solver. If we do that we get a surprising result. The LP optimium is 24000 Euros (simply crashing activity A three times) which is different from the Marginal Cost Approach. How can that be? The Marginal Cost Approach said that the total crashing cost would be 35000 Euros and now we found a solution that is 11000 Euros better.

In fact the Marginal Cost Approach is only a heuristic. Before we investigate why we only got a heuristic solution for our MasterBlaster project let us look at Figure 5.10.

There are two activities with a unit crashing cost of 30000 Euros (activity B and C) and one (activity D) with a unit crashing cost of 40000 Euros. Assume we have two critical paths from A to F one going via B the other via C. In the first step of the marginal cost analysis we will choose either B or C to crash. This will cost us 30000 Euros. If we crash B we will crash C in the next step or vice versa. Again costing us 30000 Euros. In total both paths have been reduced by

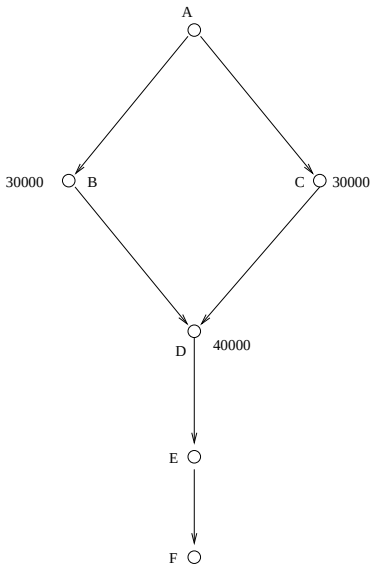


Figure 5.10: An example of where the Marginal Cost Approach makes a suboptimal choice.

1 unit each. This could also have been achieved by crashing activity D by one week, and that would only cost us 40000 Euros.

In the solution to the MasterBlaster project problem we crash activities H and F once each to reduce the critical path by one. This costs us 11000 Euros. Now crash activity A just once also reduces the length of the critical path by one, but this is only at a cost of 8000 Euros. But because we very narrowly looks at the cheapest activity a critical path to crash we miss this opportunity.

So now we have to reconsider our refusal to crash the project. Now the potential surplus is 16000 Euros.

5.6 Supplementary Notes

In the material we have discussed here all durations times were static. Of course in a real world project one thing is to estimate duration times another is to achieve them. The estimate could be wrong from the beginning or due to other reasons it could become wrong as the project is moving on. Therefore

research has been conducted into looking at stochastic elements in relation to project planning.

Further readings on project planning where we also consider stochasticity can be found in [1] that introduces the first simple approaches.

This very simple model on project planning can be extended in various ways taking on additional constraints on common resources etc. For further reading see [2].

5.7 Exercises

1. You are in charge of organizing a training seminar for the OR department in your company. Remembering the project management tool in OR you have come up with the following list of activities as in Table 5.9.

Activity	Description	Immediate predecessor	Duration (weeks)
A	Select location	–	1
B	Obtain keynote speaker	–	1
C	Obtain other speakers	B	3
D	Coordinate travel for keynote speaker	A, B	2
E	Coordinate travel for other speakers	A, C	3
F	Arrange dinners	A	2
G	Negotiate hotel rates	A	1
H	Prepare training booklet	C, G	3
I	Distribute training booklet	H	1
J	Take reservations	I	3
K	Prepare handouts from speaker	C, F	4

Table 5.9: Activities in planning the training seminar.

   (a) Find all paths and path lengths through the project network. Determine the critical path(s).

   (b) Find earliest time, latest time, and slack for each of the activities. Use this information to determine which of the paths is a critical path?

   (c) As the proposed date for the training seminar is being moved the training seminar needs to be prepare in less time. Activities with a

duration of 1 week cannot be crashed any more, but those activities that takes 2 or 3 weeks can be crashed by one week. Activity K can be crashed by 2 weeks. Given a unit crashing cost of 3000 Euros for the activities D and H, 5000 Euros for the activities J and K and 6000 Euros for activities E and F. Finally crashing activity C costs 7000 Euros. What does it cost to shorten the project by one week? and how much to shorten it by two weeks?

2. You are given the following information about a project consisting of seven activities (see Table 5.10).

Activity	**Immediate predecessor**	**Duration (weeks)**
A	–	5
B	–	2
C	B	2
D	A, C	4
E	A	6
F	D, E	3
G	D, F	5

Table 5.10: Activities for the design of a project network.

   (a) Construct the project network for this project.

   (b) Find earliest time, latest time, and slack for each of the activities. Use this information to determine which of the paths is a critical path?

   (c) If all other activities take the estimated amount of time, what is the maximum duration of activity D without delaying the completion of the project?

3. Subsequently you are put in charge of the rather large project of implementing the intra-net and the corresponding organizational changes in the company. You immediately remember something about project management from you engineering studies. To get into the tools and the way of thinking, you decide to solve the following test case using PERT/CPM:

   The available data is presented in Table 5.11. Find the duration of the project if all activities are completed according to their normal-times.

   In the LP model for finding the cheapest way to shorten a course, there is a variable x_i for each activity i in the project. There are also two constrains, "$x_i \geq 0$" and "$x_i \leq D_i - d_i$". Denote the dual variables of the latter g_i. Argue that if $d_i < D_i$, then either g_i is equal to 0 or $x_i = D_i - d_i$ in any optimal solution.

Activity	Imm. pred.	Normal-time (D_i)	Crash-time (d_j)	Crash cost
e_1		5	1	4
e_2		3	1	4
e_3	e_1	4	2	1
e_4	e_1, e_2	6	1	3
e_5	e_2	6	5	0
e_6	e_3, e_4	4	4	0

Table 5.11: Data for the activities of the test case.

What is the additional cost of shortening the project time for the test case to 13?

4. The linear relationship on the crashing cost is vital for the results. After a thorough analysis a project manager has come to a situation where one of his activities does not exhibit the linear behavior. After further analysis it has been shown that the crashing costs can actually be modeled by a piecewise linear function, in this simple case with only one "break point".

   So the situation for the given activity can be illustrated like in Figure 5.11.

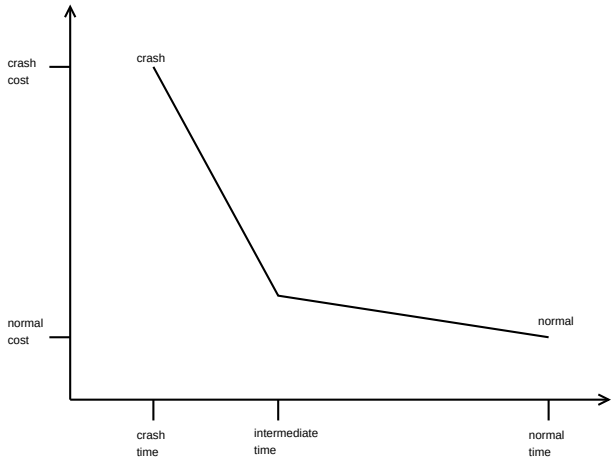


Figure 5.11: Crashing an activity

   You may assume that the slopes are as depicted. But can we deal with this situation in our LP modeling approach? If yes, how. If no, why not.

   Same question if you do not know anything about the slopes beforehand.

5. Let us turn back to our MasterBlaster project. Let us assume that the incentive scheme set forward by the top management instead of giving a fixed date and a bonus specified a daily bonus. Suppose that we will get

9500 Euros for each week we reduce the project length. Describe how this can be solved in our LP model and what the solution in the MasterBlaster project will be.

Construct the project crashing curve that presents the relationship between project length and the cost of crashing until the project can no longer be crashed.

Bibliography

[1] Hillier and Lieberman. *Introduction to Operations Research*. McGraw Hill, 2001.

[2] A. B. Badiru and P. S. Pulat. *Comprehensive Project Management: Integrating Optimization Models, Management Principles, and Computers*. Prentice-Hall, 1995.

Chapter 6

The Max Flow Problem

Let us assume we want to pump as much water from point A to B. In order to get the water from A to B we have a system of waterpipes with connections and pipelines. For each pipeline we furthermore have a capacity identifying the maximum throughput through the pipeline. In order to find how much can be pumped in total and through which pipes we need to solve the maximum flow problem.

In the maximum flow problem (for short the max flow problem) a directed graph $G = (V, E)$ is given. Each edge e has a *capacity* $u_e \in R_+$, therefore we define the graph as $G = (V, E, u)$. Furthermore two "special" vertices r and s are given; these are called resp. the *source* and the *sink*. The objective is now to determine how much flow we can get from the source to the sink. We may imagine the graph as a road network and we want to know how many cars we can get through this particular road network. An alternative application as described above the network is a network of water pipes and we now want to determine the maximal throughput in the system.

In the max flow problem we determine 1) the maximal throughput and 2) how we can achieve that through the system. In order to specify how we want to direct the flow in a solution we first need formally to define what a flow actually is:

Definition 6.1 A *flow* x in G is a function $x : E \to R_+$ satisfying:

$$\forall i \in V \setminus \{r, s\} \; : \; \sum_{(j,i)\in E} x_{ji} - \sum_{(i,j)\in E} x_{ij} = 0$$

$$\forall (i, j) \in E : 0 \leq x_{ij} \leq u_{ij}$$

Given the definition of a flow we can define $f_x(i) = \sum_{(j,i)\in E} x_{ji} - \sum_{(j,i)\in E} x_{ij}$. So $f_x(i)$ is for a given flow and a give node the difference in inflow and outflow of that node where a surplus is counted positive. $f_x(i)$ called the *net flow into* i or the *excess for* x *in* i. Specifically $f_x(s)$ is called the *value* of the flow x.

We can now state the max flow problem as an integer programming problem. The variables in the problem are the flow variables x_{ij}.

$$\begin{array}{lll} \max & f_x(s) & \\ \text{s.t.} & f_x(i) = 0 & i \in V \setminus \{r, s\} \\ & 0 \leq x_e \leq u_e & e \in E \\ & x_e \in \mathcal{Z}_+ & e \in E \end{array}$$

Notice that we restrict the flow to being integer. The reason for this will become apparent as the solution methods for the max flow problem are presented.

Related to the definition of a flow is the definition of a *cut*. There is a very tight relationship between cuts and flows that will also become apparent later. Let us consider $R \subset V$. $\delta(R)$ is the set of edges incident from a vertex in R to a vertex in $\overline{R}$. $\delta(R)$ is also called the *cut generated by* R:

$$\delta(R) = \{(i, j) \in E \mid i \in R, j \in \overline{R}\}$$

Note the difference to the cut we defined for an undirected graph in chapter 3. In a directed graph the edges in the cut are only the edges going from R to $\overline{R}$ and not those in the opposite direction.

So in an undirected graph we have $\delta(R) = \delta(\overline{R})$, but this does not hold for the definition of a cut in directed graphs. As the cut is uniquely defined by the vertex-set R we might simply call R the cut. A cut where $r \in R$ and $s \notin R$ is called an r, s*-cut*. Furthermore we define the *capacity* of the cut R as:

$$u(\delta(R)) = \sum_{(i,j)\in E, i\in R, j\in\overline{R}} u_{ij}$$

that is, the capacity of a cut is the sum of capacities of all edges in the cut.

Theorem 6.2 *For any r, s-cut $\delta(R)$ and any flow x we have: $x(\delta(R)) - x(\delta(\bar{R})) = f_x(s)$.*

Proof: Let $\delta(R)$ be an (r, s)-cut and f a flow. Now we know that $f_x(i) = 0$ for all nodes $i \in \overline{R} \setminus \{s\}$. Trivially we have $f_x(s) = f_x(s)$. If we now add these equations together we get $\sum_{i \in \overline{R} \setminus \{s\}} f_x(i) + f_x(s) = f_x(s)$ that is

$$\sum_{i \in \overline{R}} f_x(i) \quad = \quad f_x(s). \tag{6.1}$$

Consider the contribution of the different edges to the right hand side of the equation (the four different cases are illustrated in Figure 6.1).

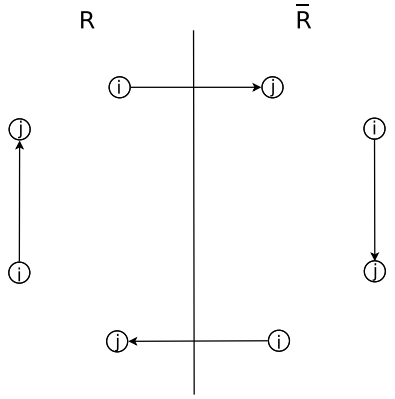


Figure 6.1: Illustration of the four different cases for accessing the contribution to a flow based on the nodes relationship to R and $\overline{R}$.

1. $(i, j) \in E, i, j \in R$: x_{ij} occurs in none of the equations added, so it does not occur in (6.1).
2. $(i, j) \in E, i, j \in \overline{R}$: x_{ij} occurs in the equation for i with a coefficient of -1 and a $+1$ in the equation for j so in (6.1) it will result in a coefficient of 0.
3. $(i, j) \in E, i \in R, j \in \overline{R}$: x_{ij} occurs in (6.1) with a coefficient of 1 and therefore the sum will have a contribution of $+1$ from this edge.

4. $(i,j) \in E, i \in \overline{R}, j \in R$: x_{ij} occurs in (6.1) with a coefficient of -1 and therefore the sum will have a contribution of -1 from this edge.

So the left hand side is equal to $x(\delta(R)) - x(\delta(\overline{R}))$. $\triangle$

Corollary 6.3 *For any feasible flow x and any r,s-cut $\delta(R)$, we have: $f_x(s) \leq u(\delta(R))$.*

Proof: Given a feasible flow x and a r,s-cut $\delta(R)$. From theorem 6.2 we have

$$x(\delta(R)) - x(\delta(\overline{R})) = f_x(s)$$

Since all values are positive (or zero) this gives us

$$f_x(s) \leq x(\delta(R)).$$

Furthermore we cannot have a flow larger than the capacity, that is, $x(\delta(R)) \leq u(\delta(R))$. So we have $f_x(s) \leq x(\delta(R)) \leq u(\delta(R))$. $\triangle$

The importance of the corollary is that it gives an upper bound on any flow value and therefore also the maximum flow value. In words it states – quite intuitively – that the capacity of the minimum cut is an upper bound on the value of the maximum flow. So if we can find a cut and a flow such that the value of the flow equals the capacity of the cut then the flow is maximum, the cut is a "minimum capacity" cut and the problem is solved. The famous Max Flow - Min Cut theorem states that this can always be achieved.

The proof of the Max Flow - Min Cut Theorem is closely connected to the first algorithm to be presented for solving the max flow problem. So we will already now define some terminology for the solution methods.

Definition 6.4 A path is called *x-incrementing* (or just incrementing) if for every forward arc e $x_e < u_e$ and for every backward arc $x_e > 0$. Furthermore a x-incrementing path from r to s is called *x-augmenting* (or just augmenting).

Theorem 6.5 *Max Flow - Min Cut Theorem. Given a network $G = (V,E,u)$ and a current feasible flow x. The following 3 statements are equivalent:*

1. *x is a maximum flow in G.*

2. *A flow augmenting path does not exist (an r-s-dipath in G_x).*

3. *An r, s-cut R exists with capacity equal to the value of x, ie. $u(R) = f_x(s)$.*

Proof:

1) $\Rightarrow$ 2) Given a maximum flow x we want to prove that there does not exist a flow augmenting path.

Assume there does exist a flow augmenting path, and seek a contradiction. If there exists a flow augmenting path we can increase the flow along the augmenting path by at least 1. This means that the new flow is at least one larger than the existing one. But this is in contradiction with the assumption that the flow was maximum. So no flow augmenting path can exist when the flow is maximum.

2) $\Rightarrow$ 3) First define R as $R = \{i \in V : \exists$ an x-incrementing path from r to $i\}$. Based on this definition we derive:

1. $r \in R$
2. $s \notin R$

Therefore R defines an r, s-cut.

Based on the definition of R we derive $(i, j) \in \delta(R)$ $x_{ij} = u_{ij}$ and for $(i, j) \in \delta(\overline{R})$ we get $x_{ij} = 0$. As $x(\delta(R)) - x(\delta(\overline{R})) = f_x(s)$ we insert $x(\delta(R)) = u(\delta(R))$ and $x(\delta(\overline{R})) = 0$ from above, and get: $u(\delta(R)) = f_x(s)$.

3) $\Rightarrow$ 1) We assume that there exists an r, s-cut R with $f_x(s) = u(\delta(R))$. From the corollary we have $f_x(s) \leq u(\delta(R))$. The corollary defines an upper bound equal to the current feasible flow, which therefore must be maximum.

This proves that the three propositions are equivalent.$\triangle$

6.1 The Augmenting Path Method

Given the Max Flow - Min Cut theorem a basic idea that leads to our first solution approach for the problem is immediate. The augmenting path method. This is the classical max flow algorithm by Ford and Fulkerson from the mid-50's.

The basic idea is to start with a feasible flow. As a feasible flow we can always use $x = 0$ but in many situations it is possible to construct a better initial flow by a simple inspection of the graph.

Based on the current flow we then try to find an augmenting path in the network, that is, a path from r to s that can accommodate an increase in flow. If that is possible we introduce the increased flow. This updates the previous flow by adding the new flow that we have managed to get from r to s. Now we again look for a new augmenting path. When we cannot find an augmenting path the maximum flow is found (due to Theorem 6.5) and the algorithm terminates.

The critical part of this approach is to have a systematic way of checking for augmenting paths.

An important graph when discussing the max flow problem is the *residual graph*. In some textbooks you might see the term "auxiliary graph" instead of residual graph. The *residual graph* identifies based on the current flow where *excess* can be send to.

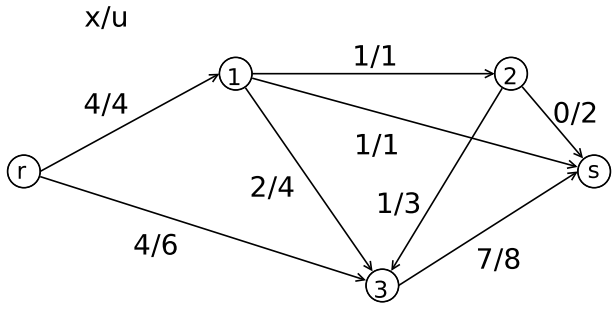
Definition 6.6 Given a feasible flow x the *residual graph* G_x for G wrt. x is defined by:

$$\begin{array}{lcl} V_x = V(G_x) & = & V \\ E_x = E(G_x) & = & \{(i,j) : (i,j) \in E \wedge x_{ij} < u_{ij}\} \cup \\ & & \{(j,i) : (i,j) \in E \wedge x_{ij} > 0\} \end{array}$$

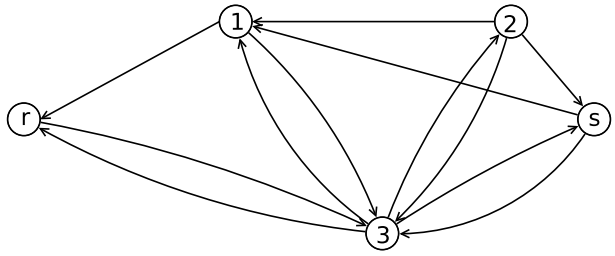
The residual graph can be built from the original graph and the current flow arc by arc. This is illustrated in Figure 6.2.

If we look at the arc $(r, 1)$ there is a flow of four and a capacity of four. Therefore we cannot push more flow from r to 1 along that arc as it is already saturated. On the other hand we can lower the flow from r to 1 along the $(r, 1)$ arc. This is represented by having an arc in the opposite direction from 1 to r. Therefore the residual graph contains an $(1, r)$ arc. As another example consider the $(1, 3)$ arc. There is room for more flow from 1 to 3, so we can push excess flow from 1 in the direction of 3 using the $(1, 3)$. Therefore the arc $(1, 3)$ is in the residual graph. But $(3, 1)$ is also included in the residual graph as excess flow from node 3 can be pushed to 1 by reducing flow along the $(1, 3)$ arc.

The nice property in a residual graph is that an augmenting path consists of forward arcs only. This is not necessarily the case in the original graph. If we return to Figure 6.2 the path $\langle r, 3, 2, s\rangle$ is in fact an augmenting path. We can push one more unit along this path and thereby increase the value of the flow by one. But the path uses the $(2, 3)$ arc in the "reverse" direction as we decrease



(a) The graph G and a flow x



(b) The residual graph G_x

Figure 6.2: The residual graph reflects the possibilities to push excess flow from from a node

the inflow from node 3 and instead route it to the sink. In the residual graph the path is simply made up by forward arcs.

Now we can describe the augmenting path algorithm where we use the residual graph in each iteration to check if there exists an augmenting path or not. The algorithm is presented in algorithm 9.

Algorithm 9: The Augmenting Path algorithm

Data: A directed graph $G = (V, E, u)$, a source r and sink s
Result: A flow x of maximum value

```
1 x_ij ← 0 for all (i, j) ∈ E
2 repeat
3 |   construct G_x
4 |   find an augmenting path in G_x
5 |   if s is reached then
6 |   |   determine max amount x' of flow augmentation
7 |   |   augment x along the augmenting path by x'
8 until s is not reachable from r in G_x
```

If s is reached then the augmenting path is given by the p. Otherwise no augmenting path exists.

Consider the following example. In order not to start with $x = 0$ as the initial flow we try to establish a better starting point. We start by pushing 4 units of flow along $\langle r, 1, 2, s\rangle$ and one unit of flow along $\langle r, 4, s\rangle$. Furthermore we push two units of flow along the path $\langle r, 1, 2, 3, 4, s\rangle$ and arrive at the situation shown in Figure 6.3 and the corresponding residual graph. Based on the residual graph we can identify an augmenting path $\langle r, 1, s\rangle$. The capacities on the path are 2 and 1. Hence, we can at most push one unit of flow through the path. This is done in the following figure and its corresponding residual graph is also shown.

Now it becomes more tricky to identify an augmenting path. From r we can proceed to 1 but from this node there is no outgoing arc. Instead we look at the arc $(r, 3)$. From 3 we can continue to 2 and then to 4 and then can get to s. Having identified the augmenting path $\langle r, 3, 2, 4, s\rangle$ we update the flow with 2 along the path and get to the situation shown in 6.3 (c).

In the residual graph is not possible to construct incrementing paths beyond nodes 1 and 3. Consequently, there is not an augmenting path in the graph and consequently we have found a maximum flow. The value of the flow is 11 and the minimum cut is identified by $R = \{r, 1, 3\}$. Note that forward edges are

saturated, that is, flow equal to capacity, and backward edges are empty in the cut.

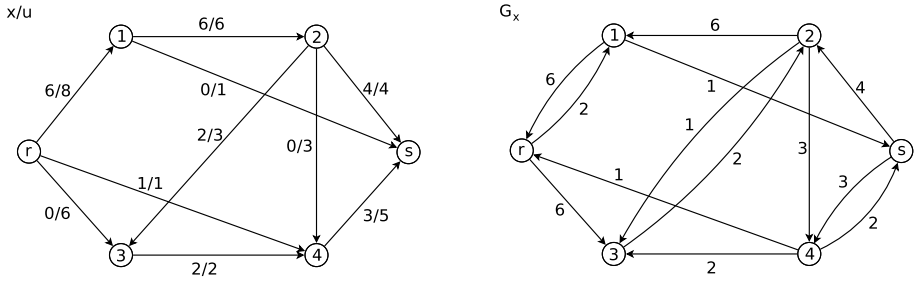
In the algorithm we need to determine the maximum amount that we can increase the flow along the augmenting path we have found. When increasing the flow we need to be aware that in arcs that are forward arcs in the original graph we cannot exceed the capacity. For arcs that are backward arcs in the original graph the flow will be reduced along the augmenting path, and we cannot get a negative flow. As a result we get the updated (augmented) flow x':

- for $(i,j) \in P : (i,j) \in E : x'_{ij} \leftarrow x_{ij} + u_s^{\max}$
- for $(i,j) \in P : (j,i) \in E : x'_{ji} \leftarrow x_{ji} - u_s^{\max}$
- for all other $(i,j) \in E : x'_{ij} \leftarrow x_{ij}$

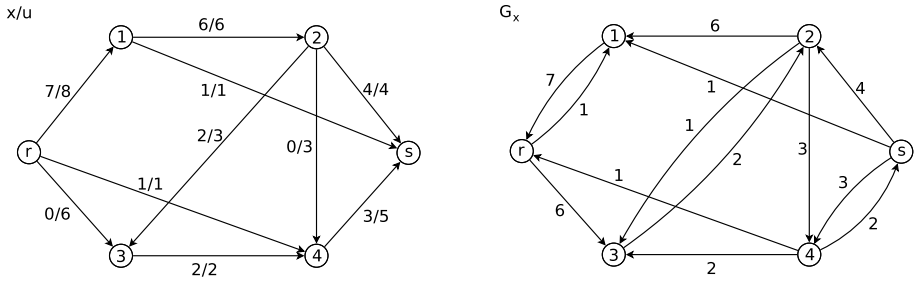
where $u_s^{\max}$ is the maximum number of units that can be pushed all the way from source to sink.

The next issue is how the search in the residual graph for an augmenting path is made. Several strategies can be used, but in order to ensure a polynomial running time we use a breath first approach as shown in algorithm 10. To realize the importance of the search strategy look at Figure 6.4.

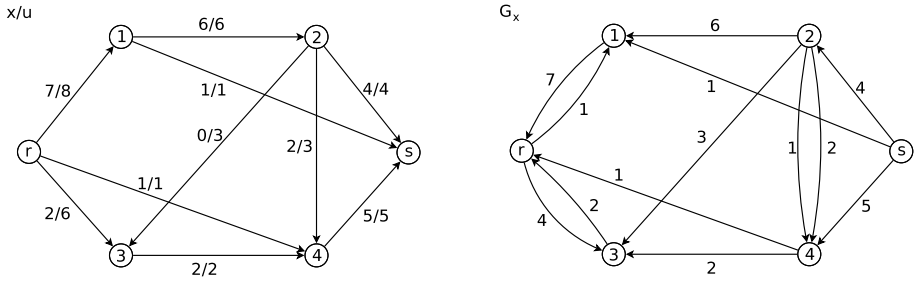
A first augmenting path could be $\langle r, 1, 2, s \rangle$. We can push 1 unit of flow through this path. Next iteration we get the augmenting path $\langle r, 2, 1, s \rangle$ again increasing the flow by one unit. Two things should be noted: 1) the running time of the algorithm is dependent on the capacities on the arcs (here we get a running time of $2M$ times the time it takes to update the flow) and 2) if M is large it will take a long time to reach the maximal flow although it is possible to find the same solution in only two iterations.



(a) The graph G with an initial flow an the corresponding residual graph



(b) Updating based on the augmenting path $\langle r, 1, s\rangle$



(c) Updating based on the augmenting path $\langle r, 3, 2, 4, s\rangle$

Figure 6.3: An example of the iterations in the augmenting path algorithm for the maximum flow problem

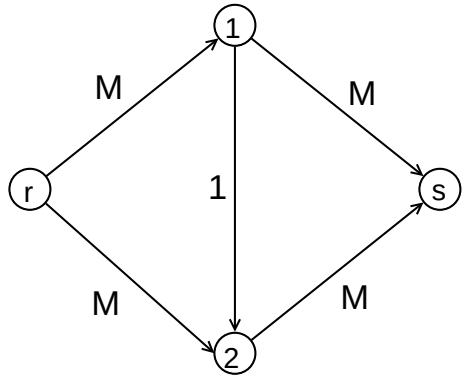


Figure 6.4: Example of what can go wrong if we select the "wrong" augmenting paths in the augmenting path algorithm

Algorithm 10: Finding an augmenting path if it exists

Data: The network $G = (V, E, u)$, a source vertex r, sink vertex s, and the current feasible flow x

Result: An augmenting path from r to s and the capacity for the flow augmentation, if it exists

all vertices are set to unlabeled
$Q \leftarrow \{r\}, S \leftarrow \emptyset$
$p_i \leftarrow 0$ for all i
$u_i^{\max} \leftarrow +\infty$ for $i \in G$
while $Q \neq \emptyset$ *and* s *is not labeled yet* **do**
    select a $i \in Q$
    scan $(i, j) \in E_x$
    label j
    $Q \leftarrow Q \cup \{j\}$
    $p_j \leftarrow i$
    if $(i, j) \in E_x$ *is forward in* G **then**
        $u_j^{\max} \leftarrow \min\{u_i^{\max}, u_{ij} - x_{ij}\}$
    if $(i, j) \in E_x$ *is backward in* G **then**
        $u_j^{\max} \leftarrow \min\{u_i^{\max}, x_{ji}\}$
    $Q \leftarrow Q \setminus i$
    $S \leftarrow S \cup \{i\}$

We call an augmenting path from r to s *shortest* if it has the minimum possible number of arcs. The augmenting path algorithm with a breadth-first search

solves the maximum flow problem in $O(nm^2)$. Breadth-first can be implemented using the queue data structure for the set Q.

6.2 The Preflow-Push Method

An alternative method to the augmenting path method is the *preflow-push* algorithm for the Max Flow problem. This method is sometimes also called *Push-relabel*.

In the augmenting flow method a flow was maintained in each iteration of the algorithm. In the preflow-push method the balance constraint – in-flow for a vertex equals out-flow for a vertex – is relaxed by introducing a *preflow*.

Definition 6.7 A **preflow** for a directed graph G with capacities u on the edges, a source vertex r and a sink vertex s is a function $x : E \rightarrow \mathcal{R}_+$ satisfying:

$$\forall i \in V \setminus \{r, s\} \ : \sum_{(j,i)\in E} x_{ji} - \sum_{(i,j)\in E} x_{ij} \geq 0$$

$$\forall (i, j) \in E \ : \ 0 \leq x_{ij} \leq u_{ij}$$

In other words: for any vertex v except r and s, the flow excess $f_x(i)$ is non-negative – "more runs into i than out of i". If $f_x(i) > 0$ then node i is called **active**. A preflow with no active nodes is a flow.

An important component of the preflow-push method is the residual graph which remains unchanged. So for a given graph G and preflow x the residual graph G_x shows where we can push excess flow.

The general idea in the preflow-push algorithm is to push as much flow as possible from the source towards the sink. However it is not possible to push more flow towards the sink s, **and** there are still active nodes we push the excess back toward the source r. This restores the balance of flow.

A push operation will move excess flow from an active node to another node along an forward arc in the residual graph. Figure 6.5 (a) illustrates the process. The nodes 2 and 3 are active nodes. In the graph we could push additional flow from node 3 to node 2. In this case node 3 would be in balance and the only remaining active node would be node 2 as illustrated in Figure 6.5 (b). If we are not careful the next push operation could potentially take one units of flow

from node 2 back to node 3 and thereby reestablish the preflow we just had (as there is an $(2, 3)$ edge in E_x). It is therefore necessary be able to control the direction of the pushes in order to avoid repeating pushes.

In order to control the direction of the pushes we introduce what will later turn out to be estimates (in fact lower bounds) on the distances in G_x called a labeling.

Definition 6.8 A **valid labeling** of the vertices in V wrt. a preflow x is a function $d : V \to \mathcal{Z}$ satisfying:

1. $d_r = n \ \wedge \ d_s = 0$
2. $\forall (i, j) \in E_x : d_i \leq d_j + 1$

The idea is that as long as it is possible we would like to send the flow "towards" the sink. In order to estimate that we use the valid labeling.

Having defined valid labeling it is natural to ask if any feasible preflow admits a valid labeling. The answer is no. Look at the simple example in Figure 6.6.

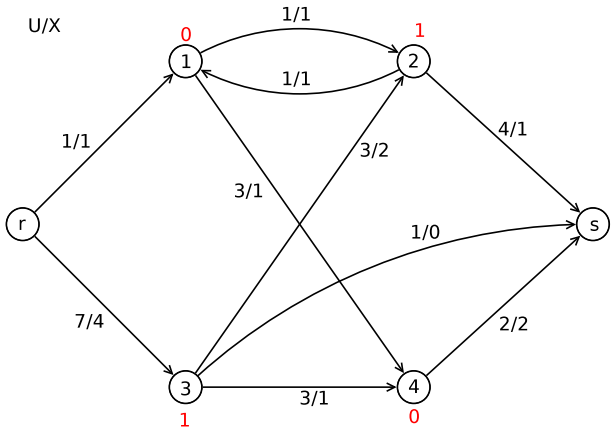
First of all d_r must be 3 and d_s 0 according to the definition. What value can we assign to d_a in order to get a valid labeling? According to rule 2 based on the arc (r, a) we have that $d_r \leq d_a + 1$ and using rule 2 on the (a, s) arc we get $d_a \leq d_s + 1$. This gives $d_a \geq 2$ and $d_a \leq 1$ which clearly does not leave room for a feasible value.

It is however possible to construct an initial feasible preflow that permits a valid labeling. We will call this procedure an initialization of x and d. In order to get a feasible preflow with a valid labeling we:

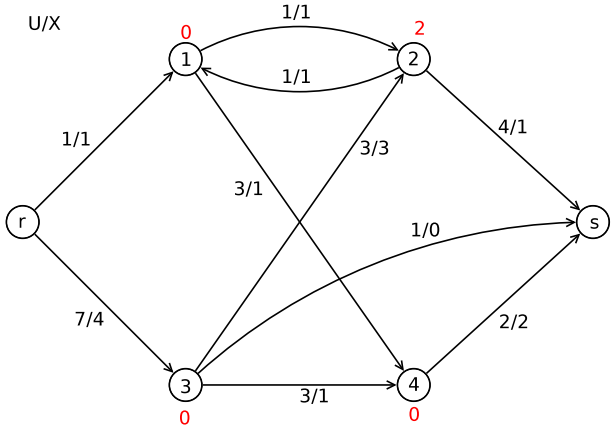
1. set $x_e \leftarrow u_e$ for all outgoing arcs of r,
2. set $x_e \leftarrow 0$ for all remaining arcs, and
3. set $d_r \leftarrow n$ and $d_i \leftarrow 0$ otherwise.

For the max flow problem in Figure 6.5 the initialization will result in the labels shown in Figure 6.7.

A valid labeling for a preflow implies an important property of the preflow, that it "saturates a cut", that is, all edges leaving the cut have flow equal to capacity. If we look at Figure 6.7 the cut defined by $R = \{r\}$ is saturated.



(a) Inititial preflow



(b) Pushing one unit from node 3 to node 2

Figure 6.5: Changing a preflow by applying a push operation

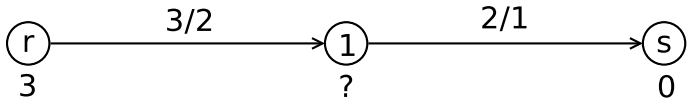


Figure 6.6: A preflow does not always admit a valid labeling as this small example shows

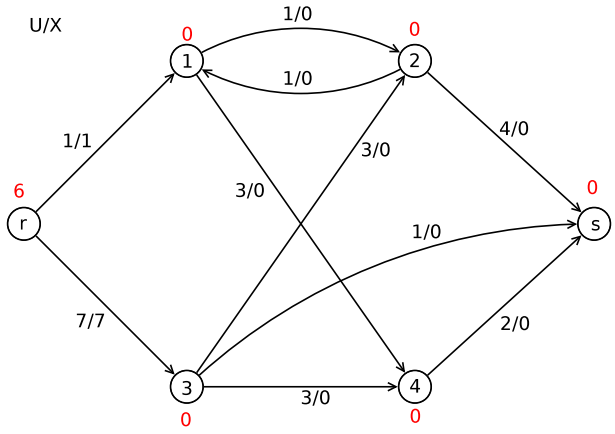


Figure 6.7: Initialization of the preflow and a corresponding valid labeling. The values on the nodes is the labeling.

In a way the augmenting path algorithm and the preflow-push algorithm are "dual" to each other. While the augmenting algorithm maintains a flow and terminates as a saturated cut is found, the preflow-push algorithm maintains a saturated cut and terminates as a feasible flow has been detected.

Theorem 6.9 *Given a feasible preflow x and a valid labeling d for x, then there exists an r,s-cut $\delta(R)$ such that $x_{ij} = u_{ij}$ for all $(i,j) \in \delta(R)$ and $x_{ij} = 0$ for all $(i,j) \in \delta(\overline{R})$.*

Proof: Since there are n nodes, there exists a value $k, 0 < k < n$ such that $d_i \neq k$ for all $i \in V$. Define $R = \{i \in V : d_i > k\}$. Clearly $r \in R$ $(d_r = n)$ and $s \notin R$ $(d_s = 0)$. R therefore defines an r,s-cut $\delta(R)$.

Let us look at an edge (i,j) in the residual graph that crosses the cut, that is $i \in R$ and $j \in \overline{R}$. As the labeling is valid we must have $d_i \neq d_j + 1$. As $i \in R$ $d_i \geq k+1$ and as $j \notin R$ $d_j \leq k-1$. No edge can fulfill the valid labeling and

consequently not edge in E_x is leaving R. $\triangle$

Let us return to the relationship between the labeling and the distance from a node to the sink. Let $d_x(i,j)$ denote the number of arcs in a shortest dipath from i to j in G_x. With the definition d we can prove that for any feasible preflow x and any valid labeling d for x we have

$$d_x(i,j) \geq d_i - d_j$$

Basically for every edge (i,j) in the shortest path P from i to j we have $d_i \leq d_j + 1$. Adding those together gives the states result.

As a consequence from this result we have:

Corollary 6.10 *Estimates on distances. As special cases we have:*

- d_i *is a lower bound on the distance from* i *to* s.
- $d_i - n$ *is a lower bound on the distance from* i *to* r.
- *If* $d_i \geq n$ *this means that there is no path from* i *to* s.
- $d_i < 2n$ *for all nodes.*

We try to push flow towards nodes j having $d_j < d_i$, since such nodes are estimated to be closer to the ultimate destination. Having $d_j < d_i$ and a valid labeling means that push is only applied to arcs (i,j) where i is active and $d_i = d_j + 1$. Such an edge (i,j) is called an **admissible edge**.

Algorithm 11: The Push operation

1 consider an **admissible** edge (i,j)
2 calculate the amount of flow, which can be pushed to w j
$(\min\{f_x(i), (x_{ji} + (u_{ij} - x_{ij}))\})$
3 push this by first reducing x_{ji} as much as possible and then increasing x_{ij} as much as possible until the relevant amount has been pushed

If an admissible edge is available the preflow-push algorithm can push flow on that edge. If we return to our max flow problem in Figure 6.7 it is obvious that no admissible edges exist. So what should we do if there are no more admissible edges and a node v is still active?

The answer is to relabel. Relabel means that we take an active node and assign it a new label that should result in generating at least one admissible edge. This is done by increasing the value of the label to the minimum of any label on an outgoing edge from the node plus one.

Algorithm 12: The Relabel operation

consider an active vertex i, for which **no** edge $(i, j) \in E_x$ with $d_i = d_j + 1$
$d_i \leftarrow \min\{d_j + 1 : (i, j) \in E_x\}$

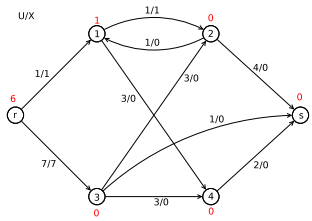
Notice that this will not violate the validity of the labeling (check for yourself!). Now we can put the elements together and get the preflow-push algorithm.

Algorithm 13: The Preflow-Push algorithm

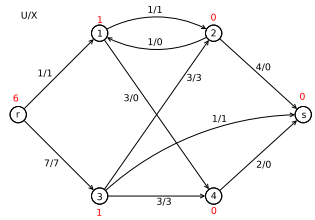
```
initialize x, d
while x is not a flow do
    select an active node i
    if no admissible arc (i, j) out of i exist then
        relabel i
    while there exists an admissible arc (i, j) do
        push on (i, j)
```

If we return to our example from before we may choose any active node and perform a relabel. Initially the nodes 1 and 3 are active, and we arbitrarily choose node 1. As no admissible edges exist we relabel to 1, and now the single unit of surplus in node 1 is pushed along $(1, 2)$ (we could also have chosen edge $(1, 4)$). Now nodes 2 and 3 are active nodes. We choose node 3 and need to relabel. Now we can push the surplus of 7 along the edges $(3, 2)$ (3 units), $(3, s)$ (1 unit) and $(3, 4)$ (3 units). Now node 3 is no longer active but 4 is. First we need to relabel to get a label of value 1 and push 2 units to s. As node 2 remains active we choose it again and following need to relabel to 2 and now we can push one unit "back" to node 3. In the next iteration nodes 2 and 3 are active and will be chosen by the preflow push algorithm for further operations.

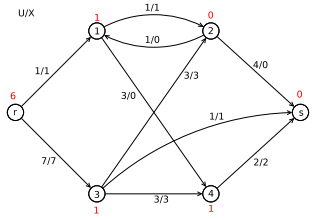
Analysis of the running time of the preflow-push algorithm is based on an analysis of the number of saturating pushes and non-saturating pushes. It can be shown that the push-relabel maximum flow algorithm can be implemented to run in time $O(n^2m)$ or $O(n^3)$.



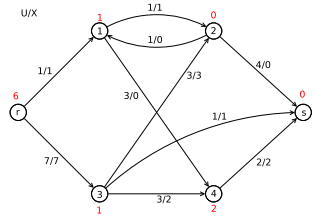
(a) Relabel and push from 1



(b) Relabel and push from 3



(c) Relabel and push from 4



(d) Relabel and push from 4 (again)

Figure 6.8: Initial iterations of the preflow push algorithm on our example graph.

6.3 Applications of the max flow problem

6.3.1 Maximum Bipartite Matching

6.4 Supplementary Notes

6.5 Exercises

1. Consider the max flow problem defined by the graph $G = (V, E, u)$ in Figure 6.9 with node 1 as the source and node 8 as the sink. Solve the problem using the augmenting path algorithm and the preflow push algorithm. Also find a minimum cut.

   In order to verify your results enter the integer programming model into your favourite MIP solver.

2. Let G be a directed graph with two special vertices s and t. Any two directed paths from s to t are called *vertex-disjoint* if they do not share any vertices other than s and t. Prove that the maximum number of directed vertex-disjoint paths from s to t is equal to the minimum number

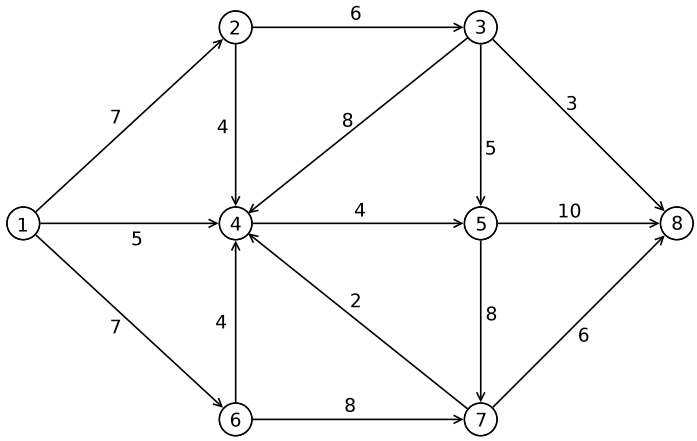


Figure 6.9: Example max flow problem

of vertices whose removal ensures that there are no directed paths from s to t.

3. Let P range over the set of s–t paths for two vertices s, t of a given graph. Let C range over cuts that separate s and t. Then show that

$$\max_{P} \min_{e \in P} c_e = \min_{C} \max_{e \in C} c_e$$

where c_e is the capacity of edge e.

Bibliography

[1] William J. Cook, William H. Cunningham, William R. Pulleyblank and Alexander Schrijver, *Combinatorial Optimization*, Wiley Interscience, 1998.

Chapter 7

The Minimum Cost Flow Problem

Recall the definition of $f_x(i)$ from the maximum flow problem:

$$f_x(i) = \sum_{(j,i)\in E} x_{ji} - \sum_{(i,j)\in E} x_{ij}$$

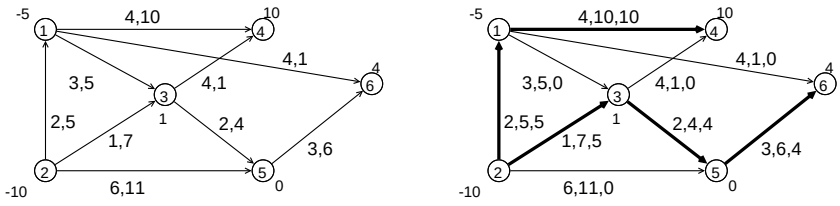
In the minimum cost flow problem a feasible flow x of minimum cost has to be determined. Feasibility conditions remains unchanged from the maximum flow problem. The minimum cost flow problem can be defined as:

$$\begin{aligned} \min \quad & \sum_{e\in E} w_e x_e & (7.1)\\ \text{s.t.} \quad & f_x(i) = b_i \qquad i \in V \\ & 0 \le x_{ij} \le u_{ij} \quad (i,j) \in E \end{aligned}$$

In the minimum cost flow problem we consider a directed graph $G = (V, E)$ where each edge e has a **capacity** $u_e \in \mathcal{R}_+ \cup \{\infty\}$ and a **unit transportation cost** $w_e \in \mathcal{R}$. Each vertex i has a **demand** $b_i \in \mathcal{R}$. If $b_i \ge 0$ then i is called a **sink**, and if $b_i < 0$ then i is called a **source**. We assume that $b_V = \sum_{i\in V} b_i = 0$,

that is, inflow in the graph is equal to outflow. This network can be defined as $G = (V, E, w, u, b)$.

An example of a minimum cost flow problem is shown in Figure 7.1. Vertices 1 and 2 are sources and 3, 4 and 6 are sinks. The objective is to direct the flow from 1 and 2 in the most cost efficient way to 3, 4 and 6.



(a) Graph (b) Feasible flow

Figure 7.1: Example of a minimum cost flow problem. Labels on edges on the graph shown on Figure (a) are represented by w_e, u_e. Figure (b) shows a feasible flow (bold edges). Labels on the edges represent w_e, u_e, x_e. The cost of the flow is 75

The minimum cost flow problem is a very versatile problem. Among others, the following flow problems can be described as special cases, we will just note the following:

1. The Transportation Problem

2. The Shortest Path Problem

3. The Max Flow Problem

A class of minimum cost flow problems arises from transportation problems. In a transportation problem $G = (V, E)$ is a bipartite graph with partitions $\{I, J\}$, and we are given positive numbers $a_i, i \in I$ and $b_j, j \in J$, as well as costs $w_{ij}, (i, j) \in E$. The set I defines a set of factories and J a set of warehouses. Given production quantities (a_i), demands (b_j), and unit transportation cost from i to j (w_{ij}) the objective is to find the cheapest transportation of products

to the warehouses. The transportation can be stated as:

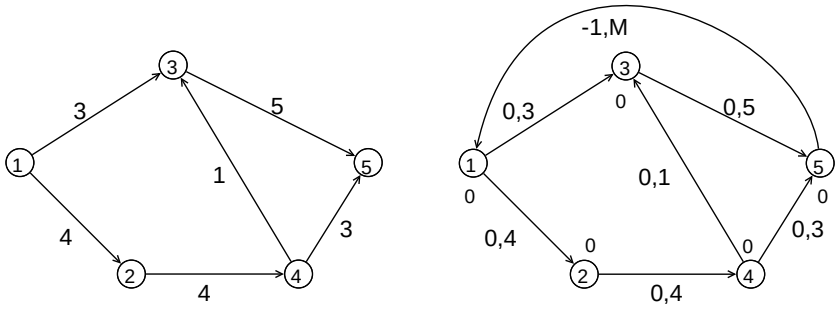
$$\begin{aligned} \min \quad & \sum_{e \in E} w_e x_e & & (7.2) \\ \text{s.t.} \quad & \sum_{i \in I, (i,j) \in E} x_{ij} = b_j & j \in J \\ & \sum_{j \in J, (i,j) \in E} x_{ij} = a_i & i \in I \\ & x_e \geq 0 & e \in E \end{aligned}$$

Adding constraints of the form $x_e \leq u_e$, where u_e is finite, to (7.2), the problem is called a **capacitated transportation problem**. In either case, multiplying each of the second group of equations by -1, and setting $b_i = -a_i$ for $i \in I$, we get a problem on the same form as minimum cost flow problem.

The shortest path problem can also be considered as a special case of the minimum cost flow problem. Define the root as a source and the remaining $n-1$ other vertices as sinks. Each of the sinks will have $d_i = 1$, while the root will have $d_r = n - 1$. This represents $n - 1$ paths are "flowing" out of the root and exactly one is ending in each of the other vertices. The unit transportation cost will be equal to the cost on the edges and we have stated the shortest path problem as a minimum cost flow problem.

The maximum flow problem can be modelled as a minimum cost flow problem by the following trick: Given a directed graph with capacities on the edges, a source and a sink we define $b_i = 0$ for all vertices, and set the unit transportation cost to 0 for each of the edges. Furthermore we introduce a new edge from the sink back to the source with unlimited capacity (at least larger than or equal to $\sum_{e \in E} u_e$) and a unit cost of -1. Now in order to minimize cost the model will try to maximize the flow from sink to source which at the same time will maximize the flow from the source through the graph to the sink as these flows must be in balance. This is illustrated in Figure 7.2.

As several of the important flow optimization problems can be stated as special cases of the minimum cost flow problem it is interesting to come up with an effective solution approach for the problem. Let us take a look at the constraint matrix of the minimum cost flow problem.



(a) The graph of a maximum flow problem

(b) The graph "restated" as a minimum cost flow problem

Figure 7.2: Figure (a) shows a maximum flow problem with labels representing u_e. In Figure (b) the same maximum flow problem has been reformulated as a minimum cost flow problem. Labels on the edges are w_e, u_e. Note that M should be chosen to be at least 20

	x_{e_1}	x_{e_2}	$\dots$	x_{ij}	$\dots$	x_{e_m}		
	w_{e_1}	w_{e_2}	$\dots$	w_{ij}	$\dots$	w_{e_m}		
1	-1	.	$\dots$		$\dots$		$=$	b_1
2	.	.	$\dots$	.	$\dots$	.	$=$	b_2
$\vdots$	.	.	$\dots$	.	$\dots$	.	$=$	$\vdots$
i	1	.	$\dots$	-1	$\dots$	.	$=$	b_i
$\vdots$	.	.	$\dots$	.	$\dots$	.	$=$	
j	.	.	$\dots$	1	$\dots$	.	$=$	b_j
$\vdots$	.	.	$\dots$	.	$\dots$	.	$=$	$\vdots$
n	.	.	$\dots$	.	$\dots$	.	$=$	b_n
e_1	-1						$\geq$	$-u_1$
e_2		-1					$\geq$	$-u_2$
$\vdots$	.	.	.	.	.	.	$\geq$	$\vdots$
(i,j)				-1			$\geq$	$-u_{ij}$
$\vdots$	.	.	.	.	.	.	$\geq$	$\vdots$
e_m						-1	$\geq$	$-u_m$

Proposition 3.2 from [2] gives a sufficient condition for a matrix being **totally unimodular**. It states that a matrix A is totally unimodular if:

1. $a_{ij} \in \{+1, -1, 0\}$ for all i, j.

2. Each column contains at most two nonzero coefficients ($\sum_{i=1}^{m} |a_{ij}| \leq 2$).

3. There exists a partition $\{M_1, M_2\}$ of the set M of rows such that each column j containing two non-zeros satisfies $\sum_{i \in M_1} a_{ij} - \sum_{i \in M_2} a_{ij} = 0$.

Consider the matrix without the $x_e \leq u_e$ constraints. If we let M_1 contain the remaining constraints and $M_2 = \emptyset$ the conditions of proposition 3.2 is fulfilled. The matrix is therefore totally unimodular. It is let to the reader to prove that the entire matrix with the $x_e \leq u_e$ constraints also is totally unimodular.

A consequence is the the minimum cost flow problem can be solved by simply removing integrality constraints from the model and solve it using an LP solver like eg. CPLEX.

In order to come up with a more efficient approach we take a look at optimality conditions for the minimum cost flow problem.

7.1 Optimality conditions

Having stated the primal LP of the minimum cost flow problem in (7.1) it is a straightforward task to formulate the dual LP. The dual LP has two sets of variables: the dual variables corresponding to the flow balance equations are denoted $y_i, i \in V$, and those corresponding to the capacity constraints are denoted $z_{ij}, (i, j) \in E$. The dual problem is:

$$\begin{aligned} \max \quad & \sum_{i \in V} b_i y_i - \sum_{(i,j) \in E} u_{ij} z_{ij} && (7.3) \\ \text{s.t.} \quad & -y_i + y_j - z_{ij} \leq w_{ij} \Leftrightarrow && (i, j) \in E \\ & -w_{ij} - y_i + y_j \leq z_{ij} && (i, j) \in E \\ & z_{ij} \geq 0 && (i, j) \in E \end{aligned}$$

Define the **reduced cost** $\bar{w}_{ij}$ of an edge (i, j) as $\bar{w}_{ij} = w_{ij} + y_i - y_j$. Hence, the constraint $-w_{ij} - y_i + y_j \leq z_{ij}$ is equivalent to

$$-\bar{w}_{ij} \leq z_{ij}$$

When is the set of feasible solutions x, y and z optimal? Let us derive the optimality conditions based on the primal and dual LP. Consider the two different cases:

1. If $u_e = +\infty$ (i.e. no capacity constraints for the edges) then z_e must be 0 and hence just $\bar{w}_e \geq 0$ has to hold. The constraint now corresponds to the primal optimality condition for the LP.
2. If $u_e < \infty$ then $z_e \geq 0$ and $z_e \geq -\bar{w}_e$ must hold. As z have **negative** coefficients in the objective function of the dual problem the best choice for z is as small as possible, that is, $z_e = \max\{0, -\bar{w}_e\}$. Therefore, the optimal value of z_e is uniquely determined from the other variables, and hence z_e is "unnecessary" in the dual problem.

Complementary slackness conditions is used to obtain optimality conditions. Generally, complementary slackness states that 1) each primal variable times the corresponding dual slack must equal 0, 2) and each dual variable times the corresponding primal slack must equal 0 in optimum). In our case this results in:

$$x_e > 0 \Rightarrow -\bar{w}_e = z_e = \max(0, -\bar{w}_e)$$
$$\text{i.e. } x_e > 0 \Rightarrow -\bar{w}_e \geq 0, \text{ that is } \bar{w}_e > 0 \Rightarrow x_e = 0$$
$$\text{and}$$
$$z_e > 0 \Rightarrow x_e = u_e$$
$$\text{i.e. } -\bar{w}_e > 0 \Rightarrow x_e = u_e, \text{ that is } \bar{w}_e < 0 \Rightarrow x_e = u_e$$

Summing up: A primal feasible flow satisfying demands respecting the capacity constraints *is optimal if and only if* there exists a dual solution $y_e, e \in E$ such that for all $e \in E$ it holds that:

$$\begin{aligned} \bar{w}_e < 0 \quad &\text{implies} \quad x_e = u_e (\neq \infty) \\ \bar{w}_e > 0 \quad &\text{implies} \quad x_e = 0 \end{aligned}$$

All pairs (x, y) of optimal solutions satisfy these conditions. The condition can be used to define a solution approach for the minimum cost flow problem.

To present the approach we initially define a residual graph. For a legal flow x in G, the **residual graph** is (like for maximum flow problem) a graph, in which the edges indicate **how flow excess can be moved** in G given that the flow x already is present. The only difference to the residual graph of the maximum flow problem is that each edge additionally has a cost assigned. The residual graph G_x for G wrt. x is defined by $V(G_x) = V$ and $E(G_x) = E_x = \{(i,j) \in E : (i,j) \in E \wedge x_{ij} < u_{ij}\} \cup \{(i,j) : (j,i) \in E \wedge x_{ji} > 0\}$.

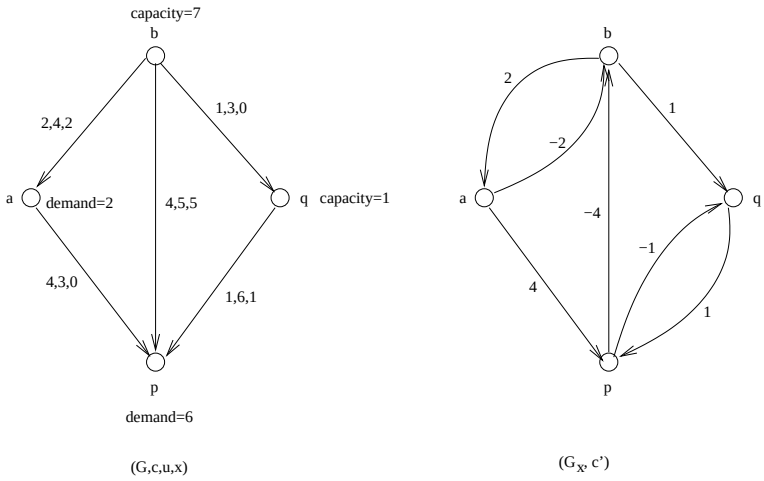


Figure 7.3: $G = (V, E, c, u, x)$ and its residual graph

The unit transportation cost w'_{ij} of an edge with $x_{ij} < u_{ij}$ is w_{ij}, while w'_{ij} of an edge with $x_{ji} > 0$ is $-w_{ji}$. The reason for this definition can be explained by considering an x-incrementing path P. P contains both forward and backward edges. Suppose we now send one unit of flow from one end to the other of the path. Then x_e will be raised by one on forward edges and lowered by one on backward edges. Subsequently the cost must reflect the change, hence we have $w'_{ij} = w_{ij}$ for forward edges and $w'_{ij} = -w_{ji}$ for backward edges.

Note that a dicircuit with negative cost in G_x corresponds to a negative cost circuit in G, if costs are added for forward edges and subtracted for backward edges, see Figure 7.3.

Note that if a set of potentials y_i, $i \in V$ are given, and the cost of a circuit wrt. the reduced costs for the edges ($\bar{w}_{ij} = w_{ij} + y_i - y_j$) are calculated, the cost remains the same as the original costs as the potentials are "telescoped" to 0.

A primal feasible flow satisfying demands and respecting the capacity constraints **is optimal if and only if** an x-augmenting circuit with negative w-cost (or negative $\bar{w}$-cost, there is no difference), does not exist.

This is the idea behind the identification of optimal solutions in the **network simplex algorithm** that will be presented later.

An initial and simple idea is to find a feasible flow and then in each iteration test

for the existence a negative circuit by trying to find a negative cost circuit in the residual graph. This test could be done using the Bellmann-Ford algorithm. It has a running time of $O(mn)$. This algorithm is called the **augmenting circuit algorithm**

Algorithm 14: The Augmenting Circuit Algorithm

```
1 Find a feasible flow x
2 while there exists an augmenting circuit do
3 |   Find an augmenting circuit C
4 |   if C has no reverse edge, and no forward edge of finite capacity then
5 |   |   STOP
6 |   Augment x on C
```

When performing a complete $O(nm)$ computation in every iteration it is paramount to keep the number of iterations low. Several implementations of the augmenting circuit algorithm exist. Most often these algorithms does not perform as well as the most effective methods. We will therefore turn our attention to one of the methods that is currently most used, namely the **Network Simplex Method**.

7.2 Network Simplex for The Transshipment problem

In order to get a better understanding of the network simplex approach we initially look at the minimum cost flow problem without capacities. This problem is also called the **transshipment problem**. Given a directed graph $G = (V, E)$ with demands d_i for each vertex $i \in V$ and unit transportation cost w_e for each edge $e \in E$ find the minimum cost flow that satisfies the demand constraints.

Assume G is connected. Otherwise the problem simply reduces to a transshipment problem for each of the components in G. A **tree** in a directed graph is a set $T \subseteq E$, such that, T is a tree in the underlying undirected graph. A **tree solution** for the transshipment-problem given by $G = (V, E)$, demands $b_i,\ i \in V$ and costs $w_e,\ e \in E$, is a flow $x \in \mathcal{R}^E$ satisfying:

$$\begin{aligned} \forall i \in V \quad &: \quad f_x(i) = b_i \\ \forall e \notin T \quad &: \quad x_e = 0 \end{aligned}$$

So there is no flow on edges not in T, and all demands are satisfied. Note that

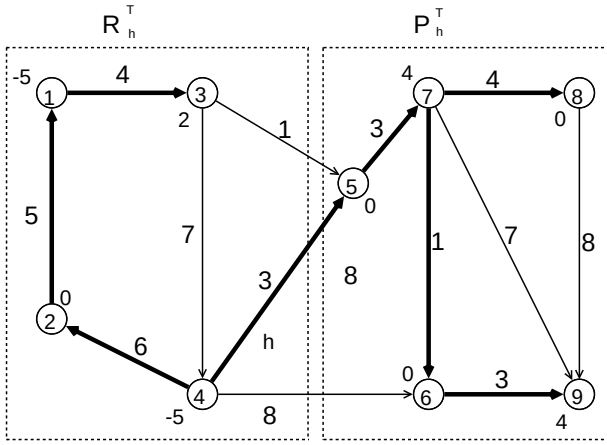


Figure 7.4: Let vertex 1 be the root vertex. Consider edge h as shown it defines two sets R_h^T and P_h^T

tree solutions are normally *not* feasible, since edges with negative flow may exist (cf. basic solutions and *feasible* basic solutions in linear programming).

Initially we may ask whether there exists a tree solution for any tree T. The answer is yes, and can be seen by the following constructive argument. Select a vertex r and call it the **root**. It can be any vertex. Consider an edge h in T. The edge h partitions V into two sets, one containing r denoted R_h^T and the remainder $P_h^T = V \setminus R_h^T$ (see Figure 7.4). If h starts in R_h^T and (consequently) ends in P_h^T then

$$x_h = \sum_{i \in P_h^T} b_i = -\sum_{i \in R_h^T} b_i. \tag{7.4}$$

Alternatively if h starts in P_h^T and ends in R_h^T then

$$x_h = \sum_{i \in R_h^T} b_i. \tag{7.5}$$

So a tree T uniquely defines a tree solution. In the example in Figure 7.4 $x_h = 8$.

It is essential to note that any feasible solution can be transformed into a tree solution. Consequently this will allow us to only look at tree solutions.

Theorem 7.1 *If the transshipment problem defined by $G = (V, E, w, b)$ has a feasible solution, then it also has a feasible tree solution.*

Proof: Let x be a feasible solution in G. If x is not a tree solution then there must exist a circuit C where all edges have a positive flow. We can assume that C has at least one backward edge. Let $\alpha = \min\{x_e : e \in C, e \text{ is a backward edge}\}$. Now replace x_e by $x_e + \alpha$ if e is a forward edge in C, and by $x_e - \alpha$ if e is a backward edge in C. The new x is feasible, and moreover, has one circuit less that the initial flow. If the flow is a tree solution we are done, if not, we continue the procedure. Note that each time we make an "update" to the flow, at least one edge with get flow equal to 0 and that we never put flow onto an edge with a flow of 0. Therefore we must terminate after a finite number of iterations. $\triangle$

If the flow x is an optimal solution and there exists a circuit C with positive flow, C must have zero cost. This means that the approach used in the proof above can be used to generate a new optimal solution in which fewer edges have a positive flow. So it can be proved that:

Theorem 7.2 *If the transshipment problem defined by $G = (V, E, w, b)$ has an optimal solution then it also has an optimal tree solution.*

The results of these two theorems are quite powerful. From now on we can restrict ourselves to look only on tree solutions. Consequently, the network simplex method moves from tree solution to tree solution using negative cost circuits C_e^T consisting of tree edges **and exactly one** non-tree edge e (think of the tree edges as basic variables and the non-tree edge as the non-basic variable of a simplex iteration). Remember that the non-tree edge e uniquely defines a circuit with T. Each edge e will define a unique circuit with the characteristics:

- $C_e^T \subseteq T \cup \{e\}$
- e is an forward edge in C_e^T

This is illustrated in Figure 7.5.

Consider the vector of potentials $y \in \mathcal{R}^V$. Set the vertex potential $y_i,\ i \in V$ to the cost of the (simple) path in T from r to i (counted with sign: plus for a forward edge, minus for a backward edge). It now holds that for all $i, j \in V$, the cost of the path from i to j in T equals $y_j - y_i$ (why ?). But then the reduced costs $\bar{w}_{ij}$ (defined by $w_{ij} + y_i - y_j$) satisfy:

$$\begin{aligned} \forall e \in T \quad &: \quad \bar{w}_e = 0 \\ \forall e \notin T \quad &: \quad \bar{w}_e = w(C_e^T) \end{aligned}$$

If T determines a *feasible* tree solution x and C_e^T has non-negative cost $\forall e \notin T$

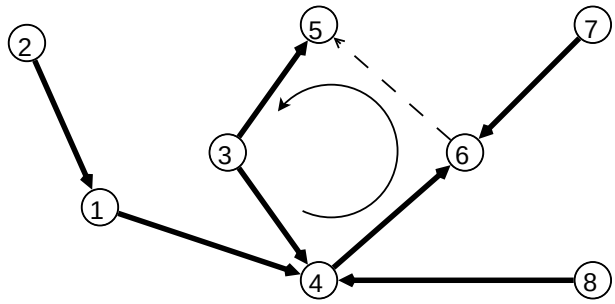


Figure 7.5: An example of the definition of the unique circuit C_e^T by the edge e. The edge $(6, 5)$ defines the circuit $\langle 3, 4, 6, 5, 3\rangle$ and the anti clockwise orientation

(i.e. $\bar{w}_e \geq 0$), then x is optimal. Now the network simplex algorithm for the transshipment problem can be stated:

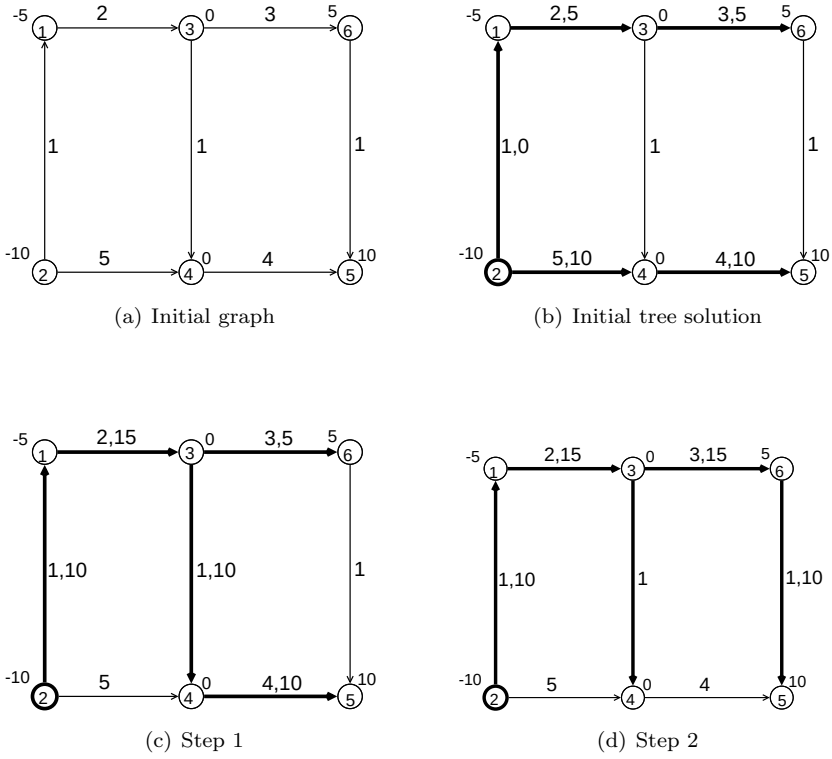
Algorithm 15: The Network Simplex Algorithm for the Transshipment Problem

find a tree T with a corresponding feasible tree solution x
select an $r \in V$ as root for T
compute y_i as the length of the $r - i$–path in T, costs counted with sign
while $\exists(i, j) : \bar{w}_{ij} = w_{ij} + y_i - y_j < 0$ **do**
    Select an edge with $\bar{w}_{ij} < 0$
    if *all edges in* C_e^T *are forward edges* **then**
        STOP `/* the problem is ‘‘unbounded’’ */`
    find $\theta = min\{x_j : j$ backward in $C_e^T\}$ and an edge h with $x_h = \theta$
    increase x with θ along C_e^T
    $T \leftarrow (T \cup \{e\}) \setminus \{h\}$
    update y

Remember as flow is increased in the algorithm it is done with respect to the orientation defined by the edge that defines the circuit, that is, increase flow in forward edges and decrease flow in backward edges.

Testing whether T satisfies the optimality conditions is relatively easy. We can compute y in time $O(n)$ and thereafter $\bar{w}$ can be computed in $O(m)$. With the simple operations involved and the modest time complexity this is certainly going to be faster than running Bellman-Fords algorithm.

An issue that we have not touched upon is how to construct the initial tree. One



(a) Initial graph

(b) Initial tree solution

(c) Step 1

(d) Step 2

Figure 7.6: The solution of an transshipment problem using the network simplex algorithm. The labels on the edges are u_e, x_e. Bold edges identify the tree solution

approach is the following: We can use the tree T whose edges are (r, i) where $i \in V \setminus \{r\}$ and $b_i \geq 0$ and edges (i, r) where $i \in V \setminus \{r\}$ and $b_i < 0$. Not all these edges may exist. If they do not exist we add them to G but assign them a large enough cost to secure they will not be part of an optimal solution.

Let us demonstrate the network simplex algorithm by the example in Figure 7.6.

First, we need to find an initial feasible tree solution. Having determined the tree as $T = \{(2,1),(1,3),(3,6),(2,4),(4,5)\}$ we let vertex 2 be the root vertex. Now the feasible solution wrt. tree solution is uniquely defined, and can be computed by (7.4) and (7.5) defined earlier. So we get the tree solution in the graph in the upper right of Figure 7.6. The flow corresponds to a value of 115.

Now we compute the potential vector $y = (1, 0, 3, 5, 9, 6)$. Let us check if we have a negative cost circuit. The edges $(3, 4)$ and $(6, 5)$ both uniquely determine two circuits with cost:

$$\begin{aligned} \bar{w}_{34} &= y_3 + w_{34} - y_4 = 3 + 1 - 5 = -1 \\ \bar{w}_{65} &= y_6 + w_{65} - y_5 = 6 + 1 - 9 = -2 \end{aligned}$$

So both edges define a negative cost circuit. We can choose either one of them. Let us select the circuit C_1 defined by $(3, 4)$. Every time we push one unit of flow around the circuit the flow on all edges but $(2, 4)$ in C_1 increase by one, while the flow decreases by one on the $(2, 4)$ edge. As the current flow is 10 on $(2, 4)$ we can increase the flow on the other edges by 10. $(2, 4)$ is discarded from the tree solution and $(3, 4)$ is added.

The resulting flow is shown in the lower left graph. Here the flow has a value of 105. We update y and get $(1, 0, 3, 4, 8, 6)$. Now we check for potential negative cost circuits:

$$\begin{aligned} \bar{w}_{24} &= y_2 + w_{24} - y_4 = 0 + 5 - 4 = 1 \\ \bar{w}_{65} &= y_6 + w_{65} - y_5 = 6 + 1 - 8 = -1 \end{aligned}$$

Naturally edge $(2, 4)$ cannot be a candidate for a negative cost circuit as we have just removed it from the tree solution, but the edge $(6, 5)$ defines a negative cycle C_2. In C_2 the forward edges are $(6, 5)$ and $(3, 6)$, while $(4, 5)$ and $(3, 4)$ are backward edges. The backward edges both have a flow of 10 so we decrease both to zero, while flows on the forward edges are increased by 10. One of the backward edges must leave the tree solution. The choice is really arbitrary, we will remove $(4, 5)$ from the tree solution and add $(6, 5)$. This gets us the solution in the lower right graph. With the updated y being $(1, 0, 3, 4, 7, 6)$ there are no more negative cost circuits and the algorithm terminates. The optimum flow has a value of 95.

7.3 Network Simplex Algorithm for the Minimum Cost Flow Problem

We now consider the general minimum cost flow problem given by the network $G = (V, E)$ with demands $b_i,\ i \in V$, capacities $u_e,\ e \in E$ and costs $w_e,\ e \in E$. In general, the algorithm for the general case of the minimum cost flow problem is a straightforward extension of the network simplex algorithm for the transshipment problem as described in the previous section. It can also be viewed as an interpretation of the bounded variable simplex method of linear programming.

Remember the optimality conditions for a feasible flow $x \in \mathcal{R}^E$:

$$\begin{aligned} \bar{w}_e < 0 &\Rightarrow x_e = u_e \ (\neq \infty) \\ \bar{w}_e > 0 &\Rightarrow x_e = 0 \end{aligned}$$

The concept of a tree solution is extended to capacity constrained problems. A non-tree edge e may now have flow either zero or u_e. $E \setminus T$ is hence partitioned into two sets of edges, L and U. Edges in L must have flow zero and edges in U must have flow equal to their capacity. The tree solution x must satisfy:

$$\begin{aligned} \forall i \in V &: f_x(i) = b_i \\ \forall e \in L &: x_e = 0 \\ \forall e \in U &: x_e = u_e \end{aligned}$$

As before, the tree solution for T is unique: Select a vertex r called the *root* of T. Consider an edge h in T. Again h partitions V into two: a part containing r denoted R_h^T and a remainder $P_h^T = V \setminus R_h^T$.

Consider now the *modified demand* $B(P_h^T)$ in P_h^T *given that a tree solution is sought*:

$$\begin{aligned} B(P_h^T) = &\sum_{i \in P_h^T} b_i \\ - &\sum_{\{(i,j) \in U : i \in R_h^T, j \in P_h^T\}} u_{ij} \\ + &\sum_{\{(i,j) \in U : i \in P_h^T, j \in R_h^T\}} u_{ij} \end{aligned}$$

If h starts in R_h^T and ends in P_h^T then set

$$x_h = B(P_h^T)$$

and if h starts in P_h^T and ends in R_h^T then set

$$x_h = -B(P_h^T).$$

Now the algorithm for the general minimum cost flow problem is based on the following theorem, that will be presented without proof:

Theorem 7.3 *If $G = (V, E, w, u, b)$ has a feasible solution, then it has a feasible tree solution. If it has an optimal solution, then it has an optimal tree solution.*

The network simplex algorithm will move from tree solution to tree solution only using circuits formed by adding a single edge to T. However, if the non-tree edge being consider has flow equal to capacity, then we consider sending flow in the opposite direction. We now search for *either* a non-tree edge e with $x_e = 0$ and negative reduced cost *or* a non-tree edges e with $x_e = u_e$ and positive reduced cost. Given e and T, L and $, U$, the corresponding circuit is denoted $C_e^{T,L,U}$. This circuit the satisfies:

- Each edge of $C_e^{T,L,U}$ is an element if $T \cup \{e\}$.
- If $e \in L$ it is a forward edge of $C_e^{T,L,U}$, and otherwise it is a backward edge.

With finite capacities it is not so easy to come up with an initial feasible tree solution. One approach is to add a "super source" $\bar{r}$ vertex and a "super sink" vertex $\bar{s}$ to G. Now edges from $\bar{r}$ to a source vertex i get capacity equal to the absolute value of the demand of i, that is, $u_{\bar{r}i} = -b_i$ and correspondingly we get $u_{i\bar{s}} = d_i$ for a sink vertex i. The capacity on edges from are set to the supply/demand of that node. On the "extended" G we solve a maximum flow problem. This will constitute a feasible solution and will present us with a feasible tree solution.

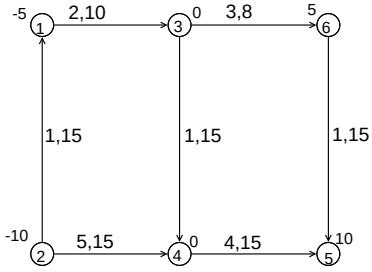
In this, the flow must be *increased* in forward edges respecting capacity constraints and *decreased* in backward edges respect the non-negativity constraints. We can now state the algorithm for the minimum cost flow problem.

Let us finish of the discussion on the network simplex by demonstrating it on an example. Figure 7.7 show the iterations.

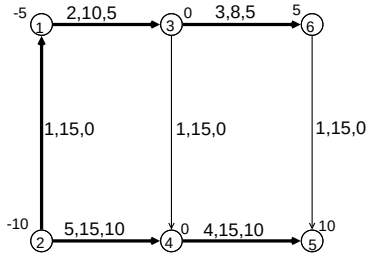
In this case it is easy to come up with a feasible tree solution. Both non-tree edges $(3, 4)$ and $(6, 5)$ are in L as their flow is equal to zero. Now we compute $y = (1, 0, 3, 5, 6, 9)$ and for each of the two non-tree edges we get:

$$\begin{aligned} \bar{w}_{34} &= y_3 + w_{34} - y_4 = 3 + 1 - 5 = -1 \\ \bar{w}_{65} &= y_6 + w_{65} - y_5 = 6 + 1 - 9 = -2 \end{aligned}$$

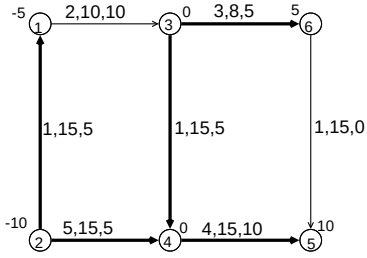
Both will improve the solution. Let us take $(3, 4)$. This defines a unique cycle with forward edges $(3, 4), (2, 1)$ and $(1, 3)$ and $(2, 4)$ being a backward edges. The bottleneck in this circuit becomes $(1, 3)$ as the flow on this edge only can be increased by five units. We increase the flow on forward edges by five and decrease it by five on $(2, 4)$ which gets us to step 1. Note now that the non-tree edge $(1, 3)$ belongs to U whereas $(6, 5)$ remains in L. We recompute y to be



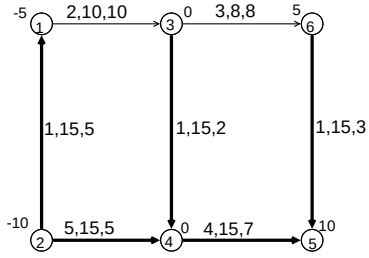
(a) Initial graph



(b) Initial tree solution



(c) Step 1



(d) Step 2

Figure 7.7: An example of the use of the network simplex algorithm. Labels on the edges represent w_e, u_e respectively w_e, u_e, x_e. Bold edges identify the tree solution

Algorithm 16: The Network Simplex Algorithm for the Minimum Cost Flow Problem

Find a tree T with a corresponding feasible tree solution x
Select a vertex $r \in V$ as root for T
Compute y_i as the length of the $r-i$–path in T, costs counted with sign (plus for forward edges, minus for backward edges)
while $\exists (i,j) \in L \; s.t \; \bar{w}_{ij} = w_{ij} + y_i - y_j < 0) \; \vee \; (\exists (i,j) \in U \; s.t \; \bar{w}_{ij} = w_{ij} + y_i - y_j > 0)$ **do**
    Select one of these
    if *no edge in* $C_e^{T,L,U}$ *is backward and no forward edge has limited capacity* **then**
        STOP
        `/* the problem is unbounded */`
    find $\theta_1 \leftarrow min\{x_j : j$ backward in $C_e^{T,L,U}\}$, $\theta_2 \leftarrow min\{u_j - x_j : j$ forward in $C_e^{T,L,U}\}$, and an edge h giving rise to $\theta \leftarrow min\{\theta_1, \theta_2\}$
    increase x by θ along $C_e^{T,L,U}$
    $T \leftarrow (T \cup \{e\}) \setminus \{h\}$
    Update L, U by removing e and inserting h in the relevant one of L and U
    Update y

$(1, 0, 4, 5, 7, 9)$. For the two non-tree edges we no get:

$$\begin{aligned} \bar{w}_{13} &= y_1 + w_{13} - y_3 = 1 + 2 - 4 = -1 \\ \bar{w}_{65} &= y_6 + w_{65} - y_5 = 7 + 1 - 9 = -1 \end{aligned}$$

Even though $\bar{w}_{13}$ is negative we can only choose $(6, 5)$ because $(1, 3) \in U$. We choose $(6, 5)$ and in the circuit defined by that edge the flow can be increase by at most three units. The edge that defines the bottleneck is edge $(3, 6)$ which is included in U, whereas $(1, 3)$ is included in T. This gets us to step 2 in the figure.

$$\begin{aligned} \bar{w}_{13} &= y_1 + w_{13} - y_3 = 1 + 2 - 4 = -1 \\ \bar{w}_{36} &= y_3 + w_{36} - y_6 = 4 + 3 - 8 = -1 \end{aligned}$$

Because both non-tree edges are in U and have negative reduced cost the tree solution is optimal.

7.4 Supplementary Notes

There is (naturally) a close relationship between the network simplex algorithm and the simplex algorithm. An in-depth discussion of these aspects of the network simplex algorithm can be found in [1].

7.5 Exercises

1. *([1])*For the minimum-cost flow problem of Figure 7.8 determine whether the indicated vector x is optimal. Numbers on edges are in order w_e, u_e, x_e, numbers at vertices are demands. If it is optimal, find a vector y that certifies its optimality.

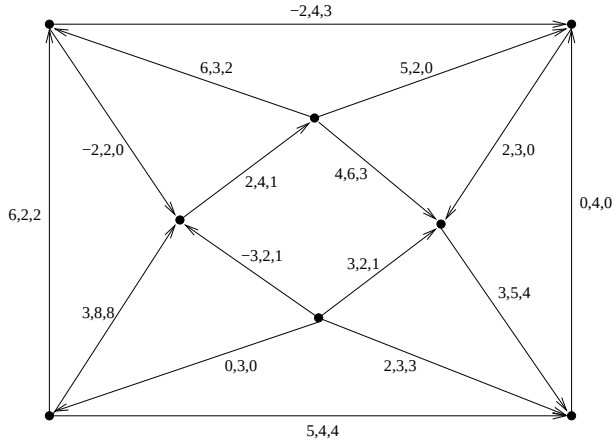


Figure 7.8: Given x, find y

2. **Bandwidth packing problem**. Consider a communications network, $G = (V, E)$, where there are n vertices and m edges. Each edge has a *bandwidth*, b_e, and unit cost w_e. A call i is defined by its starting vertex (s_i), destination vertex (d_i), bandwidth demand (w_i) and revenue (r_i). (Bandwidth demands are additive: if calls 1 and 2 both use the same edge, the total bandwidth requirement is $w_1 + w_2$.) Let N denote the number of calls, and we seek to maximize profit subject to logical flow requirements and bandwidth limits. Formulate this as a 0-1 Integer Linear Programming Problem.

3. **Piecewise linear objective function**. Suppose that the linear objective function for the minimum cost flow problem is replaced with a separable piecewise linear function. Show how to convert this problem to the standard minimum cost-flow problem.

4. **The Caterer Problem**. *([3])* A caterer has booked his services for the next T days. He requires r_t fresh napkins on the t'th day, $t = 1, 2, \ldots, T$. He sends his soiled napkins to the laundry, which has three speeds of service $f = 1, 2$, or 3 days. The faster the service the higher the cost c_f of laundering a napkin. He can also purchase new napkins at the cost c_o. He has an initial stock of s napkins. The caterer wishes to minimize the total outlay.

   (a) Formulate the problem as a network problem.

   (b) Solve the problem for the next 14 days. Daily requirement of napkins is given by Table 7.1. In addition the price of the three different laundry solutions are $f = 1$ costs 8 Danish kroner, $f = 2$ costs 4 Danish kroner and $f = 3$ costs 2 Danish kroner. The unit price of buying new napkins are 25 Danish kroner.

Day	1	2	3	4	5	6	7
Req.	500	600	300	800	2200	2600	1200
Day	8	9	10	11	12	13	14
Req.	300	1000	400	600	1400	3000	2000

Table 7.1: Daily requirement of napkins for the next 14 days.

5. **Flow restrictions**. *([3])* Suppose that in a minimum cost flow problem restrictions are placed on the total flow leaving a node k, i.e.

$$\underline{\theta}_k \leq \sum_{(k,j)\in E} x_{kj} \leq \overline{\theta}_k$$

   Show how to modify these restrictions to convert the problem into a standard minimum cost flow problem.

Bibliography

[1] William J. Cook, William H. Cunningham, William R. Pulleyblank and Alexander Schrijver. *Combinatorial Optimization*. Wiley Interscience, 1998.

[2] Laurence A. Wolsey. *Integer Programming*. Wiley Interscience, 1998.

[3] George B. Dantzig, Mukund N. Thapa. *Linear Programming*. Springer, 1997.

Part II

General Integer Programming

Chapter 8

Dynamic Programming

8.1 The Floyd-Warshall Algorithm

Another interesting variant of the Shortest Path Problem is the case where we want to find the Shortest Path between all pairs of vertices i, j in the problem. This can be resolved by repeated use one of our "one source" algorithms from earlier in this chapter. But there are faster and better ways to solve the "all pairs" problem.

This problem is relevant in many situations, most often as subproblems to other problems. One case is routing of vehicles. Having a fleet of vehicles and a set of customers that needs to be visited often we want to build a set of routes for the vehicles in order to minimize the total distance driven. In order to achieve this we need to know the shortest distances between every pair of customers before we can start to compute the best set of routes.

In the all pairs problem we no longer maintain just a vector of potentials and a vector of predecessors instead we need to maintain two *matrices* one for distance and one for predecessor.

One of the algorithms for the all pairs problem is the *Floyd-Warshall* algorithm. The algorithm is based on dynamic programming and considers "intermediate"

vertices of a shortest path. An intermediate vertex of a simple path is any of the vertices in the path except the first and the last.

Let us assume that the vertices of G are $V = \{1, 2, \ldots, n\}$. The algorithm is based on an observation that we, based on shortest paths between (i, j) containing only vertices in the set $\{1, 2, \ldots, k-1\}$, can construct shortest paths between (i, j) containing only vertices in the set $\{1, 2, \ldots, k\}$. In this way we may gradually construct shortest paths based on more and more vertices until they are based on the entire set of vertices V.

Algorithm 17: Floyd-Warshall's algorithm

Data: A distance matrix C for a digraph $G = (V, E, w)$. If the edge $(i, j) \in E$ w_{ij} is the distance from i to j, otherwise $w_{ij} = \infty$. w_{ii} equals 0 for all i

Result: Two $n \times n$-matrices, y and p, containing the length of the shortest path from i to j resp. the predecessor vertex for j on the shortest path for all pairs of vertices in $\{1, ..., n\} \times \{1, ..., n\}$

```
1 y_ij ← w_ij, p_ij ← i for all (i,j) with w_ij ≠ ∞, p_ij ← 0 otherwise.
2 for k ← 1 to n do
3     for i ← 1 to n do
4         for j ← 1 to n do
5             if i ≠ k ∧ j ≠ k ∧ y_ij > y_ik + y_kj then
6                 y_ij ← y_ik + y_kj
7                 p_ij ← p_kj
```

The time complexity of Floyd-Warshall's algorithm is straightforward. The initialisation sets up the matrices by initialising each entry in each matrix, which takes $O(n^2)$. The algorithm has three nested loops each of which is performed n times. The overall complexity is hence $O(n^3)$.

Theorem 8.1 *(Correctness of Floyd-Warshall's Algorithm) Given a connected, directed graph $G = (V, E, w)$ with length function $w : E \to R$. The Floyd-Warshall algorithm will produce a "shortest path matrix" such that for any two given vertices i and j y_{ij} is the shortest path from i to j.*

Proof: This proof is made by induction. Our induction hypothesis is that prior to iteration k it holds that for $i, j \in v$ y_{ij} contains length of the shortest path Q from i to j in G containing only vertices in the vertex set $\{1, ..., k-1\}$, and that p_{ij} contains the immediate predecesor of j on Q.

This is obviously true after the initialisation. In iteration k, the length of Q is compared to the length of a path R composed of two subpaths, $R1$ and $R2$ (see Figure 8.1).

$R1$ is a path from i to k path, that is, $R1 = \langle i, v_1, v_2, \ldots, v_s, k\rangle$ with "intermediate vertices" only in $\{1, ..., k-1\}$ so $v_a \in \{1, ..., k-1\}$, and $R2$ is a path from k to j path with "intermediate vertices" only in $\{1, ..., k-1\}$. The shorter of these two is chosen.

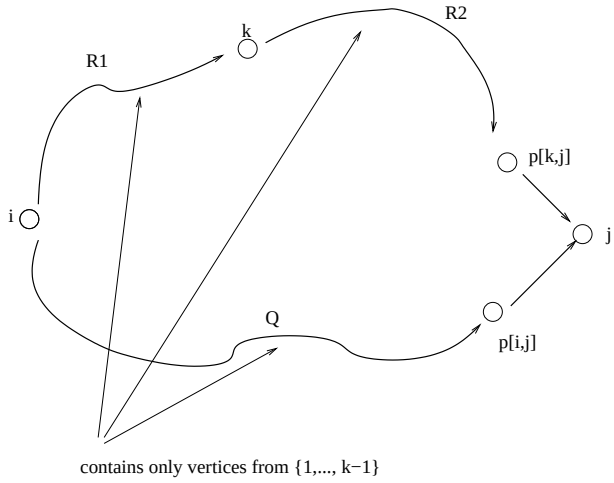


Figure 8.1: Proof of the Floyd-Warshall algorithm

The shortest path from i to j in G containing only vertices in the vertex set $\{1, ..., k\}$ either

1. does not contain k - and hence is the one found in iteration $k-1$, or
2. contains k - and then can be decomposed into an i, k followed by a k, j path, each of which, by the induction hypothesis, has been found in iteration $k-1$.

Hence the update ensures the correctness of the induction hypothesis after iteration k. $\triangle$

Finally, we note that the Floyd-Warshall algorithm can detect negative length cycles. It namely computes the shortest path from i to i (the diagonal of the W matrix) and if any of these values are negative it means that there is a negative cycle in the graph.

Chapter 9

Branch and Bound

A large number of real-world planning problems called combinatorial optimization problems share the following properties: They are optimization problems, are easy to state, and have a finite but usually very large number of feasible solutions. While some of these as e.g. the Shortest Path problem and the Minimum Spanning Tree problem have polynomial algorithms, the majority of the problems in addition share the property that no polynomial method for their solution is known. Examples here are vehicle routing, crew scheduling, and production planning. All of these problems are $\mathcal{NP}$-hard.

Branch and Bound is by far the most widely used tool for solving large scale $\mathcal{NP}$-hard combinatorial optimization problems. Branch and Bound is, however, an algorithm paradigm, which has to be filled out for each specific problem type, and numerous choices for each of the components exist. Even then, principles for the design of efficient Branch and Bound algorithms have emerged over the years.

In this chapter we review the main principles of Branch and Bound and illustrate the method and the different design issues through three examples: the symmetric Travelling Salesman Problem, the Graph Partitioning problem, and the Quadratic Assignment problem.

Solving $\mathcal{NP}$-hard discrete optimization problems to optimality is often an im-

mense job requiring very efficient algorithms, and the Branch and Bound paradigm is one of the main tools in construction of these. A Branch and Bound algorithm searches the complete space of solutions for a given problem for the best solution. However, explicit enumeration is normally impossible due to the exponentially increasing number of potential solutions. The use of bounds for the function to be optimized combined with the value of the current best solution enables the algorithm to search parts of the solution space only implicitly.

At any point during the solution process, the status of the solution with respect to the search of the solution space is described by a pool of yet unexplored subset of this and the best solution found so far. Initially only one subset exists, namely the complete solution space, and the best solution found so far is ∞. The unexplored subspaces are represented as nodes in a dynamically generated search tree, which initially only contains the root, and each iteration of a classical Branch and Bound algorithm processes one such node. The iteration has three main components: selection of the node to process, bound calculation, and branching. In Figure 9.1, the initial situation and the first step of the process is illustrated.

The sequence of these may vary according to the strategy chosen for selecting the next node to process. If the selection of next subproblem is based on the bound value of the subproblems, then the first operation of an iteration after choosing the node is branching, i.e. subdivision of the solution space of the node into two or more subspaces to be investigated in a subsequent iteration. For each of these, it is checked whether the subspace consists of a single solution, in which case it is compared to the current best solution keeping the best of these. Otherwise the bounding function for the subspace is calculated and compared to the current best solution. If it can be established that the subspace cannot contain the optimal solution, the whole subspace is discarded, else it is stored in the pool of live nodes together with it's bound. This is in [2] called the eager strategy for node evaluation, since bounds are calculated as soon as nodes are available. The alternative is to start by calculating the bound of the selected node and then branch on the node if necessary. The nodes created are then stored together with the bound of the processed node. This strategy is called lazy and is often used when the next node to be processed is chosen to be a live node of maximal depth in the search tree.

The search terminates when there is no unexplored parts of the solution space left, and the optimal solution is then the one recorded as "current best".

The chapter is organized as follows: In Section 9.1, I go into detail with terminology and problem description and give the three examples to be used succeedingly. Section 9.1.1, 9.1.2, and 9.1.3 then treat in detail the algorithmic components selection, bounding and branching, and Section 9.1.4 briefly com-

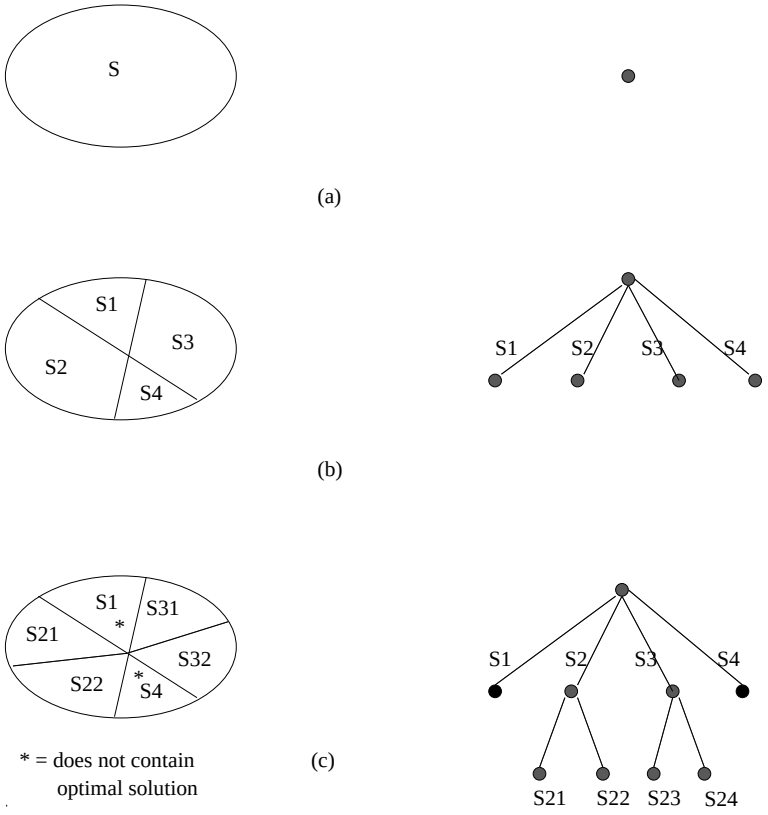


Figure 9.1: Illustration of the search space of Branch and Bound.

ments upon methods for generating a good feasible solution prior to the start of the search. I then describe personal experiences with solving two problems using parallel Branch and Bound in Section 9.2.1 and 9.2.2, and Section 9.3 discusses the impact of design decisions on the efficiency of the complete algorithm.

9.1 Branch and Bound - terminology and general description

In the following I consider minimization problems - the case of maximization problems can be dealt with similarly. The problem is to minimize a function $f(x)$ of variables $(x_1 \cdots x_n)$ over a region of *feasible solutions*, S :

$$z = \min\{f(x) : x \in S\}$$

The function f is called the *objective function* and may be of any type. The set of feasible solutions is usually determined by general conditions on the variables, e.g. that these must be non-negative integers or binary, and special constraints determining the structure of the feasible set. In many cases, a set of *potential solutions*, P, containing S, for which f is still well defined, naturally comes to mind, and often, a function $g(x)$ defined on S (or P) with the property that $g(x) \leq f(x)$ for all x in S (resp. P) arises naturally. Both P and g are very useful in the Branch and Bound context. Figure 9.2 illustrates the situation where S and P are intervals of reals.

I will use the terms *subproblem* to denote a problem derived from the originally given problem through addition of new constraints. A subproblem hence corresponds to a subspace of the original solution space, and the two terms are used interchangeably and in the context of a search tree interchangeably with the term *node*. In order to make the discussions more explicit I use three problems as examples. The first one is one of the most famous combinatorial optimization problems: the Travelling Salesman problem. The problem arises naturally in connection with routing of vehicles for delivery and pick-up of goods or persons, but has numerous other applications. A famous and thorough reference is [11].

Example 1: The Symmetric Travelling Salesman problem. In Figure 9.3, a map over the Danish island Bornholm is given together with a distance table showing the distances between major cities/tourist attractions. The problem of a biking tourist, who wants to visit all these major points, is to find a tour of

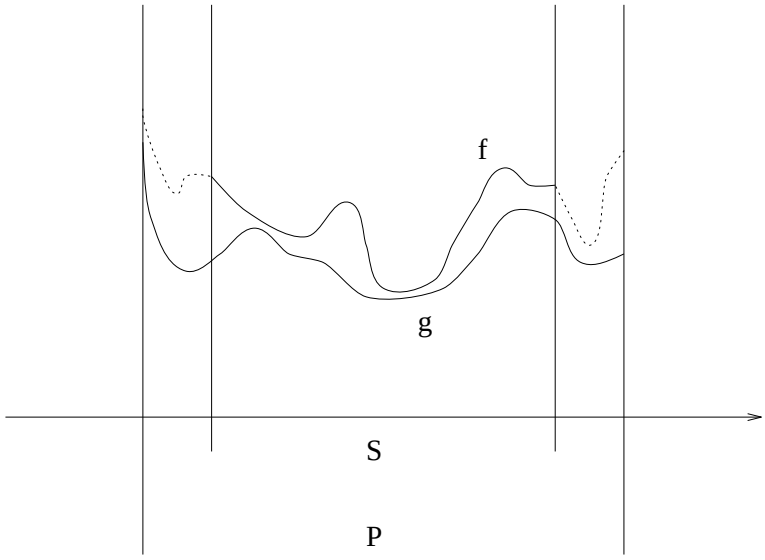
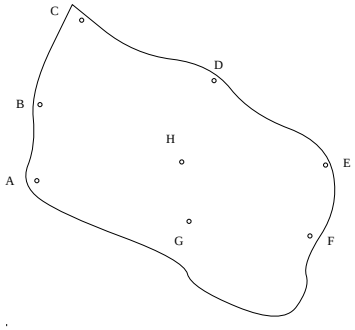


Figure 9.2: The relation between the bounding function g and the objective function f on the sets S and P of feasible and potential solutions of a problem.



	A	B	C	D	E	F	G	H
A	0	11	24	25	30	29	15	15
B	11	0	13	20	32	37	17	17
C	24	13	0	16	30	39	29	22
D	25	20	16	0	15	23	18	12
E	30	32	30	15	0	9	23	15
F	29	37	39	23	9	0	14	21
G	15	17	29	18	23	14	0	7
H	15	17	22	12	15	21	7	0

Figure 9.3: The island Bornholm and the distances between interesting sites

minimum length starting and ending in the same city, and visiting each other city exactly once. Such a tour is called a *Hamilton cycle*. The problem is called the *symmetric* Travelling Salesman problem (TSP) since the table of distances is symmetric.

In general a symmetric TSP is given by a symmetric $n \times n$ matrix D of non-negative distances, and the goal is to find a Hamilton tour of minimum length.

In terms of graphs, we consider a complete undirected graph with n vertices K_n and non-negative lengths assigned to the edges, and the goal is to determine a Hamilton tour of minimum length. The problem may also be stated mathematically by using decision variables to describe which edges are to be included in the tour. We introduce 0-1 variables $x_{ij}, 1 \leq i < j \leq n$, and interpret the value 0 (1 resp.) to mean "not in tour" ("in tour" resp.) The problem is then

$$\min \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_{ij}$$

such that

$$\sum_{k=1}^{i-1} x_{ki} + \sum_{k=i+1}^{n} x_{ik} = 2, \qquad i \in \{1, ..., n\}$$

$$\sum_{i,j \in Z} x_{ij} < |Z| \qquad \emptyset \subset Z \subset V$$

$$x_{ij} \in \{0, 1\}, \qquad i, j \in \{1, ..., n\}$$

The first set of constraints ensures that for each i exactly two variables corresponding to edges incident with i are chosen. Since each edge has two endpoints, this implies that exactly n variables are allowed to take the value 1. The second set of constraints consists of the subtour elimination constraints. Each of these states for a specific subset Z of V that the number of edges connecting vertices in Z has to be less than $|Z|$ thereby ruling out that these form a subtour. Unfortunately there are exponentially many of these constraints.

The given constraints determine the set of feasible solutions S. One obvious way of relaxing this to a set of potential solutions is to relax (i.e. discard) the subtour elimination constrains. The set of potential solutions P is then the family of all sets of subtours such that each i belongs to exactly one of the subtours in each set in the family, cf. Figure 9.4. In Section 9.1.1 another possibility is decribed, which in a Branch and Bound context turns out to be more appropriate.

A subproblem of a given symmetric TSP is constructed by deciding for a subset A of the edges of G that these must be *included* in the tour to be constructed, while for another subset B the edges are *excluded* from the tour. Exclusion of an edge (i, j) is usually modeled by setting c_{ij} to ∞, whereas the inclusion of an

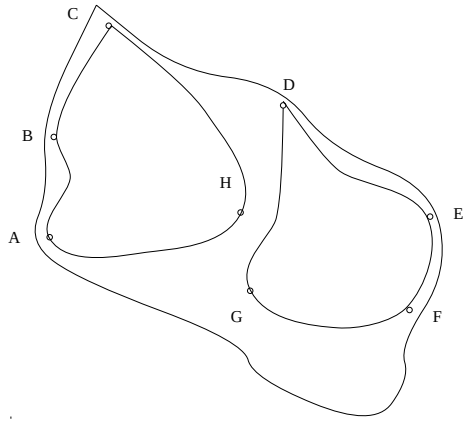


Figure 9.4: A potential, but not feasible solution to the biking tourist's problem

edge can be handled in various ways as e.g. graph contraction. The number of feasible solutions to the problem is $(n-1)!/2$, which for $n = 50$ is appr. 3×10^{62}
$\triangle$

The following descriptions follow [4, 5].

Example 2: The Graph Partitioning Problem. The Graph Partitioning problem arises in situations, where it is necessary to minimize the number (or weight of) connections between two parts of a network of prescribed size. We consider a given weighted, undirected graph G with vertex set V and edge set E , and a cost function $c : E \to \mathcal{N}$. The problem is to partition V into two disjoint subsets V_1 and V_2 of equal size such that the sum of costs of edges connecting vertices belonging to different subsets is as small as possible. Figure 9.5 shows an instance of the problem:

The graph partitioning problem can be formulated as a quadratic integer programming problem. Define for each vertex v of the given graph a variable x_v, which can attain only the values 0 and 1. A 1-1 correspondence between partitions and assignments of values to all variables now exists: $x_v = 1$ (respectively $= 0$) if and only if $v \in V_1$ (respectively $v \in V_2$). The cost of a partition is then

$$\sum_{v \in V_1, u \in V_2} c_{uv} x_v (1 - x_u).$$

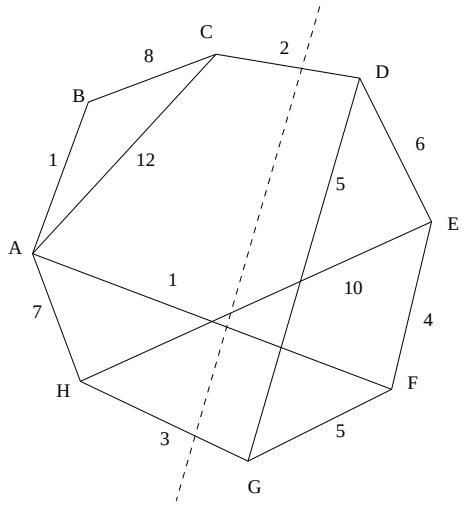


Figure 9.5: A graph partitioning problem and a feasible solution.

A constraint on the number of variables attaining the value 1 is included to exclude infeasible partitions.

$$\sum_{v \in V} x_v = |V|/2.$$

The set of feasible solutions S is here the partitions of V into two equal-sized subsets. The natural set P of potential solutions are all partitions of V into two non-empty subsets.

Initially V_1 and V_2 are empty corresponding to that no variables have yet been assigned a value. When some of the vertices have been assigned to the sets (the corresponding variables have been assigned values 1 or 0), a subproblem has been constructed.

The number of feasible solutions to a GPP with $2n$ vertices equals the binomial coefficient $C(2n, n)$. For $2n = 120$ the number of feasible solutions is appr. 9.6×10^{34}. △

Example 3: The Quadratic Assignment Problem. Here, I consider the Koopmans-Beckman version of the problem, which can informally be stated with reference to the following practical situation: A company is to decide the assignment of n of facilities to an equal number of locations and wants to minimize

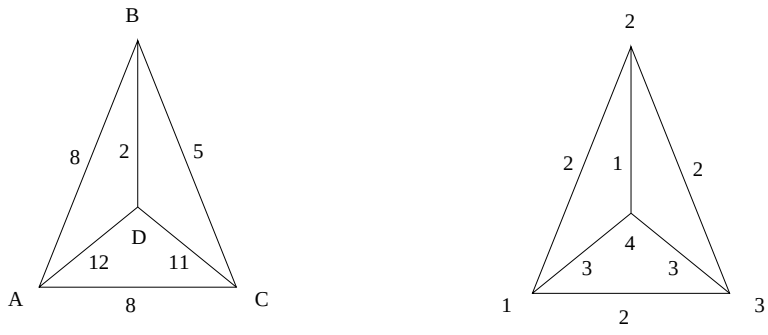


Figure 9.6: A Quadratic Assignment problem of size 4.

the total transportation cost. For each pair of facilities (i, j) a flow of communication $f_{i,j}$ is known, and for each pair of locations (l, k) the corresponding distance $d_{l,k}$ is known. The transportation cost between facilities i and j, given that i is assigned to location l and j is assigned to location k, is $f_{i,j} \cdot dl, k$, and the objective of the company is to find an assignment minimizing the sum of all transportation costs. Figure 9.6 shows a small example with 4 facilities and 4 locations. The assignment of facilities A,B,C, and D on sites 1,2,3, and 4 respectively has a cost of 224.

Each feasible solution corresponds to a permutation of the facilities, and letting S denote the group of permutations of n elements, the problem can hence formally be stated as

$$\min\{\sum_{i=1}^{n}\sum_{j=1}^{n} f_{i,j} \cdot d_{\pi(i),\pi(j)} : \pi \in S\}.$$

A set of potential solutions is e.g. obtained by allowing more than one facility on each location.

Initially no facilities have been placed on a location, and subproblems of the original problem arise when some but not all facilities have been assigned to locations.

Again the number of feasible solutions grows exponentially: For a problem with n facilities to be located, the number of feasible solutions is $n!$, which for $n = 20$ is appr. 2.43×10^{18}. △

The solution of a problem with a Branch and Bound algorithm is traditionally

described as a search through a search tree, in which the root node corresponds to the original problem to be solved, and each other node corresponds to a subproblem of the original problem. Given a node Q of the tree, the children of Q are subproblems derived from Q through imposing (usually) a single new constraint for each subproblem, and the descendants of Q are those subproblems, which satisfy the same constraints as Q and additionally a number of others. The leaves correspond to feasible solutions, and for all $\mathcal{NP}$-hard problems, instances exist with an exponential number of leaves in the search tree. To each node in the tree a *bounding function* g associates a real number called the *bound* for the node. For leaves the bound equals the value of the corresponding solution, whereas for internal nodes the value is a lower bound for the value of any solution in the subspace corresponding to the node. Usually g is required to satisfy the following three conditions:

1. $g(P_i) \leq f(P_i)$ for all nodes P_i in the tree
2. $g(P_i) = f(P_i)$ for all leaves in the tree
3. $g(P_i) \geq g(P_j)$ if P_j is the father of P_i

These state that g is a bounding function, which for any leaf agrees with the objective function, and which provides closer and closer (or rather not worse) bounds when more information in terms of extra constraints for a subproblem is added to the problem description.

The search tree is developed dynamically during the search and consists initially of only the root node. For many problems, a feasible solution to the problem is produced in advance using a heuristic, and the value hereof is used as the current best solution (called the *incumbent*). In each *iteration* of a Branch and Bound algorithm, a node is *selected* for exploration from the pool of *live* nodes corresponding to unexplored feasible subproblems using some selection strategy. If the eager strategy is used, a *branching* is performed: Two or more children of the node are constructed through the addition of constraints to the subproblem of the node. In this way the subspace is subdivided into smaller subspaces. For each of these the bound for the node is calculated, possibly with the result of finding the optimal solution to the subproblem, cf. below. In case the node corresponds to a feasible solution or the bound is the value of an optimal solution, the value hereof is compared to the incumbent, and the best solution and its value are kept. If the bound is no better than the incumbent, the subproblem is discarded (or *fathomed*), since no feasible solution of the subproblem can be better that the incumbent. In case no feasible solutions to the subproblem exist the subproblem is also fathomed. Otherwise the possibility of a better solution in the subproblem cannot be ruled out, and the node (with

the bound as part of the information stored) is then joined to the pool of live subproblems. If the lazy selection strategy is used, the order of bound calculation and branching is reversed, and the live nodes are stored with the bound of their father as part of the information. Below, the two algorithms are sketched.

Algorithm 18: Eager Branch and Bound

Data: specify
Result: specify

```
1  Incumbent ← ∞
2  LB(P_0) ← g(P_0)
3  Live ← {(P_0, LB(P_0))}
4  repeat
5      Select the node P from Live to be processed
6      Live ← Live \ {P}
7      Branch on P generating P_1, ...P_k
8      for 1 ≤ i ≤ k do
9          Bound P_i : LB(P_i) := g(P_i)
10         if LB(P_i) = f(X) for a feasible solution X and
           f(X) < Incumbent then
11             Incumbent ← f(X)
12             Solution ← X
13             Live ← ∅
14         if LB(P_i) ≥ Incumbent then
15             fathom P_i
16         else
17             Live := Live ∪ {(P_i, LB(P_i))}
18 until Live = ∅
19 OptimalSolution ← Solution
20 OptimalValue ← Incumbent
```

Algorithm 19: Lazy Branch and Bound

Data: specify
Result: specify

```
1  Incumbent ← ∞
2  Live ← {(P_0, LB(P_0))}
3  repeat
4  |  Select the node P from Live to be processed
5  |  Live ← Live \ {P}
6  |  Bound P
7  |  LB(P) ← g(P)
8  |  if LB(P) = f(X) for a feasible solution X and f(X) < Incumbent
   |  then
9  |  |  Incumbent ← f(X)
10 |  |  Solution ← X
11 |  |  Live ← ∅
12 |  if LB(P_i) ≥ Incumbent then
13 |  |  fathom P
14 |  else
15 |  |  Branch on P generating P_1, ...P_k
16 |  |  for 1 ≤ i ≤ k do
17 |  |  |  Live := Live ∪ {(P_i, LB(P_i))}
18 until Live = ∅
19 OptimalSolution ← Solution
20 OptimalValue ← Incumbent
```

A Branch and Bound algorithm for a minimization problem hence consists of three main components:

1. a *bounding function* providing for a given subspace of the solution space a lower bound for the best solution value obtainable in the subspace,
2. a *strategy for selecting* the live solution subspace to be investigated in the current iteration, and
3. a *branching* rule to be applied if a subspace after investigation cannot be discarded, hereby subdividing the subspace considered into two or more subspaces to be investigated in subsequent iterations.

In the following, I discuss each of these key components briefly.

In addition to these, an *initial good feasible solution* is normally produced using a heuristic whenever this is possible in order to facilitate fathoming of nodes as early as possible. If no such heuristic exists, the initial value of the incumbent is set to infinity. It should be noted that other methods to fathom solution subspaces exist, e.g. dominance tests, but these are normally rather problem specific and will not be discussed further here. For further reference see [8].

9.1.1 Bounding function

The bounding function is the key component of any Branch and Bound algorithm in the sense that a low quality bounding function cannot be compensated for through good choices of branching and selection strategies. Ideally the value of a bounding function for a given subproblem should equal the value of the best feasible solution to the problem, but since obtaining this value is usually in itself $\mathcal{NP}$-hard, the goal is to come as close as possible using only a limited amount of computational effort (i.e. in polynomial time), cf. the succeeding discussion. A bounding function is called *strong*, if it in general gives values close to the optimal value for the subproblem bounded, and *weak* if the values produced are far from the optimum. One often experiences a trade off between quality and time when dealing with bounding functions: The more time spent on calculating the bound, the better the bound value usually is. It is normally considered beneficial to use as strong a bounding function as possible in order to keep the size of the search tree as small as possible.

Bounding functions naturally arise in connection with the set of potential solutions P and the function g mentioned in Section 2. Due to the fact that $S \subseteq P$, and that $g(x) \leq f(x)$ on P, the following is easily seen to hold:

$$min_{x \in P}\ g(x) \leq \left\{ \begin{array}{l} min_{x \in P}\ f(x) \\ min_{x \in S}\ g(x) \end{array} \right\} \leq min_{x \in S}\ f(x)$$

If both of P and g exist there are now a choice between three optimization problems, for each of which the optimal solution will provide a lower bound for the given objective function. The "skill" here is of course to chose P and/or g so that one of these is easy to solve and provides tight bounds.

Hence there are two standard ways of converting the $\mathcal{NP}$-hard problem of solving a subproblem to optimality into a $\mathcal{P}$-problem of determining a lower bound for the objective function. The first is to use *relaxation* - leave out some of the constraints of the original problem thereby enlarging the set of feasible solutions. The objective function of the problem is maintained. This corresponds

to minimizing f over P. If the optimal solution to the relaxed subproblem satisfies all constraints of the original subproblem, it is also optimal for this, and is hence a candidate for a new incumbent. Otherwise, the value is a lower bound because the minimization is performed over a larger set of values than the objective function values for feasible solutions to the original problem. For e.g. GPP, a relaxation is to drop the constraint that the sizes of V_1 and V_2 are to be equal.

The other way of obtaining an easy bound calculation problem is to minimize g over S, i.e. to maintain the feasible region of the problem, but modify the objective function at the same time ensuring that for all feasible solutions the modified function has values less than or equal to the original function. Again one can be sure that a lower bound results from solving the modified problem to optimality, however, it is generally not true that the optimal solution corresponding to the modified objective function is optimal for the original objective function too. The most trivial and very weak bounding function for a given minimization problem obtained by modification is the sum of the cost incurred by the variable bindings leading to the subproblem to be bounded. Hence all feasible solutions for the subproblem are assigned the same value by the modified objective function. In GPP this corresponds to the cost on edges connecting vertices assigned to V_1 in the partial solution with vertices assigned to V_2 in the partial solution, and leaving out any evaluation of the possible costs between one assigned and one unassigned vertex, and costs between two assigned vertices. In QAP, an initial and very weak bound is the transportation cost between facilities already assigned to locations, leaving out the potential costs of transportation between one unassigned and one assigned, as well as between two unassigned facilities. Much better bounds can be obtained if these potential costs are included in the bound, cf. the Roucairol-Hansen bound for GPP and the Gilmore-Lawler bound for QAP as described e.g. in [4, 5].

Combining the two strategies for finding bounding functions means to minimize g over P, and at first glance this seems weaker than each of those. However, a parameterized family of lower bounds may result, and finding the parameter giving the optimal lower bound may after all create very tight bounds. Bounds calculated by so-called *Lagrangean relaxation* are based on this observation - these bounds are usually very tight but computationally demanding. The TSP provides a good example hereof.

Example 4: The 1-tree bound for symmetric TSP problems. As mentioned, one way of relaxing the constraints of a symmetric TSP is to allow subtours. However, the bounds produced this way are rather weak. One alternative is the 1-tree relaxation.

Here one first identifies a special vertex, "#1", which may be any vertex of the graph. "#1" and all edges incident to this are removed from G, and a minimum spanning tree T_{rest} is found for the remaining graph. Then the two shortest edges e_1, e_2 incident to "#1" are added to T_{rest} producing the 1-tree T_{one} of G with respect to "#1", cf. Figure 9.7.

The total cost of T_{one} is a lower bound of the value of an optimum tour. The argument for this is as follows: First note that a Hamilton tour in G consists of two edges e'_1, e'_2 and a tree T'_{rest} in the rest of G. Hence the set of Hamilton tours of G is a subset of the set of 1-trees of G. Since e_1, e_2 are the two shortest edges incident to "#1" and T_{rest} is the minimum spanning tree in the rest of G, the cost of T_{one} is less than or equal the cost of any Hamilton tour.

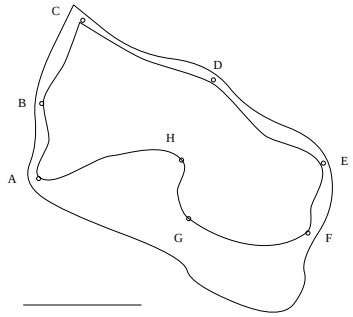
In case T_{one} is a tour, we have found the optimal solution to our subproblem – otherwise a vertex of degree at least 3 exists and we have to perform a branching.

The 1-tree bound can be strengthened using the idea of problem transformation: Generate a new symmetric TSP problem having the same optimal tour as the original, for which the 1-tree bound is tighter. The idea is that vertices of T_{one} with high degree are incident with too many attractive edges, whereas vertices of degree 1 have too many unattractive edges. Denote by π_i the degree of vertex i minus 2: $\pi_i := deg(v_i) - 2$. Note that the sum over V of the values π equals 0 since T_{one} has n edges, and hence the sum of $deg(v_i)$ equals $2n$. Now for each edge (i, j) we define the transformed cost c'_{ij} to be $c_{ij} + \pi_i + \pi_j$. Since each vertex in a Hamilton tour is incident to exactly two edges, the new cost of a Hamilton tour is equal to the current cost plus two times the sum over V of the values π. Since the latter is 0, the costs of all tours are unchanged, but the costs of 1-trees in general increase. Hence calculating the 1-tree bound for the transformed problem often gives a better bound, but not necessarily a 1-tree, which is a tour.

The trick may be repeated as many times as one wants, however, for large instances a tour seldomly results. Hence, there is a trade-off between time and strength of bound: should one branch or should one try to get an even stronger bound than the current one by a problem transformation ? Figure 9.7 (c) shows the first transformation for the problem of Figure 9.7 (b).

9.1.2 Strategy for selecting next subproblem

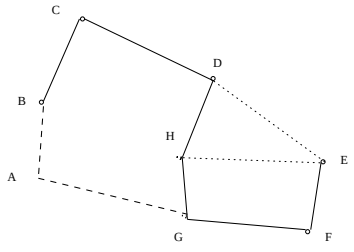
The strategy for selecting the next live subproblem to investigate usually reflects a trade off between keeping the number of explored nodes in the search tree low, and staying within the memory capacity of the computer used. If one always



Optimal tour, cost = 100

	A	B	C	D	E	F	G	H
A	0	11	24	25	30	29	15	15
B	11	0	13	20	32	37	17	17
C	24	13	0	16	30	39	29	22
D	25	20	16	0	15	23	18	12
E	30	32	30	15	0	9	23	15
F	29	37	39	23	9	0	14	21
G	15	17	29	18	23	14	0	7
H	15	17	22	12	15	21	7	0

(a)



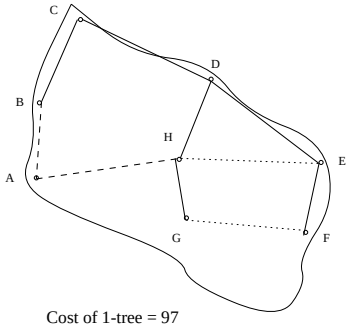
Tree in rest of G

Edge left out by Kruskal's MST algorithm

1-tree edge

Cost of 1-tree = 97

(b)



Cost of 1-tree = 97

Modified distance matrix:

	A	B	C	D	E	F	G	H
A	0	11	24	25	29	29	16	15
B	11	0	13	20	31	37	18	17
C	24	13	0	16	29	39	30	22
D	25	20	16	0	14	23	19	12
E	29	31	29	14	0	8	23	14
F	29	37	39	23	8	0	15	21
G	16	18	30	19	23	15	0	8
H	15	17	22	12	14	21	8	0

(c)

Figure 9.7: A bound calculation of the Branch and Bound algorithm for the symmetric TSP using the 1-tree bound with "#1" equal to A and Lagrangean relaxation for bounding.

selects among the live subproblems one of those with the lowest bound, called the *best first search strategy*, BeFS, no superfluous bound calculations take place after the optimal solution has been found. Figure 9.8 (a) shows a small search tree - the numbers in each node corresponds to the sequence, in which the nodes are processed when BeFS is used.

The explanation of the property regarding superfluos bound calculations lies in the concept of *critical* subproblems. A subproblem P is called *critical* if the given bounding function when applied to P results in a value strictly less than the optimal solution of the problem in question. Nodes in the search tree corresponding to critical subproblems have to be partitioned by the Branch and Bound algorithm no matter when the optimal solution is identified - they can never be discarded by means of the bounding function. Since the lower bound of any subspace containing an optimal solution must be less than or equal to the optimum value, only nodes of the search tree with lower bound less than or equal to this will be explored. After the optimal value has been discovered only critical nodes will be processed in order to prove optimality. The preceding argument for optimality of BeFS with respect to number of nodes processed is valid only if eager node evaluation is used since the selection of nodes is otherwise based on the bound value of the father of each node. BeFS may, however, also be used in combination with lazy node evaluation.

Even though the choice of the subproblem with the current lowest lower bound makes good sense also regarding the possibility of producing a good feasible solution, memory problems arise if the number of critical subproblems of a given problem becomes too large. The situation more or less corresponds to a *breath first search* strategy, in which all nodes at one level of the search tree are processed before any node at a higher level. Figure 9.8 (b) shows the search tree with the numbers in each node corresponding to the BFS processing sequence. The number of nodes at each level of the search tree grows exponentially with the level making it infeasible to do breadth first search for larger problems. For GPP sparse problems with 120 vertices often produce in the order of a few hundred of critical subproblems when the Roucairol-Hansen bounding function is used [4], and hence BeFS seems feasible. For QAP the famous Nugent20 problem [13] produces 3.6×10^8 critical nodes using Gilmore-Lawler bounding combined with detection of symmetric solutions [5], and hence memory problems may be expected if BeFS is used.

The alternative used is *depth first search*, DFS. Here a live node with largest level in the search tree is chosen for exploration. Figure 9.8 (c) shows the DFS processing sequence number of the nodes. The memory requirement in terms of number of subproblems to store at the same time is now bounded above by the number of levels in the search tree multiplied by the maximum number of children of any node, which is usually a quite manageable number.

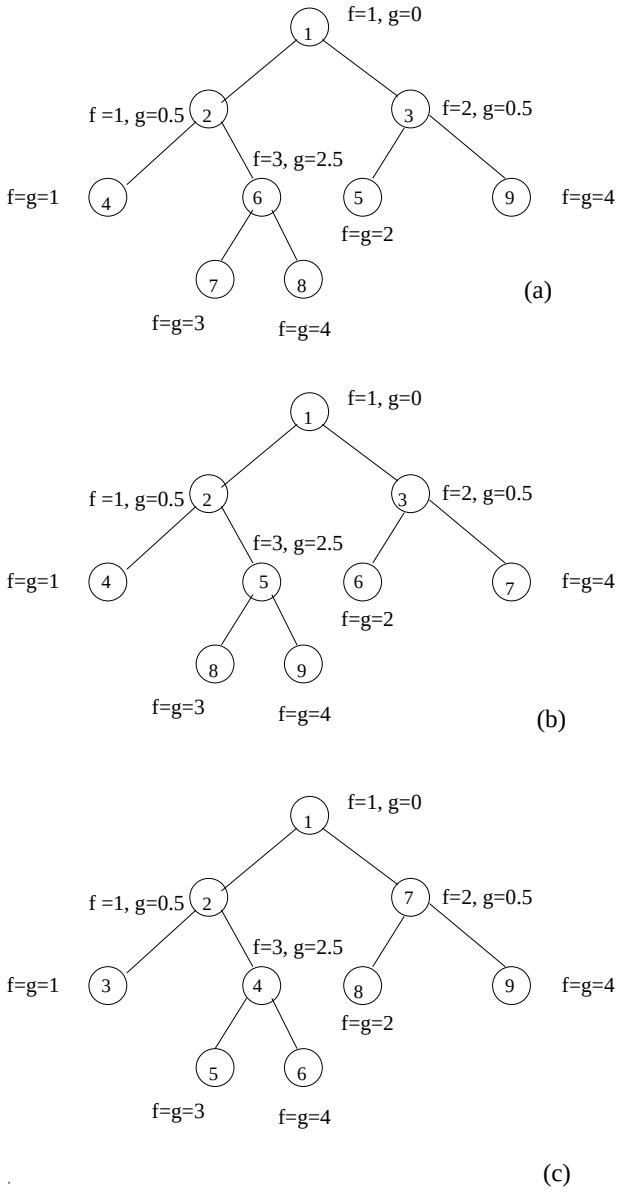


Figure 9.8: Search strategies in Branch and Bound: (a) Best First Search, (b) Breadth First Search, and (c) Depth First Search.

DFS can be used both with lazy and eager node evaluation. An advantage from the programming point of view is the use of recursion to search the tree - this enables one to store the information about the current subproblem in an incremental way, so only the constraints added in connection with the creation of each subproblem need to be stored. The drawback is that if the incumbent is far from the optimal solution, large amounts of unnecessary bounding computations may take place.

In order to avoid this, DFS is often combined with a selection strategy, in which one of the branches of the selected node has a very small lower bound and the other a very large one. The idea is that exploring the node with the small lower bound first hopefully leads to a good feasible solution, which when the procedure returns to the node with the large lower bound can be used to fathom the node. The node selected for branching is chosen as the one, for which the difference between the lower bounds of its children is as large as possible. Note however that this strategy requires the bound values for children to be known, which again may lead to superfluous calculations.

A combination of DFS as the overall principle and BeFS when choice is to be made between nodes at the same level of the tree is also quite common.

In [2] an experimental comparison of BeFS and DFS combined with both eager and lazy node evaluation is performed for QAP. Surprisingly, DFS is superior to BeFS in all cases, both in terms of time and in terms of number of bound calculations. The reason turns out to be that in practice, the bounding and branching of the basic algorithm is extended with additional tests and calculations at each node in order to enhance efficiency of the algorithm. Hence, the theoretical superiority of BeFS should be taken with a grain of salt.

9.1.3 Branching rule

All branching rules in the context of Branch and Bound can bee seen as subdivision of a part of the search space through the addition of constraints, often in the form of assigning values to variables. If the subspace in question is subdivided into two, the term *dichotomic* branching is used, otherwise one talks about *polytomic* branching. Convergence of Branch and Bound is ensured if the size of each generated subproblem is smaller than the original problem, and the number of feasible solutions to the original problem is finite. Normally, the subproblems generated are disjoint - in this way the problem of the same feasible solution appearing in different subspaces of the search tree is avoided.

For GPP branching is usually performed by choosing a vertex not yet assigned

to any of V_1 and V_2 and assigning it to V_1 in one of the new subproblems (corresponding to that the variable of the node receives the value 1) and to V_2 in the other (variable value equal to 0). This branching scheme is dichotomic, and the subspaces generated are disjoint.

In case of QAP, an unassigned facility is chosen, and a new subproblem is created for each of the free location by assigning the chosen facility to the location. The scheme is called branching on facilities and is polytomic, and also here the subspaces are disjoint. Also branching on locations is possible.

For TSP branching may be performed based on the 1-tree generated during bounding. If all vertices have degree 2 the 1-tree is a tour, and hence an optimal solution to the subproblem. Then no further branching is required. If a node has degree 3 or more in the 1-tree, any such node may be chosen as the source of branching. For the chosen node, a number of subproblems equal to the degree is generated. In each of these one of the edges of the 1-tree is excluded from the graph of the subproblem ruling out the possibility that the bound calculation will result in the same 1-tree. Figure 9.9 shows the branching taking place after the bounding in Figure 9.7. The bound does, however, not necessarily change, and identical subproblems may arise after a number of branchings. The effect of the latter is not an incorrect algorithm, but a less efficient algorithm. The problem is further discussed as an exercise.

9.1.4 Producing an initial solution

Although often not explicitly mentioned, another key issue in the solution of large combinatorial optimization problems by Branch and Bound is the construction of a good initial feasible solution. Any heuristic may be used, and presently a number of very good general heuristics as well as a wealth of very problem specific heuristics are available. Among the general ones (also called meta-heuristics or paradigms for heuristics), Simulated Annealing, Genetic Algorithms, and Tabu Search are the most popular.

As mentioned, the number of subproblems explored when the DFS strategy for selection is used depends on the quality of the initial solution - if the heuristic identifies the optimal solution so that the Branch and Bound algorithm essentially verifies the optimality, then even DFS will only explore critical subproblems. If BeFS is used, the value of a good initial solution is less obvious.

Regarding the three examples, a good and fast heuristic for GPP is the Kernighan-Lin variable depth local search heuristic. For QAP and TSP, very good results have been obtained with Simulated Annealing.

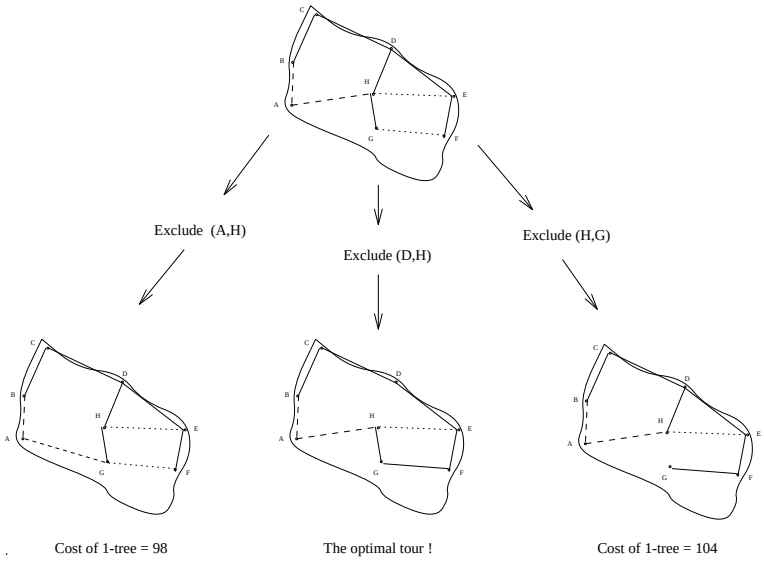


Figure 9.9: Branching from a 1-tree in a Branch and Bound algorithm for the symmetric TSP.

9.2 Personal Experiences with GPP and QAP

The following subsections briefly describe my personal experiences using Branch and Bound combined with parallel processing to solve GPP and QAP. Most of the material stems from [3, 4] and [5]. Even though parallelism is not an integral part of Branch and Bound, I have chosen to present the material, since the key components of the Branch and Bound are unchanged. A few concepts from parallel processing is, however, necessary.

Using parallel processing of the nodes in the search tree of a Branch and Bound algorithm is a natural idea, since the bound calculation and the branching in each node is independent. The aim of the parallel procesing is to speed up the execution time of the algorithm, To measure the success in this aspect, the *speed-up* of adding processors is measured. The relative speed-up using p processors is defined to be the processing time $T(1)$ using one processor divided by the processing time $T(p)$ using p processors:

$$S(p) = T(1)/T(p)$$

The ideal value of $S(p)$ is p - then the problem is solved p times faster with p processors than with 1 processor.

An important issue in parallel Branch and Bound is distribution of work: in order to obtain as short running time as possible, no processor should be idle at any time during the computation. If a distributed system or a network of workstations is used, this issue becomes particularly crucial since it is not possible to maintain a central pool of live subproblems. Various possibilities for *load balancing schemes* exist - two concrete examples are given in the following, but additional ones are described in [7].

9.2.1 Solving the Graph Partitioning Problem in Parallel

GPP was my first experience with parallel Branch and Bound, and we implemented two parallel algorithms for the problem in order to investigate the trade off between bound quality and time to calculate the bound. One - called the CT-algorithm - uses an easily computable bounding function based on the principle of modified objective function and produces bounds of acceptable quality, whereas the other - the RH-algorithm - is based on Lagrangean relaxation and has a bounding function giving tight, but computationally expensive bounds.

The system used was a 32 processor IPSC1/d5 hypercube equipped with Intel 80286 processors and 80287 co-processors each with 512 KB memory. No dedicated communication processors were present, and the communication facilities were Ethernet connections implying a large start-up cost on any communication.

Both algorithms were of the distributed type, where the pool of live subproblems is distributed over all processors, and as strategy for distributing the workload we used a combined "on demand"/"on overload" strategy. The "on overload" strategy is based on the idea that if a processor has more than a given threshold of live subproblems, a number of these are sent to neighbouring processors. However, care must be take to ensure that the system is not floated with communication and that flow of subproblems between processors takes place during the entire solution process. The scheme is illustrated in Figure 9.10.

Regarding termination, the algorithm of Dijkstra et. al. [6] was used. The selection strategy for next subproblem were BeFs for the RH-algorithm and DFS for the CT-algorithm. The first feasible solution was generated by the Kernighan-Lin heuristic, and its value was usually close to the optimal solution value.

For the CT-algorithm, results regarding processor utilization and relative speed-

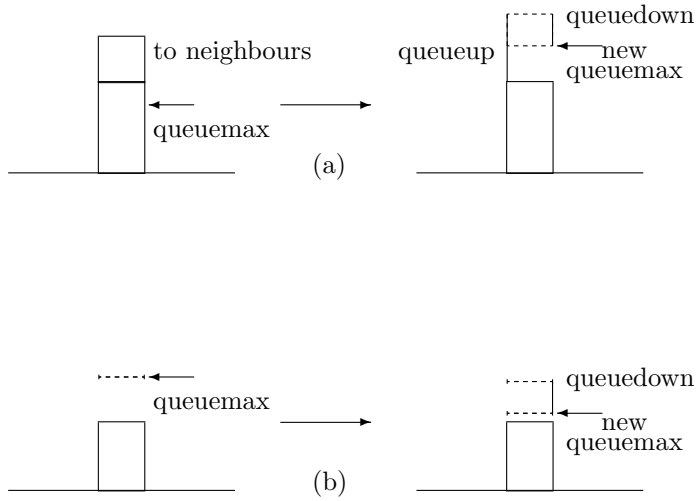


Figure 9.10: Illustration of the on overload protocol. (a) is the situation, when a processor when checking finds that it is overloaded, and (b) shows the behaviour of a non-overloaded processor

No. of proc.		4	8	16	32
CT	time (sec)	1964	805	421	294
	proc. util. (%)	97	96	93	93
	no. of bound calc.	449123	360791	368923	522817
RH	time (sec)	1533	1457	1252	1219
	proc. util. (%)	89	76	61	42
	no. of bound calc.	377	681	990	1498

Table 9.1: Comparison between the CT- and RH-algorithm on a 70 vertex problem with respect to running times, processor utilization, and number of subproblems solved.

up were promising. For large problems, a processor utilization near 100% was observed, and linear speed-up close to the ideal were observed for problems solvable also on a single processor. Finally we observed that the best speed-up was observed for problems with long running times. The RH-algorithm behaved differently – for small to medium size problems, the algorithm was clearly inferior to the CT-algorithm, both with respects to running time, relative speed-up and processor utilization. Hence the tight bounds did not pay off for small problems – they resulted idle processors and long running times.

We continued to larger problems expecting the problem to disappear, and Figure

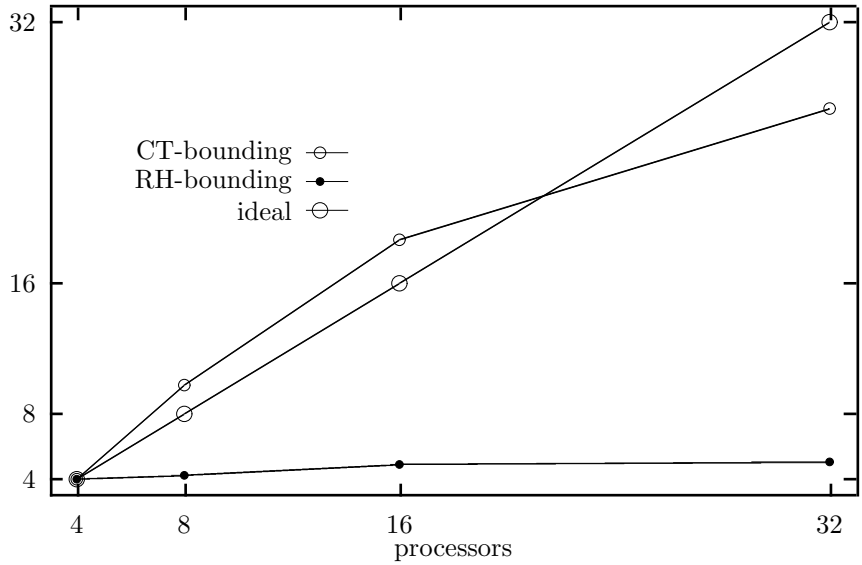


Figure 9.11: Relative speed-up for the CT-algorithm and the RH-algorithm for a 70 vertex problem.

9.11 and Table 9.1 shows the results for a 70-vertex problem for the CT- and RH-algorithms. We found that the situation did by no means improve. For the RH method it seemed impossible to use more than 4 processors. The explanation was found in the number of critical subproblems generated, cf. Table 9.2. Here it is obvious that using more processors for the RH-method just results in a lot of superfluous subproblems being solved, which does not decrease the total solution time.

9.2.2 Solving the QAP in Parallel

QAP is one of my latest parallel Branch and Bound experiences. The aim of the research was in this case to solve the previously unsolved benchmark problem Nugent20 to optimality using a combination of the most advanced bounding functions and fast parallel hardware, as well as any other trick we could find and think of.

We used a MEIKO system consisting of 16 Intel i860 processors each with 16 MB of internal memory. Each processor has a peak performance of 30 MIPS when doing integer calculation giving an overall peak performance of approxi-

	CT-algorithm			RH-algorithm		
No. Vert.	Cr. subpr.	B. calc.	Sec.	Cr. subpr.	B. calc.	Sec.
30	103	234	1	4	91	49
40	441	803	2	11	150	114
50	2215	3251	5	15	453	278
60	6594	11759	18	8	419	531
70	56714	171840	188	26	1757	1143
80	526267	901139	931	19	2340	1315
100	2313868	5100293	8825	75	3664	3462
110	8469580	15203426	34754	44	3895	4311
120	–	–	–	47	4244	5756

Table 9.2: Number of critical subproblems and bound calculations as a function of problem size.

mately 500 MIPS for the complete system. The performance of each single i860 processor almost matches the performance of the much more powerful Cray 2 on integer calculations, indicating that the system is very powerful.

The processors each have two Inmos T800 transputers as communication processors. Each transputer has 4 communication channels each with bandwidth 1.4 Mb/second and start-up latency 340 μs. The connections are software programmable, and the software supports point-to-point communication between any pair of processors. Both synchronous and asynchronous communication are possible, and also both blocking and non-blocking communication exist.

The basic framework for testing bounding functions was a distributed Branch and Bound algorithm with the processors organized as a ring. Workload distribution was kept simple and based on local synchronization. Each processor in the ring communicates with each of its neighbours at certain intervals. At each communication the processors exchange information on the respective sizes of subproblem pools, and based here-on, subproblems are sent between the processors. The speed-up obtained with this scheme was 13.7 for a moderately sized problem with a sequential running time of 1482 seconds and a parallel running time with 16 processors of 108 seconds.

The selection strategy used was a kind of breadth first search. The feasibility hereof is intimately related to the use of a very good heuristic to generate the incumbent. We used simulated annealing, and as reported in [5], spending less than one percent of the total running time in the heuristic enabled us to start the parallel solution with the optimal solution as the incumbent. Hence only critical subproblems were solved. Regarding termination detection, a tailored algorithm were used for this purpose.

	Mautor & Roucairol		Fac. br. w. symmetry	
Problem	No. nodes.	Time (s)	No. nodes.	Time (s)
Nugent 15	97286	121	105773	10
Nugent 16.2	735353	969	320556	34
Nugent 17	–	–	24763611	2936
Nugent 18	–	–	114948381	14777
Nugent 20	–	–	360148026	57963
Elshafei 19	575	1.4	471	0.5
Armour & Buffa 20	531997	1189	504452	111

Table 9.3: Result obtained by the present authors in solving large standard benchmark QAPs. Results obtained by Mautor and Roucairol is included for comparison.

Problem	Dynamic dist.	Init. subpr. per proc.	Static dist.
Nugent 8	0.040	1	0.026
Nugent 10	0.079	1	0.060
Nugent 12	0.328	6	0.381
Nugent 14	12.792	24	13.112
Nugent 15	10.510	41	11.746
Nugent 16	35.293	66	38.925

Table 9.4: Result obtained when solving standard benchmark QAPs using static workload distribution. Results obtained with dynamic distribution are included for comparison.

The main results of the research are indicated in Table 9.3. We managed to solve previously unsolved problems, and for problems solved by other researchers, the results clearly indicated the value of choosing an appropriate parallel system for the algorithm in question.

To get an indication of the efficiency of so-called static workload distribution in our application, an algorithm with static workload distribution was also tested. The results appear in Table 9.4. The subproblems distributed to each processor were generated using BeFS sequential Branch and Bound until the pool of live subproblems were sufficiently large that each processors could get the required number of subproblems. Hence all processors receive equally promising subproblems. The optimal number of subproblems pr. processors were determined experimentally and equals roughly $(p-8)^4/100$, where p is the number of processors.

9.3 Ideas and Pitfalls for Branch and Bound users.

Rather than giving a conclusion, I will in the following try to equip new users of Branch and Bound - both sequential and parallel - with a checklist corresponding to my own experiences. Some of the points of the list have already been mentioned in the preceding sections, while some are new.

- The importance of finding a good initial incumbent cannot be overestimated, and the time used for finding such one is often only few percentages of the total running time of the algorithm.
- In case an initial solution very close to the optimum is expected to be known, the choice of node selection strategy and processing strategy makes little difference.
- With a difference of more than few percent between the value of the initial solution and the optimum the theoretically superior BeFS Branch and Bound shows inferior performance compared to both lazy and eager DFS Branch and Bound. This is in particular true if the pure Branch and Bound scheme is supplemented with problem specific efficiency enhancing test for e.g. supplementary exclusion of subspaces, and if the branching performed depends on the value of the current best solution.

9.4 Supplementary Notes

9.5 Exercises

1. Finish the solution of the biking tourist's problem on Bornholm.
2. Give an example showing that the branching rule illustrated in Figure 9.9 may produce nodes in the search tree with non-disjoint sets of feasible solutions. Devise a branching rule, which ensures that all subspaces generated are disjoint.
3. Solve the symmetric TSP instance with $n = 5$ and distance matrix

$$(c_e) = \begin{pmatrix} - & 10 & 2 & 4 & 6 \\ - & - & 9 & 3 & 1 \\ - & - & - & 5 & 6 \\ - & - & - & - & 2 \end{pmatrix}$$

by Branch and Bound using the 1-tree relaxation to obtain lower bounds.

4. Solve the asymmetric TSP instance with $n = 4$ and distance matrix

$$(c_{ij}) = \begin{pmatrix} - & 6 & 4 & 17 \\ 12 & - & 7 & 20 \\ 19 & 9 & - & 14 \\ 19 & 19 & 5 & - \end{pmatrix}$$

by Branch and Bound using an assignment relaxation to obtain bounds.

5. Consider the GPP as described in Example 1. By including the term

$$\lambda \, (\sum_{v \in V} x_v - |V|/2)$$

in the objective function, a relaxed unconstrained problem with modified objective function results for any λ. Prove that the new objective is less than or equal to the original on the set of feasible solutions for any λ. Formulate the problem of finding the optimal value of λ as an optimization problem.

6. A node in a Branch and Bound search tree is called *semi-critical* if the corresponding bound value is less than or equal to the optimal solution of the problem. Prove that if the number of semi-critical nodes in the search tree corresponding to a Branch and Bound algorithm for a given problem is polynomially bounded, then the problem belongs to $\mathcal{P}$.

   Prove that this holds also with the weaker condition that the number of critical nodes is polynomially bounded.

7. Consider again the QAP as described in Example 3. The simplest bound calculation scheme is described in Section 2.1. A more advanced, though still simple, scheme is the following:

   Consider now partial solution in which m of the facilities has been assigned to m of the locations. The total cost of any feasible solution in the subspace determined by a partial solution consists of three terms: costs for pairs of assigned facilities, costs for pairs consisting of one assigned and one unassigned facility, and costs for pairs of two unassigned facilities. The first term can be calculated exactly. Bounds for each of the two other terms can be found based on the fact that a lower bound for a scalar product $(a_1, ..., a_p) \cdot (b_{\pi(1)}, ..., b_{\pi(p)})$, where a and b are given vectors of dimension p and π is a permutation of $\{1, ..., p\}$, is obtained by multiplying the largest element in a with the smallest elements in b, the next-largest in a with the next-smallest in b etc.

For each assigned facility, the flows to unassigned facilities are ordered decreasingly and the distances from the location of the facility to the remaining free locations are ordered increasingly. The scalar product is now a lower bound for the communication cost from the facility to the remaining unassigned facilities.

The total transportation cost between unassigned facilities can be bounded in a similar fashion.

(a)

Consider the instance given in Figure 6. Find the optimal solution to the instance using the bounding method described above.

(b)

Consider now the QAP, where the distances between locations are given as the rectangular distances in the following grid:

1	2	3
4	5	6

The flows between pairs of facilities are given by

$$F = \begin{pmatrix} 0 & 20 & 0 & 15 & 0 & 1 \\ 20 & 0 & 20 & 0 & 30 & 2 \\ 0 & 20 & 0 & 2 & 0 & 10 \\ 15 & 0 & 2 & 0 & 15 & 2 \\ 0 & 30 & 0 & 15 & 0 & 30 \\ 1 & 2 & 10 & 2 & 30 & 0 \end{pmatrix}$$

Solve the problem using Branch and Bound with the bounding function described above, the branching strategy described in text, and DFS as search strategy.

To generate a first incumbent, any feasible solution can be used. Try prior to the Branch and Bound execution to identify a good feasible solution. A solution with value 314 exists.

8. Consider the 0-1 knapsack problem:

$$\begin{array}{ll} \max & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} & \sum_{j=1}^{n} a_j x_j \leq b \\ & x \in B^n \end{array}$$

with $a_j, c_j > 0$ for $j = 1, 2, \ldots, n$.

(a) Show that if the items are sorted in relation to their cost/weight-ratio then the LP relaxation can be established as $x_j = 1$ for $j = 1, 2, \ldots, r-1$, $x_r = (b - \sum_{j=1}^{r-1} a_j)/a_r$ and $x_j = 0$ for $j > r$ where r is defined by such that $\sum_{j=1}^{r-1} a_j \leq b$ and $\sum_{j=1}^{r} a_j > b$.

(b) Solve the instance

$$\begin{array}{ll} \max & 17x_1 + 10x_2 + 25x_3 + 17x_4 \\ & 5x_1 + 3x_2 + 8x_3 + 7x_4 \leq 12 \\ & x \in B^4 \end{array}$$

by Branch and Bound.

Bibliography

[1] A. de Bruin, A. H. G. Rinnooy Kan and H. Trienekens, "A Simulation Tool for the Performance of Parallel Branch and Bound Algorithms", *Math. Prog.* **42** (1988), p. 245 - 271.

[2] J. Clausen and M. Perregaard, "On the Best Search Strategy in Parallel Branch-and-Bound - Best-First-Search vs. Lazy Depth-First-Search", Proceedings of POC96 (1996), also DIKU Report 96/14, 11 p.

[3] J. Clausen, J. L. Träff, "Implementation of parallel Branch-and-Bound algorithms - experiences with the graph partitioning problem", *Annals of Oper. Res.* **33** (1991) 331 - 349.

[4] J. Clausen and J. L. Träff, "Do Inherently Sequential Branch-and-Bound Algorithms Exist ?", *Parallel Processing Letters* **4, 1-2** (1994), p. 3 - 13.

[5] J. Clausen and M. Perregaard, "Solving Large Quadratic Assignment Problems in Parallel", DIKU report 1994/22, 14 p., to appear in *Computational Optimization and Applications.*

[6] E. W. Dijkstra, W. H. J. Feijen and A. J. M. van Gasteren, "Derivation of a termination detection algorithm for distributed computations", *Inf. Proc. Lett.* **16** (1983), 217 - 219.

[7] B. Gendron and T. G. Cranic, "Parallel Branch-and-Bound Algorithms: Survey and Synthesis", *Operations Research* **42 (6)** (1994), p. 1042 - 1066.

[8] T. Ibaraki, "Enumerative Approaches to Combinatorial Optimization", *Annals of Operations Research* vol. **10, 11**, J.C.Baltzer 1987.

[9] P. S. Laursen, "Simple approaches to parallel Branch and Bound", *Parallel Computing* **19** (1993), p. 143 - 152.

[10] P. S. Laursen, "Parallel Optimization Algorithms - Efficiency vs. simplicity", Ph.D.-thesis, DIKU-Report 94/31 (1994), Dept. of Comp. Science, Univ. of Copenhagen.

[11] E. L. Lawler, J. K. Lenstra, A. H. G. .Rinnooy Kan, D. B. Shmoys (ed.), "The Travelling Salesman: A Guided Tour of Combinatorial Optimization", John Wiley 1985.

[12] T. Mautor, C. Roucairol, "A new exact algorithm for the solution of quadratic assignment problems", *Discrete Applied Mathematics* **55** (1994) 281-293.

[13] C. Nugent, T. Vollmann, J. Ruml, "An experimental comparison of techniques for the assignment of facilities to locations", *Oper. Res.* **16** (1968), p. 150 - 173.

[14] M. Perregaard and J. Clausen , "Solving Large Job Shop Scheduling Problems in Parallel", DIKU report 94/35, to appear in Annals of OR.

[15] C. Schütt and J. Clausen, "Parallel Algorithms for the Assignment Problem - Experimental Evaluation of Three Distributed Algorithms", *AMS DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **22** (1995), p. 337 - 351.

Appendix A

A quick guide to GAMS – IMP

To login at IMM from the GBAR is easy:

1. Login at your favorite xterminal at the GBAR.
2. Start a xterm. (Click on the middle mouse-buttom, select *terminals* and then *terminal*).
3. In that xterm write:

   - `ssh -l nh?? serv1.imm.dtu.dk`

   where nh?? correspond to your course login name. You will be prompted for a password.

   Alternatives to `serv1.imm.dtu.dk` are

   - `serv2.imm.dtu.dk`
   - `serv3.imm.dtu.dk`
   - `sunfire.imm.dtu.dk`

You now have a shell-window working at IMM. Remember to change password at IMM with the

```
passwd
```

command.

To write a file at IMM start an emacs editor at IMM by writing (in the xterm working at IMM):

```
emacs &
```

To start cplex write (in the xterm working at IMM):

```
cplex64
```

You can start more xterms to work at IMM in the same way.

CPLEX can be run in *interactive* mode or used as a library callable from e.g. C, C++ or Java programs. In this course we will only use CPLEX in interactive mode.

To start a CPLEX session type `cplex` and press the enter-key. You will then get an introductory text something like:

```
ILOG CPLEX 9.000, licensed to "university-lyngby", options: e m b q

Welcome to CPLEX Interactive Optimizer 9.0.0
  with Simplex, Mixed Integer & Barrier Optimizers
Copyright (c) ILOG 1997-2003
CPLEX is a registered trademark of ILOG

Type 'help' for a list of available commands.
Type 'help' followed by a command name for more
information on commands.

CPLEX>
```

To end the session type `quit` and press the enter-key. Please do not quit CPLEX by closing the xterm window before having exited CPLEX. At IMM we have 20 licenses, so at most 20 persons can run CPLEX independently of each other. If

you quit by closing the xterm window before leaving CPLEX in a proper manner it will take the license manager some time to recover the CPLEX license.

CPLEX solves linear and integer programming problems. These must be entered either through the built-in editor of CPLEX or by entering the problem in "your favorite editor", saving it to a file, and then reading the problem into CPLEX by typing

`read <filename.lp>`.

The file has to have extension `.lp` and the contents has to be in the lp-format. A small example is shown below.

```
\Problem
name: soejle.lp

Minimize
 obj: x1 + x2 + x3 + x4
Subject To
 c1: x1 + 2 x3 + 4 x4 >= 6
 c2: x2 + x3 >= 3
End
```

The first line is a remark and it stretches the entire line. `Subject to` can be replaced with `st`. The text written in the start of each line containing the objective function or constraints is optional, so `obj:` can be omitted.

After having entered a problem, it can be solved by giving the command `optimize` at the `CPLEX>` prompt and press enter. To see the result, the command `display solution variabl` and press enter is used, "-" indicating that the values of all variables are to be displayed.

CPLEX writes a log-file, which records the events of the session. An example of the log-file corresponding to the solution of the example above is shown below. The events of the session has been:

```
cplex <enter>
read soejle.lp <enter>
optimize <enter>
display solution variables - <enter>
quit <enter>
```

and the resulting log-file looks like:

```
Log started (V6.5.1) Tue Feb 15 10:24:58 2000


Problem 'soejle.lp' read.
Read time =    0.00 sec.
Tried aggregator 1 time.
LP Presolve eliminated 0 rows and 1 columns.
Reduced LP has 2 rows, 3 columns, and 4 nonzeros.
Presolve time =    0.00 sec.

Iteration log . . .
Iteration:     1    Infeasibility =             3.000000
Switched to devex.
Iteration:     3    Objective     =             3.000000

Primal - Optimal:  Objective =    3.0000000000e+00
Solution time =    0.00 sec.  Iterations = 3 (2)


Variable Name           Solution Value
x3                            3.000000
All other variables in the range 1-4 are zero.
```

Now let us take our initial problem and assume that we want `x1` and `x2` to be integer variables between 0 and 10. That the variables are non-negative are implicitely assumed by CPLEX, but we need to state the upper bound and the integrality condition. In this case our program will look like:

```
\Problem
name: soejle.lp

Minimize
 obj: x1 + x2 + x3 + x4
Subject To
 c1: x1 + 2 x3 + 4 x4 >= 6
 c2: x2 + x3 >= 3
Bounds
 x1<=10
 x2<=10
Integer
```

```
 x1
 x2
End
```

`Bounds` is used to declare bounds on variables, and the section afterwards, `Integer` states that `x1` and `x2` must be integer solutions. The bounds section must be placed before the section declaring the integer variables. It does not seem intuitive nevertheless if you do not state a bounds part **CPLEX will assume the integer variables to be binary**. If you want the integer variable to have no upper bound you can `x2<=INF` in the bounds section.

The command `help` shows the possible commands in the current situation. Also, CPLEX provides help if the current command is not sufficient to uniquely to determine an action. As an example, if one types `display` CPLEX will respond with listing the options and the question "Display what ?" CPLEX also offers possibilities to change parameters in a problem already entered - these possibilities may be investigated by entering `help` as the first command after having entered CPLEX.

Appendix B

Extra Assignments

Question 1

You have just been employed as a junior OR consultant in the consulting form **Optimal Solutions**. The first task that lands on your desk is a preliminary study for a big manufacturing company **Rubber Duck**. The company has three factories F1, F2 and F3 and four warehouses W1, W2, W3 and W4. In the current setup of their supply chain products are transported from a factory to a warehouse via a **cross-dock**. A cross-dock is a special form of warehouse where goods are only stored for a very short time. Goods arrive at a cross-dock, is then possibly repacked and then put on trucks for their destination. It is used in order to consolidate the transportation of goods. Rubber Duck uses two cross-docks, CD1 and CD2, in their operations.

The availability of trucks defines an upper limit on how many truck loads we can transport from a factory to a cross-dock and from a cross-dock to a warehouse. Table B.1 show the transportation capacities measured in truck loads from factory to cross-dock, and from cross-dock to warehouse.

	F1	F2	F3	
CD1	34	28	27	
CD2	40	40	22	
	W1	W2	W3	W4
CD1	30	24	22	—
CD2	—	26	12	58

Table B.1: Capacities in the Rubber Duck supply chain measured in truck loads. Note that it is not possible to supply warehouse W4 from cross-dock CD1 and warehouse W1 from cross-dock CD2.

Question 1.1

We want to determine how much Rubber Duck at most can get through their supply chain from the factories to the warehouses. Therefore give a maximum flow formulation of it. Draw a graph showing the problem as a maximum flow problem. Your network should have 11 vertices: a source (named 0), a vertex for each factory, cross-dock and warehouse, and a sink (named 11). The graph must contain all information necessary when formulation a maximum flow problem.

Question 1.2

Solve the maximum flow problem using the augmenting path algorithm. Show the flow graphically on a figure of the network. Write the value of the maximum flow and list each of the augmenting paths. Find a minimum cut and list the vertices in the minimum cut.

Question 1.3

Just as you are about to return a small report on your findings to your manager a phone call from a planner from Rubber Ducks logistics department reveals some new information. It turns out that there is a limit on how much each of the cross-docks can handle. This can be viewed as an extension to the maximum flow model. Let c_v be the maximum number of truck loads which can pass through an internal vertex v. Given a flow x_{uv} from vertex u to vertex v, the flow capacity of vertex v can be expressed as $\sum_{(u,v)\in E} x_{uv} \leq c_v$.

Describe how to modify a network so that this new extension can be handled as a maximum flow problem in a modified network.

Given that the capacity of cross-dock CD1 is 72 truck loads and of CD2 is 61 truck loads introduce the modifications and solve the new maximum flow problem. Show the flow graphically on a figure of the network. Write the value of the maximum flow and list each of the augmenting paths. Find a minimum cut and list the vertices in the minimum cut.

Question 2

Your success on the first assignment quickly lands you another case. A medium sized company with branches geographically distributed all over Denmark has approached Optimal Solutions. The company you are going to work for has decided to build an intra-net based on communication capacity leased from a large telecommunication vendor, Yellow. This operator has a back-bone network connecting a number of locations, and you can lease communication capacity between individual pairs of points in the back-bone network and in that way build your own network. Fortunately, all branches are located in one of the locations of the back-bone network, however, not all locations house a branch of the company.

The pricing policy of Yellow is as follows: Each customer individually buys in on a set connections chosen by the customer. The price is calculated as the sum of the prices for the individual connections. Each customer is offered a "customized" price for each connections, and these prices (although usually positive) may be negative to reflect a discount given by Yellow.

Figure B.1 shows a small example, where 3 branches have to be connected in a network with 6 locations.

Your task is to analyze the situation of the company and determine which connections to lease in order to establish the cheapest intranet. Yellow guarantees that the connections will always be operative. You do therefore not have to consider failure protection.

Question 2.1

Formulate a general mathematical programming model for the optimization problem of establishing the cheapest intra-net, given that all costs are non-negative. The model should have a decision variable for each connection, and a set of constraints expressing that for each partition of the set of locations into two non-empty subsets with a branch in each, at least one connection joining

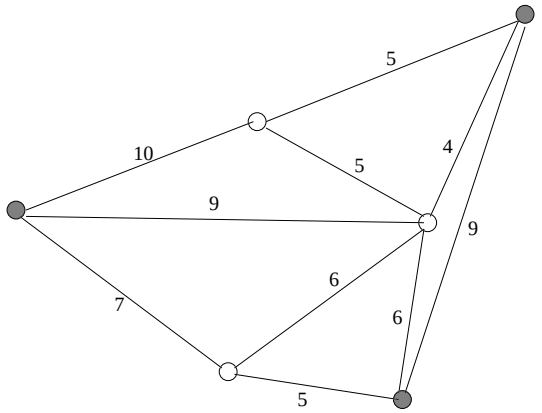


Figure B.1: An example of a telecommunications network - the branches to be connected of are the black vertices.

the two subset must be chosen.

Can the model be used also when prices may be negative? Would you recommend the use of the model for large companies with hundreds of branches?

Question 2.2

Use the model to solve the example problem of Figure B.1 using CPLEX with 0-1 variables. Solve the model also when the integrality constraints on the variables are removed. Comment on the two solutions obtained and their objective function values.

Question 2.3

An alternative to solving the problem to optimality is to use a method, which gives a good (but not necessarily optimal) solution fast. Such a method is called a *heuristic*.

As you realize the proximity of the problem to the minimum spanning tree you hit upon the following idea inspired by Prims algorithm: Start by choosing (randomly) a branch. Let us call it b_0. Now for each other branch compute the shortest distance to b_0. The branch with the shortest distance to b_0 will

be denoted b_1. Then add the connections on the path from b_1 to b_0 to the set of selected connections. In each of the subsequent iterations we continue this approach. Determine the unconnected branch b_i with the shortest distance to the tree already generated, that is, find an unconnected branch the shortest distance to any location that has been selected.

The process is terminated when all branches are added to the tree, that is, all branches are connected.

Obviously, the "killer" part of the described algorithm is the shortest path calculations. However, there are two problems: 1) the network is not directed, and 2) there may be edges of negative length. How would you cope with the non-directedness of the network, and which shortest path algorithm would you suggest to use given that negative edge lengths are present?

However, you realize that it may be easier to solve the problem regarding negative edge lengths by changing the problem to reflect that it is always beneficial to choose an edge of negative cost, even though you don't need the connection. Describe the corresponding problem transformation, whereby you end up with a network with only non-negative edges lengths.

You now are in the position that you may choose any of the shortest path algorithms touched upon in the course. In Figure B.2, the network of your clients company is shown. Which connections should be leased, if these are determined with the method just described with b_0 chosen to be A? Show the sequence in which the locations are considered, and the connections added in each iteration.

Question 3

As a result of you analysis Rubber Duck has decided to implement a supply chain optimization IT-system. They have specifically asked for your involvement in the project. You have been put in charge of the rather large project of implementing the IT-system and the corresponding organizational changes.

You immediately remember something about project management from you engineering studies. To get into the tools and the way of thinking, you decide to solve a small test case using PERT/CPM.

The available data is presented in Table B.2.

Activity	Imm. pred.	Normal time (D_i)	Crash time (d_j)	Unit cost when shortening
e_0		2	1	7
e_1	e_0	3	2	7
e_2	e_1	2	1	5
e_3	e_1	4	2	2
e_4	e_2, e_3	4	1	6
e_5	e_0	9	2	2
e_6	e_4, e_5	6	2	9

Table B.2: Data for the activities of the test case.

Question 3.1

Find the duration of the project if all activities are completed according to their normal times.

In Hillier and Lieberman Section 22.5, a general LP model is formulated allowing one to find the cheapest way to shorten a project.

Question 3.2

Formulate the model for the test project and state the dual model.

Question 3.3

For each activity i there is a variable x_i. Furthermore there are also two constrains, "$x_i \geq 0$" and "$x_i \leq D_i - d_i$". Denote the dual variables of the latter g_i. Argue that if $d_i < D_i$, then either g_i is equal to 0 or $x_i = D_i - d_i$ in any optimal solution.

Question 3.4

What is the additional cost of shortening the project time for the test case to 16?

Background for the first project.

For convenience, the background from the first project is repeated below.

You are employed by the consulting firm Optimal Solutions and one of your customers are a medium sized company with branches geographically distributed all over Denmark. The company has decided to build an intra-net based on communication capacity leased from a large telecommunication vendor, Yellow. This operator has a back-bone network connecting a number of locations, and you can lease communication capacity between individual pairs of points in the back-bone network and in that way build you own network. Fortunately, all branches are located in one of the locations of the back-bone network, however, not all locations house a branch of the company.

The pricing policy of Yellow is as follows: Each customer individually buys in on a set connections chosen by the customer. The price is the the sum of the prices for the individual connections. Each customer is offered a "customized" price for each connections, and these prices (although usually positive) may be negative to reflect a discount given by Yellow.

You are now hired to analyzed the situation of the company and determine which connections to lease in order to establish the cheapest intranet. Yellow guarantees that the connections will always be operative, so failure protection is not an issue.

Background for the second project.

After having reviewed your solution to the first project, the company comes back with the following comments: 1) As you have pointed out, the pricing strategy of Yellow seems odd in that connections with negative prices are "too attractive". 2) Though good solutions found quickly are valuable, its is necessary to be able also to find the optimal solution to the problem, even if this may be quite time consuming.

Question 4.1

In order to prepare for possible changes in the pricing strategy a model must be built, in which the intra-net solution proposed must be a *connected* network.

Formulate such a mathematical model.

Question 4.2

It now turns out that Yellow has also discovered the flaw in their original pricing strategy. They have informed you that a new price update will arrive soon. Until then you decide to continue your experiments on some other data you got from your client. The network is shown in Figure B.3 on page 193.

You decide to try to solve the problem using Branch & Cut based on your mathematical model from Question 2.1 of the first project. You initially set up an LP with objective function and the bounds "$x_{ij} \leq 1$" in CPLEX and add only one constraint, namely that the number of connections chosen must be at least 4. You then solve the problem using CPLEX. What is the result? Identify a constraint, which must be fulfilled by an optimal integer solution, but which is currently not fulfilled. Add this constraint and solve the revised problem. Comment on the result.

Question 4.3

The constraints in your mathematical model express that no cut in the network with capacity less than one (where "capacity" is measured by the sum over x-values on edges across the cut) separating two branch nodes must exist.

In order to apply Branch & Cut, you need a *separation routine*, i.e. an algorithm, which given a potential solution returns a violated inequality if such one exists. Explain how a maximum flow algorithm can be used as separation routine and estimate the complexity of this approach.

Question 4.4

You suddenly come to think about Gomory cuts. Maybe these are applicable? You realize that using CPLEX in interactive mode, it is not possible to extract the updated coefficients in a tableau row, but you can extract the updated coefficients in the objective function (the reduced costs). Is it in theory possible to

generate a Gomory cut from the updated coefficients in the objective function? If so, what is the result for the resulting LP-tableau from question 1.2?

Question 5

Your boss suddenly runs into your room and cries: "Drop everything - I have an urgent job for you. We need to win this new client. Find the best assignment of persons to jobs in the following situation:

I have 5 persons and 5 jobs, a profit matrix N showing in entry (i, j) the profit earned from letting person i do job j, and a cost matrix O showing the cost of educating person i for job j (the matrices are shown below). The educational budget is 18, and exactly one person has to be assigned to each job."

$$N = \begin{pmatrix} 4 & 10 & 6 & 5 & 4 \\ 0 & 8 & 2 & 8 & 1 \\ 3 & 10 & 5 & 6 & 6 \\ 2 & 9 & 4 & 8 & 4 \\ 1 & 8 & 4 & 9 & 3 \end{pmatrix}$$

$$O = \begin{pmatrix} 4 & 5 & 6 & 5 & 3 \\ 5 & 8 & 24 & 3 & 6 \\ 4 & 10 & 8 & 3 & 5 \\ 2 & 9 & 2 & 10 & 8 \\ 2 & 5 & 3 & 1 & 4 \end{pmatrix}$$

You decide to solve the problem to optimality using Branch-and-Bound.

You first convert the problem to a cost minimization problem, which you then solve. The lower bound computation should be done using Lagrangian relaxation as described in the next section, i.e. you have to use the objective value of the assignment problem resulting from the relaxation of the budget constraint with a non-negative multiplier. In order to explain the method to your boss, you have to explain why any non-negative multiplier gives rise to a lower bound for your given problem.

Suggested line of work for each question

Question 4.1

Consider an edge $\{i, j\}$. You have to ensure that "if $x_{ij} = 1$ then i must not be separated form any set containing all branches". How to model constraints of the type "$x_j = 0$ implies $x_i = 0$" is described in Hillier and Lieberman - figure out how to express "$x_j = 1$ implies $x_i = 1$" and use the idea in connection with the usual connectedness constraints know from the first project.

Question 4.2

In Branch-and-Cut, you first solve the LP relaxation, and as for TSP, this has exponentially many constraints. You should find one violated constraint by inspection, add it to the system, resolve, and describe the effect.

Question 4.3

As for TSP, Max Flow can be used based on that the x-values are interpreted as capacities. But you have to take care that only cuts separating two branch vertices are taken into consideration.

Question 4.4

The basic calculations leading to the classical Gomory cut from a chosen cut-row, i, with non-integral right hand side starts with the equation corresponding to the i'th row of the Simplex tableau. Start out assuming that the objective function value is non-integral, write out the "objective function equation" and check if all steps in the derivation procedure are valid.

Question 5

Converting the profit maximization problem to a cost minimization problem with non-negative cost is done by multiplying all values in the profit matrix by -1 and adding the largest element of the original profit matrix to each

element. Formulation of the Lagrangian relaxation can be done by subtracting the right-hand side of the budget constraint from the left-hand side, multiplying the difference with λ and adding the term to the objective function. Collecting terms to get the objective function coefficient for each x_{ij}, you get an assignment problem for each fixed value of λ. Choose a value of λ, e.g. 1, and solve the assignment problem. If the budget constraint is not fulfilled, then try to increase λ and resolve. If an optimal solution cannot be found through adjustments of λ (what are the conditions, under which an optimal solution to the relaxed problem is also optimal for the original problem?), choose a variable to branch on and create two new subproblems, each of which is treated as the original problem.

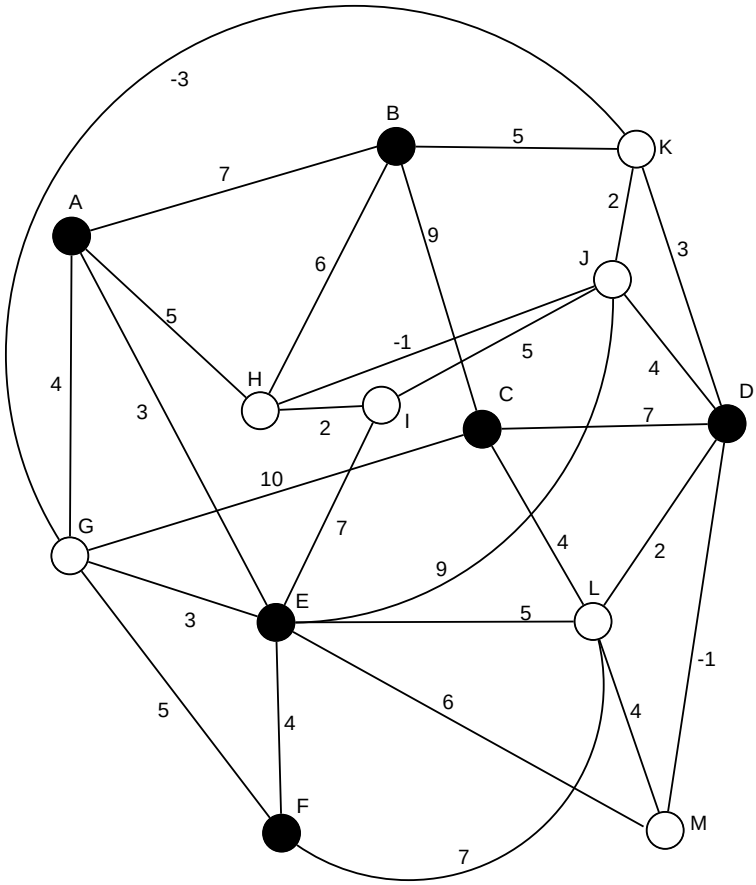


Figure B.2: The intra-network problem of your company. The black locations correspond to your branches to be connected.

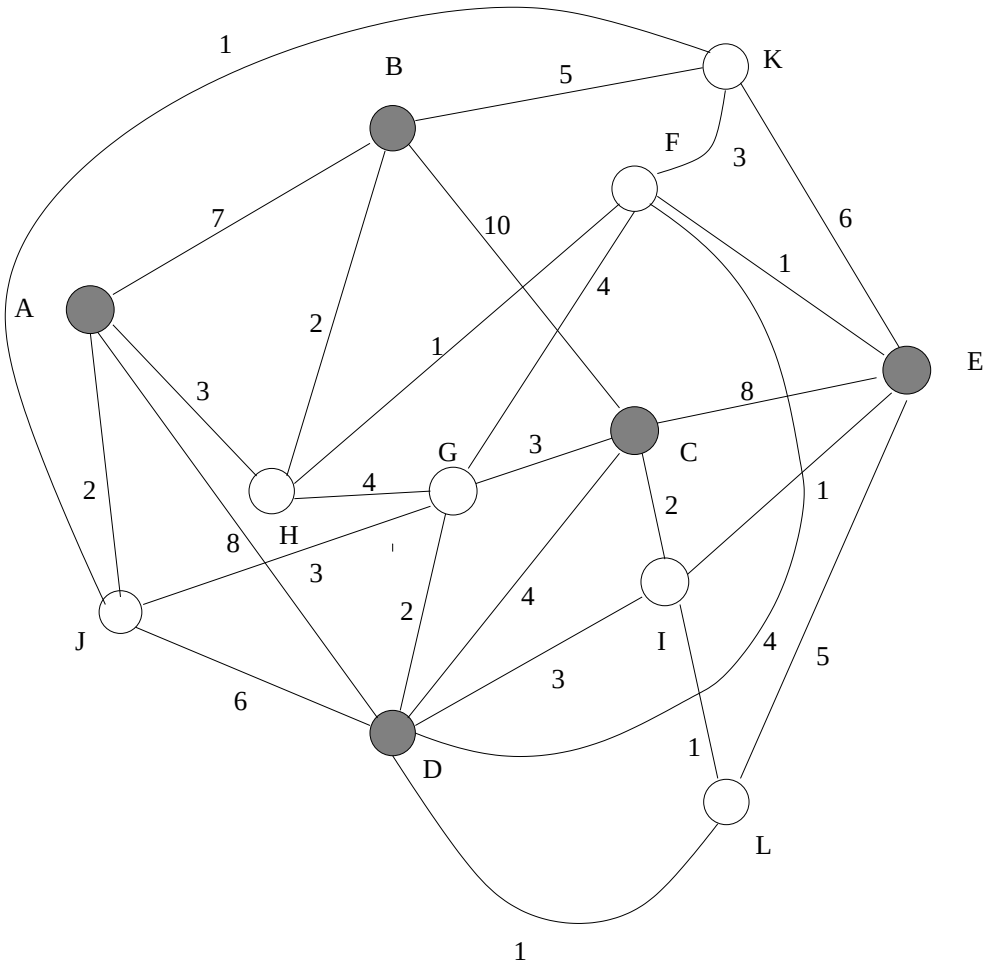


Figure B.3: An example intra-network problem from your client. The black locations correspond to your branches to be connected.