Computational Optimisation

Gregory Gutin

May 12, 2023

Contents

Abstract

This notes accompany the course CS3490: Computational Optimisation. We will study basic results, approaches and techniques of such important areas as linear and integer programming, and combinatorial optimisation. Many applications are overviewed.

Contents

Chapter 1

Introduction to Computational Optimisation

1.1 Introduction

Computational Optimisation (CS3490) will have 3 lectures a week. The aim is to introduce classical and modern methods and approaches in computational optimisation, and to overview applications and software packages available. The course covers both classical and very recent developments in the area. The main topics of the course are: linear and integer programming, construction heuristics and local search, polynomial time solvable problems, computational and theoretical analysis of heuristics, and meta-heuristics.

Most of the theory will be taught through examples with theoretical results formulated, but not proved. Only a few results will be proved. There will be a final exam (100 % mark). Any material taught at the lectures may be in the exam paper. A basic knowledge of graphs and matrices is assumed.

These notes contain areas of blank space in various places. Their purpose is to leave room for examples given in the lectures.

Unfortunately, it is impossible to recommend only one or two books covering the whole course. Several books and articles will be used in a supporting role to these notes, including the following:

- J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*, Springer, 2000 (Edition 1) or 2009 (Edition 2).

- J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, North Holland, 1976.

- M.W. Carter and C.C. Price, *Operations Research: A Practical Introduction*, CRC, 2001.
- F. Glover and M. Laguna, *Tabu Search*, Kluwer, 1997.
- G. Gutin and A. Punnen (eds.), *Traveling Salesman Problem and its Variations*, Kluwer, 2002 or Springer, 2007.
- J. Hromkovič, *Algorithmics for hard problems*, Springer, 2001.
- Z. Michalewicz and D.B. Fogel, *How to Solve It: Modern Heuristics*, Springer, 2000.
- W.L. Winston, *Operations Research*, 3rd edition, Duxbury Press, 1994,

Inevitably there are misprints in the notes. Please let me know if you have spotted one.

1.2 Algorithm efficiency and problem complexity

We start from two particular optimisation problems.

Assignment Problem (AP) We have n persons $p_1, \ldots, p_n$ and n jobs $w_1, \ldots, w_n$, and the cost c_{ij} of performing job i by person j. We wish to find an assignment of the persons to the jobs (one person per job) such that the total cost of performing the jobs is minimum. The costs are normally given by matrix $C = [c_{ij}]$.

Example (an instance):

$$C = \begin{pmatrix} 1 & 2 & 3 & 2 \\ 2 & 5 & 0 & 3 \\ 2 & 1 & 6 & 7 \\ 3 & 4 & 2 & 1 \end{pmatrix}.$$

Travelling Salesman Problem (TSP) There are n cities $t_1, \ldots, t_n$. Given distances d_{ij} from any city t_i to any other city t_j, we wish to find a shortest total distance *tour* that starts at city t_1, visits all cities (in some order) and returns to t_1.

Example (an instance):

$$C = \begin{pmatrix} 0 & 2 & 3 & 2 \\ 2 & 0 & 1 & 3 \\ 2 & 1 & 0 & 7 \\ 3 & 4 & 2 & 0 \end{pmatrix}.$$

The parameter n for both AP and TSP is the *size* of the problem. Algorithms for most optimisation problem are non-trivial and cannot be performed by hand even for relatively small sizes of a problem. Hence computers have to be used.

One possibility to predict that a certain computer code C to solve a certain problem P can handle all instances of P of size, say, $n = 100$ within a CPU hour is to carry out computational experiments for various instances of P of size 100. However, even if we have carried out computational experiments with 2000 instances of P and C solved each of them with a CPU hour, it does not mean C will spend less than one hour for the 2001st instance.

To predict the running time of algorithms and of the corresponding computer codes, researchers and practitioners compute the number of elementary operations required by a given algorithm to solve any instance of a certain problem (depending on the instance size). Elementary operations are arithmetic operations, logic operations, shift, etc.

Examples:

In most cases, we are interested in knowing how the number of performed operations depends on n asymptotically. For example, there is an algorithm A_{AP} for the AP that requires at most $O(n^3)$ operations. This means that the number of operations is at most cn^3, where c is a constant not depending on n. For the TSP there is an algorithm A_{TSP} that requires at most $O(2^n)$ operations. To sort n different integers whose values are between 1 and n there is an algorithm A_{BS} (`bucket sort`) that requires at most $O(n)$ operations.

We may draw some conclusions on the three algorithms without carrying out any computational experiments. For simplicity, assume that the constant c in each of the algorithms equals 10 and every operation takes 10^{-6} sec to perform. Then for $n = 20$, A_{BS} will take at most 2×10^{-4} sec, A_{AP} 0.08 sec and A_{TSP} 10 sec. For $n = 40$, A_{BS} will take at most 4×10^{-4} sec, A_{AP} 0.64 sec and A_{TSP} 127 days. For $n = 60$, A_{BS} will take at most 6×10^{-4} sec, A_{AP} 2.16 sec and A_{TSP} 366000 years.

Already this example indicates that while A_{BS} and A_{AP} can be used for moderate sizes, A_{TSP} may quickly become unusable. In fact, this example shows the difference between polynomial time and exponential time algorithms. Clearly, polynomial time algorithms are "good" and exponential are "bad". Unfortunately, for many optimisation problems, called NP-hard problems, polynomial time algorithms are unknown. Many 1000s of NP-hard problems are polynomially equivalent in the sense that if one of them admits a polynomial time algorithm then so does every other one. Many researchers have tried to find polynomial algorithms for various NP-hard problems, but failed. Thus, we believe that NP-hard problem cannot have polynomial time algorithms. Since TSP is NP-hard, it is very likely there does not exist a polynomial time algorithm for TSP.

1.3 Optimality and practicality

Many people with a mathematical education are trained to search for exact solutions to problems. If we are solving a quadratic equation, there is a formula for the exact solution. If a list of names needs to be sorted, we use an algorithm which produces a perfectly ordered list. Especially concerning mathematical theorems and their proofs, a respect for truth and perfection has evolved historically, and a nearly correct but incomplete or slightly flawed proof is considered of little or no value at all. Therefore the idea of solving a problem and not giving the "right" answer looks disappointing and disturbing. Yet there are justifiable reasons for accepting computational results that are imperfect or suboptimal.

First of all, models created by analysts are not perfect representations of real systems. So even if we could obtain exact solutions to the models they would not necessarily constitute exact solutions or perfect managerial advice to be applied for the real systems. Hence, costly efforts to achieve perfect solutions to mathematical models may not be warranted.

Every computer has only finite many digits to represent a number. Therefore some numbers should be rounded off. Further calculations with such numbers produce accumulated error. Often these errors may lead us far away from an optimal solution even if the algorithm is exact.

As we saw above many optimisation problems are NP-hard and do not admit polynomial time algorithms. Hence we cannot solve those problems to optimality even when the sizes of their instances are moderate. However many NP-hard problems are practically important. Even polynomial time algorithms may be impractical. Indeed, algorithms of running time $O(n^3)$ become impractical for values of n exceeding a few thousands.

Therefore we cannot and should not solve all problems to optimality. In practice, more often than not researchers and practitioners settle for suboptimal rather than optimal solutions.

Hence, in this course we will consider both exact and approximate methods and approaches in computational optimisation.

Question: Assume that every operation takes 10^{-6} sec to perform. We have two algorithms to solve a certain problem, one with running time at most $10n^5$, the other of 10×2^n. Which of the two algorithms is faster for $n = 20$, $n = 40$? Which conclusion can one draw?

Chapter 2

Introduction to Linear Programming

2.1 Linear programming (LP) model

An optimisation problem is called an *LP problem* (or an *instance of LP*) if both objective function and constraints are linear. For example,

$$\begin{array}{lll} \text{maximize} & 2x_1 + 3x_2 & \\ \text{subject to} & 3x_1 - 5x_2 & \leq 7 \\ & 4x_1 - x_2 & = 3 \\ & x_2 \geq 0 & \end{array}$$

is an LP problem.

In general, the objective function is of the form $c_1x_1+c_2x_2+\cdots+c_nx_n$, a linear function of the decision variables x_i with coefficients c_i, which is to be minimized or maximized. All constraints are of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n = (\leq, \geq)b$.

2.2 Formulating problems as LP problems

Example 2.2.1. *(W.L. Winston) An American company manufactures luxury cars and trucks. The company believes that its most likely customers are high-income women and men. To reach these groups the company has embarked on an ambitious TV advertising campaign and has decided to purchase one minute commercial spots on two types of programmes: comedy shows and football games. Each comedy commercial is seen by 7*

million high-income women and 2 million high-income men. Each football commercial is seen by 2 million high-income women and 12 million high-income men. One minute comedy advert costs $ 50000 and 1 minute football advert costs $ 100000. The company would like the commercials to be seen by at least 28 million high-income women and 24 million high-income men. Write down an LP model of this problem.

Solution: The company must decide how many comedy and football adverts should be purchased so the decision variables are: x_1 =number of one minute comedy adverts, x_2 =number of 1 minute football adverts. The company wants to minimise total advertising cost (in thousands of dollars). Total advertising cost = cost of comedy adverts + cost of football adverts = $50x_1 + 100x_2$. Thus, the company's objective function is

$$\min z = 50x_1 + 100x_2.$$

The company faces the following constraints:

Constraint 1 Commercials must reach at least 28 million high-income women.

Constraint 2 Commercials must reach at least 24 million high-income men.

Constraint 1 may be expressed as $7x_1+2x_2 \geq 28$ and Constraint 2 may be expressed as $2x_1 + 12x_2 \geq 24$. The sign restrictions $x_1 \geq 0$ and $x_2 \geq 0$ are necessary, so the company's model is given by:

$$\begin{aligned} \min z = \quad & 50x_1 + 100x_2 & \\ \text{s.t.} \quad & 7x_1 + 2x_2 & \geq 28 \\ & 2x_1 + 12x_2 & \geq 24 \\ & x_1, x_2 & \geq 0. \end{aligned}$$

Example 2.2.2. *(W.L. Winston) An auto company manufactures cars and trucks. Each vehicle must be processed in the paint shop and body assembly shop. If the paint shop were only painting trucks, 40 per day could be painted. If the paint shop were only painting cars, 60 per day could be painted. If the body shop were only producing cars it could process 50 per day. If the body shop were only producing trucks, it could process 50 per day. Each truck contributes $300 to profit and each car contributes $200 to profit. Write down an LP model of this problem.*

Solution: The company must decide how many cars and trucks should be produced daily. This leads us to define the following decision variables: x_1 = number of trucks produced daily, x_2 = number of cars produced daily. The company's daily profit (in hundreds of dollars) is $3x_1 + 2x_2$, so the company's objective function may be written as:

$$\max z = 3x_1 + 2x_2.$$

The company's two constraints are the following:

Constraint 1 The fraction of the day during which the paint shop is busy is less than or equal to 1.

Constraint 2 The fraction of the day during which the body shop is busy is less than or equal to 1.

We have

Fraction of day paint shop works on trucks $= \frac{1}{40}x_1$

Fraction of day paint shop works on cars $= \frac{1}{60}x_2$

Fraction of day body shop works on trucks $= \frac{1}{50}x_1$

Fraction of day body shop works on cars $= \frac{1}{50}x_2$

Thus, Constraint 1 may be expressed by

$$\frac{1}{40}x_1 + \frac{1}{60}x_2 \leq 1$$

and Constraint 2 may be expressed by

$$\frac{1}{50}x_1 + \frac{1}{50}x_2 \leq 1.$$

Since $x_1 \geq 0$ and $x_2 \geq 0$ must hold, the relevant model is

$$\begin{array}{rrl} \max z = & 3x_1 + 2x_2 & \\ \text{s.t.} & \frac{1}{40}x_1 + \frac{1}{60}x_2 & \leq 1 \\ & \frac{1}{50}x_1 + \frac{1}{50}x_2 & \leq 1 \\ & x_1, x_2 & \geq 0. \end{array}$$

Example 2.2.3. *(W.L. Winston) You have decided to enter the candy business. You are considering producing two types of candies: Slugger Candy and Easy Out Candy, both of which consist solely of sugar, nuts, and chocolate. At present, you have in stock 100 oz of sugar, 20 oz of nuts, and 30 oz of chocolate. The mixture used to make Easy Out Candy must contain at least 20 % nuts. The mixture used to make Slugger Candy must contain at least 10 % nuts and 10 % chocolate. Each ounce of Easy Out Candy can be sold for 25p, and each ounce of Slugger Candy for 20p. Formulate an LP model that will enable you to maximize you revenue from candy sales.*

Solution: (Fill in the details yourself)

The LP model is

$$\begin{array}{rrl} \max z = & 25x_1 + 20x_2 & \\ \text{s.t.} & x_1 + x_2 & \leq 150 (= 100 + 20 + 30) \\ & 0.2x_1 + 0.1x_2 & \leq 20 \\ & 0.1x_2 & \leq 30 \\ & x_1, x_2 & \geq 0. \end{array}$$

Example 2.2.4. *(M.W. Carter and C.C. Price) A dual-processor computing facility is to be dedicated to administrative and academic application jobs for at least 10 hours each day. Administrative jobs require 2 seconds of CPU time on processor 1 and 6 seconds on processor 2, while academic jobs require 5 seconds on processor 1 and 3 seconds on processor 2. A scheduler must choose how many of each type of job (administrative and academic) to execute, in such a way as to minimize the amount of time that the system is occupied with these jobs. The system is considered to be occupied even if one processor is idle. (Assume that the sequencing of the jobs on each processor is not an issue here, just the selection of how many of each type of job.)*

Solution: Let x_1 and x_2 denote respectively the number of administrative and academic jobs selected for execution on the dual-processor system. Because policies require that each processor be available for at least 10 hours, we have the following two constraints: $2x_1 + 5x_2 \geq 10 \times 3600$, $6x_1 + 3x_2 \geq 10 \times 3600$ and $x_1 \geq 0$ and $x_2 \geq 0$. The system is considered occupied as long as either processor is busy. Therefore, to minimize the completion time for the set of jobs, we must minimize $\max\{2x_1 + 5x_2, 6x_1 + 3x_2\}$.

This nonlinear objective can be made linear if we introduce a new variable x_3, where $x_3 = \max\{2x_1 + 5x_2, 6x_1 + 3x_2\} \geq 0$.

Now if we require $x_3 \geq 2x_1 + 5x_2$ and $x_3 \geq 6x_1 + 3x_2$ and make our objective to minimize x_3, we have the desired linear formulation $(x_1, x_2, x_3 \geq 0)$.

2.3 Graphical solution of LP problems

Example 2.3.1. *Graph the set of points* $2x_1 + 3x_2 \leq 6,\ x_1, x_2 \geq 0$.

Solution: See Figure 2.1. We start from drawing $2x_1 + 3x_2 = 6$. To draw this graph we find two points in the $x_1 - x_2$ plane: $x_1 = 0, x_2 = 2$, $x_2 = 0, x_1 = 3$.

To graph $2x_1 + 3x_2 \leq 6$, we check whether the point $x_1 = 0 = x_2$ belongs to $2x_1 + 3x_2 \leq 6$ or not. We see that $2 \times 0 + 3 \times 0 \leq 6$, so $x_1 = 0 = x_2$ does belong to $2x_1 + 3x_2 \leq 6$, and hence the South-West part of the half-plane is the graph of $2x_1 + 3x_2 \leq 6$.

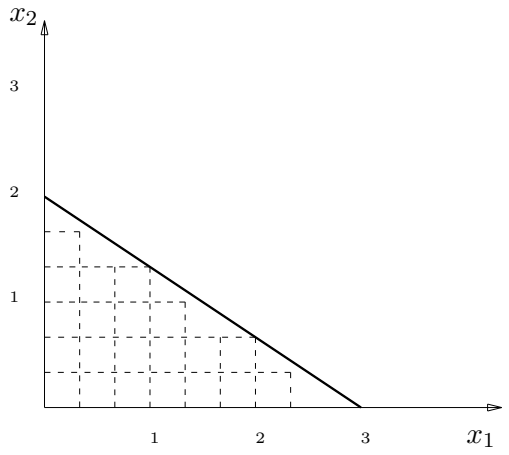


Figure 2.1: Area for $2x_1 + 3x_2 \leq 6,\ x_1, x_2 \geq 0$.

Example 2.3.2. *Consider the following instance of LP*

$$\begin{aligned} \min z = \quad & 50x_1 + 100x_2 & \\ s.t. \quad & 7x_1 + 2x_2 & \geq 28 \\ & 2x_1 + 12x_2 & \geq 24 \\ & x_1, x_2 & \geq 0. \end{aligned}$$

Solve the problem graphically.

Solution: See Figure 2.2. The line $7x_1 + 2x_2 = 28$ contains the points $x_1 = 0, x_2 = 14$ and $x_2 = 0, x_1 = 4$, and the line $2x_1 + 12x_2 = 24$ the points $x_1 = 0, x_2 = 2$ and $x_2 = 0, x_1 = 12$. The point of intersection of the two lines can be found by solving the system of the two equations which describe them. From the second equation $x_1 = 12 - 6x_2$; substituting into the first equation yields $7(12 - 6x_2) + 2x_2 = 28$, thus $56 - 40x_2 = 0$. As a result we get $x_2 = 1.4$ and $x_1 = 12 - 6 \times 1.4 = 3.6$. The graph of the four constraints, that is, the set of points which satisfy all the constraints, is the entire (infinite) North-East area of Figure 2.2.

The objective function lines are $50x_1 + 100x_2 = \text{const}$. It is easy to see that const decreases as the line moves towards the South-West corner. Thus the minimum is achieved at $x_1 = 3.6, x_2 = 1.4$. The optimal value is $z = 50 \times 3.6 + 100 \times 1.4 = 320$.

Example 2.3.3. *Add the constraint $x_1 + x_2 \leq 2$ to Example 2.3.2. Investigate the new LP problem.*

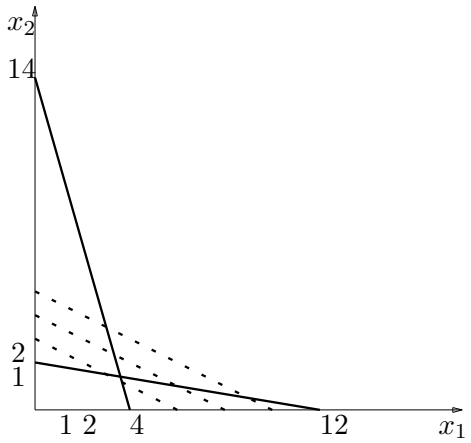


Figure 2.2: Figure for Example 2.3.2

Solution: We see that no point satisfies the 5 constraints (see Figure 2.3). So, the LP problem has no *feasible solution*.

Example 2.3.4. *Consider Example 2.3.2, but with the objective to maximize rather than minimize.*

Solution: For any const > 320 there are feasible points satisfying $50x_1 + 100x_2 =$ const. Thus, the new LP problem is *unbounded*.

Example 2.3.5. *Consider Example 2.3.2, but with objective function* $\min z = x_1 + 6x_2$.

Solution: We see that one optimal solution is still $x_1 = 3.6, x_2 = 1.4$, but there are infinitely many optimal points (an entire straight line segment). See Figure 2.4.

Conclusions

There are four possibilities for an LP problem:

- The LP problem has a unique optimal solution.
- The LP problem has infinitely many optimal solutions.
- The LP problem has no feasible solutions.
- The LP problem is unbounded.

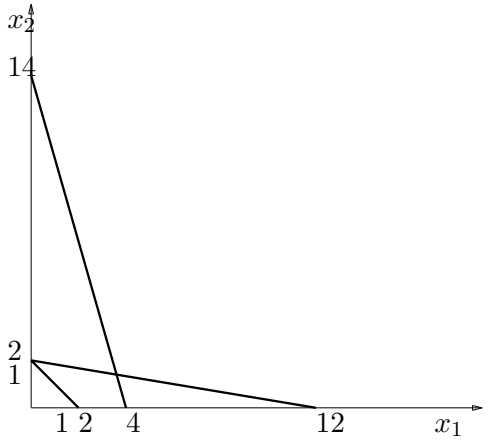


Figure 2.3: Figure for Example 2.3.3

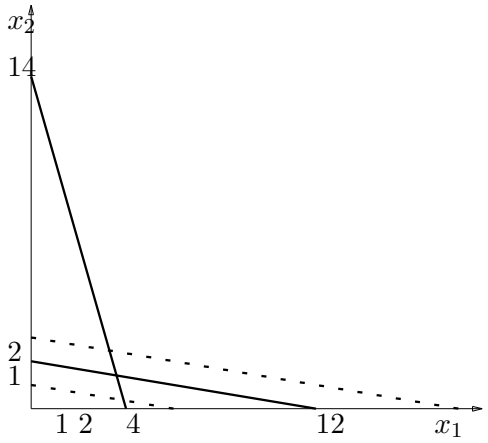


Figure 2.4: Figure for Example 2.3.5

2.4 Questions

Question 2.4.1. *(W.L. Winston) Leary Chemicals manufactures three chemicals: A, B, and C. These chemicals are produced via two production processes: 1 and 2. Running Process 1 for an hour costs \$4 and yields 3 units of A, 1 of B, and 1 of C. Running Process 2 for an hour costs \$1 and produces 1 unit of A and 1 of B. To meet customer demands, at least 10 units of A, 5 of B, and 3 of C must be produced daily. Write down an LP model to minimize the cost of meeting Leary Chemical's daily demands.*

Question 2.4.2. *A small toy company manufactures two types of wooden toys: cars and trains. Each car sells for \$25 and uses \$20 of raw material and labour. Each train sells for \$30 and uses \$24 of raw material and labour. The manufacture of both types of toys requires two types of skilled labour: carpentry and finishing. A car requires 2 hours of carpentry labour and 3 hours of finishing labour. A train requires 3 hours of carpentry labour and 4 hours of finishing labour. Each week the company can get all needed material, but only 150 finishing hours and 100 carpentry hours. The company has already orders on 20 trains, but expects to sell all manufactured toys. The company wants to decide what number of cars and trains to manufacture in order to maximize its profit. Write down an LP model to maximize the toy company profit.*

Question 2.4.3. *A television manufacturing company has to decide on the number of 27- and 20-inch sets to be produced at one of its factories. Market research indicates that at most 40 of the 27-inch sets and 10 of the 20-inch sets can be sold per month. The maximum number of work hours available is 500 per month. A 27-inch set requires 20 work hours, and a 20-inch set requires 10 work hours. Each 27-inch set sold produces a profit of £120, and each 20-inch set produces a profit of £80. A wholesaler has agreed to purchase all the television sets produced if the numbers do not exceed the maxima indicated by the market research.*

(a) Formulate a linear programming model for this problem.

(b) Solve this model graphically.

Question 2.4.4. *Goldilocks needs to find at least 16 lb of gold and at least 18 lb of silver to pay the monthly rent. There are two mines in which Goldilocks can find gold and silver. Each day that Goldilocks spends in mine 1, she finds 2 lb of gold and 2 lb of silver. Each day that Goldilocks spends in mine 2, she finds 1 lb of gold and 3 lb of silver. Formulate an LP to help Goldilocks meet her requirements while spending as little time as possible in the mines. Graphically solve the LP.*

Question 2.4.5. *Find out which of the four possibilities this LP problem belongs to.*

$$\max z = \quad x_1 + x_2$$

$$\begin{aligned} s.t. \quad x_1 + x_2 &\leq 4 \\ x_1 - x_2 &\geq 5 \\ x_1, x_2 &\geq 0. \end{aligned}$$

Justify your answer.

Question 2.4.6. *Find out which of the four possibilities this LP problem belongs to.*

$$\begin{aligned} \max z = \quad 4x_1 + x_2 & \\ s.t. \quad 8x_1 + 2x_2 &\leq 16 \\ 5x_1 + 2x_2 &\leq 12 \\ x_1, x_2 &\geq 0. \end{aligned}$$

Justify your answer.

Question 2.4.7. *Find out which of the four possibilities this LP problem belongs to.*

$$\begin{aligned} \max z = \quad -x_1 + 3x_2 & \\ s.t. \quad x_1 - x_2 &\leq 4 \\ 5x_1 + 2x_2 &\geq 4 \\ x_1, x_2 &\geq 0. \end{aligned}$$

Justify your answer.

Question 2.4.8. *A computer manufacturing company has to decide on the number of 64-processor and 32-processor computers to be assembled at one of its factories. Market research indicates that at most 20 of the 64-processor computers and 30 of the 32-processor computers can be sold per month. The maximum number of work hours available is 5100 per month. A 64-processor computer requires 200 work hours, and a 32-processor computer requires 100 work hours. Each 64-processor computer sold produces a profit of £9000, and each 32-processor computer produces a profit of £6000. A wholesaler has agreed to purchase all the computers assembled if the numbers do not exceed the maxima indicated by the market research. Write down a linear programming model to maximize the computer company profit. Why may a linear programming model not be adequate for the above problem?*

Question 2.4.9. *Solve the following LP problem graphically:*

$$\begin{aligned} \max z = \quad 25x_1 + 50x_2 & \\ s.t. \quad 7x_1 + 2x_2 &\leq 28 \\ 2x_1 + 12x_2 &\leq 24 \\ x_1, x_2 \geq 0. & \end{aligned}$$

2.5 Solutions

Question 2.4.3

x_1 is the number of cars, x_2 is the number of trains. Profit for a car \$ 25-20=\$5, profit for a train \$ 30-24=\$6.

Thus,

$$\begin{aligned} \max z = \quad & 5x_1 + 6x_2 & \\ \text{s.t.} \quad & 2x_1 + 3x_2 & \leq 100 \\ & 3x_1 + 4x_2 & \leq 150 \\ & x_1 \geq 0, x_2 \geq 20. & \end{aligned}$$

Question 2.4.3

a) The company must decide how many 27- and 20-inch television sets to produce in order to maximise their profit. Therefor the decision variables are: x_1 = number of 27-inch tv sets produced per month and x_2 = number of 20-inch tv sets produced per month. So the objective function is:

$$\max z = 120x_1 + 80x_2.$$

The company faces the following constraint:

Constraint 1: The maximum number of work hours available is 500 per month. A 27-inch set requires 20 work hours, and a 20-inch set requires 10 work hours.

$$20x_1 + 10x_2 \leq 500.$$

The wholesaler who has agreed to purchase the television sets has enforced another constraint:

Constraint 2: Market research indicates that at most 40 of the 27-inch sets and 10 of the 20-inch sets can be sold per month. The wholesaler will not buy more than the number indicated by the market research.

$$x_1 \leq 40$$

$$x_2 \leq 10.$$

Finally the LP model for the problem is:

$$\begin{aligned} \max z = \quad & 120x_1 + 80x_2 & \\ \text{s.t.} \quad & 20x_1 + 10x_2 & \leq 500 \\ & x_1 & \leq 40 \\ & x_2 & \leq 10 \\ & x_1, x_2 \geq 0. & \end{aligned}$$

b) For a graphical solution let us first draw all the lines that represent the constraints, where the "$\leq$" are replaced by the "$=$" sign.

Therefore, the line $20x_1 + 10x_2 = 500$ passes through the points $x_1 = 25$, $x_2 = 0$ and $x_1 = 0$, $x_2 = 50$. The line $x_1 = 40$ is vertical passing through $x_1 = 40$ and similar, $x_2 = 10$ is the horizontal line passing through $x_2 = 10$. See fig. 2.5. Note that the point (0,0), i.e. $x_1 = x_2 = 0$ satisfies all the constraints above. Thus, the feasibility region is the quadrangle bordered from above by (an interval of) $x_2 = 10$, from below by x_1-axis, from the left by the x_2-axis and from the right by the line $20x_1 + 10x_2 = 500$.

The dashed lines on fig. 2.5 show $120x_1+80x_2$ =const for const=2400 and 3600. It's not hard to see that the problem has one unique optimal solution at the point of intersection of lines $x_2 = 10$ and $20x_1 + 10x_2 = 500$ (before $120x_1 + 80x_2$ =const leaves the feasibility region). We find the coordinates of this point by solving the system of linear equations consisting of $x_2 = 10$ and $20x_1+10x_2 = 500$. Since $x_2 = 10$, $20x_1+10\times 10 = 500$ implying $x_1 = 400/20 = 20$. Thus the coordinates of the intersection point are $x_1 = 20$ and $x_2 = 10$. Thus, the maximal profit the company can achieve is $z = 120 \times 20 + 80 \times 10 = 3200$.

Alternatively, we can compute the values of z at the four corner points of the quadrangle and find the maximum among them. (Remember that at least one of the corner points of the feasibility region of any LP problem with at least one optimal solution, is an optimum.) For the two points on the axis x_1, i.e. $x_2 = 0$, we have $z_1 = 0$ and $z_2 = 120\times 25 = 3000$. The upper left corner point has coordinates $x_1 = 0$ and $x_2 = 10$. Thus, $z_3 = 120\times 0+80\times 10 = 800$. The upper right corner point is the intersection point of $20x_1 + 10x_2 = 500$ and $x_2 = 10$. Hence $x_2 = 10$, $x_1 = 20$ and $z_3 = 3200$ as above.

Question 2.4.4

Let x_1 be the number of days Goldilocks spends in mine 1 and x_2 the number of days she spends in mine 2. Then the LP model is

$$\begin{aligned} \min z = \quad & x_1 + x_2 & \\ \text{s.t.} \quad & 2x_1 + x_2 & \geq 16 \\ & 2x_1 + 3x_2 & \geq 18 \end{aligned}$$

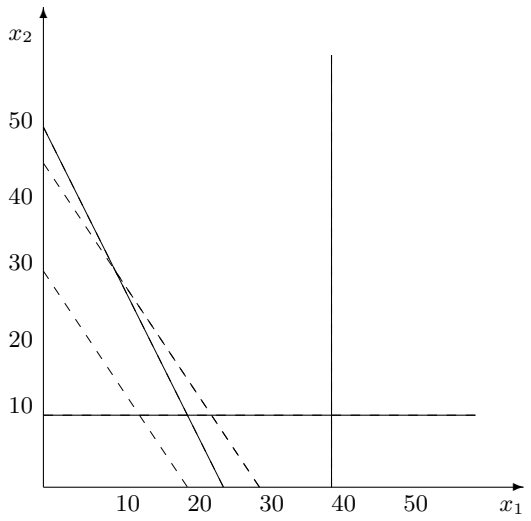


Figure 2.5: Graphical solution to Question 2.4.3

$$x_1, x_2 \quad \geq 0.$$

Solving the above problem graphically (add details!) we see that the optimum is at the point satisfying the following system of equations.

$$\begin{aligned} 2x_1 + x_2 &= 16 \\ 2x_1 + 3x_2 &= 18. \end{aligned}$$

To solve this system, subtract the 1st equation from the 2nd getting $2x_2 = 2$. So $x_2 = 1$. Substituting $x_2 = 1$ into the 1st equation, we have $2x_1 = 15$ implying $x_1 = 7.5$. Thus, $z_{opt} = 8.5$.

Question 2.4.5: The LP problem has no feasible solutions.

Question 2.4.6: The LP problem has infinitely many optimal solutions.

Question 2.4.7: The LP problem is unbounded.

Question 2.4.8

Let x_1 and x_2 be the number of 64-processor and 32-processor computers to be produced per month. Then, we have the following LP model

$$\begin{aligned} \max z = \quad & 9000x_1 + 6000x_2 & \\ \text{s.t.} \quad & 200x_1 + 100x_2 & \leq 5100 \\ & x_1 & \leq 20 \\ & x_2 & \leq 30 \\ & x_1, x_2 & \geq 0. \end{aligned}$$

This model is not adequate since it does not include the constraint that x_1 and x_2 are integers.

Question 2.4.9

Recall that we wish to solve the following LP problem graphically:

$$\begin{aligned} \max z = \quad & 25x_1 + 50x_2 & \\ \text{s.t.} \quad & 7x_1 + 2x_2 & \leq 28 \\ & 2x_1 + 12x_2 & \leq 24 \\ & x_1, x_2 \geq 0. & \end{aligned}$$

See Figure 2.6. The line $7x_1 + 2x_2 = 28$ contains the points $x_1 = 0, x_2 = 14$ and $x_2 = 0, x_1 = 4$, and the line $2x_1 + 12x_2 = 24$ the points $x_1 = 0, x_2 = 2$ and $x_2 = 0, x_1 = 12$. The point of intersection of the two lines can be found by solving the system of the two equations which describe them. From the second equation $x_1 = 12 - 6x_2$; substituting into the first equation yields $7(12 - 6x_2) + 2x_2 = 28$, thus $56 - 40x_2 = 0$.

As a result we get $x_2 = 1.4$ and $x_1 = 12 - 6 \times 1.4 = 3.6$. The graph of the four constraints, that is, the set of points which satisfy all the constraints, is the finite quadrangle of Figure 2.6.

The objective function lines are $25x_1 + 50x_2 = \text{const}$. It is easy to see that const decreases as the line moves towards the South-West corner. Thus the maximum is achieved at $x_1 = 3.6, x_2 = 1.4$. The optimal value is $z = 25 \times 3.6 + 50 \times 1.4 = 160$.

Alternatively, we can find z-values of the four corner points of the quadrangle and compare them. For the two points on the x_1-axis, we have $z_1 = 0$ and $z_2 = 25 \times 4 = 100$. For the point on the x_2-axis, we have $z_3 = 50 \times 2 = 100$. The z-value of the fourth point $z_4 = 160$ as computed above. Clearly, this point is optimal.

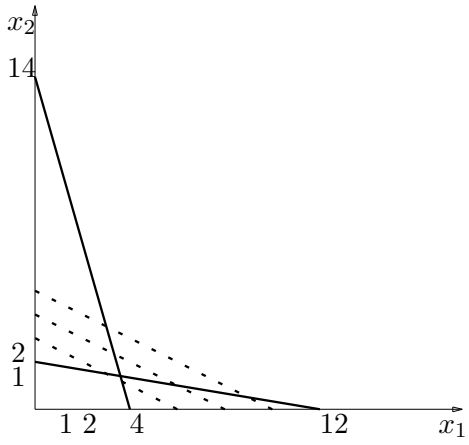


Figure 2.6: LP graph

Chapter 3

Simplex Method

3.1 LP in General Form

The general form of LP is

$$\begin{aligned} \text{opt } z = \quad & c_1x_1 + c_2x_2 + \ldots + c_nx_n \\ \text{s.t.} \quad & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \star b_1 \\ & a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \star b_2 \\ & \ldots\ldots\ldots \\ & \ldots\ldots\ldots \\ & a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \star b_m \\ & x_1, x_2, \ldots, x_n \geq 0, \end{aligned}$$

where the x_i are indeterminates, $\star$ is either $=$ or $\leq$ or $\geq$ and opt is either max or min.

Often the general form is written in a vector-matrix notation:

$$\begin{aligned} \text{opt } z = \quad & cx \\ \text{s.t.} \quad & Ax \bullet b \\ & x \geq 0, \end{aligned}$$

where $c = (c_1, \ldots, c_n)$, $\bullet$ is a column-vector in which every entry is either $=$ or $\leq$ or $\geq$,

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ \ldots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ \ldots \\ b_m \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}.$$

Note that all coefficients and right hand sides of the general form are not restricted to be of certain sign.

As an example, consider

$$\begin{array}{rll} \min z = & 5x_1 + x_2 & \\ \text{s.t.} & 2x_1 + x_2 & \leq 16 \\ & 2x_1 + 3x_2 & \geq 18 \\ & x_1 - 4x_2 & = -3 \\ & x_1, x_2 & \geq 0. \end{array}$$

For this LP, in a vector-matrix notation, we have:

$$c = (5, 1), \quad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad b = \begin{pmatrix} 16 \\ 18 \\ -3 \end{pmatrix}, \quad A = \begin{pmatrix} 2 & 1 \\ 2 & 3 \\ 1 & -4 \end{pmatrix}.$$

In the previous chapter, we said that LP may have either (a) a unique optimal solution, or (b) infinite number of optimal solutions, or (c) no solutions, (d) unbounded. This list excludes 2,3 or any other finite number of solutions. The next theorem tells us why.

Theorem 3.1.1. *If LP has at least two distinct optimal solutions, then it has infinite number of optimal solutions.*

Proof: Let $x' = (x'_1, x'_2, \ldots x'_n)$ and $x'' = (x''_1, x''_2, \ldots x''_n)$ are two distinct optimal solutions. Then

$$\lambda x' + (1-\lambda)x'' = (\lambda x'_1 + (1-\lambda)x''_1, \lambda x'_2 + (1-\lambda)x''_2, \ldots, \lambda x'_n + (1-\lambda)x''_n)$$

is another optimal solution for every $0 < \lambda < 1$. Indeed, $\lambda x' + (1-\lambda)x''$ satisfies all constraints since

$$A(\lambda x' + (1-\lambda)x'') = \lambda Ax' + (1-\lambda)Ax'' \bullet \lambda b + (1-\lambda)b = b.$$

Also, $\lambda x' + (1-\lambda)x''$ is optimal. Indeed, $z_{opt} = cx' = cx''$. So

$$c(\lambda x' + (1-\lambda)x'') = \lambda cx' + (1-\lambda)cx'' = \lambda z_{opt} + (1-\lambda)z_{opt} = z_{opt}.$$

Since the are infinite number of values of λ, there are infinitely many solutions.

3.2 Standard Form

In preparation for using the Simplex Method, it is necessary to express the linear programming problem in *standard form*. For a LP problem with n variables and m constraints the standard form is

$$\begin{aligned}
\max z = \quad & c_1x_1 + c_2x_2 + \ldots + c_nx_n \\
\text{s.t.} \quad & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\
& a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\
& \ldots\ldots\ldots \\
& \ldots\ldots\ldots \\
& a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \\
& x_1, x_2, \ldots, x_n \geq 0,
\end{aligned}$$

where the constants $b_1, \ldots, b_m$ are non-negative.

Often the standard form is written in a vector-matrix notation:

$$\begin{aligned}
\max z = \quad & cx \\
\text{s.t.} \quad & Ax = b \\
& x \geq 0,
\end{aligned}$$

where $c = (c_1, \ldots, c_n)$,

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ \ldots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ \ldots \\ b_m \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}.$$

Although this standard form is required by the Simplex Method, it is not necessarily the form that arises naturally when we first formulate our LP model. Thus, some transformations are required to translate the initial form into the standard form.

To convert a minimization problem into a maximization problem, we can simply multiply the objective function by -1 and replace min by max. For example, the problem of minimizing $z = 2x_1 - 3x_2$ is equivalent to that of maximizing $z = -2x_1 + 3x_2$.

Equality constraints require no modification. Less-than-or-equal-to ($\leq$) inequalities require introduction of *slack variables*. For example, $2x_1+5x_2 \leq 7$ becomes $2x_1+5x_2+s_1 = 7$. Greater-than-or-equal-to ($\geq$) inequalities are modified by introducing *surplus variables*. For example, $4x_1 - 6x_2 \geq 8$ becomes $4x_1 - 6x_2 - s_2 = 8$.

Finally, the LP standard form requires that every variable is non-negative. If some variable, say x_3, is not required to be non-negative in the initial formulation, then we modify it as follows: we replace x_3, in each constraint and the objective function, by $x_3' - x_3''$ and add $x_3', x_3'' \geq 0$.

Example 3.2.1. *Transform the following LP problem into standard form.*

$$\begin{aligned} \min z = \quad & -50x_1 + 100x_2 & \\ s.t. \quad & 7x_1 + 2x_2 & \geq 28 \\ & 2x_1 + 12x_2 & \leq 24 \\ & x_1, x_2 & \geq 0. \end{aligned}$$

Solution: We have

$$\begin{aligned} \max z' = \quad & 50x_1 - 100x_2 & \\ \text{s.t.} \quad & 7x_1 + 2x_2 - s_1 & = 28 \\ & 2x_1 + 12x_2 + s_2 & = 24 \\ & x_1, x_2, s_1, s_2 & \geq 0. \end{aligned}$$

Example 3.2.2. *Transform the following LP problem into standard form.*

$$\begin{aligned} \max z = \quad & 50x_1 - 10x_2 & \\ s.t. \quad & x_1 + 2x_2 & \leq 28 \\ & 2x_1 + 15x_2 & \geq 24 \\ & x_2 & \geq 0. \end{aligned}$$

Solution: We have

$$\begin{aligned} \max z = \quad & 50x_1' - 50x_1'' - 10x_2 & \\ \text{s.t.} \quad & x_1' - x_1'' + 2x_2 + s_1 & = 28 \\ & 2x_1' - 2x_1'' + 15x_2 - s_2 & = 24 \\ & x_1', x_1'', x_2, s_1, s_2 & \geq 0. \end{aligned}$$

3.3 Solutions of Linear Systems

Consider a system of independent linear equations, $Ax = b$, consisting of m equations and n unknowns x_i. The n unknowns include the original decision variables and any other variables that may have been introduced in order to achieve standard form.

If a system of equations is independent, then $m \leq n$. If $m = n$ and $\det A \neq 0$, then there is a unique solution $x = A^{-1}b$. Optimization is not an issue here.

From now on, suppose that $m < n$, i.e., there are infinitely many solutions of the system. In this case, there are $(n - m)$ degrees of freedom in solving the system. This means we can arbitrarily assign any values to $(n - m)$ of the n variables, and then solve the m equations with m unknowns.

A *basic solution* to the system of m equations and n unknowns is obtained by setting $(n-m)$ of the variables to zero, and solving for the remaining m variables. The m variables that are not set equal to zero are called basic variables, and the variables that are set to zero are called *non-basic variables*. The number of basic solutions is just the number of ways we can choose $n - m$ variables (or m variables) from the set of n variables, and this number is given by $\binom{n}{m} = \frac{n!}{m!(n-m)!}$.

Not all of the basic solutions satisfy all problem constraints and non-negativity constraints. Those that do not meet these requirements are *infeasible* solutions. The ones that do meet the restrictions are called *basic feasible solutions*. An *optimal* basic feasible solution is a basic feasible solution that optimizes the objective function. The basic feasible solutions correspond precisely to the *extreme points* of the feasible region (as defined in our earlier discussion of graphical solutions). Because some optimal feasible solution is guaranteed to occur at an extreme point (and consequently is a basic feasible solution), the search for an optimal feasible solution could be carried out by an examination of the at most $\binom{n}{m}$ basic feasible solutions and a determination of which one yields the best objective function value.

The Simplex Method performs such a search, although in a very efficient way. We define two extreme points of the feasible region (or two basic feasible solutions) as being *adjacent* if all but one of their basic variables are the same. Thus, a transition from one basic feasible solution to an adjacent basic feasible solution can be thought of as exchanging the roles of one basic variable and one non-basic variable. The Simplex Method performs a sequence of such transitions and thereby examines a succession of adjacent extreme points. A transition to an adjacent extreme point will be made only if by doing so the objective function is improved (or stays the same). It is a property of linear programming problems that this type of search will lead us to the discovery of an optimal solution (if one exists). The Simplex Method is not only successful in this sense, but it is remarkably efficient because it succeeds after examining only a fraction of the basic feasible solutions.

Since the Simplex Method is an algorithm, we must specify how an initial feasible solution is obtained, how a transition is made to a better basic feasible solution, and how to recognize an optimal solution. From any basic feasible solution, we have the assurance that, if a better solution exists at all, then there is an adjacent solution that is better than the current one. This is the principle on which the Simplex Method is based; thus, an optimal solution is accessible from any starting basic feasible solution.

3.4 The Simplex Method

We will use the following simple problem from M.W. Carter and C.C. Price.

$$\begin{aligned} \max z = \quad & 8x_1 + 5x_2 & \\ \text{s.t.} \quad & x_1 & \leq 150 \\ & x_2 & \leq 250 \\ & 2x_1 + x_2 & \leq 500 \\ & x_1, x_2 & \geq 0. \end{aligned}$$

The standard form of this problem is

$$\begin{aligned} \max z = \quad & 8x_1 + 5x_2 + 0s_1 + 0s_2 + 0s_3 & \\ \text{s.t.} \quad & x_1 + s_1 & = 150 \\ & x_2 + s_2 & = 250 \\ & 2x_1 + x_2 + s_3 & = 500 \\ & x_1, x_2, s_1, s_2, s_3 & \geq 0. \end{aligned}$$

(Zero coefficients are given to the slack variables in the objective function because slack variables do not contribute to z.) The constraints constitute a system of $m = 3$ equations in $n = 5$ unknowns. To obtain an initial basic feasible solution, we need to select $n - m = 5 - 3 = 2$ variables as non-basic variables. We can readily see in this case that by choosing the two variables x_1, x_2 as the non-basic variables, and setting their values to zero, no significant computation is required in order to solve for the three basic variables: $s_1 = 150, s_2 = 250, s_3 = 500$. The value of the objective function at this solution is 0.

Once we have a solution, a transition to an adjacent solution is made by a *pivot operation*. A pivot operation is a sequence of elementary row operations applied to the current system of equations, with the effect of creating an equivalent system in which one new (previously non-basic) variable now has a coefficient of one in one equation and zeros in all other equations.

During the process of applying pivot operations to an LP problem, it is convenient to use a tabular representation of the system of equations. This representation is referred to as a *Simplex tableau*.

In order to conveniently keep track of the value of the objective function as it is affected by the pivot operations, we treat the objective function as one of the equations in

the system of equations, and we include it in the tableau. In our example, the objective function equation is written as

$$1z - 8x_1 - 5x_2 - 0s_1 - 0s_2 - 0s_3 = 0.$$

The tableau for the initial solution is as follows:

Basis	z	x_1	x_2	s_1	s_2	s_3	Solution
z	1	-8	-5	0	0	0	0
s_1	0	1	0	1	0	0	150
s_2	0	0	1	0	1	0	250
s_3	0	2	1	0	0	1	500

Observe that the objective function row represents an equation that must be satisfied for any feasible solution. Since we want to maximize z, some other (non-basic) term must decrease in order to offset the increase in z. But all of the basic variables are already at their lowest value, zero. Therefore, we want to increase some non-basic variable that has a negative coefficient. As a simple rule, we will choose the variable with the most negative coefficient. The chosen variable is called the *entering variable*, i.e., the one that will enter the basis. In our example, x_1 is the entering variable. In general, we denote the entering variable by x_k.

How much can we increase the value of x_k (away from zero)? To answer this question consider a row i with $a_{ik} > 0$. For basic variable x_i, after x_k is increased, the new value of x_i will be

$$x_i = b_i - a_{ik}x_k.$$

Since $x_i \geq 0$, we can increase x_i only to the point where

$$x_k = b_i/a_{ik}.$$

One of the basic variables x_i must leave the basis (*leaving variable*). To find this variable we consider the column k of the entering variable and calculate $\Theta_i = b_i/a_{ik}$ for every row i for which $a_{ik} > 0$. In our example $k = 1$, and $\Theta_1 = 150/1 = 150$ and $\Theta_3 = 500/2 = 250$. This means x_1 can be increased to 150 without s_1 becoming negative and x_1 can be increased to 250 without s_3 becoming negative. We do not want either of s_1, s_3 to become negative, and thus we choose $\Theta = \min \Theta_j = 150$. So, s_1 is the leaving variable.

Let us consider what happens when none of the a_{ik} is positive. In this case, x_k can be increased by any amount without any basic variable becoming negative. This means that the corresponding LP problem is unbounded.

Returning to our example, recall that x_1 is entering and s_1 is leaving. This means that the intersection of the row of x_1 and the column of s_1 is the *pivot element*. The pivot

element must become 1 and the other entries of the column of x_1 must become zero. To achieve this we multiply the 1st row by 8 and add to the row 0 (objective function row), we also multiply the 1st row by -2 and add to the 3rd row. As a result, we get

Basis	z	x_1	x_2	s_1	s_2	s_3	Solution
z	1	0	-5	8	0	0	1200
x_1	0	1	0	1	0	0	150
s_2	0	0	1	0	1	0	250
s_3	0	0	1	-2	0	1	200

This shows the new basic feasible solution $x_1 = 150$, $s_2 = 250$, $s_3 = 200$, $x_2 = s_1 = 0$, $z = 1200$.

Now x_2 is entering and $\Theta = \min\{250/1, 200/1\} = 200$. Thus, s_3 is leaving. The intersection of the row of s_3 and the column of x_2 is the pivot element and every entry of the column of x_2 except for the pivot element must become 0. After making those entries 0, we get

Basis	z	x_1	x_2	s_1	s_2	s_3	Solution
z	1	0	0	-2	0	5	2200
x_1	0	1	0	1	0	0	150
s_2	0	0	0	2	1	-1	50
x_2	0	0	1	-2	0	1	200

Now s_1 is entering and $\Theta = \min\{150/1, 50/2\} = 25$. Thus, s_2 is leaving. The intersection of the row of s_2 and the column of s_1 is the pivot element and every entry of the column of s_1 except for the pivot element must become 0. After making those entries 0, we get

Basis	z	x_1	x_2	s_1	s_2	s_3	Solution
z	1	0	0	0	1	4	2250
x_1	0	1	0	0	-1/2	1/2	125
s_1	0	0	0	1	1/2	-1/2	25
x_2	0	0	1	0	1	0	250

Because the objective function row coefficients are all non-negative (the Solution row is not taken into consideration), the current solution is optimal. The optimal values of the decision variables are $x_1 = 125$ and $x_2 = 250$. The objective optimal value $z^* = 2250$.

Example 3.4.1. *(W.L. Winston) Solve the following LP problem using the Simplex Method:*

$$\begin{aligned} \max z = \quad & 60x_1 + 30x_2 + 20x_3 \\ s.t. \quad & 8x_1 + 6x_2 + x_3 \leq 48 \\ & 4x_1 + 2x_2 + 1.5x_3 \leq 20 \end{aligned}$$

$$2x_1 + 1.5x_2 + 0.5x_3 \leq 8$$
$$x_2 \leq 5$$
$$x_1, x_2, x_3 \geq 0.$$

Solution:

Computations are done in the following tableau:

Basis	z	x_1	x_2	x_3	s_1	s_2	s_3	s_4	Solution	Ratio
z	1	-60	-30	-20	0	0	0	0	0	
s_1	0	8	6	1	1	0	0	0	48	48/8
s_2	0	4	2	1.5	0	1	0	0	20	20/4
s_3	0	**2**	1.5	0.5	0	0	1	0	8	8/2
s_4	0	0	1	0	0	0	0	1	5	–
z	1	0	15	-5	0	0	30	0	240	
s_1	0	0	0	-1	1	0	-4	0	16	–
s_2	0	0	-1	**0.5**	0	1	-2	0	4	4/0.5
x_1	0	1	0.75	0.25	0	0	0.5	0	4	4/0.25
s_4	0	0	1	0	0	0	0	1	5	–
z	1	0	5	0	0	10	10	0	280	
s_1	0	0	-2	0	1	2	-8	0	24	
x_3	0	0	-2	1	0	2	-4	0	8	
x_1	0	1	1.25	0	0	-0.5	1.5	0	2	
s_4	0	0	1	0	0	0	0	1	5	

Since the objective function row contains only non-negative coefficients in all non-Solution columns, we have obtained an optimal solution: $x_1 = 2, x_2 = 0, x_3 = 8, z^* = 280$.

Example 3.4.2. *Solve the following LP problem using the Simplex Method:*

$$\begin{aligned} \min z = \quad & -36x_1 - 30x_2 + 3x_3 + 4x_4 \\ s.t. \quad & x_1 + x_2 - x_3 \leq 5 \\ & 6x_1 + 5x_2 - x_4 \leq 10 \\ & x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$

Solution: Let $z' = -z$. Then $\max z' = 36x_1 + 30x_2 - 3x_3 - 4x_4$.

Computations are done in the following tableau:

Basis	z'	x_1	x_2	x_3	x_4	s_1	s_2	Solution	Ratio
z'	1	-36	-30	3	4	0	0	0	
s_1	0	1	1	-1	0	1	0	5	5/1
s_2	0	**6**	5	0	-1	0	1	10	10/6
z'	1	0	0	3	-2	0	6	60	
s_1	0	0	1/6	-1	**1/6**	1	-1/6	10/3	20
x_1	0	1	5/6	0	-1/6	0	1/6	5/3	–
z'	1	0	2	-9	0	12	4	100	
x_4	0	0	1	-6	1	6	-1	20	–
x_1	0	1	1	-1	0	1	0	5	–

We see that the entering variable x_3 can be increased by any number, which means that the problem is unbounded. This also means that the original problem is also unbounded.

3.5 Questions

Question 3.5.1. *Transform the following LP problem into standard form and solve it using the Simplex Method:*

$$\begin{aligned} \max z = \quad & 2x_1 - x_2 + x_3 \\ s.t. \quad & 3x_1 + x_2 + x_3 \leq 60 \\ & x_1 - x_2 + 2x_3 \leq 10 \\ & x_1 + x_2 - x_3 \leq 20 \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

Question 3.5.2. *(W.L. Winston) Solve the following LP problem using the Simplex Method:*

$$\begin{aligned} \max z = \quad & 2x_2 \\ s.t. \quad & x_1 - x_2 \leq 4 \\ & -x_1 + x_2 \leq 1 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Question 3.5.3. *Solve the following modification of the previous LP problem using the Simplex Method:*

$$\max z = \quad 2x_2$$

$$\begin{aligned} s.t. \quad & x_1 - x_2 \leq 4 \\ & x_1 + 2x_2 \leq 1 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Question 3.5.4. *Solve the following LP problem using the Simplex Method:*

$$\begin{aligned} \max z = \quad & 2x_1 + x_2 \\ s.t. \quad & 3x_1 - x_2 \leq 2 \\ & 2x_1 - x_2 \leq 3 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Question 3.5.5. *Solve the following LP problem using the Simplex Method:*

$$\begin{aligned} \max z = \quad & 3x_1 + 5x_2 & \\ s.t. \quad & x_1 & \leq 4 \\ & 2x_2 & \leq 12 \\ & 2x_1 + 3x_2 & \leq 18 \\ & x_1, x_2 & \geq 0. \end{aligned}$$

Question 3.5.6. *(Hillier and Lieberman, 4.3-7) Consider the following LP problem.*

$$\begin{aligned} \max z = \quad & 5x_1 + 3x_2 + 4x_3 & \\ s.t. \quad & 2x_1 + x_2 + x_3 & \leq 20 \\ & 3x_1 + x_2 + 2x_3 & \leq 30 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

You are given the information that the non-zero variables in the optimal solution are x_2 and x_3. Describe how you can use this information to adapt the Simplex Method to solving the problem in the minimum possible number of iterations (when you start from the usual initial basic feasible solution). Do not actually perform any iterations.

3.6 Solutions

Question 3.5.1

The standard form of the LP problem is as follows:

$$\begin{aligned} \max z = \quad & 2x_1 - x_2 + x_3 \\ \text{s.t.} \quad & 3x_1 + x_2 + x_3 + s_1 = 60 \\ & x_1 - x_2 + 2x_3 + s_2 = 10 \\ & x_1 + x_2 - x_3 + s_3 = 20 \\ & x_i, s_i \geq 0, \ i = 1, 2, 3. \end{aligned}$$

The Simplex Method solution is given in the following tableau.

Basis	z	x_1	x_2	x_3	s_1	s_2	s_3	Solution	Ratio
z	1	-2	1	-1	0	0	0	0	
s_1	0	3	1	1	1	0	0	60	60/3
s_2	0	**1**	-1	2	0	1	0	10	10/1
s_3	0	1	1	-1	0	0	1	20	20/1
z	1	0	-1	3	0	2	0	20	
s_1	0	0	4	-5	1	-3	0	30	30/4
x_1	0	1	-1	2	0	1	0	10	–
s_3	0	0	**2**	-3	0	-1	1	10	10/2
z	1	0	0	3/2	0	3/2	1/2	25	
s_1			0					10	
x_1			0					15	
x_2	0	0	1	-3/2	0	-1/2	1/2	5	

Thus, the optimal solution is $x_1 = 15, x_2 = 5, x_3 = 0, z^* = 25$.

Question 3.5.2

Basis	z	x_1	x_2	s_1	s_2	Solution	Ratio
z	1	0	-2	0	0	0	
s_1	0	1	-1	1	0	4	–
s_2	0	-1	**1**	0	1	1	1/1
z	1	-2	0	0	2	2	
s_1	0	0	0	1	1	5	–
x_2	0	-1	1	0	1	1	–

Thus, the problem is unbounded, since all entries of the x_1 column are non-positive.

Question 3.5.3

Basis	z	x_1	x_2	s_1	s_2	Solution	Ratio
z	1	0	-2	0	0	0	
s_1	0	1	-1	1	0	4	–
s_2	0	1	**2**	0	1	1	1/2
z	1	1	0	0	1	1	
s_1	0	3/2	0	1	1/2	9/2	
x_2	0	1/2	1	0	1/2	1/2	

Thus, the optimal solution is $x_1 = 0,\ x_2 = 1/2,\ z^* = 1$.

Question 3.5.4

Basis	z	x_1	x_2	s_1	s_2	Solution	Ratio
z	1	-2	-1	0	0	0	
s_1	0	3	-1	1	0	2	
s_2	0	2	-1	0	1	3	

The problem is unbounded, since all entries of the x_2 column are non-positive.

Chapter 4

Linear Programming Approaches

4.1 Artificial variables and Big-M Method

Consider the following constraint: $3x_1 - 7x_2 = -5$. Since the right hand side of any LP constraint in standard form must be non-negative, to transform this constraint into standard form, we multiply it by $-1 : -3x_1 + 7x_2 = 5$.

Consider the following constraint: $3x_1 - 7x_2 \geq -9$. Since the right hand side of any LP constraint in standard form must be non-negative, to transform this constraint into standard form, we multiply it by $-1 : -3x_1 + 7x_2 \leq 9$.

If all constraints in an LP problem are of type $\leq$, then introduction of slack variables results in getting an initial feasible set of basic variables. Often, some of the constraints are equalities, others are of type $\geq$ (with non-negative right hand sides).

Consider the following LP problem:

$$\begin{aligned} \max z = \quad & 2x_1 - 3x_2 + 9x_3 \\ \text{s.t.} \quad & -x_1 + x_2 - 4x_3 = -10 \\ & 7x_1 - 3x_2 - 5x_3 \leq -2 \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

After transformation into standard form, we get:

$$\begin{aligned} \max z = \quad & 2x_1 - 3x_2 + 9x_3 \\ \text{s.t.} \quad & x_1 - x_2 + 4x_3 = 10 \end{aligned}$$

$$-7x_1 + 3x_2 + 5x_3 - s_1 = 2$$
$$x_1, x_2, x_3, s_1 \geq 0.$$

It is not easy to find an initial set of basic variables. Thus, we introduce so-called *artificial variables* a_1 and a_2:

$$\begin{aligned} \max z = \quad & 2x_1 - 3x_2 + 9x_3 - Ma_1 - Ma_2 \\ \text{s.t.} \quad & x_1 - x_2 + 4x_3 + a_1 = 10 \\ & -7x_1 + 3x_2 + 5x_3 - s_1 + a_2 = 2 \\ & x_1, x_2, x_3, s_1, a_1, a_2 \geq 0, \end{aligned}$$

where M is a very large positive number. The role of M is to make sure that if the LP problem in hand has a feasible solution, then the optimal solution of this new problem will include $a_1 = a_2 = 0$ as positive values of a_i will decrease z considerably. Thus, $a_1 > 0$ or $a_2 > 0$ is only possible if the LP problem has no feasible solution at all. If we obtain an optimal solution for the transformed LP problem in which $a_1 = a_2 = 0$, then that solution can be considered as an initial feasible solution for the original LP problem.

The above described method is called the *Big-M Method*. There is a reason why the Big-M Method is not always considered practical. The value of M must be significantly larger than that of any other coefficient. However, this may lead to large arithmetic errors during operation of the Simplex Method. So in the end, the solution could be different from the optimal one.

Now consider another LP problem.

$$\begin{aligned} \max z = \quad & 6x_1 - 7x_2 & \\ \text{s.t.} \quad & 7x_1 + 2x_2 & \geq 8 \\ & -2x_1 - 2x_2 - x_3 & \leq -10 \\ & x_1 - 3x_2 & = 12 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

We will explain how one can solve this LP problem using the Big-M Method. We will use x_3 as one of the basic variables. We will not attempt to actually solve the problem.

Since the second constraint has a basic variable, x_3, we need only to introduce two surplus variables s_1, s_2 and two artificial variables a_1, a_2.

$$\begin{aligned} \max z = \quad & 6x_1 - 7x_2 - Ma_1 - Ma_2 & \\ \text{s.t.} \quad & 7x_1 + 2x_2 - s_1 + a_1 & = 8 \\ & 2x_1 + 2x_2 - s_2 + x_3 & = 10 \\ & x_1 - 3x_2 + a_2 & = 12 \\ & x_1, x_2, x_3, a_1, a_2, s_1, s_2 & \geq 0. \end{aligned}$$

The purpose of introducing artificial variables is to use them in the initial basis that should be replaced by non-artificial variables after a few iterations of Simplex Method. M is used to make sure that the above will happen unless the LP problem at hand does not have a feasible solution.

4.2 Two-Phase Method

This method of solving an (initial) LP problem (with artificial variables) consists of two phases:

First phase: Solve a minimization LP problem, whose objective function is a sum of artificial variables, and whose constraints are the constraints of the initial LP problem. If the optimal solution contains a positive artificial variable, then the initial problem is infeasible. Otherwise, we proceed to the next phase.

Second phase: Solve the initial problem using the optimal solution from the first phase as a starting solution.

Consider the following LP problem from M.W. Carter and C.C. Price:

$$\begin{aligned} \max z = \quad & x_1 + 3x_2 \\ \text{s.t.} \quad & 2x_1 - x_2 \leq -1 \\ & x_1 + x_2 = 3 \\ & x_1, x_2 \geq 0. \end{aligned}$$

In the first phase we solve

$$\begin{aligned} \max z_a = \quad & -a_1 - a_2 \\ \text{s.t.} \quad & -2x_1 + x_2 - s_1 + a_1 = 1 \\ & x_1 + x_2 + a_2 = 3 \\ & x_1, x_2, s_1, a_1, a_2 \geq 0, \end{aligned}$$

The initial tableau for this phase is

Basis	z_a	x_1	x_2	s_1	a_1	a_2	Solution
z_a	1	0	0	0	1	1	0
a_1	0	-2	1	-1	1	0	1
a_2	0	1	1	0	0	1	3

We perform row operations to change the coefficients of a_1 and a_2 in the z row to 0 (a necessary condition to start the Simplex Method with basis a_1, a_2). To do that, we add the a_1, a_2 rows each multiplied by -1 to the z row. We get:

Basis	z_a	x_1	x_2	s_1	a_1	a_2	Solution
z_a	1	1	-2	1	0	0	-4
a_1	0	-2	1	-1	1	0	1
a_2	0	1	1	0	0	1	3

After two iterations of the Simplex Method we get the following final tableau (perform the iterations yourself):

Basis	z_a	x_1	x_2	s_1	a_1	a_2	Solution
z_a	1	0	0	0	1	1	0
x_2	0	0	1	-1/3	1/3	2/3	7/3
x_1	0	1	0	1/3	-1/3	1/3	2/3

This indicates that the initial LP problem has a feasible solution (the a_i are not basic). This allows us to proceed to the second phase in which we replace the tableau with a new one in which the columns of a_i are deleted and the objective function row is replaced by that of the initial problem. We get

Basis	z	x_1	x_2	s_1	Solution
z	1	-1	-3	0	0
x_2	0	0	1	-1/3	7/3
x_1	0	1	0	1/3	2/3

Now we need to create zeroes in the objective function row in place of -1 and -3 since x_1 and x_2 are basic. To do this, we add the x_1 row to the z row and the x_2 row multiplied by 3 to the z row. We get

Basis	z	x_1	x_2	s_1	Solution
z	1	0	0	-2/3	23/3
x_2	0	0	1	-1/3	7/3
x_1	0	1	0	1/3	2/3

Applying one more iteration of the Simplex Method we get

Basis	z	x_1	x_2	s_1	Solution
z	1	2	0	0	9
x_2	0	1	1	0	3
s_1	0	3	0	1	2

The optimal solution is $x_1 = 0, x_2 = 3, z^* = 9$.

If one of the artificial variables remains positive in the optimal solution of the transformed problem (containing a_is), then the original problem is infeasible. It may require some amount of computational time to discover this. Still it can be quite useful to make

a note of the artificial variables that remain positive. Should infeasibility occur in any real world problem, then it usually indicates an error in the formulation of the particular constraint associated with such an artificial variable. Knowing where the error is likely to be found, it may more easily be corrected. This remark applies equally to the Big-M Method.

4.3 Dual problem

We say that an LP problem is in *normal form* if it is a maximization problem and all constraints are $\leq$ inequalities, i.e.

$$\begin{aligned} \max z = \quad & c_1x_1 + c_2x_2 + \ldots + c_nx_n \\ \text{s.t.} \quad & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \leq b_2 \\ & \ldots\ldots\ldots \\ & \ldots\ldots\ldots \\ & a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq b_m \\ & x_1, x_2, \ldots, x_n \geq 0, \end{aligned}$$

Observe that we do not require that $b_i \geq 0$.

In short matrix-vector form, we have

$$\begin{aligned} \max \quad & z = cx \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0. \end{aligned}$$

The following LP problem is called *dual* (the problem above is called *primal*):

$$\begin{aligned} \min w = \quad & b_1y_1 + b_2y_2 + \ldots + b_my_m \\ \text{s.t.} \quad & a_{11}y_1 + a_{21}y_2 + \ldots + a_{m1}y_m \geq c_1 \\ & a_{12}y_1 + a_{22}y_2 + \ldots + a_{m2}y_m \geq c_2 \\ & \ldots\ldots\ldots \\ & \ldots\ldots\ldots \\ & a_{1n}y_1 + a_{2n}y_2 + \ldots + a_{mn}y_m \geq c_n \\ & y_1, y_2, \ldots, y_m \geq 0. \end{aligned}$$

In short matrix-vector form, we have

$$\begin{aligned} \min \quad & w = by \\ \text{s.t.} \quad & A^T y \geq c \\ & y \geq 0, \end{aligned}$$

where A^T is the transposition of A, i.e. rows of A become columns in A^T and columns in A rows in A^T.

Example 4.3.1. *Find the dual of the following LP problem.*

$$\begin{aligned} \min z = \quad & -50x_1 + 100x_2 - x_3 & \\ s.t. \quad & 7x_1 + 2x_2 + 2x_3 & \geq 28 \\ & 2x_1 + 12x_2 & \leq 24 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

Solution: We first transform the original LP problem into normal form:

$$\begin{aligned} \max z' = \quad & 50x_1 - 100x_2 + x_3 & \\ \text{s.t.} \quad & -7x_1 - 2x_2 - 2x_3 & \leq -28 \\ & 2x_1 + 12x_2 & \leq 24 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

Now we can find the dual:

$$\begin{aligned} \min w = \quad & -28y_1 + 24y_2 & \\ \text{s.t.} \quad & -7y_1 + 2y_2 & \geq 50 \\ & -2y_1 + 12y_2 & \geq -100 \\ & -2y_1 & \geq 1 \\ & y_1, y_2 & \geq 0. \end{aligned}$$

Example 4.3.2. *Find the dual of the following LP problem.*

$$\begin{aligned} \min z = \quad & -50x_1 + 100x_2 - x_3 & \\ s.t. \quad & 7x_1 + 2x_2 + 2x_3 & = 28 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

Solution: We get

$$\begin{aligned} \max z' = \quad & 50x_1 - 100x_2 + x_3 & \\ \text{s.t.} \quad & 7x_1 + 2x_2 + 2x_3 & \leq 28 \\ & 7x_1 + 2x_2 + 2x_3 & \geq 28 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

Normal form:

$$\begin{aligned} \max z' = \quad & 50x_1 - 100x_2 + x_3 & \\ \text{s.t.} \quad & 7x_1 + 2x_2 + 2x_3 & \leq 28 \\ & -7x_1 - 2x_2 - 2x_3 & \leq -28 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

Dual:

$$\begin{aligned} \min w = \quad & 28y_1 - 28y_2 & \\ \text{s.t.} \quad & 7y_1 - 7y_2 & \geq 50 \\ & 2y_1 - 2y_2 & \geq -100 \\ & 2y_1 - 2y_2 & \geq 1 \\ & y_1, y_2 & \geq 0. \end{aligned}$$

There is a very apparent structural similarity between the primal and the dual in a dual pair of problems, but how are their solutions related? In the course of solving a (primal) maximization problem, the Simplex Method generates a series of feasible solutions with successively *larger* objective function values (cx). Solving the corresponding (dual) minimization problem may be thought of as a process of generating a series of feasible solutions with successively *smaller* objective function values (by). Assuming that an optimal solution does exist, the current objective function value for the primal problem will converge to its maximum value from below. The primal objective function evaluated at x never exceeds the dual objective function evaluated at y; and at optimality, the two problems actually have the same objective function value. This can be summarized in the following *duality property*:

Duality Property: If x and y are feasible solutions to the primal and dual problems, respectively, then $cx \leq by$ throughout the optimization process; and finally at optimality $cx^* = by^*$.

If follows from this property that, if feasible objective function values are found for a primal and dual pair of problems, and if these values are equal to each other, then both of the solutions are optimal solutions.

Not only do primal and dual problems share the same objective function values at their optima. In order to find the complete solutions to both problems, it is actually sufficient to solve just one of them by the Simplex Method. In fact, the *shadow prices*, which appear in the top row of the optimal tableau of the primal problem, are precisely the *optimal values of the dual variables*. Similarly, if the dual problem were solved using the Simplex Method, the shadow prices in the optimal tableau would be the optimal values of the primal variables.

In the illustrative problem (considered in Section "Simplex Method")

$$\begin{aligned} \max z = \quad & 8x_1 + 5x_2 & \\ \text{s.t.} \quad & x_1 & \leq 150 \\ & x_2 & \leq 250 \\ & 2x_1 + x_2 & \leq 500 \\ & x_1, x_2 & \geq 0 \end{aligned}$$

the dual objective of minimizing $w = 150y_1 + 250y_2 + 500y_3$ is met when the dual variables (shadow prices) have the values $y_1 = 0, y_2 = 1, y_3 = 4$. Thus, from the dual point of view,

$$w^* = 150 \times 0 + 250 \times 1 + 500 \times 4 = 2250,$$

which is equal to the primal objective value

$$z^* = 8x_1 + 5x_2 = 8 \times 125 + 5 \times 250 = 2250$$

for optimal x values of $x_1 = 125, x_2 = 250$.

Because the pertinent parameters and goals of any LP problem can be expressed in either a primal or dual form, and because solving either problem yields enough information to easily construct a solution to the other, we might reasonably wonder which problem, primal or dual, should we solve when using the Simplex Method. From the standpoint of computational complexity, we might wish to choose to solve the problem with the fewest constraints. This choice becomes more compelling when the LP problem has thousands of constraints, and of much less importance for more moderate-sized problems of a few hundred or less constraints.

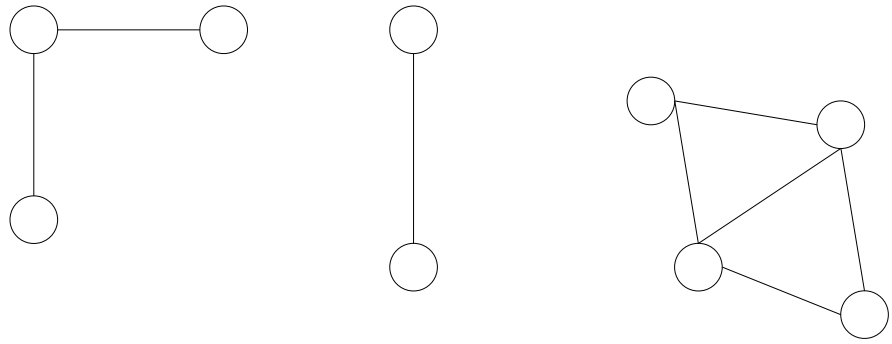


Figure 4.1: Disconnected graph with three components

4.4 Decomposition of LP problems

Even though the Simplex Method is relatively fast, it is too slow when the numbers n and m are very large. Practical problems sometimes require solution of LP problems with n and m being several tens of thousands. At the same time practical problems have *sparse matrices* A in which at least 95% of the entries equal 0. For this reason, several methods have been derived to deal with large sparse LP problems. Here we present one of them. We start with a few notions in graph theory.

A graph $G = (V, E)$ is *connected* if there is a path between every pair of vertices in G. (G is "in one piece".) If G is not connected, then it consists of several *connectivity components* that are largest connected subgraphs of G (i.e., "pieces" of G). There are very fast algorithms to find connectivity components of graphs such as depth-first search (DFS). See Figure 4.1 for a graph with three connectivity components.

Let $Ax = b,\ x \geq 0$ be the constraints of an LP problem in standard form. We construct a graph G corresponding to A as follows. The variables x_i of A correspond to vertices v_i of G. Two variables x_i and x_j are linked by an edge in G if and only if they are in the same constraint (row of A). If G is not connected, then an LP problem with the constraints $Ax = b,\ x \geq 0$ can be *decomposed* into several LP problems of smaller sizes. Let us consider the following simple example.

An objective function is $\max z = 2x_1 + 6x_2 + 3x_3$. The constraints are $x_1 + 5x_3 \leq 7$, $x_2 \leq 5$, $x_1, x_2, x_3 \geq 0$.

The graph G has vertices v_1, v_2, v_3 and edge v_1v_3. So, it has two components with vertex sets $\{v_1, v_3\}$ and $\{v_2\}$. See Figure 4.2. This shows that the initial LP problem can be decomposed into two problems, one with variables x_1 and x_3, and the other with just one variable x_2, as follows.

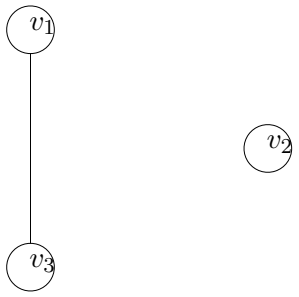


Figure 4.2: Two components

1. The problem with objective function $\max z' = 2x_1 + 3x_3$ and constraints $x_1 + 5x_3 \leq 7$, $x_1, x_3 \geq 0$;

2. The problem with objective function $\max z'' = 6x_2$ and constraints $x_2 \leq 5$, $x_2 \geq 0$.

Problem 1 has optimal solution $x_1 = 7, x_3 = 0, z' = 14$. Problem 2 has optimal solution $x_2 = 5, z'' = 30$. Thus, the optimal solution of the initial problem is $x_1 = 7, x_2 = 5, x_3 = 0, z = 44$.

4.5 LP software

LP problems are normally solved on computers due to a significant amount of calculation needed. One of the methods normally implemented in commercial and free LP software is *Revised Simplex Algorithm*, a variation of the Simplex Method implemented in matrix form and avoiding unnecessary computations. Another algorithm often implemented in commercial software is the *Interior Point Method* (IPM) . Unlike the Simplex Method, which is not of polynomial time complexity in the worst case, IPM is of polynomial complexity. In practice, IPM is normally slower than Simplex for $n + m \leq 2000$ and both methods compete evenly for $2000 \leq n + m \leq 10000$. Many software packages allow to combine Simplex with IPM when solving very large LP problems.

There is a number of commercial, open access and free LP solvers (packages for LP problems), see e.g. `http://www.statistik.tuwien.ac.at/forschung/CS/CS-2012-1complete.pdf`.

4.6 Questions

Question 4.6.1. *Why do we need the Two-Phase Method? Provide an example of an LP problem, where the Two-Phase Method is needed.*

Question 4.6.2. *(W.L. Winston) Using the Two-Phase Method solve the following:*

$$\begin{aligned} \max z = \quad & -2x_1 - 3x_2 \\ s.t. \quad & \tfrac{1}{2}x_1 + \tfrac{1}{4}x_2 \leq 4 \\ & x_1 + 3x_2 \geq 20 \\ & x_1 + x_2 = 10 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Hint: Use s_1 in the first constraint as a basic variable.

Question 4.6.3. *Find the dual of the following LP problem.*

$$\begin{aligned} \min z = \quad & 7x_1 - 100x_2 - x_3 & \\ s.t. \quad & 9x_1 + 12x_2 - 2x_3 & \leq 18 \\ & -2x_1 + 22x_2 - x_3 & \geq 14 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

Question 4.6.4. *Find the dual of the following LP problem.*

$$\begin{aligned} \max z = \quad & 8x_1 + 100x_2 - 5x_3 & \\ s.t. \quad & 9x_1 - 12x_2 - 9x_3 & = 8 \\ & 2x_1 - 22x_2 + x_3 & \geq 4 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

Question 4.6.5. *Formulate the Duality Property in Linear Programming.*

Question 4.6.6. *Let x' and y' be feasible solutions to the primal and dual LP problems, respectively. Suppose that $cx' = y'b$, i.e., the objective functions of both problems coincide for x' and y', respectively. Why are both x' and y' optimal?*

Question 4.6.7. *Explain the main ideas of the Decomposition Method in Linear Programming (an application of graphs) using your own example.*

Question 4.6.8. *Transform the LP problem*

$$\begin{aligned} \max z = \quad & 4x_1 - 5x_2 - 3x_3 & \\ s.t. \quad & x_1 - x_2 + x_3 & \geq -2 \\ & x_1 + x_2 + 2x_3 & \leq 3 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

into normal form and find the dual. The value of the optimal solution of the dual is $w_{opt} = 12$. *Why for* $x_1 = 4, x_2 = x_3 = 0$ *do we have* $z = 16 > w_{opt}$*?*

Question 4.6.9. *Find the optimal value of the objective function of the following LP problem by directly solving the dual.*

$$\begin{aligned} \max z = \quad & 6x_1 - 15x_2 - 4x_3 & \\ s.t. \quad & -3x_1 + 2x_2 - 2x_3 & \geq -20 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

4.7 Solutions

Question 4.6.2

In the first phase we solve

$$\begin{aligned} \max z_a = \quad & -a_1 - a_2 \\ \text{s.t.} \quad & \tfrac{1}{2}x_1 + \tfrac{1}{4}x_2 + s_1 = 4 \\ & x_1 + 3x_2 - s_2 + a_1 = 20 \\ & x_1 + x_2 + a_2 = 10 \\ & x_1, x_2, s_1, s_2, a_1, a_2 \geq 0, \end{aligned}$$

(We do not need an artificial variable for the first constraint since s_1 can and will be used for the initial basis.) The initial tableau for this phase is

Basis	z_a	x_1	x_2	s_1	s_2	a_1	a_2	Solution
z_a	1	0	0	0	0	1	1	0
s_1	0	1/2	1/4	1	0	0	0	4
a_1	0	1	3	0	-1	1	0	20
a_2	0	1	1	0	0	0	1	10

We perform row operations to change the coefficients of a_1 and a_2 in the z row to 0 (a necessary condition to start Simplex Method with basis a_1, a_2). To do that, we add the a_1, a_2 rows multiplied by -1 to the z row. We get:

Basis	z_a	x_1	x_2	s_1	s_2	a_1	a_2	Solution	Ratio
z_a	1	-2	-4	0	1	0	0	-30	
s_1	0	1/2	1/4	1	0	0	0	4	16
a_1	0	1	**3**	0	-1	1	0	20	20/3
a_2	0	1	1	0	0	0	1	10	10/1

After the first iteration of the Simplex Method we get:

Basis	z_a	x_1	x_2	s_1	s_2	a_1	a_2	Solution	Ratio
z_a	1	-2/3	0	0	-1/3	4/3	0	-10/3	
s_1	0	5/12	0	1	1/12	-1/12	0	7 /3	28/5
x_2	0	1/3	1	0	-1/3	1/3	0	20/3	20/1
a_2	0	**2/3**	0	0	1/3	-1/3	1	10/3	10/2

After two iterations of the Simplex Method we have:

Basis	z_a	x_1	x_2	s_1	s_2	a_1	a_2	Solution	Ratio
z_a	1	0	0	0	0	1	1	0	
s_1	0	0	0	1	-1/8	1/8	-5/8	1/4	
x_2	0	0	1	0	-1/2	1/2	-1/2	5	
x_1	0	1	0	0	1/2	-1/2	3/2	5	

This indicates that the initial LP problem has a feasible solution (a_i are not basic). This allows us to proceed to the second phase in which we replace the tableau with a new one in which the columns of a_i are deleted and the objective function row is replaced by that of the initial problem. We get:

Basis	z	x_1	x_2	s_1	s_2	Solution	Ratio
z	1	2	3	0	0	0	
s_1	0	0	0	1	-1/8	1/4	
x_2	0	0	1	0	-1/2	5	
x_1	0	1	0	0	1/2	5	

After adding to the z row the x_1 row multiplied by -2 and the x_2 row multiplied by -3, we have:

Basis	z	x_1	x_2	s_1	s_2	Solution	Ratio
z	1	0	0	0	1/2	-25	
s_1	0	0	0	1	-1/8	1/4	
x_2	0	0	1	0	-1/2	5	
x_1	0	1	0	0	1/2	5	

This is an optimal tableau for the second phase, i.e., we do not need to enter the second phase. The optimal solution is $x_1 = x_2 = 5,\ z^* = -25$.

Question 4.6.8

Normal form:

$$\begin{aligned} \max z = \quad & 4x_1 - 5x_2 - 3x_3 & \\ \text{s.t.} \quad & -x_1 + x_2 - x_3 & \leq 2 \\ & x_1 + x_2 + 2x_3 & \leq 3 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

The dual:

$$\begin{aligned} \min w = \quad & 2y_1 + 3y_2 & \\ \text{s.t.} \quad & -y_1 + y_2 & \geq 4 \\ & y_1 + y_2 & \geq -5 \\ & -y_1 + 2y_2 & \geq -3 \\ & y_1, y_2 \geq 0. & \end{aligned}$$

$z = 16 > w_{opt}$ because $x_1 = 4, x_2 = x_3 = 0$ is not a feasible solution of the primal.

Question 4.6.9

Normal form:

$$\begin{aligned} \max z = \quad & 6x_1 - 15x_2 - 4x_3 & \\ \text{s.t.} \quad & 3x_1 - 2x_2 + 2x_3 & \leq 20 \\ & x_1, x_2, x_3 & \geq 0. \end{aligned}$$

Dual:

$$\begin{aligned} \min w = \quad & 20y & \\ \text{s.t.} \quad & 3y \geq 6,\ -2y \geq -15\ 2y \geq -4 & \\ & y & \geq 0. \end{aligned}$$

This is equivalent to

$$\begin{aligned} \min w = \quad & 20y \\ \text{s.t.} \quad & 2 \leq y \leq 7.5. \end{aligned}$$

Thus, $z_{opt} = w_{opt} = 20 \times 2 = 40$.

Chapter 5

Integer Programming Modeling

5.1 Integer Programming vs. Linear Programming

For many LP problems, we cannot be satisfied by non-integer values of decision variables x_i. Indeed, we cannot be satisfied if $x_3 = 12.3$, where x_3 is the number of lorries required to transport a certain product from place to place. LP problems with the additional requirement that all decision variables are integers are called *Integer Programming* (IP) problems.

An obvious approach to solving IP problems is to "forget" the integrality requirement (i.e., that the decision variables are integer) and solve the corresponding LP problem (called the *LP relaxation* of the IP problem). In general, the relaxation will give us fractional values of decision variables. In certain cases, rounding up or down the decision variables will give if not optimal then near-optimal solutions of the IP problem, but there are many IP problems for which the rounding up or down procedure will often bring "bad" solutions.

One such family of IP problems are so-called 0,1-problems, in which all decision variables are required to be 0 or 1 (they are IP problems as we can require $0 \leq x_i \leq 1$ and x_i is integer for every x_i). Indeed, our choice, for each x_i is 0 or 1 and we often do not have enough information to decide whether to choose 0 or 1.

Observe that, unlike LP problems, IP problems do not have continuous feasible region. For example, see a graph of a simple two-dimensional IP problem in Figure 5.1.

We will start our Integer Programming part of the course by considering some very important IP problems.

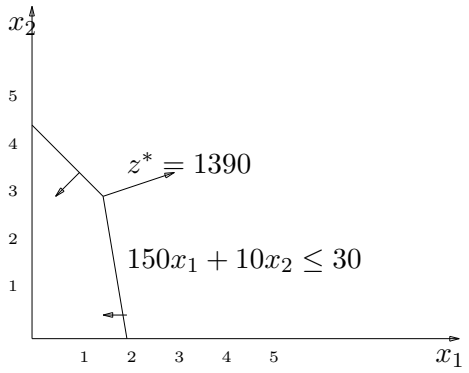


Figure 5.1: Graphical representation of an IP problem

5.2 IP problems

We will provide IP formulations of a few important optimisation problems.

5.2.1 Travelling salesman problem

We have already considered the travelling salesman problem (TSP). In short, the TSP is the problem of visiting a number of cities and come back to the point of origin, all in the cheapest possible way. This is one of the most challenging and most extensively studied problems in the field of combinatorics. The formulation is deceptively simple, and yet it has proven to be notoriously difficult to solve. Define zero-one variables $x_{ij} = 1$ if city i is visited immediately prior to city j. Let c_{ij} represent the distance between cities i and j. Suppose that there are n cities that must be visited. Then the TSP can be expressed as:

$$\begin{aligned}
\min z = \quad & \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} & \\
\text{s.t.} \quad & \textstyle\sum_{i=1}^{n} x_{ij} & = 1, \ j = 1, 2, \ldots, n \\
& \textstyle\sum_{j=1}^{n} x_{ij} & = 1, \ i = 1, 2, \ldots, n \\
& \textstyle\sum_{i \in S} \sum_{j \in S} x_{ij} & \leq |S| - 1 \text{ for all } |S| < n \\
& x_{ij} & = 0 \text{ or } 1, \quad i, j = 1, \ldots, n.
\end{aligned}$$

The first constraint says that you must go into every city j exactly once, and the second constraint says that you must leave every city i exactly once. These constraints ensure that there are two edges adjacent to each city, one in and one out, as we would expect. However, this does not prevent so-called sub-tours. A sub-tour occurs when there is a cycle containing a subset of the cities. Instead of having one tour of all of the cities,

the solution can be composed of two or more sub-tours. The third constraint eliminates sub-tours; it states that no proper subset of cities, S, can have a total of $|S|$ edges.

The TSP has a number of practical industrial applications. Consider the problem of placing components on a circuit board. To minimize the time required to produce a board, one of the primary considerations is often the distance that a placement head has to travel between components. Another example occurs in routing trucks or ships delivering products to customers. (When we allow multiple trucks, this problem becomes the vehicle routing problem.) Another application occurs in a production environment when it is desired to minimize sequence-dependent setup times. When multiple jobs are to be processed on a machine, the total setup time for each job frequently depends on which job preceded it. This situation can be modeled as a TSP, where we sequence jobs rather than sequencing the order in which cities are visited.

5.2.2 Knapsack Problem

Assume that we have a number of items, and we must choose some subset of the items to fill our "knapsack", which has limited space b. Each item, i, has a value v_i and takes up w_i units of space in the knapsack. We wish to choose a collection of the items with total space less than b and with maximum total value. Let the zero-one variables $x_i = 1$ if item i is selected, and let b represent the total space in the knapsack. Then we can formulate the knapsack problem as follows:

$$\begin{array}{rll} \max z = & \sum_{i=1}^{n} v_i x_i & \\ \text{s.t.} & \sum_{i=1}^{n} w_i x_i & \leq b \\ & x_i & = 0 \text{ or } 1 \text{ for all } i. \end{array}$$

The zero-one version of the knapsack problem states that every item is unique, and each can either be selected or not. A slight generalization of the knapsack problem states that you can choose more than one copy of each item, so that the variables can take on general integer values (probably with upper bounds on each variable).

5.2.3 Bin packing problem

Bin packing is somewhat similar to the knapsack problem. Suppose that we are given a set of m bins of equal size, b; and a set of n items that must be placed in the bins. Let w_i be the size of item i. We define the zero-one variable $x_{ij} = 1$ if item i is placed in bin j. Bin packing is usually expressed as a problem of minimizing the number of bins required to pack all of the items. We can let $y_j = 1$ if we use bin j and $y_j = 0$, otherwise. The objective function minimizes the number of bins required.

$$\begin{aligned} \min z = \quad & \textstyle\sum_{j=1}^{m} y_j & \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^{n} w_i x_{ij} & \leq b y_j \text{ for all } j \\ & \textstyle\sum_{j=1}^{m} x_{ij} & = 1 \text{ for all } i \\ & x_{ij} & = 0 \text{ or } 1 \text{ for all } i, j \\ & y_j & = 0 \text{ or } 1 \text{ for all } j \end{aligned}$$

Bin packing has applications in industry where, for example, there is a limited amount of work that can be assigned to each person working at stations on an assembly line. This model may also be applicable when deciding which products should be produced at each of several possible manufacturing plants, or which customer should be assigned to each delivery truck. Of course, each of these problems involves additional criteria and constraints.

5.2.4 Set partitioning/covering/packing problems

Many problems in combinatorial optimization include (as subproblems) partitioning a group of items into "optimal" subsets. For example, vehicle routing requires that we allocate customers to vehicles. Airline crew scheduling requires that we allocate flight legs to a crew. Municipal garbage pickup requires that we allocate specific street blocks to trucks. Each of these subproblems can be modeled in the following form as a *set partitioning problem*:

$$\begin{aligned} \min z = \quad & \textstyle\sum_{j=1}^{m} c_j y_j & \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{m} a_{ij} y_j & = 1 \text{ for all } i = 1, \ldots, n \\ & y_j & = 0 \text{ or } 1 \text{ for all } j = 1, \ldots, m, \end{aligned}$$

where $a_{ij} = 1$ if item i belongs to (potential) subset j and $a_{ij} = 0$, otherwise. Each column of the $n \times m$ constraint matrix A represents a feasible combination of items. For example, each column might represent the items that could feasibly be loaded into a truck for delivery to customers; or the items could be road segments that require garbage collection, and a column would represent a feasible route for a truck to pick up garbage. The cost c_j represents the cost of delivering (or travelling, or producing) that subset of items. A variable $y_j = 1$ if we decide to include that particular subset in our solution.

In the set partitioning problem, all of the items must be included exactly once. In vehicle routing, for example, we might typically require that exactly one truck travel to each customer. In a slightly different problem the *set covering problem*, we require that

each item be selected at least once. For example, in the garbage collection problem, and in the crew scheduling problem, every street (every flight leg) must be covered at least once; but it is also feasible to cover the same street (flight leg) twice, if this turns out to be the most efficient solution. (The second truck would not pick up any garbage, and the second flight crew would ride as passengers.) Set covering differs from set partitioning by having $\geq$ inequality constraints instead of equalities.

The *set packing problem* describes another similar situation. In some production scheduling problems, we are given a list of orders, and we have possible subsets of orders that can be combined on different machines. In some cases, there may not be sufficient resources to satisfy all of the demands. The problem is to select the optimal subset of orders to maximize the combined profit of those orders that are processed. This problem can be formulated as:

$$\begin{array}{rll} \max z = & \sum_{j=1}^{n} p_j x_j & \\ \text{s.t.} & \sum_{j=1}^{n} a_{ij} x_j & \leq 1 \text{ for all } i = 1, \ldots, m \\ & x_j & = 0 \text{ or } 1 \text{ for all } j = 1, \ldots, n, \end{array}$$

We select as many items as possible, but we are not allowed to process any items more than once.

5.2.5 Assignment problem

We have already stated the assignment problem in Chapter 1. We have n persons $p_1, \ldots, p_n$ and n jobs $j_1, \ldots, j_n$, and the cost c_{ij} of having person i perform job j. We wish to find an assignment of the persons to the jobs (one person per job) such that the total cost of performing the jobs is minimum. The costs are normally given by matrix $c = [c_{ij}]$.

The assignment problem (AP) can be formulated as follows.

$$\begin{array}{rll} \min z = & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} & \\ \text{s.t.} & \sum_{i=1}^{n} x_{ij} & = 1, \ j = 1, 2, \ldots, n \\ & \sum_{j=1}^{n} x_{ij} & = 1, \ i = 1, 2, \ldots, n \\ & x_{ij} & = 0 \text{ or } 1, \quad i, j = 1, \ldots, n. \end{array}$$

Thus, $x_{ij} = 1$ if employee i has been assigned to job j and $x_{ij} = 0$, otherwise. The first constraint requires every job to be assigned to exactly one employee; and the second constraint states that every employee must do exactly one job.

5.3 Questions

Question 5.3.1. *What is the difference between Linear Programming and Integer Programming problems? Why can one not in general use the Simplex algorithm to solve Integer Programming problems?*

Question 5.3.2. *Provide an Integer Programming formulation of the travelling salesman problem. Explain the meaning of all parameters and variables.*

Question 5.3.3. *Provide an Integer Programming formulation of the knapsack problem. Explain the meaning of all parameters and variables.*

Question 5.3.4. *Provide an Integer Programming formulation of the bin packing problem. Explain the meaning of all parameters and variables.*

Question 5.3.5. *Provide an Integer Programming formulation of the generalized assignment problem. Explain the meaning of all parameters and variables.*

Question 5.3.6. *(a) Of which problem is the following an instance:*

$$\begin{aligned} \max z = \quad & 2x_1 + 3x_2 + 5x_3 - 4x_4 & \\ s.t. \quad & x_1 + x_2 + 2x_3 + 7x_4 & \leq 12 \\ & x_i & = 0 \text{ or } 1 \text{ for all } i? \end{aligned}$$

(b) Formulate the problem whose instance is given in (a).

Question 5.3.7. *(a) Of which problem is the following an instance:*

$$\begin{aligned} \max z = \quad & 2x_{1,1} + x_{1,2} + 5x_{2,1} - 5x_{2,2} & \\ s.t. \quad & x_{1,1} + x_{2,1} & = 1 \\ & x_{1,2} + x_{2,2} & = 1 \\ & x_{1,1} + x_{1,2} & = 1 \\ & x_{2,1} + x_{2,2} & = 1 \\ & x_{i,j} & = 0 \text{ or } 1 \text{ for } 1 \leq i, j \leq 2? \end{aligned}$$

(b) Formulate the problem whose instance is given in (a).

Chapter 6

Branch-and-Bound Algorithm

6.1 A Simple Example for Integer and Mixed Programming

Branch-and-Bound algorithms are widely considered to be the most effective methods for solving medium-sized general integer programming problems. These algorithms make no assumptions about the structure of a problem except that the objective function and the constraints are linear. Even these restrictions can be relaxed without changing the basic framework of the technique.

In its simplest form, *Branch-and-Bound* is just an organized way of taking a hard problem and splitting it into two or more smaller (and hence easier) subproblems. If these subproblems are still too hard, we "branch" again and further subdivide the problems. The process is repeated until each of the subproblems can be easily solved. Branching is done in such a way that solving each of the subproblems (and selecting the best answer found) is equivalent to solving the original problem.

Consider the following simple example (from M.W. Carter and C.C. Price) in two variables. A manufacturer has 300 person-hours available this week and 1800 units of raw material. These resources can be used to build two products A and B. The requirements and the profit for each item are given as follows:

Product	**Person-hours**	**Raw Material**	**Profit ($)**
A	150	300	$600
B	10	400	$100

Let x_1 and x_2 represent the integer number of units of products A and B, respectively. We can formulate this problem as an integer programming (IP) problem:

$$\text{maximize} \qquad z = 600x_1 + 100x_2$$

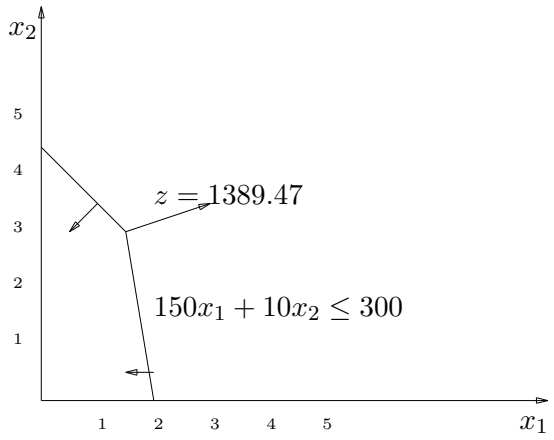


Figure 6.1: An IP problem.

$$\begin{array}{lrl} \text{subject to} & 150x_1 + 10x_2 & \leq 300 \\ & 300x_1 + 400x_2 & \leq 1800 \\ & x_1, x_2 \geq 0 \text{ and integer} & \end{array}$$

This problem is illustrated in Figure 6.1. The feasible region is given by the discrete set of integer points within the constraint region. The optimal LP solution occurs at x_1=1.789 and x_2=3.158 with a profit of z=1,389.47. Unfortunately, we cannot sell a fractional number of items. One obvious alternative is to round down both values to x_1=1 and x_2=3, for a profit of \$900. We will call the feasible integer solution $x^I = (1, 3)$ the *current incumbent* solution, and we will update the current incumbent. Before reading any further, try to locate the optimal integer solution to the problem in Figure 6.1, and consider how integer solutions might be found in general.

The basic branch-and-bound algorithm stems from the following observations: The feasible integer solution x=(1,3) with z=900 was fairly easy to find. The optimal integer solution cannot have a lower value of z than \$900 and we call this a *lower bound* on the optimal solution. Each time we find a higher valued integer solution, we replace the lower bound z^I. This is the "bound" part of branch-and-bound methods.

Over the whole feasible region, the largest possible value of z=1389.47, which is the real valued solution obtained from the LP. We call this an *upper bound* on the optimal integer function value.

The graphical solution shows that $x_2 = 3.158$. This is infeasible because it is a fractional solution. Since x_2 must be an integer, apparently either $x_2 \leq 3$ or $x_2 \geq 4$. This is equivalent to saying that x_2 cannot lie part way between 3 and 4.

Consider the following two subproblems:

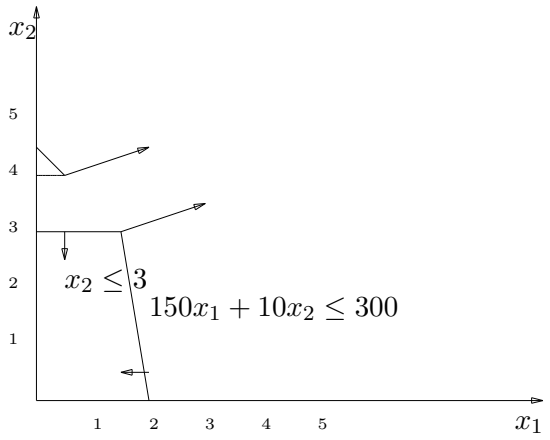


Figure 6.2: Problems (1) and (2).

$$
\begin{array}{lrl}
\text{(1) maximize} & z = 600x_1 + 100x_2 & \\
\text{subject to} & 150x_1 + 10x_2 & \leq 300 \\
& 300x_1 + 400x_2 & \leq 1800 \\
& x_1 \geq 0 \text{ and integer} & \\
& x_2 \geq 4 \text{ and integer} &
\end{array}
$$

$$
\begin{array}{lrl}
\text{(2) maximize} & z = 600x_1 + 100x_2 & \\
\text{subject to} & 150x_1 + 10x_2 & \leq 300 \\
& 300x_1 + 400x_2 & \leq 1800 \\
& x_1, x_2 \geq 0 \text{ and integer} & \\
& x_2 \leq 3 &
\end{array}
$$

Observe that if we find the best integer solutions of these subproblems, then one of them must be the optimal solution to the original problem. These subproblems are represented graphically in Figure 6.2. We say that we have *separated* on variable x_2.

Consider problem (1) first. The LP solutions occurs at $x = (2/3, 4)$ with an objective function value of $z = 800$. Notice that x_2 is now integer valued. We will see that each time we separate, the chosen variable will always be integer, although it does not necessarily stay integer on subsequent iterations.

By definition, the linear programming solution is the largest value possible for the problem. Therefore, the value z=800 is an upper bound on all possible solutions in the feasible region for problem (1). Any integer solution to (1) must be $\leq$ 800. However, we already have a feasible integer solution with $z^I = 900$. Therefore problem (1) can be ignored. It cannot contain any answer better than 900. In branch-and-bound terminology, we say that problem (1) can be "fathomed".

In general, a subproblem is called *fathomed* whenever it is no longer necessary to branch any maximization problem, when the LP solution is infeasible, or when the LP relaxation produces an integer solution.

Problem (2) has its optimal LP solution at $x = (1.8, 3)$ with a function value of $z = 1380$. This value gives us a new upper bound on the optimal integer solution. At each iteration of the branch-and-bound process, the upper and lower bounds can be revised until they eventually converge to the optimal solution. We now know that the optimal value lies between 900 and 1380. Variable x_2 is integer valued, but x_1 is still fractional. We can now further divide problem (2) into two subproblems based on the fact that $x_1 \leq 1$ or $x_1 \geq 2$ as follows:

$$
\begin{array}{lrl}
\text{(3) maximize} & z = 600x_1 + 100x_2 & \\
\text{subject to} & 150x_1 + 10x_2 & \leq 300 \\
& 300x_1 + 400x_2 & \leq 1800 \\
& x_1, x_2 \geq 0 \text{ and integer} & \\
& x_1 \leq 1 & \\
& x_2 \leq 3 &
\end{array}
$$

$$
\begin{array}{lrl}
\text{(4) maximize} & z = 600x_1 + 100x_2 & \\
\text{subject to} & 150x_1 + 10x_2 & \leq 300 \\
& 300x_1 + 400x_2 & \leq 1800 \\
& x_2 \geq 0 \text{ and integer} & \\
& x_1 \geq 2 \text{ and integer} & \\
& x_2 \leq 3 &
\end{array}
$$

For problem (3), it is easy to see that the optimal LP solution occurs at point $x = (1, 3)$ with a function value $z = 900$. Since x is now integer valued, it must be optimal for this subproblem. This subproblem is considered to be fathomed because it gives us an integer solution; there is no need for further branching. It is also considered fathomed because

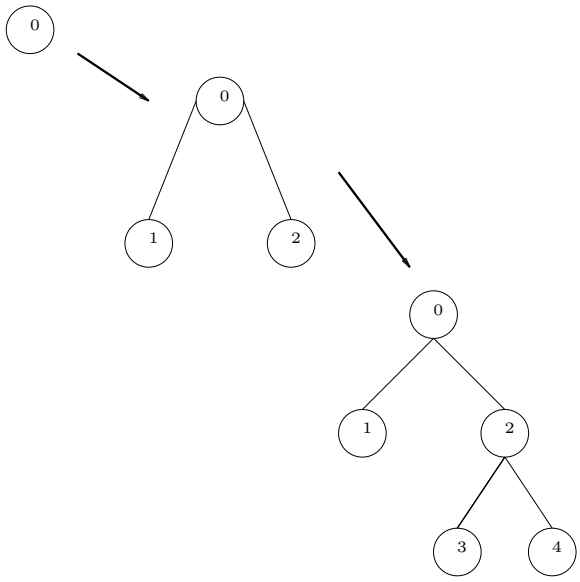


Figure 6.3: Branch-and-Bound tree

the solution of 900 is no better than the one we already obtained earlier. In either case, problem (3) is finished.

Problem (4) consists of the single point $x = (2, 0)$ with a function value of $z = 1200$. This solution is both integer, and better than the previous lower bound. Since x is integer, subproblem (4) is fathomed and no further branching is required. Our new lower bound increases to $z^I = 1200$ and $x^I = (2, 0)$ becomes the new current incumbent.

At this point, we observe that all of our subproblems have been fathomed. Therefore, $x^I = (2, 0)$ is the optimal integer solution and $z^I = 1200$ is the optimal function value.

This example illustrates, in particular, the fact that the rounding up/down approach does not work in general.

It is often convenient to display this procedure in the form of a *branch-and-bound tree*. The tree corresponding to the previous example is illustrated in Figure 6.3. Each subproblem is represented by a node in the tree. Each node must either be fathomed or split into subproblems, which are shown by lower level nodes.

Now consider a mixed programming (MP) problem

$$\begin{array}{lll} \text{maximize} & z = 600x_1 + 100x_2 & \\ \text{subject to} & 150x_1 + 10x_2 & \leq 300 \end{array}$$

$$\begin{aligned} 300x_1 + 400x_2 &\leq 1800 \\ x_1, x_2 \geq 0 \text{ and } x_2 \text{ integer,} & \end{aligned}$$

which is a modification of the IP problem above. To solve this problem by the branch-and-bound algorithm, it suffices to consider only Problems (1) and (2). Problem (2) gives the optimal solution to the MP problem: $x_1 = 1.8, x_2 = 3, z^* = 1380$.

6.2 Knapsack example

Solve the following instance of the knapsack problem:

$$\begin{aligned} \max z = \quad & 10x_1 + 8x_2 + 4x_3 + 7x_4 \\ \text{s.t.} \quad & 2x_1 + 2x_2 + 4x_3 + 5x_4 \leq 8 \\ & x_1, x_2, x_3, x_4 = 0 \text{ or } 1. \end{aligned}$$

Recall: When the 0-1 constraints are relaxed to solve the LP, we replace them with the linear constraints:

$$0 \leq x_i \leq 1.$$

To solve the LP relaxation of the knapsack problem the ratio choice rule defined below is used. Recall the general knapsack formulation:

$$\begin{aligned} \max z = \quad & \textstyle\sum_{j=1}^{n} v_j x_j \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{n} w_j x_j \leq b \\ & x_i = 0 \text{ or } 1 \text{ for all } i, \end{aligned}$$

where v_i denotes the value of item i and w_i denotes the unit of space of item i.

The LP relaxation of the knapsack problem is:

$$\begin{aligned} \max z = \quad & \textstyle\sum_{j=1}^{n} v_j x_j \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{n} w_j x_j \leq b \\ & 0 \leq x_i \leq 1 \text{ for all } i. \end{aligned}$$

For each i we compute the ratio v_j/w_j, which indicates the relative value of item i. It is intuitively clear that we should assign $x_i = 1$ first to the item i of highest ratio v_i/w_i, than $x_i = 1$ to the one with next highest ratio, etc. When we cannot assign $x_i = 1$ to the item i of the highest remaining ratio, we assign the corresponding fraction to x_i and 0 to all x_i of smaller ratio. (Ties are broken in an arbitrary manner.) We call this rule the *ratio choice*. The ratio choice gives an optimal solution to the LP relaxation, i.e., we do not need use Simplex algorithm for the LP relaxation of the knapsack problem.

Let us return to our example and call (P0) the linear relaxation of

$$\begin{aligned} \max z = \quad & 10x_1 + 8x_2 + 4x_3 + 7x_4 \\ \text{s.t.} \quad & 2x_1 + 2x_2 + 4x_3 + 5x_4 \quad \leq 8 \\ & x_1, x_2, x_3, x_4 = 0 \text{ or } 1. \end{aligned}$$

To solve (P0), find the ratios

$$r_1 = 10/2 = 5,\ r_2 = 8/2 = 4,\ r_3 = 4/4 = 1,\ r_4 = 7/5.$$

Ordering the ratios gives, $r_1 > r_2 > r_4 > r_3$.

To obtain a solution for (P0), we will now start to fill up the knapsack in the order the ratios give us. Therefore, we assign $x_1 = 1$ ($(2 \times 1) \leq 8$, true). Next assign $x_2 = 1$ ($(2 \times 1) + (2 \times 1) = 4 \leq 8$, true). We now have to look at x_4 and we can see that it is only possible to assign a fraction of x_4 to the knapsack, i.e. $x_4 = 4/5$. Therefore, the solution of (P0) is $x_1 = x_2 = 1, x_4 = 4/5, x_3 = 0$, which is an infeasible solution of the original IP problem.

Let us now consider the subproblems (P1)=(P0) plus the extra constraint ($x_4 = 1$), and (P2)=(P0) with the extra constraint($x_4 = 0$).

To work out a solution of (P1), we assign $x_4 = x_1 = 1$ ($(5 \times 1) + (2 \times 1) \leq 8$, true) and x_2 has to be a fraction again to satisfy the constraint ($\frac{8-7}{2} = 1/2$). The solution of (P1) is $x_4 = x_1 = 1$, $x_2 = 1/2, x_3 = 0$, $z = 21$.

Finding the solution of (P2) we assign $x_4 = 0, x_1 = x_2 = 1$, as required $((5 \times 0) + (2 \times 1) + (2 \times 1) = 4 \leq 8$, true). We can now see that by assigning $x_3 = 1$ we are still satisfying the constraints and the solution is $z = 22$.

The last solution is feasible and is better than that of (P1). Thus, the optimal solution is $x_4 = 0, x_1 = x_2 = x_3 = 1$, $z = 22$.

6.3 Branching strategies

To control the selection of the next node for branching, it is typical to restrict the choice of nodes from the list of currently active nodes in one of the following ways.

The Backtracking or Depth-First-Search Strategy: Always select a node that was most recently added to the tree. Evaluate all nodes in one branch of the tree completely to the bottom, and then work back up to the top following all indicated side branches. A typical order of evaluating nodes is illustrated in Figure 6.4 (the upper tree). The number inside each node represents the time at which it is selected.

The Jumptracking (unrestricted) Strategy: As the name implies, each time the algorithm selects a node, it can choose any active node anywhere in the tree. For example, it might always choose the active node corresponding to the highest LP solution, z^*. A possible order of solving subproblems under Jumptracking is illustrated in Figure 6.4 (the lower tree).

At first glance, the Backtracking procedure appears to be unnecessarily restrictive. The major advantages are conservation of storage required and a reduction in the amount of computation required to solve the corresponding LP at each node. Observe that the number of active subproblems in the list at any time is equal to the number of levels in the current branch of the tree. Using Jumptracking, the size of the active list can grow exponentially. Each node in the active list corresponds to a linear programming problem with its own set of constraints. Consequently, storage space for subproblems is an important consideration.

Computation time is an even more serious issue with Jumptracking. Observe that each time we solve a subproblem, we solve an LP complete with a full Simplex tableau. When we move down the tree, we add one new constraint to the LP. This can be done relatively efficiently if the old tableau is still available.

To do this using the Jumptracking strategy, we would have to save the Simplex tableau for each node (or at least enough information to generate the tableau easily). Hence, Backtracking can save a large amount of LP computation time at each node. The efficiency of solving subproblems is crucial to the success of a branch-and-bound method because practical problems will typically generate trees with literally thousands of nodes.

The major advantage of Jumptracking is that, by judicious selection of the next active node, we can usually solve the problem by examining far fewer nodes. Observe that when we find the optimal integer solution, many of the nodes can be eliminated by the bounding test. With Jumptracking, the integer solution is represented by a node at the bottom of the branching tree. With Backtracking, each time we choose a branch, one is "correct" and the other is "wrong". If we choose the wrong branch, we must evaluate all nodes in

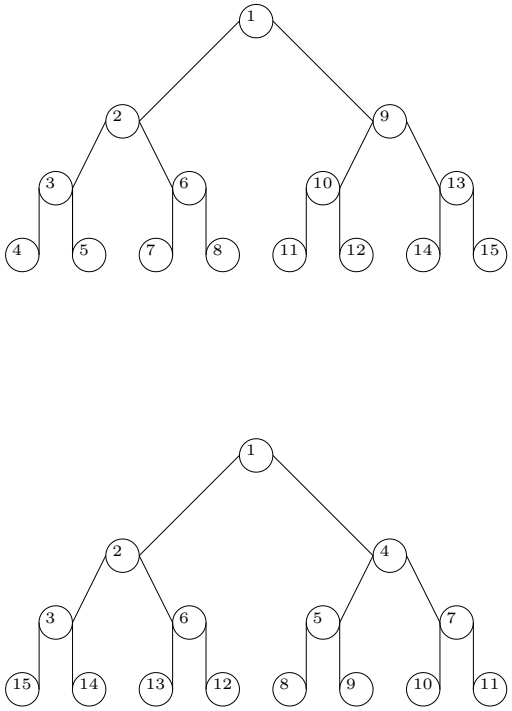


Figure 6.4: Branching strategies: Backtracking (the upper tree) and Jumptracking (the lower tree).

that branch before we can get back on the correct branch. Using Jumptracking, we can return to the correct branch as soon as we realize that we may have made a mistake. When we find the optimal solution, many of the nodes in the "wrong" branch will be fathomed at a higher level of the tree by the bounding test.

In short, there is a trade-off between Backtracking and Jumptracking, and many commercial algorithms use a mixed strategy. Backtracking is used until there is a strong indication of being in the wrong branch; then there is a jump to a more promising node in the tree and a resumption of a Backtracking strategy from that point. The amount of Jumptracking is determined by the definition of "wrong".

6.4 MAX-SAT Example

Often one is interested in variables and functions that take only two possible values, say TRUE/FALSE or 1/0. These are called *Boolean variables/functions*, and are used to model situations when it is only desirable to know whether something occurs or not, e.g.

whether a switch is on or off, or whether there is current running through a wire or not (but we do not care how much current). We will asume that the value of a Boolean variable or function is always either 0 or 1. We denote *negation* by $\neg$, that is, $\neg 0 = 1$ and $\neg 1 = 0$. If x, y are Boolean variables then the *conjunction* $x \wedge y$ has value 1 precisely when both x and y have value 1 (logical AND), and the *disjunction* $x \vee y$ has value 1 precisely when at least one of x and y has value 1 (logical OR). An example of a Boolean function of three Boolean variables would be $F(x_1, x_2, x_3) = (\neg x_1 \vee x_2) \wedge \neg x_3$. A *clause* is a special Boolean function consisting only of disjunctions (and no conjunctions) between single variables or their negations. For example $\neg x_1 \vee x_2 \vee \neg x_3$ is a clause containing three variables. If values are assigned to the variables of a clause in a way which makes its value equal to 1, then we say that this assignment *satisfies* the clause.

An instance of the Maximum Satisfiability Problem (MAX-SAT) is a list of clauses $F_1, F_2, \ldots, F_m$ containing Boolean variables $x_1, x_2, \ldots, x_n$. The goal is to find an assignment of values to the variables such that the number of satisfied clauses is as large as possible.

Consider the following example (from J. Hromkovič) with 10 clauses and 4 variables:

$$\begin{aligned}
F_1 &= x_1 \vee \neg x_2 \\
F_2 &= x_1 \vee x_3 \vee \neg x_4 \\
F_3 &= \neg x_1 \vee x_2 \\
F_4 &= x_1 \vee \neg x_3 \vee x_4 \\
F_5 &= x_2 \vee x_3 \vee \neg x_4 \\
F_6 &= x_1 \vee \neg x_3 \vee \neg x_4 \\
F_7 &= x_3 \\
F_8 &= x_1 \vee x_4 \\
F_9 &= \neg x_1 \vee \neg x_3 \\
F_{10} &= x_1.
\end{aligned}$$

First we will solve the problem with Backtracking. At each node the following rule is used: Assign a value to the first variable among x_1, x_2, x_3, x_4 which currently has no value assigned to it, and let this value be 1 the first time when visiting the node, 0 the second time.

The search tree is shown in Figure 6.5. At each interior node of the tree we note the clauses which become violated at the moment the node is reached (they can no longer be satisfied no matter which values are assigned to the variables having no values assigned to them yet). Below any bottom node in the tree we note the value of the objective function which follows from the current assignments. Thus at the bottom node corresponding to the assignments $x_1 = x_2 = x_3 = x_4 = 1$, all clauses except F_9 are satisfied, hence the

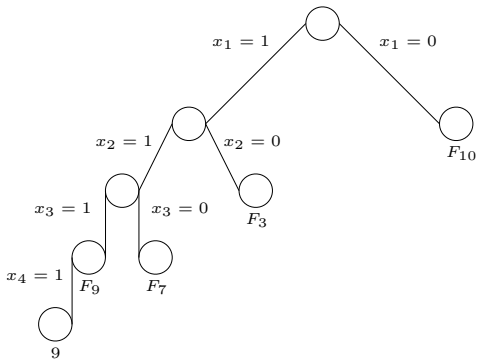


Figure 6.5: Backtracking in MAX-SAT to the left.

number of satisfied clauses is 9. With z^I now equal to 9, we will never separate on any node for which some clause gets violated, since the optimal solution at such a node cannot be better than the current incumbent.

In Figure 6.6 is shown the tree which is searched by Backtracking if, when separating each node, we first assign the value 0 instead of 1 to the next variable. We observe that the number of nodes visited by the algorithm, and hence its efficiency, depends very much on the order in which the nodes are searched.

6.5 Questions

Question 6.5.1. *Solve the problem stated in Section 6.1 by first separating on x_1 instead of x_2.*

Question 6.5.2. *Solve the following problem by the Branch-and-Bound method:*

$$\begin{array}{lcl} \textit{maximize} & z = x_1 + 5x_2 & \\ \textit{subject to} & x_1 + 10x_2 & \leq 20 \\ & x_1 & \leq 2 \\ & x_1, x_2 \geq 0 \textit{ and integer} & \end{array}$$

Question 6.5.3. *Solve the following IP problem by the Branch-and-Bound method:*

$$\begin{array}{ll} \textit{maximize} & z = x_1 + x_2 \end{array}$$

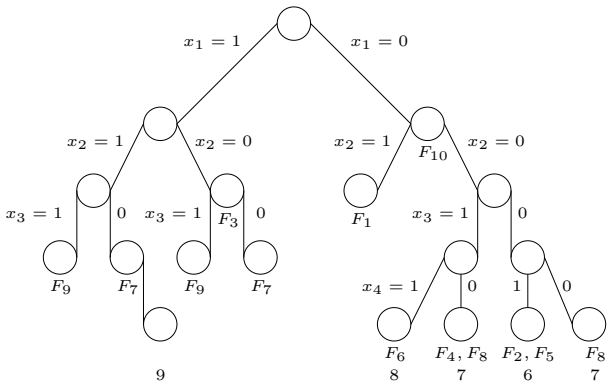


Figure 6.6: Backtracking in MAX-SAT to the right.

$$\begin{array}{lcl} \textit{subject to} & x_1 & \leq 2 \\ & x_1 + 2x_2 & \leq 5 \\ & x_1, x_2 \geq 0 \textit{ and integer} & \end{array}$$

Question 6.5.4. *Solve the following IP problem by the Branch-and-Bound method:*

$$\begin{array}{lcl} \textit{maximize} & z = x_1 + 2x_2 & \\ \textit{subject to} & x_1 + x_2 & \leq 2 \\ & x_2 & \leq 1.5 \\ & x_1, x_2 \geq 0 \textit{ and integer} & \end{array}$$

Question 6.5.5. *Solve the following MP problem by the Branch-and-Bound method:*

$$\begin{array}{lcl} \textit{minimize} & z = x_1 + x_2 & \\ \textit{subject to} & 2x_1 + 3x_2 & \geq 6 \\ & 3x_1 + x_2 & \leq 3 \\ & x_1, x_2 \geq 0 \textit{ and } x_1 \textit{ integer} & \end{array}$$

Question 6.5.6. *Describe the Branch-and-Bound algorithm for IP maximization problems. What does it mean that a node (i.e., subproblem) is fathomed.*

Question 6.5.7. *(a) Describe the Backtracking and Jumptracking strategies for branching.*

(b) What are he advantages and disadvantages of each of the two strategies?

Question 6.5.8. *Solve the MAX-SAT example of Section 6.5 by Jumptracking, using the strategy to always separate on an active node with a minimal number of currently violated clauses. Can you say something about the efficiency of Jumptracking compared to Backtracking applied to this problem?*

Question 6.5.9. *Solve the following instance of the knapsack problem:*

$$\begin{aligned} \max z = \quad & 5x_1 + 8x_2 + 6x_3 + 3x_4 \\ s.t. \quad & 2x_1 + 2x_2 + 4x_3 + 3x_4 \quad \leq 7 \\ & x_1, x_2, x_3, x_4 = 0 \text{ or } 1. \end{aligned}$$

Question 6.5.10. *Using the backtracking branch-and-bound algorithm, solve the following instance of MAX-SAT. In the algorithm, assign a variable the value 1/true before assigning it the value 0/false. Depict the search tree and justify your answer.*

$$\begin{array}{lll} F_1 = x_1 \vee \neg x_2, & F_2 = x_1 \vee x_3 \vee x_4, & F_3 = \neg x_1 \vee x_2, \\ F_4 = x_1 \vee \neg x_3 \vee \neg x_4, & F_5 = x_2 \vee \neg x_3 \vee x_4, & F_6 = x_1 \vee \neg x_3 \vee x_4, \\ F_7 = x_1 \vee x_3, & F_8 = x_1 \vee \neg x_4, & F_9 = \neg x_1 \vee \neg x_3, \\ F_{10} = x_1 \vee \neg x_3, & F_{11} = \neg x_1 \vee x_4. & \end{array}$$

6.6 Solutions

Question 6.5.2: $x_1 = 0, x_2 = 2, z^* = 10$.

Question 6.5.3: $x_1 = 1, x_2 = 2$ and $x_1 = 2, x_2 = 1$, $z^* = 3$.

Question 6.5.4: $x_1 = 1, x_2 = 1, z^* = 3$.

Question 6.5.5: $x_1 = 0, x_2 = 2, z^* = 2$.

Question 6.5.9

We call (P0) the linear relaxation of

$$\begin{aligned} \max z = \quad & 5x_1 + 8x_2 + 6x_3 + 3x_4 \\ \text{s.t.} \quad & 2x_1 + 2x_2 + 4x_3 + 3x_4 \quad \leq 7 \\ & x_1, x_2, x_3, x_4 = 0 \text{ or } 1. \end{aligned}$$

To solve (P0), find the ratios

$$r_1 = 5/2 = 2.5,\ r_2 = 8/2 = 4,\ r_3 = 6/4 = 1.5,\ r_4 = 3/3 = 1.$$

Thus, $r_2 > r_1 > r_3 > r_4$. Therefore, the solution of (P0) is $x_2 = x_1 = 1, x_3 = 3/4, x_4 = 0$. This is an infeasible solution of the original IP problem. Consider the subproblems (P1)=(P0)+$(x_3 = 1)$ and (P2)=(P0)+$(x_3 = 0)$.

The solution of (P1) is $x_3 = x_2 = 1$, $x_1 = 1/2, x_4 = 0$, $z = 16.5$. The solution of (P2) is $x_3 = 0, x_1 = x_2 = x_4 = 1$, $z = 16$. The last solution is feasible, but it is not better than that of (P1). Thus, we have to branch at (P1).

Consider two subproblems of (P1): (P3)=(P0)+$(x_3 = 1)$+$(x_1 = 1)$ and (P4)=(P0)+$(x_3 = 1)$+$(x_1 = 0)$. For (P3), $x_1 = x_3 = 1, x_2 = 1/2, x_4 = 0$ and $z = 15$. We can fathom (P3) as (P2) as z for (P2) is larger than z for (P3). For (P4), $x_1 = 0, x_2 = x_3 = 1, x_4 = 1/3$ and $z = 15$. We can fathom (P4) as (P2) as z for (P2) is larger than z for (P4). Thus, the solution for (P2) is optimal.

Question 6.5.10

At each node the following rule is used: Assign a value to the first variable among x_1, x_2, x_3, x_4 which currently has no value assigned to it, and let this value be 1 the first time when visiting the node, 0 the second time.

At each interior node of the search tree we note the clauses which become violated at the moment the node is reached (they can no longer be satisfied no matter which values are assigned to the variables having no values assigned to them yet). At the bottom node

corresponding to the assignments $x_1 = x_2 = x_3 = x_4 = 1$, all clauses except F_9 are satisfied, hence the number of satisfied clauses is 10.

If we change x_4 to 0, then all clauses except F_9, F_{11} are satisfied, i.e. only 9 clauses are satisfied. However, if we change x_3 to 0 and assign $x_4 = 1$, then all clauses are satisfied and thus we've found an optimal solution.

(The tree is not depicted.)

Chapter 7

Polynomially solvable Problems

In the previous chapters we have been considering heuristic methods for solving problems so difficult that their precise solutions cannot be found within reasonable time by any known methods. We will now discuss a few important problems that do admit polynomial time algorithms. These are often useful when dealing with NP-hard problems that can in some way be broken down into many such subproblems, each of which can be solved quickly. This idea is similar to the solution of problems using the Branch-and-Bound method.

7.1 The Minimum Spanning Tree Problem

The Minimum Spanning Tree Problem, MSTP, is not only a basic example of a polynomially solvable optimisation problem, but it is also a fundamental ingredient in both algorithmic and heuristic approaches to various combinatorial optimisation problems.

7.1.1 Graph Theory Basics

In a graph G, a *path* is an alternating sequence $v_1e_1v_2 \dots v_{p-1}e_{p-1}v_p$ of vertices v_i and edges e_j such that for every j, v_j and v_{j+1} are end-vertices of edge e_j, and the vertices $v_1, \dots, v_{p-1}$ are distinct. A path can be viewed as a subgraph of G. Since the edges of $v_1e_1v_2 \dots v_{p-1}e_{p-1}v_p$ are defined by their vertices, we can omit the edges and write down the path as $v_1v_2 \dots v_{p-1}v_p$. A graph G is *connected* if there is a path between every pair of distinct vertices of G. A path $v_1v_2 \dots v_{p-1}v_p$ is called a *cycle* if $v_1 = v_p$.

A *forest* is a graph with no cycle. A *tree* is a connected forest. Trees and forests play an important role in many applications of graph theory especially in computer science.

Theorem 7.1.1. *Let T be a tree with n vertices. Then*

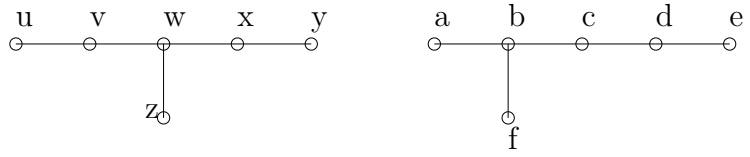


Figure 7.1: Trees

(a) T has $n-1$ edges

(b) addition of an edge between two non-adjacent vertices in T creates exactly one cycle

(c) there is exactly one path between any pair of vertices in T

(d) deletion of an edge from T creates a disconnected graph with two connectivity components

According to (d) a tree is minimally connected graph. Thus, if we want to connect a set of newly created camps by roads with minimum expenses, we should construct a tree system of the roads.

A vertex of degree 1 is called a *leaf*.

Theorem 7.1.2 (Leaf Theorem)**.** *Every tree with at least two vertices has at least two leaves.*

A tree of graph G is *spanning* if it contains all vertices of G.

7.1.2 MST

In many applications, weighted graphs are of interest. A graph $G = (V, E)$ is *weighted* if there is an assignment of edges to non-negative real numbers such that every edges has weight. The weights may reflect various parameters such as distances between vertices, time or cost of going between vertices.

The *weight of a graph* is the sum of the weights of its edges. The following minimum connector problem is of interest: Given a weighted connected graph G, find a spanning connected subgraph of G of minimum weight. This problem arises in applications. For example, suppose we created a number of new villages in 'the middle of nowhere' and want to connect the villages with roads such that the total distance of the roads is minimum.

According to the properties of trees, the minimum weight connector is a spanning tree of minimum cost. To find this tree T the following *Kruskal's algorithm* can be used. Order the edges of G $e_1, e_2, \ldots, e_m$ such that $w(e_1) \leq w(e_2) \leq w(e_3) \leq \cdots \leq w(e_m)$. Pick edges in that order one by one and add them to (initally empty) T except when the current edge creates a cycles with previously chosen edges.

The usefulness of this algorithm is due to the following result.

Theorem 7.1.3. *Kruskal's algorithm always finds a minimum weight spanning tree.*

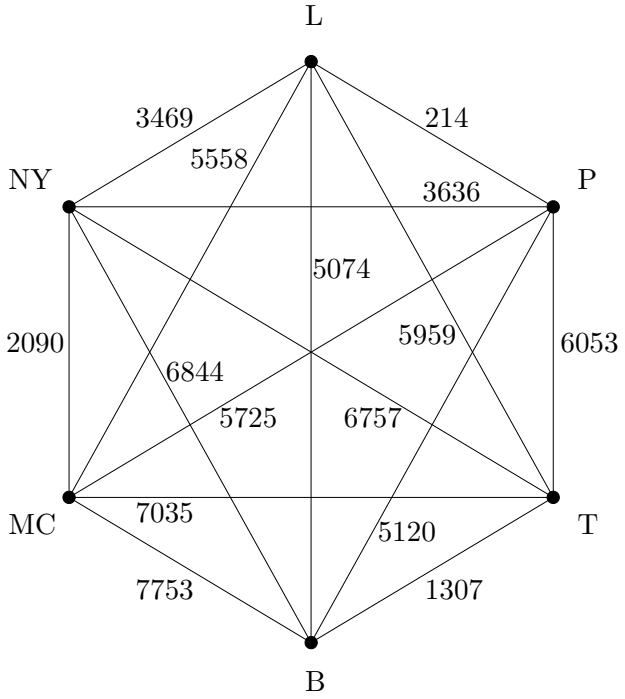


Figure 7.2: An edge-weighted complete graph

Consider an example (from J.A. Bondy and U.S.R. Murty) of an airline that wants to offer connections between the cities Beijing, London, Mexico City, New York, Paris and Tokyo. The distances in miles between each pair of cities is given in the following table:

	B	L	MC	NY	P	T
B	–	5074	7753	6844	5120	1307
L	5074	–	5558	3469	214	5959
MC	7753	5558	–	2090	5725	7035
NY	6844	3469	2090	–	3636	6757
P	5120	214	5725	3636	–	6053
T	1307	5959	7035	6757	6053	–

The goal is to schedule flight connections that are sufficient to allow traveling between any pair of the cities, at the lowest possible cost, as measured by the total distance covered by the planes in operation. The problem is described by an edge-weighted complete graph with vertices B,L,MC,NY,P and T, see Figure 7.2.

We will apply Kruskal's algorithm to this problem. When applying Kruskal's algorithm, we will initially choose the edge L–P, of weight 214. Afterwards, we choose first B–T (weight 1307), then MC–NY (weight 2090), followed by L-NY (weight 3469), see Figure 7.3.

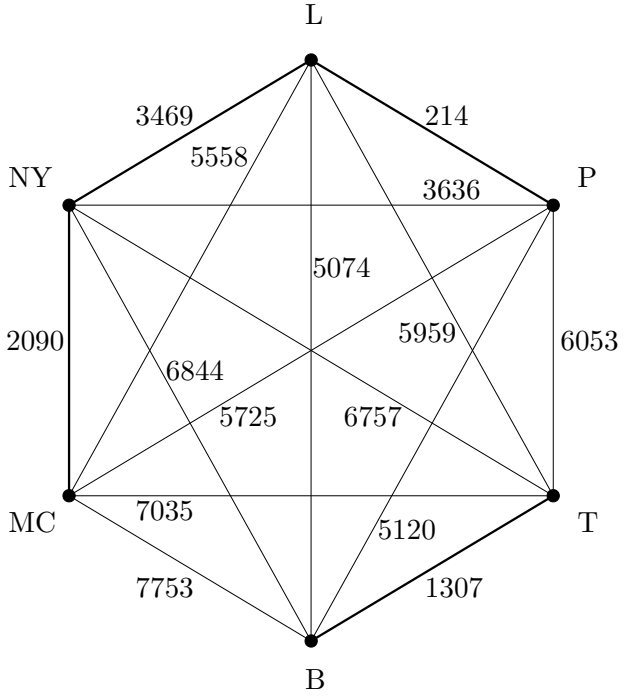


Figure 7.3: The first four edges chosen by Kruskal's Algorithm

After having chosen four edges, the cheapest remaining edge will be NY-P, of weight 3636. This edge, however, does not qualify for being chosen, since there is already a path of chosen edges from NY to P (via L) and so this edge will create a cycle. Hence we choose edge B-L instead (weight 5074), after which there are no more edges that can be chosen, and the algorithm stops (Figure 7.4).

Consider a bit more interesting question.

Question 7.1.4. *Find a minimum weight spanning tree in graph G in Figure 7.5. How many minimum weight spanning trees G has?*

Solution: We use Kruskal's algorithm. We order edges of G in the following order: $cd, bc, ef, bf, be, ad, ab$. We start from empty T. We pick edges cd, bc, ef and bf without creating any cycle and thus add them to T. Edge be cannot be added to T as it creates cycle $befb$ with edges ef and bf chosen earlier. We add edge ad to T, but we do not add ab to T as it creates cycle $abcda$ with previously chosen edges. As a result we get T in Figure 7.5.

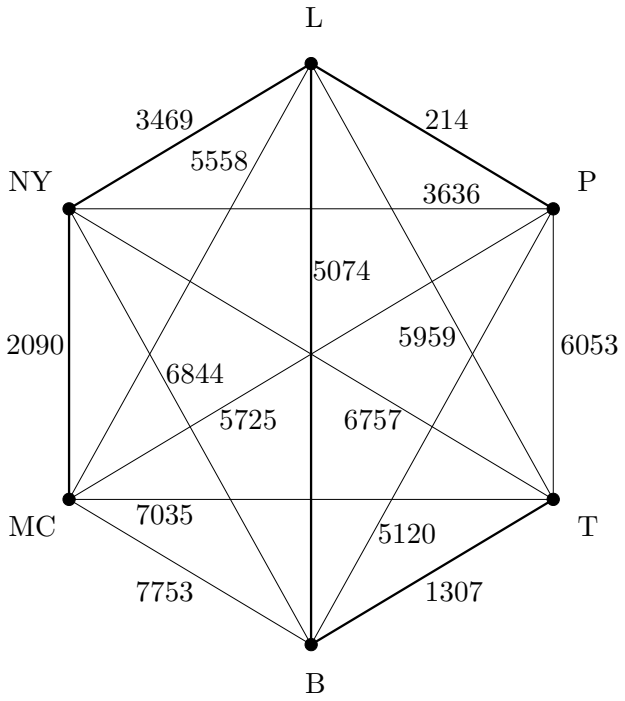


Figure 7.4: The optimal solution

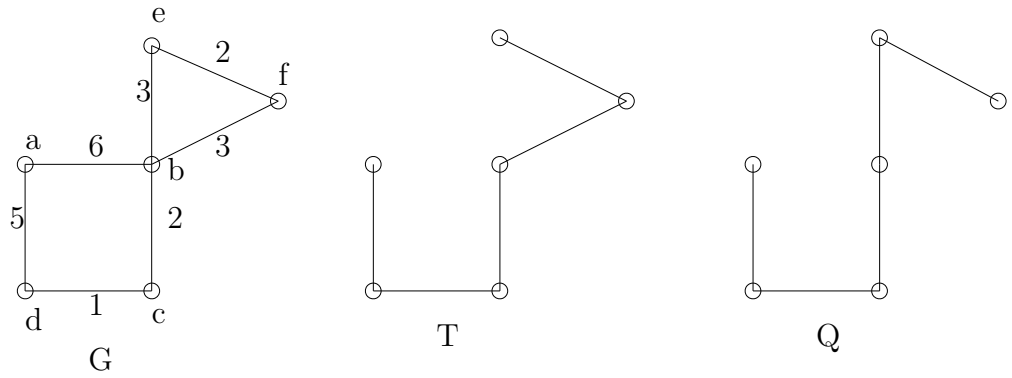


Figure 7.5: A weighted graph and its minimum weight spanning trees

We canorder edges of G slightly differently: $cd, bc, ef, be, bf, ad, ab$. Then Kruskal's algorithm constructs Q in Figure 7.5 (edge be gets chosen before bf and bf cannot be picked up as it creates cycle with previously chosen edges). Thus, we get another minimum weight spanning tree. Since bc and ef have the same weight we can have several orders of the edges, but the order of the last two edges does not matter since they both will be chosen no matter what order is considered. Thus, G has exactly two minimum weight

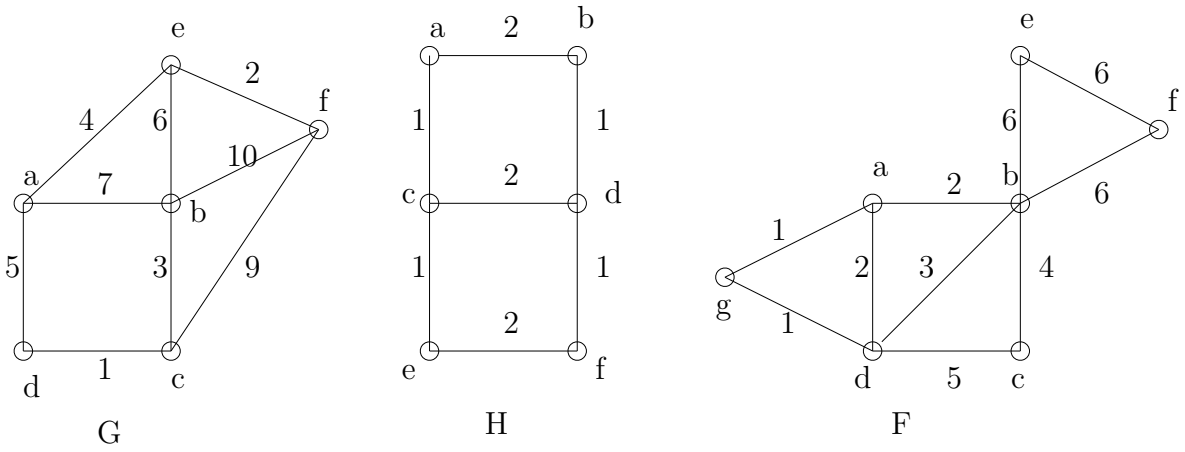


Figure 7.6: Weighted graphs

spanning trees.

Question 7.1.5. *Find a minimum weight spanning tree and the number of minimum weight spanning trees in the graphs of Figure 7.6.*

7.2 Assignment Problem

We return to the assignment problem (AP) discussed already in Chapters 1 and 5, and we will see how to solve the problem and find an optimal solution with a polynomial time algorithm.

7.2.1 Bipartite Matching Problem

Consider the following variation of AP. There are n persons $p_1, \ldots, p_n$ available to perform n jobs $j_1, \ldots, j_n$ as before, but some of the persons are only qualified to perform certain jobs, and we ask whether there exists any assignment of persons to jobs such that each job is performed by a qualified person.

To model this question within the 0/1-LP formulation of AP,

$$\begin{array}{rll} \min z = & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} & \\ \text{s.t.} & \sum_{i=1}^{n} x_{ij} & = 1, \ j = 1, 2, \ldots, n \end{array}$$

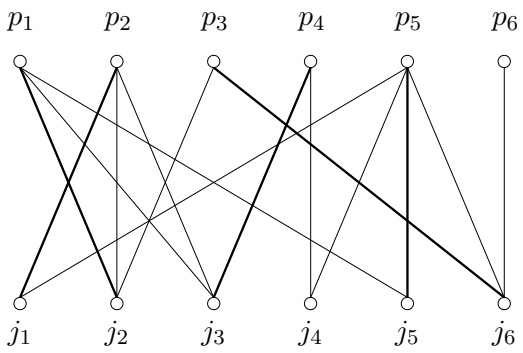


Figure 7.7: A bipartite graph with a partial matching

$$\begin{array}{ll} \sum_{j=1}^{n} x_{ij} & = 1,\ i = 1, 2, \ldots, n \\ x_{ij} & = 0 \text{ or } 1, \quad i, j = 1, \ldots, n. \end{array}$$

we may introduce some suitable costs,

$$c_{ij} = \begin{cases} 0 & \text{if person } i \text{ qualifies for job } j \\ N & \text{otherwise.} \end{cases}$$

where N is a positive number. The reformulated problem asks whether the objective function for the optimal solution has value 0.

If we consider $p_1, \ldots, p_n$ and $j_1, \ldots, j_n$ as nodes of a graph in which $p_i j_k$ is an edge precisely when person i is qualified to perform job k, then the problem is represented by a *bipartite graph*, as in the example of Figure 7.7 (C.H. Papadimitriou and K. Steiglitz). There we have 6 persons and 6 jobs, and we assume that we have guessed a partial assignment of persons to 5 of the jobs, as indicated by highlighted edges. These edges form a *matching* in the graph, $M = \{p_1 j_2, p_2 j_1, p_3 j_6, p_4 j_3, p_5 j_5\}$, a set of edges no two of which meet at a common node. A node that appears as an end-node of an edge of the matching is called *saturated*. We seek a *perfect matching*, that is, a matching such that every node of the graph is saturated. We note, since each of j_4 and p_6 is unsaturated, that M is not perfect. However, we will use M to construct a matching which is perfect, thereby solving our assignment problem.

The idea of the following method is to search for a reassignment of some of the persons to other jobs, in such a way that at least one more job gets a person assigned to it than in the currently available assignment. In other words, we search for a matching which saturates all nodes that are already saturated by M, and contains more edges than M.

An M-*augmenting* path is a path whose first and last nodes are both unsaturated, and whose edges are alternately contained in M, that is, the first edge of the path is not contained in M (since the first node is not saturated), its second edge is in M, its third edge is not in M, and so on, until reaching its last edge, which does not belong to M. A special case of an M-augmenting path would be a path consisting of a single edge between two unsaturated vertices. In the example of Figure 7.7 we have an M-augmenting path $P = j_4p_4j_3p_1j_2p_3j_6p_6$. We can now switch the roles of the edges of P by removing the edges of P from M and forming a new matching from the rest of M together with the remaining edges of P. Thus we keep p_2j_1 and p_5j_5 as matching edges, but remove p_4j_3, p_1j_2 and p_3j_6, and we add instead p_4j_4, p_1j_3, p_3j_2 and p_6j_6 to get $M' = \{p_2j_1, p_5j_5, p_4j_4, p_1j_3, p_3j_2, p_6j_6\}$, which is a perfect matching.

In general, if we have a matching in a bipartite graph and an M-augmenting path P, then we can use P in the same way to construct a matching which saturates more nodes than M (why does this construction always result in another matching?). Thus if we know how to find M-augmenting paths, then we may gradually improve an initial matching and obtain new matchings with an ever growing set of saturated vertices. The following algorithm finds an M-augmenting path if one exists:

Consider a bipartite graph G with bipartite sets $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$, and let M be a matching in G which does not saturate every node of A.

Step 0: Let a_i be an unsaturated node. Mark a_i as *active*, give a_i a *label* of 0, and leave all other nodes unmarked and without label.

Step 1: If there are no active nodes, then stop. Then there is no M-augmenting path with a_i as its first node.

Step 2: If there is an active node in A, say a_j, then consider all unmarked neighbours of a_j, mark each of them active and label it j, and finally mark a_j as *passive*.

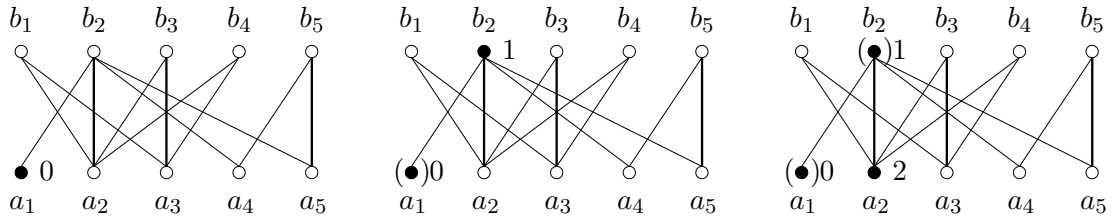
Step 3: Let b_k be an active node in B.

3a) If b_k is unsaturated, then stop. There exists an M-augmenting path with a_i as its first node and b_k as its last. To find this path, trace the sequence of nodes starting from b_k and back to a_i by using the successive labels.

3b) If b_k is saturated, then let a_j be the node which is matched to b_k by M. Mark a_j active and label it k. Then mark b_k passive, and go to Step 1.

The above procedure may require looking at each edge of the graph once before it stops, hence its running time is $O(m)$, if m is the number of edges in G.

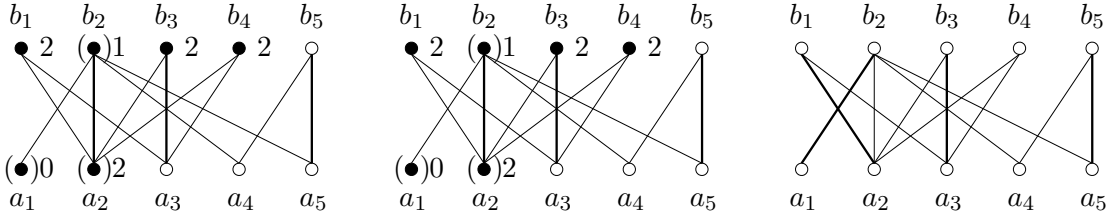
Figure 7.8 shows an example of applying the algorithm to search for an M-augmenting path. Note that when the unsaturated node b_1 is found in Step 3a, its label of 2 is used to determine that a_2 is going to be the neighbour of b_1 on the M-augmenting path P.



Step 0: a_1 is active (label 0).

Step 2: b_2 becomes active with label 1, and a_1 passive.

Step 3b: a_2 becomes active with label 2, and b_2 passive.



Step 2: b_1, b_3, b_4 become active with label 2, and a_2 passive.

Step 3a: b_1 is unsaturated: $P = b_1a_2b_2a_1$ is M-augmenting.

The new and larger matching.

Figure 7.8: An example of a search for an M-augmenting path.

Similarly, the label of 2 for a_2 means that b_2 is the other neighbour for a_2 on P, and so on.

Let us suppose that the above algorithm when applied to some instance G with bipartite sets A, B and matching M does not find an M-augmenting path, but that it terminates in Step 1. We will consider the set $S \subseteq A$ consisting of those nodes from A that become marked during execution of the algorithm, and we will let $N(S) \subseteq B$ denote the *neighbour set of* S, the set of those nodes from B which each has at least one neighbour in G that belongs to S. Because of the way in which nodes get marked in Steps 2 and 3b, and because every marked node from B is saturated (or we would have found an M-augmenting path in Step 3a), each node of $N(S)$ is matched to one of the nodes of S, and each node of S except a_i is matched to one of the nodes of $N(S)$, so the number of nodes in the two sets differ by one, that is, we have $|S| = |N(S)| + 1$. So we see that if we fail to find an M-augmenting path, then we find $S \subseteq A$ such that $|S| > |N(S)|$. We say that there exists a *bottleneck*. Indeed, no matching which saturates all nodes of A will fit into a graph with such a narrow bottleneck S: the nodes of any set $S \subseteq A$ will always have to be matched to equally many different nodes of B, each of which belongs to the neighbour set $N(S)$. Clearly this is only possible if $|S| \leq |N(S)|$ holds.

We can find a bottleneck, if it exists, using the algorithm above. Consider the right-bottom subfigure of Figure 7.8. The depicted matching leaves a_4 unsaturated and we can try to find a larger matching starting from a_4. However, our search will label only vertices a_4, b_2, b_5, a_1, a_5 without producing an M-augmenting path. This means the graph of Figure 7.8 has no perfect matching and so has a bottleneck. Since a_4, b_2, b_5, a_1, a_5 are only labeled vertices, let $S = \{a_1, a_4, a_5\}$ (labeled a_i vertices) and observe that $N(S) = \{b_2, b_5\}$ and $|S| > |N(S)|$.

Theorem 7.2.1 (Hall's Theorem)**.** *A bipartite graph with bipartite sets A, B contains a matching that saturates every node of A if and only if*

$$|S| \leq |N(S)| \quad \textit{for every } S \subseteq A.$$

For the special case of equally many nodes on each side of the bipartition, which is relevant to the Assignment Problem, we have:

Theorem 7.2.2. *A bipartite graph with bipartite sets A, B and $|A| = |B|$ contains a perfect matching if and only if*

$$|S| \leq |N(S)| \quad \textit{for every } S \subseteq A.$$

In summary, the following algorithm, called the *Hungarian Method*, applies to the problem of finding a perfect matching in a bipartite graph G with equally large bipartite sets A, B:

Step 0: Let M be any matching (possibly empty).

Step 1: If every node of A is saturated, then stop. The current matching M is perfect.

Step 2: Otherwise, let $a \in A$ be an unsaturated node. Apply the M-augmenting path algorithm to search for an M-augmenting path P that starts in a.

Step 3: If no M-augmenting path is found, then stop. No perfect matching exists (there is a bottleneck).

Step 4: If an M-augmenting path P is found, remove those edges from M that occur in P, and add those edges of P that are not in M. Replace M by this new larger matching, and go to Step 1.

Since we have to search for an augmenting path each time a new node from A gets saturated, and since this search takes time $O(m)$, where m is the number of edges in G, we see that the running time for the Hungarian Method is $O(|A|m)$, which is the same as $O(nm)$, where n is the number of nodes in G.

7.2.2 Minimum Leaf Out-Branchings in Acyclic Digraphs (non-examined)

We say that a subdigraph T of a digraph D is an *out-tree* if T is an oriented tree with only one vertex s of in-degree zero (its *root*). The vertices of T of out-degree zero are *leaves*. If T is a spanning out-tree, i.e. $V(T) = V(D)$, then T is an *out-branching* of D.

In Figure 7.9, $D - zy$ is a minimum leaf out-branching and $D - xy$ is a maximum leaf out-branching. (These out-branchings have minimum and maximum number of leaves.) The leaves of $D - zy$ are y and t, and the leaves of $D - xy$ are x, y, t. The root of both out-branchings is r.

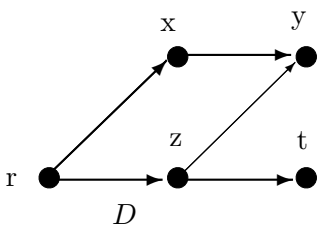


Figure 7.9: Out-Branching Example

Not every digraph has an out-branching:

Proposition 7.2.3. *A digraph D has an out-branching iff D has only one initial strong component (a strong component C is* initial *if there are no arcs entering C).*

Corollary 7.2.4. *An acyclic digraph (i.e., a digraph with no directed cycles) has an out-branching iff only one vertex of in-degree 0.*

In an application to databases, one is interested in finding a minimum leaf out-branching in an acyclic digraph. The following algorithm allows to find a minimum leaf out-branching in an acyclic digraph efficiently (in time $O(n^{2.5})$, where n is the number of vertices in D).

MINLEAF(D)
Input: An acyclic digraph D with vertex set V.
Output: A minimum leaf out-branching T of D if one exists / "NO" otherwise.

Step 1 Find a vertex r of in-degree 0. If there is another vertex of in-degree 0, return "no out-branching".

Step 2 Construct a bipartite graph $B = B(D)$ of D with partite sets $V, V' - r'$ and edge xy' for each arc $xy \in A(D)$.

Step 3 Find a maximum matching M in B.

Step 4 $M^* := M$. For all $y' \in V'$ not covered by M, set $M^* := M^* \cup \{$an arbitrary edge incident with $y'\}$.

Step 5 $A(T) := \emptyset$. For all $xy' \in M^*$, set $A(T) := A(T) \cup \{xy\}$.

Step 6 Return T.

Figure 7.10 illustrates MINLEAF. There $M = \{rx', xy', zt'\}$ and $T = D - zy$.

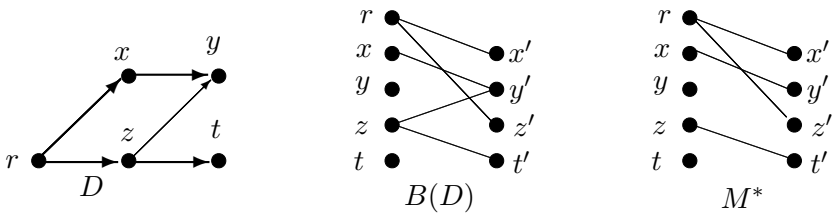


Figure 7.10: Illustration for MINLEAF

7.2.3 Weighted Assignment Problem

After having discussed the Bipartite Matching Problem in some detail, we turn again to the general Assignment Problem in which the objective is to find a smallest cost assignment, when given any cost matrix $(c_{ij})_{i,j=1,\ldots,n}$.

We will describe the *Hungarian Method for the Assignment Problem*, sometimes also called the *Kuhn-Munkres Algorithm* after its inventors.

The steps of the algorithm are illustrated by an instance of AP with the costs of assigning persons $p_1, \ldots, p_6$ to each of the jobs $j_1, \ldots, j_6$ given by the matrix

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{array} \begin{array}{c} \begin{array}{cccccc} j_1 & j_2 & j_3 & j_4 & j_5 & j_6 \end{array} \\ \left[\begin{array}{cccccc} 7 & 12 & 10 & 9 & 8 & 7 \\ 7 & 7 & 10 & 5 & 11 & 7 \\ 5 & 8 & 5 & 4 & 6 & 7 \\ 9 & 6 & 8 & 10 & 4 & 5 \\ 8 & 8 & 10 & 8 & 8 & 9 \\ 5 & 6 & 9 & 6 & 4 & 6 \end{array}\right] \end{array}$$

Step 0: Set M equal to the cost matrix of the AP instance.

Step 1: For each row of the matrix M, find the smallest number that appears in it, and subtract this number from each entry in the row. Replace M by this new matrix.

$$\left[\begin{array}{cccccc} 7 & 12 & 10 & 9 & 8 & 7 \\ 7 & 7 & 10 & 5 & 11 & 7 \\ 5 & 8 & 5 & 4 & 6 & 7 \\ 9 & 6 & 8 & 10 & 4 & 5 \\ 8 & 8 & 10 & 8 & 8 & 9 \\ 5 & 6 & 9 & 6 & 4 & 6 \end{array}\right] \begin{array}{c} -7 \\ -5 \\ -4 \\ -4 \\ -8 \\ -4 \end{array} \longrightarrow \left[\begin{array}{cccccc} 0 & 5 & 3 & 2 & 1 & 0 \\ 2 & 2 & 5 & 0 & 6 & 2 \\ 1 & 4 & 1 & 0 & 2 & 3 \\ 5 & 2 & 4 & 6 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 1 \\ 1 & 2 & 5 & 2 & 0 & 2 \end{array}\right]$$

Step 2: For each column of the matrix M, find the smallest number that appears in it, and subtract this number from each entry in the column. Replace M by this new matrix.

$$\begin{array}{c} \left[\begin{array}{cccccc} 0 & 5 & 3 & 2 & 1 & 0 \\ 2 & 2 & 5 & 0 & 6 & 2 \\ 1 & 4 & 1 & 0 & 2 & 3 \\ 5 & 2 & 4 & 6 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 1 \\ 1 & 2 & 5 & 2 & 0 & 2 \end{array}\right] \\ \begin{array}{cccccc} -0 & -0 & -1 & -0 & -0 & -0 \end{array} \end{array} \longrightarrow \left[\begin{array}{cccccc} 0 & 5 & 2 & 2 & 1 & 0 \\ 2 & 2 & 4 & 0 & 6 & 2 \\ 1 & 4 & 0 & 0 & 2 & 3 \\ 5 & 2 & 3 & 6 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 2 & 4 & 2 & 0 & 2 \end{array}\right]$$

Step 3: Construct a bipartite graph G with a node for each row and a node for each column of M, and with an edge joining the node of a row to the node of a column if M contains a zero in its corresponding entry. Then use the Hungarian Method to find either a perfect matching in G, or a bottleneck set S of rows in M (a set S such that fewer than $|S|$ columns of M have a zero entry in some row of S).

Step 4: If a perfect matching is found in Step 3, then stop with this matching as optimal solution to the AP.

Step 5: Otherwise, mark the bottleneck rows found in Step 3. Then mark those columns that have a zero in at least one of the marked rows.

$$\begin{array}{cc} & \begin{array}{cccccc} & & & & * & \end{array} \\ & \begin{bmatrix} 0 & 5 & 2 & 2 & 1 & 0 \\ 2 & 2 & 4 & 0 & 6 & 2 \\ 1 & 4 & 0 & 0 & 2 & 3 \\ 5 & 2 & 3 & 6 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 2 & 4 & 2 & 0 & 2 \end{bmatrix} \\ \begin{array}{c} \\ \\ \\ * \\ \\ * \end{array} & \end{array}$$

Step 6: Find the smallest number which appears as an entry in a marked row and an unmarked column. Subtract this number from all entries in every marked row and add it to all entries in every marked column. Replace M by the new matrix, and go to Step 3.

$$\begin{bmatrix} 0 & 5 & 2 & 2 & 1 & 0 \\ 2 & 2 & 4 & 0 & 6 & 2 \\ 1 & 4 & 0 & 0 & 2 & 3 \\ 5 & 2 & 3 & 6 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 2 & 4 & 2 & 0 & 2 \end{bmatrix} \begin{array}{c} \\ \\ \\ -1 \\ \\ -1 \end{array} \longrightarrow \begin{bmatrix} 0 & 5 & 2 & 2 & 2 & 0 \\ 2 & 2 & 4 & 0 & 7 & 2 \\ 1 & 4 & 0 & 0 & 3 & 3 \\ 4 & 1 & 2 & 5 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 3 & 1 & 0 & 1 \end{bmatrix}$$
$$\begin{array}{cccccc} & & & & +1 & \end{array}$$

When we repeat Step 3 for the matrix M obtained in Step 6 for our example, we find a perfect matching and stop with an optimal assignment:

$$\begin{bmatrix} \boxed{0} & 5 & 2 & 2 & 2 & 0 \\ 2 & 2 & 4 & \boxed{0} & 7 & 2 \\ 1 & 4 & \boxed{0} & 0 & 3 & 3 \\ 4 & 1 & 2 & 5 & 0 & \boxed{0} \\ 0 & \boxed{0} & 1 & 0 & 1 & 1 \\ 0 & 1 & 3 & 1 & \boxed{0} & 1 \end{bmatrix}$$

By comparing with the original cost matrix, we see that the cheapest possible assignment of persons to jobs has cost $7 + 5 + 5 + 5 + 8 + 4 = 34$.

By analysing the running time of the Hungarian Alorithm for the Assignment Problem, one gets that it runs in $O(n^3)$ steps, where n is the number of columns and rows in the input matrix. However, in most practical problems, it will run in substantially shorter time than given by this upper bound.

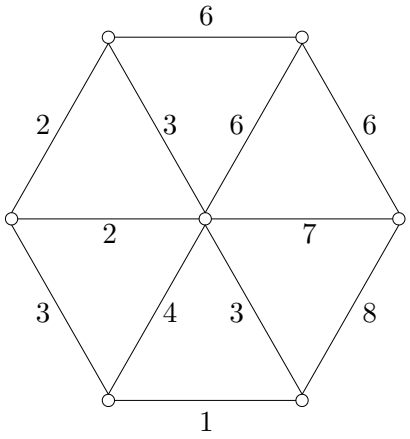


Figure 7.11: An edge-weighted graph

7.3 Questions

Question 7.3.1. *Find a spanning tree of minimal weight for the edge-weighted graph in Figure 7.11.*

Question 7.3.2. *Find a spanning tree of maximal weight for the edge-weighted graph in Figure 7.11.*

Question 7.3.3. *Use the augmenting path algorithm to find a perfect matching for the example in Figure 7.8 starting from the last matching shown in the figure.*

Question 7.3.4. *Find a maximum matching in a bipartite graph with bipartite sets* $\{a_1, a_2, \ldots, a_5\}$ *and* $\{b_1, b_2, \ldots, b_5\}$*, and edge set* $\{a_1b_1, a_1b_4, a_2b_2, a_2b_3, a_3b_3, a_4b_4, a_4b_5, a_5b_1, a_5b_2\}$.

Question 7.3.5. *Solve the Assignment Problem given by the cost matrix below.*

$$\begin{bmatrix} 10 & 3 & 8 & 9 & 8 \\ 7 & 4 & 7 & 6 & 7 \\ 5 & 11 & 2 & 4 & 3 \\ 8 & 7 & 7 & 8 & 9 \\ 9 & 8 & 3 & 7 & 7 \end{bmatrix}$$

Question 7.3.6. *Solve the Assignment Problems given by the following cost matrices:*

$$\begin{bmatrix} 9 & 4 & 8 & 9 & 8 \\ 5 & 2 & 7 & 6 & 7 \\ 3 & 1 & 2 & 4 & 3 \\ 7 & 5 & 7 & 8 & 9 \\ 8 & 9 & 3 & 7 & 7 \end{bmatrix} \qquad \begin{bmatrix} 5 & 0 & 2 & 3 & 4 & 5 \\ 5 & 5 & 2 & 7 & 1 & 5 \\ 7 & 4 & 7 & 8 & 6 & 5 \\ 3 & 6 & 4 & 2 & 8 & 7 \\ 4 & 4 & 2 & 4 & 4 & 3 \\ 7 & 6 & 3 & 6 & 8 & 6 \end{bmatrix}$$

Question 7.3.7. *Explain how to modify the data such that the Hungarian Method scan be used to find an assignment of largest possible weight instead of smallest.*

Then use this approach to find a maximal weight assignment for the example treated in Section 7.2.2.

Question 7.3.8. *(a) Formulate Hall's theorem on matchings in bipartite graphs. Using Hall's theorem, prove that graph R of Figure 7.12 has no perfect matching. (b) Find a maximum matching in graph Q of Figure 7.12. Justify your answer.*

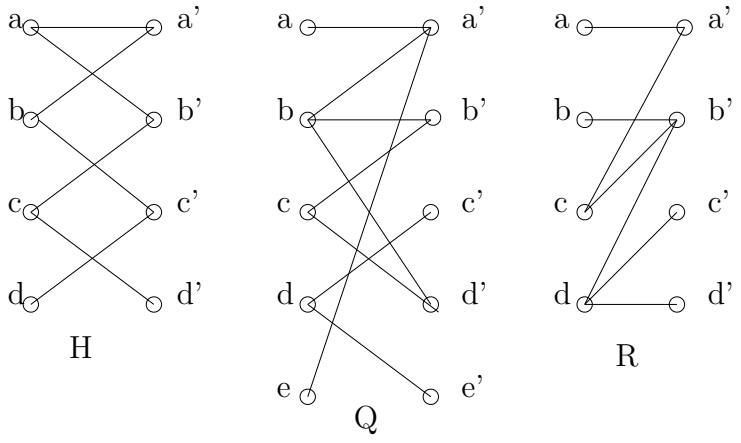


Figure 7.12: Bipartite graphs

7.4 Solutions

Question 7.3.4

The following edge set is a maximum matching in the graph: $\{a_1b_4, a_2b_2, a_3b_3, a_4b_5, a_5b_1\}$.

Question 7.3.5

By subtracting min entries from each row and then each column, we get

$$\begin{bmatrix} 6 & 0 & 5 & 5 & 4 \\ 2 & 0 & 3 & 1 & 2 \\ 2 & 9 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 5 & 5 & 0 & 3 & 3 \end{bmatrix}$$

A bottleneck: rows 1 and 2 and col. 2. [3 marks] Subtracting 1 from the rows and adding 1 to the column, we get

$$\begin{bmatrix} 5 & \mathbf{0} & 4 & 4 & 3 \\ 1 & 0 & 2 & \mathbf{0} & 1 \\ 2 & 10 & 0 & 1 & \mathbf{0} \\ \mathbf{0} & 1 & 0 & 0 & 1 \\ 5 & 6 & \mathbf{0} & 3 & 3 \end{bmatrix}$$

The bold zeros indicate the optimal solution.

Question 7.3.6

$$\begin{bmatrix} 2 & \mathbf{0} & 3 & 1 & 1 \\ \mathbf{0} & 0 & 4 & 0 & 2 \\ 0 & 1 & 1 & 0 & \mathbf{0} \\ 0 & 1 & 2 & \mathbf{0} & 2 \\ 3 & 7 & \mathbf{0} & 1 & 2 \end{bmatrix} \qquad \begin{bmatrix} 3 & \mathbf{0} & 2 & 2 & 3 & 3 \\ 3 & 5 & 2 & 6 & \mathbf{0} & 3 \\ 1 & 0 & 3 & 3 & 1 & \mathbf{0} \\ 0 & 5 & 3 & \mathbf{0} & 6 & 4 \\ \mathbf{0} & 2 & 0 & 1 & 1 & 0 \\ 2 & 3 & \mathbf{0} & 2 & 4 & 2 \end{bmatrix}$$

Question 7.3.7

Replace every entry c_{ij} of the matrix by $\max\{c_{pq} : \ 1 \leq p, q \leq 6\} - c_{ij} = 12 - c_{ij}$ and get

$$\begin{bmatrix} 5 & 0 & 2 & 3 & 4 & 5 \\ 5 & 5 & 2 & 7 & 1 & 5 \\ 7 & 4 & 7 & 8 & 6 & 5 \\ 3 & 6 & 4 & 2 & 8 & 7 \\ 4 & 4 & 2 & 4 & 4 & 3 \\ 7 & 6 & 3 & 6 & 8 & 6 \end{bmatrix}$$

After subtracting minimum of every row and then the minimum of every column, we get

$$\begin{bmatrix} 4 & 0 & 2 & 3 & 4 & 4 \\ 3 & 4 & 1 & 6 & 0 & 3 \\ 2 & 1 & 3 & 4 & 2 & 0 \\ 0 & 4 & 1 & 0 & 6 & 4 \\ 1 & 2 & 0 & 2 & 2 & 0 \\ 3 & 3 & 0 & 3 & 5 & 2 \end{bmatrix}$$

Note that zeros in columns 1 and 4 are all in row 4. Thus, we have a bottleneck. The minimum entry columns 1 and 4 but not row 4 is 1. Thus subtract 1 from columns 1 and 4 and add 1 to row 4 getting

$$\begin{bmatrix} 3 & \mathbf{0} & 2 & 2 & 4 & 4 \\ 2 & 4 & 1 & 5 & \mathbf{0} & 3 \\ 1 & 1 & 3 & 3 & 2 & \mathbf{0} \\ 0 & 5 & 2 & \mathbf{0} & 7 & 5 \\ \mathbf{0} & 2 & 0 & 1 & 2 & 0 \\ 2 & 3 & \mathbf{0} & 2 & 5 & 2 \end{bmatrix}$$

We must choose zeros such that every row and every column had exactly one zero. Now choose zero in every row which has just one zero: rows 1,2,3 and 6 (all choices here and later are in bold). Choose zero in column 4 as it has only one zero. It remains to choose zero in row 5 and column 1 as this is the only choice available.

Question 7.3.8

(a) Hall's Theorem: A bipartite graph G with partite sets X and Y has a matching saturating all vertices of X iff $|N(S)| \geq |S|$ for each $S \subseteq X$.

Since $|N(\{a, b, c\})| = |\{a', b'\}| = 2 < |\{a, b, c\}| = 3$, R has no perfect matching.

(b) We start from applying the greedy matching algorithm. It gives us a matching $M = \{aa', bb', cd', de'\}$. In order to find an M-augmenting path, we start from an unsaturated

vertex e. Using the usual an M-augmenting path algorithm, we obtain an M-alternating path $ea'a$, which is not M-augmenting. As there are only two non-saturating vertices and there is no an M-augmenting path starting from one of them, there is no M-augmenting path. Thus, M is a maximum matching.

Chapter 8

Construction Heuristics and Local Search

Unfortunately, the vast majority of optimisation problems are NP-hard, and Branch-and-Bound algorithms cannot solve them to optimality even for moderate instances, since sometimes they need to search every node of the Branch-and-Bound tree. If we have an IP problem with only $n = 20$ variables, each taking $m = 3$ possible values, then the Branch-and-Bound tree may have as many as $m^n = 3^{20}$ nodes at the lowest level of the tree. However, 3^{20} is already more than 3 billion. Of course, one may stop branching and accept the current best solution of Branch-and-Bound algorithm (partial Branch-and-Bound algorithms).

We will consider approaches that produce good solutions (but normally not optimal) faster than partial Branch-and-Bound algorithms.

8.1 Combinatorial optimisation problems

To illustrate possible approaches, we will consider some selected optimisation problems. One of them is the *asymmetric travelling salesman problem* (ATSP). This problem is one of the most famous and studied optimisation problems. In the ATSP, we are given a complete digraph D with vertices $V = \{1, \ldots, n\}$ and cost c_{ij} of every arc (i, j) and we wish to find a cheapest *tour* in D. (A tour starts at a vertex, traverses a sequence of arcs in their forward direction, thereby visiting every other vertex once, and returns to the initial vertex.) See Figure 8.1.

The second problem is *Max Cut*. Here we are given an undirected graph $G = (V, E)$ with vertices V and edges E. A cost $c(e)$ is assigned to every edge $e \in E$. A cut $(X, V - X)$ is the set of edges between X and $V - X$. We are to find a cut of maximum total cost.

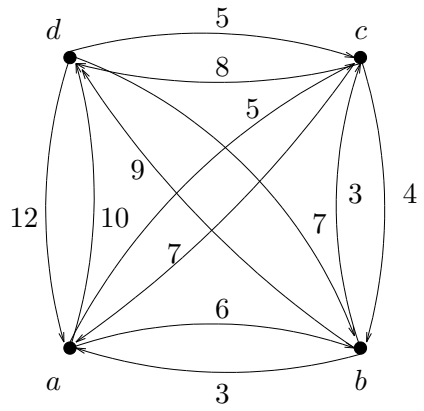


Figure 8.1: A complete digraph

Max Cut is of interest in many practical problems when we are to break graphs into pieces with minimum possible number of edges (and thus maximum possible number of edges between the pieces).

The two problems are among very many *combinatorial optimisation problems*. A combinatorial optimisation problem is given by a set $S = \{s_1, \ldots, s_n\}$ of elements, each of some cost $c(s_i)$, and a collection $\mathcal{F}$ of subsets of S. We wish to find a set F in $\mathcal{F}$ such that the sum of the costs of elements in F is minimum/maximum among all sets in $\mathcal{F}$.

8.2 Greedy-type algorithms

The *greedy algorithm* for a minimisation combinatorial optimisation problem works as follows. We choose an element s_{i_1} of S that is contained in at least one set of $\mathcal{F}$ and of minimum cost among such elements. Form $X = \{s_{i_1}\}$. At every iteration choose an element $s_{i_k} \in S - X$ such that $X \cup \{s_{i_k}\}$ is a subset of some set in $\mathcal{F}$ and s_{i_k} is of minimum cost among all such elements. Add s_{i_k} to X and proceed to the next iteration.

For example, let $S = \{a, b, d, e, f\}$, $c(a) = 0,\ c(b) = 1,\ c(d) = 5,\ c(e) = 0.5,\ c(f) = 2$, and let $\mathcal{F} = \{Y, Z\}$, where $Y = \{b, d, e\},\ Z = \{e, f\}$. At iteration 0 we choose $X = \{e\}$. At iteration 1, we have $X = \{e, b\}$. At iteration 2, we have $X = \{e, b, d\} = Y$.

This example shows the the greedy algorithm not always gives an optimal solution: indeed, $c(Y) = 1 + 5 + 0.5 = 6.5 > c(Z) = 0.5 + 2 = 2.5$. In "Theoretical Analysis of Heuristics", we will see that even for the assignment problem the greedy algorithm may produce the worst solution!

In Figure 8.1 for the ATSP, we see that the greedy algorithm may have a problem to

start: there are two cheapest arcs (b, a) and (b, c). If the greedy algorithm chooses (b, a), then it proceeds to choosing (c, b), (d, c) and (a, d). (It does not choose (a, c) rather than (d, c) because addition of (a, c) would create a cycle shorter than a tour.) The total cost of this tour $T = badcb$ is $3+4+5+10 = 22$. If the greedy algorithm chooses (b, c), then it proceeds to choosing (a, b). (It cannot choose (b, a), (c, b), (d, c) since if we have added any of them, we'd not be able to complete a tour: according to the description of the greedy algorithm at every iteration we can only add arcs such that a subset of some tour(s) is created.) Then we choose (c, d) and (d, a). The cost of this tour, $T' = bcdab$, is 29. Thus, the issue of ties for the greedy algorithm might be important.

A natural problem for the instance of the ATSP in Figure 8.1 is to find a cheapest tour. You can solve this question by examining all six tours of the instance. In general, if the ATSP has n vertices, then there are $(n-1)! = (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1$ tours there. To see that, fix a vertex, say vertex 1. We can move to one of the remaining vertices $2, 3, \ldots, n$ ($n-1$ vertices). After choosing one of them, we can move to one of $n-2$ remaining vertices, etc.

Despite the fact the greedy algorithm does not always produce "good" solutions, it is of use since it is a very simple algorithm and it gives relatively good results for some problems, see "Computational Analysis of Heuristics". There are even problems, for which the greedy algorithm gives always optimal solutions. The most famous such problem is the minimum spanning tree problem.

Even though the greedy algorithm is easy to describe it is not always easy to implement, since we need to check which elements can be added to X and which cannot. This also leads to the fact that the greedy algorithm is not as fast as we would like an algorithm to be for a given combinatorial optimisation problem.

In such difficult cases, specialized greedy-type algorithms can be useful. For the ATSP, such an algorithm is the *nearest neighbour algorithm* (NN). NN proceeds as follows. We start from some vertex, say 1. We move to the nearest to 1, from there to the nearest to that one, etc. (we never create a cycle shorter than a tour). For example, in Figure 8.1, if we start at a, then we move to c, to b, to d, to a, and obtain the tour $T' = acbda$ of cost 30. If we start NN at d, however, we move to c, to b, to a, to d. We have obtained tour $T = dcbad$ of cost 22. This example suggests that it is a good idea to start from each vertex in turn. However, this strategy slows down the algorithm. Indeed, NN is of complexity $O(n^2)$ (see below), but if we start from every vertex we get $O(n \times n^2) = O(n^3)$. Observe that $O(n^3)$ is too large for the ATSP as we cannot use NN even for n equal a few thousands (we'd need several days to get the result). We consider the behavior of NN and its repetitive modification in "Computational Analysis of Heuristics" and "Theoretical Analysis of Heuristics" and see that NN produces results similar to the greedy algorithm even though it is computationally more efficient for the ATSP.

In practice, NN is faster than the greedy algorithm. This is easy to predict by calcu-

lating and comparing the time complexities of the two algorithms. The complexity of NN is $O(n^2)$. The greedy algorithm starts from sorting the costs of arcs in increasing order. Algorithms to sort N numbers are of complexity $O(N \log N)$. There are $N = n(n-1)/2$ numbers to sort for the greedy algorithm. Hence, we need $O(n^2 \log n)$ time to sort the costs. The greedy algorithm can be implemented such that its complexity is $O(n^2 \log n)$. Clearly, $n^2 \log n > n^2$ and thus NN is faster than the greedy algorithm.

8.3 Special algorithms for the ATSP

The *random insertion heuristic* (RI) chooses randomly two initial vertices i_1 and i_2 and forms the cycle $i_1 i_2 i_1$. Then, in every iteration, it chooses randomly a vertex ℓ which is not in the current cycle $i_1 i_2 \dots i_s i_1$ and inserts ℓ in the cycle (i.e., replaces an arc $i_m i_{m+1}$ of the cycle with the path $i_m \ell i_{m+1}$) such that the cost of the cycle increases as little as possible. The heuristic stops when all vertices have been included in the current cycle, i.e., a tour is formed.

We illustrate RI using Figure 8.1. Let's start from the cycle aba and insert c in the optimal manner. We have to choose between the cycles $acba$ and $abca$. While the first cycle increases the cost of aba by $cost(ac) + cost(cb) - cost(ab) = 3$, the second one increases the cost of aba by $cost(bc) + cost(ca) - cost(ba) = 7$. Hence we choose $acba$. Now we have to choose between the cycles $adcba$, $acdba$ and $acbda$. They cause an increase to the cost of the current cycle of $cost(ad) + cost(dc) - cost(ac) = 10$, $cost(cd) + cost(db) - cost(cb) = 11$ and $cost(bd) + cost(da) - cost(ba) = 18$, respectively. Hence, we choose the tour $T = adcba$ of cost 22.

The complexity of RI is $O(n^2)$.

Our next heuristic is based on the operation called *patching*. Let $C = i_1 i_2 \dots i_k i_1$ and $Z = j_1 j_2 \dots j_\ell j_1$ be a pair of disjoint cycles. For any pair s, t of indices the corresponding patching is deletion of the arcs (i_s, i_{s+1}) and (j_t, j_{t+1}) and addition of arcs (i_s, j_{t+1}) and (j_t, i_{s+1}). As a result, we get one cycle $X = i_s j_{t+1} j_{t+2} \dots j_1 j_2 \dots j_t i_{s+1} i_{s+2} \dots i_1 i_2 \dots i_s$.

There are $k\ell$ choices of pair s, t and thus $k\ell$ different patchings of the pair of cycles above. The *cheapest patching* is the one for which the resulting cycle X is cheapest. We can choose the cheapest by selecting X with minimum $cost(i_s j_{t+1}) + cost(j_t i_{s+1}) - cost(i_s i_{s+1}) - cost(j_t j_{t+1})$. So, to find the cheapest patching we need $O(k\ell)$ time.

The patching algorithm can be outlined as follows:

1. Construct a collection F of disjoint cycles covering all vertices of minimum cost.
2. Choose two longest (not cheapest) cycles C and Z in the current F and replace C and Z in F by their cheapest patching.
3. Repeat Step 2 until the current F is reduced to a single cycle, i.e., a tour.

To find a collection F of disjoint cycles covering all vertices of minimum cost, it suffices to solve the Assignment Problem. Indeed, consider a complete digraph $D = (V, A)$ with cost $cost_D(a)$ on every arc a. Construct a complete bipartite graph $B = (V, V'; E)$ in which $V = \{v' : v \in V\}$, i.e., V' is a copy of V, and $cost_B(uv') = cost_D(uv)$ if $u \neq v$ and $cost_B(uv') = M$, where M is a very large constant, otherwise. A minimum cost perfect matching $u_1v'_1, u_2v'_2, ..., u_nv'_n$ in B corresponds to a minimum cost collection of cycles in D with arcs $u_1v_1, u_2v_2, ..., u_nv_n$.

Taking into consideration that to find a cheapest patching for a current pair of cycles we need to consider only arcs not considered before, and that each arc is considered only once, we conclude that the search for a patching will take $O(n^2)$ time. However, to find a collection of disjoint cycles covering all vertices we need to solve the AP. Algorithms for the AP take $O(n^3)$ time (in practice, they are much faster and close to $O(n^2)$). Thus, the patching algorithm's complexity is $O(n^3)$. In practice, however, the complexity is lower and not much larger than $O(n^2)$, which makes the patching algorithm fast enough.

8.4 Improvement local search

The algorithms presented so far are called *construction heuristics*. They produce a solution (a tour for the ATSP) and stop. Their advantage is the fact they are fast, but their disadvantage is that their solution could be of poor quality. To improve a solution produced by a construction heuristic, several approaches are used. The simplest one is *improvement local search*, which we discuss here. However, there are other approaches called meta-heuristics, which we consider later.

The idea of improvement local search is as follows. We have a solution sol_1 produced by a construction heuristic. We look at a collection of solutions somewhat close to sol_1 and try to find there a better (or best solution) sol_2. (We call a collection of solutions somewhat close to sol_1 a *neighbourhood* of sol_1.) At iteration i, we have sol_i found in iteration $i-1$. We proceed by looking at a neighbourhood of sol_i and try to find a solution sol_{i+1} better than sol_i. If sol_{i+1} is not found (none of the solution in the neighbourhood are better than sol_i) we stop.

There are two types of improvement local search: the one where sol_{i+1} is better than sol_i and the other one where sol_{i+1} is the best in the neighborhood of sol_i. In practice mostly the first type is used, in theoretical investigations the second type is mostly used.

To specify improvement local search, we have to define a neighbourhood for every solution. So, we proceed by considering neighbourhoods for the Symmetric TSP (STSP) and Asymmetric TSP (ATSP).

One of the easiest and most useful are k-Opt neighbourhoods. Local search that uses k-Opt neighbourhoods is called *k-Opt*. The k-Opt neighbourhood of a tour T is obtained

by deleting k edges/arcs from T followed by adding k edges/arcs to form a tour.

Let us run one iteration of 2-Opt on the tour *abcdea* of the instance of STSP below, where the vertices are a, b, c, d, e.

$$C = \begin{pmatrix} 0 & 14 & 13 & 11 & 13 \\ - & 0 & 10 & 11 & 13 \\ - & - & 0 & 7 & 8 \\ - & - & - & 0 & 8 \\ - & - & - & - & 0 \end{pmatrix}.$$

In 2-Opt we are given a tour T. The neighbourhood of T is the set of all tours that can be obtained from T by first deleting 2 edges and then adding 2 edges.

The tours belonging to the neighbourhood of tour *abcdea* are:

abcdea 52 (original)
abedca 55 (deleted edges: ea, bc)
abceda 51 (deleted edges: ea, cd)
adcbea 54 (deleted edges: ab, de)
acbdea 55 (deleted edges: ab, cd)
abdcea 53 (deleted edges: bc, de).
2-Opt now picks the best of the above tours.

Now consider 3-Opt for ATSP. If we delete three arcs from a tour, there are only two ways to add three arcs (not necessarily different from the deleted ones) in order to reconstruct a tour. To see this, contract each of the three paths obtained after deletion of three arcs to a vertex. As a result, we get a complete digraph with 3 vertices. This digraph has only two tours.

We can choose three arcs to delete in $n(n-1)(n-2)/6$ ways and each such way leads to 2 tours. Thus, a 3-Opt neighbourhood has $O(n^3)$ tours. Similarly, one can see that a k-Opt neighbourhood has $O(n^k)$ tours. In order to see whether deletion of three arcs and addition of three arcs to form a tour leads to improvement it suffices to find the difference in the cost of deleted and added arcs. If the difference is positive, we have found an improvement. It is important that to examine any new tour we need only constant time. This means that $O(n^k)$ time is enough to find the best tour in a k-Opt neighbourhood. However, even for $k = 3$, the time is too large for even moderate instances of the ATSP. Hence we have to try to restrict our choice of candidates for improvement to only some tours in a 3-Opt neighbourhood, and only if this fails, we may look at the entire neighbourhood. Several possibilities to implement this strategy are considered in the literature on the ATSP, but we will not look at them here.

Question 8.4.1. *Design neighbourhoods for Max Cut and show how we can economically find a better cut in the neighbourhood of a given cut.*

Normally, only 3-Opt neighbourhoods are used in practical implementations since otherwise restrictions on the fraction of tours to consider must be very strong. The reason is that we spend a constant time on a tour. Can we do better? A positive answer to this question is provided in the rest of this section.

Let $C = x_1x_2...x_kx_1$ be a cycle. The operation of *removal* of a vertex x_i $(1 \leq i \leq k)$ results in the cycle $x_1x_2...x_{i-1}x_{i+1}\dots x_kx_1$ (thus, removal of x_i is not deletion of x_i from C; *deletion* of x_i gives the path $x_{i+1}x_{i+2}\dots x_kx_1x_2\dots x_{i-1}$). Let y be a vertex not in C. The operation of *insertion* of y into an arc (x_i, x_{i+1}) results in the cycle $x_1x_2...x_iyx_{i+1}\dots x_kx_1$. The *cost* of the insertion is defined as $c(x_i, y) + c(y, x_{i+1}) - c(x_i, x_{i+1})$. For a set $Z = \{z_1, \dots, z_s\}$ $(s \leq k)$ of vertices not in C, an *insertion* of Z into C results in the tour obtained by inserting the nodes of Z into different arcs of the cycle. In particular, insertion of y into C involves insertion of y into one of the arcs of C.

Let $T = x_1x_2...x_nx_1$ be a tour and let $Z = \{x_{i_1}, x_{i_2}, ..., x_{i_s}\}$ be a set of non-adjacent vertices of T, i.e., $2 \leq |i_k - i_r| \leq n-2$ for all $1 \leq k < r \leq s$. The *assign neighbourhood* of T with respect to Z, $N(T, Z)$, consists of the tours that can be obtained from T by removal of the vertices in Z one by one followed by an insertion of Z into the cycle derived after the removal. For example,

$$N(x_1x_2x_3x_4x_5x_1, \{x_1, x_3\}) =$$
$$\{x_2x_ix_4x_jx_5x_2, x_2x_ix_4x_5x_jx_2, x_2x_4x_ix_5x_jx_2 : \{i, j\} = \{1, 3\}\}.$$

The neighbourhood $N(T, Z)$ has exactly $(n-s)!/(n-2s)!$ tours. In particular, if $s = n/2$, then $N(T, Z)$ has $(n/2)!$ tours. Thus, $N(T, Z)$ has exponential number of tours. Interestingly we do not need to spend exponential time to find the best tour in that neighbourhood.

Theorem 8.4.2. *The best tour in the neighbourhood $N(T, Z)$ can be found in time $O(n^3)$.*

Proof (*non-examined*): Let $C = y_1y_2...y_{n-s}y_1$ be the cycle obtained from T after removal of Z and let $Z = \{z_1, z_2, ..., z_s\}$. By the definition of insertion, we have $n - s \geq s$. Let ϕ be an injective mapping from Z to $Y = \{y_1, y_2, ..., y_{n-s}\}$. (The requirement that ϕ is injective means that $\phi(z_i) \neq \phi(z_j)$ if $i \neq j$.) If we insert some z_i into an arc (y_j, y_{j+1}), then the cost of C will be increased by $c(y_j, z_i) + c(z_i, y_{j+1}) - c(y_j, y_{j+1})$. Therefore, if we insert every z_i, $i = 1, 2, ..., s$, into $y_{\phi(i)}y_{\phi(i)+1}$, the cost of C will be increased by

$$g(\phi) = \sum_{i=1}^{s} c(y_{\phi(i)}, z_i) + c(z_i, y_{\phi(i)+1}) - c(y_{\phi(i)}, y_{\phi(i)+1}).$$

Clearly, to find a cheapest tour of $N(T, Z)$, it suffices to minimize $g(\phi)$ on the set of all injections ϕ from Z to Y. This can be done using the following complete bipartite graph G. The partite sets of G are Z and Y. The cost of an edge z_iy_j is set to be $c(y_j, z_i) + c(z_i, y_{j+1}) - c(y_j, y_{j+1})$.

By the definition of G, every maximum matching M of G corresponds to an injection ϕ_M from Z to Y. Moreover, the costs of M and ϕ_M coincide. A cheapest maximum matching in G can be found by solving the assignment problem. Therefore, in $O(n^3)$ time, we can find the best tour in $N(T, Z)$. QED

8.5 Questions

Question 8.5.1. *Give the definition of a combinatorial optimisation (CO) problem. Formulate the greedy algorithm for CO. Give examples when the greedy algorithm finds the worst solution.*

Question 8.5.2. *Apply* Nearest Neighbour, Repeated Nearest Neighbour *and* Greedy *algorithms to the STSP instance of Fig. 7.2.*

Question 8.5.3. *Describe the greedy, nearest neighbour and random insertion algorithms for the asymmetric travelling salesman problem. Illustrate the algorithms on an instance of the ATSP with 5 vertices.*

Question 8.5.4. *Describe the greedy algorithm for Max Cut.*

Question 8.5.5. *Describe the ideas of local search algorithms for the Symmetric TSP, in general, and k-Opt, in particular.*

Question 8.5.6. *Let us run one iteration of 2-Opt on the tour abedca of the instance of STSP given in Section 8.4.*

Chapter 9

Computational Analysis of Heuristics

9.1 Experiments with ATSP heuristics

This section is based on a chapter by D.S. Johnson, G. Gutin, L. McGeoch, A. Yeo, W. Zhang and A. Zverovich in the book "The Traveling Salesman Problem and its Variations", G. Gutin and A. Punnen (eds.), Kluwer, 2002. In this section we consider only part of heuristics analyzed in the chapter, namely, `Patch, COP, 3opt, Greedy` and `NN`. We described all these heuristics, apart from `COP`, earlier. `COP` is an improved version of `Patch`, which is not described in these notes.

In many cases it is impossible to find optimal tours for the instances of the ATSP considered here. To see how far the tour obtained by a heuristic is from optimum, we use lower bounds. One lower bound is the AP lower bound, i.e., the cost of a cheapest collection of disjoint cycles. To see that this is indeed a lower bound, it suffices to notice that a tour is a collection of disjoint cycles (that consists of a unique cycle). Another lower bound is the so-called Held-Karp lower bound (HK), which is normally better than AP. To compute HK one needs to solve several LP problems related to the ATSP (some LP relaxations of the ATSP); we will not provide details on HK.

9.2 Testbeds

There are various families of instances of the ATSP that are of practical and theoretical interest. Thus, it makes sense to study the behavior of ATSP heuristics not on one family of instances, but a set of families. We start by giving short description of the families of instances and analyzing their properties. We first family, `rect`, has not been used in the

experiments, but provided a basis for some other families.

Random 2-Dimensional Rectilinear Instances (`rect`). The cities correspond to random points uniformly distributed in a 10^6 by 10^6 square, and the distance between points (x_1, y_1) and (x_2, y_2) is $|x_2 - x_1| + |y_2 - y_1|$.

Notation. In what follows, ATSP instances have N cities (vertices) $c_1, c_2, \ldots, c_N$. The distance from c_i to c_j is denoted by $d(c_i, c_j)$.

Random Asymmetric Matrices (`amat`). The random asymmetric distance matrix generator chooses each distance $d(c_i, c_j)$ as an independent random integer x, $0 \leq x \leq 10^6$. For these instances it is known that both the optimal tour length and the AP bound approach a constant (the same constant) as $n \to \infty$. The rate of approach appears to be faster if the upper bound U on the distance range is smaller.

Shortest-Path Closure of `amat` (`tmat`). One of the reasons the previous class is uninteresting is the total lack of correlation between distances. Note that instances of this type are unlikely to obey the triangle inequality, i.e., there can be three cities c_1, c_2, c_3 such $d(c_1, c_3) > d(c_1, c_2) + d(c_2, c_3)$. A somewhat more reasonable instance class can be obtained by taking Random Asymmetric Matrices and closing them under shortest path computation. That is, if $d(c_i, c_j) > d(c_i, c_k) + d(c_k, c_j)$ then set $d(c_i, c_j) = d(c_i, c_k) + d(c_k, c_j)$ and repeat until no more changes can be made. This is also a commonly studied class.

Tilted Drilling Machine Instances, Additive Norm (`rtilt`). These instances correspond to the following potential application. One wishes to drill a collection of holes on a tilted surface, and the drill is moved using two motors. The first moves the drill to its new x-coordinate, after which the second moves it to its new y-coordinate. Because the surface is tilted, the second motor can move faster when the y-coordinate is decreasing than when it is increasing. The generator starts with an instance of `rect` and modifies it based on three parameters: u_x, the multiplier on $|\Delta x|$ that tells how much time the first motor takes, and u_y^+ and u_y^-, the multipliers on $|\Delta y|$ when the direction is up/down. For this class, the parameters $u_x = 1$, $u_y^+ = 2$, and $u_y^- = 0$ were chosen, which yields the same optimal tour lengths as the original symmetric `rect` instances because in a cycle the sum of the upward movements is precisely balanced by the sum of the downward ones.

Tilted Drilling Machine Instances, Sup Norm (`stilt`). For many drilling machines, the motors operate in parallel and so the proper metric is the maximum of the times to move in the x and y directions rather than the sum. This generator has the same three parameters as for `rtilt`, although now the distance is the maximum of $u_x|\Delta x|$ and $u_y^-|\Delta y|$ (downward motion) or $u_y^+|\Delta y|$ (upward motion). For this class, the parameters $u_x = 2$, $u_y^+ = 4$, and $u_y^- = 1$ were chosen.

Random Euclidean Stacker Crane Instances (`crane`). In the *Stacker Crane Problem* one is given a collection of source-destination pairs s_i, d_i in a metric space where

for each pair the crane must pick up an object at location s_i and deliver it to location d_i. The goal is to order these tasks so as to minimize the time spent by the crane going between tasks, i.e., moving from the destination of one pair to the source of the next one. This can be viewed as an ATSP in which city c_i corresponds to the pair s_i, d_i and the distance from c_i to c_j is the metric distance between d_i and s_j. The generator has a single parameter $u \geq 1$, and constructs its source-destination pairs as follows. The sources are picked as in an instance of `rect`. Then we pick two integers x and y uniformly and independently from the interval $[-10^6/u, 10^6/u]$. The destination is the vector sum $s + (x, y)$. In order to preserve a sense of geometric locality, we let u vary as a function of N, choosing values so that the expected number of other sources that are closer to a given source than its destination is roughly a constant, independent of N. These instances do not necessarily obey the triangle inequality since the time for traveling from source to destination is not counted.

Disk Drive Instances (`disk`). These instances attempt to capture some of the structure of the problem of scheduling the read head on a computer disk. This problem is similar to the stacker crane problem in that the files to be read have a start position and an end position in their tracks. Sources are again generated as in `rect` instances, but now the destination has the same y-coordinate as the source. To determine the destination's x-coordinate, we generate a random integer $x \in [0, 10^6/u]$ and add it to the x-coordinate of the source modulo 10^6, thus capturing the fact that tracks can wrap around the disk. The distance from a destination to the next source is computed based on the assumption that the disk is spinning in the x-direction at a given rate and that the time for moving in the y direction is proportional to the distance traveled at a significantly slower rate. To get to the next source we first move to the required y-coordinate and then wait for the spinning disk to deliver the x-coordinate to us.

Pay Phone Coin Collection Instances (`coin`). These instances model the problem of collecting money from pay phones in a grid-like city. We assume that the city is a k by k grid of city blocks with 2-way streets running between them and a 1-way street running around the exterior boundary of the city. The pay phones are uniformly distributed over the boundaries of the blocks. We can only collect from a pay phone if it is on the same side of the street as we are currently driving on, and we cannot make "U-turns" either between or at street corners. Finding the shortest route is trivial if there are so many pay phones that most blocks have one on all four of their sides. This class is thus generated by letting k grow with n, in particular as the nearest integer to $10\sqrt{n}$.

No-Wait Flowshop Instances (`shop`). In a k-processor no-wait flowshop, a job $\bar{u}$ consists of a sequence of tasks $(u_1, u_2, \ldots, u_k)$ that must be performed by a fixed sequence of machines. The processing of u_{i+1} must start on machine $i + 1$ as soon as processing of u_i is complete on machine i. This models the processing of heated materials that must not be allowed to cool down and situations where there is no storage space to hold waiting jobs. These instances have $k = 50$ processors and task lengths are independently chosen

random integers between 0 and 1000. The distance from job $\bar{v}$ to job $\bar{u}$ is the minimum possible amount by which the finish time for u_k can exceed that for v_k if $\bar{u}$ is the next job to be started after $\bar{v}$.

Approx. Shortest Common Superstring Instances (`super`). This class is intended to capture some of the structure in a computational biology problem relevant to genome reconstruction. Given a collection of strings C, we wish to find a short superstring S in which all are (at least approximately) contained. If we did not allow mismatches the distance from string A to string B would be the length of B minus the length of the longest prefix of B that is also a suffix of A. Here we add a penalty equal to twice the number of mismatches, and the distance from string A to string B is the length of B minus $\max\{j+2k$: there is a prefix of B of length j that matches a suffix of A in all but k positions$\}$. The generator uses this metric applied to random binary strings of length 20.

Instance Properties. In what follows we shall consider which measurable properties of instances correlate with heuristic performance. Likely candidates include (1) the gap between the AP and HK bounds, (2) the extent to which the distance metric departs from symmetry, and (3) the extent to which it violates the triangle inequality. The specific metrics we use for these properties are as follows. For (1) we use the percentage by which the AP bound falls short of the HK bound. For (2) we use the ratio of the average value of $|d(c_i, c_j) - d(c_j, c_i)|$ to the average value of $|d(c_i, c_j) + d(c_j, c_i)|$, a quantity that is 0 for symmetric matrices and has a maximum value of 1. For (3) we first compute, for each pair c_i, c_j of distinct cities, the minimum of $d(c_i, c_j)$ and $\min\{d(c_i, c_k) + d(c_k, c_j) : 1 \leq k \leq n\}$ (call it $d'(c_i, c_j)$). The metric is then the average, over all pairs c_i, c_j, of $(d(c_i, c_j) - d'(c_i, c_j))/d(c_i, c_j)$. A value of 0 implies that the instance obeys the triangle inequality.

Table 9.1 reports the values for these metrics on our randomly generated classes. For the random instances, average values are given for $n = 100$, 316, and 1,000. In Table 9.1 the classes are ordered by increasing value of the HK-AP gap for the 1,000-city entry. For the random instance classes, there seems to be little correlation between the three metrics (1),(2) and (3), although for some there is a dependency on the number n of vertices.

9.3 Comparison of TSP heuristics

Since TSP heuristics are normally used when $n \geq 1000$, to analyze the relative value of heuristics, we will mostly consider computational results for $n = 3162, 1000$ (see Tables 6 and 7), but the rest of results is also of interest to predict the behaviour of the heuristics for $n > 3162$.

We first analyze the relative performance of the heuristics for the family instances separately and then make more general conclusions.

For `tmat`, `COP` appears to be the best heuristic w.r.t. both time and quality. `Patch`

	% HK-AP			Asymmetry			Triangle		
	100	316	1,000	100	316	1,000	100	316	1,000
tmat	.34	.16	.03	.232	.189	.165	–	–	–
amat	.64	.29	.05	.333	.332	.333	.633	.752	.837
shop	.50	.22	.15	.508	.498	.515	–	–	–
disk	2.28	.71	.34	.044	.045	.046	.250	.313	.354
super	1.04	1.02	1.17	.076	.075	.075	–	–	–
crane	7.19	6.34	5.21	.061	.035	.020	.101	.087	.066
coin	15.04	13.60	13.96	.010	.007	.003	–	–	–
stilt	18.41	14.98	14.65	.329	.333	.336	–	–	–
rtilt	20.42	17.75	17.17	.496	.500	.503	–	–	–

Table 9.1: For the 100-, 316-, and 1000-city instances of each class, the average percentage shortfall of the AP bound from the HK bound and the average asymmetry and triangle inequality metrics as defined in the text. "–" stands for .000.

appears to be the second best choice. For amat, all heuristic are of similar running time, but the COP and Patch solutions are of higher quality, which COP being the best. For shop, Greedy, NN and 3opt are fast heuristics, while Patch and COP are rather slow. COP is particularly slow, and moreover it is too slow for a heuristic algorithm. But, if the running time is not an issue, then Patch and COP are the heuristics of choice as they produce tours close to optimal. If the time is an issue, then 3opt should be used.

Similar comments are for disk. For super, COP is again the heuristic of choice if the time is not very limited. If the time is limited, 3opt provides the best choice. We leave it to the reader to comment on crane. We observe that 3opt is of higher quality than Patch or COP for coin, stilt and rtilt and is the heuristic of choice for the tree families of TSP instances.

Overall, we see that COP and 3opt are the best candidates and should be used together with 3opt running first. If the quality of its solution is sufficient, we stop, and otherwise run COP. The problem with COP is that its running time grows very quickly and the heuristic becomes too slow for large values of n.

`Greedy`

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
`tmat`	31.23	29.04	26.53	26.25	.03	.26	1.7	20
`amat`	243.09	362.86	418.56	695.29	.04	.27	1.9	21
`shop`	49.34	56.07	61.55	66.29	.03	.26	2.1	40
`disk`	188.82	307.14	625.76	1171.62	.03	.28	2.7	23
`super`	6.03	5.40	5.16	5.79	.03	.22	1.5	18
`crane`	41.86	44.09	39.70	41.60	.03	.27	1.9	21
`coin`	48.73	46.76	42.33	35.94	.04	.24	1.7	20
`stilt`	106.25	143.89	178.34	215.84	.04	.28	1.9	23
`rtilt`	350.12	705.56	1290.63	2350.38	.03	.28	2.0	23

`NN`

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
`tmat`	38.20	37.10	37.55	36.66	.03	.24	1.7	20
`amat`	195.23	253.97	318.79	384.90	.03	.26	1.9	21
`shop`	16.97	14.65	13.29	11.87	.03	.23	2.5	20
`disk`	96.24	102.54	115.51	161.99	.04	.27	1.9	23
`super`	8.57	8.98	9.75	10.62	.03	.21	1.5	18
`crane`	40.72	41.66	43.88	43.18	.03	.26	1.9	21
`coin`	26.08	26.71	26.80	25.60	.03	.23	1.7	20
`stilt`	30.31	30.56	27.62	24.79	.03	.30	1.9	22
`rtilt`	28.47	28.28	27.52	24.60	.04	.26	1.9	22

`3opt`

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
`tmat`	6.44	9.59	12.66	16.20	.19	1.71	5.5	20
`amat`	39.23	58.57	83.77	112.08	.19	1.75	5.8	21
`shop`	3.02	7.25	10.22	10.88	.23	1.78	5.6	21
`disk`	12.11	16.96	20.85	25.64	.19	1.82	6.1	23
`super`	3.12	4.30	5.90	7.94	.15	1.43	4.8	18
`crane`	9.48	9.41	10.65	10.64	.19	1.76	7.3	22
`coin`	8.06	9.39	9.86	9.92	.18	1.62	5.3	20
`stilt`	11.39	12.65	12.62	12.27	.19	1.80	8.2	22
`rtilt`	10.04	13.09	18.00	19.83	.19	2.05	6.6	23

Table 9.2: Tour quality and running times for Greedy, NN, and 3-Opt.

Patch

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
tmat	.84	.64	.17	.00	.03	.22	1.8	29
amat	10.95	6.50	2.66	1.88	.03	.22	1.9	18
shop	1.15	.59	.39	.24	.04	.48	8.4	260
disk	9.40	2.35	.88	.30	.03	.26	2.9	75
super	1.86	2.84	3.99	6.22	.02	.19	1.7	29
crane	9.40	10.18	9.45	8.24	.03	.21	1.5	23
coin	16.48	16.97	17.45	18.20	.02	.18	1.4	17
stilt	23.33	22.79	23.18	24.41	.03	.24	2.2	29
rtilt	17.03	18.91	18.38	19.39	.03	.28	2.9	54

COP

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
tmat	.57	.36	.16	.00	.01	.12	.7	15
amat	9.31	3.15	2.66	1.01	.01	.15	.6	26
shop	.68	.36	.19	.10	.08	1.41	29.1	1152
disk	6.00	1.13	.51	.15	.03	.31	8.7	297
super	1.01	1.20	1.22	2.06	.03	.24	4.6	243
crane	10.32	9.08	7.28	6.21	.04	.44	3.5	53
coin	16.44	17.68	16.23	16.06	.02	.10	1.2	22
stilt	22.48	23.31	22.80	22.90	.07	.94	8.1	105
rtilt	19.62	22.86	20.95	20.37	.05	.33	5.6	117

Table 9.3: Tour quality and running times for the patching algorithm and COP.

Chapter 10

Theoretical Analysis of Heuristics

Previously we have considered experimental performance of TSP heuristics. While experimental analysis is of a certain importance, it cannot cover all possible families of TSP instances and, in particular, it normally does not cover the hardest ones. Experimental analysis provides little theoretical explanation why certain heuristics are successful while some others are not. This limits our ability to improve on the quality and efficiency of existing algorithms. It also limits our ability to extend approaches successful for the TSP to other combinatorial optimization (CO) problems.

This part of the course is devoted to theoretical approaches that allow one to analyze properties of optimal solutions of heuristics.

10.1 Property of 2-Opt optimal tours

An instance of the Euclidean TSP is given by a collection of points in the plane (vertices); the distance between any pair of points $u = (x_u, y_u)$, $v = (x_v, y_v)$ is the Euclidean distance between the points, i.e. $\text{dist}(u, v) = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}$. Clearly, the Euclidean TSP is symmetric, i.e. $\text{dist}(u, v) = \text{dist}(v, u)$.

Recall that the 2-Opt neighbourhood of a tour T consists of all tours that can be obtained from T by deleting two edges of T and than adding two edges.

Theorem 10.1.1. *For the Euclidean TSP, a tour T which is the best in its 2-Opt neighbourhood, does not have self-intersections.*

Proof: Suppose that the theorem is wrong. Let T be a tour which is the best in its 2-Opt neighbourhood but has self-intersection; let y be such an intersection. See Figure 10.1; note that y is not a vertex (no vertex can be visited twice). To prove that the intersection in the figure is impossible, it suffices to see that the tour T' that is obtained

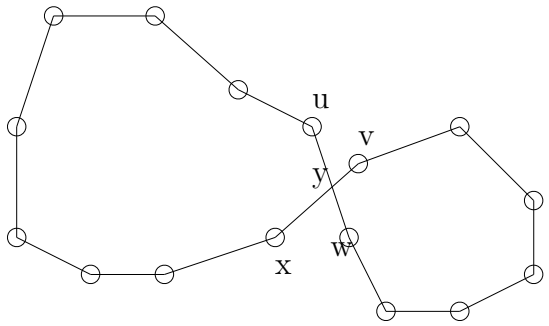


Figure 10.1: A tour with self-intersection

from T by deleting edges xv and uw and adding edges uv and xw is shorter than T, i.e. $\text{dist}(x,v) + \text{dist}(u,w) > \text{dist}(u,v) + \text{dist}(x,w)$. (Indeed, T' is in the 2-Opt neighbourhood of T and thus has to be longer than T by the condition of the theorem.)

By the triangle inequality, we have:

$$\text{dist}(u,y) + \text{dist}(y,v) > \text{dist}(u,v) \text{ and } \text{dist}(x,y) + \text{dist}(y,w) > \text{dist}(x,w).$$

Add these two inequalities:

$$(\text{dist}(u,y) + \text{dist}(y,w)) + (\text{dist}(x,y) + \text{dist}(y,v)) > \text{dist}(u,v) + \text{dist}(x,w),$$

or $\text{dist}(x,v) + \text{dist}(u,w) > \text{dist}(u,v) + \text{dist}(x,w)$. QED

10.2 Approximation Analysis

In Approximation Analysis, we study the *approximation ratios* of heuristics. The *approximation ratio* of a heuristic H for the (TSP) is an upper bound on the ratio c/c^*, where c is the cost of a tour found by H and c^* is the optimal cost of a tour.

10.2.1 Travelling Salesman Problem

In what follows, we assume that all **costs** are **non-negative**.

As an example, we consider the double tree heuristic (DTH) for the Symmetric TSP with triangle inequality, i.e. with the inequality $\text{cost}(x,y) + \text{cost}(y,z) \geq cost(x,z)$ for any triple x, y, z of vertices.

To introduce DTH we recall some notions of graph theory. A *multigraph* G is a graph that can have parallel edges. The *degree* of a vertex v of G is the number of edges with

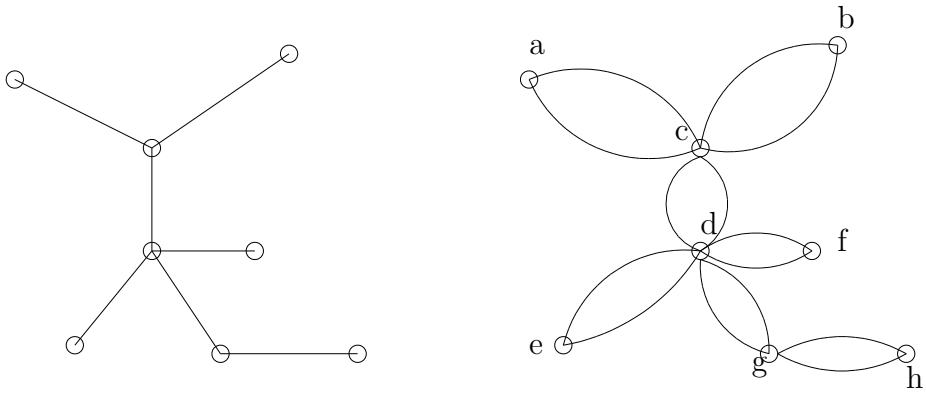


Figure 10.2: An example for DTH

v as an end-vertex. In an undirected multigraph G a *trail* is a sequence of distinct edges such that every two consecutive edges have a common vertex. If every edge belongs to a trail and the trail starts and ends at the same vertex, then the trail is called *Euler*. Not every multigraph has an *Euler* trail. Multigraphs that have Euler trails are called *Euler*.

Theorem 10.2.1 (Euler's Theorem)**.** *A multigraph G has an Euler trail if and only if G is connected and every vertex has even degree.*

In DTH, we

1. Find a minimum cost spanning tree S^* in the complete graph of the STSP
2. Double its edges obtaining an Euler multigraph G_E (see Euler Theorem)
3. Find an Euler trail F in G_E
4. Going along F delete any repetition of vertices in F (apart from the first and last ones), obtaining a tour T

DTH is illustrated in Figure 10.2.

Suppose that a minimum cost spanning tree is given in the left hand side of the figure. After doubling its edges, we get the graph in the right hand side of the figure. We find an Euler trail (as a sequence of vertices for simplicity):

$$acbcdfdedghgdca.$$

After deleting vertex repetitions (apart from the first and last ones), we get a tour $T = acbdfegha$.

DTH is also illustrated in Figures 10.3 and 10.6. Figure 10.3 depicts an STSP instance and Figure 10.6 a minimum cost spanning tree. Doubling edges of this tree, we obtain an

Euler graph with Euler trail *adcbcecda*. After removal of repeated vertices (apart from the first and last ones), we get a tour *adcbea* of cost 23.

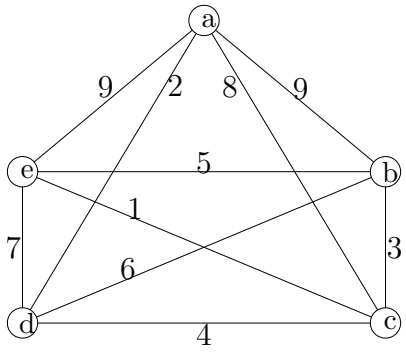


Figure 10.3: STSP instance.

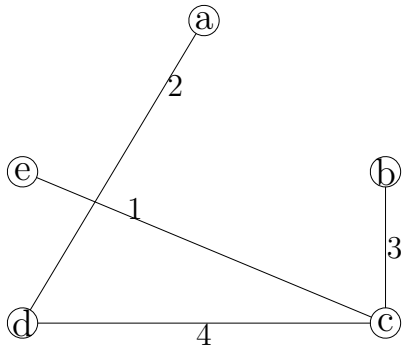


Figure 10.4: Minimum cost spanning tree.

Theorem 10.2.2. *For the Symmetric TSP with triangle inequality, a tour found by DTH is at most twice as long as an optimal tour.*

Proof: Consider an instance of the Symmetric TSP with triangle inequality. Let T^* be an optimal tour of the instance. Observe that after deleting an edge from T^*

we get a spanning tree S of the complete graph. Clearly, $\text{cost}(S) \leq \text{cost}(T^*)$. By the definition of S^*, $\text{cost}(S^*) \leq \text{cost}(S)$. Observe that by the definitions of G_E and F, we have $\text{cost}(F) = \text{cost}(G_E) = 2\text{cost}(S^*)$. By the triangle inequality, any short cuts (i.e. deleting repetitive vertices in F) cannot increase the cost of the derived tour T. Hence, $\text{cost}(T) \leq \text{cost}(F)$. Thus,

$$\text{cost}(T) \leq \text{cost}(F) = 2\text{cost}(S^*) \leq 2\text{cost}(S) \leq 2\text{cost}(T^*).$$

QED

DTH's approximation of at most 100% is not the best known for STSP with triangle inequality. In 1976 Nicos Christofides obtained an algorithm with approximation at most 50 %. In 1998 A.I. Serdyukov independently designed the same algorithm. It took more than 40 years to obtained a very slightly better (but only randomized) algorithm. Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan obtained such an algorithm in July 2020.

Theorem 10.2.3. *For the Symmetric TSP with triangle inequality, a tour found by Christofides's algorithm is at most 50% longer than an optimal tour.*

Christofides's algorithm uses the following result in graph theory.

Theorem 10.2.4. *In a graph, the number of vertices of odd degree is even.*

Let K be a weighted complete graph, let Q be a spanning subgraph of G and let O be the set of vertices of Q of odd degree. (By Theorem 10.2.4, the size of O, $|O|$, is odd.) A *matching* of vertices of O is a set of $|O|/2$ edges of K such that every vertex of O belongs to one of the edges. A matching is *cheapest* if the cost of the matching is cheapest possible. There is a polynomial-time algorithm for finding a cheapest matching, but we do not study it in this module.

Christofides's algorithm is similar to DTH (only Step 2 is different):

1. Find a minimum cost spanning tree S^* in the complete graph of the STSP.

2. Let O be the set of odd-degree vertices of S^*. Construct the complete graph K with vertex set O in which the cost of an edge xy ($x \neq y \in O$) equals the cost of xy in the complete graph of the STSP. Find a cheapest matching M of K and add the edges of M to S^* obtaining an Euler multigraph $G_E = S^* + M$ (see Euler's Theorem).

3. Find an Euler trail F in G_E.

4. Going along F delete any repetition of vertices in F (apart from the first and last ones), obtaining a tour T.

Consider again the STSP of Figure 10.3. Figure 10.6 depicts a minimum cost spanning tree S^*. The vertices of odd degree in S^* are a, b, c, e. Figure depicts the complete graph

K on vertices a, b, c, e. Recall that the costs in K are the lengths of cheapest paths, i.e., the cost of ab is 8 because adb is the cheapest path between a and b. (Check yourself that all other costs are correct in K.)

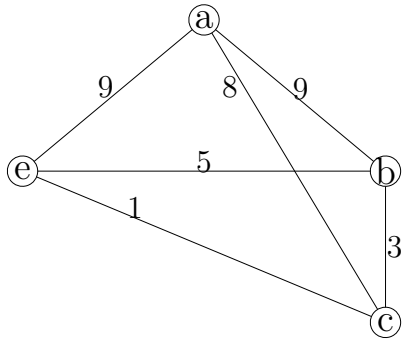


Figure 10.5: Graph K.

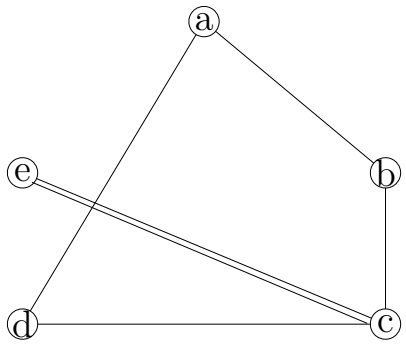


Figure 10.6: Graph G_E.

The cheapest matching M of the odd-degree vertices of the spanning tree S^* of Figure 10.6 is ab and ec (M can be found by inspection). The graph $G_E = S^* + M$ is Euler and has an Euler trail $abcecda$. After removal of repeated vertices (apart from the first and

last ones), we get a tour *abceda* of cost 22. (It's a bit cheaper than the tour obtained by DTH.)

10.2.2 Knapsack Problem

Recall the knapsack problem. Assume that we have a number of items, and we must choose some subset of the items to fill our "knapsack", which has limited space b. Each item, i, has a value v_i and takes up b_i units of space in the knapsack. We wish to choose a collection of the items with total space less than b and with maximum total value.

In what follows we assume that $b_i \leq b$ since any item j with $b_j > b$ cannot be put in the knapsack. Also assume that $b_1 + b_2 + \cdots + b_n > b$ since otherwise the problem is trivial.

For the branch-and-bound algorithm above, we used the following simple heuristic H_1: order items in the non-increasing value of the ratio $r_i = v_i/b_i$ and place in the knapsack as many items as possible putting them in the obtained order. The heuristic H_1 seems very good, but there are examples that show that this is not true.

Consider the following example, let $b = 400$ and there are four items with $v_1 = v_2 = v_3 = 1$, $b_1 = b_2 = b_3 = 1$, $v_4 = 399$, $b_4 = 400$. The heuristic will compute the ratios $r_1 = r_2 = r_3 = 1$ and $r_4 = \frac{399}{400}$ and put the first three items in the knapsack. The value obtained is 3. However, if we put only the forth item in the knapsack, we get value equal 399. This is the optimal solution. Thus, the solution obtained by the heuristic is 133 times smaller than that of the optimal solution.

This example show that the heuristic is not that good after all. Perhaps, the reason for that is that we do not consider the largest value item for inclusion in our solution. Consider another heuristic H_2 that puts the most valuable item in the knapsack first and then applies H_1 to the remaining items. This heuristic would find the optimal solution in the example above, but it'll fail badly on other examples (for example, take $v_4 = 2$).

Let us combine H_1 and H_2, i.e., run both of them on input and choose the best among the obtained solutions. We denote this heuristic by H. We show that there is a guarantee of the quality of solutions obtained by H.

Theorem 10.2.5. *The value of the solution obtained by H is always at least half of the value of the optimal solution.*

Proof: (**Not examined**) Consider an arbitrary instance of the knapsack problem given by n items with values $v_1, v_2, \ldots, v_n$ and sizes $b_1, b_2, \ldots, b_n$. Let $r_i = v_i/b_i$ as before. We may assume that $r_1 \geq r_2 \geq \cdots \geq r_n$. Let v_{opt} be the value of the optimal solution and v_H the value of the solution obtained by H. Clearly, $v_H = \max\{v_{H_1}, v_{H_2}\}$, where v_{H_k} is the value of the solution obtained by H_k.

For some j, the heuristic H_1 will put, in the knapsack, items $1, 2, \ldots, j-1$ one by one. Suppose that item j will not fit into knapsack with the fist $j-1$ items already there. Thus,

$$\hat{v}_j := v_1 + v_2 + \cdots + v_{j-1} = v_{H_1} \leq v_H$$

and

$$\hat{b}_j = b_1 + b_2 + \cdots + b_{j-1} \leq b.$$

Since $r_1 \geq r_2 \geq \cdots \geq r_n$, if we were allowed to place, in the knapsack, part of item j, we would get the (LP) optimal solution $\hat{v}_j + (b - \hat{b}_j)r_j$. This is not worse than v_{opt}. Thus,

$$v_{opt} \leq \hat{v}_j + (b - \hat{b}_j)v_j/b_j < \hat{v}_j + v_j.$$

The last inequality follows from the fact that $(b - \hat{b}_j)/b_j < 1$.

To complete the proof we consider two possible cases. If $v_j \leq \hat{v}_j$ then

$$v_{opt} < \hat{v}_j + v_j \leq 2\hat{v}_j \leq 2v_{H_1} \leq 2v_H.$$

If $v_j > \hat{v}_j$ then $v_{max} > \hat{v}_j$, where v_{max} is the maximum value of an item. Thus,

$$v_{opt} < \hat{v}_j + v_j \leq \hat{v}_j + v_{max} < 2v_{max} \leq 2v_H.$$

In both cases, the theorem follows. QED

Section 6.3 has an instance of the knapsack problem for which H gives the optimal solution, while the use of H_1 requires lengthy computations by branch-and-bound to get the optimal solution.

10.2.3 Bin Packing Problem

Recall that in the Bin Packing Problem (BPP), we are given a positive integer number b and a sequence of N items of sizes $L = (s_1, s_2, \ldots, s_N)$ such that $0 < s_i \leq b$. Our aim is to pack the items into minimum number of bins, each of capacity b. For example, if $b = 1$ and $L = (\frac{1}{2}, \frac{1}{2}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, then the minimum number of bins is 2 as we can pack the first two items in Bin 1 and the remaining items in Bin 2.

One of the simplest heuristics for BPP is the Next Fit heuristic (NF). In NF we start from Bin 1. We pack items into Bin 1 one by one as long as its capacity b is not exceeded. Once the capacity is exceeded, we put the current item (not fitting into Bin 1) into Bin 2 and pack items into Bin 2 as long as its capacity b is not exceeded, etc.

Example. Let $b = 1$ and

$$L = (\frac{1}{2}, \frac{1}{6}, \frac{1}{2}, \frac{1}{6}, \frac{1}{2}, \frac{1}{6}, \frac{1}{2}, \frac{1}{6}, \frac{1}{2}, \frac{1}{6}, \frac{1}{2}, \frac{1}{6}).$$

Since a pair 1/2, 1/2 completely fill in a bin and six 1/6 also fill in a bin, the optimal number of bins is 4 (no wasted space at all). Check that NF fills in 6 bins!

In what follows, assume $b = 1$.

Theorem 10.2.6. *Let p be the minimum number of bins required in BPP; then NF always packs items in at most $2p$ bins.*

Proof: Let p' be the number of bins used by NF and let $b_1, b_2, \ldots, b_{p'}$ be the packed part of Bins 1,2,$\ldots, p'$, respectively. By the definition of NF, $b_i + b_{i+1} > 1$ for each $i = 1, 2, \ldots, p' - 1$ (otherwise Bin i can be used instead of Bins i and $i + 1$). Summing up and adding b_1 and $b_{p'}$, we get $2(b_1 + b_2 + \cdots + b_{p'}) > p' - 1$. However, $p \geq b_1 + b_2 + \cdots + b_{p'}$ and, thus,

$$2p \geq 2(b_1 + b_2 + \cdots + b_{p'}) > p' - 1.$$

Therefore, $2p > p' - 1$ and, since $2p$ is an integer, $2p \geq p'$. QED.

There are examples that generalize the example above that show that the theorem is asymptotically sharp (i.e., cannot be improved).

Consider a little bit more sophisticated heuristic First Fit (FF). In FF we start from Bin 1. We pack items into Bin 1 one by one as long as its capacity b is not exceeded. Once the capacity is exceeded, we put the current item (not fitting into Bin 1) into Bin 2. The next item is placed in Bin 1 if it can be put there. If not, it is placed in Bin 2 if it can be put there. In the case neither of Bin 1 and Bin 2 can accommodate the item, the item is put in Bin 3, etc.

The following theorem (without proof) holds.

Theorem 10.2.7. *Let p be the minimum number of bins required in BPP; then FF always packs items in at most $1.7p + 2$ bins.*

As we can see, FF appears to be better than NF in the worst case.

The First Fit Decreasing heuristic (FFD) puts the items in non-increasing order of their sizes.

Theorem 10.2.8. *Let p be the minimum number of bins required in BPP; then FFD always packs items in at most $1.5p$ bins.*

Proof: (scheme) Let $s_1 \leq s_2 \leq \ldots \leq s_N$. Partition the items into four sets

$$A = \{i : \ s_i > 2/3\}, \ B = \{i : \ 1/2 < s_i \leq 2/3\},$$

$$C = \{i : \ 1/3 < s_i \leq 1/2\}, \ D = \{i : \ s_i < 1/3\}.$$

Consider the solution obtained by FFD. If there is at least one bin that contains only items from D, then there is at most one used bin of occupancy less than $2/3$, and the bound follows.

If there is no bin containing only items from D, then it possible to prove that the solution is optimal. QED.

10.2.4 Further Examples

Let us run the three heuristics above on the following example: Let $b = 1$ be the capacity of each bin and $L = (\frac{1}{2}, \frac{1}{6}, \frac{1}{2}, \frac{1}{4}, \frac{1}{3}, \frac{3}{4}, \frac{1}{2})$.

In this example NF will pack the items into the following 5 bins: $\{\frac{1}{2}, \frac{1}{6}\}$, $\{\frac{1}{2}, \frac{1}{4}\}$, $\{\frac{1}{3}\}$, $\{\frac{3}{4}\}$ and $\{\frac{1}{2}\}$.

FF will pack the items into the following 4 bins: $\{\frac{1}{2}, \frac{1}{6}, \frac{1}{4}\}$, $\{\frac{1}{2}, \frac{1}{3}\}$, $\{\frac{3}{4}\}$ and $\{\frac{1}{2}\}$.

In this example FFD will pack the items into the following 3 bins: $\{\frac{3}{4}, \frac{1}{4}\}$, $\{\frac{1}{2}, \frac{1}{2}\}$, $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$. All bins are full and so FFD finds an optimal solution.

Give an example when FFD does not find an optimal solution (or see the lecture slides).

10.2.5 Online Problems and Algorithms

Often in practice, items are become available one by one and once they are available they must be packed irreversibly. In such cases we have the *online BPP*. Heuristics NF and FF can be used for the online BPP; FFD cannot. In fact, it is proved that no algorithm for the online BPP can satisfy Theorem 10.2.8.

There are many other online problems, for example the online AP and TSP. Algorithms for online problems are called online algorithms.

10.3 Domination Analysis

Domination analysis provides an alternative to approximation analysis. In domination analysis, we are interested in the number of solutions that are worse or equal in quality to the heuristic one, which is called the domination number of the heuristic solution. In many cases, domination analysis is very useful. In particular, some heuristics have domination number 1 for the TSP. In other words, those heuristics, in the worst case, produce the unique worst possible solution. At the same time, the approximation ratio is not bounded by any constant. In this case, the domination number provides a far better insight into the performance of the heuristics.

The *domination number* $\text{domn}(H, n)$ of a heuristic H for the TSP is the maximal number of tours that are more expensive or equal in cost to the tour produced by H, for every instance of the TSP on n vertices. For example, $\text{domn}(H, n) \geq (n-2)!$ means that H always produces a tour that is better or of the same cost as at least $(n-2)!$ tours for every instance on n vertices.

Theorem 10.3.1. *For the STSP with triangle inequality, DTH has domination number 1.*

Proof: Consider an instance of the DTH with vertices $v_1, v_2, \ldots, v_{n-1}, v_n$. Let the cost of the edges $v_2v_3, v_3v_4, \ldots, v_{n-1}v_n$ be 2 and the cost of the rest of the edges be 1. Notice that the tour $v_1v_2v_3v_4 \ldots v_{n-1}v_nv_1$ is the unique most expensive tour as it is the only tour that includes all edges of cost 2.

The tree S with edges v_1v_i, $i = 2, 3, ..., n$ is a minimum cost spanning tree. Suppose that DTH chooses S (DTH may choose any minimum cost spanning tree). After doubling edges of S, we get an Euler graph G;

$$F = v_1v_2v_1v_3v_1v_4v_1 \ldots v_1v_nv_1$$

is an Euler trail of G and suppose that DTH constructs F. After deleting repeated vertices, we get the tour $v_1v_2v_3v_4...v_{n-1}v_nv_1$, which is unique most expensive. QED

Recall that in the Assignment Problem, we are given a complete bipartite graph B with n vertices in each partite set and a non-negative cost assigned to each edge of B. We are required to find a perfect matching (i.e. matching with n edges) in B of minimum cost.

Theorem 10.3.2. *For the Assignment Problem, the greedy algorithm has domination number 1.*

Proof: Let B be a complete bipartite graph with n vertices in each partite set and let $u_1, u_2, ..., u_n$ and $v_1, v_2, ..., v_n$ be the two partite sets of B. Let M be any number greater than n. We assign cost Mi to the edge u_iv_i for $i = 1, 2, ..., n$ and cost $M \times \min\{i, j\} + 1$ to every edges u_iv_j, $i \neq j$; see Figure 10.7 for an illustration in the case $n = 3$.

We classify edges of B as follows: u_iv_j is horizontal (forward, backward) if $i = j$ ($i < j$, $i > j$). See Figure 10.7.

The greedy algorithm will choose edges $u_1v_1, u_2v_2, ..., u_nv_n$ (and in that order). We call this perfect matching P and we will prove that P is the unique most expensive perfect matching in B. The cost of P is $\text{cost}(P) = M + 2M + ... + nM$.

Choose an arbitrary perfect matching P' in B distinct from P. Assume that P' has edges $u_1v_{p_1}, u_2v_{p_2}, ..., u_nv_{p_n}$. By the definition of the costs in B, $\text{cost}(u_iv_{p_i}) \leq Mi + 1$.

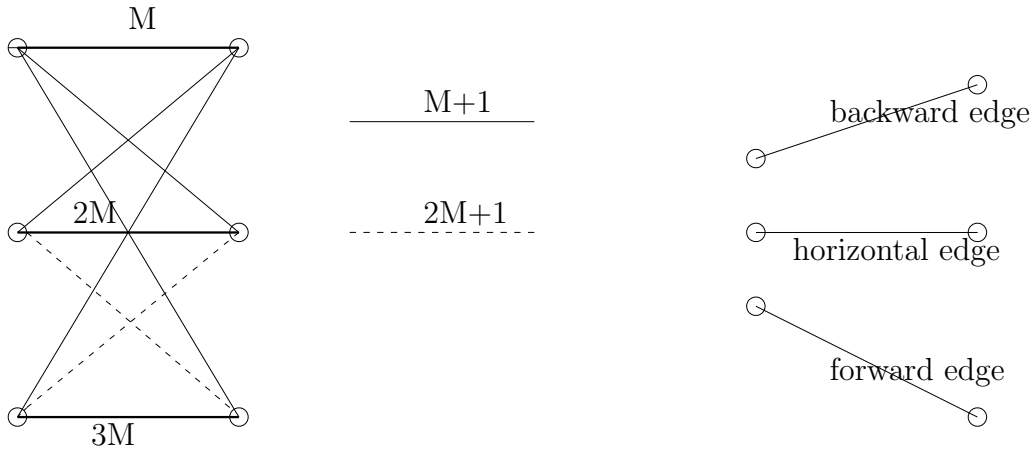


Figure 10.7: Assignment of costs for $n = 3$; classification of edges

Since P' is distinct from P, it must have edges that are not horizontal. This means it has backward edges. If $u_k v_{p_k}$ is a backward edge, i.e. $p_k < k$, then $\text{cost}(u_k v_{p_k}) \leq M(k-1)+1 = (Mk+1) - M$. Hence,

$$\text{cost}(P') \leq (M + 2M + ... + nM) + n - M = \text{cost}(P) + n - M.$$

Thus, $\text{cost}(P') < \text{cost}(P)$. QED

Notice that there are algorithms for the Assignment Problem with much larger domination number. The Assignment Problem can be solved to optimality by the $O(n^3)$-time Hungarian algorithm. The Hungarian algorithm is of domination number equal to the number of all perfect matchings in B, hence equal to $n!$

It is possible to prove that the greedy algorithm and NN for the TSP are of domination number 1. However, the vertex insertion heuristic is proved to be of domination number at least $(n-2)!$ for the Asymmetric TSP. Clearly, the vertex insertion heuristic appears to be more robust than the greedy algorithm and NN.

10.4 Questions

Question 10.4.1. *Find an optimal tour for the STSP with cost matrix*

$$(a)\quad C=\begin{pmatrix}0 & 2 & 4 & 3\\ 2 & 0 & 2.5 & 5\\ 4 & 2.5 & 0 & 3.5\\ 3 & 5 & 3.5 & 0\end{pmatrix}\qquad (b)\quad C=\begin{pmatrix}0 & 2 & 4 & 3\\ 2 & 0 & 1 & 4\\ 4 & 1 & 0 & 7\\ 3 & 4 & 7 & 0\end{pmatrix}$$

Find tours by DTH. Are the found tours optimal? Why?

Can we apply Theorem 10.2.2 to the instances above? Why?

Question 10.4.2. *Prove that for the Euclidean TSP, if a tour T is the best in its 2-Opt neighbourhood, then T does not have self-intersections.*

Question 10.4.3. *Describe the Double Tree Heuristic (DTH) for the Symmetric TSP. Illustrate DTH using an instance of the STSP with 5 vertices (given by the cost matrix).*

Question 10.4.4. *Prove that, for the STSP with triangle inequality, the Double Tree Heuristic always produces a tour of cost at most twice the cost of the optimal tour.*

Question 10.4.5. *Define the domination number of a heuristic. What does it mean that a heuristic has domination number 1 for the problem in hand ?*

Question 10.4.6. *Prove that for the Assignment Problem, the greedy algorithm has domination number 1.*

Question 10.4.7. *(For interested students) Prove that NN is of domination number 1 for the Asymmetric TSP. Hint: Consider an instance of the ATSP with vertices 1,2,...,n and arc costs given as follows:* $\text{cost}(i,i+1)=in$ *for* $1\le i\le n-1$, $\text{cost}(n,1)=n^3$ *and* $\text{cost}(i,j)=n\times\min\{i,j\}+1$ *for any arc* (i,j) *whose cost is not defined earlier. Prove that NN will choose the tour* $(1,2,3,...,n,1)$ *and this tour is unique most expensive. See Figure 10.8.*

Question 10.4.8. *Let* $b=1$ *and*

$$L=(\frac{1}{2},\frac{1}{8},\frac{1}{2},\frac{1}{8},\frac{1}{2},\frac{1}{8},\frac{1}{2},\frac{1}{8},\frac{1}{2},\frac{1}{8},\frac{1}{2},\frac{1}{8},\frac{1}{2},\frac{1}{8},\frac{1}{2},\frac{1}{8}).$$

What is the minimum number of bins required and what is the number of items found by Next Fit? Justify your answers.

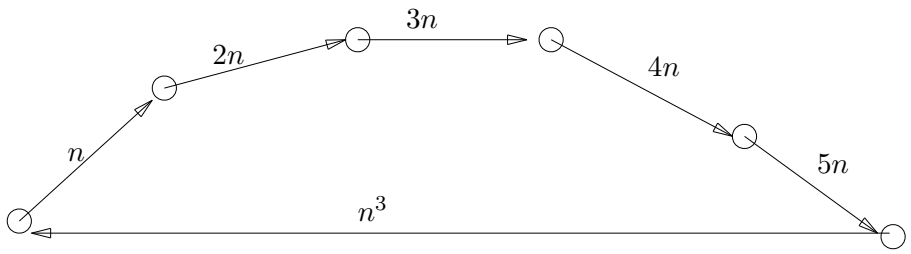


Figure 10.8: NN is of domination number 1 for the ATSP

10.5 Solutions

Question 10.4.8

Since a pair 1/2, 1/2 completely fill in a bin and eight 1/8 also fill in a bin, the optimal number of bins is 5 (no wasted space at all). NF fills in 8 bins.

Chapter 11

Advanced Local Search and Meta-heuristics

11.1 Advanced Local Search Techniques

Most of improvement local search algorithms have the advantage of being generally applicable and flexible, i.e., they require only a specification of solutions, a cost function, a neighbourhood function, and an efficient method of exploring a neighbourhood, all of which can be readily obtained for most optimisation problems. Nevertheless, poor local optima are found in several cases. To remedy this drawback - while maintaining the paradigm of neighbourhood search - many researchers have been investigating the possibilities of broadening the scope of local search to obtain neighbourhood search algorithms that can find high-quality solutions in possibly low running times. A straightforward extension of local search would be to run a simple local search algorithm a number of times using different start solutions and to keep the best solution found as the final solution. Several researchers have investigated this *multistart* approach, but no major successes have been reported.

Instead of restarting from an arbitrary solution, one could consider an approach that applies multiple runs of a local search algorithm by combining several neighbourhoods, i.e., by restarting the search in one neighbourhood from a solution found in another one. Such approaches are called *multilevel*. An example is *iterated local search*, in which the start solutions of subsequent local searches are obtained by modifying the local optima of a previous run. Examples for the Symmetric TSP are iterated Lin-Kernighan algorithms investigated by some researchers, which are currently the best known heuristics for the Symmetric TSP.

In addition there are several approaches that take advantage of search strategies in which cost-deteriorating neighbours are accepted. We briefly describe these strategies in

the following section.

11.2 Meta-heuristics

All of the approaches below are together called meta-heuristics as they are very general and can be applied to a vast number of different optimisation problems. The approaches model certain kinds of optimisation in Nature.

11.2.1 Simulated Annealing

Simulated annealing (SA) is based on analogy with the physical process of annealing, in which a pure lattice structure of a solid is made by heating up the solid in a heat bath until it melts, then cooling it down slowly until it solidifies into a low-energy state. From the point of view of combinatorial optimisation, simulated annealing is a randomized neighbourhood search algorithm. In addition to better cost neighbours, which are always accepted if they are selected, worse-case neighbours are also accepted, although with a probability that is gradually decreased in the course of the algorithm's execution. Lowering the *acceptance probability* is controlled by a set of parameters whose values are determined by a *cooling schedule*.

SA has been widely used with considerable success. The randomized nature enables asymptotic convergence to optimal solutions under certain mild conditions. Unfortunately, the convergence typically requires exponential time, making simulated annealing impractical as a means of obtaining optimal solutions. Instead, like most local search algorithms, it is normally used as an approximation algorithm, for which much faster convergence rates are acceptable.

11.2.2 Genetic Algorithms

Genetic algorithms use concepts from population genetics and evolution theory to construct algorithms that try to optimise the *fitness* of a *population* of elements through *recombination* and *mutation* of their genes. There are many variations known in the literature of algorithms that follow these concepts. As an example, we discuss *genetic local search*, also called the Memetic Algorithm. The general idea of genetic local search is explained below.

- Step 1, *initialize*. Construct an initial population of n solutions.
- Step 2, *improve*. Use local search to replace the n solutions in the population by n local optima.

i	String x_i	Fitness $f_i = f(x_i)$	Selection probability $f_i / \sum f_i$
1	11000	16	0.242
2	00101	6	0.091
3	10110	36	0.545
4	00100	8	0.122
	Sum	66	1.000
	Average	16.5	0.250
	Max	36	0.545

Table 11.1: Four strings and their fitness values.

- Step 3, *recombine*. Select pairs of individuals from the population to participate in recombination. Augment the population by adding the m offspring solutions; the population size now equals $n + m$.
- Step 4, *improve*. Use local search to replace the m offspring solutions by m local optima.
- Step 5, *select*. Reduce the population to its original size by selecting n solutions from the current population.
- Step 6, *evolute*. Repeat Steps 3-5 until a stop criterion is satisfied.

Eventually, *recombination* is an important step, since here one must try to take advantage of the fact that more than one local optimum is available, i.e., one must exploit the structure present in the available local optima.

The above scheme has been applied to several problems, and good results have been reported. The general class of genetic algorithms contains many other approaches that are significantly different from the scheme above.

To illustrate the main ideas, we will consider an example (due to Stephen D. Scott) of a genetic local search applied to a simple optimization problem. To make the example even simpler, we will skip Steps 2 and 4 of the general description, which involve the local search part of the heuristics.

Consider strings with five bits 0 or 1, $x = x^5x^4x^3x^2x^1$, where $x^j = 0$ or 1, for $j = 1, 2, 3, 4, 5$, and an objective function $f(x) = 32x^5 - 16x^4 + 8x^3 - 4x^2 + 2x^1$ (the j's stand for superscripts, not powers). For example $x = 10011$ is such a string having objective function value $f(x) = 32 - 4 + 2 = 30$. The goal is to maximize the objective function $f(x)$ over all such strings. Now imagine a population of the four strings in Table 11.1, generated at random. The fitness values come from the objective function $f(x)$.

After reprod.	Parents	Parent strings	Crossover point	After crossover	Fitness $f_i = f(x_i)$
x_5	x_1	11|000	2	11110	20
x_6	x_3	10|110	2	10000	32
x_7	x_3	1|0110	1	10100	40
x_8	x_4	0|0100	1	00110	4
Sum					96
Average					24
Max					40

Table 11.2: The next generation after selection and crossover.

The values in the "$f_i / \sum f_i$" column provide the probability of each string's selection. So initially 11000 has a 24.2% chance of selection, 00101 has an 9.1% chance, and so on. Based on these probabilities, we randomly select two pairs of strings to be recombined; suppose that the random selection produces the pairs (x_1, x_3) and (x_3, x_4).

After selecting the pairs, the genetic algorithm looks at each pair individually. For each pair (e.g. $x_1 = A = 11000$ and $x_3 = B = 10110$), the algorithm decides whether or not to perform crossover recombination. If it does not, then both strings in the pair are placed into the population with possible mutations (described below). If it does, then a random crossover point is selected and crossover proceeds as follows: $A = 11|000$ $B = 10|110$ are crossed and become $A' = 11110$ $B' = 10000$.

Then the children A' and B' are placed in the population with possible mutations. The genetic algorithm invokes the mutation operator on the new strings very rarely (usually on the order of < 0.01 probability), generating a random number for each bit and flipping that particular bit only if the random number is less than or equal to the mutation probability.

After the current generation's selections, crossovers, and mutations are complete, the new strings representing the next generation is shown in Table 11.2. In this example, average fitness increased from one generation to the next by approximately 45% and maximum fitness increased by 11%.

Reducing the population to its original size 4, we select the best solutions x_1, x_3, x_5, x_7 from the current population, see Table 11.3. This finishes one iteration of the algorithm. The simple process would continue for several generations until a stopping criterion is met.

Question 11.2.1. *Give a brief description of genetic local search.*

i	String x_i	Fitness $f_i = f(x_i)$	Selection probability $f_i / \sum f_i$
3	10110	36	0.281
5	11110	20	0.156
6	10000	32	0.250
7	10100	40	0.313
	Sum	128	1.000
	Average	32.0	0.250
	Max	40	0.313

Table 11.3: The population after one iteration.

11.2.3 Modern Genetic Local Search Algorithms

A sophisticated state-of-the-art genetic local search algorithm for the Generalized TSP is given in the paper G. Gutin and D. Karapetyan, A Memetic Algorithm for the Generalized Traveling Salesman Problem, *Natural Computing* 9 (2010), 47–60. This paper refers to several other genetic local search algorithms for the Generalized TSP.

11.2.4 Tabu Search

Tabu search "models" intelligent human argumentation while searching for a good solution.

Tabu search combines the deterministic iterative improvement algorithm with a possibility to accept cost-increasing solutions. In this way the search is directed away from local minima[1], such that other parts of the search space can be explored. The next solution visited is always chosen to be a *legal neighbour* of the current solution with the best cost, even if that cost is worse than that of the current solution. The set of legal neighbours is restricted by a *tabu list* designed to prevent us from going back to recently visited solution. The tabu list is dynamically updated during the execution of the algorithm. The tabu search list defines solutions that are not acceptable in the next few iterations. However, a solution on the tabu list may be accepted if its quality is in some sense good enough, in which case it is said to attain a certain *aspiration level*.

Tabu search has also been applied to large variety of problems with considerable successes. Tabu search is a general search scheme that must be tailored to the details of the problem at hand. Unfortunately, there is little theoretical knowledge that guides this tailoring process, and users have to resort to the available practical experience.

We will now consider a basic algorithm of tabu search, where only so-called short term memory is used and no aspiration criteria are of use. We will describe the basic algorithm

[1]We are speaking of minimisation problems; the similar approach applies to maximisation problems.

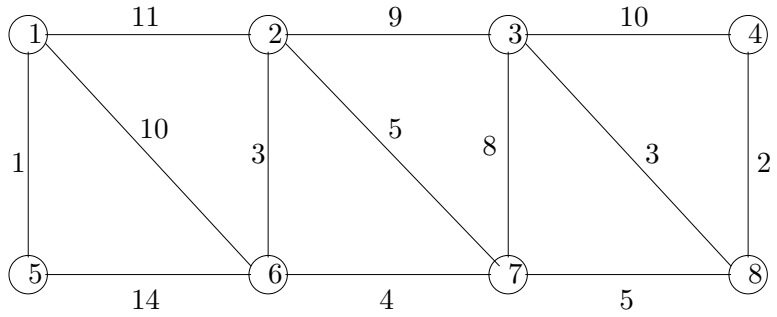


Figure 11.1: Graph G.

applied to the so-called *k-tree problem*. In this problem, we are given a positive integer k and a graph G with costs on it edges. We are required to compute a minimum cost subgraph of G, which is a tree with exactly k edges.

Consider the graph G depicted in Figure 11.1. We will try to solve the 3-tree problem for G.

We start from an initial feasible solution T_0 consisting of the edges $\{1,2\}$, $\{1,5\}$ and $\{1,6\}$. The cost of T_0 is 22, see Figure 11.2 (a). No edge of T_0 is *tabu*.

Now we will move to another solution by deleting a non-tabu edge in T_0 and adding to the remaining graph another non-tabu edge such that the result is a tree. This way, we have a so-called *swap neighbourhood*. We will make such a move that brings us the cheapest possible solution. It is easy to see that we will obtain T_1 with edges $\{1,5\},\{1,6\},\{2,6\}$ and of cost 14. See Figure 11.2 (b). We have deleted $\{1,2\}$ and make it tabu for 2 iterations. We cannot add $\{1,2\}$ back during the next two iterations. In other words, the *tabu time* for deleted edges is fixed to 2. We have added $\{2,6\}$ and make it tabu for 1 iteration. We cannot delete $\{2,6\}$ back during the next iteration. In other words, the *tabu time* for added edges is fixed to 1. See Table 11.2.4.

Observe that no swap of an edge in T_1 can produce a solution cheaper than T_1. In other words, the improvement local search is the swap neighbourhood of T_1 will not be able to find a better solution, i.e., T_1 is local minimum and the improvement local search would be stuck in it.

Since edge $\{1,2\}$ is tabu, our next move gives us T_2 consisting of edges $\{1,6\},\{2,6\},\{6,7\}$ of cost 17. (The increase of cost is not surprising in the light of the previous paragraph.) For the iteration 2 tabu times, see Table 11.2.4.

In the next iteration, we can delete $\{1,6\}$ and add $\{7,8\}$. We get a solution of cost 12, see Figure 11.2 (d). The results of the last two iterations are depicted in Figure 11.2 (e) [the panultimate iteration in Fig. 11.2 contains a mistake; find it!] and (f). For the

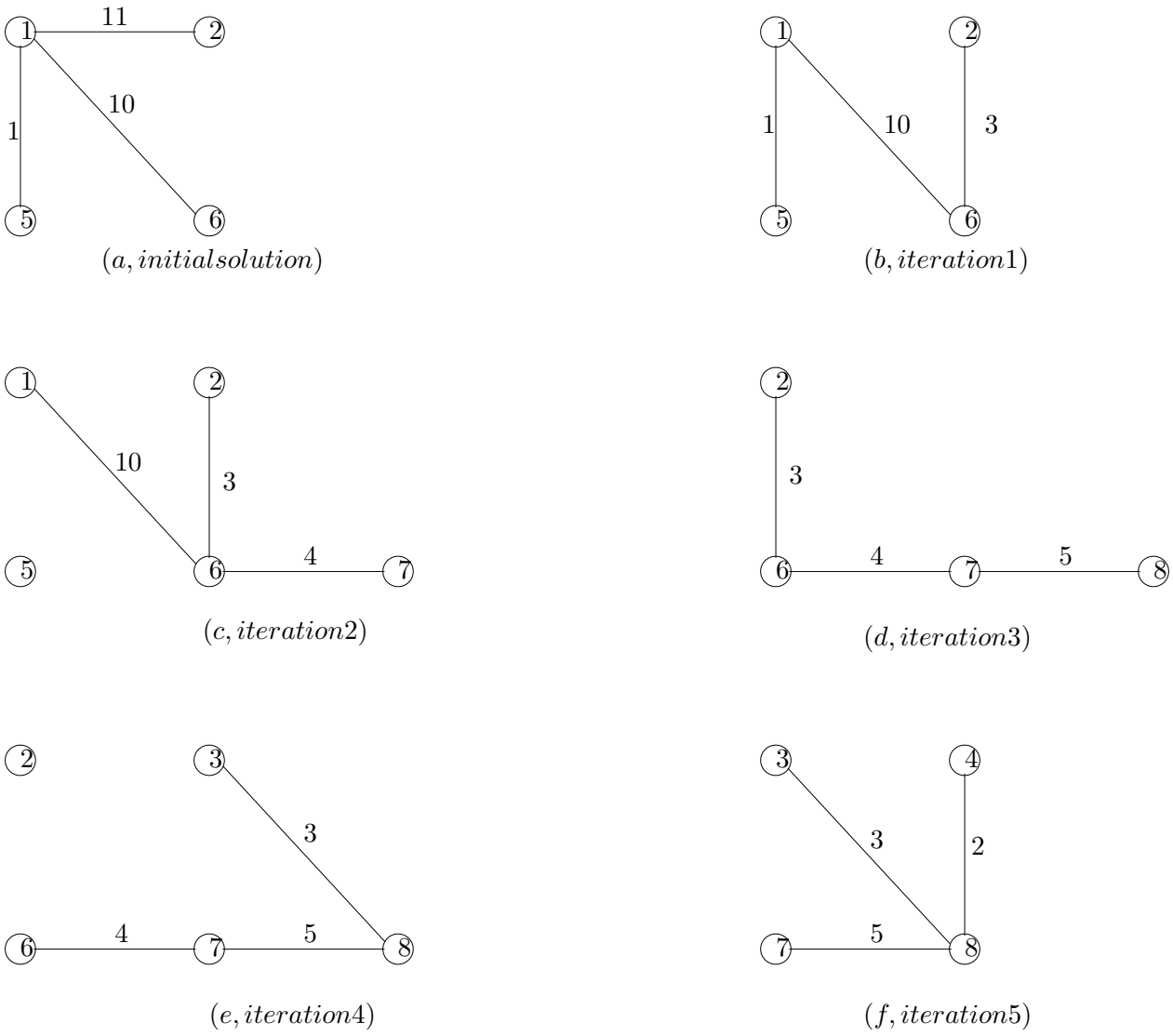


Figure 11.2: Iterations of the basic tabu search algorithm.

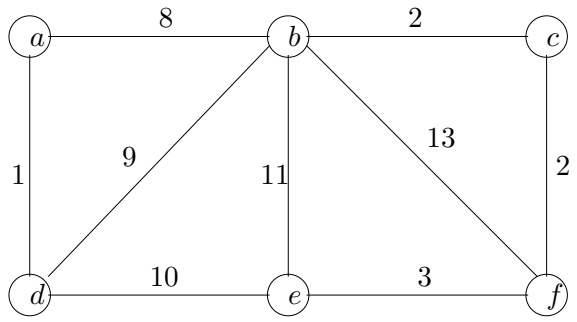


Figure 11.3: Graph H.

iterations tabu times, see Table 11.2.4. Interestingly, the last solution is optimal.

iteration	1	2	3	4	5
TT 2	$\{1,2\}$	$\{1,5\}$	$\{1,6\}$	$\{2,6\}$	$\{6,7\}$
TT 1	$\{2,6\}$	$\{1,2\}$, $\{6,7\}$	$\{1,5\}$, $\{7,8\}$	$\{1,6\}$, $\{3,8\}$	$\{2,6\}$, $\{4,8\}$
tree edges	$\{1,5\},\{1,6\},\{2,6\}$	$\{1,6\},\{2,6\},\{6,7\}$	$\{2,6\},\{6,7\},\{7,8\}$	$\{6,7\},\{7,8\},\{4,8\}$	$\{7,8\},\{3,8\},\{4,8\}$

Table 11.4: Tabu Times (TT) and tree edges for iterations 1-5.

One can modify the basic tabu search algorithm by adding *aspiration* criteria. One of the most used aspiration criteria is the *record-improving* one. Here we may add or delete tabu edges as long as we get a solution of cost lower than the best known one.

Question 11.2.2. *Consider the example for the 3-tree problem above. Replace the basic tabu search by that with the record-improving aspiration criterium. Will any solution be changed?*

In the *k-path problem*, we are given a positive integer k and a graph G with costs on it edges. We are required to compute a minimum cost subgraph of G, which is a path with exactly k edges.

Question 11.2.3. *Consider the 2-path problem for the graph G in Figure 11.1. Let the initial solution consist of the edges $\{1,5\}$ and $\{1,6\}$. Starting from the initial solution, carry out five iterations of the basic tabu search algorithm to improve the initial solution. The tabu times for deleted and added edges are 2 and 1, respectively.*

Question 11.2.4. *Consider the 2-path problem for the graph H in Figure 11.3. Let the initial solution be the path dab. Starting from the initial solution, carry out five iterations of the basic tabu search algorithm to improve the initial solution. The tabu times for deleted and added edges are 2 and 1, respectively.*

Some Solutions

Solution to Question 11.2.3

iteration	1	2	3	4	5
TT 2	af	ae	ab	bf	fg
TT 1	ab	af, ef	ae, fg	ab, gh	bf, hd
2-path	eab	abf	bfg	fgh	ghd

Table 11.5: Tabu Times (TT) and 2-paths for Question 11.2.3, where for vertex $a = 1$, $b = 2$, $c = 3$, $d = 4$, $e = 5$, $f = 6$, $g = 7$, $h = 8$.

Solution to Question 11.2.4

iteration	1	2	3	4	5
TT 2	ad	ab	bc	cf	fe
TT 1	bc	ad, cf	ab, fe	bc, ed	cf, ad
2-path	abc	bcf	cfe	fed	eda

Table 11.6: Tabu Times (TT) and 2-paths for Question 11.2.4.