Contents

Chapter 9

Heuristic Methods for Combinatorial Optimization Problems

This is Chapter 9 of "Junior Level Web-Book for Optimization Models for decision Making" by Katta G. Murty.

9.1 What Are Heuristic Methods?

The word **heuristic** comes from the Old Greek word **heuriskein** which means "discovering new methods for solving problems" or "the art of problem solving." In computer science and artificial intelligence, the term "heuristic" is applied usually to methods for intelligent search. In this sense "heuristic search" uses all the available information and knowledge to lead to a solution along the most promising path, omitting the least promising ones. Here its aim is to enable the search process to avoid examining dead ends, based on information contained in the data gathered already.

However, in operations research the term "heuristic" is often applied to methods (which may or may not involve search) that are based on intuitive and plausible arguments likely to lead to reasonable solutions but are not guaranteed to do so. They are methods for the problem under study, based on rules of thumb, common sense, or adaptations of exact methods for simpler models. They are methods used to find reasonable solutions to problems that are hard to solve exactly. In optimization in particular, a heuristic method refers to a practical and quick method based on strategies that are likely to (but not guaranteed to) lead to a solution that is approximately optimal or near optimal. Usually they provide robust approaches to obtain high-quality solutions to problems of a realistic size in reasonable time. So, while discussing these heuristic methods, the verb "solve" has the connotation of "finding a satisfactory approximation to the optimum." Thus heuristic methods can, but do not guarantee the finding of an optimum solution; although good heuristic methods in principle determine the best solution obtainable within the allowed time. Many heuristic methods do involve some type of search to look for a good approximate solution.

9.2 Why Use Heuristic Methods?

Heuristic methods are as old as decision making itself. Until the 1950s when computers became available and machine computation became possible, inelegant but effective heuristics were the only methods used to tackle large scale decision making.

Exact Algorithms for Linear, Convex Quadratic, and Convex Nonlinear Programming Problems

By an **exact algorithm** for an optimization problem, we mean an algorithm that is guaranteed to find an optimum solution if one exists, within a reasonable time.

In the 1960s and 70s exact algorithms based on sophisticated mathematical constructs were developed for certain types of optimization problems such as linear programs, convex quadratic programs, and nonlinear convex programming problems. The special distinguishing feature of all these problems is that optimality conditions providing efficient characterizations for optimum solutions for them are known. The exact algorithms for them are based on these optimality conditions. Because of this special feature, these problems are considered to be **nice problems** among optimization models. The development of exact algorithms for these nice problems has been a significant research achievement for optimization theory.

By the 1980s, software packages implementing these sophisticated algorithms, and computer systems that can execute them, became very widely available. So, now-a-days there is no reason to resort to heuristic methods to solve instances of these nice problems, as they can be solved very efficiently by these exact algorithms.

Status of Algorithms for Discrete, Integer, Combinatorial, and Nonconvex Nonlinear Programs; and the Need to Use Heuristic Approaches

Unfortunately, research though extensive, did not lead to any reliable exact solution method for other optimization problems such as the discrete and integer programming problems and combinatorial optimization problems discussed in Chapters 7 and 8 that are not in the nice class. The B&B approach of Chapter 8 based on partial enumeration can solve instances of moderate sizes of these problems, but in general the time requirement of this approach grows exponentially with the size of the instance. Real world applications of combinatorial optimization usually lead to large scale models. We illustrate this with an application in the automobile industry.

Example 9.2.1: A task allocation problem

This problem, posed by K. N. Rao, deals with determining a minimum cost design for an automobile's microcomputer architecture. In the modern automobile, many tasks such as integrated chassis and active suspension monitoring, etc. are performed by microcomputers linked by high speed and/or slow speed communication lines. The system's cost is the sum of the costs of the processors (microcomputers), and of the data links that provide inter-processor communication bandwidth.

Each task deals with the processing of data coming from sensors, actuators, signal processors, digital filters, etc., and has a throughput requirement in KOP (kilo operations per second). Several types of processors are available. For each, we are given its cost, maximum number of tasks it can handle, and its throughput capacity in terms of the KOP it can handle.

The tasks are inter-dependent. To complete a task we may need data from another. So, the typical communication pattern between tasks is that if two tasks are assigned to different processors, they need communication link capacity (in bits/second) between them. Tasks executing in the same processor do not have communication overhead. Here is the notation for the data.

n	$=$	number of tasks to be performed (varies between 50 - 100 in applications)
a_i	$=$	throughput requirement (in KOP) of task i, $i = 1$ to n
T	$=$	maximum number of processors that may be needed
$\rho_t, \gamma_t, \beta_t$	$=$	cost (\$), capacity in KOP, and upper bound on the number of tasks to be allotted, to processor t, $t = 1$ to T
c_{ij}, d_{ij}	$=$	low speed and high speed communication link capacity (in bits/second) needed for task pair i, j if they are assigned to different processors
L, H	$=$	unit cost of installing low speed and high speed communication bandwidth

To model this problem, we define the following decision variables for $i, j = 1$ to n and $t = 1$ to T.

$$\text{For } i \neq j, x_{ijt} = \begin{cases} 1, & \text{if both tasks } i \text{ and } j \text{ are assigned to processor } t \\ 0, & \text{otherwise} \end{cases}$$

$$x_{iit} = \begin{cases} 1, & \text{if task } i \text{ is assigned to processor } t \\ 0, & \text{otherwise} \end{cases}$$

$$y_t = \begin{cases} 1, & \text{if processor } t \text{ is used (i.e., it is allotted some tasks)} \\ 0, & \text{otherwise} \end{cases}$$

In terms of these decision variables the model for the minimum cost design is

$$\begin{array}{rrcl}
\text{Min.} & \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (Lc_{ij} + Hd_{ij})(1 - \sum_{t=1}^{T} x_{ijt}) & + & \sum_{t=1}^{T} \rho_t y_t \\
\text{s. to} & \sum (x_{ijt} : \text{over } j \neq i) - (\beta_t - 1)x_{iit} & \leq & 0, \quad \begin{array}{r}\text{for } i = 1 \text{ to } n \\ t = 1 \text{ to } T\end{array} \\
 & \sum_{i=1}^{n} a_i x_{iit} & \leq & \gamma_t y_t, \quad \text{for } t = 1 \text{ to } T \\
 & \sum_{i=1}^{n} x_{iit} & \leq & \beta_t y_t, \quad \text{for } t = 1 \text{ to } T \\
 & \sum_{i=1}^{n} x_{iit} & \geq & y_t, \quad \text{for } t = 1 \text{ to } T \\
 & \sum_{t=1}^{T} x_{iit} & = & 1, \quad \text{for } i = 1 \text{ to } n \\
 & x_{ijt}, y_t \text{ are all} & & 0 \text{ or } 1
\end{array}$$

The first constraint guarantees that the processor t to which task i is assigned, is not assigned more than $\beta_t - 1$ other tasks. The second constraint guarantees that the total KOP requirements of all the tasks assigned to processor t is $\leq$ its KOP capacity of γ_t. The third and fourth constraints together guarantee that processor t is either not used, or if it is used then it is assigned no more than β_t tasks. The fifth constraint guarantees that each task is assigned to a processor.

This is a 0−1 IP model with $T(n^2 + 1)$ integer variables. Even for $n = 50$, and $T = 10$, the number of 0−1 variables in the model is over 25,000, which is very large. $\bowtie$

In the same manner, problems in the optimum design of many manufactured items, in telecommunication system design, and other areas, lead to large scale combinatorial optimization models.

Research in computational complexity and NP-completeness theory since the 1970s has shown that many of the integer programming and combinatorial optimization problems discussed in Chapters 7 and 8 are hard intractable problems. It has provided evidence that there may be no effective exact algorithms to solve large scale versions of these problems, i.e., algorithms which can find optimum solutions to these problems within acceptable computer time.

As a consequence, it has been recognized that the only practical alternative to attack large scale instances of these problems is through good heuristic methods. In fact, practitioners facing these problems have always had an interest in heuristics as a means of finding good approximate solutions. And experience indicates that there are many heuristic methods which are simple to implement relative to the complexity of the problem, and although they do not always necessarily yield a solution close to the optimum, they quite often do. Moreover, at the termination of a heuristic method, we can always improve performance by resorting to another heuristic search algorithm to resume the search for a better solution.

For some of the hard combinatorial optimization problems such as the TSP, a detailed study based on their mathematical structure has made it possible to construct special bounding schemes. B&B algorithms based on them have successfully solved several large scale instances of these problems within reasonable times. There is no guarantee that these special algorithms will give the same effective performance on all large scale instances of these problems, but their record so far is very impressive. However, many practitioners still seem to prefer to solve these problems approximately using much simpler heuristic methods. One reason for this is the fact that real world applications are often messy, and the data available for them is liable to contain unknown errors. Because of these errors in the data, an optimum solution of the model is at best a guide to a reasonable solution for the real problem, and an approximate solution obtained by a good but simple heuristic would serve the same purpose without the need for expensive computer hardware and software for a highly sophisticated algorithm.

For all these reasons, heuristic methods are the methods of choice for handling large scale combinatorial optimization problems.

9.3 General Principles in Designing Heuristic Methods

The literature classifies heuristic algorithms into two broad classes: **constructive heuristic algorithms** (these methods develop a solution to the problem element by element, or part by part, and they terminate when a complete solution is constructed); and **iterative improvement heuristic algorithms** (these methods start with some initial solution, and search for ways of changing it into a better solution).

Heuristic methods are always problem-specific, but there are several widely applicable principles for designing them.

The Greedy Principle

A popular principle for developing constructive heuristics is the **greedy principle** which leads to greedy methods, perhaps the most important constructive methods among **single pass heuristics** that create a solution in a single sweep through the data. Each successive step in these methods is taken so as to minimize the immediate cost (or maximize the immediate gain). The characteristic features of greedy methods are the following.

The incremental feature They represent the problem in such a way that a solution can be viewed either as a subset of a set of elements, or as a sequencing of a set of elements in some order. The approach builds up the solution set, or the solution sequence, one element at a time starting from scratch, and terminates with the first complete solution, see Section 9.4 for examples.

The no-backtracking feature Once an element is selected for inclusion in the solution set (or an element is included in the sequence in the current position) it is never taken back or replaced by some other element (or its position in the sequence is never altered again). That is, in a greedy algorithm, decisions made at some stage in the algorithm are never revised later on.

The greedy selection feature Each additional element selected for inclusion in the solution set, or selected to fill the next position in the sequence, is the best among those available for selection at that stage by some criterion, in the sense that it contributes at that stage the least amount to the total cost, or the maximum amount to the total gain, when viewed through that criterion.

The myopic feature When selecting an item for inclusion in the solution set at some stage, or selected to fill the next position in the sequence, usually, only the contribution to the overall cost of that inclusion at that stage is considered; and not the consequences of that inclusion in later stages.

Several different criteria could be used to characterize the "best" when making the greedy selection, depending on the nature of the problem being solved. The success of the approach depends critically on the choice of this criterion.

Thus the greedy approach constructs the solution stepwise. In each step it selects the element to include in the solution to be the cheapest among those that are eligible for inclusion at that time. It is very naive. The selection at each stage is based on the situation at that time, without any features of look-ahead, etc. Hence greedy methods are also known as **myopic methods**.

Neighborhood Search Process

Another approach for designing heuristic methods is based on starting with a complete solution to the problem, and trying to improve it by a local search in the neighborhood of that solution by an iterative improvement process. The initial solution may be either a randomly generated solution, or one obtained by another method like the greedy method. Each subsequent step in the method takes the solution at the end of the previous step, and tries to improve it by either exchanging a small number of elements in the solution with those not in the solution, or some other technique of local search. The process continues until no improving solution can be found by such local search, at which point we have a local minimum. These methods are variously known as **interchange heuristic methods** or **local search heuristics** or **descent methods**. A local optimum is at least as good as or better than all solutions in its neighborhood, but it may not be a global optimum, i.e., it may not be the best solution for the problem. One of the shortcomings of a descent method is the fact that it obtains a local minimum which in most cases may not be a global minimum. To overcome this limitation people normally apply the descent method many times, with different initial solutions, and take as the final output the best among all the local minima obtained. This restart approach is known as **the iterated descent method**.

The general design principle of local improvement through small changes in the feasible solution is also the principle behind the **simulated annealing**, and **tabu search techniques**, which also admit steps that decrease solution quality based on a probabilistic scheme. After reaching a local optimum, these techniques move randomly for a period and then resume a trajectory of descent again.

And then there are heuristic methods known as **genetic algorithms** which are probabilistic methods that start with an initial population of likely problem solutions and then evolve towards better solution versions. In these methods new solutions are generated through the use of genetic operators patterned upon the reproductive processes in nature.

Sometimes several heuristic methods may be applied on a problem in a sequence. If the first heuristic starts from scratch to find an initial solution, the second may have the aim of improving it. And when this heuristic comes to its end, a third may succeed it. This could continue until all the heuristics in the list fail in a row to improve the current solution. This type of strategy of using a combination of different heuristic methods leads to a **metaheuristic method**.

There are major differences between the techniques appropriate to different problems. As in the B&B approach, details of a heuristic algorithm depend on the structure of the problem being solved. In the following sections we discuss the essential ideas behind the popular heuristic methods, and illustrate their application on several problem types discussed in Chapter 7.

9.4 The Greedy Approach

We will now discuss greedy methods for various problems from Chapter 7.

9.4.1 A Greedy Method for the 0−1 Knapsack Problem

Consider the 0−1 knapsack problem in which there are n objects that could be loaded into a knapsack of weight capacity w_0 weight units. For $j = 1$ to n, v_j in money units is the value, and w_j in weight units is the weight, of object j. Only one copy of each object is available to be loaded into the knapsack. None of the objects can be broken; i.e., each object should be either loaded whole into the knapsack, or should be left out. The problem is to decide the subset of objects to be loaded into the knapsack so as to maximize the total value of the objects included, subject to the weight capacity of the knapsack. So, defining for $j = 1$ to n

$$x_j = \begin{cases} 1, & \text{if } j\text{th article is packed into the knapsack} \\ 0, & \text{otherwise} \end{cases}$$

the problem is

$$\begin{aligned} \text{Maximize} \quad z(x) = \sum_{j=1}^{n} v_j x_j & & \\ \text{subject to} \quad \sum_{j=1}^{n} w_j x_j & \leq w_0 & \quad (9.4.1) \\ 0 \leq x_j \leq 1 \quad & \text{for all} \quad j & \\ x_j \quad \text{integer} \quad & \text{for all } j & \quad (9.4.2) \end{aligned}$$

To apply the greedy approach on this problem, the criterion to be greedy upon for selecting objects to include in the knapsack, could be either the value of the object, or its density = value/weight. Once this criterion is decided, the objects are arranged in decreasing order of the criterion and loaded into the knapsack in this order. At some stage, if an object's weight is $>$ remaining knapsack's weight capacity, we leave it out and continue the process with the next object in this order, until all the objects are examined. The set of objects loaded into the knapsack at the end of this process is the solution set determined by the greedy algorithm with the selected criterion.

As an example, consider the $0-1$ knapsack problem with knapsack's weight capacity of 35 weight units, and 9 different objects available for loading into it, discussed in Example 8.4.2, with the following data.

Object j	Weight w_j	Value v_j	Density $d_j = v_j/w_j$
1	3	21	7
2	4	24	6
3	3	12	4
4	21	168	8
5	15	135	9
6	13	26	2
7	16	192	12
8	20	200	10
9	40	800	20

The solution set obtained by the greedy algorithm with object's value as the criterion to be greedy upon is {objects 8, 5} using up the knapsack's weight capacity completely, and attaining the value of 335 money units for the total value of objects loaded into the knapsack.

The solution set obtained by the greedy method with density as the criterion to be greedy upon is {objects 7, 5, 1} with a total value of 348 money units.

The optimum objective value in this problem, found by the B&B algorithm in Example 8.4.2 is 351 money units. So, neither of the solution sets obtained by the greedy algorithms above are optimal. However, the greedy algorithm with the density criterion yielded a much better solution than the one with the object's value criterion.

What is the Best Criterion to be Greedy upon?

In general, the greedy algorithm with object's density as the criterion to be greedy upon yields much better solutions than the one with the object's value as the criterion. Thus, for the 0−1 knapsack problem, the greedy algorithm is always implemented with object's density as the criterion to be greedy upon. And "the greedy solution for this problem" usually means the solution obtained by this version of the greedy algorithm.

For this algorithm, the following result has been proved.

Theorem 9.4.1: *Consider the 0−1 knapsack problem (9.4.1), (9.4.2) with w_0 = knapsack's weight capacity, n = number of available objects; and $w_j, v_j, d_j = v_j/w_j$, as the weight, value, and density of object j, for j = 1 to n. Eliminate all objects j with $w_j > w_0$ since they won't fit into the knapsack (i.e., fix $x_j = 0$ for all such j). So, assume $w_j \leq w_0$ for all j = 1 to n. Let $\hat{x} = (\hat{x}_j)$ be the solution obtained by the greedy algorithm with object's density as the criterion to be greedy upon (i.e., $\hat{x}_j = 1$ if object j is included in the knapsack by this algorithm, $\hat{x}_j$ = 0 otherwise), and $\hat{v} = \sum_{j=1}^{n} v_j x_j$, $\hat{w} = \sum_{j=1}^{n} w_j x_j$.*

(i) The greedy solution $\hat{x}$ is an optimum solution for the original problem (9.4.1), (9.4.2), if the following conditions hold:

- *$\hat{w} = w_0$ (i.e., the greedy solution uses up the knapsack's weight capacity exactly),*
- *and all the objects j left out of the greedy solution set (i.e., with $\hat{x}_j = 0$) have density $d_j \leq$ the density of every one of the objects in the greedy solution.*

(ii) Let v_* denote the unknown optimum objective value in (9.4.1), (9.4.2). If the conditions in (i) are not satisfied $\hat{x}$ may not be optimal to (9.1), (9.2), but $v_* - \hat{v} \leq \max\{v_1, \ldots, v_n\}$.

For a proof of Theorem 9.4.1, see [G. L. Nemhauser and L. A. Wolsey, 1988]. It gives an upper bound for the difference between the optimum objective value and the objective value of the greedy solution.

In practice the greedy heuristic with density as the criterion to be greedy upon, usually yields solutions close to the optimum, and hence is very widely used for tackling 0−1 knapsack problems.

Exercises

9.4.1: Consider the 0−1 knapsack problem with $w_0 = 16 =$ knapsack's weight capacity, and 4 objects with data given below, available to load into the knapsack.

Object j	Weight w_j	Value v_j
1	2	16
2	15	105
3	1	6
4	13	13

Find the optimum solution of this problem by total enumeration. Apply the greedy heuristic with density as the criterion to be greedy upon and obtain the greedy solution for the problem. Verify that the greedy solution uses up the knapsack's weight capacity exactly, but that it is not optimal because the second condition in (i) of Theorem 9.4.1 does not hold.

9.4.2 A Greedy Heuristic for the Set Covering Problem

The set covering problem discussed in Section 7.3 is a pure 0−1 IP of the following form:

$$\begin{array}{rrcl} \text{Minimize} & z(x) = cx & & \\ \text{subject to} & Ax & \geq & e \\ & x_j & = & 0 \text{ or } 1 \text{ for all } j \end{array} \qquad (9.4.3)$$

where $A = (a_{ij})$ is a 0−1 matrix of order $m \times n$ and e is the column vector of all 1s in R^m. We will use the following problem as an example.

$$\begin{array}{ll} \text{Min.} & z(x) = 3x_1 + 2x_2 + 5x_3 + 6x_4 + 11x_5 + x_6 \\ & \quad +12x_7 + 7x_8 + 8x_9 + 4x_{10} + 2x_{11} + \end{array}$$

$$\begin{array}{rcl}
6x_{12} + 9x_{13} - 2x_{14} + 2x_{16} & & \\
\text{subject to } \; x_7 + x_9 + x_{10} + x_{13} & \geq & 1 \\
x_2 + x_8 + x_9 + x_{13} & \geq & 1 \\
x_3 + x_9 + x_{10} + x_{12} & \geq & 1 \\
x_4 + x_5 + x_8 + x_9 & \geq & 1 \\
x_3 + x_6 + x_8 + x_{11} & \geq & 1 \\
x_3 + x_6 + x_7 + x_{10} & \geq & 1 \\
x_2 + x_4 + x_5 + x_{12} & \geq & 1 \\
x_4 + x_5 + x_6 + x_{13} & \geq & 1 \\
x_1 + x_2 + x_4 + x_{11} & \geq & 1 \\
x_1 + x_5 + x_7 + x_{12} & \geq & 1 \\
x_{14} + x_{16} & \geq & 1 \\
x_{15} + x_{16} & \geq & 1 \\
x_j = 0 \text{ or } 1 \text{ for all } j & &
\end{array} \qquad (9.4.4)$$

In (9.4.3) a variable x_j is said to **cover** the ith constraint if x_j appears with a coefficient of 1 in this constraint. If x_j covers the ith constraint, any $0-1$ vector x in which the variable $x_j = 1$ satisfies this constraint automatically. We will now discuss some results which help to fix the values of some of the variables at 1 or 0, and eliminate some constraints, and thereby reduce the problem into an equivalent smaller size problem.

Result 9.4.1: *If $c \leq 0$, an optimum solution for (9.4.3) is $x = e_n$, the vector in R^n with all entries equal to 1. Terminate.*

In Result 9.4.1, e_n is the column vector in R^n with all entries equal to 1. As an example, consider the set covering problem with $n = 4$, i.e., the variables in this problem are $x = (x_1, x_2, x_3, x_4)^T$, all binary. If the vector of cost coefficients is $c = (-2, 0, 0, -7)$, which is ≤ 0, $\bar{x} = (1, 1, 1, 1)^T$ is an optimum solution of the problem yielding a value of -9 to the objective function (because -9 is the smallest value that cx can have in binary variables, and $\bar{x}$ will be clearly feasible because every variable has the value of 1 in it).

Result 9.4.2: *In (9.4.3) suppose $c \not\leq 0$. If j is such that $c_j \leq 0$ we can fix the corresponding variable x_j at 1 and eliminate all the constraints covered by this variable. If there are no more constraints left, fix all the remaining variables at 0, and this leads to an optimum solution to the problem in this case, terminate.*

For an example of Result 9.4.2, consider the following small set covering problem.

$$\begin{array}{rrcl} \text{Minimize} & z(x) = -6x_1 + 2x_3 + 5x_4 + 6x_5 & & \\ \text{subject to} & x_1 + x_3 & \geq & 1 \\ & x_2 + x_4 & \geq & 1 \\ & x_j = 0 \text{ or } 1 \text{ for all } j & & \end{array}$$

In this problem, the smallest value that $z(x)$ can have in binary variables is -6, and $\bar{x} = (1, 1, 0, 0, 0)^T$ attains this value for $z(x)$ and is clearly feasible to the problem, so it is an optimum solution to the problem.

Result 9.4.3: *If j is such that $c_j > 0$ and the variable x_j does not appear in any of the remaining constraints, fix the variable x_j at 0.*

As an example, consider the set covering problem given under Result 9.4.2. The binary variable x_5 there has a positive cost coefficient of 6, and does not appear in any of the constraints. So, making $x_5 = 1$ does not in any way help in satisfying any constraint, and costs a positive amount, hence it is optimal to fix $x_5 = 0$.

Apply the above results as many times as possible and reduce the problem. At the end we are left with a reduced problem of the same form as (9.4.3), in which every variable has a positive coefficient in the objective function. The greedy method is applied on this reduced problem, and it consists of applying the following general step repeatedly. Here, a **free variable** is one whose value is not fixed at 1 or 0 already.

GENERAL STEP: In the remaining problem, for each free variable x_j, let d_j be the number of remaining constraints covered by

x_j. c_j/d_j can be interpreted as the cost per constraint covered, associated with the free variable x_j at this stage. Find a free variable x_r which is associated with the smallest cost per constraint covered in the remaining problem. So, $c_r/d_r = \min\{c_j/d_j : j$ such that x_j is a free variable$\}$. Fix x_r at 1, and eliminate all the constraints covered by x_r. If there are no constraints left, fix all the remaining free variables at 0, and terminate with the vector obtained as the greedy solution vector. Otherwise apply Result 9.4.3 to the remaining problem and then go to the next step.

The solution vector at termination is the greedy solution for the set covering problem.

As an example, we will find the greedy solution for the set covering problem (9.4.4). First, applying Result 9.4.2, we fix $x_{14} = x_{15} = 1$ since their coefficients in $z(x)$ are ≤ 0 and eliminate the last two constraints covered by these variables. Now applying Result 9.4.3, we fix $x_{16} = 0$. The remaining problem is given in Table 1. All blank entries in the table are zero.

Table 1

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	
						1		1	1			1	≥ 1
	1						1	1				1	≥ 1
		1						1	1		1		≥ 1
			1	1			1	1					≥ 1
		1			1		1			1			≥ 1
		1			1	1			1				≥ 1
	1		1	1							1		≥ 1
			1	1	1							1	≥ 1
1	1		1							1			≥ 1
1				1		1					1		≥ 1
3	2	5	6	11	1	12	7	8	4	2	6	9	$= z(x)$

$x_j = 0$ or 1 for all j. Minimize $z(x)$

Letting d_j = number of remaining constraints covered by free variable x_j, we have the following information on the free variables at this stage.

Free var.	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}
c_j	3	2	5	6	11	1	12	7	8	4	2	6	9
d_j	2	3	3	4	4	3	3	3	4	3	2	3	3
$\frac{c_j}{d_j}$	$\frac{3}{2}$	$\frac{2}{3}$	$\frac{5}{3}$	$\frac{6}{4}$	$\frac{11}{4}$	$\frac{1}{3}$	4	$\frac{7}{3}$	2	$\frac{4}{3}$	1	2	3

The free variable with the smallest c_j/d_j of 1/3 at this stage is x_6. So we fix $x_6 = 1$, and eliminate constraints 5, 6, 8 in the above tableau covered by x_6. The remaining problem is given in Table 2.

Table 2

x_1	x_2	x_3	x_4	x_5	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	
					1		1	1			1	≥ 1
	1					1	1				1	≥ 1
		1					1	1		1		≥ 1
			1	1		1	1					≥ 1
	1		1	1						1		≥ 1
1	1		1						1			≥ 1
1				1	1					1		≥ 1
3	2	5	6	11	12	7	8	4	2	6	9	$= z(x)$

$x_j = 0$ or 1 for all j. Minimize $z(x)$

We have the following information on the free variables at this stage.

Free var.	x_1	x_2	x_3	x_4	x_5	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}
c_j	3	2	5	6	11	12	7	8	4	2	6	9
d_j	2	3	1	3	3	2	2	4	2	1	3	2
$\frac{c_j}{d_j}$	$\frac{3}{2}$	$\frac{2}{3}$	5	2	$\frac{11}{3}$	6	$\frac{7}{2}$	2	2	2	2	$\frac{9}{2}$

The free variable with the smallest c_j/d_j of 2/3 at this stage is x_2. We fix $x_2 = 1$, and eliminate constraints 2, 5, 6 in Table 2 covered by x_2. x_{11} with a cost coefficient of 2 does not appear in any of the remaining constraints, so we fix it at 0. The remaining problem is given in Table 3.

Table 3

x_1	x_3	x_4	x_5	x_7	x_8	x_9	x_{10}	x_{12}	x_{13}	
				1		1	1		1	≥ 1
	1					1	1	1		≥ 1
		1	1		1	1				≥ 1
1			1	1				1		≥ 1
3	5	6	11	12	7	8	4	6	9	$= z(x)$

$x_j = 0$ or 1 for all j. Minimize $z(x)$

We have the following information on the free variables at this stage.

Free var.	x_1	x_3	x_4	x_5	x_7	x_8	x_9	x_{10}	x_{12}	x_{13}
c_j	3	5	6	11	12	7	8	4	6	9
d_j	1	1	1	2	2	1	3	2	2	1
$\frac{c_j}{d_j}$	3	5	6	$\frac{11}{2}$	6	7	$\frac{8}{3}$	2	3	9

The free variable with the smallest c_j/d_j of 2 at this stage is x_{10}. We fix $x_{10} = 1$ and eliminate constraints 1, 2 in Table 3. x_3, x_{13}, with positive cost coefficients, do not appear in any of the remaining constraints, so we fix them at 0. The remaining problem is given in Table 4.

Table 4

x_1	x_4	x_5	x_7	x_8	x_9	x_{12}	
	1	1		1	1		≥ 1
1		1	1			1	≥ 1
3	6	11	12	7	8	6	$= z(x)$

$x_j = 0$ or 1 for all j. Minimize $z(x)$

We have the following information on the free variables at this stage.

Free var.	x_1	x_4	x_5	x_7	x_8	x_9	x_{12}
c_j	3	6	11	12	7	8	6
d_j	1	1	2	1	1	1	1
$\frac{c_j}{d_j}$	3	6	$\frac{11}{2}$	12	7	8	6

The free variable x_1 has the smallest c_j/d_j of 3 at this stage. We fix $x_1 = 1$ and eliminate constraint 2 in the above tableau covered by it. x_7, x_{12} which do not appear in the remaining constraint are fixed at 0. The remaining problem is:

$$\begin{array}{rl} \text{Minimize} & 6x_4 + 11x_5 + 7x_8 + 8x_9 \\ \text{subject to} & x_4 + \ x_5 + \ x_8 + \ x_9 \ \geq \ 1 \\ & x_j = 0 \text{ or } 1 \text{ for all } j \end{array}$$

This remaining problem has only one constraint. The free variable x_4 has the smallest c_j/d_j of 6, so we fix it at 1 and the remaining free variables x_5, x_8, x_9 at 0.

Collecting the values given to the variables at various stages, we see that the greedy solution obtained is (x_1, to x_{16}) = (1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0)^T, with an objective value of 14.

9.4.3 Greedy-Type Methods for the TSP

An n-city TSP with cost matrix $c = (c_{ij})$ is the problem of determining a minimum cost tour, which is an order of visiting n cities each once and only once, beginning with a starting city and terminating at the initial city in the end. Hence it is a sequencing problem.

Let the cities in the problem be 1, 2, ..., n. If the tour begins in city 1, travels the remaining cities in the order $i_2, i_3, \ldots, i_n$, and then returns to the initial city 1 from i_n; the tour will be denoted by $1, i_2, i_3, \ldots, i_n; 1$. The arcs in this tour are: $(1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)$, $(i_n, 1)$.

For example, if the cities are traveled in serial order $1, 2, 3, \ldots, n-1, n$ and then finally back to 1, the tour will be denoted by $1, 2, 3, \ldots, n; 1$.

The greedy methods for the TSP try to construct a near-optimal tour by building the sequence one element at a time using a greedy approach. Hence they are classified as **tour construction procedures or heuristics**. We describe some of the popular ones here.

1. **Nearest neighbor heuristic** This nearest neighbor method is obtained by applying the greedy approach to the TSP subject to the constraint that the tour being constructed grow in a connected fashion.

   Starting with an initial city, this procedure builds a sequence one city at a time. The next city in the sequence is always the closest to the current city among the unincluded cities. In the end the last city is joined to the initial city.

   Typically, this process is repeated with each city selected as the initial one. The best among the n tours generated in this process is selected as the output of this algorithm.

   It can be proved [D. Rosenkratz, R. Sterns, and P. Lewis, 1977] that for the Euclidean TSP (i.e., the distance matrix is positive, symmetric, and satisfies the triangle inequality), the following result holds.

$$\frac{\text{Length of the nearest neighbor tour}}{\text{Length of an optimum tour}} \leq \frac{1}{2}(1 + \log_2 n)$$

Example 9.4.1

$$c = (c_{ij}) = \begin{array}{r|rrrrrr}
\text{to } j = & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\text{from } i = 1 & \times & 14 & 23 & 25 & 36 & 42 \\
2 & 14 & \times & 17 & 23 & 30 & 36 \\
3 & 23 & 17 & \times & 29 & 35 & 28 \\
4 & 25 & 23 & 29 & \times & 17 & 11 \\
5 & 36 & 30 & 35 & 17 & \times & 6 \\
6 & 42 & 36 & 28 & 11 & 6 & \times \\
\hline
\end{array}$$

Consider a 6 city TSP with the cost matrix given above. Here are the tours obtained by the nearest neighbor heuristic in this problem.

Starting city	Nearest neighbor tour	cost
1	1, 2, 3, 6, 5, 4; 1	107
2	2, 1, 3, 6, 5, 4; 2	111
3	3, 2, 1, 4, 6, 5; 3	108
4	4, 6, 5, 2, 1, 3; 4	113
5	5, 6, 4, 2, 1, 3; 5	112
6	6, 5, 4, 2, 1, 3; 6	111

So, the output of this algorithm is the tour 1, 2, 3, 6, 5, 4; 1 with a cost of 107.

2. **The Clark and Wright savings heuristic** Select an initial city, say city 1. Think of the initial city as a central depot, beginning at which all the cities have to be visited. For each ordered pair of cities not containing the initial city, (i, j) say, compute the savings s_{ij} of visiting the cities in the order 1, i, j, 1 as opposed to visiting each of them independently from 1 as in the orders 1, i, 1 and 1, j, 1. This savings s_{ij} is therefore equal to $(c_{1i} + c_{i1}) + (c_{1j} + c_{j1}) - (c_{1i} + c_{ij} + c_{j1})$. If the cost matrix $c = (c_{ij})$ is symmetric, we have $s_{ij} = s_{ji} = c_{i1} + c_{1j} - c_{ij}$.

   Order these savings values in decreasing order from top to bottom. Starting at the top of the savings list and moving downwards, form ever larger subtours by inserting new cities, one at a time, adjacent to the initial city on either side of the subtour, as indicated by the pair corresponding to the present savings, whenever it is feasible to do so. Repeat until a tour is formed.

   Typically this process is repeated with each city as the initial one, and the best of all the tours obtained is taken as the output.

   As an example, consider the TSP of order 6 with the cost matrix given in Example 9.4.1. Since the cost matrix is symmetric, the savings $s_{ij} = s_{ji}$ for all i, j; so we need to compute them only for $j > i$. Suppose city 1 is selected as the initial city. $s_{23} = c_{12} + c_{13} - c_{23} = 14 + 23 - 17 = 20$. The savings coefficients computed this way are given below.

	s_{ij} for $j > i$				
$j =$	2	3	4	5	6
$i = 2$	$\times$	20	16	20	20
3		$\times$	19	24	37
4			$\times$	44	56
5				$\times$	72
6					$\times$

The savings coefficients arranged in decreasing order, and the subtour grown are shown below (here we used the fact that the cost matrix is symmetric).

Savings coeff.	Its value	Present subtour
s_{56}	72	1, 5, 6, 1
s_{46}	56	1, 5, 6, 4, 1
s_{45}	44	”
s_{36}	37	”
s_{35}	24	1, 3, 5, 6, 4, 1
s_{23}	20	1, 2, 3, 5, 6, 4; 1

So, the tour 1, 2, 3, 5, 6, 4; 1 with a cost of 108 is obtained by this procedure beginning with city 1 as the initial tour. The same process can be repeated with other cities as initial cities. The best of all the tours generated is the output of the algorithm.

3. Nearest insertion heuristic The insertion procedure grows a subtour until it becomes a tour. In each step it determines which node not already in the subtour should be added next, and where in the subtour it should be inserted.

The algorithm selects one city as the initial city, say city i. Then find $p \neq i$ such that $c_{ip} = \min\{c_{ij} : j \neq i\}$. The initial subtour is i, p, i.

Given a subtour, S say, find a city r not in S, and a city k in S such that $c_{kr} = \min\{c_{pq} : p \in S, q \notin S\}$. City r is known as the **closest or nearest city to** S among those not in it. It is selected as the city to be added to the subtour at this stage. Find an arc (i, j) in subtour which minimizes $c_{ir} + c_{rj} - c_{ij}$. Insert r between i and j on the subtour S.

Repeat until the subtour becomes a tour.

As an example, consider the TSP of order 6 with the cost matrix given in Example 9.4.1. Suppose city 1 is selected as the initial city. Min$\{c_{1j} : j \neq 1\}$ is c_{12}. So, the initial subtour is 1, 2, 1. The closest outside city to this subtour is city 3, and by symmetry inserting it on any arc of the subtour adds the same to the cost, so we take the next subtour to be 1, 3, 2, 1. The nearest outside city to this subtour is city 4. Min$\{c_{14} + c_{43} - c_{13}, c_{34} + c_{42} - c_{32}, c_{24} + c_{41} - c_{21}\} = c_{14} + c_{43} - c_{13} = 31$. So, the new subtour is 1, 4, 3, 2, 1. Continuing this way we get the subtour 1, 4, 6, 3, 2, 1; and finally the tour 1, 4, 5, 6, 3, 2; 1 with a cost of 107. The procedure can be repeated with each city as the initial city, and the best of the tours obtained taken as the output of the algorithm.

It has been proved [D. Rosenkratz, R. Sterns, and P. Lewis, 1977] that on a Euclidean TSP, the tour obtained by this method has a cost no more than twice the cost of an optimum tour.

4. **Cheapest insertion heuristic** This procedure also grows a subtour until it becomes a tour. It is initiated the same way as the nearest insertion heuristic. Given a subtour S say, it finds an arc (i, j) in S and a city r not in S such that the index $c_{ir} + c_{rj} - c_{ij}$ is minimal, and then inserts r between i and j. Repeat until a tour is obtained. This procedure also can be repeated with each city as the initial city, and the best of the tours obtained taken as the output. On Euclidean TSPs, this method has the same worst case bound as the nearest insertion heuristic.

   As an example, consider the TSP of order 6 with the cost matrix given in Example 9.4.1. Suppose city 1 is selected as the initial city. Min$\{c_{1j} : j \neq 1\}$ is c_{12}. So, the initial subtour is 1, 2; 1 with two arcs (1, 2), (2, 1). The indices for selecting the next insertion among cities 3, 4, 5, 6 missing in present subtour are:

Arc (i,j) in	Index $c_{ir}+c_{rj}-c_{ij}$ for $r=$			
present subtour	3	4	5	6
(1, 2)	26	34	52	64
(2, 1)	26	34	52	64

So, inserting $r = 3$ between arc (1, 2) or (2, 1) in the present subtour provides the cheapest insertion (smallest index value = 26). Hence at this stage we can insert 3 between arc (1, 2) or (2,1) in the present subtour. Suppose we insert 3 between 1 and 2, leading to the new subtour 1,3,2;1 with arcs (1, 3), (3, 2), (2,1). The indices for the next insertion are:

Arc (i,j) in	Index $c_{ir}+c_{rj}-c_{ij}$ for $r=$		
present subtour	4	5	6
(1, 3)	31	48	47
(3, 2))	35	48	47
(2, 1)	34	52	64

Here the cheapest insertion (index value of 31) is to insert 4 in arc (1, 3) of the present subtour, leading to the next subtour 1, 4, 3, 2;1. The indices for the next insertion are:

Arc (i,j) in	Index $c_{ir}+c_{rj}-c_{ij}$ for $r=$	
present subtour	5	6
(1, 4)	28	28
(4, 3)	23	10
(3, 2))	48	47
(2, 1)	52	64

Here the cheapest insertion (index value of 10) is to insert 6 in arc (4, 3) of the present subtour, leading to the next subtour 1, 4, 6, 3, 2;1. Only city 5 remains to be inserted now. The indices for for its insertion are:

Index	Arc (i, j) in present subtour				
	(1, 4)	(4, 6)	(6, 3)	(3, 2)	(2, 1)
$c_{i5} + c_{5j} - c_{ij}$	28	12	13	48	52

So, the cheapest insertion for 5 is on arc (4, 6) of the present subtour. It leads to the tour 1, 4, 5, 6, 3, 2;1 which is the output of this heuristic.

5. Nearest merger heuristic This procedure is initiated with n subtours, each consisting of a single city and no arcs. In each step it finds the least costly arc, (a, b) say, that goes between two subtours in the current list, and merges these two subtours into a single subtour.

If a, b are two single city subtours in the current list, their merger replaces them with the subtour a, b, a.

If one of a, b is in a single city subtour, say a, and the other in a multi-city subtour; insert a into the subtour containing b using the cheapest way of inserting it as discussed under the cheapest insertion heuristic.

If the subtours in the current list containing cities a and b each have two or more cities, find the arc (p_1, q_1) in the first subtour, and the arc (p_2, q_2) in the second subtour, such that $c_{p_1p_2} + c_{q_2q_1} - c_{p_1q_1} - c_{p_2q_2}$ is minimized. Then merge these subtours by deleting arcs $(p_1, q_1), (p_2, q_2)$ from them, and adding $(p_1, p_2), (q_2, q_1)$ to them.

As an exercise, we ask the reader to apply this heuristic on the TSP of order 6 with the cost matrix given in Example 9.4.1.

We discussed a variety of greedy-type single pass heuristics for the TSP to give the reader some idea of how greedy methods can be developed for combinatorial optimization problems. All the methods discussed here for the TSP produce reasonably good tours with objective values usually close to the optimum objective value.

On the same problem different heuristics may give different results, as the reader can verify from the results on the 6-city TSP with cost matrix given in Example 9.4.1. That's why some times people solve their problem with different heuristics, and take for implementation the best solution obtained.

9.4.4 A Greedy Method for the Single Depot Vehicle Routing Problem

This problem is concerned with delivering goods to customers at various locations in a region by a fleet of vehicles based at a depot. The index $i = 0$ denotes the depot, and $i = 1$ to n denote the customer locations. N denotes the number of vehicles available at the depot. The following data is given.

c_{ij}	$=$	distance (or cost, or driving time for traveling) from i to j, for $i, j = 0$ to n.
k_v	$=$	capacity of vehicle v in tons or some other units, $v = 1$ to N.
T_v	$=$	maximum distance (or cost, or driving time) that vehicle v can operate, $v = 1$ to N.
d_i	$=$	demand or amount of material (in tons or other units in which vehicle capacities are also measured) to be delivered to customer i, $i = 1$ to n; $d_0 = 0$.

All customer demands need to be delivered. The problem is to determine: (i) the subset of customers to be allotted to each vehicle that is used, and (ii) the route that each vehicle should follow (i.e., the order in which it should visit its allotted customers) so as to minimize the total distance (or cost or driving time) of all the vehicles used to make the deliveries. This is a prototype of a common problem faced by many warehouses, department stores, parcel carriers, and trucking firms and is therefore a very important problem. We will discuss a greedy-type method known as the **Clarke and Wright savings heuristic** [G. Clarke and J. Wright, 1964] that is very popular. It is an exchange procedure, which in each step exchanges the current set of routes for a better set. Initially, think of each customer being serviced by a separate vehicle from the depot. See the left part of Figure 9.1.

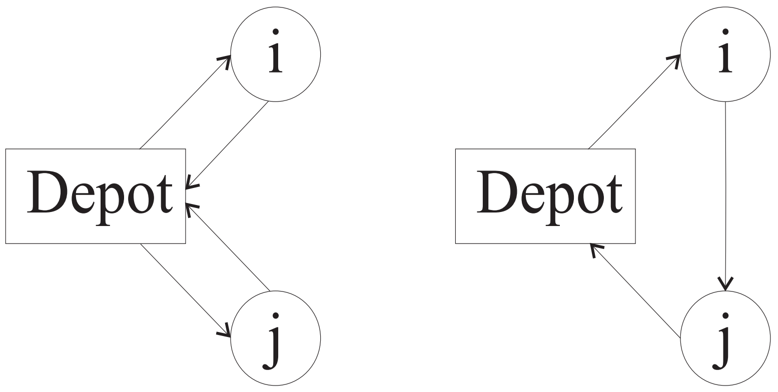


Figure 9.1: On the left, customers i and j are serviced by two vehicles from the depot. On the right, they are both serviced by the same vehicle.

If it is feasible to service customer j by the same vehicle which serviced customer i (i.e., if the vehicle capacity and maximum distance constraints are not violated by doing this) before returning to the depot (see the right part of Figure 11.1), the savings in distance will be $s_{ij} = c_{0i} + c_{i0} + c_{0j} + c_{j0} - (c_{0i} + c_{ij} + c_{j0}) = c_{i0} + c_{0j} - c_{ij}$. These savings coefficients s_{ij} are computed for all $i \neq j = 1$ to n, and ordered in decreasing order from top to bottom. Starting at the top of their savings list, form a route for vehicle 1 (which will be a subtour beginning and ending at the depot) by inserting new customers, one at a time, adjacent to the depot on either side of the depot as discussed in the Clark and Wright savings heuristic for the TSP, until either the vehicle capacity is used up or the maximum distance it can travel is reached. Now delete the customers allotted to vehicle 1 from the list.

Repeat the same process to form a route for vehicle 2 with the savings coefficients for pairs of remaining customers; and continue in the same way until all the customers are allotted to a vehicle.

The above process determines the subset of customers to be serviced by each vehicle used, and a tour to be followed by each vehicle to service its customers. One can now try to find a better tour for each vehicle to service its allotted customers using some of the algorithms discussed in Sections 9.4.3 and 9.5.

As an example, consider the problem involving 12 customers and the following data. The symmetric distance matrix (c_{ij}) (c_{ij} is the distance in miles between customers i, j) is:

Symmetric distance matrix (miles) = (c_{ij})

to	0	1	2	3	4	5	6	7	8	9	10	11	12
from 0	×	9	14	21	23	22	25	32	36	38	42	50	50
1		×	5	12	22	21	24	31	35	37	41	49	51
2			×	7	17	16	23	26	30	36	36	44	46
3				×	10	21	30	27	37	43	31	37	39
4					×	19	28	25	35	41	29	31	29
5						×	9	10	16	22	20	28	30
6							×	7	11	13	17	25	27
7								×	10	16	10	18	20
8									×	6	6	14	16
9										×	12	12	20
10											×	8	10
11												×	10
12													×

There is no limit on the distance that a truck can travel. For $i = 1$ to 12, data on d_i = the amount to be delivered to customer i in gallons is:

Customer i	1	2	3	4	5	6
d_i	1200	1700	1500	1400	1700	1400
Customer i	7	8	9	10	11	12
d_i	1200	1900	1800	1600	1700	1100

Truck capacity (gallons)	Up to 4000	4000-5000	5000-6000
Number of trucks available	10	7	4

Since the distance matrix is symmetric, the matrix of savings coefficients is also symmetric. For example $s_{2,1} = s_{1,2} = c_{1,0} + c_{0,2} - c_{1,2}$ $= 9 + 14 - 5 = 18$. All the savings coefficients are computed in the same manner and are given below.

Symmetric savings matrix (miles) = (s_{ij})

$j =$	1	2	3	4	5	6	7	8	9	10	11	12
$i = 1$	×	18	18	10	10	10	10	10	10	10	10	8
2		×	28	20	20	16	20	20	16	20	20	18
3			×	34	22	16	26	20	16	32	34	32
4				×	26	20	30	24	20	36	42	44
5					×	38	44	42	38	44	44	42
6						×	50	50	50	50	50	48
7							×	58	54	64	64	62
8								×	68	72	72	70
9									×	68	76	68
10										×	84	82
11											×	90
12												×

The largest savings coefficient is $s_{11,12} = 90$. So, the initial subtour for vehicle 1 is 0, 11, 12; 0. The demand at customers 11, 12 put together is 1700 + 1100 = 2800 gallons. The next biggest savings coefficient is $s_{10,11} = 84$. So, we insert customer 10 into the subtour for vehicle 1, leading to the new subtour 0, 10, 11, 12; 0 with a total demand of 1600 + 2800 = 4400 gallons. Customers 10, 11, 12 are already assigned to vehicle 1. The next largest savings coefficient involving one of the remaining customers is $s_{9,11} = 76$, but adding customer 9 to vehicle 1 will make the total demand = 1800 + 4400 = 6200 gallons; but the depot has no vehicles of this capacity, so we drop customer 9 from consideration for vehicle 1. The next highest savings coefficients are $s_{8,10} = s_{8,11} = s_{8,12} = 72$, but again, customer 8 cannot be allocated to vehicle 1 because this allocation will make the total demand > the largest capacity of available vehicles.

The next highest savings coefficient is $s_{8,9} = 68$ and both customers 8 and 9 are presently unassigned. So, we select 0, 8, 9; 0 as initial subtour for vehicle 2. The combined demand of these two customers is $1900 + 1800 = 3700$ gallons.

The next highest savings coefficients are $s_{7,10} = s_{7,11} = s_{7,12} = 64$. So we insert customer 7 into the subtour for vehicle 1, leading to the new subtour 0, 7, 10, 11, 12; 0 with a total demand of 5600 gallons for vehicle 1. So, we make vehicle 1 to be one of the vehicles with capacity 5000-6000 gallons, and $\tau_1 = 0, 7, 10, 11, 12; 0$ as the subtour for it to follow. No more customers can be added to this vehicle because of the capacity constraint. Since this vehicle is now full, in the sequel we ignore all the savings coefficients involving one of the customers 7, 10, 11, or 12 assigned to this vehicle. And there are 3 vehicles of capacity 5000-6000 gallons still available.

The next highest savings coefficient involving an unassigned customer is $s_{6,8} = 50$. So, we combine customer 6 in vehicle 2 leading to the new subtour for it of 0, 6, 8, 9; 0 with a total demand of 5100 gallons. No more customers can be assigned to vehicle 2 because of the capacity constraint. Thus we make vehicle 2 to be another vehicle with capacity 5000-6000 gallons, and $\tau_2 = 0, 6, 8, 9; 0$ as the subtour for it to follow. And there are 2 more vehicles of capacity 5000-6000 gallons left.

The next highest savings coefficient involving unassigned customers is $s_{3,4} = 34$. So, we combine customers 3, 4 into the subtour 0, 3, 4; 0 which will be the initial subtour for vehicle 3 with a total demand of 2900 gallons. The next highest savings coefficient not involving a customer assigned to the already full vehicles 1, 2, is $s_{2,3} = 28$. So, we insert customer 2 into the subtour for vehicle 3, changing it into 0, 2, 3, 4; 0 with a total demand of 4600 gallons.

The next highest savings coefficients of $s_{45} = 26, s_{12} = 18$ cannot be used because adding any of customers 5 or 1 to vehicle 3 will exceed maximum available vehicle capacity. This leads to the next highest savings coefficient $s_{1,5} = 10$. Hence we combine customers 1, 5 into the subtour 0, 1, 5; 0 with a total demand of 2900 gallons for vehicle 4. Now all the customers are assigned. Here is a summary of the assignments.

Vehicle	Its subtour	Total demand	Vehicle capacity
1	$\tau_1 = 0, 7, 10, 11, 12; 0$	5600 gal.	5000-6000 gal.
2	$\tau_2 = 0, 6, 8, 9; 0$	5100 gal.	"
3	$\tau_3 = 0, 2, 3, 4; 0$	4600 gal.	4000-5000 gal.
4	$\tau_4 = 0, 1, 5; 0$	2900 gal.	4000 gal.

We should now try to find better tours for each vehicle to cover the customers assigned to it, using some of the other methods discussed in Sections 9.4.3 and the following sections.

9.4.5 General Comments on Greedy Heuristics

So far we discussed a variety of ways of developing single pass heuristic methods for a variety of problems based on the greedy principle. One important point to notice is that heuristic methods are always tailormade for the problem being solved, taking its special structure into account. Practical experience indicates that for the problems discussed in this section, the heuristic methods discussed usually lead to satisfactory near-optimal solutions.

A point of caution. It is perfectly reasonable to use greedy or other single pass heuristic methods if either theoretical worst case analysis, or extensive computational testing, has already established that the method leads to reasonable near optimal solutions. In the absence of encouraging theoretical results on worst case error bounds, or encouraging results from computational tests, one should be wary of relying solely on a greedy or any other single pass heuristic. In this case it is always better to combine it with some heuristic search methods discussed in the following sections.

Exercises

9.4.1: A bank account location problem A business firm has clients in cities $i = 1$ to m, and can maintain bank accounts in locations $j = 1$ to n. When the payment for a client is mailed by a check, there is usually some time lag before the check is cashed (time for the mail to reach back and forth), in that time the firm continues to collect interest on that money. Depending on the volume of business in city i, and the time it takes for mail to go between city i and location j, one can estimate the float = expected benefit s_{ij} in the form of this interest if clients in city i are paid by checks drawn out of a bank account in location j, $i = 1$ to m, $j = 1$ to n. The following data is given.

c_j=	cost in money units for maintaining a bank account in location $j = 1$ to n, per year
s_{ij}=	total float (= expected benefit in the form of interest on money between the time a check for it is mailed, and the time that check is cashed) per year, if payments due for customers in city i are mailed in the form of checks drawn out of a bank account in location j, $i = 1$ to m, $j = 1$ to n.
N=	upper bound on the number of bank accounts that the firm is willing to maintain in locations 1 to n.

Here is the data for a numerical example: $m = 7$, $n = 5$, $N = 3$, and

	$j =$	1	2	3	4	5
	$i = 1$	2	11	6	9	8
	2	7	1	8	2	10
	3	7	3	2	3	4
$(s_{ij}) =$	4	10	9	4	2	1
	5	3	8	5	6	2
	6	4	3	4	1	6
	7	6	5	1	8	4
	c_j	3	2	1	3	4

(i) Formulate the problem of determining the subset of locations where bank accounts should be maintained, and the bank accounts through which customers in each city should be paid, so as to maximize (the total annual float earned − yearly cost of maintaining the bank accounts), as a 0−1 pure IP.

(ii) Consider the numerical example with data given above. Suppose

$\mathbf{J} \subset \{1, \ldots, n\}$, the subset of locations where bank accounts are to be maintained, is given. Then clearly for each $i = 1$ to m, customers in city i should be paid by checks drawn out of location r where r attains the maximum in $\max\{s_{ij} : j \in \mathbf{J}\}$, i.e., each customer should be paid from the bank account in location with the maximum float value in the row of the customer city, to maximize total float.

For illustration, if bank accounts are opened in locations $j = 1$ and 3 only in the example given above: customers in cities $i = 1, 2, 5$ should be paid out of locations $j = 3$; customers in cities $i = 3, 4, 7$ should be paid out of location $j = 1$; and customers in city $i = 6$ can be paid out of locations $j = 1$ or 3 (the float values are equal). Also if a new bank account is opened in location 4, only customers in cities 1, 5, and 7 should be switched from their current account to this new account, because this will increase the float coming from them. Thus when bank accounts are already available in locations 1, 3, opening a new bank account in location 4 leads to a net extra profit of $(9 - 6) + (6 - 5) + (8 - 6) - 3 = 3$ money units (here the terms $9 - 6$, $6 - 5$, $8 - 6$ are the extra floats that will be obtained when customers in cities 1, 5, 7 are switched from their present account to this new account; and the last term 3 is the cost of maintaining an account in the new location). This net quantity 3 is called the **evaluation** of location 4 when bank accounts are already available in locations 1, 3. It measures the net extra profit that can be gained by opening a new account at location 4.

Using such evaluations as the criterion to be greedy upon, develop a greedy method for finding the subset of locations where bank accounts should be maintained in this problem. The method should open one new account at a time, until either N accounts are opened, or it turns out that opening a new account only decreases the net income. Solve the problem with data given above, using this method.

9.5 Interchange Heuristics

Interchange (or exchange) heuristics are local search methods that start with a solution and search for better solutions through local improvement, i.e., through small changes in the solution in each step. When the problem is represented as one of selecting an optimal subset from a set (or as one of arranging a set of objects in a sequence optimally), the method starts with an initial solution which may either be randomly generated or obtained by a single pass heuristic such as the greedy method; and attempts to improve it by exchanging a small number of elements in the solution with those outside it (or by changing the positions of a small number of objects in a sequence). If a better solution is found by such interchanges, the same process is repeated on it. This process is continued until a solution that cannot be improved by such interchanges is found. This final solution is a local minimum under such interchange operations. The procedure is usually repeated with several initial solutions, and the best of the local minima found is taken as the output of the algorithm. Heuristic methods based on this type of search are called **interchange** or **exchange heuristic methods**.

For TSP, the most popular exchange heuristic methods are **2-Opt** or **3-Opt** (starting with a tour, these procedures try to find a better tour by exchnging two (in 2-opt) or three (in 3-opt) arcs in the tour with two or three arcs not in the tour). We discuss these metods next.

2-Opt Heuristic for the Symmetric Traveling Salesman Problem (TSP), and 3-Opt Heuristic for the Symmetric or Asymmetric TSP

We now consider the n-city TSP with cost matrix $c = (c_{ij})$, where c_{ij} = cost of travel from city i to city j, for $i, j = 1$ to n, $i \neq j$. We are required to find a tour, beginning at some city, visiting each of the other cities once and only once in some order, returning to the starting city at the end; that has the smallest cost among all such tours.

If the cost matrix c is symmetric, i.e., $c_{ij} = c_{ji}$ for all i, j; then this TSP is known as a **symmetric TSP**. In a symmetric TSP, a link between any pair of cities i, j can be traveled in either of the two directions (i to j, or j to i) for the same cost. Hence in this case all links between various pairs of cities can be treated as edges without any specified orientation, so every tour is a cycle containing all the cities, its cost is the same whether it is traveled in the clockwise or anticlockwise direction. See Figure 9.2.

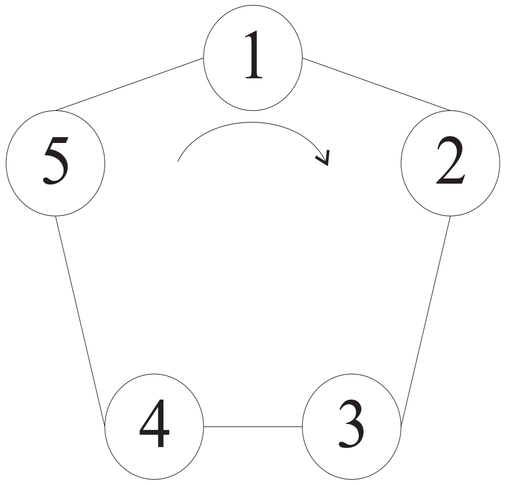


Figure 9.2: Tour covering cities 1 to 5 in a symmetric TSP is a cycle without specified orientation. Orienting this in the clockwise direction, as indicated by the arrow, gives the tour 1, 2, 3, 4, 5; 1. When oriented in the anticlockwise direction, it leads to the tour 1, 5, 4, 3, 2; 1. Both the tours have the same cost, and correspond to the same set of edges.

If c is not symmetric, i.e., $c_{ij} \neq c_{ji}$ for at least one pair of cities i, j; then this TSP is known as an **asymmetric TSP**. In this case the cost of traveling on the link between cities i, j in the two directions may be different; so all the links joining pairs of cities are treated as directed arcs with travel allowed only in the direction specified for that arc.

The interchange heuristic for either TSPs begins with a tour τ, and searches for a better tour among all those that differ from τ in 2, 3, or a small number of arcs, or edges. If such a tour is found, the method moves to that and continues in the same way. The final tour obtained by the 2 edge interchange scheme is called a **2-0pt tour**, and the one obtained by the 3 arc interchange scheme is called a **3-Opt tour** for the TSP.

2-edge interchange works only for the symmetric TSP. Figure 9.3

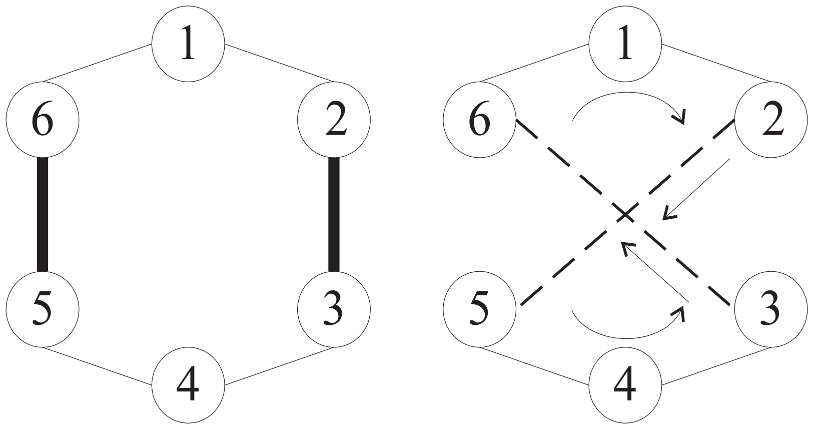


Figure 9.3: Original tour shown on the left, with the thick edges (2; 3), (5, 6) to be deleted from it in the 2-edge interchange operation. Since the remaining four edges in this tour are staying, the only two edges that can replace the dropped edges are (2; 5), (6; 3). New tour shown on right. For each tour we mark the clockwise orientation at node 1.

shows the 2-edge interchange operation carried out on a tour (on the left) covering $n = 6$ cities (each city represented by a node with the city number entered inside it), 1 to 6; and the new tour obtained after the interchange on the right.

Notice that in the new tour obtained after the 2-edge interchange, some of the remaining old edges will be travelled in the direction opposite to that in the original tour, but in a symmetric TSP this is OK, since each edge can be traveled in either direction.

The 3-link interchange works both for the symmetric or asymmetric TSP, we illustrate it treating each link as a directed arc. In Figure 9.4 we display a 3-arc interchange. The nodes represent the cities with their numbers entered inside. The initial tour τ_1 is drawn in solid lines. The second tour τ_2 is obtained by exchanging the three thick arcs in τ_1 with the three dashed arcs.

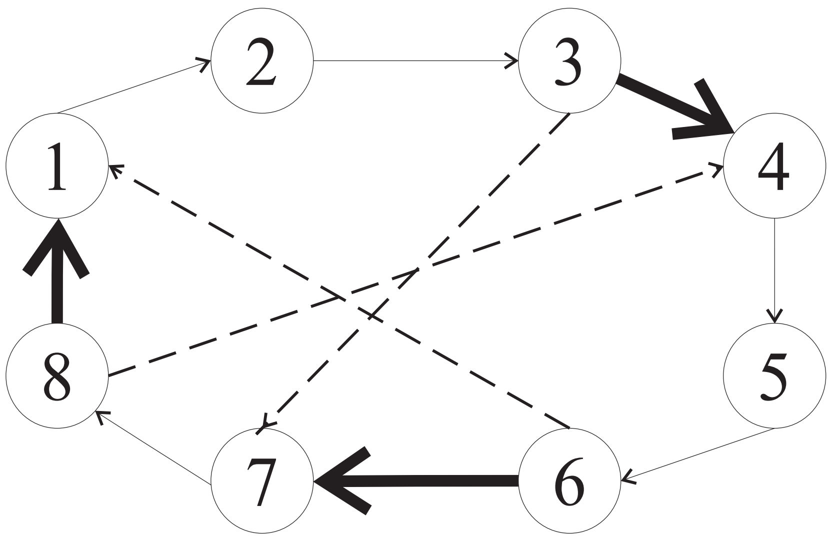


Figure 9.4: The three arc swap.

Deleting the thick arcs in the solid tour τ_1 in Figure 9.4 is equivalent to looking at the restricted problem with the arcs (1, 2), (2, 3); (4, 5), (5, 6); (7, 8) already fixed in the tour. To avoid subtours, this implies that arcs (3, 1), (6, 4), and (8, 7) are forbidden. Hence the three best outside arcs to replace the thick arcs in Figure 9.4 are the arcs in an optimum tour for the TSP of order 3 with the following cost matrix:

to	1	4	7
from 3	$\times$	c_{34}	c_{37}
6	c_{61}	$\times$	c_{67}
8	c_{81}	c_{84}	$\times$

which can be solved easily by inspection, since a TSP of order 3 has only 2 possible tours. If this produces a tour τ_2 with total cost less than that of τ_1, the choice of the set of three thick arcs to exchange from τ_1 has been successful, and the process is now repeated with the new tour τ_2. If the cost of τ_2 is $\geq$ the cost of τ_1, the process is continued with τ_1 and a different subset of three arcs from it to exchange. If every subset of three arcs to exchange from τ_1 leads to a tour whose cost is $\geq$ that of τ_1, the three arc interchange heuristic terminates with τ_1 as a near optimum (3-opt) tour. To obtain a close approximation to an optimum tour, one should repeat this interchange procedure with a number of initial tours, and take the best of all the tours obtained as the final output.

We will now discuss an interchange heuristic method for a location problem.

9.5.1 An Interchange Heuristic for a Training Center Location Problem

A large company has offices in many cities around the country. Due to the continuing development of new technologies, they expect to have a steady demand in the future for the training of their employees. Hence they are embarking on a huge employee training and education program. They want to develop a few training centers, these will be located in a subset of cities where the company has offices. Once these centers are established, employees from various cities will be sent to these centers for training. We assume that each center will have the capacity to take an unlimited number of trainees.

All the employees needing training in a city will be assigned to the same training site (i.e., they will not be split between different training sites). Also, a trainee may have to make several trips back and forth before his training is complete. We are given the following data.

n	$=$	number of cities where offices are located
s_i	$=$	expected number of employees at city i needing training annually, $i = 1$ to n
m_i	$=$	expected number of trips between city i and training center annually by trainees from city i, $i = 1$ to n

c_{ij}	$=$	cost per trip between cities i and j, $i, j = 1$ to n
d_j	$=$	expected staying cost per trainee during training program, if a training center is located in city j, $j = 1$ to n
r_{ij}	$=$	$s_i d_j + m_i c_{ij}$ = total cost (travel + staying) incurred annually by trainees from city i if they are assigned to a training center located in city j, i, j $= 1$ to n
p	$=$	number of training centers to be opened

Suppose $V = \{j_1, \ldots, j_p\}$, the set of cities among 1 to n where training centers will be opened, is given. Then, to minimize the total cost, we should assign the trainees from city i to the cheapest training site in V, i.e., to $j_t \in V$ where j_t satisfies $r_{ij_t} = \min\{r_{ij_k} : k = 1$ to $p\}$, for each $i = 1$ to n. Thus given the set of training sites $V = \{j_1, \ldots, j_p\}$, the minimum total annual cost (expected annual cost of travel + stay at assigned training centers during training for all the trainees) is $\sum_{i=1}^{n}(\min\{r_{ij_k} : k = 1 \text{ to } p\})$.

The problem is to find the set of training sites that minimizes the total annual cost. This problem is known as the **p-median problem**.

The interchange heuristic for this problem is initiated with a set of p sites for training centers, and applies the following general step repeatedly:

General Step Let V be the present set of sites for training centers. For each $a \in V, b \notin V$ define Δ_{ab} as the change in the total cost if a in V is replaced by b. For each $a \in V$ define

T_a	$=$	market set for city a, i.e., the set of cities which send their trainees to a, it is $\{i : r_{ia} = \min\{r_{ij} : j \in V\}\}$.

For any $a \in V, b \notin V$, to compute Δ_{ab} it is necessary to find the "cheapest" new assignments for trainees from cities in T_a when b replaces a from V; and any other cities outside T_a which will also be switched from their present assignments to b.

Start computing Δ_{ab} for $a \in V, b \notin V$. In this process, if Δ_{gq} for $g \in V, q \notin V$ is the first negative quantity obtained, let $V' = \{q\} \cup (V \setminus \{g\}$. With V' as the new set of sites for training centers repeat this general step.

On the other hand, if $\Delta_{ab} \geq 0$ for all $a \in V, b \notin V$, accept the present set V as a near optimum set of training sites, and terminate.

Example 9.5.1

As an example, consider the problem with the following data. $n = 8$, $p = 2$, s_i = number of trainees from city i annually, m_i = number of trips by trainees from city i to training site annually [J. G. Klincewicz, 1980].

i	City	s_i	m_i	d_i = staying cost/trainee
1	Dallas	2	8	\$1800
2	Denver	3	12	1590
3	G. Falls	6	24	1290
4	L.A.	8	32	2100
5	Omaha	5	20	1560
6	St. Louis	4	16	1650
7	S.F.	7	28	2130
8	Seattle	1	4	1680

c_{ij} = travel cost/trip (symmetric)

to j	1	2	3	4	5	6	7	8
from $i = 1$	0							
2	170	0						
3	266	163	0					
4	262	197	223	0				
5	158	141	202	273	0			
6	152	191	260	318	115	0		
7	301	216	204	113	292	343	0	
8	333	227	146	217	282	340	172	0

We compute $r_{ij} = s_i d_j + m_i c_{ij}$ = total cost of trainees from city i to be trained at a training center in city j, and give them below.

	r_{ij}							
$j =$	1	2	3	4	5	6	7	8
$i = 1$	3600	4510	4708	6296	4384	4516	6668	6024
2	7440	4770	5826	8664	6372	6242	8982	7764
3	17184	13452	7740	17952	14208	16140	17676	13584
4	22784	19024	17456	16800	21216	23376	20656	20384
5	12160	10770	10490	15960	7800	10550	16490	14040
6	9632	9506	9800	13488	8080	6600	14128	12160
7	21028	17178	14742	17864	19096	21154	14910	16576
8	3132	2498	1874	2968	2688	3010	2818	1680

Suppose the initial set of sites for training centers is $V_1 = \{6, 8\}$. With this set of training centers, since $r_{16} = 4516 < r_{18} = 6024$, trainees from city 1 will be assigned to the training center at city 6, i.e., city 1 is in the market set for training center at city 6. In the same way, we find that the market set for the center at city 6 is $T_6 = \{1, 2, 5, 6\}$; and the market set for the center at city 8 is $T_8 = \{3, 4, 7, 8\}$.

If 8 in V_1 is replaced by 1, we verify that cities 3 and 8 in T_8 will join the market set of 6 after the change, but 4 and 7 will join the market set of 1. Also, city 1 will move from the market set of 6 to that of 1. So, $\Delta_{8,1} = (3600 - 4516) + (16140 - 13584) + (22784 - 20384) + (21028 - 16576) + (3010 - 1680) = 9822 > 0$. So, replacing 8 by 1 in V_1 only increases the total cost.

Similarly we compute $\Delta_{8,2} = -4048 < 0$. Thus replacing 8 by 2 in V_1 reduces the total cost by \$4048. So, we make the exchange and have the new set of sites for training centers $V_2 = \{6, 2\}$.

The algorithm can be continued with the new set V_2 in the same way. It will terminate when a set of sites for training centers which cannot be improved by such interchanges is obtained.

To get even better solutions, the procedure should be repeated with different initial sets, and the best of all the solutions obtained is taken as the output.

Practical experience indicates that the interchange heuristic discussed here for the training center location problem, and other p-

median type location problems similar to it, gives excellent results. In a computational experiment, this heuristic obtained solutions verified to be optimal by the B&B approach in 26 out of the 27 cases tested, and within 1% of the optimum cost in the other case.

In general, a composite heuristic approach consisting of something like a greedy method to generate one or more good initial solutions, and an interchange method to search for better solutions by local improvement beginning with the initial solutions produced by the first method, leads to reasonable solutions for large scale combinatorial optimization problems in applications.

Exercises

9.5.1: Formulate the problem of finding the best locations for training centers, and the assignment of cities to training centers, for the numerical example in Example 9.5.1 as a pure $0-1$ IP.

9.6 General Local Search Methods

The interchange heuristic methods discussed in Section 9.5 are special types of local search methods that have yielded excellent results on some types of combinatorial optimization problems. In this section, we will summarize the basic principles behind local search methods.

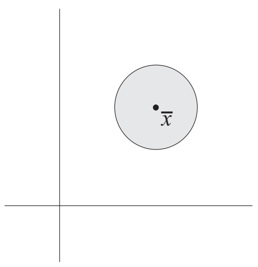
Some Classical Concepts from Nonlinear Programming

The classical concepts of **neighborhood of a feasible solution**, **local optimum** (**minimum or maximum**) have been developed in continuous variable nonlinear programming many ceuturies ago, for dealing with the problem of minimizing (maximizing) a real valued function, say $f(x)$, over R^n (this is the classical problem of finding the unconstrained optimum of $f(x)$ over R^n). For this problem, given an $\bar{x} \in R^n$,

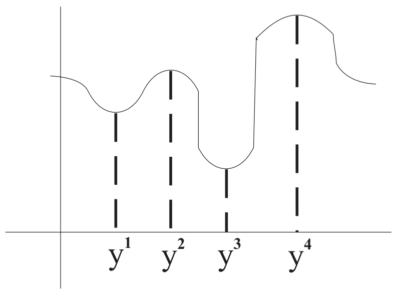
a **neighborhood** of $\bar{x}$ is defined to be the set of all $x \in R^n$

satisfying $||x - \bar{x}|| < \epsilon$ for some $\epsilon > 0$, where for any $y = (y_1, \ldots, y_n)^T \in R^n$, $||y|| = \sqrt{y_1^2 + \ldots + y_n^2}$ is the Euclidean norm of y (i.e., the Euclidean disctance between 0 and y).

$\bar{x}$ is said to be a **local minimum** for $f(x)$ if for some $\epsilon > 0$, $\bar{x}$ is the global minimum for $f(x)$ in the neighborhood $\{x : ||x - \bar{x}|| < \epsilon\}$; i.e., $f(x) \geq f(\bar{x})$ for all x in this neighborhood. See Figure 9.5.



For n = 2 (i.e., in R^2) the shaded area inside the circle, with $\bar{x}$ as center and a small radius, is a neighborhood of $\bar{x}$ unconstrained optimization in continuous optimization.



Plot of a function f(y) defined for y ∈ R^1. The points y^1 and y^2 are local minima; and y^3 and y^4 are local maxima for f(y).

Figure 9.5:

The concept of a local maximum is defined similarly if the original problem is to maximize $f(x)$ over R^n. But we will continue our discussion in terms of minimization, since maximizing $f(x)$ is the same as minimizing $-f(x)$ subject to the same constraints if any.

Neighborhood Structures for Combinatorial Optimization Problems

The concepts of neighborhood of a feasible solution and local minimum have been extended recently to the setting of combinatorial (discrete) optimization problems, as a prelude to developing very effective **local search** (also called **neighborhood search**) algorithms for handling such problems arising in many application domains.

For continuous optimization over R^n, the set $\{x : ||x - \bar{x}|| < \epsilon\}$ for $\epsilon > 0$ offers a very natural *neighborhood* for $\bar{x} \in R^n$. Combinatorial optimization problems form a wide variety, and historically there has not been a concept of the neighborhood of a feasible solution in them. Now many different neighborhood structures have been defined for combinatorial optimization problems, and neighborhood search algorithms have been developed using them. These algorithms have now become important tools for handling these problems.

For a problem, once a neighborhood structure is selected, a solution $\bar{x}$ is defined to be a **local minimum (optimum)** with respect to this neighborhood structure if $\bar{x}$ is the best solution in its neighborhood $N(\bar{x})$ (i.e., a locally optimal solution is one that does not have a better neighbor). The local (neighborhood) search algorithm proceeds this way to solve the problem.

Step 1: Initial solution: Select an initial solution. This may be constructed by a different algorithm, or generated randomly.

Step 2: Neighborhood search: Let $\bar{x}$ be the current solution. Search for a better solution than $\bar{x}$ in $N(\bar{x})$, the neighborhood of $\bar{x}$.

If no such solution is found in $N(\bar{x})$, terminate with $\bar{x}$ as a locally optimal solution.

If a better solution than $\bar{x}$ is found in $N(\bar{x})$, replace $\bar{x}$ with it as the current solution, and repeat this Step 2 with it.

Step 3: Running the Algorithm with Different starting Points: In using local search methods to solve a problem, one generally performs many runs of it with different starting points, and selects the best solution obtained in all these runs as the output of the approach.

As an example consider a symmetric TSP. the 2-opt (3-opt) interchange heuristics for it discussed earlier are local search methods in which the neighborhood of a tour τ is the set of all tours that differ from τ in exactly two (or three) arecs or edges.

Small or Large Neighborhoods

The choice of the neighborhood structure, (i.e., how the neighborhood of a feasible solution is defined), plays a critical role in the quality of solutions produced by the neighborhood search approach. Most of the neighborhood search algorithms in the literature use small neighborhoods (i.e., those with small number of solutions in them) as they explicitly enumerate and evaluate all neighbors to find a better solution than $\bar{x}$ in the neighborhood $N(\bar{x})$ of $\bar{x}$.

However, intutively it seems natural to expect that the larger the neighborhood, the better the quality of the final local optimum solution obtained. But if the search for a better solution in the neighborhood is carried out by enumeration, the larger the neighborhood, the longer it takes to search the neighborhood in each iteration. So, larger neighborhoods become practical only if one can search them for a better solution using an efficient algorithm. For a few specialized combinatorial optimization problems, techniques based on efficient search algorithms that can identify an improved neighbor without explicitly enumerating and evaluating all the neighbors have been developed. For these problems we have local search methods using very large neighborhoods.

Types of Combinatorial Optimization Problems Solved by Local Search

Local search methods are usually applied to solve unconstrained combinatorial optimization problems (i.e., those for which a feasible solution can be generated easily), and in fact all feasible solutions can be enumerated one by one easily. Examples are the unconstrained traveling salesman problem (TSP), for which the feasible solutions are all the tours, etc.

If the problem is a constrained combinatorial optimization problem, the constraints may be simple ones that can be handled easily, or hard constraints that make even the problem of finding a feasible solution a hard problem.

For an example of a constrained problem subject to hard constraints, consider a constrained TSP with time windows. In this problem, in addition to the cost matrix (c_{ij}), the travel time matrix (t_{ij}) where t_{ij} = time to travel from city i to city j directly, is given, ‘ the starting city (origin of the tour) is specified, say city 1; and for each city other than the origin, the time windows during which the salesman must arrive there, are specified. For example, if city 1 is the origin, and the time window for city 2 is 10 to 12 hours; this requires that the salesman must arrive at city 2 between 10 to 12 hours after starting at city 1. Even finding a feasible solution for this problem is hard; and it is very difficult to apply local search to handle this problem. One way of handling such a problem is to relax the constraints and include a penalty term for their violation in the objective function to be minimized. But this makes local search messy, and since it cannot guarantee that the penalty term can be made zero, it may not even produce a feasible solution for the problem. We will not consider the application of local search to solve such constrained problems subject to hard constraints in this book.

If the problem to be solved is a constrained problem in which the constraints are simple (i.e., it is easy to find a feasible solution for it, and in fact all feasible solutions for it can be enumerated easily one by one, if necessary), there are two ways in which one can proceed to apply local search to solve it. These are:

Exclusion: Infeasible solutions are always discarded, and only feasible solutions are kept. That is, the set of neighbors of a feasible solution is taken as the set of neighbors defined for the corresponding unconstrained problem after discarding the infeasible solutions in it.

Repair: The local search method is applied on the corresponding unconstrained problem, ignoring the constraints. If the method yields an infeasible solution, it is changed or repaired so that it becomes feasible. This requires a method to find a feasible solution that retains the essential characeristics of the infeasible solution which has been produced. Often this is taken as the best feasible solution in the neighborhood of that infeasible solution, if the computational expense of finding it is not too much.

General Issues

To apply local search to a problem, a number of choices must be made.

First we must decide how to choose the initial solution to apply the algorithm. Sometimes, people use another constructive heuristic like the greedy heuristic to obtaian the initial feasible solution for local search. Often, local search is executed from several different randomly chosen starting points, and then the best result from all the runs is selected for implementing.

Next we must choose a neighborhood structure for the problem, and a method for searching the neighbor hood of a point. This is a major challenge for getting good results from local search. Since local search is applied on a very diverse set of problems, and each application has its own peculiarities and difficulties to be overcome, making this choice depends on the specific problem being solved. Illustrating the use of large neighborhoods developed for some problems, and discussing the efficient methods used to search them is beyond the scope of this book. So, we will discuss the application of local search on some problems using small neighborhoods and enumeration to search them and refer the interested reader to more advanced references for other developments.

Examples of Application of Local search

1. Partitioning Problems: In general these problems have the following features: We are given a set $A = \{a_1, \ldots, a_n\}$ of elements which is required to be partitioned into k subsets $S = \{S_1, \ldots, S_k\}$ (S is said to be a partition of A; if $\cup_{i=1}^{k} S_i = A$, and $S_i \cap S_j = \emptyset$ for all $i \neq j$). $c_i(S_i)$ is the cost of forming the elements in the set S_i as a set in the partition; and the total cost $z = \sum_{i=1}^{k} c_i(S_i)$ is to be minimized. Usually the statement of the problem includes a procedure for computing $c_i(S_i)$ for any given set of elements S_i. Many combinatorial optimization problems belong to this partitioning framework.

For example, the task allocation problem discussed in Example 9.2.1

becomes a partitioning problem when it is viewed as the problem of allocating tasks in $\{1, \ldots, n\}$ to the T processors; i.e., one of partitioning the set of tasks $\{1, \ldots, n\}$ into $S = \{S_1, \ldots, S_n\}$ where for $i = 1$ to T, S_i is the set of tasks allocated to the ith processor. Then

$$c_i(S_i) = \begin{cases} 0, \text{ if } S_i = \emptyset \\ \rho_i + \sum_{j \in S_i} \sum_{p \notin S_i} (Lc_{jp} + Hd_{jp}) \text{ otherwise.} \end{cases}$$

Delivery and routing problems discussed in Section 7.3 are essentially partitioning problems in which the set of all customers to whom deliveries have to be made is to be partitioned into $S = \{S_1, \ldots, S_n\}$, where S_i is the set of customers handled by a single truck. Given S_i, $c_i(S_i)$ is the total mileage of the truck to start from the depot, cover all the customers in S_i in an optimum order, and then return to the depot.

The set partitioning problem and its various applications discussed in Section 7.3 can be seen to belong to the partitioning framework directly.

The training center location problem discussed in Section 9.5 is actually the problem of partitioning the set $\{1, \ldots, n\}$ of cities where offices of the company are located, into subsets $\{S_1, \ldots, S_k\}$ where each S_i is a set of cities from which employees will all train at the same training center. Given S_i, $c_i(S_i)$ is the cost of training all the employees from cities in S_i at a single center located optimally within S_i.

Neighborhood Structures Commonly Used to Solve Partitioning Problems Using Local Search

Perhaps the simplest and most popular neighborhood for partitioning problems is the **two-exchange neioghborhood**. In this, the neighbors of a given partition $S = \{S_1, \ldots, S_k\}$ are all the partitions obtained by transferring single elements between two different subsets in S; i.e., partitions of the form $\{S_1, \ldots, S_{i-1}, ((S_i \backslash \{a_{i1}\}) \cup \{a_{j1}\}), S_{i+1}, \ldots, S_{j-1}, ((S_j \backslash \{a_{j1}\}) \cup \{a_{i1}\}), S_{j+1}, \ldots, S_k\}$, where a_{i1} is an element in S_i and a_{j1} is an element in S_j.

As a numerical example, suppose $n = 8$, and consider the partition $S = \{S_1, S_2, S_3\}$ where $S_1 = \{1, 2\}$, $S_2 = \{3, 4, 5\}$, $S_3 = \{6, 7, 8\}$.

Exchanging the element 2 in S_1 with the element 6 in S_3, we get the neighbor partition $S' = \{\{1,6\},\{3,4,5\},\{2,7,8\}$.

If n is the number of elements in the original set, in this neighborhood structure each solution (i.e., partition) has about $O(n^2)$ neighbors. So, it is computationally feasible to identify the best solution in the neighborhood of a solution by explicitly examining the entire neighborhood.

On these problems, the local search algorithm starts off with an initial partition. In each step it searches the neighborhood of the current partition for another of lower cost. If none found, the current partition is a local optimum and the method terminates. If a better partition is found in the neighborhood, it replaces the current partition and the search continues.

2. The Traveling Salesman Problem (TSP): The most famous local search algorithms for the TSP are the 2-opt and 3-opt discussed in Section 9.5.

The neighborhood of a tour τ used in 2-opt for the symmetric TSP is the 2-change (or the 2-exchange, or 2-interchange) neighborhood, it is the set of all tours that differ from τ in exactly two edges as explained in Section 9.5.

The neighborhood of a tour τ used in 3-opt for the asymmetric or symmetric TSP is the 3-change (or the 3-exchange, or 3-interchange) neighborhood, that consists of all the tours that differ from τ in at most three arcs or edges.

With these neighbors, the local search algorithm is exactly the interchange heuristic method for the TSP discussed in Section 9.5.

3. Sequencing Problems: Many scheduling problems in computer, manufacturing, and other systems deal with efficient allocation of one or more resources to activities over time.

Using the terminology from manufacturing, in these problems we need to perform a set of jobs, each may require operations on some machines which are the resources that can perform at most one activity at a time. Many such machine scheduling problems deal with the problem of finding an optimal order, or sequence, in which the jobs are to be processed. Given the processing sequence for the jobs, the cost corresponding to that sequence can easily be computed. The problem is to find the optimal sequence that minimizes the cost.

As an example, suppose there are $n = 3$ jobs, J_1, J_2, J_3, each of which needs to be processed on two machines M_1, M_2; on M_1 first and then on M_2 (this type of problem is called a **flowshop scheduling problem** in the literature). The processing times of the jobs on the machines are given below.

Job	Units of processing time on	
	M_1	M_2
J_1	2	1
J_2	3	1
J_3	2	3

Assuming that the processing of this set of jobs begins at time 0, let t_i denote the time at which the processing of J_i is finished on M_2. Then the criterion to be optimized in these problems is usually a function of $(t_1, \dots, t_n)$ (here $n = 3$); for example the total processing duration $=$ $\max\{t_1, \dots, t_n\}$, or sum finishing time $= \sum_{i=1}^{n} t_i$, etc. These objective functions depend on the order in which the jobs are processed.

For example, if the jobs are processed in the order (J_1, J_2, J_3), it can be verified from Figure 9.6 that $(t_1, t_2, t_3) = (3, 6, 10)$. So the total processing duration under this order is 10, the sum finishing time is 19. Figure 9.6 indicates the jobs being processed on the two machines in the various time units from 1 to 10.

Time $\to$ 0	1	2	3	4	5	6	7	8	9	10
M_1	J_1	J_1	J_2	J_2	J_2	J_3	J_3			
M_2			J_1			J_2		J_3	J_3	J_3

Figure 9.6: Processing order is (J_1, J_2, J_3). Blank entry for a machine in a time period indicates that the machine is idle during that period. Verify that processing of J_1, J_2, J_3 is finished at the end of times units 3, 6, 10 respectively.

Local Search for Sequencing Problems

Among the small neighborhoods used for solving these problems, there are four, in each the neighbors of a sequence S are all the sequences that can be obtained by carrying out a specified operation once. These operations for the four different neighborhoods are listed below, and we illustrate each by considering a typical neighbor of a sequence $S_1 = (A, B, C, D, E, F, G, H)$ of eight jobs labeled A to H.

Transpose: Swap (or interchange) two adjacent jobs in the sequence. Thus (A, B, D, C, E, F, G, H) is a neighbor of S_1 in the neighborhood defined by this operation.

Insert (or **Shift**)**:** Remove a job from one position in the sequence and insert it at another position (either before or after the original position). Thus (A, E, B, C, D, F, G, H) and (A, B, C, D, F, G, E, H) are both neighbors of S_1 in the neighborhood defined by this operation.

Swap: Swap two jobs that may not be adjacent. Thus (A, F, C, D, E, B, G, H) is a neighbor of S_1 in the neighborhood defined by this operation.

Block Insert: Move a subsequence of jobs from one position in the sequence, and insert that subsequence in another position.

There are of course more complex neighborhoods. The neighborhood defined by transpose has $(n - 1)$ neighbors, that by insert has $(n - 1)^2$ neighbors, that by swap has $n(n - 1)/2$ neighbors, and that by block insert has $n(n + 1)(n - 1)/6$ neighbors.

Once a neighborhood structure is selected, the local search method begins with an initial sequence as the current sequence, and in each step replaces it by a better neighbor until a local optimum is reached.

Summary

In this section we discussed the basic principles behind designing local search methods for combinatorial optimization problems, and illustrated them with some examples.

9.7 Simulated Annealing

Simulated annealing (SA) is a type of local search heuristic involving some random elements in the way the algorithm proceeds.

For a problem in which the objective function is to be minimized, the simplest form of local search is a **descent method** that starts with an initial solution. The method should have a mechanism for generating a neighbor of the current solution. If the generated neighbor has a smaller objective value, it becomes the new current solution, otherwise the current solution is retained. The process is repeated until a solution is reached with no possibility of improvement in its neighborhood, such a point is a **local minimum**, and the descent method terminates. This is one of the disadvantages of simple local search methods. By requiring that the iterative steps move only downhill on the objective function surface, they may get stuck at a local minimum which may be far away from any global minimum. Simple local search methods try to avoid this difficulty by running the descent method several times, starting from different initial solutions, and finally taking the best of the local minima found.

On the other hand, SA avoids getting trapped at a local minimum by sometimes accepting a neighborhood move that increases the objective value, using a probabilistic acceptance criterion. These uphill moves make it possible to move away from local minima and explore the feasible region in its entirety. In the course of an SA algorithm, the probability of accepting such uphill moves slowly decreases to 0.

The motivation for the SA algorithm, and its name, come from an analogy with a highly successful Monte Carlo simulation model for the physical annealing process of finding low energy states of a solid. Physical annealing is the process of finding the ground state of a solid which corresponds to the minimum energy configuration, by initially melting the substance, and then lowering the temperature slowly, spending a long time at temperatures close to the freezing point. Metropolis et al. [1953] introduced the simple Monte Carlo simulation algorithm that modeled the physical annealing process very successfully. At each iteration of this algorithm, the system is given a small displacement, and the resulting change δ in the energy of the system is calculated.

If $\delta < 0$, the resulting change is accepted, but if $\delta > 0$ the change is accepted with probability $\exp(-\delta/T)$ where T is a constant times the temperature, which we will refer to as the temperature. If a large number of iterations are carried out at each temperature, the model finds the thermal equilibrium that the system attains at that temperature. Simulating the transition to the equilibrium, and decreasing the temperature, one can find states of the system with smaller and smaller values of mean energy.

By first melting the model system at a high effective temperature, and then lowering the temperature in slow deliberate steps after waiting for equilibrium to be established at each temperature, one has in effect performed a simulated annealing procedure. Experimentally it is precisely such annealing that has the best chance of bringing a solid to a good approximation of its true ground state rather than freezing it into a metastable configuration that corresponds to a local but not global minimum energy level. The sequence of temperatures used, the number of rearrangements attempted to reach equilibrium at each temperature, and the criterion used for stopping, are collectively known as the **cooling or annealing schedule**.

In the analogy, the different feasible solutions of a combinatorial optimization problem correspond to the different states of the substance. The objective function to be minimized corresponds to the energy of the system. However, the concept of temperature in the physical system has no obvious equivalent in combinatorial optimization problems. In SA algorithms for optimization, this temperature is simply a control parameter in the same units as the cost function. The probability of accepting an uphill move which causes an increase $\delta > 0$ in the objective function, $\exp(-\delta/T)$, is called the **acceptance function**. This acceptance function implies that small increases in the objective function are more likely to be accepted than large increases, and that when T is high, most moves will be accepted; but as T approaches 0, most uphill moves will be rejected. So, in SA, the algorithm is started with a high value of T to avoid being permanently trapped at a local minimum. The algorithm drops the temperature parameter gradually, making a certain number of neighborhood moves at each temperature.

The simple local search method that accepts only rearrangements that lower the cost function, corresponds to extremely rapid quenching where the temperature is reduced quickly, so it should not be surprising that the resulting solutions are usually metastable. SA provides a generalization of iterative improvement in which controlled uphill moves are incorporated in the search for a better solution. This helps to attain some of the speed and reliability of descent algorithms while avoiding their propensity to stick at local minima.

Let $\mathbf{X}$ denote the set of feasible solutions of a combinatorial optimization problem, and $z(x)$ the objective function to be minimized over $\mathbf{X}$. $|\mathbf{X}|$ is exponentially large in terms of the natural measure of the size of the problem: For example, in the TSP of order n, $|\mathbf{X}| = (n-1)!$. To apply SA on this problem we need to define a **neighborhood** for each $x \in \mathbf{X}$. The essential feature of these neighborhoods is: from any point in $\mathbf{X}$ we should be able to reach any other point in $\mathbf{X}$ by a path consisting of moves from a point to an adjacent point. Also, usually neighborhoods are symmetric, i.e., y is in the neighborhood of x iff x is in the neighborhood of y. The efficiency of SA depends on the neighborhood structure that is used.

If the problem is posed as one of finding an optimum sequence of a set of elements, it is convenient to incorporate any constraints on the desired sequence, in the objective function using appropriate penalty function terms corresponding to them. Then $\mathbf{X}$ becomes the set of all permutations of the elements. The neighbors of a sequence could be considered as all those that can be obtained by interchanging the elements in two positions, or those obtained by reversing the order of the elements in a segment of the sequence, etc. By designing neighborhoods taking advantage of the problem structure, the efficiency of the SA algorithm can be improved substantially.

We also need an $x^0 \in \mathbf{X}$ to initiate the algorithm, the initial value T_0 of the temperature parameter T, the decreasing sequence $T_t, t = 0$, 1, ... of values of temperature to be used, the number of iterations to be performed at each temperature (N_t at temperature T_t, $t = 0, 1, \ldots$), and a stopping criterion to terminate the algorithm. We also need a mechanism to select a solution y from the neighborhood of the current point x in each step of the algorithm. Once these choices are made, the algorithm proceeds as below.

GENERAL SA ALGORITHM

Initialization Let x^0, T_0 be the initial solution and temperature, respectively.

General Step When the temperature is T_t, do the following. Set iteration counter n to 0. If x^i is the current solution, find a solution y in the neighborhood of x^i at random. If $z(y) \leq z(x^i)$ make $x^{i+1} = y$. If $z(y) > z(x^i)$ make

$$x^{i+1} = \begin{cases} y & \text{with probability } \exp\Big(-\frac{z(y)-z(x^i))}{T_t}\Big) \\ x^i & \text{with probability } 1 - \exp\Big(-\frac{z(y)-z(x^i)}{T_t}\Big) \end{cases}$$

Increase the iteration count n by 1 and continue with x^{i+1} as the current solution. When $n = N_t$, change T to T_{t+1} and start the next step.

Continue until the stopping criterion is met.

Discussion

The cooling schedule may be developed by trial and error for a given problem, but a great variety of cooling schedules have now been suggested. $T_{t+1} = \alpha T_t$ where α is a number between 0.8 to 0.99 is sometimes used, with N_t being determined as a sufficient number of iterations subject to a constant upper bound. The cooling schedule $T_t = d/\log t$ where d is some positive constant, is also quite popular.

As an example we consider the TSP of order n with $c = (c_{ij})$ as the cost matrix. We will represent the tour $x = p_1, p_2, \ldots, p_n; p_1$ by the permutation $p_1, p_2, \ldots, p_n$, and its cost is $z(x) = \sum_{r=1}^{n-1} c_{p_r,p_{r+1}} + c_{p_n,p_1}$.

We take the neighborhood of a tour to be the set of all tours corresponding to permutations obtained by selecting a pair of positions in its permutation and reversing the segment between them. For example, consider $n = 7$, and the tour x^0 corresponding to the permutation 6, 3, 7, 2, 5, 4, 1. The tour x^1 corresponding to the permutation 6, 4, 5, 2, 7, 3, 1 is obtained by reversing the segment between positions 1 and 6 in the permutation for x^0; it is a neighbor of x^0.

We now describe the various steps in the SA algorithm for the TSP based on this definition of neighborhood. Here the symbol n denotes the number of cities, i.e., the order of the TSP. We take N_t, a target for the number of iterations to be performed at temperature T_t, to be n for all t. The actual value of N_t used may be more than n depending on the observed performance during the algorithm. We use the symbol i as an iteration counter, and also use it in defining the neighborhood of the current tour from which the next tour will be selected.

AN SA ALGORITHM FOR THE TSP

Step 1 Select the initial permutation $x^0 = p_1^0, \ldots, p_n^0$ and initial temperature T_0.

Step 2 Let $x = p_1, \ldots, p_n$ be the present permutation, $z(x)$ the cost of the corresponding tour, and T the current temperature.

Step 3 Set $i = 1$.

Step 4 Let $x = p_1, \ldots, p_n$ be the present permutation. Select an integer $j \neq i$ between 1 to n at random. Define $a = \min\{i, j\}, b = \max\{i, j\}$. Define y to be the permutation obtained by reversing the segment between a and b in the present permutation x, and $z(y)$ the cost of the tour corresponding to y.

If $z(y) \leq z(x)$ accept y as the new current permutation. If $z(y) > z(x)$, let the new current permutation be:

$$\begin{cases} y & \text{with probability } \exp\Big(-\frac{z(y)-z(x)}{T}\Big) \\ x & \text{with probability } 1 - \exp\Big(-\frac{z(y)-z(x)}{T}\Big) \end{cases}$$

where T is the current temperature. Go to Step 5.

Step 5 If $i < n$, increase it by 1 and go back to Step 4. If $i = n$ and enough number of iterations have been performed at the current temperature, go to Step 6; otherwise, go to Step 3.

Step 6 If the temperature has reached the smallest value, terminate with the best tour obtained so far. Otherwise, change the temperature to the next value in the temperature sequence and go back to Step 2.

One has to repeat the iterations at each temperature until an equilibrium seems to have been reached. Then the temperature is decreased and the process repeated. Repeating this, solutions of improved cost will result, and one has to decide suitable stopping criteria. In the end, one can perform a deterministic local search beginning with the best solution obtained in the algorithm and continue as long as better solutions are found.

The attraction of SA is that it is general, yet simple to apply. Solving a problem with it requires a neighborhood structure to be specified and a procedure for generating neighbors of solution points at random. Researchers are using SA extensively on various problems and obtaining good results.

9.8 Genetic Algorithms

Inspired by biological systems that adapt to the environment and evolve into highly successful organisms over many generations, J. Holland [1975] proposed heuristic search methods for hard combinatorial optimization problems based on operations called **mating, reproduction, cloning, crossover**, and **mutation**; these are patterned upon biological activities bearing the same names. Hence these methods are appropriately called **genetic algorithms** (GA). GAs are robust and effective iterative adaptive search algorithms with some of the creativity of human reasoning.

The first step to develop a GA for an optimization problem is to represent it so that every solution for it is in the form of a string of bits (integers or characters), all of them consisting of the same number of elements, say n. Each candidate solution represented as a string is known as an **organism** or a **chromosome**. So each chromosome is a bit string of length n. The variable in a position on the chromosome is called the **gene** at that position, and its value in a particular chromosome is called its **allele** in that chromosome. For example, if $n = 3$, a general chromosome is $x = (x_1, x_2, x_3)$ where x_1, x_2, and x_3 are the genes on this chromosome in the three positions. In the chromosome (3, 8, 9), the second gene has allele 8.

We will discuss GAs as they apply to minimization problems. We assume that the objective function value at every chromosome is positive; this can be arranged by adding a suitable positive constant to the objective function value of every chromosome, if necessary.

To develop a GA, the problem has to be transformed into an unconstrained optimization problem so that every string of length n can be looked upon as a solution vector for the problem. For this purpose, a penalty function, consisting of nonnegative penalty terms corresponding to each constraint in the original problem, is constructed. Each penalty term is always 0 at every point satisfying it, and positive at every point violating it. So, the penalty function has value 0 at every feasible solution to the original problem, and a positive value at every infeasible point. Also, the value of the penalty function at an infeasible point increases rapidly as the point moves farther away from the feasible region. The construction of the penalty function is illustrated later with an example. The **fitness measure** is defined to be the objective function plus the penalty function. It is also called the **evaluation function**. Thus at every feasible solution to the original problem, the fitness measure is equal to the objective function value at that point. Hence associated with each chromosome is an objective function value and a fitness measure. From the way the fitness measure is defined, among two points the one with a smaller fitness measure is better than the other.

GAs start with an initial population of likely problem solutions, and evolve towards better solutions. The population changes over time, but always has the same number of members. New solutions are generated through operations resembling reproductive processes observed in nature. To evolve towards better solutions, it is necessary to reject the worst solutions and only allow the best ones to survive and reproduce. This incorporates nature's law of survival of the fittest which only allows organisms that adapt best to the environment to thrive. When applying GA to a minimization problem, the role of the environment is played by the fitness measure, the degree of adaptation of a solution point to the environment is interpreted as getting better as its fitness measure decreases. In successive generations, solutions improve until the best in the population is near-optimal.

We will now discuss the essential components for applying a GA on a minimization problem. After each component is discussed, we show how it applies on two problems; one is the TSP, and the other the task allocation problem modeled in Example 9.2.1.

Genetic representation of solutions As mentioned above, the problem is transformed and represented in such a way that every solution can be represented by a string of bits. All strings corresponding to solution vectors of the problem contain the same number of bits, say n.

For some problems, developing this representation may be a nontrivial effort requiring careful thought, but for many others, a natural representation is usually available. For example, if the problem is one of finding an optimum sequence for n elements numbered 1 to n, every solution is a permutation of $\{1, \ldots, n\}$. In this case the permutation of $\{1, \ldots, n\}$ provides a string representation for solutions to the problem. Valid strings are those which are permutations of $\{1, \ldots, n\}$; i.e., strings in which each of the symbols $1, \ldots, n$ appears once and only once.

For the TSP involving cities 1 to n; a tour $i_1, i_2, \ldots, i_n; i_1$ can be represented by the permutation $i_1, \ldots, i_n$ (i.e., the sequence of cities in the order in which they are visited). So, here again, valid strings are those in which each of the symbols $1, \ldots, n$ appears once and only once.

For the task allocation problem involving the allocation of n tasks to T processors discussed in Example 9.2.1, a solution can be represented as a string of n numbers $x_1, \ldots, x_n$ where for each $j =$ 1 to n, x_j is the number of the processor to which task j is allotted. So, here valid strings are all sequences of the form $x_1, \ldots, x_n$ where each x_j is an integer between 1 to T. As an example, if the number of tasks $n = 6$, and the number of processors $T = 4$, the string 1, 1, 3, 2, 1, 3 is a valid chromosome. It represents allocating tasks 1, 2, 5 to processor 1; task 4 to processor 2; and tasks 3, 6 to processor 3; and not using processor 4 at all.

Developing evaluation function We assume that the objective function to be minimized has a positive value at every solution. GAs deal with a relaxed version of the problem in an unconstrained form, to allow the search to be carried out among all valid strings. The evaluation function or the fitness measure of any valid chromosome is the sum of its objective function value and that of the penalty function providing an infeasibility measure of the corresponding solution to the constraints in the original problem. GAs use the evaluation function value at a chromosome to verify its degree of fitness to the environment, and to lower the probability allotted to undesirable chromosomes to survive and to reproduce. This evolutionary aspect of the algorithm provides for the elimination of trial solutions that are relatively unsuccessful. Hence the choice of the evaluation function has a great influence on the overall performance of the algorithm.

For the TSP involving n cities and positive cost matrix (c_{ij}), when solutions are represented by permutations of $\{1, \ldots, n\}$, the evaluation function value of a chromosome can be taken to be the cost of the corresponding tour which is $\sum_{j=1}^{n-1} c_{x_j, x_{j+1}} + c_{x_n, x_1}$. There are no penalty terms needed here as every valid string corresponds to a feasible tour.

Now consider the task allocation problem involving the allocation of n tasks to T processors discussed in Example 11.1.1. As discussed above, we represent each solution by a string $x_1, \ldots, x_n$ where x_j is the number of the processor to which task j is allotted. A string $x_1, \ldots, x_n$ is valid if x_j is an integer between 1 to T for all j. A valid string $x = x_1, \ldots, x_n$ is infeasible to the problem if either (i) it allots a processor more tasks than it can handle, or (ii) if the sum of the KOP requirements of tasks assigned to a processor exceeds its throughput capacity. So, to represent this problem in an unconstrained fashion, we need two penalty terms, one for each of the above types of infeasibility. We can use **quadratic penalty functions** in which

Penalty for exceeding the capacity on no. of tasks allotted = δ_1(excess no. of tasks)^2

Penalty for exceeding throughput capacity = δ_2(excess throughput over capacity)^2

where δ_1, and δ_2 are appropriate positive penalty coefficients. Such penalty terms are commonly used, as they seem to produce good results. We get the total penalty function value at x by summing the above penalties for each processor for which infeasibility of type (i) or (ii) mentioned above, or both, occur in x.

The objective function value corresponding to this string x is the sum of the costs of the processors used plus the sum of the costs of data link capacity needed by various pairs of tasks allotted to different processors in it. And the evaluation function for x is the sum of the objective value and the penalty function value at x. As an illustration, we will now provide a numerical example to show how to compute the evaluation function in this problem.

Consider the instance with number of tasks $n = 6$, number of processors $T = 4$, and the following data:

Processor t	Cost ρ_t	Max. no. tasks β_t	Throughput capacity γ_t
1	40	1	425 KOP
2	30	3	300
3	20	1	350
4	45	2	500

Task i	1	2	3	4	5	6
Throughput requirement a_i	150	150	250	250	150	80

	Cost of data link capacity for task pair i, j if allotted to different processors (symmetric)					
$j =$	1	2	3	4	5	6
$i = 1$	$\times$	2	3	1	5	7
2		$\times$	4	2	3	2
3			$\times$	5	2	4
4				$\times$	4	1
5					$\times$	3
6						$\times$

Consider the string $x = 1, 1, 3, 2, 1, 3$ discussed above. It allots tasks 1, 2, 5 to processor 1 with a total throughput requirement of $150 + 150 + 150 = 450$ KOP, exceeding the throughput capacity of 425 KOP of this processor. Also, the number of tasks allotted to this processor, 3, exceeds the capacity of 1 task that it can handle. So the penalty terms for processor 1 total to $\delta_1(3 - 1)^2 + \delta_2(450 - 425)^2 = 4\delta_1 + 625\delta_2$. Similarly, x allotted only task 4 with a throughput requirement of 150 KOP to processor 2, this is within the specified capacity of this processor, so there is no penalty from processor 2 for x. x has allotted tasks 3, 6 with a total throughput requirement of $250 + 80 = 330$ KOP to processor 3 which has a throughput capacity of 350 KOP, but it can handle only 1 task. So, the penalty from processor 3 for x is $\delta_1(2 - 1)^2 = \delta_1$. And processor 4 is not used. Hence, the overall penalty function value at x is $5\delta_1 + 625\delta_2$.

The objective function value at x is the sum of the costs of the processors used + the data link costs. Since tasks 1, 2, 5 are allotted to processor 1; task 4 to processor 2; and tasks 3, 6 to processor 3; data link costs are incurred for the pairs of tasks (1, 4), (1, 3), (1, 6), (2, 4), (2, 3), (2, 6), (5, 4), (5, 3), (5, 6), (4, 3), (4, 6) allotted to different processors. Thus, the objective function value at x is $(40 + 30 + 20) + (1 + 3 + 7 + 2 + 4 + 2 + 4 + 2 + 3 + 5 + 1) = 124$.

So, the overall evaluation function value at x is $124 + 5\delta_1 + 625\delta_2$. Given appropriate positive values to the penalty coefficients δ_1 and δ_2, this fitness measure can be computed.

Initial population An initial population of solutions is created usually randomly. In some applications, the initial population is generated by using some other method.

The population size is maintained constant through successive generations. It is usually 40 to 250 or larger, depending on the size of the problem being solved.

Developing genetic operators, reproduction, cloning, crossover, and mutation A GA evaluates a population and generates a new one iteratively. Each successive population is called a **generation**.

Individuals in the population are selected for survival into the next generation, or for mating, according to certain probabilities. This probability is increased as the individuals fitness measure gets better. In our case smaller values of the evaluation function are more desirable, so we make the probability of selection of a chromosome to be inversely proportional to its evaluation function value. Through this artificial evolution, GAs seek to breed solutions that are highly fit (i.e., optimal or near-optimal).

A certain percentage (typically between 10% to 40%) of the chromosomes in the population are usually copied as they are into the next generation. There are two possible ways (called **reproduction** and **cloning** or **clonal propagation**) for selecting these individuals. We discuss them below.

Reproduction This operation is probabilistic; it selects individuals from the current population according to probabilities inversely proportional to their evaluation function value as discussed above, and copies the selected individuals into the next generation. The process is repeated until the required number of individuals are selected.

Cloning This operation is deterministic. It selects the required number of individuals who have the best values for the evaluation function in the current population, and copies them as they are into the next generation. It is an elitist type of strategy. The advantage of using cloning over reproduction is that the best solution is monotonically improving from one generation to another.

A majority of the remaining individuals in the next population are generated by **mating**, and a small percentage by **mutation**. We discuss mating first. Two parent chromosomes are selected probabilistically as described above, from the current population, to mate. The mating operation is called **crossover**. It creates children whose genetic material resembles the parents genes in some fashion. Many different crossover mechanisms have been developed. We describe some of them.

One-point crossover This operation generates two children. Given parent chromosomes $x = x_1, \ldots, x_n$; $y = y_1, \ldots,$ y_n to mate; this operation selects a position called the

crossover point, r, between 1 to n at random. The two children are obtained by exchanging the blocks of alleles between positions r to n among the two parents. Thus the children are $c_1 = x_1, \ldots, x_{r-1}, y_r, \ldots, y_n$ and $c_2 = y_1, \ldots, y_{r-1},$ $x_r, \ldots, x_n$.

Now we have a choice between two possible strategies. Strategy 1 includes both the children in the next generation. Strategy 2 includes only the child with the best evaluation function value in the next generation, and discards the other.

Two-point crossover Given parent chromosomes $x = x_1, \ldots, x_n$; $y = y_1, \ldots, y_n$ to mate; this operation selects two positions $r < s$ between 1 to n at random, and swaps the blocks of alleles between positions r to s among the two parents, to get the two children. So, the two children are $c_1 = x_1, \ldots, x_{r-1}, y_r, \ldots, y_s, x_{s+1}, \ldots, x_n$, and $c_2 = y_1, \ldots, y_{r-1}, x_r, \ldots, x_s, y_{s+1}, \ldots, y_n$. Either both the children, or the best among them, get included in the next generation as discussed above.

Random crossover Given parent chromosomes $x = x_1, \ldots, x_n$; and $y = y_1, \ldots, y_n$; this operation creates children $u = u_1, \ldots, u_n$; and $v = v_1, \ldots, v_n$ where for $j = 1$ to n

$$u_j = \begin{cases} x_j & \text{with probability } \alpha \\ y_j & \text{with probability } 1 - \alpha \end{cases}$$

$$v_j = \begin{cases} y_j & \text{with probability } \alpha \\ x_j & \text{with probability } 1 - \alpha \end{cases}$$

for some preselected $0 < \alpha < 1$. Values of α between 0.5 to 0.8 are often used. Either both the children, or the best among them, get included in the next generation as discussed above.

In problems in which the order of the alleles in the chromosome has no significance, the above crossover operations produce valid child strings for the problem. For the task allocation problem with the representation discussed above, all the above crossover operations produce valid child strings.

However, for the TSP with each tour represented by a permutation of the cities, each of the above crossover operators may produce invalid child strings. As an example consider the two strings x = 4, 5, 2, 1, 3 and y = 1, 2, 4, 3, 5 for a 5-city TSP. With position 3 as the crossover point, the one-point crossover operator generates the children c_1 = 4, 5, 4, 3, 5 and c_2 = 1, 2, 2, 1, 3 both of which are invalid strings for this problem since neither of them is a permutation of $\{1, 2, 3, 4, 5\}$. So, for the TSP and for other problems in which solutions are represented by permutations of $\{1, \ldots, n\}$, the following custom designed crossover operator called **partially matched crossover operator** or **PMX** can be used.

PMX for permutation strings Let $x = x_1, \ldots, x_n$; and $y = y_1, \ldots, y_n$; be two parent permutations. Select two crossover positions $r < s$ randomly as in the two-point crossover operator. To get child 1, do the following for each $t = r$ to s in this order: if $x_t \neq y_t$ swap x_t and y_t in the permutation x. To get child 2, carry out exactly the same work on the permutation y instead of on x. It can be verified that both the children produced are permutations and hence valid strings for the problem.

As an example, consider $n = 6$, and the parents

$$\begin{array}{rcr|rrrr|l} x & = & 4, & 5, & 6, & 2, & 1, & 3 \\ y & = & 1, & 2, & 6, & 4, & 3, & 5 \end{array}$$

Suppose the crossover positions are 2 and 5 marked by bars above. Then child 1 is obtained by swapping 5 and 2, 2 and 4, and then 1 and 3, in x. As we carry these operations in this order x changes to 4, 2, 6, 5, 1, 3; then to 2, 4, 6, 5, 1, 3 and finally to $p =$ 2, 4, 6, 5, 3, 1. Carrying out the same operations on the permutation y we are lead to the second child $q =$ 3, 5, 6, 2, 1, 4. So, p, q are the children produced when this crossover operator is carried out with the parental pair x, y.

The **crossover ratio** (typically between 0.6 to 0.9) is the proportion of the next generation produced by crossover. The operation of mating pairs of randomly selected pairs of parents from the present population is continued until enough children to make up the next generation are produced.

Mutation Mutation makes random alterations, such as changing one or more randomly chosen genes, or swapping positions of two randomly selected bits, etc., on a randomly selected chromosome. The probability of mutation is usually set to be quite low (e.g., 0.001). A small percentage of the next generation is produced by applying the mutation operator on randomly selected chromosomes from the current population.

The processes of crossover and mutation are collectively referred to as **recombination operations**.

When all these operations are completed we have the new population which constitutes the next generation, and the whole process is repeated with it.

Stopping criterion The process of producing successive generations is usually continued until there is no improvement in the best solution for several generations, or until a predetermined number of generations have been simulated.

Usually one applies a local search heuristic beginning with the best solution in the final population, to make any possible final improvement. The solution obtained at the end of this process is the output of the algorithm.

Discussion

When a GA works well, the population quality gradually improves over the generations. After many generations, the best individual in the population is likely to be close to a global optimum of the underlying optimization problem.

As an example, we solved an instance of the task allocation problem discussed in Example 11.1.1 involving $n = 20$ tasks and $T = 7$ processors by the GA [A. Ben Hadj-Alouane, J. C. Bean, and K. G. Murty, 1999]. The representation discussed above for the problem was used. We maintained the population size at 50, with the initial population consisting of randomly generated solutions. In each generation, 10% of the population was obtained by cloning the best solutions in the previous population; 85% was obtained by mating using random crossover; and 5% was obtained by mutation. All the chromosomes in the initial population corresponded to infeasible solutions with infeasibility due to exceeding the throughput capacity on some processors, and due to allotting more than the number of tasks they can handle on some others.

The positive values given to the penalty coefficients had an effect on the performance of the algorithm. Starting small, their values were increased until infeasible solutions which are at the top of the population due to small penalty became highly penalized and are replaced with feasible solutions reasonably rapidly. When the penalty coefficients are large, and solutions at the top of the population are feasible, it turned out to be advantageous to decrease their values. Best results were obtained by adjusting the values of the penalty coefficients adaptively in this manner.

After 10 generations, the population had chromosomes corresponding to feasible solutions for the problem. After 110 generations the best chromosome in the population gave a solution to the problem which was considered to be very satisfactory. This solution was obtained in a few minutes of cpu time on an IBM RS/6000-320H workstation. The $0-1$ IP formulation of this problem given in Example 11.1.1, has about 2800 integer variables. We tried to solve this $0-1$ IP using the OSL software package based on B&B, on the same workstation. This program did not terminate even after running for 3 days continuously, when it was stopped. The best incumbent at that time was not better than the solution that GA found for this problem in a few minutes of cpu time.

In summary, the essential feature of GAs is that they search using a whole population of solutions rather than a single solution as other methods do. There are three essential requirements to apply a GA on a problem. First, the problem must be represented in such a way that every solution can be represented by a string of constant length. Second, a fitness measure to evaluate potential solutions needs to be developed. This measure is usually the sum of the objective function in the problem and of penalty terms corresponding to the violation of any of the constraints in the problem. Third, a suitable crossover operator has to be developed. The success of GA depends critically on these items, so they have to be developed very carefully.

The crossover operator can be designed in many different ways. In some problems, standard crossover operations may produce children strings which are invalid, as was shown for the case of the TSP. In such problems the crossover operation should be specialized and customized.

Without an appropriate representation and an effective crossover operator, genetic search may be slow and unrewarding. But with the appropriate representation and suitable genetic operators, it can produce high quality solutions very fast.

9.9 Heuristics for Graph Coloring

A **graph** $G = (\mathcal{N}, \mathcal{A})$ is defined by a set $\mathcal{N} = \{1, \ldots, n\}$ of **nodes** (also called **vertices**), and a set $\mathcal{A}$ of lines called **edges**, where each edge in $\mathcal{A}$ joins exactly a pair of nodes in $\mathcal{N}$ and has no orientation. If an edge joins nodes i and j, it is denoted by $(i; j)$. A pair of nodes in $\mathcal{N}$ are said to be **adjacent** if there is an edge joining them in $\mathcal{A}$. The **degree** of a node is the number of edges containg it.

Every subset $N \subset \mathcal{N}$ of nodes defines a **subgraph** of G, that subgraph is the graph (N, A) where A is the set of all edges in $\mathcal{A}$ that have both their nodes in the set N. In Figure 9.7 we show a graph with $\mathcal{A} = \{(1;2), (1;5), (1;4), (2;4), (2;3), (3;4), (3;5)\}$ consisting of 7 edges on the set of nodes $\mathcal{N} = \{1, 2, 3, 4, 5\}$ on the left, and its subgraph defined by the subset of nodes $N = \{1, 2, 4\}$ on the right. In the graph on the left, nodes 1, 2, 3, 4 have degree 3; and node 5 has degree 2.

In Section 7.9 we modeled the problem of avoiding conflicts in scheduling a set of meetings, as a graph coloring problem. The graph coloring problem on G is to color the nodes in $\mathcal{N}$ subject to the constraint that the pair of nodes on every edge in $\mathcal{A}$ get different colors, using the smallest number of colors. This problem finds many applications in scheduling, resource allocation, document classification and clustering, and several other areas.

We will now discuss constructive heuristics known as **sequential coloring algorithms** that are very popular for solving graph coloring problems, and perform very well in practice. All these algorithms have many features in common with the greedy approach. In these algorithms, the colors used are numbered serially 1, 2, ... The steps in these algorithms are:

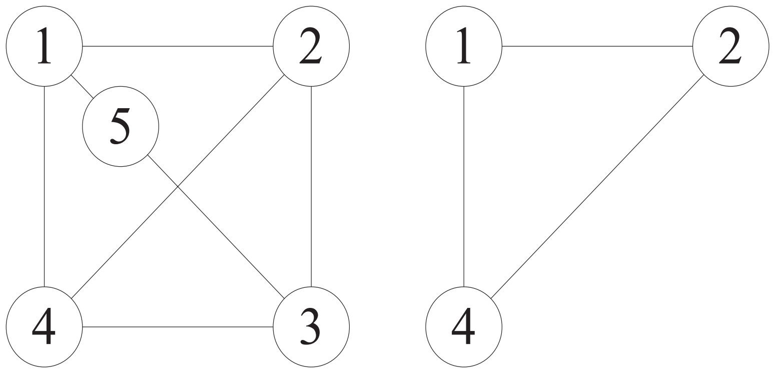


Figure 9.7:

Sequential Graph Coloring Algorithms

BEGIN

Step 1: Sequencing the vertices of the graph for coloring: Arrange all the vertices of the graph in a sequence for coloring.

Step 2: Coloring: Color the vertices in the order of the sequence selected in Step 1. To each vertex assign the minimum possible color which has not been assigned to any previously colored adjacent vertex. Terminate when all the vertices are colored.

END.

The various sequential coloring algorithms differ in the rules used for sequencing the vertices in Step 1. We provide below the rules used in the three most popular sequential coloring algorithms.

1. LF Algorithm (Largest First Vertex Ordering): In this algorithm the vertices are ordered in nonincreasing order of their degree in the graph. So, if the sequence of vertices is $V_1, V_2, \ldots, V_n$, and their degrees are $d_1, \ldots, d_n$ respectively; then $d_1 \geq d_2 \geq \ldots \geq d_n$.

2. SL Algorithm (Smallest Last Vertex Ordering): In this algorithm the vertices are arranged in the order $V_1, \ldots, V_n$ satisfying the property: for each i, V_i has the smallest degree in the subgraph induced by the subset of vertices $\{V_1, \ldots, V_i\}$.

This sequence is easily determined by applying the following procedure to generate the vertices $V_n, V_{n-1}, \ldots, V_1$ in the reverse order (i.e., reverse order to the sequence to be used for coloring).

Procedure to generate the reverse order to the sequence for coloring nodes in the SL algorithm: Start with the orginal graph G as the "current graph". Put the smallest degree vertex in it as the first element V_n in the reverese order. Delete this vertex, and all the edges containing it from the current graph. Make the remaining graph the next "current graph" and go to the General Step.

General Step: Put the smallest degree vertex in the current graph as the next element in the reverese order. Delete this vertex, and all the edges containing it from the current graph. Make the remaining graph the next "current graph" and repeat the General Step until all the nodes are included in the reverese order.

3. DSATUR Algorithm (Degree Saturation Vertex Ordering): In this algorithm, the vertex ordering is dynamic, i.e., Steps 1 and 2 are combined into a single step, and each vertex in the sequence for coloring is selected at the time of coloring.

The first vertex to be colored, V_1, is one of maximum degree in G. At any stage of the algorithm, the **saturation degree** of a vertex V not yet colored is defined to be the number of different colors assigned to vertices adjacent to V at that stage.

When vertices $V_1, \ldots, V_{i-1}$ have been colored; among the uncolored vertices V_i is selected as the one with maximum saturation degree (if there is a tie, among those tied choose V_i as the one having the maximum degree in the subgraph of uncolored vertices at that stage). The selected vertex V_i is then given the minimum possible color which has not yet been given to any of its adjacent vertices. The algorithm continues the same way.

Example: We will color the vertices of the following 10 vertex graph using the three different sequential coloring algorithms discussed above.

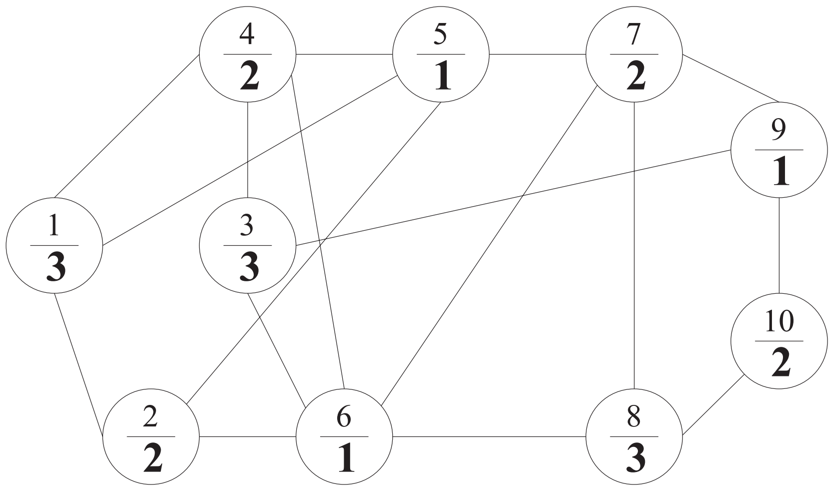


Figure 9.8: Each node is represented by a circle. The number in the top of the circle in normal style is the node number, the number at the bottom in bold style is the number of the color assigned to this node by the LF algorithm using the LF vertex ordering.

The degrees of nodes 1 to 10 in serial order in this graph are $(3, 3, 3, 4, 4, 5, 4, 3, 3, 2)$. So a sequence for coloring nodes in this graph by the LF vertex ordering is $(6, 4, 7, 5, 1, 2, 3, 9, 8, 10)$. Coloring the nodes in this sequence leads to the coloring shown in Figure 9.8 in bold style numbers, obtained by the LF algorithm.

The smallest degree node in the graph in Figure 9.8 is 10 with degree 2. After removing node 10 and the edges (10; 8), (10; 9) from this graph; node 9 is a node of smallest degree in the remaning graph. Continuing this way, we find that the reverse order for coloring nodes by the SL vertex ordering is $(10, 9, 8, 7, 3, 6, 2, 5, 4, 1)$. So, the sequence for coloring nodes in the SL algorithm is $(1, 4, 5, 2, 6, 3, 7, 8, 9, 10)$. Coloring the nodes in this sequence leads to the coloring shown in Figure 9.9 in bold style numbers.

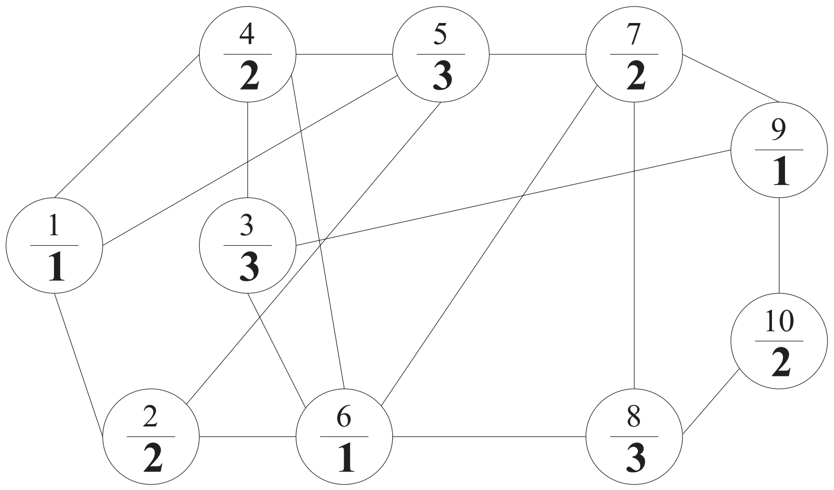


Figure 9.9: Numbers inside nodes in bold style are the numbers of the colors assigned to the nodes by the SL algorithm using the SL vertex ordering given above.

The first node to color by the DSATUR algorithm is node 6 with the highest degree, so it gets color 1. At that stage, its adjacent vertices 2, 3, 4, 7, 8 all have the highest saturation degree of 1. Among these, 7 is the one with the highest degree in the subgraph of uncolored vertices at this stage, so it is colored next. Continuing this way, we get the coloring shown in Figure 9.10.

In this example, all three sequential algorithms use 3 colors, the optimum number of colors for this graph. Computational experiments have shown that on an average DATUR algorithm gives the best results in general, followed by the LF algorithm, and then the SL algorithm. See [Brélaz, 1979] and [Matula, Marble, and Isaacson, 1972]. The

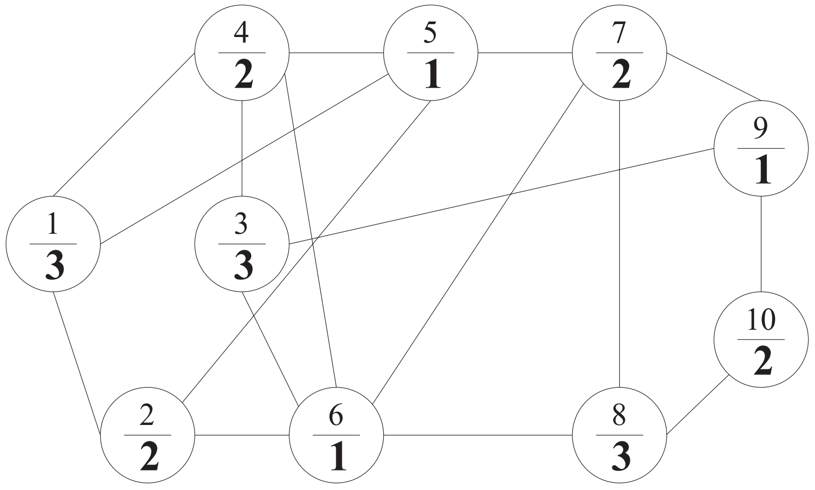


Figure 9.10: Numbers inside nodes in bold style are the numbers of the colors assigned to the nodes by the DSATUR algorithm.

following table gives a summary from the final output of each algorithm, of the average number of colors used to color the nodes in a 1000 node random network generated randomly such that for each $i \neq j$, the probability of there being the edge $(i; j)$ in the graph is 0.5; taken from [Johri, Matula, 1982] .

Algorithm	Avarage no. colors used
LF	122.7
SL	124.3
DSATUR	115.8

.

9.10 The Importance of Heuristics

A consummate skill in modeling problems is a great help to anyone aspiring to be a practitioner of optimization methodology. Knowledge of exact algorithms for well solved problems such as linear programs and convex programming problems, and an understanding of how these algorithms work is of course very important. But in the increasingly complex world of modern technology, skill in designing good heuristic methods for problems for which no effective exact algorithms are known, is an essential component in a successful optimization analyst's toolbox. The development of heuristic methods is being driven by the ever increasing needs for them in many fields.

9.11 Exercises

9.1: In a textile firm there is a special loom for weaving extra-'wide fabrics of a special type. On the first day of a month the firm has 7 jobs or orders which can be processed on this loom. For $i = 1$ to 7, p_i, d_i, r_i are respectively the processing time in days, due date (day number), and profit from, job i. This data is given below.

i	1	2	3	4	5	6	7
p_i	9	10	12	5	11	8	13
d_i	4	13	15	8	20	30	30
r_i	90	130	85	35	77	68	100

If job i is accepted, the material has to be delivered on the due date d_i for that job (d_i is the day number counting from the first day of the month). Jobs are independent, and the loom can process only one job at a time. Formulate the problem of selecting the jobs to accept to maximize the total profit subject to the constraint that all the accepted jobs should be completed by their respective due dates. Develop a heuristic method for obtaining a good solution to this problem.

9.2: Consider a company producing a single product to meet known demand over a finite number of time periods. The cost function for producing x units of the product in a period may be written

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ px + g & \text{if } x > 0 \end{cases}$$

where g is a fixed cost (or setup cost) that is incurred for producing a positive quantity of the product, and p is the variable cost for producing each unit of product once the setup cost is incurred.

Suppose the planning horizon consists of n time periods. For $i = 1$ to n we are given the following data: d_i = demand for the product in period i (in units) that must be met, k_i = production capacity in period i (in units), g_i = fixed (or setup) cost to be incurred to make a positive quantity of the product in period i, p_i = variable cost per unit of making product in period i after the fixed charge is incurred, c_i = holding or storage cost per unit for storing product from period i to period $i+1$.

All demand has to be met exactly in each period. Product made in any period can be used to meet the demand in that period, or stored to fulfill the demand in later periods.

Develop a heuristic method to obtain good production-storage plans of minimal cost. Apply your method on the numerical problem in which $n = 4$, $k_i = 100$ for all i, $(d_1, d_2, d_3, d_4) = (50, 40, 30, 50)$; and for all i, $g_i = \$100$, $p_i = \$10$, $c_i = \$1$. ([T. E. Ramsay Jr., and R. R. Rardin, Jan. 1983]).

9.3: A large percentage of world seaborne trade in high value general cargo goods

Westbound values top half, Eastbound values bottom half

	Max. cargo (TEU/week)				Revenue ($100 units/TEU)			
	HFX	NYC	BLT	POR	HFX	NYC	BLT	POR
HAV	100	200	50	50	13	10	12	11
BRH	60	100	150	100	12	9	12	12
GOT	60	200	60	60	12	12	12	12
LIV	150	300	80	60	9	11	11	10
ROT	80	300	200	200	11	8	11	10
HAV	80	150	40	40	8	12	8	7
BRH	60	50	50	100	9	12	9	8
GOT	60	300	70	70	10	10	10	10
LIV	80	80	120	80	9	11	10	10
ROT	60	100	180	270	8	11	8	7

now moves in containers called TEU, because high port labor costs make capital intensive container operations much more economic than conventional methods. Purpose built, cellular container ships are used for this purpose. Consider a shipping company operating in the North Atlantic with container ships of capacity 1000 TEUs each. The ports that this company operates in Europe are HAV (Le Havre), BRH (Bremerhaven), GOT (Gothenburg); and in North America are HFX (Halifax), NYC (New York), BLT (Baltimore), and POR (Portsmouth). Assume that the travel time between any pair of ports in Europe is 10 hours, and between any pair of ports in North America is 8 hours; and that the travel time between the coasts is 150 hours. Also assume that the ships spend 24 hours at each port of call plus 6 hours of pilotage in and out of the port. The tables above and below give the cargo market data.

Critical time (hrs.), Westbound values left, Eastbound values right

	HFX	NYC	BLT	POR	HFX	NYC	BLT	POR
HAV	142	170	230	225	300	170	300	300
BRH	208	200	250	250	300	220	260	240
GOT	160	250	300	300	260	190	300	300
LIV	185	180	300	300	300	300	300	300
ROY	200	166	208	203	300	196	200	180

Assume that the demand for cargo on a ship's route drops by approximately 10% of the figure quoted above for each 24 hours that the transit time to destination exceeds the critical time given. Develop a heuristic method that builds good 3 week roundtrip ship routes for this company maximizing the revenue per roundtrip. The method can begin with 2 port routes and successively add one port at a time until the limit on roundtrip duration is reached. How many ships can the company operate profitably? Develop routes for all these ships using this heuristic method. ([T. B. Boffey, E, D, Edmond, A. I. Hinxman, and C. J. Pursglove, May 1979]).

9.4: A trucking company has a depot at location 1 from where they have to deliver a material to customers at locations 2, 3, 4, 5. Following table contains the data.

From	Driving time (mts.) to location						Units to
	1	2	3	4	5	6	deliver
Location 1		30	20	10	10	20	
2			10	20	40	50	100
3				10	30	40	10
4					20	30	20
5						10	100
6							170

Each truck can carry at most 200 units, and has a driving limit of 100 minutes. Develop an effective heuristic to find good routes for trucks in such a problem, and apply it on this numerical example. ([I. M. Cheshire, A. M. Malleson, and P. F. Naccache, Jan. 1982]).

9.5: One-dimensional Cutting Stock Problem Material such as lumber, pipe, or cable is supplied in *master pieces* of a standard length C. Demands occur for pieces of the material of arbitrary lengths not exceeding C. The problem is to use minimum number of standard length master pieces to accommodate a given list of required pieces. Develop a heuristic method for producing a good solution for this problem. Apply your heuristic on the numerical problem in which $C = 100$, and one piece of length each 84, 63, 14, 33, 71, 94, 54, 39, 56, 41, 50 are required.

9.6: Single Machine Tardiness Sequencing There are n jobs to be processed by a single machine. All the jobs are available for processing at time point 0. For $i = 1$ to n, p_i, d_i are the positive processing time and due date of job i, and w_i is a given positive weight. The machine processes only one job at a time without interruption.

Given the order or sequence in which the jobs are to be processed on the machine, the earliest completion time c_i and tardiness $t_i = \max\{c_i - d_i, 0\}$ of job i can be computed for all i. In the *total weighted tardiness problem*, the aim is to find a processing order for the jobs that minimizes $\sum_{i=1}^{n} w_i t_i$. When all the job weights are equal, minimizing $\sum_{i=1}^{n} t_i$ is called the *total tardiness problem*. Develop effective heuristics for solving both these problems. Apply your algorithm on the numerical problem with $n = 12$ and the following data.

Job i	p_i	d_i	w_i
1	33	35	2
2	17	110	1
3	6	43	3
4	89	119	1
5	5	23	3
6	13	36	4
7	21	74	1
8	15	69	2
9	63	210	3
10	34	184	4
11	12	39	1
12	9	51	2

([C. N. Potts and L. N. Van Wassenhove, Dec. 1991]).

9.7: Develop a heuristic method for obtaining a good solution to the multidimensional 0−1 knapsack problem. Apply your method on the following problem.

$$\begin{array}{rrrrrrcr}
\text{Maximize} & 4x_1 & +3x_2 & x_3 & +6x_4 & +5x_5 & & \\
\text{subject to} & x_1 & +3x_2 & +4x_3 & +3x_4 & +2x_5 & \leq & 8 \\
 & 8x_1 & +x_2 & +9x_3 & & +x_5 & \leq & 10 \\
\end{array}$$
$$x_j = 0 \text{ or } 1 \text{ for all } j$$

([A. Volgenant and J. A. Zoon, Oct. 1990]).

9.8: Graph Coloring Problem The nodes of a graph are to be colored. The same color can be used to color any number of nodes, but if there is an edge joining any pair of nodes, those two nodes must have different colors. It is required to find a node coloring satisfying this constraint that uses the smallest number of colors. Color the nodes of the graph in Figure 9.11 using the heuristic algorithms discussed in Section 9.9, and compare these algorithms using the results obtained.

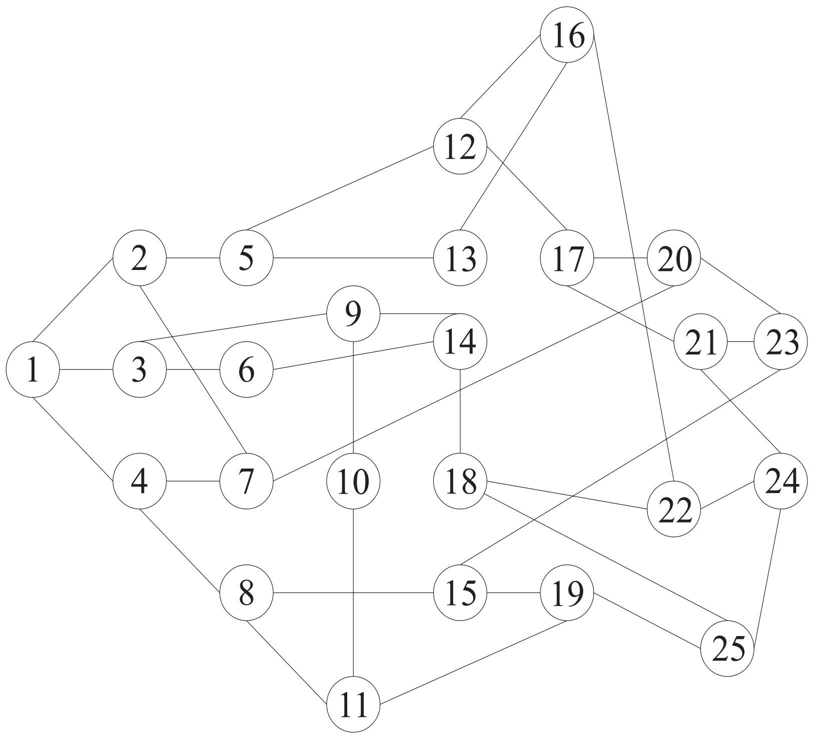


Figure 9.11:

9.9: The Linear Placement Problem This exercise is concerned with locating n facilities at n sites along a one dimensional line where adjacent sites are a unit distance apart. For $i \neq j$ between 1 to n, t_{ij} is the total traffic between facilities i and j, all these t_{ij} are given.

For $i = 1$ to n, if p_i is the number of the facility located at site i, then the distance between the facilities at sites i and j is $|i - j|$ and the cost incurred between them is $|i-j|t_{p_ip_j}$. Hence the total cost of the placement $(p_1, \ldots, p_n)$ is $\sum_{i=1}^{n} \sum_{j=i+1}^{n} (j-i)t_{p_ip_j}$. The problem is to find a placement of facilities to sites that minimizes this total cost

This problem has many applications. An example is the assignment of flights to gates in a horseshoe-shaped airport terminal. The traffic between two flights F_1 and F_2 would be defined as the number of passengers scheduled to fly F_2 following F_1 plus the number scheduled to fly F_1 following F_2. An optimum placement would minimize overall passenger inconvenience.

Develop a good heuristic method for this problem. Apply your heuristic method on the numerical problem in which the traffic data (t_{ij}) is given below.

$j =$	1	2	3	4	5	6
$i = 1$	0	1	5	5	7	8
2		0	3	4	1	5
3			0	7	8	1
4				0	6	4
5					0	10

9.10: A TSP With Side Constraints We are given n cities, in which 1 is the hometown. For $i \neq j = 1$ to n, v_i is the positive valuation for city i, d_i is the positive entrance fees for visiting city i, and c_{ij} is the positive airline fare to go from city i to city j. The problem is to find a roundtrip (either a tour or a subtour covering a subset of cities) starting and ending at the hometown that maximizes the sum of valuations of the cities visited, while satisfying a budget constraint that the total cost (total of airline fares plus the entrance fees for the cities visited) has to be $\leq$ a specified budgeted amount b. Among subtours or tours having identical total valuation, the one with the least total cost is considered superior. Develop either an exact or a good heuristic algorithm for solving this problem. Apply your algorithm on the numerical problem with data $n = 11$, $b = 3000$, and the rest of the data given in the following table.

	c_{ij}										
$j =$	1	2	3	4	5	6	7	8	9	10	11
$i = 1$	0	320	220	250	330	220	600	310	150	420	550
2	270	0	290	410	460	230	780	310	360	580	620
3	190	250	0	230	260	100	640	130	240	380	450
4	220	490	270	0	200	330	400	250	150	250	430
5	390	550	300	230	0	370	290	240	300	190	340
6	190	200	120	280	310	0	600	170	270	440	490
7	500	320	270	340	250	510	0	290	430	140	270
8	260	370	140	290	210	200	340	0	290	340	370
9	170	430	280	170	350	310	520	340	0	410	630
10	490	690	450	300	220	520	150	410	350	0	360
11	660	750	530	520	280	590	320	440	530	310	0
d_j	0	100	100	100	100	100	100	100	100	100	100
v_j	6	10	16	8	6	10	20	6	6	6	6

([M. Padberg and G. Rinaldi, Nov. 89]).

9.11:

Product	Demand in batches, in period									
	1	2	3	4	5	6	7	8	9	10
1	6.4	6.4	6.4	8.0	6.4	6.4	6.4	6.4	8.5	6.4
2	2.2	2.4	2.6	4.3	2.7	2.8	3.0	3.1	3.9	3.3
3	3.6	3.6	4.2	5.1	4.9	4.3	4.7	4.3	6.0	4.2
4	6.8	6.8	6.8	7.9	7.3	6.5	6.5	6.5	7.3	6.4
5	3.6	3.6	2.7	7.0	5.5	6.4	6.5	5.5	8.2	6.4
6	3.6	3.6	3.6	5.5	4.2	2.4	2.4	2.4	3.0	2.4
7	2.7	2.4	2.6	4.2	3.0	3.2	3.4	3.6	4.5	4.2
8	4.2	2.4	2.5	2.9	2.6	2.8	3.0	2.6	3.7	3.2
9	5.5	6.4	5.5	4.8	5.5	5.5	6.4	5.5	6.2	5.5
10	7.6	7.6	7.6	8.0	7.6	8.2	8.3	8.4	9.3	8.9
11	4.6	4.6	4.6	4.8	4.6	4.6	4.6	4.6	6.0	4.6
12	3.8	3.6	3.3	5.3	3.6	3.6	3.9	4.0	5.1	4.1
13	9.1	6.5	6.5	8.4	6.5	6.5	6.5	6.5	7.0	6.5
14	2.7	2.9	2.9	3.9	3.5	3.6	3.8	4.2	5.4	4.7
15	2.2	2.4	2.6	3.0	3.0	3.2	3.4	3.5	4.8	3.5

A chemicals company manufactures 15 different products using a chemical reactor. This problem deals with planning the manufacture of these products over a 20-week planning horizon which is divided into 10 periods of two weeks each. On a three shift basis, 336 production hours are available on the reactor in each period. Quantities of products are measured in batches, the batch size being 60 tons for each product. The demand and relevant production data is given in the tables above and below.

i	Switch-over time (in hours) from product i to product														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0.5	2.0	6.6	8.0	7.6	3.1	3.7	8.0	6.3	8.0	6.6	5.9	5.6	6.9	2.0
2	2.4	0.5	3.6	8.0	8.0	4.3	2.5	3.2	3.8	3.3	5.6	2.2	3.3	5.1	2.0
3	6.1	5.1	0.5	3.8	8.0	2.0	2.3	2.0	6.3	7.8	5.6	7.1	2.0	3.4	8.0
4	5.0	8.0	2.0	0.5	6.7	3.1	8.0	5.4	5.1	6.1	2.0	8.0	8.0	2.0	3.3
5	7.4	2.1	4.8	8.0	0.5	2.8	7.9	6.8	2.7	8.0	5.3	5.8	4.6	6.7	2.8
6	6.0	2.5	8.0	2.0	4.1	0.5	2.0	2.8	8.0	5.4	5.4	5.8	5.8	6.2	8.0
7	2.1	2.0	2.6	4.8	2.0	8.0	0.5	3.2	3.8	8.0	4.4	8.0	6.7	7.5	5.9
8	6.6	3.1	8.0	4.5	6.7	2.7	5.1	0.5	7.9	8.0	2.8	2.4	3.8	2.0	7.4
9	4.6	8.0	6.5	5.6	5.2	3.6	6.1	7.8	0.5	2.7	8.0	4.8	5.7	4.4	4.9
10	2.0	2.6	8.0	5.8	8.0	8.0	2.5	5.4	8.0	0.5	8.0	7.8	2.1	8.0	8.0
11	2.0	4.6	4.9	5.5	4.5	4.9	2.0	2.1	4.8	5.8	0.5	7.0	8.0	6.0	2.0
12	5.2	3.0	5.2	7.0	8.0	8.0	2.0	6.9	8.0	7.5	4.4	0.5	8.0	3.1	5.9
13	3.5	8.0	5.0	8.0	4.8	4.4	4.4	7.6	8.0	2.5	6.2	2.4	0.5	3.3	5.2
14	2.0	5.3	3.9	8.0	5.2	4.6	6.4	5.1	2.0	6.2	2.2	2.1	8.0	0.5	2.5
15	7.6	8.0	8.0	8.0	8.0	3.0	4.8	4.2	3.5	2.0	4.5	2.0	2.0	2.0	0.5

Relevant production data

i	I_i	k_i	p_i	i	I_i	k_i	p_i	i	I_i	k_i	p_i
1	0	600	4	6	50	600	2.4	11	0	600	2
2	0	500	6	7	0	600	3.4	12	0	500	4
3	0	500	4	8	0	600	6	13	0	600	2
4	450	500	4	9	350	500	4	14	0	500	3.6
5	0	500	4	10	0	600	4	15	100	600	3.2

I_i = beginning inventory (tons), k_i = tank capacity (tons),
p_i = production time (hrs./batch), of product i

Inventory holding costs are \$1,000 per batch per period for each product. Opportunity cost for lost production on the reactor during time spent in switching over from one product to another is estimated at $20,000/hour.

Develop a heuristic method that determines a good operational plan (that determines which products are to be manufactured in each period, the lot size for each, and the sequence in which these products are manufactured in each period) to minimize the total cost (inventory holding cost plus the opportunity costs due to setups between production runs) while meeting the demands for all the products. ([W. J. Selen and R. M. J. Heuts, Mar. 1990]).

Additional exercises for this chapter are available in Chapter 13 at the end.

9.12 References

JORS = Journal of the Operational Research Society; EJOR = European Journal of Operational Research.

A. BEN HADJ-ALOUANE, J. C. BEAN, and K. G. MURTY, 1993, "A Hybrid Genetic/Optimization Algorithm for a Task Allocation Problem", *Journal of Scheduling*, 2(1999)189-201.

T. B. BOFFEY, E. D. EDMOND, A. I. HINXMAN, and C. J. PURSGLOVE, May 1979, "Two Approaches to Scheduling Container Ships With an Application to the North Atlantic Route", *JORS*, 30, no. 5(413-425).

D. BRÉLAZ, 1979, "New Methods to Color the Vertices of a Graph", CACM, 22(251-256).

I. M. CHESHIRE, A. M. MALLESON, and P. F. NACCACHE, Jan. 1982, "A Dual Heuristic for Vehicle Scheduling", *JORS*, 33, no. 1(51-61).

G. CLARKE, and J. WRIGHT, 1964, "Scheduling of Vehicles from a Central Depot to a Number of Delivery Points", *Operations Research*, 12(568-581).

G. CORNUEJOLS, M. FISHER, and G. NEMHAUSER, 1977, "Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms", *Management Science*, 23(789-810).

L. DAVIS, 1991, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, NY.

R. W. EGLESE, 1990, "Simulated Annealing: A Tool for Operational Research", *EJOR*, 46(271-281).

F. GLOVER, M. LAGUNA, E. TAILLARD, and D. DE WERRA, (eds.) 1993, *Tabu Search*, *Annals of Operations Research*, Vol. 41.

D. GOLDBERG, 1989, "*Genetic Algorithms in Search, Optimization and Machine Learning*", Addison-Wesley, Reading, MA.

J. HOLLAND, 1975, *"Adaptation in Natural and Artificial Systems"*, The University of Michigan Press, Ann Arbor, MI.

A. JOHRI, D. W. MAtula, 1982, "Probabilistic bounds and Heuristic Algorithms for Coloring Random Graphs", Dept. Computer Science & Engineering, Southern Methodist University, Dallas, Texas 75275, USA.

S. KIRKPATRICK, C.D. GELATT Jr., and M. P. VECCHI, 1983, "Optimization by Simulated Annealing", *Science*, 220(671-680).

J. G. KLINCEWICZ, 1980, "Locating Training Facilities to Minimize Travel Costs", Bell Labs. Technical Report, Holmdel, NJ.

J. R. KOZA, 1992, "*Genetic Programming: On the Programming of Computers by Means of Natural Selection*", The MIT Press, Cambridge, MA.

D. W. MATULA, G. MARBLE, and J. D. ISAACSON, 1972, "Graph Coloring Algorithms", in R. C. read (ed.), *Graph Theory and Computing*, Academic Press.

G. L. NEMHAUSER and L. A. WOLSEY, 1988, *Integer and Combinatorial Optimization*, Wiley, NY.

C. OKONJA-ADIGWE, July 1989, "The Adult Training Center Problem: A Case Study", *JORS*, 40, no. 7, (637-642).

M. PADBERG, and G. RINALDI, Nov. 1989, "A Branch-and-Cut Approach to a Traveling Salesman Problem With Side Constraints", *Management Science*, 35, 11 (1393-1412).

C. N. POTTS, and L. N. VAN WASSENHOVE, Dec. 1991, "Single Machine Tardiness Sequencing Heuristics", *IIE Transactions*, 23, no. 4, 346-354.

T. E. RAMSAY Jr., and R. R. RARDIN, Jan. 1983, "Heuristics for Multistage Production Planning Problems", *JORS*, 34, no. 1 (61-70).

C. R. REEVES (ed.), 1993, *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publications, Oxford, UK.

D. ROSENKRANTZ, R. STERNS, and P. LEWIS, 1977, "An Analysis of Several Heuristics for the Traveling Salesman Problem", *SIAM J. on Computing*, 6(563-581).

W. J. SELEN, and R. M. J. HEUTS, March 1990, "Operational Production Planning in a Chemical Manufacturing Environment", *EJOR*, 45, no. 1, (38-46).

A. VOLGENANT, and J. A. ZOON, Oct. 1990, "An Improved Heuristic for Multidimensional 0−1 Knapsack Problems", *JORS*, 41, no. 10, 963-970.

Index

For each index entry we provide the section number where it is defined or discussed first.