CHAPTER 1

Introduction

1.1 Combinatorial Optimization

Combinatorial optimization constitutes one specific class of problems. The word *combinatorial* is derived from the word *combinatorics*, which is a branch of mathematics concerned with the study of arrangement and selection of discrete objects. In *combinatorics* one is usually concerned with finding answers to questions such as *"does a particular arrangement exist?"* or *"how many arrangements of some set of discrete objects exist?"* Finding the number of orderings of some set of discrete objects usually consists of deriving a mathematical formula or relation which, when evaluated for the parameters of the problem leads to the answer. On the other hand, *combinatorial optimization* is not concerned with whether a particular arrangement or ordering exists but rather, concerned with the determination of an *optimal* arrangement or order [Law76].

In most general terms, a problem is a question whose answer is a function of several parameters. Usually the problem is stated by articulating the properties that must be satisfied by its solution. A particular instance of the problem is obtained by fixing the values of all its parameters. Let's take a simple example.

Example 1.1 *The shortest path problem.*

Problem: Given a connected graph[1] $G = (V, E)$, where V is a set of n vertices and E is a set of edges. Let $D = [d_{i,j}]$ be a distance matrix, where $d_{i,j}$ is the distance between vertices v_i and v_j (weight or length of the edge $(v_i, v_j) \in E$). For convenience, we assume $d_{i,j} = d_{j,i} > 0$, $d_{i,i} = 0, \forall v_i,\ v_j \in V$, and $d_{i,j} = \infty$ if there is no edge between v_i and v_j.

[1]For definition of terms from graph theory the reader is referred to the text *Algorithmic Graph Theory* by Alan Gibbons, Cambridge University Press, 1985.

Objective: Find the shortest path from some source node v_i to some target node v_j. A path $\pi(v_i, v_j)$ from v_i to v_j is a sequence of the form $[v_i, v_{i_1}, v_{i_2}, \ldots, v_{i_l}, v_j]$, such that $(v_i, v_{i_1}) \in E$, $(v_{i_k}, v_{i_{k+1}}) \in E$, $1 \leq k \leq l-1$, and $(v_{i_l}, v_j) \in E$. The length of the path is the sum of the length of its constituent edges. That is,

$$length(\pi(v_i, v_j)) = d_{i,i_1} + \sum_{k=1}^{l-1} d_{i_k,i_{k+1}} + d_{i_l,j}$$

■

A particular instance of the above problem is defined when one fixes the graph, the distance measure, and decides the source and target vertices. For example, Figure 1.1 is one instance of the shortest path problem.

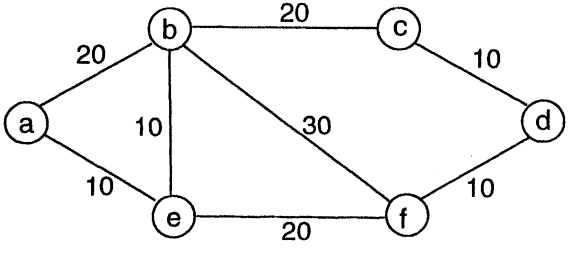

Figure 1.1: An instance of the shortest path problem: The shortest a-to-d path is $\pi(a, d) = [a, e, f, d]$ and $length(\pi(a, d)) = d_{a,e} + d_{e,f} + d_{f,d} = 10 + 20 + 10 = 40$.

A solution (optimal or not) to a *combinatorial optimization problem* usually requires that one comes up with a suitable algorithm, which when applied to an instance of the problem produces the desired solution.

An *algorithm* is a finite step-by-step procedure for solving a problem or to achieve a required result. The word *algorithm* is named after the ninth-century scholar *Abu-Jaafar Muhammad Ibn Musa Al-Khowarizmi* who authored among other things a book on mathematics.

Combinatorial optimization problems are encountered everywhere, in science, engineering, operation research, economics, and so forth. The general area of *combinatorial optimization* came to the fore with the advent of the digital computer. Algorithmic solutions to typical combinatorial optimization problems involve an extremely large number of computational steps and are impossible to execute by hand. The last 30 years have witnessed the development of numerous algorithms for almost any imaginable combinatorial optimization problem. Such algorithmic solutions were unthinkable before the advent of the era of modern computing.

Let us consider three examples of combinatorial problems.

Example 1.2 *Sorting.*
Problem: Given an array of n real numbers A[1:n].
Objective: Sort the elements of A in ascending order of their values.

There are $n!$ possible arrangements of the elements of A. In case all elements are distinct only one such arrangement is the answer to the problem. Several algorithms have been designed to sort n elements. One such algorithm is the *Bubble-Sort* algorithm.

```
Algorithm BubbleSort (A[1:n]);
Begin /* Sort array A[1:n] in ascending order */
 var integer i, j;
   For i = 1 To n − 1 Do
     For j = i + 1 To n Do
       If A[i] > A[j] Then
           swap (A[i],A[j]);
     EndFor;
   EndFor;
End Algorithm;
```

■

Example 1.3 *Maximum set bipartitioning.*
Problem: Given a set of n positive integers $x_1, x_2, \ldots, x_n$ (n even).
Objective: Partition the set X into two subsets Y and Z such that

1. $|Y| = |Z| = \frac{n}{2}$,
2. $Y \cup Z = X$, and
3. the difference between the sums of the two subsets is maximized.

There are $\binom{n}{\frac{n}{2}}$ possible bipartitions of the set X. To find the required bipartition, we can follow the steps of the following algorithm.

```
Algorithm MaxBipartition (X[1:n]);
Begin
   BubbleSort(X[1:n]) /* Sort array X[1:n] in ascending order */
   Put the n/2 smaller integers in Y;
   Put the n/2 larger integers in Z;
   Return (Y, Z)
End Algorithm;
```

■

Example 1.4 *Minimum set bipartitioning.*

Problem: Given a set of n positive integers $x_1, x_2, \ldots, x_n$ (n even).

Objective: Partition the set X into two subsets Y and Z such that

1. $|Y| = |Z| = \frac{n}{2}$,
2. $Y \cup Z = X$, and
3. the difference between the sums of the two subsets is minimized.

■

The two problems of set bipartitioning appear to be similar; only one word has changed (maximized became minimized). However the minimum set bipartition problem is much more difficult to solve. Actually, the two problems belong to two different classes of problems: maximum set bipartitioning belongs to the class of *easy problems* for which there are several efficient algorithms, whereas minimum set bipartitioning belongs to the class of *hard problems* with no known efficient algorithm (typically only full enumeration will guarantee finding an optimal solution).

Before we clarify the distinction between *easy* and *hard problems*, we first need to define the notions of *time* and *space complexity* of algorithms and how we measure them.

1.1.1 Complexity of Algorithms

Two important ways to characterize the *effectiveness* of an algorithm are its *space complexity* and *time complexity. Time complexity* of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat coarse, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count [SB80]. Asymptotic analysis makes use of the Big-Oh notation.

Big-Oh Notation

We say that $f(n) = O(g(n))$ if there exist positive constants n_0 and c such that for all $n > n_0$, we have $f(n) \leq c \cdot g(n)$. Alternately, we say that $f(n)$ is *upper bounded* by $g(n)$. The Big-Oh notation is used to describe the space and time complexity of *algorithms.*

Example 1.5 Consider the "BubbleSort" algorithm to sort n real numbers (page 3). The procedure "BubbleSort" requires $O(n)$ storage (for the array A) and $O(n^2)$ running time (two nested **for** loops). The above statement should be taken to mean that the BubbleSort procedure requires no more than linear amount of storage and no more than a quadratic number of steps to solve the

sorting problem. In this sense, the following statement is also equally true: the procedure BubbleSort takes $O(n^2)$ storage and $O(n^3)$ running time! This is because the Big-Oh notation only captures the concept of "upper bound." However, in order to be informative, it is customary to choose $g(n)$ to be as small a function of n as one can come up with, such that $f(n) = O(g(n))$. Hence, if $f(n) = a \cdot n + b$, we will state that $f(n) = O(n)$ and not $O(n^k)$, $k > 1$.

■

Big-Ω and Big-Θ Notation

The Big-Oh notation is one of several convenient notations used by computer scientists in the analysis of algorithms. Two other notational constructs are frequently used: Big-Ω (Big-Omega) and Big-Θ (Big-Theta) notation.

The Big-Oh notation is easier to derive. Typically, we prove that an algorithm is $O(f(n))$ and try to see whether it is also $\Omega(f(n))$.

Definition 1 Big-Omega Notation.
We say that $f(n) = \Omega(g(n))$ if there exist positive constants n_0 and c such that for all $n > n_0$, we have $f(n) \geq c \cdot g(n)$. Alternately, we say that $f(n)$ is *lower bounded* by $g(n)$.

Definition 2 Big-Theta Notation.
We say that $f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that for all $n > n_0$, we have $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

The Θ notation is used to state an exact bound on the time complexity of a given algorithm. For example, the time complexity of BubbleSort is $O(n^2)$, $\Omega(n^2)$, as well as $\Theta(n^2)$.

How useful are these complexity functions? For example, can we use them to find out how much time the algorithm would run? Asymptotic analysis does not tell us the execution time of an algorithm on a particular problem instance, it barely characterizes the growth rate of the algorithm runtime as a function of the problem size. For example, if the sorting of 1,000 real numbers with the BubbleSort algorithm takes 1 millisecond on a particular computer, then we expect the sorting of 5,000 numbers by the same algorithm will require 25 milliseconds on the same computer. The complexity functions are also used to compare algorithms. For example, if algorithm A_1 has time complexity $\Theta(n \log n)$ and algorithm A_2 has time complexity $\Theta(n^2)$, then A_1 is a better algorithm (more efficient or superior to A_2).

In many situations, the runtime of the algorithm is data dependent. In that case, one talks of the best case, the worst case, and average time complexity.

1.1.2 Hard Problems versus Easy Problems

An algorithm is said to be a *polynomial-time* algorithm if its time complexity is $O(p(n))$, where n is the problem size and $p(n)$ is a polynomial function of n. The BubbleSort algorithm of Example 1.2 is a polynomial-time algorithm. The function $p(n)$ is a polynomial of degree k if $p(n)$ can be expressed as follows:

$$p(n) = a_k n^k + \cdots + a_i n^i + \cdots + a_1 n + a_0$$

where $a_k > 0$ and $a_i \geq 0$, $1 \leq i \leq k-1$. In that case, the time complexity function of the corresponding algorithm is said to be $O(n^k)$.

In contrast, algorithms whose time complexity cannot be bounded by polynomial functions are called exponential time algorithms. To be more accurate, an *exponential-time* algorithm is one whose time complexity is $O(c^n)$, where c is a real constant larger than 1. A problem is said to be *tractable* (or *easy*) if there exists a polynomial-time algorithm to solve the problem. From Example 1.2 above, we may conclude that the sorting of real numbers is tractable, since the BubbleSort algorithm given on page 3 solves it in $O(n^2)$ time. Similarly, maximum set bipartitioning is tractable since the algorithm given on page 3 solves it in $O(n^2)$ time.

Unfortunately, there are problems of great practical importance that are not computationally easy. In other words, polynomial-time algorithms have not been discovered to solve these problems. The bad news is that it is unlikely that a polynomial-time algorithm will ever be discovered to solve any of these problems. Such problems are also known as "hard problems" or "intractable problems." For example, the minimum set bipartitioning introduced on page 4 is intractable since finding an optimum partition requires the exploration of $\binom{n}{\frac{n}{2}}$ bipartitions, which is a function that grows as an exponential function of n.[2]

Below, we recall several representative hard problems which find numerous applications in various areas of science and engineering. We shall be using these and other problems throughout the book. Readers interested in a thorough discussion of the subject of NP-completeness are referred to the classic work of Garey and Johnson [GJ79].

Example 1.6 *The traveling salesman problem* (TSP).

Problem: Given a complete graph $G(V, E)$ with n vertices. Let $d_{u,v}$ be the length of the edge $(u, v) \in E$ and $d_{u,v} = d_{v,u}$. A path starting at some vertex $v \in V$, visiting every other vertex exactly once, and returning to vertex v is called a *tour*.

[2] By Stirling's formula we can show that $\binom{n}{\frac{n}{2}} \approx 2^n$. The proof is left as an exercise (see Exercise 1.1).

Objective: Find a *tour* of minimum length, where the length of a *tour* is equal to the sum of lengths of its defining edges.

■

Example 1.7 *Hamiltonian cycle problem* (HCP).

Problem: Given a graph $G(V, E)$ with n vertices.
A *Hamiltonian cycle* is a simple cycle which includes all the n vertices in V. A graph containing at least one *Hamiltonian cycle* is called a *Hamiltonian graph*. A complete graph on n vertices contains $n!$ Hamiltonian cycles.

Objective: Find a Hamiltonian cycle on the n vertices of the graph.

■

Example 1.8 *The vehicle routing problem* (VRP).

Problem: Given an unspecified number of identical vehicles, having a fixed carrying capacity Q, we have to deliver from a single depot quantities q_i $(i = 1, \ldots, n)$ of goods to n cities. A distance matrix $D = [d_{ij}]$ is given, where d_{ij} is the distance between cities i and j $(i, j = 1, \ldots, n$, and city 0 is the depot).

Objective: Find tours for the vehicles (a vehicle tour starts and terminates at the depot) such that

1. the total distance traveled by the vehicles is minimized,
2. every city is serviced by a unique vehicle, and
3. the quantity carried by any vehicle during any single delivery does not exceed Q.

There are several other variations of the VRP problem. For example, the distances may be Euclidean or non-Euclidean, there may be several depots, the vehicles may be different, and the goods may be delivered as well as picked up. Furthermore, there may be timing constraints for each delivery, that is, each customer at a particular (city) has a time window for service. A delivery outside its time window may be acceptable but incurs a penalty, or it may be unacceptable altogether [BGAB83, DLSS88, GPR94, Tai93].

■

Example 1.9 *The graph bisection problem* (GBP).

Problem: Given a graph $G(V, E)$ where, V is the set of vertices, E the set of edges, and $|V| = 2n$. Partition the graph into two subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ such that, (1) $|V_1| = |V_2| = n$, (2) $V_1 \cap V_2 = \emptyset$, and (3) $V_1 \cup V_2 = V$.

Objective: Minimize the number of edges with vertices in both V_1 and V_2.

■

Example 1.10 *Quadratic assignment problem* (QAP).

Problem: Given a set M of $\mid M \mid$ modules and a set L of $\mid L \mid$ locations, $\mid L \mid \geq \mid M \mid$. Let $c_{i,j}$ be the number of connections between elements i and j, and $d_{k,l}$ be the distance between locations k and l.

Objective: Assign each module to a distinct location so as to minimize the wire-length needed to interconnect the modules.

■

Example 1.11 *Minimum set partitioning problem.*

Problem: Given a set of n positive integers $X = \{x_1, x_2, \ldots, x_n\}$.

Objective: Partition the set into two subsets Y of size k and Z of size $n - k$ $(1 \leq k \leq \frac{n}{2})$ such that the difference between the sums of the two subsets is minimized.

■

Example 1.12 *Vertex cover problem.*

Problem: Given a graph $G(V, E)$.

A *vertex cover* of a graph $G(V, E)$ is a subset $V_c \subseteq V$ such that, for each edge $(i, j) \in E$, at least one of i or $j \in V_c$.

Objective: Find a vertex cover of minimum cardinality.

■

All of the above problems are NP-hard [GJ79]. The only way to deal with NP-hard problems is to be satisfied with an approximate solution to the problem. Such an approximate solution must satisfy the constraints, but *may not* necessarily possess the best cost.

1.2 Optimization Methods

There are two general categories of combinatorial optimization algorithms: (1) *exact algorithms*[3] and (2) *approximation algorithms*. Most well known among the first category are linear programming, dynamic programming, branch-and-bound, backtracking, and so forth [HS84].

[3] Several exact algorithms tend to be enumerative.

The linear programming approach formulates the problem as the minimization of a linear function subject to a set of linear constraints. The linear constraints define a convex polytope. The vertices of the polytope correspond to feasible solutions of the original problem. The number of vertices in the polytope is extremely large. For example, an $n \times n$ assignment problem would require $2n$ linear inequalities, together with non-negativity constraints on n^2 variables, which describe a convex polytope with $n!$ vertices, corresponding to the extreme points of the feasible region of the assignment problem [Law76].

Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions. During the search for a solution, dynamic programming avoids full enumeration by pruning early partial decision sequences that cannot possibly lead to optimal sequences. In many practical situations, dynamic programming hits the optimal sequence in a polynomial sequence of decision steps. However, in the worst case, such a strategy may end up performing full enumeration.

Branch-and-bound search methods explore the state space search tree in either a depth-first or breadth-first manner. Bounding functions are used to prune subtrees that do not contain the required optimal state.

Many of the significant optimization problems encountered in practice are NP-hard. For relatively large instances of such problems, it is not possible to resort to optimal enumerative techniques; instead, we must resort to *approximation algorithms*. Approximation algorithms are also known as *heuristic* methods. Insight into the problem through some observations, when properly exploited, usually enables the development of a reasonable heuristic that will quickly find an "acceptable" solution. A heuristic algorithm will only search inside a subspace of the total search space for a "good" rather than the best solution which satisfies design constraints. Therefore, the time requirement of a heuristic is small compared to that of full enumerative algorithms. A number of heuristics have been developed for various problems. Examples of approximation algorithms are the *constructive greedy method*, *local search*, and the modern general iterative algorithms such as *simulated annealing*, *genetic algorithms*, *tabu search*, *simulated evolution*, and *stochastic evolution*.

The greedy method constructs a good feasible solution in stages. It starts from a seed input. Then other inputs are selected in succeeding steps and added to the partial solution until a complete solution is obtained. The selection procedure is based on some optimization measure strongly correlated with the objective function. At each stage, the inputs that optimize the selection measure are added to the partial solution, hence the term *greedy*.

A common feature of all of the aforementioned search algorithms (whether exact or approximate) is that they constitute general solution methods for combinatorial optimization.

This book is concerned with iterative approximation algorithms. Solution techniques such as linear programming, dynamic programming, and branch-and-bound have been the subject of several other books (see, for example, [Fou84, HS84, Hu82, PS82]).

One of the oldest iterative approximation algorithms is the *local search* heuristic. All other more modern iterative heuristics such as *simulated annealing*, *tabu search*, or *genetic algorithms* are generalizations of *local search*. Before we describe *local search*, we need to explain several important concepts that are customarily encountered in combinatorial optimization.

1.3 States, Moves, and Optimality

In most general terms, *combinatorial optimization* is concerned with finding the best solution to a given problem. The class of problems we are concerned with in this book are those with finite discrete state space and which can be stated in an unambiguous mathematical notation.

Combinatorial optimization algorithms seek to find the extremum of a given objective function $Cost$. Without any loss of generality we shall assume that we are dealing with a minimization problem.

Definition 3 An *instance of a combinatorial optimization problem* is a pair $(\Omega, Cost)$, where Ω is the finite set of feasible solutions to the problem and $Cost$ is a *cost function*, which is a mapping of the form,

$$Cost : \Omega \longrightarrow \Re$$

The *cost function* is also referred to as an *objective* or *utility function*. The function $Cost$ assigns to every solution $S \in \Omega$ a (real) number $Cost(S)$ indicating its worth.

Definition 4 A feasible solution S of an instance of a combinatorial optimization problem $(\Omega, Cost)$ is also called a state ($S \in \Omega$). The set of feasible solutions Ω is called the *state space*.

The function $Cost$ allows us to establish an ordering relation. Let S_1 and S_2 be two solutions to the problem. S_1 is judged better than or of equal value to S_2 if $Cost(S_1) \leq Cost(S_2)$.

Solution configurations in the neighborhood of a solution $S \in \Omega$ can always be generated by performing small perturbations to S. Such local perturbations are called moves. For example, for the quadratic assignment problem (Example 1.10 on page 8), a move may consist of the swapping of the locations of two modules.

Definition 5 A neighborhood $\aleph(S)$ of solution S is the set of solutions obtained by performing a simple move $m \in \mathcal{M}$, where $\mathcal{M}$ is the set of simple moves that are allowed on solution S.

A property of most combinatorial optimization problems is that they possess noisy objective functions, that is, the function $Cost$ has several minima over the the state space Ω.

Definition 6 $\hat{S} \in \Omega$ is a *local minimum* with respect to $\aleph(S)$ if $\hat{S}$ has a lower cost than any of its neighboring solutions, that is,

$$Cost(\hat{S}) \leq Cost(S_m), \ \forall S_m \in \aleph(S), \ \forall m \in \mathcal{M}$$

Definition 7 $S^* \in \Omega$ is a *global minimum* iff

$$Cost(S^*) \leq Cost(S), \ \forall S \in \Omega$$

The objective of combinatorial search algorithms is to identify such a global optimum state S^*.

1.4 Local Search

The *local search* heuristic is one of the oldest and easiest optimization methods. Although the algorithm is simple, it has been successful with a variety of hard combinatorial optimization problems. The algorithm starts at some initial feasible solution $S_0 \in \Omega$ and uses a subroutine *Improve* to search for a better solution in the neighborhood of S_0. If a better solution $S \in \aleph(S_0)$ is found, then the search continues in the neighborhood $\aleph(S)$ of the new solution. The algorithm stops when it hits a local optimum. The subroutine *Improve* behaves as follows:

$$Improve(S) = \begin{cases} any\ T \in \aleph(S) & s.t.\ Cost(T) < Cost(S) \\ nil & otherwise \end{cases}$$

An outline of the general local search algorithm is given below.

Algorithm LocalSearch(S_0);
Begin
  $S_2 = S_0$;
  Repeat
    $S_1 = S_2$;
    $S_2 = Improve(S_1)$
  Until $S_2 = nil$;
  Return (S_1)
End /* of LocalSearch */

To use the *local search* heuristic one has to address several issues, namely, (1) how to construct the initial solution, (2) how to choose a good neighborhood for the problem at hand, and (3) the manner in which the neighborhood is searched, that is, the *Improve* subroutine.

(a) Initial solution.

Should one start from a good solution obtained by a constructive algorithm or from a randomly generated solution? Another possibility is to make several runs of *local search* starting from different initial solutions and to select the best among the obtained final solutions. These alternatives have varying computational requirements and would usually result in final solutions of varying quality.

(b) Choice of neighborhood.

Here one has to select the appropriate perturbation function to explore a good neighborhood around current solution. Elaborate perturbations (moves) are more complex to implement, require more time to execute, and usually result in large neighborhoods. In contrast, simple perturbation functions are easier to implement, require less time to execute, and would result in smaller neighborhoods. Hence, one can see a clear trade-off here: a larger neighborhood would require more time to search but holds the promise of reaching a good local minimum while a smaller neighborhood can be quickly explored but would lead to a premature convergence to a poor local minima. This issue can best be resolved through experimentation.

We should note here that if one decides to work with a small neighborhood then one has to start from a good initial solution; otherwise the search will end in a poor-quality local minima. In contrast, if one opts for a large neighborhood, then the initial solution would not have as much effect on the quality of final solution. In that case, starting from a quickly generated random solution or from a good initial solution would result in final solutions of similar quality.

(c) The subroutine "Improve."

The *Improve* subroutine can follow one of the following two strategies: (1) *first-improvement* strategy, where the first favorable cost change is accepted, or (2) the *steepest descent* strategy, where the entire neighborhood is searched, and then a solution with lowest cost is selected. The first strategy may converge sooner to a poorer local minima. However, the decision as to which strategy to use may best be made empirically.

1.4.1 Deterministic and Stochastic Algorithms

Combinatorial optimization algorithms can be broadly classified into *deterministic* and *stochastic* algorithms. A deterministic algorithm progresses toward the solution by making deterministic decisions. For example, *local search* is a deterministic algorithm. On the other hand, stochastic algorithms make random decisions in their search for a solution. Therefore deterministic algorithms produce the same solution for a given problem instance while this is not the case for stochastic algorithms.

Heuristic algorithms can also be classified as *constructive* and *iterative* algorithms. A constructive heuristic starts from a seed component (or several seeds). Then, other components are selected and added to the partial solution until a complete solution is obtained. Once a component is selected, it is never moved during future steps of the procedure. Constructive algorithms are also known as *successive augmentation algorithms.*

An *iterative* heuristic such as *local search* receives two things as inputs, one, the description of the problem instance, and two, an initial solution to the problem. The iterative heuristic attempts to modify the given solution so as to improve the cost function; if improvement cannot be attained by the algorithm, it returns a "NO," otherwise it returns an improved solution. It is customary to apply the iterative procedure repeatedly until no cost improvement is possible. Frequently, one applies an iterative improvement algorithm to refine a solution generated by a reasonable constructive heuristic. To come up with the best constructive algorithm requires far more insight into the problem and much more effort than to set up an iterative improvement scheme of the aforementioned type. Nevertheless, one may argue that it is always certain that if any iterative technique fares well on a problem, then a good constructive/deterministic heuristic has been overlooked. However, the elaboration of such good heuristics is not always possible for many practical problems.

Alternately, one could generate an initial solution randomly and pass it as input to the iterative heuristic. Random solutions are of course generated quickly; but the iterative algorithm may take a large number of iterations to converge to either a local or global optimum solution. On the other hand, a constructive heuristic takes up time; nevertheless the iterative improvement phase converges rapidly if started off with a constructive solution.

Figure 1.2 gives the flowchart of a constructive heuristic followed by an iterative heuristic. The "stopping criteria met" varies depending on the type of heuristic applied. In case of deterministic heuristics, the stopping criterion could be the first failure in improving the present solution. While in the case of nondeterministic heuristics the stopping criterion could be the runtime available, or, k consecutive failures in improving the present solution.

Typically, constructive algorithms are deterministic while iterative algorithms may be deterministic or stochastic.

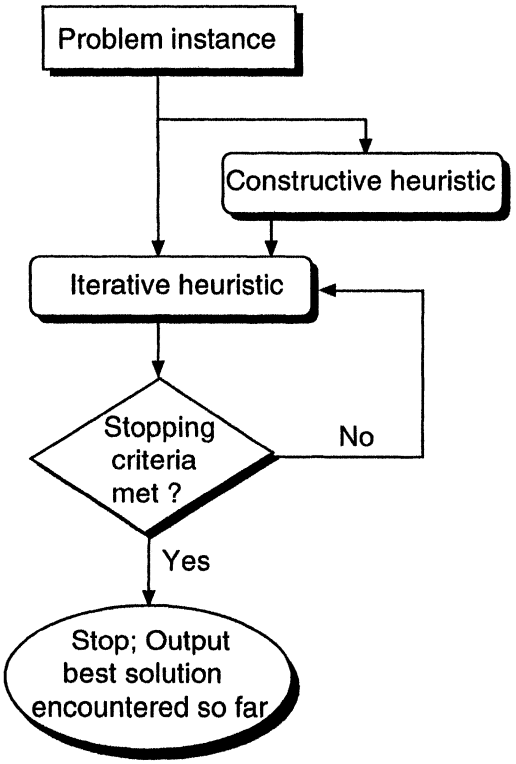

Figure 1.2: General structure combining constructive and iterative heuristics.

Justification of Iterative Improvement Approach

Constructive procedures have the advantage of being faster than iterative procedures such as those described in this book. However, at each decision step, due to its greedy nature, a constructive procedure has only a local view. Therefore, the procedure might reach the point where design constraints are not met. This will require several iterations to attempt various modifications to the solution to bring it to a feasible state. For practical problems, it is unthinkable to manually perform these modifications.

Automatic iterative improvement procedures which combine quality of constructive algorithms and iterative improvement procedures constitute effective approaches to produce feasible solutions with the desired performance. However, in order to speed up the search, care must be taken so that the iterative procedure is tuned to quickly converge to a solution satisfying all design constraints.

1.5 Optimal versus Final Solution

When is a problem solved? A key requirement of a combinatorial optimization algorithm is that it should produce a solution in a reasonably small number of computational steps. The approximation algorithms described in this book are recommended for hard combinatorial optimization problems. It will be unwise to use any of these iterative heuristics to solve problems with known efficient algorithms. For example, one should not use *local search* or *simulated annealing* (Chapter 2) to find the shortest path in a graph; we must instead use one of the known polynomial time algorithms such as *Dijkstra's* algorithm [Dij59].

Exact algorithms for hard problems require in the worst case an exponential (and sometimes a factorial) number of steps to find the optimal solution. For example, suppose that for a given hard problem, a computer is programmed to perform a brute force search for an optimal solution and that the computer is capable of examining one billion solutions per second. Assume that the search space consists of 2^n solutions. Then for $n = 20$ the optimal solution will be found in about 1 *millisecond*. For $n = 100$, the computer will need over 40,000 centuries! The situation would be much worse if we had a problem whose search space consisted of $n!$ solutions. Obviously, a combinatorial optimization problem will not be considered solved if one does not live to see the answer! Hence, a fundamental requirement of any reasonable optimization algorithm is that it should produce an answer to the problem (not necessarily the best) in a reasonably small amount of time. The words *reasonable* and *small* are fuzzy and usually are interpreted differently by different people (depending on the problem, what the answer is needed for, and how soon).

The approximation algorithms described in this book are all iterative, nondeterministic, and keep on searching the solution space until some stopping criteria

are met. Examples of stopping criteria are: (1) the last k iterations did not identify a better solution, (2) a runtime limit has been exceeded, (3) some parameter of the iterative algorithm has reached a threshold limit; and so forth. Once the algorithm stops, it outputs the best solution found. For most practical applications, the runtime of such algorithms may be a few hours. Furthermore, none of these iterative algorithms guarantee finding the optimal solution (if such a solution exists) in a finite amount of time.

1.6 Single versus Multicriteria Constrained Optimization

Constrained optimization consists of finding a solution which satisfies a specified set of constraints and optimizes an analytically defined objective function. A solution which satisfies the problem constraints is a *feasible solution.* If it also optimizes the stated objective function then it is an *optimal solution.* The objective function is to be computed for each combination of the input variables. Values of the input variables change as the search moves from one solution to another. The solution with an optimal value of the objective function is an optimal solution.

A single objective constrained optimization problem consists of the minimization/maximization of a utility function $Cost$ over the set of feasible solutions Ω. For example, for a minimization problem we have something of the following form:

$$\min_{S \in \Omega} Cost(S) \tag{1.1}$$

If $Cost$ is linear and Ω is defined by linear constraints, the problem is *a single objective linear programming problem.* If in addition the problem variables are restricted to be integers, then the problem becomes an *integer programming problem.* In case either the utility function or any of the constraints are nonlinear the problem becomes *a single objective nonlinear optimization problem.*

In most practical cases, optimization problems are multiple objective problems. In such situations, one is typically confronted with several conflicting utility functions $Cost_1, \ldots, Cost_i, \ldots, Cost_n$, that is,

$$\min_{S \in \Omega} Cost_i(S) \qquad 1 \leq i \leq n \tag{1.2}$$

Unlike single-objective optimization problems, no concept of optimal solution is universally accepted for multiobjective optimization. In practical cases, the rating of individual objectives reflects the preference of the decision-maker. At best, a compromise between competing objectives can be expected.

A commonly used approach to transform a multiobjective optimization problem into a single objective optimization problem is to define another utility function as a weighted sum of the individual criteria, that is,

$$Cost(S) = \sum_{i=1}^{n} w_i Cost_i$$

The w_i's are positive weights that reflect the relative importance of criteria or goals in the eyes of the decision maker. More important criteria are assigned higher weights. Usually, the weight coefficients sum to one. Furthermore, prior to computing the weighted utility function, the individual criteria are normalized to fall in the same range.

Another approach to tackle multicriteria optimization problems is to rely on the *ranking* of the individual objectives. In this approach one does not attempt to seek a solution that is minimum with respect to all objectives, since anyhow, in most cases such a solution does not exist; rather the objective function is seen as a vector function. Without loss of generality let us assume that $Cost_i$ is more important than $Cost_{i+1}$, $1 \leq i \leq n-1$. Then a *preference relation* $\prec$ is defined over the solution space Ω as follows:

$$\begin{aligned}
&\forall S \in \Omega, \forall S' \in \Omega : \; S \preceq S' \text{ if and only if} \\
&\exists\, i,\; 1 \leq i \leq n, \text{ such that } Cost_i(S) \leq Cost_i(S'), \text{ and} \\
&\forall\, j < i,\; Cost_j(S) = Cost_j(S')
\end{aligned}$$

The above preference relation defines a partial order on the elements of the state space of feasible solutions Ω.

In many cases, it is not clear how one can balance different objectives by a weight function especially when the various objectives are defined over different domains. Also, it is not always possible to have a crisp ranking of the individual objectives. Another difficulty is that the outcome of such ranking is not always predictable especially when some of the criteria are correlated. Fuzzy logic provides a convenient framework for solving this problem [Zad65, Zad73, Zad75, Zim91]. It allows one to map values of different criteria into linguistic values, which characterize the level of satisfaction of the designer with the numerical values of objectives. Each linguistic value is then defined by a membership function which maps numerical values of the corresponding objective criterion into the interval [0,1]. The desires of the decision maker are conveniently expressed in terms of fuzzy logic rules and fuzzy preference rules. The execution/firing of such rules produces numerical values that are used to decide a solution goodness. In practice, this approach has been proven powerful for finding compromise solutions in different areas of science and engineering [KLS94, LS92, Ped89, RG90, TS85, Wan94, Zim87, Zim91]. We shall address in more detail the subject of using fuzzy logic for multicriteria optimization in Chapter 7.

The algorithms described in this book are general optimization techniques suitable for single as well as multiple objective problems. However, for the sake of simplicity, we shall confine ourselves to single-objective optimization problems.

Interested readers in the general subject of multi-criteria optimization may consult the book by Steuer [Ste86].

1.7 Convergence Analysis of Iterative Algorithms

Unlike constructive algorithms, which produce a solution only at the end of the design process, iterative algorithms operate with design solutions defined at each iteration. A value of the objective function is used to compare results of consecutive iterations and to select a solution based on the maximal (minimal) value of the objective function.

1.7.1 Configuration Graph

The state space being searched can be represented as a directed graph called the *configuration graph*. Let Ω be the set of feasible configurations (states) for some instance of a discrete minimization problem. Ω can be considered as the set of vertices of a directed configuration graph C_G [Len90].

Definition 8 A directed graph C_G=(Ω, E) is called a configuration graph where $S \in \Omega$, and $\aleph(S) = \{T \in \Omega | (S,T) \in E\}$; Ω is the set of legal configurations and E=$\{(S,T) | S \in \Omega, T \in \Omega$ and $T \in \aleph(S)\}$. An edge between two states indicates that they are neighbors. A state S is called a "local minimum" if $Cost(S) \leq Cost(T)$ for all $T \in \aleph(S)$. In addition, if S is an optimal solution then it is called a "global minimum."

Example 1.13 An example of a configuration graph with eight states is given in Figure 1.3. For the moment we will concentrate only on the structure and the values in the circles and ignore the labels on the edges. The numbers in the circles indicate the cost of the configurations. For example, the circle with label 3 represents a state with cost equal to 3. State 3 is a local minimum because it has no neighbors with a lower cost. State 1 is a global minimum because it is a local minimum with the lowest cost, and is the optimal solution. It is not possible to go from state 3 to state 1 without going through states with cost greater than 3, that is, through states with costs 4 and 7, or through 5 (climbing the hill). On the other hand, starting in state 8 we can apply a greedy heuristic that will take us to state 1 (that is, through states 5 and 2, or through state 6).

■

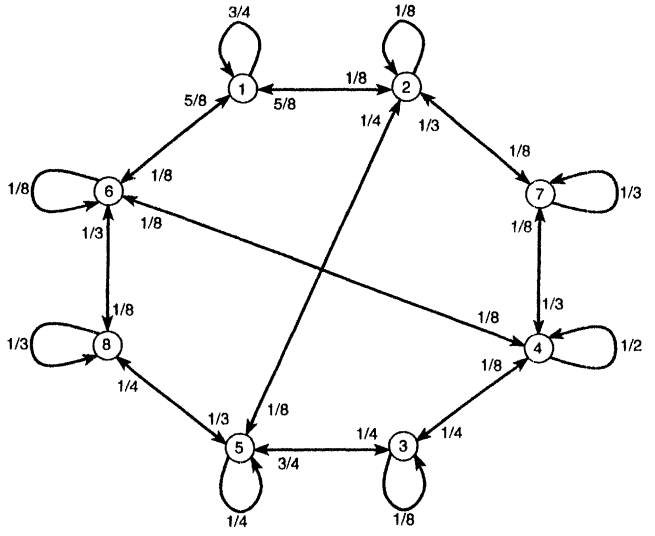

Figure 1.3: An example of a configuration graph.

A "search" from configuration $S \in \Omega$ is a directed path in C_G that starts at S and ends in the solution the search has found. The search is said to be "greedy" if the costs of successive vertices along the search path are decreasing.

The goal of the search is to find a solution that is as close as possible to the optimum. As illustrated in Figure 1.3, greedy heuristics such as *local search* usually lead to local optima and not global optima. The reason is that they provide no mechanism for the search to escape from a local optimum. Two possibilities exist that can help one avoid getting trapped in a local optimum.

1. Accommodate nongreedy search moves, that is, moves in which the cost increases

2. Increase the number of edges in the configuration graph

As for the first possibility, care must be taken to see that such moves are not too frequent. There are probabilistic and deterministic ways of doing so.

In the second possibility, for a configuration graph with many edges and a large neighborhood there are less chances to hit a local optimum. In addition, for a given initial configuration, shorter search paths to the "global optimum" may exist. An extreme case is when C_G is a fully connected directed graph. In that case, every local optimum is also a global optimum and a single step is enough to go from any state to the global optimum. However, the denser the configuration graph is, the more inefficient the search step will be. This is because in each

search step we optimize over the neighborhood of the current configuration, and the larger the neighborhood is, the more time we need to find a good configuration to move to, from where the search can proceed. Therefore, it is important to keep the neighborhood small and not add too many edges to C_G. Researchers have been looking at such issues with mathematical rigor. The mathematical framework used to study the convergence properties of iterative approximation algorithms is the theory of Markov chains.

1.8 Markov Chains

A randomized local search technique operates on a state space. As mentioned above, the search proceeds step by step, by moving from a certain configuration (state) S_i to its neighbor S_j, with a certain probability Prob(S_i, S_j) denoted by p_{ij}. At the end of each step the new state reached represents a new configuration. The states in $\aleph(S_i) = \{S_j \in \Omega | (S_i, S_j) \in E\}$ are said to be connected to S_i by a single move. We can assume that choices of all neighbors out of S_i are independent. The corresponding mathematical structure is a labeled configuration graph with edge labels corresponding to *transition* probabilities. Such a configuration graph is a Markov chain (see Figure 1.3).

1.8.1 Time-Homogeneous Markov Chains

Let $C_G = (\Omega, E)$ be a directed graph with $\Omega = \{S_1, S_2, \ldots, S_i, \ldots, S_n\}$ the set of all possible states, and $Cost : \Omega \rightarrow \Re$, be a function which assigns a real number $Cost_i$ to each state $S_i \in \Omega$, and $p : E \rightarrow [0, 1]$ be an edge-weighting function such that

$$\sum_{S_j \in \aleph(S_i)} p_{ij} = 1 \qquad \forall S_i \in \Omega \tag{1.3}$$

(C_G, p) is a finite *time-homogeneous* Markov chain. In our case $Cost_i$ denotes the cost of configuration S_i, and p_{ij} represents the transition probability from state S_i to S_j. The restriction on p, the edge-weighting function, is that the sum of transition probabilities of edges leaving a vertex add up to unity (p is a probability distribution). Also, in a time-homogeneous Markov chain the transition probabilities are *constant* and independent of past transitions. The configuration graph C_G given in Figure 1.3 represents a time-homogeneous Markov chain.

Two numbers are associated with each pair of states. One is called the *selection probability* or the *perturbation probability*, denoted by p_{ij}, and the other is the acceptance probability A_{ij}.

1.8.2 Perturbation Probability

The number associated with each pair of states (edge label denoted by p_{ij}) is called the *perturbation probability*. This number actually gives the probability of generating a configuration S_j from S_i.

Let $\aleph(S_i)$ be the configuration subspace for state S_i which is defined as the space of all configurations that can be reached from S_i by a single perturbation. For pairs of states connected by at least a single move the perturbation probability p_{ij} is never zero. The probability p_{ij} depends on the structure of the configuration graph, and in the simplest case it is defined as follows:

$$p_{ij} = \begin{cases} \frac{1}{|\aleph(S_i)|} & \text{if } S_j \in \aleph(S_i) \\ 0 & \text{if } S_j \notin \aleph(S_i) \end{cases} \tag{1.4}$$

This is a uniform probability distribution for all configurations in the subspace. The probabilities p_{ij} can also be represented using a matrix P known as the *generation matrix* or *perturbation matrix*. Matrix P is a stochastic matrix.[4]

1.8.3 Ergodic Markov Chains

A Markov chain is called ergodic if and only if it is

1. irreducible, that is, all states are reachable from all other states;
2. aperiodic, that is, for each state, the probability of returning to that state is positive for all steps;
3. recurrent, that is, for each state of the chain, the probability of returning to that state at some time in the future is equal to one; and
4. non-null, that is, the expected number of steps to return to a state is finite.

Let $\pi(t) = (\pi_1(t), \pi_2(t), \ldots, \pi_i(t), \ldots, \pi_n(t))$ be the probability state vector, where $\pi_i(t)$ is the probability of being in state S_i at time t (iteration t).

The probability transition matrix P is used to describe how the process evolves from state to state. If at step t, the probability state vector is $\pi(t)$, then the probability state vector one step later is given by

$$\pi(t+1) = \pi(t) \cdot \mathrm{P} \tag{1.5}$$

Hence, the probability $\pi_i(t+1)$ of being in state S_i at step $t+1$, is given by

$$\pi_i(t+1) = \sum_{j=1}^{n} \pi_j(t) \cdot p_{ji} \tag{1.6}$$

[4] A square matrix whose entries are non-negative, and whose row sums are equal to unity, is called a *stochastic* matrix. Sometimes an additional condition is that the column sums are also not zero.

For an ergodic Markov chain, the state probability vector changes at each step and is guaranteed to converge to a limit probability vector $\pi = (\pi_1, \ldots, \pi_i, \ldots)$, that is, $\lim_{t\to\infty} \pi(t) = \pi$. The probability state vector π, which no longer depends on the time step, is the steady-state distribution of the search process.

A fundamental theorem on Markov chains states that an ergodic Markov chain has a unique stationary distribution π which is a solution to the following equation [Kle75].

$$\pi = \pi \cdot \mathrm{P} \tag{1.7}$$

The stationary distribution can also be obtained by finding the stationary matrix $\mathrm{P_S}$ given by

$$\mathrm{P_S} = \lim_{k\to\infty} \mathrm{P}^k \tag{1.8}$$

where P^k is the k-fold matrix product of P with itself. If the Markov chain is ergodic, then $\mathrm{P_S}$ will have the characteristic that all its rows are identical. We say that the Markov chain has converged to its stationary distribution.

If we start the Markov chain in any state S_i, it will converge to the distribution given by $\lim_{k\to\infty} I_i \cdot \mathrm{P}^k$, where I_i is the i^{th} unit vector (1 in the i^{th} position and 0 elsewhere). Since $I_i \cdot \mathrm{P_S}$ is equal to $I_j \cdot \mathrm{P_S}$ for all i, j, at steady state, the probability of being in any state is independent of the initial state. We will now illustrate the above concepts with examples.

Example 1.14 Figure 1.3 is an example of an ergodic Markov chain. The labels on the edges connecting two states S_i and S_j indicate the transition probability from state S_i to state S_j. The corresponding transition matrix is given below.

$$\mathrm{P} = \begin{Bmatrix} \frac{3}{4} & \frac{1}{8} & 0 & 0 & 0 & \frac{1}{8} & 0 & 0 \\ \frac{5}{8} & \frac{1}{8} & 0 & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 \\ 0 & 0 & \frac{1}{8} & \frac{1}{8} & \frac{3}{4} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{4} & \frac{1}{2} & 0 & \frac{1}{8} & \frac{1}{8} & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & \frac{1}{4} \\ \frac{5}{8} & 0 & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 & \frac{1}{8} \\ 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \end{Bmatrix}$$

Let us raise the matrix P to a large power, say 100. Using Mathematica [Wol91], this can be achieved by the command

```
Q=MatrixPower[P,100];
Print[MatrixForm[N[Q]]]
```

which produces the following output.

$$P^{100} = \left\{ \begin{array}{cccccccc} 0.5334 & 0.1106 & 0.0385 & 0.0562 & 0.0786 & 0.1028 & 0.0313 & 0.0488 \\ 0.5334 & 0.1106 & 0.0385 & 0.0562 & 0.0786 & 0.1028 & 0.0313 & 0.0488 \\ 0.5334 & 0.1106 & 0.0385 & 0.0562 & 0.0786 & 0.1028 & 0.0313 & 0.0488 \\ 0.5334 & 0.1106 & 0.0385 & 0.0562 & 0.0786 & 0.1028 & 0.0313 & 0.0488 \\ 0.5334 & 0.1106 & 0.0385 & 0.0562 & 0.0786 & 0.1028 & 0.0313 & 0.0488 \\ 0.5334 & 0.1106 & 0.0385 & 0.0562 & 0.0786 & 0.1028 & 0.0313 & 0.0488 \\ 0.5334 & 0.1106 & 0.0385 & 0.0562 & 0.0786 & 0.1028 & 0.0313 & 0.0488 \\ 0.5334 & 0.1106 & 0.0385 & 0.0562 & 0.0786 & 0.1028 & 0.0313 & 0.0488 \end{array} \right\}$$

■

From the above example we note that starting from any initial state, say state 3, denoted by the unit vector $I_3 = [0, 0, 1, 0, 0, 0, 0, 0]$, the probability of being in any state after 100 state transitions (or moves) is given by $I_3 \cdot P^{100}$, that is, the third row of the matrix P^{100}. The probability of being in state 1 after 100 moves is 0.5334, of being in state 2 is 0.1106, and so on. Note that in this case the value 100 can be defined as *large*. Sometimes the matrix will have to be raised to a larger power to get the stationary distribution.

Observe that since all rows are identical, irrespective of which row we start our search, we will always get the same probability of being in any state. We can also verify Equation 1.6. For example, when $i = 4$,

$$\pi_4 = \sum_{j=1}^{8} \pi_j \cdot p_{j4} = \pi_3 \cdot p_{34} + \pi_4 \cdot p_{44} + \pi_6 \cdot p_{64} + \pi_7 \cdot p_{74}$$

$$\pi_4 = \frac{0.0385}{8} + \frac{0.0562}{2} + \frac{0.1028}{8} + \frac{0.0313}{3} = 0.0562$$

which is the same as column 4 of our matrix P^{100} which gives the value of π_4, the steady-state probability of being in state S_4.

Example 1.15 For the same ergodic Markov chain of the previous example, the stationary distribution can be accurately obtained by solving the set of linear equations $\pi = \pi \cdot P$, and the equation $\sum_{i=1}^{n} \pi_i = 1$. Again, using Mathematica, this can be obtained as follows.

Solution:
```
Solve
 [{
 P[[1,1]]p1+P[[2,1]]p2+P[[3,1]]p3+P[[4,1]]p4+P[[5,1]]p5
                             +P[[6,1]]p6+P[[7,1]]p7+P[[8,1]]p8==p1,
 P[[1,2]]p1+P[[2,2]]p2+P[[3,2]]p3+P[[4,2]]p4+P[[5,2]]p5
                             +P[[6,2]]p6+P[[7,2]]p7+P[[8,2]]p8==p2,
 P[[1,3]]p1+P[[2,3]]p2+P[[3,3]]p3+P[[4,3]]p4+P[[5,3]]p5
                             +P[[6,3]]p6+P[[7,3]]p7+P[[8,3]]p8==p3,
 P[[1,4]]p1+P[[2,4]]p2+P[[3,4]]p3+P[[4,4]]p4+P[[5,4]]p5
```

```
                        +P[[6,4]]p6+P[[7,4]]p7+P[[8,4]]p8==p4,
P[[1,5]]p1+P[[2,5]]p2+P[[3,5]]p3+P[[4,5]]p4+P[[5,5]]p5
                        +P[[6,5]]p6+P[[7,5]]p7+P[[8,5]]p8==p5,
P[[1,6]]p1+P[[2,6]]p2+P[[3,6]]p3+P[[4,6]]p4+P[[5,6]]p5
                        +P[[6,6]]p6+P[[7,6]]p7+P[[8,6]]p8==p6,
P[[1,7]]p1+P[[2,7]]p2+P[[3,7]]p3+P[[4,7]]p4+P[[5,7]]p5
                        +P[[6,7]]p6+P[[7,7]]p7+P[[8,7]]p8==p7,
P[[1,8]]p1+P[[2,8]]p2+P[[3,8]]p3+P[[4,8]]p4+P[[5,8]]p5
                        +P[[6,8]]p6+P[[7,8]]p7+P[[8,8]]p8==p8,
p1+p2+p3+p4+p5+p6+p7+p8==1},{p1,p2,p3,p4,p5,p6,p7,p8}];
Simplify[%]
```

Here `P[[i,j]]` represents p_{ij} the elements of matrix `P`, and pi represents π_i (i=1,2,...,8). The distribution thus obtained is

$$\pi = \left(\frac{3020}{5662} \quad \frac{626}{5662} \quad \frac{218}{5662} \quad \frac{318}{5662} \quad \frac{445}{5662} \quad \frac{582}{5662} \quad \frac{177}{5662} \quad \frac{276}{5662} \right)$$

That is,

$$\pi = \left(0.5334 \quad 0.1106 \quad 0.0385 \quad 0.0562 \quad 0.0786 \quad 0.1028 \quad 0.0313 \quad 0.0488 \right)$$

Note that this is identical to one of the rows of our matrix P^{100}.

■

1.8.4 Acceptance Probability

In many cases, the transition probabilities of a random process depend on a control parameter T which is a function of time. The probabilities now take the form $f(Cost_i, Cost_j, T)$, where T is a parameter that depends on the step number of the Markov chain, and $Cost_i$ and $Cost_j$ are the costs of the current and next states, respectively. The corresponding Markov chains are called *time-inhomogeneous* Markov chains. Let $\Delta Cost_{ij} = Cost_j - Cost_i$. Then the acceptance probability A_{ij} may be defined as

$$A_{ij}(T) = \begin{cases} f(Cost_i, Cost_j, T) & \text{if } \Delta Cost_{ij} > 0 \\ 1 & \text{if } \Delta Cost_{ij} \leq 0 \end{cases} \tag{1.9}$$

Thus the probability that the generated new state will be the next state depends on its cost, the cost of the previous state, and the value of the control parameter T. We always accept cost-improving moves. A move that deteriorates the cost will be accepted with a probability $f(Cost_i, Cost_j, T)$. The sequence of states thus generated corresponds to a *time-inhomogeneous Markov chain*. We have a Markov chain because of the important property that the next state depends only on where we are now and does not depend on the states that have preceded the current state. Therefore, for this time-inhomogeneous Markov chain, given S_i as the current state, the probability $\Theta_{ij}(T)$ to transit to state S_j is defined as follows:

$$\Theta_{ij}(T) = \begin{cases} A_{ij}(T)p_{ij} & \text{if } i \neq j \\ 1 - \sum_{k,k\neq i} A_{ik}(T)p_{ik} & \text{if } i = j \end{cases} \tag{1.10}$$

where p_{ij} is the perturbation probability, that is, the probability of generating a configuration S_j from configuration S_i (usually independent of T); $A_{ij}(T)$ is the acceptance probability, (see Equation 1.9), i.e., the probability of accepting configuration S_j if the system is in configuration S_i; and T is the control parameter.

The transition probabilities for a certain value of T can be conveniently represented by a matrix $\Theta(T)$, called the *transition matrix*. The probabilities $A_{ij}(T)$ can also be represented using a matrix $A(T)$ (*acceptance matrix*). Like the perturbation matrix P, the transition matrix Θ is also stochastic. The acceptance matrix $A(T)$ however, is not stochastic.

1.8.5 Transition Probability

Let $\Theta_{ij}(T)$ be the transition probability from state S_i to state S_j for a particular value of the control parameter T, that is, $\Theta_{ij}(T) = p_{ij} \cdot A_{ij}(T)$. At a particular value of the parameter T, the transition matrix $\Theta(T)$ is constant and thus corresponds to a *homogeneous* Markov chain.

Under the assumption that all states of current neighborhood $\aleph(S_i)$ are equally likely, p_{ij} is equal to the following:

$$p_{ij} = \frac{1}{|\aleph(S_i)|}$$

Therefore, in summary, we have the following expressions for the probabilities $\Theta_{ij}(T)$:

$$\Theta_{ij}(T) = \begin{cases} \frac{1}{|\aleph(S_i)|} & \text{if } \Delta Cost_{ij} \leq 0 \quad S_j \in \aleph(S_i) \\ \frac{1}{|\aleph(S_i)|} f(Cost_i, Cost_j, T) & \text{if } \Delta Cost_{ij} > 0 \quad S_j \in \aleph(S_i) \\ 1 - \sum_{k,k\neq i} p_{ik}A_{ik}(T) & \text{if } i = j \quad S_j \in \aleph(S_i) \\ 0 & S_j \notin \aleph(S_i) \end{cases} \tag{1.11}$$

As we shall see, in the following chapter in the case of the simulated annealing algorithm, a steady-state distribution $\pi(T)$ exists for each value of the parameter T, provided T is maintained constant for a large enough number of iterations. The steady-state probability vector $\pi(T)$ satisfies the following equation:

$$\pi(T) = \pi(T) \cdot \Theta(T)$$

Furthermore, following an adequate updating schedule of the parameter T, the process will converge to the steady state whose stationary distribution π (also called *optimizing distribution*) satisfies the following equality:

$$\pi = \pi \cdot \Theta$$

1.9 Parallel Processing

In this section, we introduce the necessary terminology that will be used in the discussion of the parallel implementations of the various iterative algorithms that are described in this book.

Need for Parallel Processing

Exact as well as approximate iterative algorithms for hard problems have large runtime requirements. There is ever increasing interest in the use of parallel processing to obtain greater execution speed. Parallel computation offers a great opportunity for sizable improvement in the solution of large and hard problems that would otherwise have been impractical to tackle on a sequential computer. A general problem with parallel computers is that they are harder to program. Every computer scientist knows how to design and implement algorithms that run on sequential computers. In contrast, only relatively few have the skill of designing and implementing parallel algorithmic solutions.

A *parallel computer* is one that consists of a collection of processors, which can be programmed to cooperate together to solve a particular problem. In order to achieve any improvement in performance, the processors must be programmed so that they work concurrently on the problem. The goal, of course, is usually to reach, in much less time, a solution of similar quality to that obtained from running a sequential algorithm. Actually, the ratio of the sequential runtime to parallel runtime is an important performance measure called the *speed-up*. Sometimes, parallel search is used to find a better solution in the same time required by the sequential algorithm rather than to reach a similar quality solution in shorter time.

Parallel Algorithm Evaluation Measures

Let A_1 and A_p, respectively, be a sequential algorithm and a parallel algorithm for p processors to solve the same problem. The goodness of the parallel algorithm is usually characterized by several measures, such as

1. The time t_p taken to run A_p.
2. The space s_p required to run A_p.

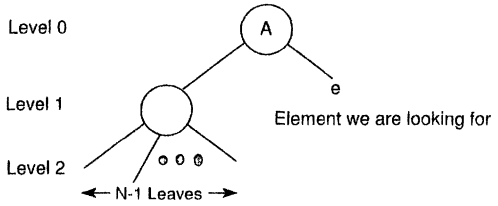

Figure 1.4: Imbalanced tree. Sequential depth first for element e would require $N + 1$ operations. If the search is split among two processors, each looking in a subtree, then element e will be returned in two steps.

3. *Speed-up*: how much did we gain in speed by using p processors. If t_1 is the runtime of the sequential algorithm, then the *speed-up* $\mathcal{S}_p$ is defined as follows:

$$\mathcal{S}_p = \frac{t_1}{t_p} \tag{1.12}$$

Normally, $0 < \mathcal{S}_p \leq p$. However, this is not always the case. For instance, assume that we would like to look for a particular node in a tree using the *depth-first* search algorithm. Assume that the tree has N nodes. Then the maximum time that will be taken by the sequential *depth-first* algorithm will be $t_1 = N + 1$. This happens in case the tree is imbalanced with all the $N - 1$ nodes in the left subtree and being at the second level (root at level 0), and the node we are looking for is the only node in the right subtree (Figure 1.4). Suppose we have two processors and that each processor takes a subtree and expands it. In that case $t_2 = 2$. Therefore the *speed-up* is $\mathcal{S}_p = \frac{N+1}{2}$, which is greater than 2, the number of processors. One might wonder why this is happening? The answer is simply because in the first place one should not have used *depth-first* search to locate a particular node in a tree, that is, one must use the best possible sequential algorithm for the problem at hand. The parallel algorithm should not mimic the sequential algorithm nor should the sequential algorithm be a simple serialization of the parallel algorithm.

The above definition of *speed-up* applies to deterministic algorithms only. For a nondeterministic iterative algorithm such as simulated annealing, *speed-up* is defined in a different manner. It is equal to the number of parallel tasks into which each move is divided [RK86]. Sometimes, the number of processors that are concurrently working is taken as a measure of the speed-up achieved.

Another suggested definition [DRKN87] bears closer resemblance to the definition of *speed-up* for deterministic algorithms. *Speed-up* is defined as the ratio of the execution time of the serial algorithm to that of the parallel

implementation of the algorithm, averaged over several runs and various final values of the cost function.

4. *Efficiency* E_p: this performance measure indicates how well we are using the p processors,

$$E_p = \frac{\mathcal{S}_p}{p} \tag{1.13}$$

Under normal conditions, $0 < E_p \leq 1$.

5. *Isoefficiency* ie_p: this measure is an estimate of the efficiency of the algorithm as we change the number of processors, while maintaining the problem instance fixed. It is desirable to have parallel algorithms with ie_p close to p (linear in p). Hence, we guarantee no processor starvation as we increase the number of processors.

Amdahl's Law

Amdahl's law was introduced to convince the computing community that parallelism is not good after all.

Let f be the fraction parallelized in the algorithm. Then, with p processors, the best possible parallel algorithm would require the following runtime given by

$$t_p = (1 - f)t_1 + \frac{f}{p}t_1 \tag{1.14}$$

Therefore, the maximum speed-up in this case would be

$$\mathcal{S}_p = \frac{1}{(1 - f) + \frac{f}{p}} \tag{1.15}$$

For example, for $f = 0.5$, according to Amdahl's law, the speed-up cannot exceed 2 even if an infinite number of processors are made available! This is indeed a disturbing conclusion both to the manufacturers of parallel machines, as well as to researchers in parallel algorithms. Fortunately, a closer examination of Amdahl's law uncovers a major flaw. The main problem with Amdahl's law is that it does not capture how much time the algorithm spends in the parallelized fraction of the code. If, for example, 90 percent of the time is spent in the parallelized piece of code, then the speed-up can be as high as $0.9p$. Hence, parallelism can indeed be extremely good!

Parallel Computer Models and Properties

There are several ways one can classify computers. A possible classification is that of *multiprocessor* versus *multicomputer*. A *multiprocessor* machine is a computer

with several processors that are tightly coupled, that is, they either have a shared memory or a shared memory address space. When programming a multiprocessor machine one does not have to explicitly indicate from which processor he or she wants the data. An example of such a machine is the Butterfly [Lei92].

A *multicomputer* also consists of several processors; however, the processors have no shared memory or shared address space. When programming a multicomputer, one has to explicitly request/send data from/to a given processor. An example of a multicomputer machine is the NCUBE [Lei92]. In practice, we may find combinations that fit in both categories. Both classes of parallel computer models are illustrated in Figure 1.5.

A classification of parallel machine models based on the work of Flynn [Fly66] distinguishes between the parallel machine models on the basis of the number of instructions and data streams concurrently accepted by the machine. Flynn identified four classes of parallel machine models.

1. SISD—Single Instruction, Single Data Stream
   Here one instruction at a time is executed on one data set at a time. The classic sequential Von Neumann machines fall into this class.

2. SIMD—Single Instruction, Multiple Data Stream
   For this class, one instruction at a time is executed concurrently on several data sets. Examples of machines that fall into this class are *vector computers* and *array processors*.

3. MISD—Multiple Instructions, Single Data Stream
   These machines are capable of executing concurrently several instructions at a time on one data set.

4. MIMD—Multiple Instructions, Multiple Data Stream
   Multiple instructions at a time are concurrently executed on multiple data sets. MIMD machines can be either synchronous or asynchronous. The processors of a synchronous MIMD machine are synchronized on a global clock, thus forcing the execution of each successive group of instructions simultaneously. For asynchronous MIMD machines the processors execute the instructions independently of each other. Typical examples of MIMD machines are hypercube computers (such as the NCUBE) [Lei92].

The four machine models are illustrated in Figure 1.6. The reader should note, however, that machines that perform some lower level of parallelism, such as *pipelining*, do not fit into Flynn's classification.

1.10 Summary and Organization of the Book

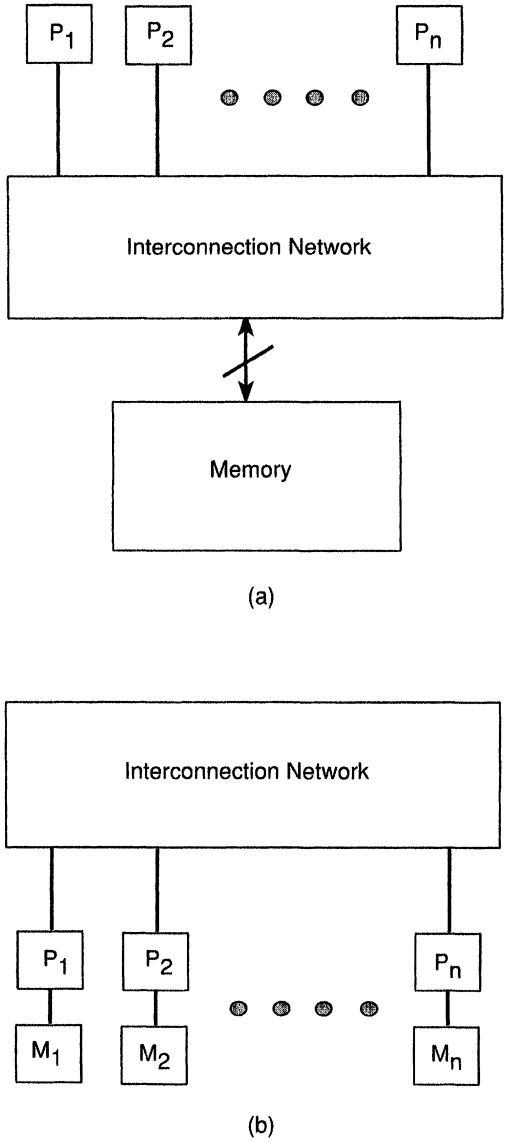

Figure 1.5: Models of parallel computers: (a) tightly coupled multiprocessor; (b) loosely coupled multicomputer.

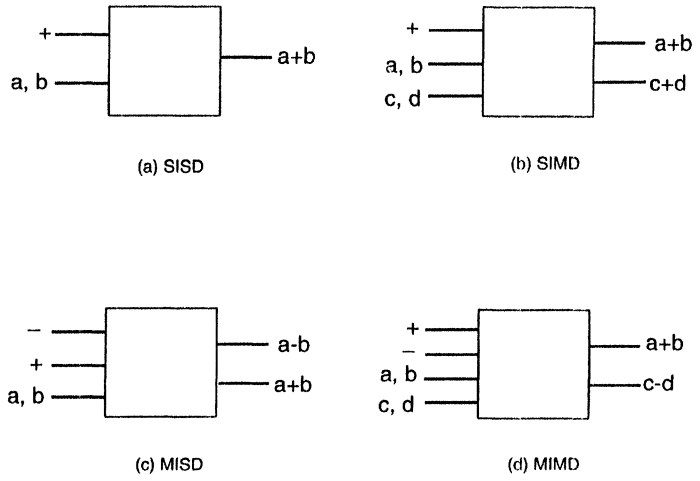

Figure 1.6: Classification of parallel machine models: (a) SISD—Single Instruction, Single Data Stream; (b) SIMD—Single Instruction, Multiple Data Stream; (c) MISD—Multiple Instructions, Single Data Stream; (d) MIMD—Multiple Instructions, Multiple Data Stream.

This chapter has introduced basic concepts of combinatorial optimization and algorithm complexity. There are two general categories of algorithms for combinatorial optimization: (1) exact or full-enumeration algorithms, and (2) approximation algorithms, also known as heuristics. In this book, we are concerned with hard problems. For such class of problems, exact algorithms are impractical as they have prohibitive runtime requirements. Approximation algorithms constitute the only practical alternative solution method.

Approximation algorithms can further be classified into problem-specific heuristics and general heuristics. As their names indicate, problem-specific algorithms are tailored to one particular problem. A heuristic designed for one particular problem would not work for a different problem. General heuristics on the other hand can be easily tailored to solve (reasonably well) any combinatorial optimization problem. There has been increasing interest in such heuristic search algorithms.

In the following chapters the reader will find detailed descriptions of five well-thought-out general iterative approximation algorithms, namely, *simulated annealing*, *genetic algorithms*, *tabu search*, *simulated evolution*, and *stochastic evolution*. Simulated annealing mimics the thermodynamic process of annealing. Genetic algorithms, simulated evolution, and stochastic evolution simulate biological processes according to the Darwinian theory of evolution. Tabu search attempts to imitate intelligent search processes through the use of a memory com-

ponent in order to learn from its (long- or short-term) past, thus making better search decisions.

This is the only book that describes these five heuristics in a single volume. Two of these heuristics have been the subject of several books [AK89, Aze92, Dav91, Gol89, OvG89]. The tabu search algorithm has been widely used in the literature [GL97, Ree95]. However, the remaining two heuristics, simulated evolution and stochastic evolution, have not witnessed yet similar success. We believe that simulated evolution and stochastic evolution are extremely effective general combinatorial optimization techniques that deserve much more attention than they have received. The objective of this book is to provide a uniform treatment of all these techniques.

The book has seven chapters organized around these five iterative heuristics. The purpose of this introductory chapter has been to motivate the student to study and use the general iterative approximate algorithms. The chapter also introduced the basic terminology needed in the remaining chapters.

The following five chapters are dedicated to the five selected heuristics. For each search heuristic, we start by providing a historical account of the search method. We then describe the basic algorithm and its parameters and operators. This will be followed by addressing the convergence aspects of the algorithm. Examples are included to illustrate the operation of the heuristic on a number of practical problems. Parallelization strategies of the algorithm will also be presented. In each chapter, the final section, "Conclusions and Recent Work," discusses several other relevant variations of the described techniques (recent or otherwise).

Finally, in Chapter 7, we shall touch upon some work that has been reported in the area of hybridization. This area concerns combining key features of various heuristics to design new effective search techniques. In this chapter we also discuss multiobjective optimization, and give a brief overview of how fuzzy logic is used to represent multiobjective cost functions. Optimization using neural networks, and other relevant issues such as solution quality, measure of performance, and so forth are also covered.

The body of available literature on some of the techniques, namely, *simulated annealing* and *genetic algorithms*, is enormous. Therefore, it is impossible to describe and discuss every single reported work. Rather, we concentrate on describing those works we are most familiar with and which we feel are the most significant.

References

[AK89] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, 1989.

[Aze92] R. Azencott, editor. *Simulated Annealing Parallelization Techniques*. John Wiley & Sons, 1992.

[BGAB83] L. Bodin, L. Goldin, A. Assad, and M. Ball. Routing and scheduling of vehicles and crews: The state of art. *Computers & Operations Research*, 10:63–211, 1983.

[Dav91] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DLSS88] M. Desrochers, J. K. Lenstra, M. W. P. Savelsbergh, and F. Soumis. Vehicle routing with time windows: Optimization and approximation. In B. L. Goldin and A. A. Assad, editors, *Vehicle Routing: Methods and Studies*, pages 65–84. North Holland, Amsterdam, 1988.

[DRKN87] F. Darema-Rogers, S. Kirkpatrick, and V. A. Norton. Parallel algorithms for chip placement by simulated annealing. *IBM Journal of Research and Development*, 31:391–402, May 1987.

[Fly66] M. J. Flynn. Very high-speed computing systems. *Proceedings of IEEE*, 54:1901–1909, 1966.

[Fou84] L. R. Foulds. *Combinatorial Optimization for Undergraduates*. Springer-Verlag, 1984.

[GJ79] M. Garey and D. Johnson. *Computer and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[GL97] F. Glover and M. Laguna. *Tabu Search*. Kluwer, MA, 1997.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[GPR94] B.-L. Garica, J.-Y. Potvin, and J.-M. Rousseau. A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers & Operations Research*, 21(9):1025–1033, November 1994.

[HS84] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, MD, 1984.

[Hu82] T. C. Hu. *Combinatorial Algorithms.* Addison-Wesley, 1982.

[Kle75] L. Kleinrock. *Queueing Systems, Volume I: Theory.* Wiley-Interscience, New York, 1975.

[KLS94] E. Kang, R. Lin, and E. Shragowitz. Fuzzy logic approach to VLSI placement. *IEEE Transactions on VLSI Systems*, 2:489–501, Dec. 1994.

[Law76] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* B. G. Teubner and John Wiley & Sons, 1990.

[LS92] R. Lin and E. Shragowitz. Fuzzy logic approach to placement problem. *Proceedings of the ACM/IEEE 29th Design Automation Conference*, pages 153–158, 1992.

[NSS89] S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. *Journal of Computer-Aided Design*, 1:1–23, 1989.

[OvG89] R.H.J.M. Otten and L.P.P.P. van Ginneken. *The Annealing Algorithm.* Kluwer, MA, 1989.

[Ped89] W. Pedrycz. *Fuzzy Control and Fuzzy Systems.* John Wiley & Sons, New York, 1989.

[PS82] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[Ree95] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Optimization Problems.* McGraw-Hill, Europe, 1995.

[RG90] M. Razaz and J. Gan. Fuzzy set based initial placement for IC layout. *Proceedings of the European Design Automation Conference*, pages 655–659, 1990.

[RK86] R. A. Rutenbar and S. A. Kravitz. Layout by simulated annealing in a parallel environment. *Proceedings of International Conference on Computer Design: VLSI in Computers & Processors, ICCD-86*, pages 434–437, 1986.

[Sah81] S. Sahni. *Concepts in Discrete Mathematics*. The Camelot Publishing Company, MN, 1981.

[SB80] S. Sahni and A. Bhatt. The complexity of design automation problems. *Proceedings of the Design Automation Conference*, pages 402–410, 1980.

[Ste86] R. E. Steuer. *Multiple Criteria Optimization: Theory, Computation, and Application*. John Wiley & Sons, 1986.

[Tai93] E. Taillard. Parallel iterative search methods for the vehicle routing problem. *Networks*, 23:661–673, 1993.

[TS85] T. Takagi and M. Sugeno. Fuzzy identification of systems and its applications to modeling and control. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(1), Jan 1985.

[Wan94] L.-X. Wang. *Adaptive Fuzzy Systems and Control*. Prentice-Hall, 1994.

[Wol91] S. Wolfram. *Mathematica–A System for Doing Mathematics by Computer*. Addison-Wesley, 1991.

[Zad65] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[Zad73] L. A. Zadeh. Outline of a new approach to the analysis of complex systems and decisions processes. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-3(1):28–44, Jan 1973.

[Zad75] L. A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning. *Information Sciences*, 8:199–249, 1975.

[Zim87] H. J. Zimmermann. *Fuzzy Sets, Decision Making, and Expert Systems*, 2nd Ed. Kluwer Academic Publishers, 1987.

[Zim91] H. J. Zimmermann. *Fuzzy Set Theory and Its Applications*, 2nd Ed. Kluwer, 1991.

Exercises

Exercise 1.1
Stirling approximation of the factorial function is as follows [Sah81]:

$$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n(1 + \frac{1}{12n})$$

Using the above approximation, show that for large n,

$$\binom{n}{\frac{n}{2}} \approx 2^n$$

Exercise 1.2
Given a set of n distinct positive integers $X = \{x_1, x_2, \ldots, x_n\}$. The objective is to partition the set into two subsets Y of size k and Z of size $n - k$ ($1 \leq k \leq \frac{n}{2}$) such that the difference between the sums of the two subsets is minimized. This problem is known as the set partitioning problem.

1. For a fixed k, how many partitions exist?
2. How many partitions are there for all possible values of k.
3. Assume that $k = \frac{n}{2}$. One possible heuristic algorithm for this problem is the following.

```
Algorithm SetPartition(X, Y, Z);
Begin
  Sort Array X[1:n] in descending order;
  For i = 1 To n Do
  Begin
    Assign X[i] to the set which has currently the smaller sum;
  EndFor;
  Return (Y, Z)
End Algorithm;
```

   (a) Find the time complexity of the above algorithm.
   (b) Implement the above heuristic and experiment with it on several randomly generated problem instances.
   (c) Generalize the above heuristic to partition the set X for any value of $k \leq \frac{n}{2}$.

Exercise 1.3
1. Experiment with the *local search* heuristic (page 11) on a number of randomly generated instances of the set partitioning problem. Compare quality of solutions obtained with those of the greedy heuristic outlined in Exercise 1.2.

2. The *Improve* subroutine can follow one of two strategies: (a) *first-improvement* strategy where the first favorable cost change is accepted, or (b) the *steepest descent* strategy where the entire neighborhood is searched, and then a solution with lowest cost is selected. Discuss the merits and demerits of both strategies.
3. Experiment with both strategies and report the effect of each strategy on quality of solution, runtime, and so forth.

Exercise 1.4
Given a graph $G(V, E)$ with n nodes and m edges. Show that there are at most:

1. 2^m subsets of E that might be *edge coverings*.
2. 2^m possible *cuts*.
3. 2^m possible *paths*.
4. $(2m)!$ possible *tours*.
5. n^{n-2} possible *spanning trees*.

Exercise 1.5
For an $n \times n$ quadratic assignment problem (QAP), show that there are at most $n!$ feasible solutions.

Exercise 1.6
Write a program to generate a random connected graph. The inputs to the program are the number of nodes, (an even number) and, the range of degree of the nodes (for example, between 2 and 5). In a graph $G(V, E)$, the *degree* d_i of a node $i \in V$ is defined as the number of (other) nodes i is connected to.

Exercise 1.7

1. Given a graph of $2 \cdot n$ elements, show that the number of *balanced* two-way partitions is $P(2n) = \frac{(2n)!}{2 \cdot n! \cdot n!}$.
2. Use Stirling's approximation for $n!$ to simplify the expression for $P(2n)$. Express $P(2n)$ using the Big-Oh notation.
3. A *brute force* algorithm for the two-way partition problem enumerates all the $P(2n)$ solutions and selects the best. Write a computer program which implements such a brute force algorithm . What is the time complexity of your program?

4. Plot the running time of the *brute force* partition program for $n = 1, \ldots, 10$. If the maximum permitted execution time for the program is 24 hours, what is the maximum value of n for which your program can run to completion?

Exercise 1.8

Suppose we are given a graph with $2n$ nodes, and a matrix C that specifies the connectivity information between nodes; for example, c_{ij} gives the number of connections between elements i and j. Let A and B represent a balanced partition of the graph, that is, $|A| = |B| = n$. Use the *local search* algorithm to divide the graph into a balanced partition such that the cost of edges cut is minimum. Experiment with the following neighbor functions.

1. Pairwise exchange. Here two elements, one from each partition are swapped to disturb the current solution.
2. Swap a subset of elements selected from each partition.
3. Select for swap those elements whose contribution to the external cost is high, or those that are internally connected to the least number of vertices.

Exercise 1.9

1. Repeat Exercise 1.8 using instead the *random-walk* search heuristic given below.

```
Algorithm RandomWalk(S0);
Begin
  S = S0; BestS = S;
  BestCost = Cost(S0);
  Repeat
    S = Perturb(S); /* Generate another random feasible solution */
    CostS = Cost(S)
    If CostS < BestCost Then
       BestCost = CostS;
       BestS = S
    EndIf
  Until time-to-stop;
  Return (BestS)
End Algorithm;
```

2. Compare the *local-search* and *random-walk* heuristics.

Exercise 1.10

A variation of the *random-walk* heuristic is to adapt a *steepest descent* strat-

egy. That is, a new feasible solution is accepted only if it improves the cost. A random search of this type is known as *random-sampling*.

Algorithm RandomSampling(S_0);
Begin
  $S = S_0$; $BestS = S$;
  $CostS = Cost(S_0)$;
  $BestCost = Cost(S_0)$;
  Repeat
    $NewS = Perturb(S)$; /* Generate another random feasible solution */
    $NewCost = Cost(NewS)$;
    If $NewCost < CostS$ **Then**
      $BestCost = NewCost$;
      $CostS = NewCost$;
      $S = NewS$
    EndIf
  Until *time-to-stop*;
  Return ($BestS$)
End Algorithm;

Using the problem instances and perturbation functions suggested in Exercise 1.8, do the following:

1. Experiment with *random-sampling* and compare it with *random-walk*.
2. Compare *random-sampling* with *local-search*.

Exercise 1.11
Another iterative search heuristic is known as *sequence-heuristic* [NSS89]. In this heuristic a new solution with a higher cost (uphill move) is accepted if the last k perturbations on current solution S failed to generate a $NewS$ with $Cost(NewS) < Cost(S)$. The *sequence-heuristic* algorithm is given below.

Algorithm SequenceHeuristic(S_0, L_0);
/* S_0 is initial solution and L_0 is initial sequence length. */
Begin
  $S = S_0$; $BestS = S$;
  $CostS = Cost(S_0)$;
  $BestCost = Cost(S_0)$;
  $L = L_0$; /* initial sequence length */
  Repeat
    $length = 0$; /* current length of bad perturbations */
    Repeat

```
    NewS = Perturb(S);
    NewCost = Cost(NewS);
    If NewCost < CostS Then
      CostS = NewCost;
      S = NewS;
      If NewCost < BestCost Then
        BestCost = NewCost;
        BestS = NewS
      EndIf
    Else length = length + 1
    EndIf
  Until length > L;
  L = UpdateLength(L)
Until time-to-stop;
Return (BestS)
End Algorithm;
```

The function $UpdateLength$ could perform an additive increase ($L = L + \beta$ for some $\beta > 0$) or geometric increase ($L = \beta \times L$ for some $\beta > 1$).

Experiment with *sequence-heuristic* and compare it with *random-walk*, *local-search*, and *random-sampling* heuristics. Use the problem instances and perturbation functions suggested in Exercise 1.8.

Exercise 1.12
Construct an example of a graph with 10 nodes, such that the nodes have a large degree, say 5–10.

1. Assume that all the nodes have unit sizes. Apply the *local-search* algorithm to obtain a two-way balanced partition of the graph.
2. Randomly assign weights to nodes say between 1 and 10 and generate an almost balanced partition with a minimum weighted cut-set using *local-search*. Since nodes have different sizes, a pairwise swap may not be the best move to generate the neighbor function. One possibility is to select a random partition (A or B), and to move the node to the other partition. Use the following cost function:

   $$Cost(A, B) = W_c \times Cut - set\ Weight(A, B) + W_s \times Imbalance(A, B)$$

   where,

   $$\begin{aligned} Imbalance(A, B) &= Size\ of\ A - Size\ of\ B \\ &= \sum_{v \in A} s(v) - \sum_{v \in B} s(v) \end{aligned}$$

$s(v)$ is the size of vertex (or node) v. W_s and W_c are constants in the range of [0,1] which indicate the relative importance of balance and minimization of cut-set, respectively.

3. Experiment with different values of W_c and W_s. Does increasing the value of W_c (W_s) necessarily reduce the value of cut-set (imbalance)?

Exercise 1.13

1. Repeat Exercise 1.12 using the *random-walk* search heuristic.
2. Repeat Exercise 1.12 using the *random-sampling* search heuristic.
3. Compare the three heuristics with respect to runtime, quality of solution, and quality of solution subspace explored.

Exercise 1.14

1. Construct a connected graph with 10 nodes and 25 edges. Starting from a random partition, apply both the greedy pairwise exchange and the *local search* algorithm to this graph and generate balanced two-way partitions.
2. Starting from the solution obtained from the greedy pairwise technique, apply the *local search* algorithm. Comment on any noticeable improvement in quality of solution and runtime.

Exercise 1.15

Given n modules to be placed in a row, show that there are $\frac{n!}{2}$ unique linear placements of n modules. When n is large, show that the number of placements is exponential in n.

Exercise 1.16

Write a procedure *CALC-LEN* to evaluate the total connection length of a given assignment. The inputs to the procedure are,

1. The number of modules n,
2. the connectivity matrix C; C is an $n \times n$ matrix of non-negative integers, where c_{ij} indicates the number of connections between modules i and j,
3. The assignment surface is a uniform grid of dimensions $M \times N$. The array P[$1 \ldots M, 1 \ldots N$] is used to represent the placement information. P[i, j] contains the number of the module placed in row i and column j.

You may assume that $M \cdot N = n$. What is the complexity of your procedure?

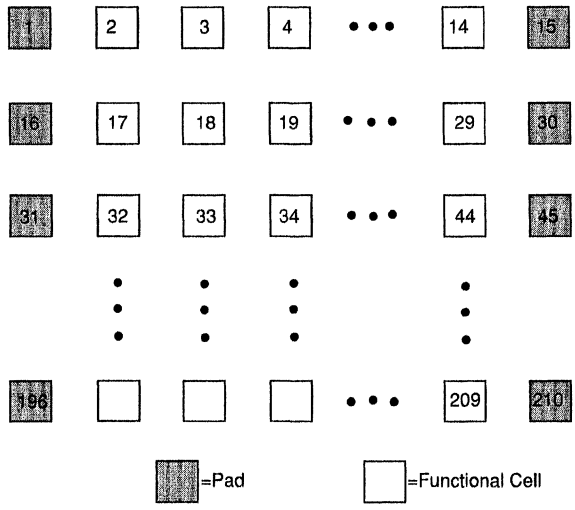

Figure 1.7: 210-cell mesh for Exercise 1.18.

Exercise 1.17
Suppose that n modules are placed in a row. The placement information is represented by array $p[1, \ldots, n]$, where $p[i]$ indicates the module placed in the ith location. If the modules are numbered $1, \ldots, n$, then p is simply a permutation of $1, \ldots, n$.

Write a procedure *DELTA-LEN* to compute the change in total wire-length when two modules in p are swapped. Assume that the connectivity information is represented by a connectivity matrix C as in Exercise 1.16.

Exercise 1.18
Implement a placement algorithm based on *local-search*. Assume that there are 210 modules to be placed on a 15×14 mesh. There are two types of modules, functional blocks and input/output (I/O) pads. The I/O pads must be placed only on the periphery of the mesh, whereas a functional block may be placed in any empty slot. Assume 28 I/O pads and 182 functional blocks.

Generate a random initial placement which satisfies the pad position constraint. Experiment with various perturbation functions. The *perturb* function must respect the pad position constraint. Use the *DELTA-LEN* procedure of Exercise 1.17 to evaluate the change in cost function Δc.

1. Test your program for the sample circuit shown in Figure 1.7. In other words, synthesize the connectivity matrix for the circuit and give it as input to your program.

2. Run your program for several *random* initial placements. Does the initial solution influence the final solution?

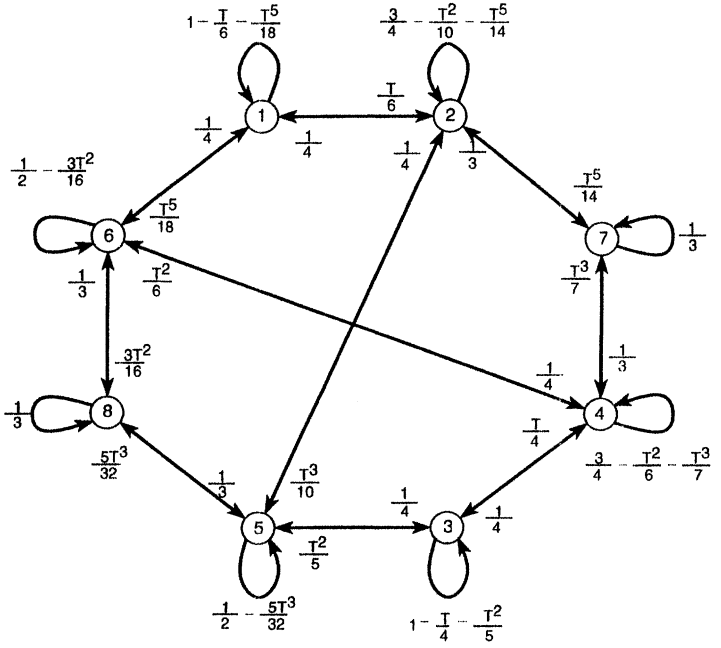

Figure 1.8: Configuration graph for Exercise 1.19.

Exercise 1.19

Consider the configuration graph of Figure 1.8 labeled with probabilities that are composed of uniform probabilities over each neighborhood and the acceptance function given by [Len90].

$$f(Cost_i, Cost_j, T) = \begin{cases} \frac{Cost_i}{Cost_j} \cdot T^{Cost_j - Cost_i} & \text{if } \Delta Cost_{ij} > 0 \\ 1 & \text{if } \Delta Cost_{ij} \leq 0 \end{cases} \tag{1.16}$$

The transition matrix $\Theta(T)$ is given on the following page.

$$\Theta(T) = \left\{ \begin{array}{cccccccc} 1 - \frac{T}{6} - \frac{T^5}{18} & \frac{T}{6} & 0 & 0 & 0 & \frac{T^5}{18} & 0 & 0 \\ \frac{1}{4} & \frac{3}{4} - \frac{T^3}{10} - \frac{T^5}{14} & 0 & 0 & \frac{T^3}{10} & 0 & \frac{T^5}{14} & 0 \\ 0 & 0 & 1 - \frac{T}{4} - \frac{T^2}{5} & \frac{T}{4} & \frac{T^2}{5} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{4} & \frac{3}{4} - \frac{T^2}{6} - \frac{T^3}{7} & 0 & \frac{T^2}{6} & \frac{T^3}{7} & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{2} - \frac{5T^3}{32} & 0 & 0 & \frac{5T^3}{32} \\ \frac{1}{4} & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{2} - \frac{3T^2}{16} & 0 & \frac{3T^2}{16} \\ 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \end{array} \right\}$$

1. Show that the stationary distribution at T is,

$$\pi(T) = \left(\frac{2520}{N}, \frac{1680\,T}{N}, \frac{840\,T^2}{N}, \frac{840\,T^3}{N}, \frac{672\,T^4}{N}, \frac{560\,T^5}{N}, \frac{360\,T^6}{N}, \frac{315\,T^7}{N}\right)$$

where

$$N = 2520+1680T+840T^2+840T^3+672T^4+560T^5+360T^6+315T^7$$

2. Verify that the generating chain is time reversible. For a reversible Markov chain, the transition probabilities of the forward and reversed chains are identical ($\theta_{ij} = \theta_{ji}$). That is, a time reversible Markov chain is identical to itself when viewed in reverse time.
3. Show that the stationary distribution converges to an optimizing distribution given by

$$\lim_{T\to 0} \pi(T) = (1, 0, 0, 0, 0, 0, 0, 0)$$

Exercise 1.20
Consider the configuration graph of Figure 1.9 labeled with probabilities, and the acceptance function of Exercise 1.19. The corresponding transition matrix is given below. Answer the following questions.

$$\Theta(T) = \begin{Bmatrix} 1-\frac{T}{6}-\frac{T^4}{15} & \frac{T}{6} & 0 & 0 & \frac{T^4}{15} & 0 \\ \frac{1}{3} & \frac{2}{3}-\frac{T^4}{9} & 0 & 0 & 0 & \frac{T^4}{9} \\ 0 & 0 & 1-\frac{T}{4}-\frac{T^2}{5} & \frac{T}{4} & \frac{T^2}{5} & 0 \\ 0 & 0 & \frac{1}{3} & \frac{2}{3}-\frac{2\,T^2}{9} & 0 & \frac{2\,T^2}{9} \\ \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{2}-\frac{5\,T}{24} & \frac{5\,T}{24} \\ 0 & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{Bmatrix}$$

(a) Verify the transition matrix representing the configuration graph in Figure 1.9.

(b) What is the stationary distribution ($\pi(T)$) and the optimizing distribution ($\pi = \lim_{T\to 0}\pi(T)$)?

(c) To the configuration graph of Figure 1.9 add an additional edge between states 2 and 3, rewrite the transition matrix, and find the stationary distribution. Should the distribution be different from the one obtained in part (b) of this question. Justify your answer.

(d) In the configuration graph obtained in part (c) above, delete edges between nodes 5 and 6 and between nodes 2 and 3, and add edges between nodes 2 and 5 and between nodes 3 and 6 and compute the stationary distribution. Is the stationary distribution obtained the same as in part (b) of this question? Why?

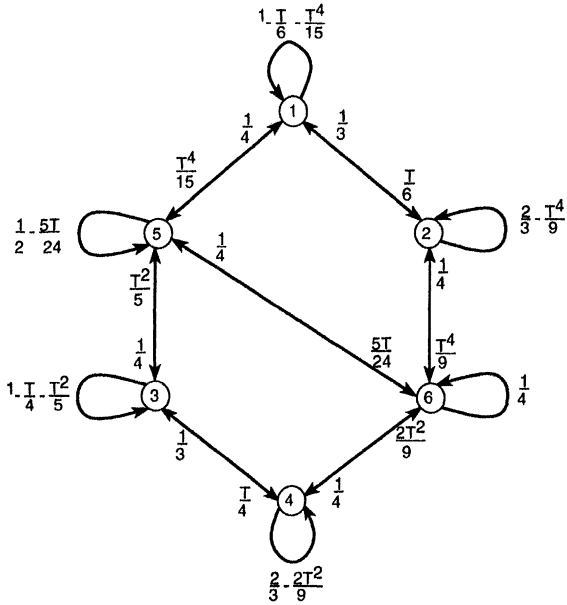

Figure 1.9: Configuration graph for Exercise 1.20.

Exercise 1.21
Consider the configuration graph given in Figure 1.10. The cost of the five states is as follows: $Cost_1 = 1$, $Cost_2 = 2$, $Cost_3 = 3$, $Cost_4 = 4$, $Cost_5 = 1$. $Cost_n$ represents the cost of state S_n. If the acceptance criterion used is as given in Equation 1.17 below,

$$f(Cost_i, Cost_j, T) = \begin{cases} e^{-\frac{\Delta Cost_{ij}}{T}} & \text{if } \Delta Cost_{ij} > 0 \\ 1 & \text{if } \Delta Cost_{ij} \leq 0 \end{cases} \tag{1.17}$$

determine the transition matrix and find the stationary distribution.

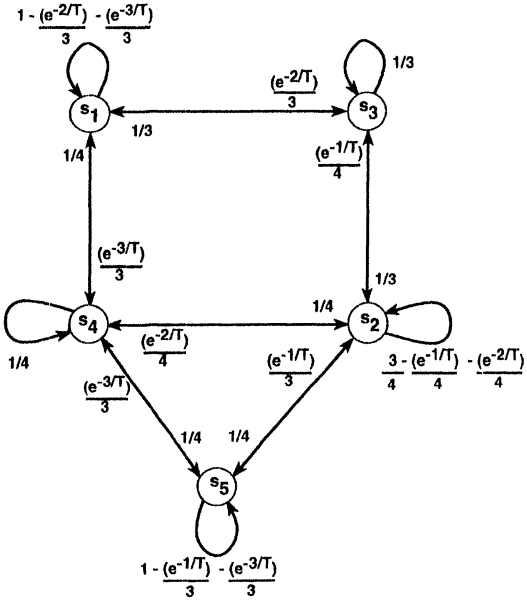

Figure 1.10: Configuration graph for Exercise 1.21.

Exercise 1.22
For the Markov chain given in Exercise 1.21, find the optimizing distribution.