CHAPTER ONE

COMBINATORIAL OPTIMIZATION PROBLEM

1.4 Solving Methods for COP

For many COP it is hard to find an optimal solution. A lot of effort has therefore been put in the design and analysis of approximation algorithms for these types of problems. Such algorithms do not necessarily find an optimal solution, but attempt to find a good solution.

Provided a problem has a finite number of solutions, it is possible, in theory, to find the optimal solution by trying every possible solution. An algorithm which tries every solution to a problem in order to find the best is known as a **brute force algorithm**. Cryptographic algorithms are almost always designed to make a brute force attack of their solution space (or key space) infeasible. For example, the key space is large enough so that it is not plausible for an attacker to try every possible key. CO techniques attempt to solve problems using techniques other than brute force since many problems contain variables which may be unbounded, leading to an infinite number of possible solutions. In the case where the number of solutions is finite it is generally infeasible to use a brute force approach to solve it so other techniques must be found.

Algorithms for solving problems from the field of CO fall into two broad groups - exact algorithms and approximate algorithms. An **exact algorithm** guarantees that the optimal solution to the problem will be found. The most basic exact algorithm is a brute force one. Other examples are branch and bound, and the simplex method. Approximate algorithms attempt to find a “good” solution to the problem. A “good” solution can be defined as one which satisfies a predefined list of

expectations. Often it is impractical to use exact algorithms because of their prohibitive complexity (time or memory requirements). In such cases **approximate algorithms** are employed in an attempt to find an adequate solution to the problem. Examples of approximate algorithms (or, more generally, heuristics) are simulated annealing, the genetic algorithm and the tabu search.

1.5 Algorithm and Complexity

Algorithm: An algorithm is a method for solving a class of problems on a computer.

Complexity: The complexity of an algorithm is the cost, measured in running time, or storage, or whatever units are relevant, of using the algorithm to solve one of those problems.

The time complexity of a calculation is measured by expressing the running time of the calculation as a function of some measure of the amount of data that is needed to describe the problem to the computer.

The general rule is that if the running time is at most a polynomial function of the amount of input data, then the calculation is an easy one, otherwise it's hard. For instance, matrix inversion is easy. The familiar Gaussian elimination method can invert an n×n matrix in time at most cn^3. For instance, think about this statement: I just bought a matrix inversion program, and it can invert an n×n matrix in just $1.2n^3$ minutes. We see here a typical description of the complexity of a certain algorithm. The running time of the program is being given as a function of the size of the input matrix. A faster program for the same job might run in $0.8n^3$ minutes for an n×n matrix.

A significant research topic in scheduling is the use of complexity theory to classify scheduling problems into two classes:

- P (polynomial), which the class P contains all decision problems that are polynomially solvable.
- NP (non-deterministic polynomial), contains many difficult problems for which no polynomial algorithm has been found.

However there is always a possibility that someone will prove that P=NP but the chance would seem to be a very small one.

The NP-hardness of a problem suggests that there are instances for which the computation time required for finding an optimal solution increases exponentially with problem size. If large computation times for such problems are unacceptable, then a heuristic method or an approximation algorithm is used to give approximate solutions.

1.6 Exact Solution Methods

There are many methods of exact solution, We will focus in complete enumeration, and branch and bound methods.

1.6.1 Complete Enumeration Method (CEM)

Complete enumeration methods generate one by one, all feasible solutions and then pick the best one. For example, for a single machine problem of n jobs there are n! different sequences. Hence for the corresponding m machines problem, there are $(n!)^m$ different sequences. This method may take considerable time as the number $(n!)^m$ is very large even for relatively small values of n and m.

1.6.2 Branch and Bound Methods (BAB)

Branch and bound (BAB) methods are implicit enumeration techniques which can find an optimal solution by systematically examining subsets of feasible solutions. These methods are usually described by means of search tree with nodes that corresponding to these

subsets. The BAB method uses a search tree. Each node in the tree contains a partial sequence of jobs. For n job problem, there are n-1 numbers of levels for a tree. At level zero, root node will be placed with all n empty sequence positions. At level 1, there will be n number of nodes. Each node will contains a partial sequence of jobs. The first position in the sequence will be occupied by a job in numerical order. Similarly, each node at $(n-1)^{th}$ level will be branched to (n-2) number of nodes. The process will continue till each node has exactly one leaf.

Generation of all sequences is combinatorial in nature and, will result in enormous number of sequences even for a small number of jobs. For example, for a 10-job problem there will be 10! sequences. To reduce the computational effort, lower bounds are calculated at every level for each node. The formula used to compute the lower bound is pertained to objective function of the COP. Branching is carried out only from those nodes with a minimum lower bound. By doing so, only small proportion of the nodes is explored resulting in fewer amounts of computations. The BAB method is applied in almost every COP problem.