CHAPTER VI
ANALYSIS OF ALGORITHMS

6.1 Introduction

Algorithms lead to special kinds of solutions to problems not answers but rather precisely defined procedures for getting answers. The algorithm design techniques can be interpreted as problem-solving strategies that can be useful regardless of whether a computer is involved. An algorithm is defined as a sequence of well-defined, unambiguous instructions for solving a problem. i.e. for obtaining a required output for any legitimate input in a finite amount of time. The following points are to be considered while defining the notion of algorithm.

- The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- Several algorithms for solving the same problem may exist.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.
6.2 Design and Analysis of Algorithm

The sequence of steps typically goes through in designing and analysis of algorithms are given below.

Understand the problem

Decide on:
- computational means,
- exact vs approximate solving,
- data structures,
- algorithm design techniques

Design an algorithm

Prove correctness

Analyze the algorithm

Code the algorithm

6.2.1 Understanding the Problem

From a practical perspective, the first thing we need to do before designing an algorithm is to understand completely the problem given. Read the description of the problem carefully and raise questions if there is any doubt in the problem. Do a few small examples by hand, think about special cases, and raise questions again if needed.
6.2.2 Ascertaining the Capabilities of a computational device

Once the problem is completely understood, we need to ascertain the capabilities of the computational device the algorithm is intended for. Most majority of algorithms in use today are still destined to be programmed for a computer. In this programming concept, the central assumption is that instructions are executed one after another, one operation at a time. Such designed algorithms are called sequential algorithms. Some of the special kind of computers can execute operations concurrently i.e. in parallel. Such type of classic algorithms known as parallel algorithms.

6.2.3 Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an exact algorithm; in the latter case, an algorithm is called an approximation algorithm. Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly. Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem are the intrinsic. Third, an approximation algorithm can be a part of a more sophisticated algorithm that solves a problem
exactly.

6.2.4 Deciding on Appropriate Data Structures

Some algorithms do not demand any ingenuity in representing their inputs. But others are, in fact, predicated on ingenious data structures. In addition, some algorithm design techniques depend intimately on structuring or restructuring data specifying a problem’s instance. In the new world of object oriented programming, data structures remain crucially important for both design and analysis of algorithms.

6.2.5 Algorithm Design Techniques

An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms. There are various design techniques by which an algorithm can be developed are:

- Incremental
- Divide and conquer
- Sub goal
- Hill climbing
• Working backward
• Branch and Bound
• Backtracking
• Heuristic

Algorithms derived to solve the same problem often differ dramatically in their efficiency. The choice of a design technique, which is often highly dependent on the choice of model, can greatly influence the effectiveness of a solution algorithm. Two different algorithms may be correct, but may differ tremendously in their effectiveness.

6.2.6 Methods of specifying an Algorithm

Once an algorithm is designed, we need to specify it in some fashion. There are two options that are most widely used nowadays for specifying algorithms namely pseudocode and flowchart.

A pseudocode is a mixture of a natural language and programming language like constructs. A pseudocode is usually more precise than a natural language, and its usage often yields more succinct algorithm descriptions.

Flowchart is another option for specifying algorithm. It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the steps
of the algorithm.

### 6.2.7 Proving an Algorithm’s Correctness

Once an algorithm is specified, we have to check it for its correctness. That is, we have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For some algorithms, a proof of correctness is quite easy, for others it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm’s iteration provide a natural sequence of steps needed for such proofs. The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. For an approximation algorithm, it would like to be able to show that the error produced by the algorithm does not exceed a predefined limit.

### 6.2.8 Analyzing an Algorithm

After correctness, the most important aspect to be considered is efficiency. There are two kinds of algorithm efficiency: **Time efficiency** and **Space efficiency**. Time efficiency indicates how fast the algorithm leads to a solution. Space efficiency indicates how much memory the algorithm needs. Time efficiency is measured by counting the number of times the algorithm’s basic operation is executed. Space efficiency is measured by counting the amount of memory space consumed.
by an algorithm. Simply, analyzing an algorithm means predicting the resources that the algorithm requires, resources like memory, communication bandwidth, or logic gates are of primary concern, that most often it is the measurement of computational unit.

6.2.9 Orders of Growth

The order of growth of running time of an algorithm gives a simple characterization of the algorithm efficiency. It allows us to compare the relative performance of alternative algorithm to determine the exact running time of an algorithm. The running time of an algorithm increases with the size of the input. Usually, an algorithm that is asymptotically more efficient will be the best choice of all but very small inputs.

6.2.10 Worst-case, Best-case and Average case efficiencies

Generally, the efficiency of an algorithm is measured as a function of a parameter indicating the size of the input. But there are many algorithms for which running time depends not only on the input size but also on the specifics of a particular input. The efficiency of algorithms may differ significantly for input of the same size. For such algorithms, we need to distinguish between the worst-case, best-case and average-case efficiencies.

The **worst-case efficiency** of an algorithm is its efficiency
for the worst-case input of size n, for which the algorithm takes the longest among all possible inputs of size n. The way to determine the worst-case efficiency of an algorithm is, in principle, quite straightforward: analyze the algorithm to see what kind of inputs yield the largest value of the basic operation’s count among all possible inputs of size n and then compute the worst-case value. The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed its running time on the worst-case inputs.

The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size. The best case efficiency can be analyzed as follows: First, we determine the kind of inputs for which the count will be the smallest among all possible inputs of size n. Then, we should ascertain the value of basic operation’s count on these most convenient inputs. The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency. But it is not completely useless, either. Though we should not expect to get best-case inputs, we might be able to take advantage of the fact that for some algorithms, a good best-case performance extends to some useful types of inputs.
close to being the best-case ones. If the best case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

Neither the worst-case analysis nor its best-case analysis yields the necessary information about an algorithm’s behaviour on a “typical” or “random” input, then the **average-case efficiency** seeks to provide. To analyze the algorithm’s average-case efficiency, we must make some assumptions about possible inputs of size $n$. The direct approach for doing analysis involves dividing all instances of size $n$ into several classes so that for each instance of the class the number of times the algorithm’s basic operation is executed is the same. Then a probability distribution of inputs needs to be obtained so that the expected value of the basic operation’s count can then be derived.

### 6.3 Computational aspects

As the algorithms have been devised by individuals, they may have different computational efficiencies. Generally, in solving Linear Integer Programming Problems, researchers compare more than one algorithm with the developed one based on the total number iterations to reach the optimal solution of the problem. But in this thesis, the comparison is based on the number of iterations and time for typical problems that are given in the Table 6.1 and Table 6.2. The execution time
may vary depend on the multiply/divide operations in solving problems. From the execution of the algorithm the **Direct method** of solving Linear Integer Programming Problem has always superior to the Cutting plane, Branch and Bound on the basis of number of multiply/divide operations as given in Table 6.3, 6.4 and 6.5. It was observed that the direct method the solution is improved in each iteration gradually.

### 6.3.1 Computational effort required in Revised Simplex method

In the Revised Simplex Method [35], the product form of inverse method is used to update the basis inverse matrix. If ‘m’ is the number of constraints in the problem, then m+1 columns with m+1 elements in column must be transformed and one multiplication is required for each transformation. This results about \((m^2 + 2m)\) operations. Computation of \((n-m)\) values \(z_j - c_j\) requires \((n-m)m\) operations since ‘m’ multiplications are needed to find each \(z_j - c_j\). Also, \(m^2\) multiplications are required to compute the last \(m\) components of \(B^{-1}P_j\) and \(m\) division to determine \(\frac{(B^{-1}P_j)_i}{(B^{-1}P_j)_k}\). The total number of operations per iteration for the revised simplex method is approximately, \(nm + m^2 + 3m\). For \(n= 10\) and \(m = 5\), the total number of operations for the Revised Simplex Method undergoing \(m\) iterations accounts to 450 multiplication/division operations.
6.3.2 Computational effort required for Cutting Plane Algorithm

Stage 1 Solving the initial problem by using Revised Simplex Method. Number of Multiply/Divide operations required
\[ m^* (nm + m^2 + 3m) \].

Stage 2 If the solution is not feasible add a constraint to the problem. So number of constraint increased by 1. Solve by using Revised Simplex Method and repeat stage 2 until feasible solution occurs. The number of multiply/divide operations
\[
(m+1)(n(m+1)+(m + 1)^2+3*(m+1))+...+
\]
\[
(m+k)(n(m+k)+(m + k)^2+3*(m+k))
\]
\[
= \sum_{z=1}^{k} (m + z) \times (n(m + z) + (m + z)^2 + 3 \times (m + z))
\]
Where \( k \) is the number of constraints added to the original problem.

The total number of operations for the Cutting plane method is approximately,
\[
nm+m^2+3m+(m+1)(n(m+1)+(m + 1)^2+3*(m+1))+...
\]
\[
+(m+k)(n(m+k)+(m + k)^2+3*(m+k)).
\]
\[
= \sum_{z=0}^{k} (m + z) \times (n(m + z) + (m + z)^2 + 3 \times (m + z)) \text{ For } n=10, m=5 \text{ and } k=3 \text{ the total number of operations for the Cutting plane accounts to 3458 multiplication/division operations.}
6.3.3 Computational effort required for Branch and Bound Algorithm

Stage 1 Solving the initial problem by using Revised Simplex Method. Number of Multiply/Divide operations required \( m^*(nm+m^2+3m) \).

Stage 2 The solution is not feasible the original problem is subdivided into two problems with lower and upper bounds. Solve these two problems by using Revised Simplex Method and repeat stage 2 until feasible solution occurs.

\[ S^*(m^*(nm+m^2+3m)) \]

Where \( S \) is the number of sub problems.

The total number of operations for the Branch and Bound algorithm is approximately,

\[ m^*(nm+m^2+3m)+S^*(m^*(nm+m^2+3m)) \]

\[ =(S+1)^*(m^*(nm+m^2+3m)) \]

For \( n=10, m=5 \) and \( S=8 \) the total number of operations for the Branch and Bound accounts to 4050 multiplication/division operations.

6.3.4 Computational effort required for Proposed Method

6.3.4.1 Phase I Arrangement of promising variables

a. \( n^*m \) division operations are required to find the \( \frac{b_i}{a_{ij}} \) (intercepts).

b. \( n \) multiplication operations are required to find \( c_jx_j \).
6.3.4.2 Phase II  Process on Arranged variables

c. m division operations required to find the minimum intercept value of the entering variable.

d. One multiplication required to find the initial value of the decision variable.

e. m multiplication required to find the $P_{0new}$ value.

f. m multiplication and (m-1)division operation required to find the next entering variable.

g. One multiplication required to find the initial value of the next entering variable.

h. To obtain the optimal solution the required number of multiplication /division is $m^*(3m)$ (From (e),(f)&(g))

So total number of mul/div operation required per iterations (upto Phase II) is $3m^2+m+m*n+n+1$

6.3.4.3 Phase III

a. To find objective function value needs n multiplication as there are n variables.

b. Updations of $P_0$ value requires $m^*(m-1)$ multiplication operations.

c. $3m^2+m+m*n+n+1$ multiplication / division operation is required to find $X_{Bnew}$

d. To find objective function value needs n multiplications.
Phase III requires \(4m^2n + n^2m + 2n^2 + 2n\) mul/div operations.

To Complete all the Phases, it requires
\[\{(4m^2n + n^2m + 3m^2 + 2n^2 + mn + m + n + 1)\times I\}\] mul/div operations.

I is the number of improvement, it depends on the problem.

For \(n = 18, m = 4\) and \(I = 10\) the total number of operations for the proposed new algorithm accounts to 32390 multiplication/division operations.

6.4 Comparison of algorithms based on multiply/divide operations

In solving Linear Integer Programming Problem, the new methods proposed are considered as the superior to the existing Cutting Plane and Branch and Bound. Algorithms based on the number of multiply/divide operations as given in Table 6.2. Formulae were derived to compute the multiply/divide operations. As the packages have been developed for implementing the algorithm and number of multiply/divide operations are computed. The multiply/divide operations computed through formulae and packages are compared. It is observed that the number of multiply/divide operations from the package is less than the computed one from the formulae, as the multiplication is carried out in the package only if both the
quantities are non-zero. The zero terms in the multiplications are excluded in the package.

6.5 Conclusion

In this chapter, various aspects of designing and analysis of algorithm have been discussed. Also the computational aspects of the Direct methods have been presented. Formulae were derived to compute the number of multiply/divide operations required for the algorithm. A package has been developed to implement the algorithm. The number of multiply/divide operations are computed by the formulae and by the package are compared.