# RESEARCH PROJECT SUBMISSION

## OPTIMIZATION APPROACHES

**Program: Computer Engineering and Software Systems**

*Course Code: CSE224*

*Course Name: Design and Analysis of Algorithms*

*Examination Committee*


**Dr. Gamal A. Ebrahim**

**Dr. Mohamed A. Taher**

**Dr. Mahmoud I. Khalil**

**Ain Shams University**

**Faculty of Engineering**


**Spring 2020 Semester**

## Student Personal Information

**Student ID:** 17P8113
**Student Full Name:** Ahmed Abd El-Nasser Korany

## Plagiarism Statement

I certify that this assignment / report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they are books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment / report has not been previously been submitted for assessment for another course. I certify that I have not copied in part or whole or otherwise plagiarized the work of other students and / or persons.

**Signature/Student Name:** **Ahmed Abd El-Nasser** **Date:** **2020, June 9**

## Submission Contents

TABLE OF CONTENT

# Optimization Approaches

Ahmed Abd El-Nasser Korany (17P8113)

Computer Engineering and Software Systems (CESS)
Faculty of Engineering, Ain Shams University

17P8113@eng.asu.edu.eg

***Abstract*** **This paper discusses four different optimization approaches with an interest in each one's characteristics, alongside with an explanation of an implemented code that applied to an optimization problem.**

***Keywords*** **optimization approaches, dynamic programming, genetic algorithms, ant colony, branch and bound, optimization problems, knapsack problem**

## I. INTRODUCTION

Optimization problems are of great interest in computer science. They grab the focus of attention of algorithm designers and creators. The term optimization refers to a refinement process where we seek to maximize or minimize some variables over a function (called the objective function) with or without respect to some constraints. Accordingly, optimization problems are of several types. Therefore, the solving approaches also comes in several types and techniques to cover this variety of problems. Each approach has his own characteristics, operations, and methods to attack the problem and solve it. Some of them successes to find an exact optimal solution (called exact algorithms), while others are satisfied with a near-optimal one (called approximation algorithms). Although each approach has advantages to use, yet still, nothing is perfect, every approach suffers from some drawbacks and limitations. Therefore, it is convenient to describe it as a trade of deal, you sacrifice space to save time, or do the opposite, win them both, or lose them both. Not just space and time, but other metrics also can play a significant role in the analyzing processes. In this report, we are going to survey four of the optimization approaches exposing each one's characteristics, operations, functionalities, and efficiencies. We also are going to implement some codes to see how these approaches get applicated in a practical manner. Eventually, a comparison of time complexity analysis between approaches alongside with the ordinary techniques; brute force, divide and conquer, and greedy. (Skiena, 2008)

## II. DYNAMIC PROGRAMMING

"A tabular recursive-with "memoization" technique"

The big challenging algorithms problems usually involve optimization, where our target is to obtain a solution that maximizes or minimizes variables or functions with respect to some constraints. Take the Traveling salesman as an example. A well-known classic optimization problem, where the traveler wants the cycle to visit all vertices of a graph to be at its minimum. Although it sounds easy to propose multiple different algorithms to solve the TSP and produce acceptable solutions, that did not always give us the minimum tour cycle we want.

Most of the optimization problems algorithms needs to be proved to always return the best, or better say the optimum possible solution. The Greedy algorithms making the optimal local decision in each step are supposed to be efficient, but the problem is that, in most cases, they do not guarantee us global optimality.

It is known that the exhaustive search algorithms trying all possible ways, then select the best, always generate the optimal solution, but sometimes this comes with a cost in the manner of time complexity, a prohibitive cost!

Dynamic programming (DP) introduces us with a way of designing and customizing algorithms that search in all the possibilities systematically (thus guaranteeing correctness). At the same time, it stores the results already calculated to avoid redundant computations (thus, it provides efficiency). We shall be exposed with the exact methodology of who this technique works in a later section. But the main idea I would like to present here is that by storing all possible decisions consequences and using this information, the amount of work needed to solve the problem is totally minimized. In the meanwhile, the Dynamic programming optimization techniques are probably considered to be easy algorithm design techniques to apply for practice.

Dynamic programming can be considered as both a computer programming and a mathematical optimization method. The method was first developed in the 1950s by Richard Bellman and has so many applications in so multiple fields. But we will come to this brief history of this optimization technique when speaking about the foundation.

Also, we will discuss in detail how this technique works. But, briefly here in this introduction, I can say that in both of the mentioned contexts, Dynamic programming in general speaking refers to the simplification of a complicated problem by breaking it down into more simpler sub-problems of it in a recursive way. At the same time some decision problems cannot be considered apart this way, decisions that span over several points in time, often do break apart recursively. The same way, in computer science, if we can optimally solve a problem by breaking it down into sub-problems and after that we aim to find the solutions that are optimal to these sub-problems recursively, then we can say that we have an optimal substructure. (Billman, 1954)

Eventually, If a punch of sub-problems can get glued or nested recursively into some larger problems, in a way that dynamic programming methods gets applicable, then there exist a relation between the larger problem value and the values of the original sub-problems. In the literature of optimization, they call this relationship the Bellman equation, The man behind this technique. That was a little introduction to what Dynamic programming is.

### *A. Theoretical Foundation*

Dynamic programming is a technique for algorithm designing which has an interesting history. It was invented by the prominent mathematician of U.S, Richard Bellman, in the 1950s as to be a general method to optimize multistage decision problems. Thus, the word "programming" appearing in the name of the technique stands for the concept of "planning" and does not refer to programming languages. After giving the proof that its worth as an important technique for applied mathematics, dynamic programming has eventually appeared to be considered, at least in computer science manners, as a general technique for algorithm designing. A technique that does not necessarily has limits to special types of optimization problems. And this is how this technique considered from a point of view.

The definition “dynamic programming” was first used upon the 1940s by the U.S. mathematician Richard Bellman in a description to the process of problems solving where we want to find the best-to take decisions by one after another. Among this time to 1953, he had added some refines to this description to have a modern meaning, specially referring to the process of collecting small decision problems to form larger decision problems, thereafter, IEEE admitted this field to be a system analysis and an engineering topic. His innovation is well known under the name of the Bellman’s equation, which represents a central result of dynamic programming techniques that recursively reformulate an optimization problem.

let us have a brief presentation to the survey of the fundamental concepts the dynamic programming was inspired by. As a beginning, the theory was originally created to deal with the mathematical problems appearing from the studies on a variety of decision processes requiring multiple stages of decision, which may be roughly described in like this: assume a physical system whose current state is defined by a set of parameters which we variables of the state. At specific times, which may be pre-scribed in advance, or which may be left for the process to set by itself, we need to make decisions which shall affect the system’s state. These decisions we are going to make have a close equivalence to the transformations taken place by the state variables, the decision we choose is to be identical with the

transformation choice. After that, we will use the outcome of the past decisions to lead the choice of ones to come in the future, of course following the purpose of this whole process -just like an optimization technique- that is maximizing some function of the variables that describe the final state.

As an Example for processes that fit this brief description are found by virtually almost in every phase of our modern life, the production lines of industry planning, choice of optimum inventory policy for the department stores and establishments of military, medical clinic patients scheduling, programming of training policies, and long term university programs determination. All of these -and more- life examples involve the theoretical base of the dynamic programming technique, that is a multi-stage decision process by dividing a large problem into sub-problems.

Now, let's move on to more precise discussion to introduce some terminology about this technique, The classical approach to solve mathematical problems involved in the situations introduced above is about considering all possible decisions in sequence, to put it another way, the set of all feasible strategies, compute the return of each one, and then to maximize that return over the whole set.

Although it is very obvious that a reasonable as such a process is, sometimes it is not practical. Consider processes that involve a large number of stages and a wide range of choices at each of these stages, the size of the resultant optimization problem will be annoyingly high, that's because of the continuous processes which require optimization over function space.

In a practical re-examination to the situation, we will see that this price of excessive size or dimension, a price that can easily make a modern computer struggle, comes from a requirement for large number of information. But the question is how much required information does a process of multi-stage decision actually need to carry out? Does it require a knowledge of the complete decision sequence, the ones to be executed now at the current stage, those at the upcoming one, and so on? The Answer is No, Not at all! It is sufficiently enough to formulate a general description that determines the decision we need to take at each stage according to the current state of the system. To put it another way, if at any specific time we know what decision to make or what to do, then, we don't necessarily need to know the required decisions for the sub-sequent times. (Billman, 1954)

We shall discuss in more mathematical details an example of the formulation process applied to discrete deterministic optimization using Dynamic programming when we speak about how it works in a later section.

But as for now, this was a close look at the foundation of this technique and how its founder Bellman managed to formulate it. With also a brief history about it.

### *B. How It works*

We have gone through an introduction to this technique and the theoretical foundation of it. And now it is time for some technical stuff. Here, in this section we will have a look to how this technique works and how it is applicable to its problem's domain.

Here is the whole idea briefly before going in detail. We bring a large problem and break it into sub-problems. And this step is called "Recursive formulation". The solutions of these sub-problems can be combined together to solve the global problem. So, we have two important major things to be concerned about. These are like the pivot to solve a problem using this technique. First, we need to find a way to break the problem into the smaller sub-problems using recursive formulation. Then this formulation shall represent the optimum solution to each problem in terms of the solutions of the sub-problems. That's the whole idea of dynamic programming briefly. But we will discuss further details and issues that arise during the process.

### *The fundamental approach*

The fundamental approach, as stated above, and the base idea behind the dynamic programming theory is that of forming an optimal sequence of decisions as one and determine the decision required in accordance with the present state of the system. (Billman, 1954)

Before we jump to discuss some examples of mathematical formulation, let us have a look on some basic elements that characterize an algorithm of dynamic programming.

### *Dynamic Programing Algorithms Characteristics:*

At this stage, it's very important that I define some characteristics before mentioning and using them in the mathematical formulation and also, I shall use them later in the NP problem optimization solution using dynamic programming approach. Also, these characteristics should provide somehow a walkthrough into the steps involved to apply optimization technique. (Skiena, 2008)

**Substructure (Overlapping Sub-problems):** Decompose the global problem into some set of smaller sub-problems. And express the solution of the large problem in terms of the solutions of the smaller ones, which is the main idea of dynamic programming.

**Table-structure:** Store the answers of each of the subproblems in a table. This is very important because we will need to reuse these solutions many times later to solve the global main problem.

**Bottom-up and Top-down computations:** We have stored the answers of the subproblems into a table. Now it's time to combine them and get the global solution. So, basically this is defined to be: "Combining solutions of smaller subproblems to solve larger ones". For that we have two approaches that are two basic ones for solving dynamic programming problems (Karumanchi, 2017), which are:

- **Bottom-up (AKA "Tabulation"):** We evaluate the function by starting with the smallest possible input value and then we move on through possible values, slowly increasing the input argument value. In the same time of computing values, we store all computed ones in a table.
- **Top-Down (AKA "Memoization"):** In this approach, the problem is divided into sub problems; each of which is solved; and the solutions are remembered, so we don't have to compute it again when needed.

**Optimal substructure (AKA the Principle of optimality):** It's very simple to define and understand but yet very essential for solving dynamic programming problems, It states that "for the global problem to have an optimal solution, each of its subproblem should also be solved optimally".

### *Mathematical Formulation of Discrete Deterministic Optimization Process*

To formulate the functional equations arising from the application of the principle of optimality, we shall begin with a simple case of a deterministic process. We can describe system at any instance of time by an M-dimensional vector $V(V_1, V_2, V_3,\ldots,V_N)$ lying within some region R. Let $T = \{T_K\}$ be a set of transformations respecting the property of $V \in R$ implies that $T_k(V) \in R$ for all k.

Consider we have an N-stage process of optimization (maximize some scalar function). S(P) is to be called the return state function. The sequence of decisions consists of a selection of N-transformations, $V - (T_1, T_2, T_3,\ldots,T_N)$. This will successively yield the states:

$$V_1 = T_1(V)$$
$$V_2 = T_2(V_1)$$
$$V_3 = T_3(V_2)$$
$$.$$
$$.$$
$$V_N = T_N(V_{N-1})$$

If R is a finite region, also if each $T_K(V)$ is continuous in V, and if S(V) is a continuous function of V for V ∈ R. It's then very clear that there exists an optimal policy (sequence of decisions). The max value of S(VN) will be a function of only the initial vector V and the no. of stages N.

Assume we define:

$$f_N(V) = max_V S(V_N)$$

We now employ the principle of optimality to derive a functional equation for $f_N(V)$. Assume to choose a transformation $T_K$, as the result of the first decision, obtaining thereby a new state $T_K(V)$.The maximum return from the following (N-1) stages is $f_{N-1}(T_K(V))$. Then K must now get chosen to maximize this. The result would be the basic functional equation we are looking for:

$$f_N(V) = max_K f_{N-1}\big(T_K(V)\big) \quad , where\ N = 2,3, \dots$$

### *C. When to Use*

Remember when we spoke about the major point to solve an optimization problem by an algorithm using dynamic programming technique? That is we need to find a way to break the problem into some set of sub-problems. Well, that's a big part of the answer to the title question. If the problem somehow can be formulated into some solvable sub-problem then re-combine these solutions to get the big picture and solve the global problem, then, this is when the dynamic programming is applicable.

The essential characteristic describing the multi-stage decision procedure is important in the process of transforming the complex large problem into a set of sequential simpler sub-problems.

Dynamic programming approach is an applicable technique for solving problems that observes the principle of optimality. Problems with overlapping sub-problems. In most cases, these subproblems appears from a recurrence that is related to a considered problem's solution to solutions of its divided smaller subproblems. The advantage is that rather than we resolve these overlapping sub-problems again and again, dynamic programming technique is to solve each of the each smaller subproblem only once and keep the results saved in a table from which we can obtain the solution of the original problem using memoization by either the two approaches discussed earlier (Top-down and Bottom-up). (Alfred V. Aho, 1974)

It's also an important thing to note that this recursive technique is most useful not only if the problem is dividable, but also with a reasonable effort with a sum of sub-problems that can be kept tolerably small.

Eventually, we get to a conclusion that two dynamic programming properties can tell whether it can solve a problem given or not: Overlapping sub-problems and Optimal substructure.

### *D. Problem Domain*

Here are some commonly known problems with algorithmic solutions that are based the dynamic programming technique for optimization:

- Fibonacci numbers
- Binomial Coefficient
- Longest Common Subsequence (AKA LCS)
- Longest Common Substring
- Lobb Number
- Bellman–Ford Algorithm
- The NP problem we are going to discuss later: 0-1 Knapsack Problem
- Weighted job scheduling
- Word Wrap Problem
- Matrix Chain Multiplication
- Longest Arithmetic & Longest Geometric Progression

### *E. Limitations*

The question is, can dynamic programming solve any problem? Certainly, the answer is no. Just like other approaches and techniques, Dynamic programming has its weak points and limitations; cases where it's not applicable or at least not the proper technique to use.

Dynamic programming is well known for being a trade-off between memory and time. Accordingly, the biggest limitation when using dynamic programming is the number of the sub-problems and partial solutions that we must keep track of.

Another concern is the concept of memoization of recursive calls. which keeps the dynamic programming special. So, once this strength point gets useless, dynamic programming does so as well. If the divided sub-problems are independent of each other and there is no repetition, then memoization cannot help us here.

### *F. Global Convergence*

It is clear to everyone that dynamic programming is a very powerful technique for solving multi-stage optimization problems. But does it guarantee an optimal solution? Well, to answer a question like that, we need to examine other approaches that can solve the same domain type of problems, and if we found that dynamic programming offers the best solution in terms of algorithms analysis aspects, then we can say it's close to optimality. Among those techniques is of course Divide and Conquer which is very similar to dynamic programming.

Basically, yes, dynamic programming provides an optimal solution to its problem. But it is not ALLWAYS guaranteed. When some problems go very large, DP might not be applicable as it will produce non correct solution

So, compared to other techniques, there is a consensus that dynamic programming guarantees to generate an optimal solution as it generally considers all possible cases and then choose the best. This opinion is supported by the fact that dynamic programming involves the principle of optimality. It constructs only optimal solutions of sub-problems, So, it is expected to build an optimal solution to the original global problem.

# III. Genetic Algorithms

Among the history of the human mankind, Nature has been an inspiration source for us in many situations. Genetic Algorithms are search-based algorithms that are based on two pure natural concepts: genetics and natural selection. Genetic algorithms are subset algorithms from a higher level of computations known as "Evolutionary Algorithms" which also by their turn are grouped under a much higher and larger branch: the meta-heuristic optimization.

So, we can define it as follows, Genetic Algorithms (GAs) are adaptive nature-inspired global-search meta-heuristic algorithms that belong to the larger family of the "Evolutionary algorithms". The basis of the ideas of natural selection and genetics are the intelligent usage of searching randomly supported by historical data to guide the search into the region of optimality in a space of solutions. Genetic algorithms are used to produce high quality solutions for both search and optimization problems. (Mathew)

Without going deep in the methodology of these algorithms because we shall discuss this later, but in quick words, Genetic Algorithms works by encoding a potential or possible solution to a given specific problem on a simple chromosome-like data structure and then apply recombine these structures in order to keep the critical information. More discussion over how it works is to come later.

## *A. Theoretical Foundation*

As I mentioned earlier, Genetic algorithms are nature-inspired from the since of genetics and the concept of "natural selection", which is the core principal of Darwinian evolution. So, theoretically, they have a biological-based terminology.

Genetic algorithms were first invented by John Holland during the 60s of the 19th century and then got developed by a contribution of him with his students and colleagues at the University of Michigan. Unlike usual evolutionary programming and evolution strategies, Holland's original target was not to create a design for algorithms that can solve specific problems, but rather to formally provide a study to the phenomenon of adaptation as it takes place in nature and to develop some ways of how we can import the mechanisms by which natural adaptation works to use in computer systems. (Simon, 2013)

### *Biological terminology*

It is not our major goal of discussion here but, we must have a spinoff to formally introduce some biological terminologies that are used as a base for the genetic algorithms.

Every living organism consists of cells. Each of which contains chromosomes in it. The chromosome can be divided into genes. A cell can have multiple chromosomes in each cell for some organisms. If we collected the whole collection of the genetic stuff (all chromosomes), then we get something called organism's genome. A specific set of genes contained by a genome is called genotype. If two individuals have identical genomes, then they are said to be having the same genotype. (Melanie, 1998)

### *Genetic algorithms and biology*

Now, let us get back to computer since to see how this terminology was imported and applied to introduce the genetic algorithms.

As for genetic algorithms, a chromosome is a candidate solution for solving a problem and is encoded as a bit string of zeros and ones (alleles). The term gene is represented by either single bits or short blocks of bits encoding a particular element of the solution candidate. An allele is a string of a bit, either 0 or 1. The Crossover process is to exchange genetic material between two single parent chromosomes. The Mutation process is to flip the bit at a randomly chosen position (locus).

Now that we have set the transformation of terms from biology to computer science. Let us see how these things are combined together. A genetic algorithm is a method that moves from one population of "chromosomes" (zeros and ones strings or simply “bits”) to a new population by using the natural selection concept alongside with the genetics inspired operators of crossover, mutation, and inversion. Each chromosome mainly consists of "genes" (represented by bits), and each gene is an instance of a specific "allele" (represented by zero or one). The operator of selection will then choose the chromosomes in the population that will have an allowance to reproduce, and on average the more fit a chromosome is, the more it produces offspring over the less fit ones. (Melanie, 1998)

### *B. How It Works*

Any genetic algorithm basically follows some steps that visualizes an abstraction of how this technique works, these steps also involve three important types of operators. We shall discuss this in details in this section.

Given a well-defined problem with a bit string that represent a candidate solution, an application of a genetic algorithm should work as follows:

1. Formulate the initial population
2. Randomly initialize the population

These two steps illustrate the process of Starting with creating a randomly generated population of n l−bit candidate solutions to the problem.

3. Evaluate objective function and calculate the fitness f(x) of each chromosome(solution) in the population.
4. Apply the genetic operators: Reproduction, Crossover, and Mutation.
5. Repeat steps (3) and (4) until some stopping criteria.

Each iteration of these steps we call a **generation**. And as for an entire set of generations we refer to as a **run**. Now that we have seen the basic working principle and the major steps involved in a genetic algorithm, we discuss these steps and what does these operators mean and what are their functionalities. (Shiffman, 2012)

#### *Part I: Creating a Population*

Since genetic algorithm usually processes a number of solutions simultaneously. In this first step, the algorithm generates a population with P individuals using some pseudo random generators. Those individuals represent a feasible solution. This representation of a vector solution in a solution space is called initial solution. This ensures that the search is robust and unbiased, since it begins from a wide range of solutions in the solution space.

#### *Part II: Selection*

Briefly in this part, we evaluate individual members of the population to find the value of the objective function. The method of exterior penalty function is utilized in order to transform a optimization problem that is constrained to an unconstrained one. This procedure is exclusively problem specific.

In this part, we apply the **Darwinian principle of selection.** We can divide the selection process into two parts:

- **Evaluate fitness:** For a proper functionality, we are in a need to a fitness function. Such function will produce a numeric score that expresses the fitness of each member in the population. In the real world of genetics - which we mimic here- This of course is not how it works at all. Organisms are not given a score; instead, they simply either survive or not, based on “fertility” which is here referred to as fitness. But in genetic algorithm, we seek an optimal solution to the problem. So, we need an ability to evaluate the solutions numerically.
- **Mating pool:** Once we have calculated the fitness for all of the population members, we can then select which of these members are fit enough to become parents and give them a place in a mating pool. Now, you may wonder, how many will we take.

Actually, there are several different approaches for this. One of them and is probably the easiest one to do, called **elitist**, we take the two highest fitness score members as parents to generate all the children of the next generations. Although, as I've just mentioned that it is the easiest way, but it flies in the face of the principle of variation. The children will depend on only two members of perhaps a population that contains thousands. Thus, these the next generations have little variety, and this may stunt the process of evolutionary. Instead of this method, we could make a **mating pool** containing a larger number of members. For example, the top X percentage of the population. This should guarantee some variety to the children. However, it doesn't guarantee producing optimal results.

*Part III: Reproduction*

The process of reproducing new generations mainly takes place through three operators:

- **Reproduction:** Sometimes referred to as (Selector). An operator that creates more copies of better strings in a new population. Usually it is the first operator applied on a population. Reproduction basically picks the fit members in a population to form a mating pool. And we have discussed this process earlier.
- **Crossover:** is the process of creating a child out of two parents using their genetic code. To put it another way, a crossover operator is recombination process of two population members (parents) to get a better member (child). During crossover operation, the recombination process generates different individuals in the successive generations. This is done by combining the material of two individuals that are members of the previous generation.
- **Mutation:** Once the child has been created by crossover operator, we apply one single final process which is by the way an optional one before we add this child to the next generation. That process is mutation. This operator exists to ensure the Darwinian principle of variation. On mentioning variation, here is briefly how it works. Mutation operator randomly adds some genetic information by flipping some of the bits in a chromosome (solution string).

### *C. When to Use*

Genetic algorithms are known to be robust ones that provide high quality solutions for optimization problems and search problems. The strong motivation for using genetic algorithms is that they can handle very large search space with very large number of parameters involved. That is the effect of using natural selection to pick only strong (fit) solutions.

It is not only the problem size but also the elapsed time to solve it. While most powerful computers struggle with hard optimization problems (like NP-Hard) and can take very long time to solve, Genetic algorithms provide usable-near optimal solutions in short time. (Melanie, 1998)

### *D. Problem Domain*

Problems domain that GAs appear to be appropriate to solve with include timetabling and scheduling problems. Also often applied as a technique to solve global optimization problems.

Genetic algorithms are useful in the domain of problems that have a complex fitness landscape as mixing. Mutation in combination with crossover, is designed to push the population away from the local optimal point that a traditional hill climbing algorithm might get stuck in.

**Examples of problems solved by genetic algorithms:**

- Recurrent Neural Network
- Code breaking
- Designing radio pick up antennas in space
- Learning fuzzy rule base

### *E. Limitations*

So, genetic algorithms deliver good enough solutions in fast enough time. But like any other technique, it suffers from a few limitations, which include:

- GAs can take a very long time solving nontrivial problems. (Skiena, 2008)
- Fitness function for complex problems gets repeatedly calculated. Seeking to find optimal solution for complex high-dimensional optimization problems often requires computationally expensive evaluations to the fitness function.
- GAs stand useless against decision problems where the only fitness measure is a single yes/no decision.
- If a genetic algorithm is not properly implemented, it may not converge to the optimal solution.
- No guarantee for optimality nor quality.

### *F. Global Convergence*

Just like we earlier mentioned. To get a convergence of a technique to optimal solution, we compare it to other techniques solving same problems. In his book “The Algorithm Design Manual”, Steven Skiena mentions that he has never subjected to any problem where genetic algorithms seemed to him the appropriate way to attack it. Further, he spoke about the computational results reported using GAs, that they have never favorably impressed him. Although this is a personal opinion, but if we ignore it from consideration, there’s a convergence that GAs provide near-optimal solutions. (Skiena, 2008)

# IV. ANT COLONY OPTIMIZATION

Ant colony optimization (ACO) is a probabilistic metaheuristic technique to solve combinatorial problems that is reducible to find optimal paths in graphs. The technique involves a colony of artificial ants cooperating together to find good solution or solutions to hard discrete optimization problems. Therefore, cooperation is an important key design concept for an ACO algorithms: it works by allocating the computational resources to some set of artificial ants (stands for relatively simple multi-agents) that communicate with each other indirectly using a biological pheromone-based communication (called Stigmergy). Which is an environment-mediated indirect communication. Good optimization solutions are a property of emergency to the agents' (artificial ants) cooperative interaction. (Dorigo, 2006)

## *A. Theoretical Foundation*

The brief history description of the ant colony metaheuristic optimization is basically a history of experimental researches. Trial and error was the main guide for all early researchers and still is for most of the running research efforts recently. The field of "ant algorithms" focuses on studying models that are derived and formulated by observing the real ants' behavior, then uses these derived models as an inspiration source to design novel algorithms that finds a solution for distributed control and optimization problems.

The main theoretical idea is all about the self-organizing principles of real ants which allow them to highly coordinate their behavior. This can be mimicked by exploiting these principles to coordinate the populations of the artificial multi-agents collaborating to find a solution for computational problems. Several different aspects and principles involving some behaviors of ant colonies have been an inspiration for several kinds of ant algorithms. One of these algorithms and a most successful one as an example of ant algorithms is our topic here in hand "the ant colony optimization (ACO)". ACO is basically inspired by some behaviors of ant colonies like "foraging" and communication activities via "stigmergy". The technique mainly targets discrete optimization problems. (Marco Dorigo, 2004)

## *B. How It Works*

The ACO meta-heuristic algorithm is basically running by three main procedures described briefly by this pseudo code:

```
ALGORITHM ACO_META-HEURISTIC
Schedule_Activities
Construct_Ants_Solutions
Update_Pheromones
DaemonActions % optional
End ScheduleActivities
End Algorithm
```

- **`Construct_Ants_Solutions:`** procedure manages a colony of ants that visits adjacent states of the given problem in a concurrent and asynchronous manner, by travelling in problem's construction graph through neighbor nodes. They move by performing a local decision stochastic policy that uses pheromone trails and heuristic information. By this way, ants can incrementally build or construct solutions to the optimization problem.
- **`Update_Pheromones:`** is the procedure of modifying the pheromone trails. The trails value either gets increased when ants deposit the pheromone on the connections or components they use, or decrease, because of pheromone evaporation. Depositing new pheromone increases the probability that those components or connections that were either used by at least one ant or many ants which have given us a very good solution again will be used by future ants.

- Finally, the optional **`Daemon_Actions`** procedure mission is to implement centralized actions that single ants cannot perform. Examples of these daemon actions include the activation of a local optimization process, or collecting global information which can help deciding whether it is useful or not to deposit additional new pheromone to push the process of search from a non-local perspective.

The **`Schedule_Activities`** procedure constructs these three procedures but does not specify how they are scheduled and synchronized. To put it another way, it does not determine whether they should execute completely parallel and independent, or a synchronization is necessary. Such decisions are freely left for the designer taking into consideration the problem's characteristics. (Marco Dorigo, 2004)

### *C. When to Use*

An ACO's artificial ant (agent) is represented by a stochastic constructive procedure that builds a solution incrementally by adding formal defined components of solutions to a partial solution that is currently under construction. Therefore, we can apply the ACO metaheuristic to any combinatorial optimization problem where we can define a constructive heuristic for the problem.

Although this seems to mean that we can use ACO metaheuristic to solve any interesting combinatorial optimization problems including NP-Hard optimizations of course, the real arising issue to apply this technique is how the considered problem can be mapped to a representation that artificial ants (agents) can use to build solutions. Meanwhile, ACO algorithms can be applied to find a solution for both dynamic and static combinatorial optimization problems. (Dorigo, 2006)

### *D. Problem Domain*

Here is some problem that are solved using ACO:

- Traveling Sales-man Problem (TSP)
- Job-shop Scheduling problem (JSP)
- Sequential Ordering Problem (SOP)
- Quadratic Assignment Problem (QAP)

### *E. Limitations*

The ACO algorithm is known for its strong robustness, and positive feedback among the problems it solves. However, no matter how efficient it is, still, it has with some drawbacks and limitations. In the following, some of ACO limitations (Wang, 2018):

- Ant colony algorithm is not that efficient when dealing with large-scale combinatorial problems. Due to high time complexity of ACO, it takes much longer time to handle large-scale problems.
- Since ACO involves many parameters, an improper initialization of values may lead the algorithm to get stuck at a local optimum solution.
- The ACO algorithm cannot properly deal with continuation problems.

### *F. Global Convergence*

I will get right into it. Clearly, some of the best ACO performing algorithms has been proven with and without local search, "converge in value". which means that they, sooner or later, will find the optimal solution. Concerning converge in solution, it is possible to force ACO algorithms to generate the same, optimal solution over and over each time the algorithm is run. (Marco Dorigo, 2004)

# V. BRANCH AND BOUND

Branch and Bound (B&B) is a systematic enumerative design technique for algorithms that produce a global optimization solution for nonconvex and combinatorial problems, as well as mathematical optimization. Branch and bound has wide range of use as an approach to combinatorial optimization, also including mixed integer programming, and structured prediction. (Gilles Brassard, 1996)

Although branch-and-bound often shares the discussion with the context of integer programming, and some consider it specifically limited to integer programming problems, it is in fact a general solution approach with the ability to be applied to obtain a solution for many combinatorial optimization problems even when these problems are not in the formulation of integer programs. As an example for this, there exist several efficient branch and bound algorithms that are designed specifically for job shop scheduling or quadratic assignment problems. Both branch and bound algorithmic solutions for these problems are based on their combinatorial properties and do not use their integer programming formulations. Nevertheless, it is confirmed that almost all integer programming solving techniques aim to use branch and bound to. Therefore, although branch and bound applications are very wide, but, the application of branch and bound among the context of integer programming is of special importance.

## *A. Theoretical Foundation*

The branch and bound approach first appeared as a research for discrete programming, at the School of Economics in London sponsored by British Petroleum. It was held by Ailsa Land and Alison Doig back in 1960. Then it became the most commonly used technique used to solve NP-hard optimization problems.

As I mentioned earlier, it is a systematic enumerative technique. So, the main idea is to create an enumeration of all candidate solutions. These candidate solutions are thought are thought of as to form a rooted tree. What the algorithm do is exploring the branches of this tree which represent subsets of the set of solution. It's important to note that Before the algorithm enumerates the candidate solutions of a tree branch, the branch must first be checked against lower and upper estimated bounds on the optimal solution, if the branch is found not to give us a better solution the best one found so far, it gets discarded. So, we can say that a branch and bound algorithm depends on efficiently estimated lower bounds and upper bounds of the space of search branches. And if no bounds exist nor available, the algorithm disintegrates to an exhaustive search. (Gilles Brassard, 1996)

## *B. How It Works*

The branch and bound approach is built over the idea of partitioning the overall set of feasible solutions into smaller solution sub-sets. Then this smaller sub-sets are evaluated systematically until the best solution is found.

The target of the algorithm is to find a specific value that maximizes or minimizes the value of a given objective function. This process is done among some set of admissible candidate solution. This set of solutions is called the search space or the feasible region of solutions. Now that this solution search space exists, the algorithm operates in accordance with two main principals:

- First, the algorithm recursively partitions or divides the search space into smaller spaces, then optimize the objective function (either minimizing or maximizing) on these smaller spaces; this splitting process is called **branching.** To implement this process in an algorithm, we need some-kind of a data structure to represent these sets of candidate solutions. Such a representation is referred to as an instance of the problem. In this case, the branching operation forms a tree of instances.
- The second is keeping track of lower and upper bounds of the optimization value we are trying to find. And why is that ? because his is what differs and improves the performance of this technique from and over the exhaustive search of the brute force method, as branching alone would lead to an enumeration by brute force to the candidate solutions and test all of them. So, we use these bounds to prune the search space and eliminate or disqualify the candidate solutions that is proven to be lacking the capability of providing an optimal solution. And this process is called **bounding.** And there comes the name **Branch and Bound.**

### *C. When to Use*

Branch and Bound is a widely used technique. But the main interest of it is solving discrete optimization and large scaled NP-hard combinatorial optimization problems. Such problems require to find an optimal solution from a finite set of solutions known as the feasible region. In many problems like these, an exhaustive search method is not tractable. The algorithm operational environment is among the domain of those optimization problems where the set of feasible solutions is discrete or at least can be reduced to discrete, and when the target is to find the optimal solution. So, there is where this technique is applicable.

### *D. Problem Domain*

As I earlier mentioned, branch and bound technique is widely used to solve NP-hard combinatorial optimization problems, including:

- Integer Linear Programming (ILP)
- Non-linear Programming
- Flow shop scheduling
- 0/1 knapsack problem
- Travelling salesman problem (TSP)
- Cutting stock problem
- Quadratic assignment problem (QAP)
- Set inversion
- Parameter estimation
- Computational phylogenetics

### *E. Limitations*

A combinatorial problem with limited size of enumeration where the feasible region and search space are well defined, a branch and bound algorithm can efficiently search and find an optimal solution. Although the worst case running time is exponential in the, a branch and bound algorithm can be utilized to a ‘reduced-space’ algorithms that requires branching only over a subset of the variables instead of the “full-space” to guarantee convergence.

### *F. Global Convergence*

A combinatorial problem with limited size of enumeration where the feasible region and search space are well defined, a branch and bound algorithm can efficiently search and find an optimal solution. Although the worst case running time is exponential in the, a branch and bound algorithm can be utilized to a “reduced-space” algorithms that requires branching only over a subset of the variables instead of the “full-space” to guarantee convergence.

# VI. THE 0/1 KNAPSACK PROBLEM

The knapsack problem is one of the most famous, proved to be, NP problem. Although its description is pretty easy to understand, but, among the history of computer since, large number of algorithms has been applied to solve it. Being a combinatorial optimization problem, allows it to be a source of interest for algorithm designers. In this section we are going, first, to describe the problem, then explain the application of different surveyed techniques to solve it, and eventually introduce a comparison between time complexities metrics of each one.

## *A. Problem Description*

Given a group of items, each of which is associated with a weight and a value. The target is to fill a limited capacity-knapsack with the most valuable items possible. The representation of these parameters may vary according to the solving algorithm characteristics, it may be a list of an encapsulated type Item including its data variables, or they may be represented by two arrays (weights[] , and values[]) associated to each other. But this a coding issue. What we are concerned with here is some properties of the problem:

- **Optimization problem:** the target is to maximize a variable (value).
- **Combinatorial:** meaning that we are searching for the optimal solution among a finite set of solutions where the set of feasible solutions is discrete.
- **Bounded:** meaning that an item either be picked in the knapsack or not (repetition of items is not allowed), that's why it's called " 0/1 Knapsack"
- **Constrained:** meaning that the Knapsack has a max weight capacity limit that cannot be exceeded.

## *B. Problem Solutions*

In this part we will explain the application of each technique's handling methods steps to solve the 0/1 Knapsack problem.

It is important to note that, for ease of comparison and analysis, the code architecture and the programming language used are the same. Using java: a derived main test class to initialize a Knapsack with the problem's parameters, and then invoke the method:

```
Solve_Technique.Name_Knapsack
```

With the `Technique.Name` replaced with the initials of the technique used to solve each time. This method implements each technique's steps to solution.

### *Dynamic programming (bottom-up manner approach)*

Just like any other basic DP algorithm, we do the following:

- In order to avoid the regular recursion method re-computations of same overlapping sub-problems, construct a temporary array K[][] in bottom-up manner. Where all weights (1:max_capacity) as columns, and possible to keep weights as rows.
- Fill each K[i][j] with maximum values of (j-weight) with consideration to all values in the range of (1:ith)
- Eventually, return a maximum of these values.

*Genetic algorithms*

Following the procedures of a typical GA algorithm and applying the genetic operators as follows:

- First of all, RANDOMLY initialize the population of first generation of possible solutions. These solutions are represented by binary strings with bits number equal to the number of items (e.g. 101011 if there is 6 items) where 1 means item is picked in the knapsack for this string-solution, and 0 means not picked.
- For each generation, there is a variable that holds the best fit candidate solution. And what is that? It is according to the fitness function which is evaluated for each and every candidate solution in each generation. Here in our problem, the fitness of a solution simply represents the total value of its items.
- Using natural selection principle by employing the **elitist** method. Only the **most two** fit solutions are picked as parents to produce the next generations.
- The crossover process takes place with the parent chromosomes (candidate solutions)
- In order to guarantee variation among successive generations, the new generation children get mutated by randomly flipping their bits.
- Crossover and mutation processes are applied with respect to a pre-defined probability.
- Keep producing new generations by applying the past processes.
- The most fit solution in the last generation is returned as the near-optimal or may be the optimal solution. Since GA is evolutionary approximation approach not an exact one.

*Ant colony*

The following pseudo-code illustrates the applied steps of ACO technique in the implemented java code:

```
ACO KNAPSACK ALGORITHM
while (stopping criteria not true)
      while (an ant hasn't worked yet exists)
            while ((int C = max_capacity) > 0 && (int i = 0) < items)
                  get a next object O from N with a probability
                  add O to a partial solution S += O
                  update the max weight capacity C -= O.weight
                  update the current_value += O.value
                  update the current state neighborhood N = {O:O.weight = C}
            end while
            if a better solution was found, update the best solution
      end while
      if a better solution was found, update global best solution
      max_value = current_value
      apply the pheromone evaporation mechanism x[j] = Px[j]
      update pheromone trails x[j] += Px[j], P = {0,1}
end while
end algorithm
```

*Branch and bound*

- Start by sorting all items in a descending manner according to their ratio of value per unit weight (value/weight), so that we can use a greedy approach to compute an upper bound.
- The idea of using greedy approach is using it to check either or not a specific node can produce us a better solution than the one we have. The greedy approach is perfect for this mission since it uses local search technique.
- Initialize the solution to return with max_value = 0
- Create an empty queue, instantiate a dummy decision node with weight and value equal zero, and push it to the queue.
- While the queue is not empty, repeat the following steps:
    1. Extract an item from the queue.
    2. Evaluate the value of next level node. If it gives us a higher value than the one we have, then update the max_value.
    3. Calculate bound of next level node. If it gives a bound more than max_value, then add next level node to the queue.

*Divide and conquer and recursive brute force*

The brute force algorithm uses the exhaustive search to find the optimal solution. With n items to pick from, there is (2n) different combinations of solutions to be generated. Then certify each of them to satisfy the max_capacity constraint and save the max_value certified solution to return it. If we use a recursive brute force instead of the iterative style, it will be similar to the divide and conquer approach as both will yield a time complexity of O(2n). Therefore, if the size of the problem is large, both will not be applicable.

*Greedy algorithm*

A greedy approach to solve this problem, just like any other greedy algorithm, will search for the next local optimal value to pick. So, it will decreasingly pick items according to their value per unit weight until the max_capacity constraint is reached. Therefore, the greedy approach, eventually in this case may not produce the correct solution (it might coincide to be the optimal solution in some cases). But in general, the greedy approach is not used to solve the 0/1 knapsack problem.

### *C. Time Complexity Analysis and Comparison*

| Approach | Big O | Big Θ | Big Ω | Where |
|---|---|---|---|---|
| DP | O(n*C) | Θ(n*C) | Ω(n*C) | C = max capacity |
| GA | O(n) | Θ(n) | Ω(n) | n = no of items |
| ACO | O(A*n*C) | Θ(A*n*C) | Ω(A*n*C) | A = no of ants |
| B&B | $O(2^n)$ | $\Theta(2^n)$ | $\Omega(2^n)$ | all defined |
| Brute Force | $O(2^n)$ | $\Theta(2^n)$ | $\Omega(2^n)$ | all defined |

## VII. REFERENCES

[1] Alfred V. Aho, J. E. (1974). *The Design and Analysis of Computer Algorithms.* Boston: Addison-Wesley.

[2] Billman, R. (1954). *The Theory of Dynamic Programming.* Santa Monica: Rand Corporation.

[3] Dorigo, M. B. (2006). *Ant Colony Optimization, Artificial Ants as a Computational Intelligence Technique.* IEEE COMPUTATIONAL INTELLIGENCE MAGAZINE.

[4] Gilles Brassard, a. P. (1996). *Fundamentals of Algorithmics.* Pearson.

[5] Karumanchi, N. (2017). *Data Structures and Algorithms Made Easy.* CareerMonk Publications.

[6] Marco Dorigo, a. T. (2004). *Ant Colony Optimization.* London: Massachusetts Institute of Technology.

[7] Mathew, T. V. (n.d.). *Genetic Algorithm.* Mumbai: Indian Institute of Technology.

[8] Melanie, M. (1998). *An Introduction to Genetic Algorithms.* London: Massachusetts Institute of Technology.

[9] Shiffman, D. (2012). *The Nature of Code.* California: Daniel Shiffman.

[10] Simon, D. (2013). *Evolutionary Optimization Algorithms.* New Jersey: John Wiley & Sons, Inc.

[11] Skiena, S. S. (2008). *The Algorithm Design Manual.* New York: Springer.

[12] Wang, G. (2018). *A Comparative Study of Cuckoo Algorithm and Ant Colony.* Beijing: Beijing University of Posts and Telecommunications.