CHAPTER 1: Background & Problem Formulation

1.1 Background

1.1.1 Subset Sum Problem (SSP)

SSP is an important problem in complexity theory and cryptography. SSP can simply be described as: given a set of positive integers S and a target sum t, is there a subset of S whose sum is t? For example, given the set {1, 2, 3, 4} and t=5, the answer is yes because the subsets {1, 4} and {2, 3} sum to 5. An instance of the SSP is a pair (S, t), where S = {x1, x2, ..., xn} is a set of positive integers and t (the target) is a positive integer. The decision problem asks for a subset of S whose sum is t [27].

In the optimization problem, finding the subset S' of S whose sum is largest number but not larger than t. Given a set of n data items with positive weights and a capacity c, the decision version of SSP asks whether there exists a subset whose corresponding total weight is exactly the capacity c; the maximization version of SSP is to find a subset such that the corresponding total weight is maximized without exceeding the capacity c [24]. For example, a truck that can ship no more than t pounds, and n different boxes to ship. Suppose weight of i^{th} box is xi pounds. Fill the truck with as many boxes as possible without exceeding its weight limit [17].

Problem has many applications; for example, a decision version of SSP with unique solutions represents a secret message in a SSP-based cryptosystem. It also appears in more complicated combinatorial problems [8], scheduling problems [2] [20], 0-1 integer programs [10] [1], and bin packing algorithms [22] [32]. The Subset-Sum Problem (SSP) is one the most fundamental NP-complete problems, and perhaps the simplest of its kind. The complexity of subset sum can be viewed as depending on two parameters: n, the number of values, and m, the precision of the problem (number of bits required to state the problem).

1.1.2 Complexity classes

Computational complexity theory is a branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other. A computational problem is understood to be a task that is in principle amenable to being solved by a computer, which is equivalent to stating that the problem may be solved by mechanical application of mathematical steps, such as an algorithm.

The purposes of complexity theory are to ascertain the amount of computational resources required to solve important computational problems, and to classify problems according to their difficulty [6]. The complexity class is a set of problems of related resource-based complexity. The resource may be time or space. The complexity classes could also be defined in terms of decision problems whose output is a single boolean value: Yes or No.

Typically, a complexity class is defined by a model of computation, a resource (or collection of resources) and a function known as the complexity bound for each resource. The models used to define complexity classes fall into two main categories: machine based models, and circuit-based models. Turing machines (TMs) and random-access machines (RAMs) are the two principal families of machine models. Circuits were originally studied to model hardware. The hardware of electronic digital computers is based on digital gates, connected into combinational and sequential networks. Also, circuits well capture the notion of non-branching, straight-line computation.

The class of decision problems that are solvable in polynomial time is denoted by P [19]. The class P contains many familiar problems that can be solved efficiently, such as finding shortest paths in networks, parsing context-free grammars, sorting, matrix multiplication, and linear programming. P contains all problems that can be solved by (deterministic) programs of reasonable worst-case time complexity [6].

The class NP can also be defined by means other than nondeterministic Turing machines. NP equals the class of problems whose solutions can be verified quickly, by deterministic machines in polynomial time. Equivalently, NP comprises those languages whose membership proofs can be checked quickly. The set of decision problems where the verification by a Yes and a No answer quickly with a certificate is Class NP and Class co-NP respectively. For example, one language in NP is the set of composite numbers,

written in binary. A proof that a number z is composite can consist of two factors $z1 \geq 2$ and $z2 \geq 2$ whose product z1z2 equals z. This proof is quick to check if z1 and z2 are given, or guessed. Correspondingly, one can design a nondeterministic Turing machine N that on input z branches to write down "guesses" for z1 and z2, and then deterministically multiplies them to test whether z1.z2 = z. Then L(N), the language accepted by N, equals the set of composite numbers, since there exists an accepting computation path if and only if z really is composite. Note that N does not really solve the problem, it just checks the candidate solution proposed by each branch of the computation. [6]

There exist a large number of practical problems in NP such that if any one of them were in P then the whole of NP would be equal to P. The evidence that supports the conjecture $P \neq NP$ therefore also lends credence to the view that none of these problems can be solved by a polynomial-time algorithm in the worst case. Such problems are called NP-complete. To be NP-complete, a decision problem must belong to NP and it must be possible to polynomially reduce any other problem in NP to that problem [15]. The hardest problem in NP is contained in NP-complete (NPC) class. There is no fast solution for NPC. Also, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. These are the hardest problems in NP, in the sense that if there would be a solution to an NP-complete problem then there would be a solution to any problem in NP [18].

NP-hard class problems are as hard as the hardest problems in NP, the problems do not have to be elements of NP, indeed, they may not even be decidable problems, for example the halting problem. No NP-hard problem can be solved in polynomial time in the worst case under the assumption that $P \neq NP$ [15].

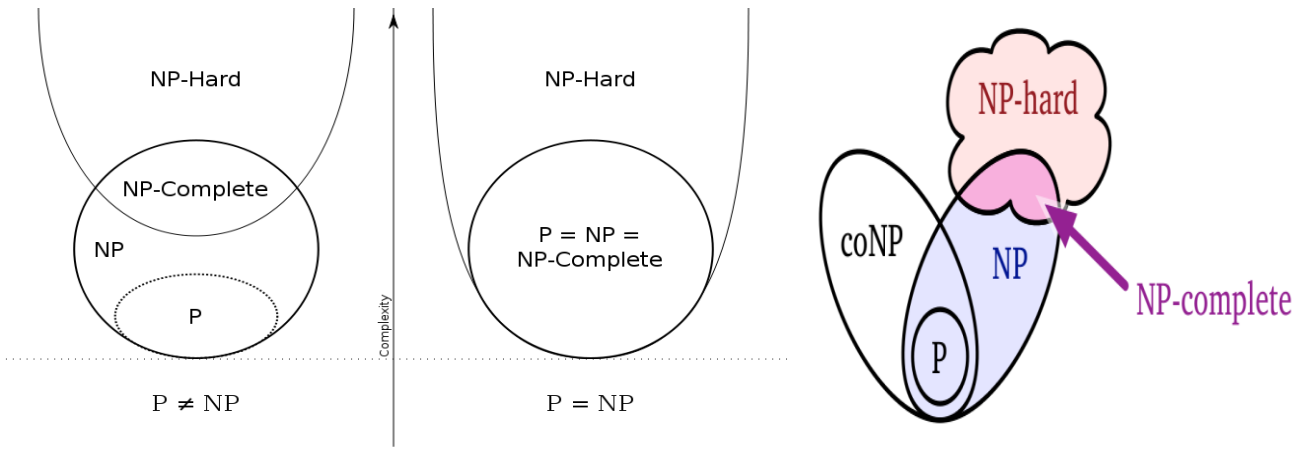

Figure1: Euler diagram for P, NP, NP-complete, and NP-hard set of problems

A numeric algorithm runs in pseudo-polynomial time if its running time is polynomial in the numeric value of the input, but is exponential in the length of the input – the number of bits required to represent it.

Let's consider the problem of checking whether a number N is prime or not. Suppose N is 3127. One way of solving this would be to check whether any number from 2 to 3127 divides N or not. Assuming that the divisor check in constant time O(1) can be performed, this would take at most 3127 steps. It can easily be seen that for a general number N, this would need O(N) steps. So is there any polynomial time algorithm for primality testing? Not really. This is because 3127 when given as input is not 3127 bits long. It is just 4 digits long (in base 10) or 12 binary bits long. This is real input and its length is not equal to N. It is of the order of log(N). And hence apparently polynomial time algorithm actually takes $O(2^{\log(N)})$ steps, that is, it is exponential in the input size. Algorithm that runs in time which is a polynomial in the input size, not the value the input represents. And hence the above algorithm isn't truly a polynomial time algorithm. Such algorithms are called pseudo-polynomial time algorithms.

The term sub-exponential time is used to express that the running time of some algorithm may grow faster than any polynomial but is still significantly smaller than an exponential. Or, a sub-exponential-time algorithm is one whose running time is a function of the size x of its input grows more slowly than b^x for every base b > 1 [26]. The term $2^{O(\sqrt{x})}$ denotes the sub-exponential complexity.

1.1.3 Approaches for solving SSPs

1.1.3.1 Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution. It is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem. It is also known as depth-first search or branch and bound. By inserting more knowledge of the problem, the search tree can be pruned to avoid considering cases that don't look promising. Representing it in a

binary state space tree as:

1. Starting at Root, the options are A and B. Choose A.
2. At A, options are C and D. Choose C.
3. C is not a solution. Go back to A.
4. At A, already tried C, and it failed. Try D.
5. D is not a solution. Go back to A.
6. At A, no options left to try. Go back to Root.
7. At Root, already tried A. Try B.
8. At B, options are E and F. Try E.
9. E is a solution. Congratulations!

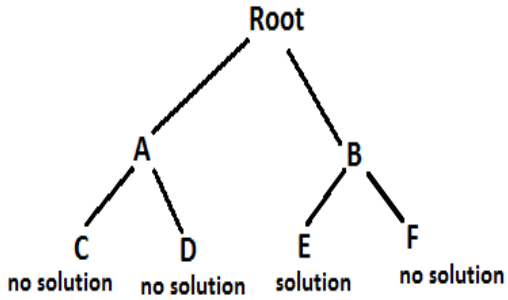

Figure2: Backtracking

1.1.3.2 Dynamic Programming

Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler sub-problems. It is applicable to problems exhibiting the properties of overlapping sub-problems and optimal substructure. When applicable, the method takes far less time than other methods that don't take advantage of the sub-problem overlap.

In order to solve a given problem, using a dynamic programming approach, solve different parts of the problem (sub-problems), and then combine the solutions of the sub-problems to reach an overall solution. The dynamic programming approach seeks to solve each sub-problem only once, thus reducing the number of computations: once the solution to a given sub-problem has been computed, it is stored or "memo-ized": the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating sub-problems grows exponentially as a function of the size of the input. Let's consider the set S = {2, 3, 4, 5} and let t = 8. The worked out DP can be tabulated as:

Table1: Dynamic Programming

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
2	T	F	T	F	F	F	F	F	F
3	T	F	T	T	F	T	F	F	F
4	T	F	T	T	T	T	T	T	F
5	T	F	T	T	T	T	T	T	T

Since A[5, 8] = T rue , there exists a subset of S that sum up to t(8). [5]

1.1.3.3 Dynamic Dynamic Programing (DDP)

This algorithm is the extension of the dynamic programming with a dynamically allocated list of target sums. It splits the input array into two sub-arrays. Perform dynamic programming to produce a target map, and backtracking to enumerate the targets of subsets. The search terminates successfully when a subset T is discovered such that $t - \Sigma(S)$ is in the target map. This hybrid approach has some of the advantages of each previous method, while quadratically reducing the space complexity.

The list of target sums is initialized with the original target. For each yi in a set of positive integers S= {y1,y2,.....yn}, targets is added to the list by subtracting yi from the existing targets. The list grows with each of the early iterations, reaching a peak in iteration i when 2^i first exceeds the maximum value on the current target list. From that point on, the list shrinks in size. The new targets added to the list are pruned from the list when the sum of the remaining numbers in the set is not sufficient to reach them, also suppress duplication of targets on the list. [3]

1.2 Introduction

SSP is an important problem in complexity theory and cryptography. SSP can simply be described as: given a set of positive integers S and a target sum t, is there a subset of S whose sum is t? For example, given the set {1, 2, 3, 4} and t=5, the answer is yes because the subsets {1, 4} and {2, 3} sum to 5. Thus, SSP is decision problem that seeks answer either yes or no. the optimization problem associated with this decision problem arises in many practical applications. In the optimization problem subset S' of S whose sum is largest number but not larger than t it to be found.

The complexity of subset sum can be viewed as depending on two parameters: n, the number of decision variables or number of values, and m, the precision of the problem (number of binary place values or bits that it takes to state the problem). The complexity of the best known algorithms is exponential in the smaller of the two parameters m and n. Thus, the problem is most difficult if n and m are of the same order. It only becomes easy if either n or m becomes very small. If n is small, then an exhaustive search for the solution is practical. If m is a small fixed number, then there are dynamic programming algorithms

that can solve it exactly [7]. There are two ways to count the solution space in the SSP. One is to count the number of ways the variables can be combined. There are 2^n possible ways to combine the variables. However, with n = 10, there are only 1024 possible combinations to check. These can be counted easily with a branching search. The other way is to count all possible numerical values that the combinations can take. There are 2^m possible numerical sums. However, with m = 5 there are only 32 possible numerical values that the combinations can take. These can be counted easily with a dynamic programming algorithm.

SSP is interesting because, depending on what parameter is used to characterize the size of the problem instance, it can be shown to have polynomial, sub-exponential, or strongly exponential worst-case time complexity. It is known to be NP Complete [3] and hence difficult problem to solve generally. There are several ways to solve SSP in exponential and polynomial time. A naive algorithm with time complexity $O(n.2^n)$ solves SSP by iterating all the possible subsets and each for its subset comparing its sum with target t. A backtracking algorithm for SSP can be modeled as a binary tree where each node represents a single activation of the recursive code. Each activation processes one element of S, and it makes at most two recursive calls. So the total number of recursive calls cannot exceed the number of nodes in a full binary tree of depth n, and the worst-case time complexity is $O(2^n)$ when size of the input set (denoted n) is used as the complexity parameter [7]. When the maximum value in the set (denoted m) is used as the complexity parameter, dynamic programming can be used to solve the problem in $O(m.n^2)$ time, which is polynomial in n and m. If $m = 2^n$, $O(m.n^2)$ is really $O(n^2. 2^n)$, which is called pseudo-polynomial time complexity. This is actually worse than $O(2^n)$ – worse than backtracking. A variant of dynamic programming called Dynamic Dynamic programming (DDP) has been shown to have a worst-case sub-exponential time complexity of $2^{O(\sqrt{x})}$ when the total bit length x of the input set is used as the complexity parameter [7].

Thomas E. O'Neil [7] showed that DDP has lower step counts than both of the other algorithms for medium-density to low-density problem instances. This dissertation work evaluates the performance of BT, DP and DDP algorithms empirically in terms of total bit length used to represent input sets.

1.3 Problem Definition

All the algorithms: Backtracking, Dynamic Programming, and Dynamic Dynamic Programming have strongly exponential time complexity when the complexity parameter is the number of integers in the

input set. At the same time, DDP is known to have sub-exponential worst-case time complexity when the complexity parameter is the total bit length of the input set [7]. Now the question is: What happens when total bit length of input set is used to evaluate performance of above mentioned algorithms for solving SSP? This dissertation work answers the question.

1.4 Objective

The main objective of this research work is to evaluate performance of the algorithms: BT, DP and DDP for solving SSPs in terms of total bit length used to represent the inputs.

1.5 Motivation

Subset Sum Problem is an important problem in cryptography, scheduling and many more. It is an NP Complete problem. The general algorithms for solving SSP have exponential complexity. The complexity of SSP could be studied depending on two parameters: n and m, where n is the input size and m is the total bit length. When n is considered as the complexity parameter, the algorithms: DDP, DP and BT have exponential complexity. And when m is considered as the complexity parameter, DDP has sub-exponential complexity, but the complexity of DP and BT was unknown. So, to explore their response through the empirical analysis is motivational.

1.6 Report Organization

The background part of this dissertation work focuses on Subset Sum Problems (SSP), the complexity classes and the algorithms: Dynamic Dyamic Programming (DDP), Dynamic Programming (DP) and Backtracking (BT), for solving SSP. Also, a briefly introduced the SSP in context of the complexities of the SSP solving algorithms when the parameters are input-size and total bit-length.

The problem formulation part states the main problem for which this dissertation work is going to have it as its goal. The goal is stated in its 'Objective'. And the motivation to this dissertation is described in 'Motivation' section.

Chapter 2 consists of literature review which verifies reviews the related topics. Literature review includes summary of definitions of SSP, algorithms: DDP, DP & BT for solving SSP and their complexities when the evaluating parameter is input size or total bit length of the input set. This chapter also contains the research methodology which shows the flow of this dissertation work.

Chapter 3 consists of 'Design & Implementation' section which lists the tools, programming language and data-structure used. It also describes the algorithms for solving SSP, their flow and trace each algorithm in detail.

Chapter 4 consists of 'Data collection & Analysis' section. The data collection part describes the number of data collected, the maximum number and which ranges they are limited to. The analysis part analyzes the different cases of N (input size) and M (total bit-length) by plotting the graph: M versus the step-counts.

Finally, Chapter 5 consists of 'Conclusion and Limitations' of this whole dissertation work. This section also shows the guidelines for future research.

Chapter 2: Literature Review & Methodology

2.1 Literature Review

SSP is a well-known hard (NP-complete) problem that is generally solved by algorithms: BT, DP, and DDP. It is known to have a pseudo-polynomial-time solution [5]. Time complexity may be polynomial, sub-exponential, or exponential depending on the parameter chosen to characterize the size of the problem instance. Let n represent the size of the input set S, and let m be the maximum value in the set. A standard dynamic programming algorithm for the problem can be shown to have polynomial time complexity $O(n^2 m)$. On the other hand, there is an algorithmic model that includes both backtracking and dynamic programming in the research literature that is shown to have a strongly exponential lower

bound of $2^{\Omega(n)}$ on the closely related Knapsack problem when n alone is used as the complexity parameter. And finally, a variant of dynamic programming called Dynamic Dynamic Programming has been shown to have a worst-case sub-exponential time complexity of $2^{O(\sqrt{x})}$ when the total bit length x of the input set is used as the complexity parameter. [7]

The SSP is a special case of the knapsack problem and in the cryptology literature is often referred to as the knapsack problem. The SSP is hard; its decision problem was shown to be NP-complete by Karp [25]. SSP is the general form of Partition problem. In Partition problem the Set is partitioned into two subsets that have the same sum where as in the SSP a subset of the Set has to meet a target integer [3]. It will be shown that the related search problem of actually finding a solution, even when a solution is known to exist, is at least as hard as any NP-complete problem [29].

The experiment in which the number of integers in the input set is considered as the complexity parameter, all three algorithms have strongly exponential time complexity. But DDP is known to have sub-exponential worst-case time complexity when the complexity parameter is the total bit length of input set. This suggests that an additional experiment, one in which step counts are plotted as a function of total bit length of input, is needed to further corroborate published analytical results. [7]

A slightly more efficient algorithm checks out all possible 2^n subsets. One typical way to do this is to express all numbers from 0 to $(2^n - 1)$ in binary notation and form a subset of elements whose indexes are equal to the bit positions that correspond to 1. For example, if n is 4 and the current number, in decimal, is say 10 which in binary is 1010. Then check the subset that consists of 1^{st} and 3^{rd} elements of S. One advantage of this approach is that it uses constant space. In each iteration, examine a single number. But this approach will lead to a slower solution if |S' | is small. Consider the case where t = S[n /2]. Examine around $O(2^{n/2})$ different subsets to reach the solution.[5]

An obvious exponential-time search BT algorithm successively generates all subsets and computes their sums. But DP is not so obviously exponential, it uses a Boolean array A[t] with index range from 0 to t (target) with A[0] initially true and the rest false. The problem is solved in n passes over the array, one pass for each x in the set. During the i^{th} pass $A[j+x_i]$ is set to true for each j where $A_{i-1}[j]$ is true. A subset with the target sum t is discovered if A[t] becomes true. The complexity depends on m (max number in the set). The complexity is generally polynomial. But the hard instances of the problem have $m = O(2^n)$.

In computational complexity theory, problems within the NP-complete class have no known algorithms that run in polynomial time. NP-complete problems can still be solved, but either the input data must be restricted to reasonably small sizes to accommodate super-polynomial time algorithms or accuracy must be compromised in implementing faster approximation algorithms, neither of which are amenable conditions. [30]

The DP algorithm is considered to be pseudo-polynomial because it behaves as a polynomial time algorithm for large elements in S and relatively small T, but it is not actually polynomial time. However, it is reasonable to conclude that its runtime is $O(n\ 2^n)$ because this represents the worst-case conditions according to order of growth analysis, and one cannot ensure that T is indeed bounded by the sum of the elements in the set. Note that the complete search algorithm given earlier also runs in $O(n\ 2^n)$. Although the time complexities of both algorithms are identical, the dynamic programming one is generally faster due to its use of optimal substructure and overlapping sub-problems. In fact, this is the fastest known runtime of any classical algorithm for the SSP. [30]

DDP is the algorithm that combines BT and DP. The input set is ordered and partitioned into a denser and sparser subset. BT is employed on the sparse subset, while DP is used for the dense subset. The results are combined to achieve time complexity $2^{O(\sqrt{x}),}$ where x is the total length in bits of the input set. A simpler algorithm that achieves a similar time complexity is defined and it can be used for both SSP and Partition problem. [3]

In order to apply DP, the SSP must exhibit optimal substructure and overlapping sub-problems. Optimal substructure appears when the solution to a problem relies on the solutions to smaller cases. In the SSP, suppose that one element xj of the solution subset is known. The original problem is now reduced to finding a subset of n - 1 elements that adds up to T - xj, so this sub-problem consists of fewer elements and a smaller sum. Thus, an algorithm for the SSP can utilize optimal substructure by iterating over all xi to create the sub-problems, iterating over xi recursively on those sub-problems until a base case is reached, and then conflating the solutions in order to solve the original problem, thereby reducing the number of time-consuming operations done.[25]

The DP algorithm is considered to be pseudo-polynomial because it behaves as a polynomial time algorithm for large elements in S and relatively small T, but it is not actually polynomial time as previously shown. However, it is reasonable to conclude that its runtime is $O(n\ 2^n)$ because this represents

the worst-case conditions according to order of growth analysis, and one cannot ensure that T is indeed bounded by the sum of the elements in the set. Note that the complete search algorithm given earlier also runs in $O(n\ 2^n)$. Although the time complexities of both algorithms are identical, the dynamic programming one is generally faster due to its use of optimal substructure and overlapping sub-problems. In fact, this is the fastest known runtime of any classical algorithm for the SSP.[25]

There is no mutual dependence between the number of objects in the set n and the maximum value m. The bit length of a problem instance is $O(n \cdot \log m)$, and analysis based on this measure actually yields a result that is distinct from $2^{O(n/2)}$ and $O(m^3)$. Stearns and Hunt [30 used input length x to demonstrate that an algorithm for the Partition problem (a special case of Subset Sum) exhibits sub-exponential time: $2^{O(\sqrt{x})}$. The significance of this result was probably obscured by the claim in the same paper that the Clique problem is also sub-exponential, while its dual problem Independent Set remains strongly exponential. This apparent anomaly is a representation-dependent distinction, and it disappears when a symmetric representation for the problem instance is used [28]. Sub-exponential time for Partition, however, appears to have stronger credibility. This result was replicated explicitly for Subset Sum (using a different algorithm) in [3], and it seems unlikely that symmetric representation will make it disappear. This sets the stage for the current study, in which empirical evidence that instances of Subset Sum where the input set is dense (n is $\Theta(m)$) are very easy to solve. The ultimate goal, beyond the scope of this paper, is to develop an algorithm for Subset Sum that remains sub-exponential even under the test of symmetric representation for the input set. This would solidify the argument that Subset Sum is truly an easier hard problem. [21]

Subset Sum is apparently has upper bound $O(\ 2^{n/2})$ when size of the input set (denoted n) is used as the complexity parameter . When the maximum value in the set (denoted m) is used as the complexity parameter, DP can be used to solve the problem in $O(m^3)$ time. The SSP is known to be NP-Complete [12] and hence difficult problem to solve generally. Cook, Karp and others, defined such class of problems as NP Hard problem [16]. Some of the NP Hard problems include Travelling Salesman Problem (TSP), Boolean Satisfiability Problem, Knapsack Problem, Hamiltonian Path Problem, Post Correspondence Problem (PCP), and Vertex Cover Problem (VCP). There are several ways to solve SSP in exponential and polynomial time. A naive algorithm with time complexity $O(n2^n)$ solves SSP by iterating all the possible subsets and each for its subset comparing its sum with target X. A better algorithm proposed in 1974 using the Horowitz and Sahni decomposition scheme which achieves time $O(n\ 2^{n/2})$. If the target T is relatively small then there exist dynamic programming algorithms that can run much faster. A classic Pseudo-Polynomial algorithm Bellman Recursion solves SSP in both time and

space O(nc). And there are many other algorithms, for example Ibam and Kim [13] developed a fully polynomial approximation scheme for the SSP in 1975. It was improved upon by Lawler [14] and Lam by Martello and Toth [31]. Martello and Toth reported very good results for several approximation schemes in their survey and experimental analysis [9]. [4]

2.2 Methodology

The research is totally experimental. It is a traced driven approach in which random numbers of input size (n) varying from 15 to 25 are generated for the input set, the maximum number (m) of the input set is set to the number of the parameterized bit length. That is, if the parameterized bit length is 10 then the maximum number in the set is set to 1024. For the better result, m is varied from 10 to 20. The algorithms: BT, DP and DDP are implemented on those randomly generated sets on above bounding cases (n = 15 to 25 and m = 10 to 20) and step counts are noted for each set instances. Average of 50 similar instances is calculated and a graph of the average step-count verses the total bit length is plotted.

Chapter 3: Design & Implementation

3.1 Tools & Language: The tools used during the dissertation work are listed as follows:

1. **Sublime Text 2:** It's the text editor in which the ruby code in .rb file extension is written. The text editor is very easy to use for multiple selections to rename variables quickly.
2. **iTerm:** The written code are run through the terminal to display the output: step-counts. Its very easy for the repeating command and copy paste feature.
3. **LibreOffice Calc:** This is the spreadsheet program for storing the step-counts for different set instances. It is intuitive, easy to learn and has a comprehensive range of advanced functions.
4. **QtiPlot:** It is a fully fledged plotting software. It can make two and three dimensional plots of

publication quality, both from datasets and functions. It can do non-linear fitting and multi-peak fitting. The tool has been used to plot the graph: step-count vs. total bit length.

5. **Ruby:** The programming language used is "**ruby 1.9.3p484**". It is an object oriented programming language. It can easily be formatted to fit the needs and work as efficiently as possible. There are alots of built-in functions which can be used to make the task easy and efficient. Some of the used built-in functions are **to_a.sample**, **times**, **sort**, **reverse**, **length**, **include?**, **Math.log2** etc. The only array data-structure is enough for the implementation.

3.2 Algorithms

3.2.1 Dynamic Dynamic Programming (DDP)

DDP is the extension of the dynamic programming. In dynamic programming the list of sum set is maintained. The length of the sum-set list is fixed. But in DDP the list of target is maintained which can grow and shrink in size. The search terminates successfully when the processing number is in the list of targets. The algorithm for the DDP can be stated as follows:

Algorithm

procedure: Dynamic Dynamic Programming

input: S-Set of numbers, target

output: true or false

1. if (target = 0 or target = $\Sigma(S)$)
   - solution found
2. else if (target > $\Sigma(S)$)
   - solution not found
3. else
   - if (target > $\Sigma(S)/2$)
     - target = $\Sigma(S)$ − target
   - if (target $\in$ S)
     - solution found
   - else
     - List target_list = {target}
     - sum_of_ rest = $\Sigma(S)$

- for each num in S from high to low
    - sum_of_ rest = sum_of_ rest − num;
    - List newlist = {}
    - for each t in target_list
        - if (t − num = 0 or t − num = sum_of_ rest)
            - solution found
        - else if (t − num > 0 and t − num < sum_of_ rest)
            - newlist.append(t − num)
        - else
            - newlist.truncate(sum_of_ rest)
    - target_list.mergewith(newlist)

4. Terminate

At first, the list of targets is initialized with the original target. Then for each number in the given set is processed and subtracted from each target in the list of targets. The solution is found if the result from subtraction is found in the targets list. Otherwise, the result is appended in targets list if its positive and the result is less than the sum of the remaining numbers to be processed in the given set. The targets which are greater than the sum of numbers to be processed are removed. In this way, the appending increases the length of targets list and removal of targets decreases the length of the targets list.

3.2.2 DDP Example Trace

Lets trace DDP for SSP in which S = {2, 3, 5, 7, 10}, and t = 14.

Here, $\Sigma S = 27$.

The steps: 1 and 2 do not get satisfied and second 'else' of the step 3 is reached assigning target_list = {14} and sum_of_rest = 27.

processing each number in S = {10, 7, 5, 3, 2}(descending order),

Step1: Take number 10

sum_of_rest = 27 – 10 = 17

new_list = {}

for each target in target_list ie {14}

processing target 14

checking the condition (t − num = 0 or t − num = sum_of_ rest) => (14 – 10 ==0 or 14 – 10 == 17) results negative

checking the condition (t − num > 0 and t − num < sum_of_ rest) => (14 – 10 > 0 and 14 – 10 < 17) results positive, so new_list += {t – num} => new_list = {14 – 10 } => new_list = {4}

after merging new_list with target_list, target_list = {14, 4}

Step2: Take number 7

sum_of_rest = 17 – 7 = 10

new_list = {}

for each target in target_list {14, 4}

processing target is 14,

checking the condition (t − num = 0 or t − num = sum_of_ rest) => (14 – 7 ==0 or 14 – 7 == 10) results negative

checking the condition (t − num > 0 and t − num < sum_of_ rest) => (14 – 7 > 0 and 14 – 7< 10) results positive, so new_list += {t – num} => new_list = {14 – 7} => new_list = {7}

now, processing target is 4,

checking the condition (t − num = 0 or t − num = sum_of_ rest) => (4 – 7 ==0 or 4 – 7 == 10) results negative,

checking the condition (t − num > 0 and t − num < sum_of_ rest) => (4 – 7 > 0 and 4 – 7 < 10) results negative, so no update to new_list here

after merging new_list with target_list, target_list = {14, 4, 7}

Step3: Take number 5

sum_of_rest = 10 – 5 = 5

new_list = {}

for each target in target_list {14, 4, 7}

processing target is 14,

checking the condition (t − num = 0 or t − num = sum_of_ rest) => (14 – 5 == 0 or 14 – 5 == 5) results negative

checking the condition (t − num > 0 and t − num < sum_of_ rest) => (14 – 5 > 0 and 14 – 5 < 5)

results negative, so no new_list is updated, ie new_list = {}

processing target is 4,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (4 – 5 == 0 or 4 – 5 == 5)
results negative,
checking the condition (t − num > 0 and t − num < sum_of_ rest) => (4 – 5 > 0 and 4 – 5 < 5)
results negative, again, no new_list is updated, ie new_list = {}

processing target is 7,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (7 – 5 == 0 or 7 – 5 == 5)
results negative,
checking the condition (t − num > 0 and t − num < sum_of_ rest) => (7 – 5 > 0 and 7 – 5 < 5)
results positive, so new_list = {7 – 5} => new_list = {2}

after merging new_list with target_list, target_list = {14, 4, 7, 2}

Step4: Take number 3
sum_of_rest = 5 – 3 = 2
new_list = {}

for each target in target_list {14, 4, 7, 2}
processing target is 14,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (14 – 3 == 0 or 14 – 3 == 2)
results negative
checking the condition (t − num > 0 and t − num < sum_of_ rest) => (14 – 3 > 0 and 14 – 3 < 2)
results negative, so no new_list is updated, ie new_list = {}

processing target is 4,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (4 – 3 == 0 or 4 – 3 == 2)
results negative
checking the condition (t − num > 0 and t − num < sum_of_ rest) => (4 – 3 > 0 and 4 – 3 < 2)
results positive, so new_list = {4 – 3} => new_list = {1}

processing target 7,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (7 – 3 == 0 or 7 – 3 == 2)
results negative
checking the condition (t − num > 0 and t − num < sum_of_ rest) => (7 – 3 > 0 and 7 – 3 < 2)
results negative, no new_list is updated, ie new_list = {1}
processing target is 2,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (2 – 3 == 0 or 2 – 3 == 2)
results negative,
checking the condition (t − num > 0 and t − num < sum_of_ rest) => (2 – 3 > 0 and 2 – 3 < 2)
results negative, no new_list is updated, ie new_list = {1}

after merging new_list with target_list, target_list = {14, 4, 7, 2, 1}

Step5: Take number 2
sum_of_rest = 2 – 2 = 0
new_list = {}

processing target is 14,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (14 – 2 == 0 or 14 – 2 == 0)
results negative,
checking the condition (t − num > 0 and t − num < sum_of_ rest) => (14 – 2 > 0 and 14 – 2 < 0)
results negative, so no new_list is updated, ie new_list = {}

processing target is 4,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (4 – 2 == 0 or 4 – 2 == 0)
results negative,
checking the condition (t − num > 0 and t − num < sum_of_ rest) => (4 – 2 > 0 and 4 – 2 < 0)
results negative, so no new_list is updated, ie new_list = {}

processing target is 7,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (7 – 2 == 0 or 7 – 2 == 0)
results negative,
checking the condition (t − num > 0 and t − num < sum_of_ rest) => (7 – 2 > 0 and 7 – 2 < 0)

results negative, so no new_list is updated, ie new_list = {}

processing target is 2,
checking the condition (t − num = 0 or t − num = sum_of_ rest) => (2 – 2 == 0 or 2 – 2 == 0)
results positive, solution found.

3.2.3 Dynamic Programming (DP)

DP is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions [11]. It is the technique related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. It is applicable when the sub-problems are not independent, that is, when sub-problems share sub-sub-problems. It solves every sub-subproblems just once and then saves the answers in a table, thereby avoiding the work of re-computing the answer every time the sub-subproblems are encountered [17].

Algorithm

procedure: Dynamic Programming
input: S-Set of numbers, target
output: true or false

1. if (target = 0 or target = Σ(S))
   - solution found
2. else if (target > Σ(S))
   - no solution found
3. else
   - if (target > Σ(S)/2)
     - target = Σ(S) − target
   - if (target ∈ S)
     - solution found
   - else
     - BitMap summap: summap[0] = 1
     - for each num in S from high to low
       - BitMap newmap = summap >> num
       - summap = summap or newmap

- if (summap[target] = 1)
    - solution found
- else
    - solution not found

4. Terminate

The symmetry heuristic (target = Σ(S) − target if (target > Σ(S)/2)) can result in significant savings of time and space, since the size of the sum map depends directly on the target t.

The implementation employs bitwise shift. The boolean array summap[], with index range from 0 to t, is initially set as summap[0] true and the rest of the array false[7]. Each number x in the set is considered and summap[x] is set true. Also, previously true index i of summap[i] plus x, ie summap[i + x], is also set to true. The solution to the SSP is present if summap[t] is true. The tabular work out for DP would be:

Table2: Tabular work out for Dynamic Programming

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
2	T	F	T	F	F	F	F	F	F	F	F	F	F	F	F
3	T	F	T	T	F	T	F	F	F	F	F	F	F	F	F
5	T	F	T	T	F	T	F	T	T	F	T	F	F	F	F
7	T	F	T	T	F	T	F	T	T	T	T	F	T	F	T
10															

3.2.4 DP Example Trace

Lets trace DP for SSP in which S = {2, 3, 5, 7, 10}, and t = 14.

Here, ΣS = 27.

The above algorithm's steps: 1 and 2 get false, and reach second 'else' part of step 3.

First, set summap array as summap[0] = 1

Then, for each number in S from high to low, ie {10, 7, 5, 3, 2}

Step1: Take number 10

new_map = sum_map[10] = 1

then, sum_map[0] = sum_map [10] = 1

as sum_map[14] == 1 is false,

Step2: Take number 7

new_map = sum_map[7] = 1

then, sum_map[0] = sum_map[7] = sum_map[10] = 1

the sum_map[17] would have value 1, but since sum_map array has max index = 14, sum_map[17] = 1 is rejected.

Step3: Take number 5

new_map = sum_map[5] = 1,

and, sum_map[0] = sum_map[5] = sum_map[7] = sum_map[12] = 1

Step4: Take number 3

new_map = sum_map[3] = 1,

sum_map[0] = sum_map[3] = sum_map[5] = sum_map[7] = sum_map[8] = sum_map[10] = sum_map[12] = 1

again, the sum_map[15] would have value 1, but since sum_map array has max index = 14, sum_map[15] = 1 is rejected.

Step5: Take number 2

new_map = sum_map[3] = 1,

sum_map[0] = sum_map[2] = sum_map[3] = sum_map[5] = sum_map[7] = sum_map[8] = sum_map[9] = sum_map[10] = sum_map[12] = sum_map[14] = 1

Since, sum_map[t], that is , sum_map[14] = 1, there is solution to this subset-sum problem.

3.2.5 Backtracking (BT)

BT is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem. It is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. In its basic form, backtracking resembles a depth-first search in a directed graph [15]. By inserting more knowledge of the problem, the search tree can be pruned to avoid considering cases that don't look promising. A node is said to be promising if

there is possibility of reaching to the solution from this node. A function that computes whether a node is promising or not can be implemented as below:

wtSoFar = weight of node, i.e., sum of numbers included in partial solution node represents
possibleLeft = weight of the remaining items i+1 to n (for a node at depth i)

```
boolean promising ( i )
{
return(wtSoFar + possibleLeft ≥ S)&&( wtSoFar == S || wtSoFar + w[i] ≤ S)
}
```

Algorithm

procedure: Backtracking
input: S-Set of numbers, target, sor- sum of rest
output: true or false

1. if (index < 0 ∨ target < 0 ∨ target > Σ(S))
   - no solution found
2. else
   - if (target > setsum/2)
     - target = Σ(S) - target
   - current_num = S[index]
   - increment index
   - if (sor = target ∨ current_num = target ∨ target = 0)
     -solution found
   - else if current_num < target
     - if promising
       - include current_num and check for the remaining numbers
       - sor = sor - current_num
       - target = target - current_num
       - Call BT recursively
     - else
       - exclude current_num and check for the remaining numbers
       - sor = sor - current_num

- Call BT recursively

- else
  - no solution found

3. Terminate

Check if (target > Σ(S)/2) to implement a symmetry-motivated heuristic that looks for the smaller of targets t and Σ(S) − t. The rationale is that a smaller target should have a shallower depth of recursion, but choosing a smaller target may also minimize the benefit bounding condition in line 1.

Backtracking has a simple recursive formulation, and with the proper bounding conditions. The logic is simply to branch on the numbers in the set S. For any element y of S, if there is a subset S with sum t, it either contains y or it doesn't. The element y is checked through the promising function. If it seems promising then put y in the subset to get by recursive call on S − {y} and t − y. Otherwise, skip y and find S by recursive call on S − {y} and t.

3.2.6 BT Example Trace

Lets trace BT for SSP in which S = {2, 3, 5, 7, 10}, and t = 14.

Here, ΣS = 27.

The set S with its element in non-increasing order, S = {10, 7, 5, 3, 2}

The tracing can be shown by the following binary tree

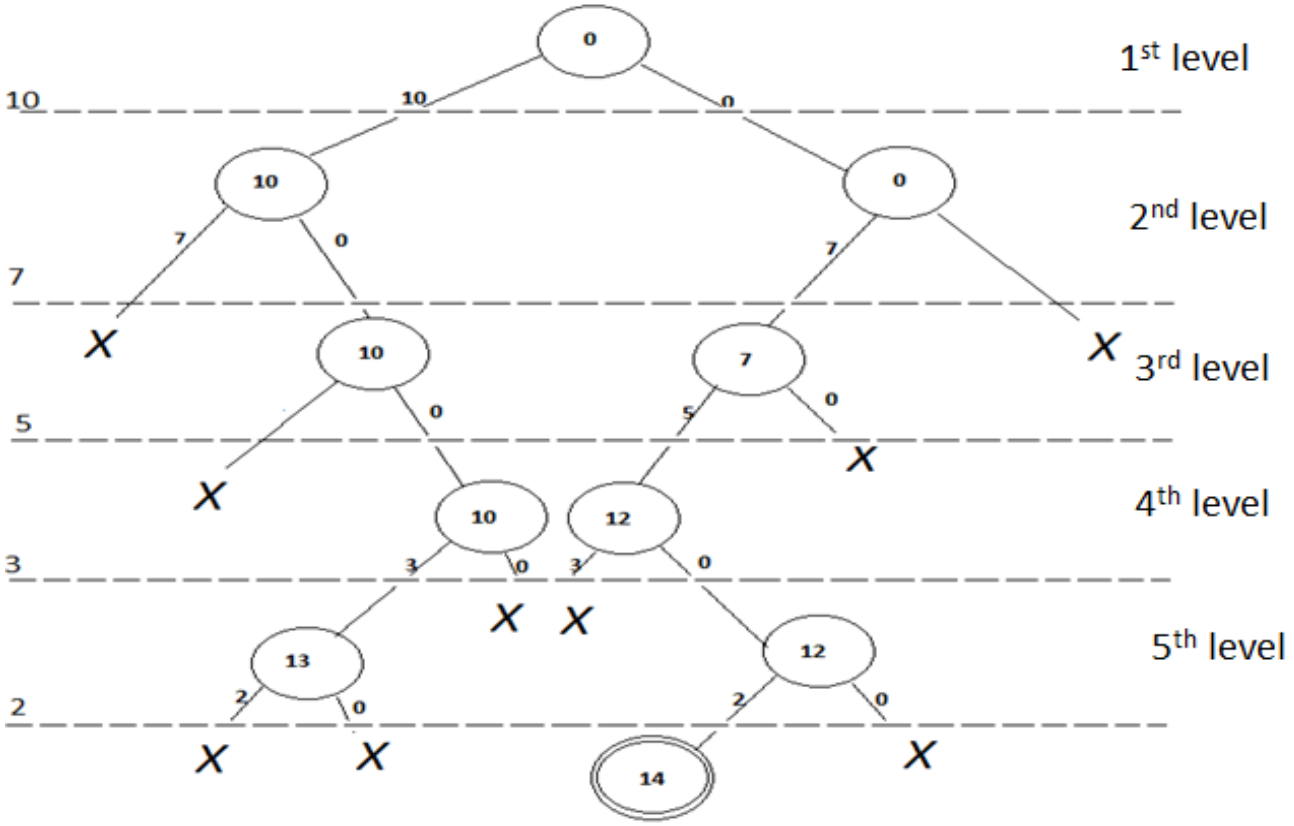

Figure 3: Tracing Backtracking Algorithm

Step1:

Starting from the root node, the first processing element is 10, the promising condition is satisfied so 10 is included and proceed in left branch to 2nd level 10-node.

Step2:

The processing number is 7, since the promising condition is violated, the number 7 is excluded and no branch could be extended. This pruned state is represented by X. The branch is right extended excluding the number 7.

Step3:

The processing number is 5, on 3rd level 10-node. The promising condition is again violated including 5, so proceed to 4th level 10-node excluding number 5.

Step4:

The processing number is 3. The number is included as the promising condition is satisfied and reached

5^{th} level 13-node.

Step5:

The processing number is 2, the 5^{th} level 13-node cannot meet the target including or excluding the number 2.

So it back tracks to 4^{th} level 10-node. The promising condition is not satisfied excluding the leveled processing number 3. Again do backtrack to 3^{rd} level 10-node and to 2^{nd} level 10-node, and finally to the root node.

Step6:

The tracing begins excluding the number 10, the branching starts from 2^{nd} level 0-node. Here the processing number is 7. As the promising condition gets satisfied, proceed to 3^{rd} level 7-node.

Step7:

The processing number is 5. The promising condition is satisfied and proceed to 4^{th} level 12-node. The promising condition is not satisfied for the next processing number 3. So, it proceeds excluding 3 to reach 5^{th} leveled 12-node.

Step8:

The processing number is 2 and the target is met including number 2 from 5^{th} level 12-node. That is, 12 + 2 = 14.

All the X-marked nodes are the pruned nodes. From these pruned nodes the target could not be met.

Chapter 4: Data Collection & Analysis

4.1 Test Case Design

For the test case, a set of 15 random numbers and the maximum number (m) in the set was set to 1024. All the three algorithms were tested on that set and noted the step-counts for each algorithm. This is an instance of the test case design. 50 similar instances were generated and tested. Finally, the average step-count of those 50 instances is calculated and a graph of 'm' versus 'average step-count' is plotted.

4.2 Data Collection

The required data for the experimental analysis is generated. The sets of size (n) from 15 to 25 and bit length (m) from 10 to 20 are generated. For each size and bit length, 50 set instances have been generated. The step-counts for 50 instances of the specific value of n and m are averaged.
For example, 50 instances of the set for n = 15 and m = 10 are generated as:

4.2.1 For the Dataset with n = 15 and m = 10

[1024, 327, 78, 360, 469, 666, 83, 717, 224, 189, 788, 671, 790, 269, 399],
[569, 177, 591, 146, 319, 702, 596, 780, 348, 247, 133, 259, 1024, 489, 335],
[562, 169, 247, 614, 193, 399, 397, 579, 790, 1024, 420, 91, 435, 535, 367],
[58, 656, 124, 324, 588, 1024, 278, 873, 493, 307, 969, 121, 740, 9, 55],
[655, 612, 74, 827, 80, 337, 1024, 713, 544, 367, 49, 13, 97, 395, 556],
[1024, 345, 783, 1, 111, 728, 360, 302, 569, 642, 920, 591, 431, 449, 222],
[1024, 677, 505, 741, 630, 424, 607, 288, 452, 264, 589, 701, 208, 765, 368],
[124, 888, 951, 1024, 841, 406, 444, 949, 930, 101, 721, 835, 926, 376, 188],
[350, 518, 197, 283, 640, 340, 514, 59, 326, 1024, 51, 311, 261, 554, 437],
[744, 645, 377, 820, 322, 414, 129, 406, 598, 1024, 790, 764, 753, 211, 539],
[349, 683, 771, 803, 628, 455, 552, 35, 413, 405, 712, 377, 423, 1024, 167],
[509, 207, 237, 154, 212, 320, 194, 770, 1024, 477, 93, 512, 395, 601, 912],
[768, 1024, 285, 621, 296, 669, 342, 376, 645, 951, 926, 340, 86, 289, 145],
[644, 113, 950, 14, 263, 216, 280, 471, 108, 1024, 554, 312, 642, 834, 529],
[230, 138, 137, 79, 1024, 858, 579, 642, 385, 673, 891, 664, 32, 200, 668],
[826, 887, 729, 342, 621, 612, 367, 902, 549, 201, 299, 326, 80, 586, 1024],

[360, 56, 521, 153, 933, 1024, 46, 52, 552, 723, 209, 692, 728, 362, 617],
[856, 611, 770, 509, 939, 321, 804, 577, 363, 525, 314, 701, 384, 1024, 383],
[97, 843, 445, 776, 745, 799, 421, 552, 999, 75, 945, 398, 268, 1024, 301],
[959, 633, 803, 1024, 880, 917, 13, 666, 867, 331, 232, 700, 981, 909, 765],
[618, 362, 66, 441, 1024, 913, 67, 661, 505, 546, 863, 70, 583, 473, 329],
[477, 939, 392, 1024, 630, 72, 102, 23, 346, 141, 622, 768, 607, 525, 938],
[246, 324, 319, 1024, 408, 360, 901, 78, 399, 118, 917, 542, 312, 845, 684],
[776, 826, 178, 540, 729, 792, 616, 248, 154, 836, 1024, 415, 853, 760, 576],
[1024, 663, 66, 460, 775, 152, 405, 79, 31, 543, 838, 794, 805, 669, 457],
[558, 726, 832, 61, 728, 696, 83, 441, 866, 836, 611, 392, 417, 1024, 173],
[390, 576, 117, 816, 852, 893, 866, 6, 377, 120, 176, 1024, 633, 157, 841],
[727, 502, 348, 230, 1024, 15, 641, 86, 902, 874, 685, 458, 943, 506, 811],
[945, 1024, 711, 591, 16, 677, 701, 535, 2, 807, 772, 416, 744, 723, 933],
[767, 644, 237, 974, 710, 997, 303, 1024, 621, 230, 232, 928, 560, 835, 48],
[237, 345, 167, 137, 454, 166, 930, 250, 899, 445, 924, 1024, 337, 529, 328],
[8, 317, 574, 88, 412, 105, 468, 209, 3, 459, 1024, 314, 755, 567, 216],
[169, 693, 170, 883, 471, 565, 635, 828, 345, 43, 736, 393, 1024, 144, 547],
[903, 945, 366, 229, 769, 805, 926, 95, 978, 373, 625, 1024, 323, 142, 583],
[295, 751, 923, 246, 1024, 967, 79, 284, 26, 758, 784, 977, 369, 875, 39],
[534, 43, 191, 560, 74, 856, 802, 1024, 503, 803, 50, 278, 185, 398, 5],
[924, 457, 262, 909, 872, 161, 163, 563, 146, 248, 1024, 331, 791, 122, 499],
[851, 796, 575, 703, 130, 459, 46, 1000, 670, 184, 974, 505, 515, 714, 1024],
[287, 326, 130, 166, 760, 303, 732, 1024, 331, 742, 675, 31, 106, 148, 273],
[9, 108, 433, 536, 20, 344, 844, 667, 328, 597, 791, 704, 1024, 44, 766],
[115, 640, 647, 242, 354, 887, 581, 953, 544, 87, 111, 1024, 947, 126, 347],
[373, 705, 656, 1024, 382, 410, 585, 70, 87, 239, 447, 483, 619, 223, 733],
[170, 228, 553, 376, 1024, 756, 698, 514, 517, 845, 247, 613, 32, 234, 308],
[147, 7, 1024, 883, 556, 746, 553, 418, 850, 605, 232, 109, 685, 72, 434],
[954, 386, 152, 667, 409, 319, 292, 736, 30, 469, 309, 1024, 212, 660, 226],
[781, 778, 207, 639, 269, 916, 475, 534, 666, 297, 677, 485, 1024, 433, 471],
[684, 853, 244, 1024, 859, 85, 182, 80, 505, 382, 275, 319, 812, 679, 493],
[7, 829, 173, 275, 358, 456, 270, 31, 1024, 219, 645, 716, 383, 877, 597],
[360, 616, 532, 316, 102, 483, 371, 192, 172, 294, 282, 1024, 677, 332, 536],

[691, 622, 1024, 508, 642, 525, 308, 669, 605, 592, 437, 95, 670, 480, 68]

50 individual set for each instance when m varies from 10 to 20 and n varies from 15 to 25 is generated. Altogether, 11 x 11 x 50 = 6050 sets have been generated. The algorithms: BT, DP and DDP are implemented for each instance set. The step-count as an output of implemented algorithm is collected in a column of an excel file. The average of 50 step-counts with fixed value of n and m is calculated as shown in below:

Table 3: Average step-count for n = 15, m = 10..11

N=15,	**M = 10 (2^{10} = 1024)**			**M = 11 (2^{11} = 2048)**		
	BT	**DP**	**DDP**	**BT**	**DP**	**DDP**
Instance 1	34114	23344	333	33441	24479	528
Instance 2	34727	23548	351	33623	26787	198
Instance 3	34783	23209	1472	33770	27721	114
Instance 4	33309	21663	714	35138	31935	726
Instance 5	33313	20843	283	n	n	n
Instance 6	33841	22127	199	34282	26422	614
Instance 7	35537	24624	69	33495	27069	652
Instance 8	34197	26336	1036	33550	26298	667
Instance 9	34118	21948	606	33617	33164	650
Instance 10	34904	23363	1096	35516	28998	134
Instance 11	34444	23667	409	34508	28973	122
Instance 12	34465	24331	348	33782	29294	639
Instance 13	34299	23391	848	34646	32540	1704
Instance 14	33726	20864	700	34371	26777	334
Instance 15	33674	21797	441	33900	27837	566
Instance 16	34550	24101	686	34330	27389	785
Instance 17	33583	21303	618	33976	28039	1006
Instance 18	36114	25780	428	34189	28726	833
Instance 19	34163	21729	893	34222	29785	69
Instance 20	34462	24122	238	34565	26299	722
Instance 21	33926	21395	384	33592	30551	423
Instance 22	33544	22216	717	35310	27041	360
Instance 23	34290	21974	576	34217	27250	911

Instance 24	34806	26259	40	33386	25949	1090
Instance 25	33591	23406	1515	34078	26196	1157
Instance 26	34017	21498	600	34047	24864	541
Instance 27	33546	22630	226	33193	31406	1190
Instance 28	33815	24895	706	35197	24448	445
Instance 29	33946	24892	999	34628	27057	674
Instance 30	n	n	n	33496	25894	331
Instance 31	34230	21873	727	33680	24688	200
Instance 32	34619	19425	268	34543	26942	739
Instance 33	33228	19982	857	33919	31858	1303
Instance 34	34018	23941	826	34550	27228	688
Instance 35	34252	22875	901	34005	28460	1430
Instance 36	33474	23749	486	35987	30578	1430
Instance 37	33222	21402	517	34203	26493	577
Instance 38	34351	23736	1093	34163	29277	433
Instance 39	34113	25124	252	34968	30053	433
Instance 40	33769	19225	467	33889	30007	611
Instance 41	33254	22550	114	33791	29629	1245
Instance 42	34280	21975	1278	34152	29986	1459
Instance 43	34207	24140	894	33736	28492	915
Instance 44	33465	20806	440	34342	23931	710
Instance 45	34100	22641	335	34096	25767	605
Instance 46	35520	25955	1882	33643	30166	717
Instance 47	34109	21111	790	33500	31887	657
Instance 48	33520	21842	292	33413	26942	657
Instance 49	34794	23368	1033	35460	27041	1806
Instance 50	34412	24327	1145	35184	28998	674
AVERAGE	**34137.57**	**22883.71**	**655.6735**	**34189.57**	**28114.51**	**723.9592**

Note: n denotes the solution for the instance is not found

4.3 Analysis

The integer value of average step-count is calculated and plotted the integer value of step-count versus the bit-length (m).

Case1: For n = 15

Table 4: Step-count for n = 15, m = 10..20

M	Step-counts BT	Step-counts DP	Step-counts DDP
10	**34137**	**22883**	**655**
11	**34189**	**28114**	**723**
12	34140	31998	1292
13	32945	34860	1226
14	33922	35206	1225
15	34149	36745	1282
16	34339	38105	1521
17	34103	37656	1327
18	33696	36815	792
19	34030	37468	1730
20	34994	40811	3098

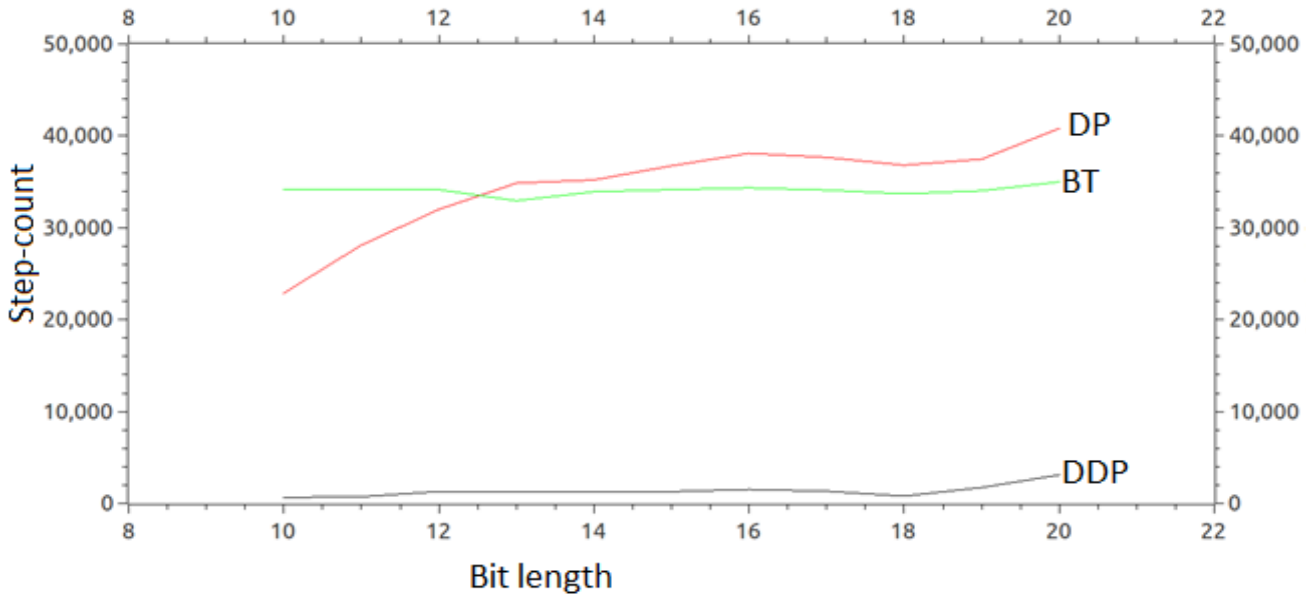

Figure 4: Step-counts for n = 15, m = 10..20

Case2: For n = 16

Table 5: Step-counts for n = 16, m = 10..20

M	Step-counts BT	Step-counts DP	Step-counts DDP
10	68186	36191	889
11	68366	46469	1292
12	67871	55474	1711
13	68194	63817	2448
14	68140	68721	2746
15	67809	71315	3224
16	67501	74238	3579
17	68736	79832	3875
18	68597	82638	4156
19	67158	85080	4364
20	67880	88028	4551

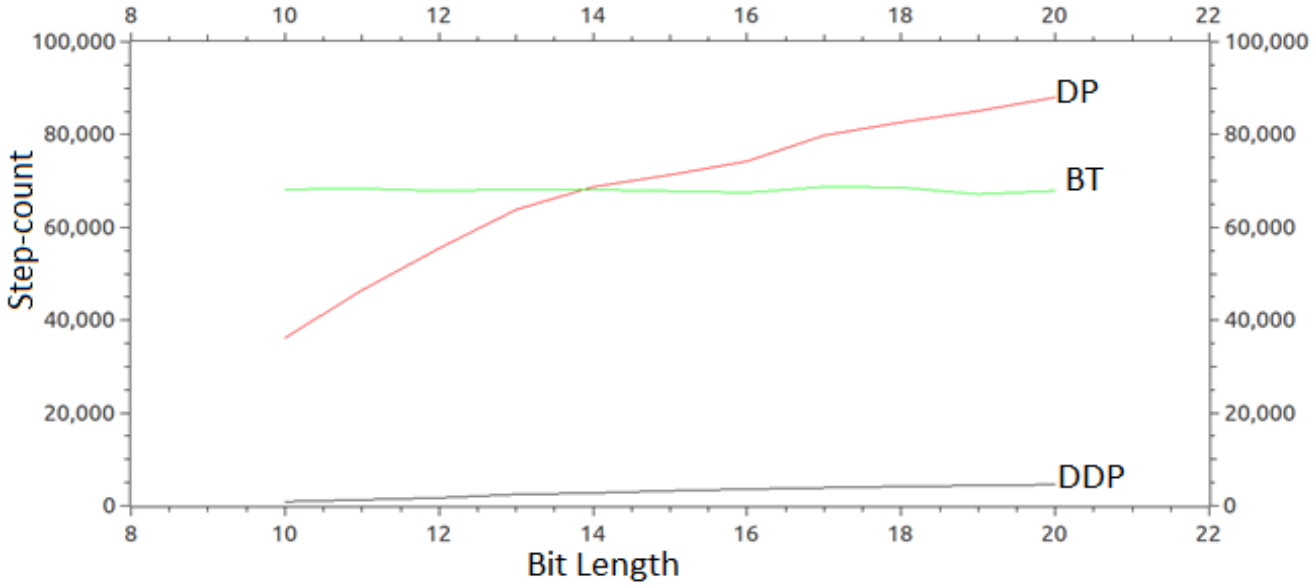

Figure 5: Step-counts for n = 16, m = 10..20

Case3: For n = 17

Table 6: Step-counts for n = 17, m = 10..20

m	Step-counts BT	Step-counts DP	Step-counts DDP
10	135530	51893	953
11	135948	72117	1135
12	135698	91609	2064
13	135344	110537	3117
14	135907	125746	4162
15	132621	138349	6189
16	135100	140441	7384
17	135015	155551	7978
18	135823	147399	8266
19	135631	171309	8688
20	136790	151689	8830

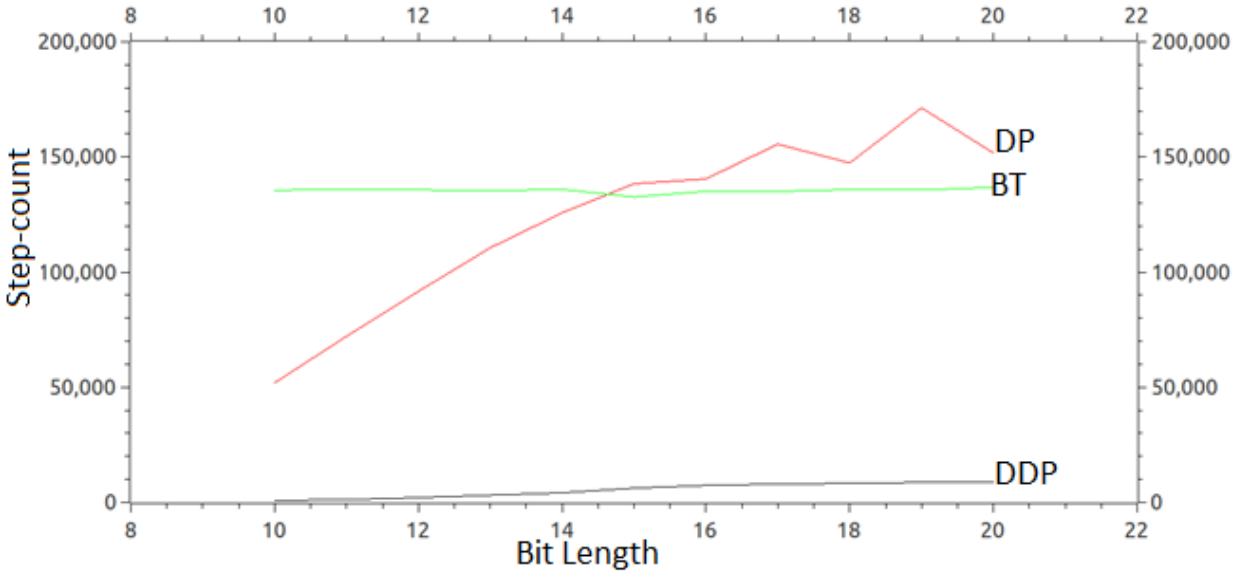

Figure 6: Step-counts for n = 17, m = 10..20

Case4: For n = 18

Table 7: Step-counts for n = 18, m = 10..20

m	Step-counts BT	Step-counts DP	Step-counts DDP
10	270000	74239	870
11	270827	106037	1658
12	270388	142246	2437
13	270917	185609	4576
14	271570	224010	6295
15	270437	247523	7153
16	270204	270017	9295
17	271295	284253	13008
18	273189	299123	14040
19	270008	310605	14938
20	277357	312974	14859

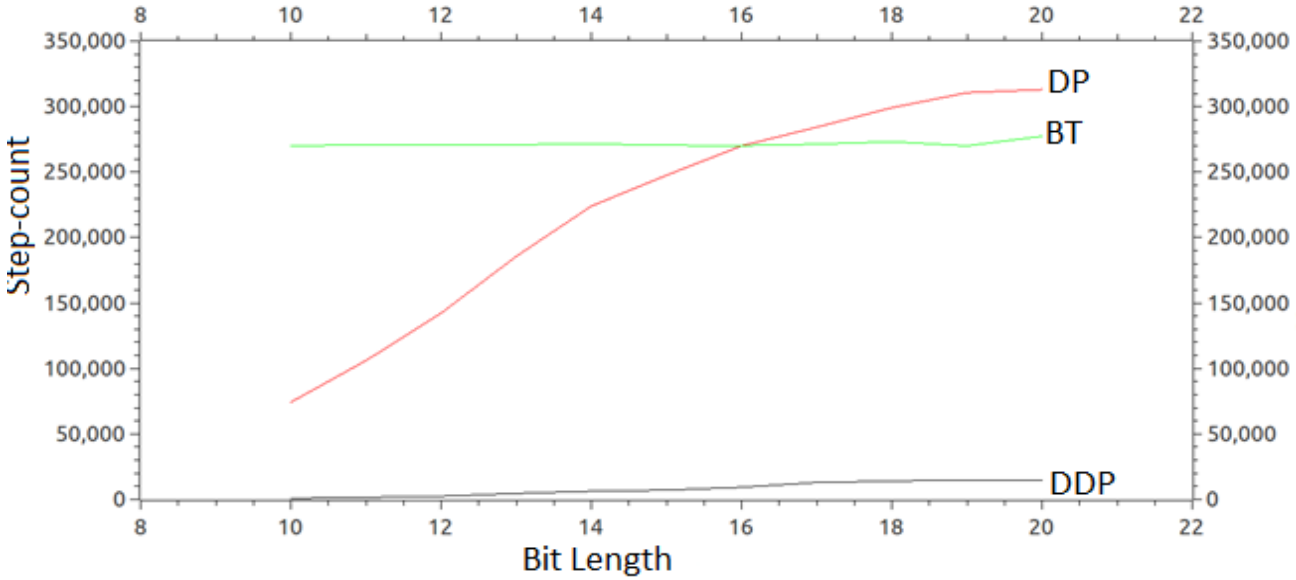

Figure 7: Step-counts for n = 18, m = 10..20

Case5: For n = 19

Table 8: Step-counts for n = 19, m = 10..20

m	Step-counts BT	Step-counts DP	Step-counts DDP
10	540354	107493	1318
11	539243	151401	2027
12	539836	212913	3773
13	540370	284038	6116
14	540289	366225	8341
15	538544	433597	11570
16	531016	502671	12624
17	541319	541084	13862
18	537815	549261	14140
19	535592	557941	17510
20	533607	562729	18905

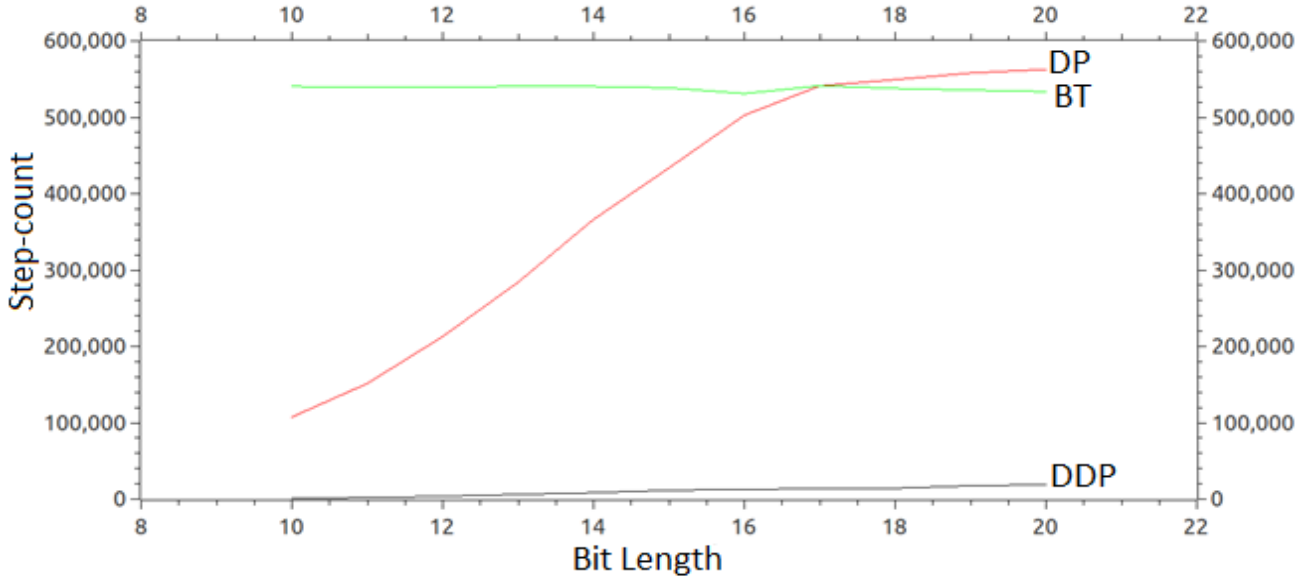

Figure 8: Step-counts for n = 19, m = 10..20

Case6: For n = 20

Table 9: Step-counts for n = 20, m = 10..20

m	Step-counts BT	Step-counts DP	Step-counts DDP
10	1077975	139991	1502
11	1076292	203791	2158
12	1078141	305599	3812
13	1079323	425135	5730
14	1077242	570356	8813
15	1102725	749590	16893
16	1077795	875468	24387
17	1077487	988457	28311
18	1075648	1058718	29947
19	1077022	1101863	31737
20	1073837	1126061	33037

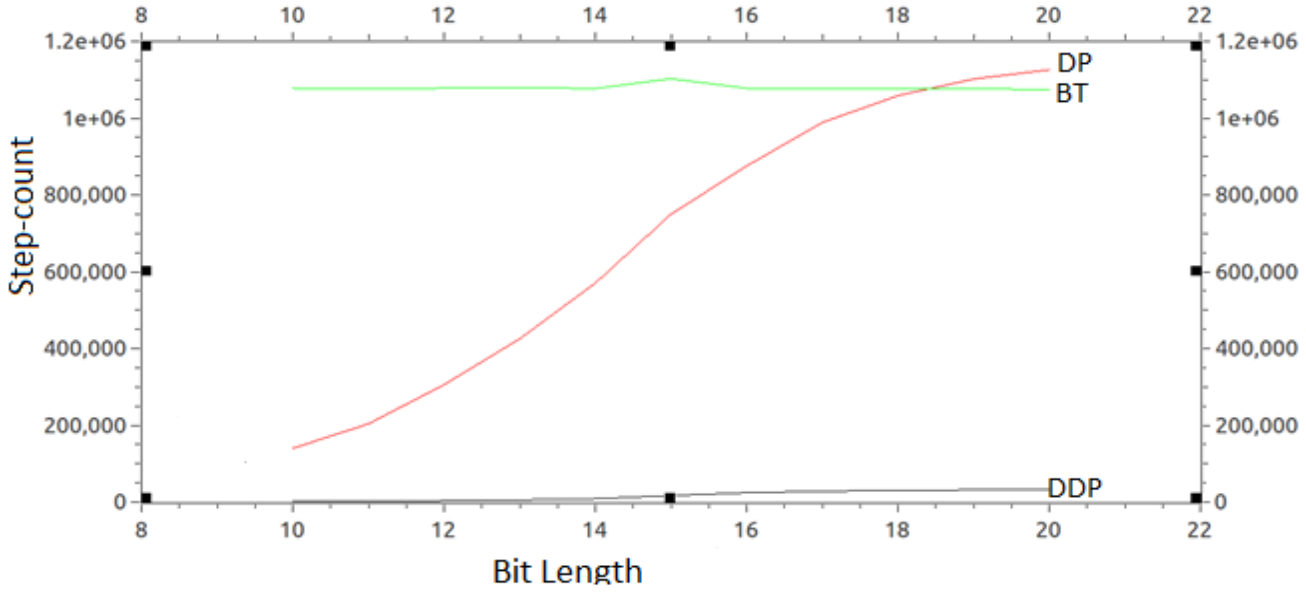

Figure 9: Step-counts for n = 20, m = 10..20

Case7: For n = 21

Table 10: Step-counts for n = 21, m = 10..20

m	Step-counts BT	Step-counts DP	Step-counts DDP
10	2154987	180800	1297
11	2146534	282771	2977
12	2159079	416478	3887
13	2153435	604629	7241
14	2151755	846146	12361
15	2158925	1144460	20567
16	2154908	1469388	25624
17	2153033	1741033	37716
18	2149218	1973957	47433
19	2206845	2171567	57991
20	2147656	2232050	65733

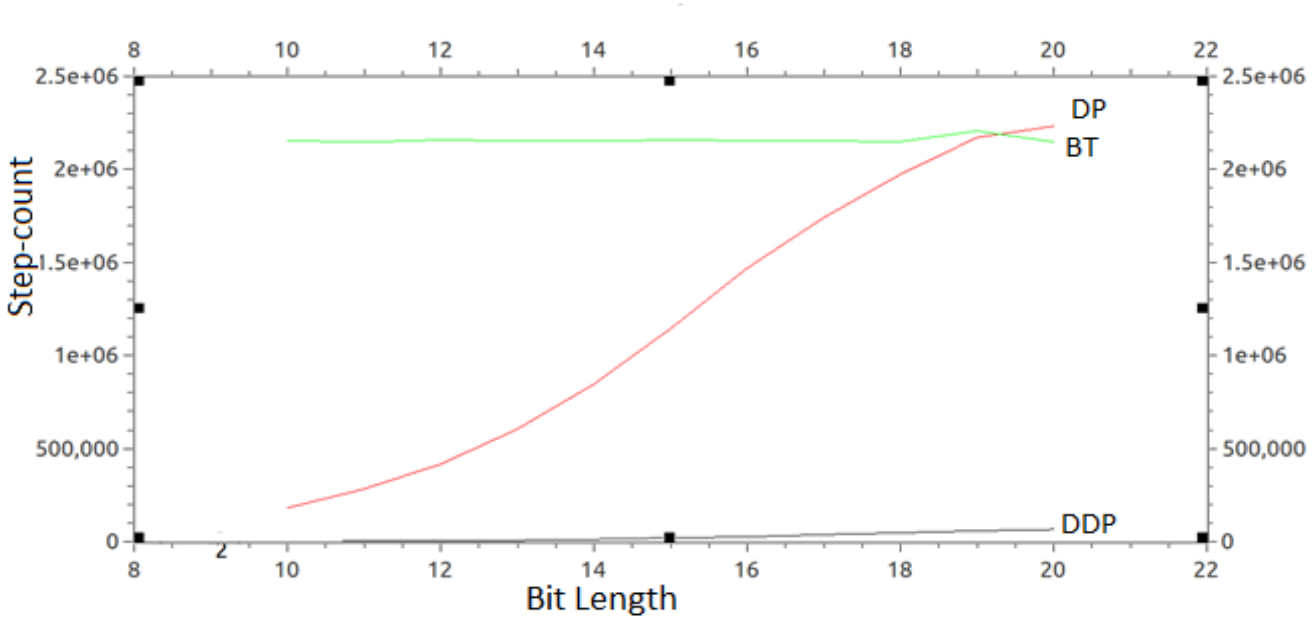

Figure 10: Step-counts for n = 21, m = 10..20

Case8: For n = 22

Table 11: Step-counts for n = 22, m = 10..20

m	Step-counts BT	Step-counts DP	Step-counts DDP
10	4287897	230208	1650
11	4306153	360472	2565
12	4291283	566394	4979
13	4300560	846314	9110
14	4298776	1216764	11402
15	4291727	1696116	21751
16	4295210	2268459	35044
17	4316306	2342682	55482
18	4301736	3523562	70042
19	4290882	3858119	77930
20	4286604	4181512	81249

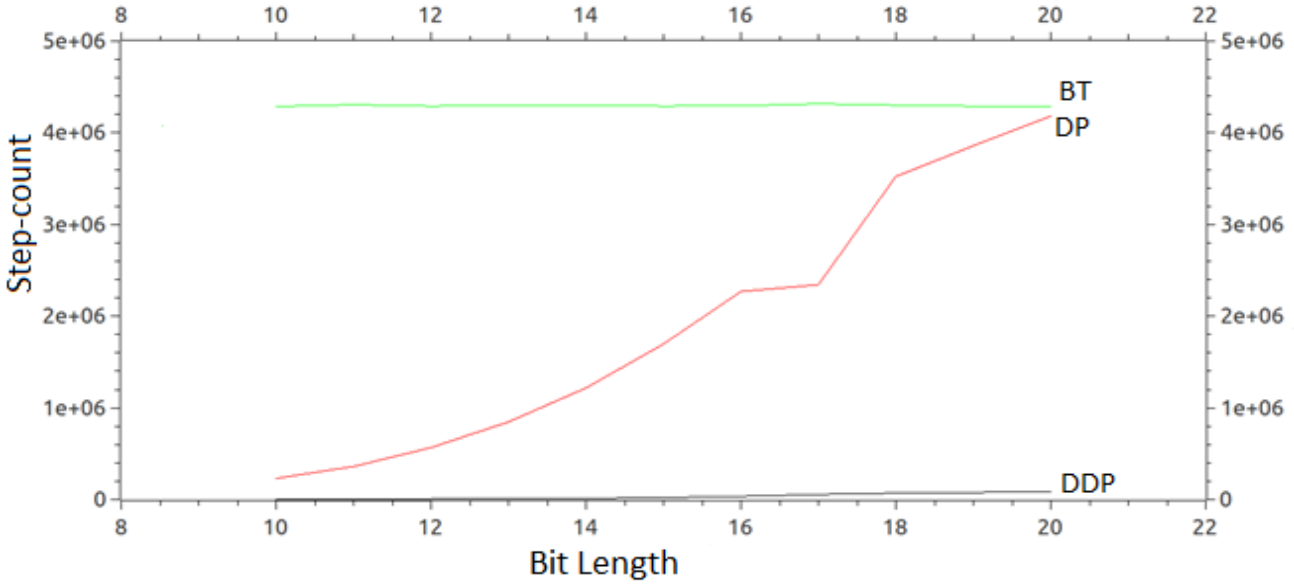

Figure 11: Step-counts for n = 22, m = 10..20

Case9: For n = 23

Table 12: Step-counts for n = 23, m = 10..20

m	Step-counts BT	Step-counts DP	Step-counts DDP
10	8606483	292304	2093
11	8584878	468073	3599
12	8602117	732351	6747
13	8578296	1111385	9722
14	8553415	1677176	14574
15	8575921	2410639	34151
16	8583548	3422278	50751
17	8561775	4532963	62927
18	8570257	5754037	71609
19	8573390	6239741	83186
20	8610575	8573390	102355

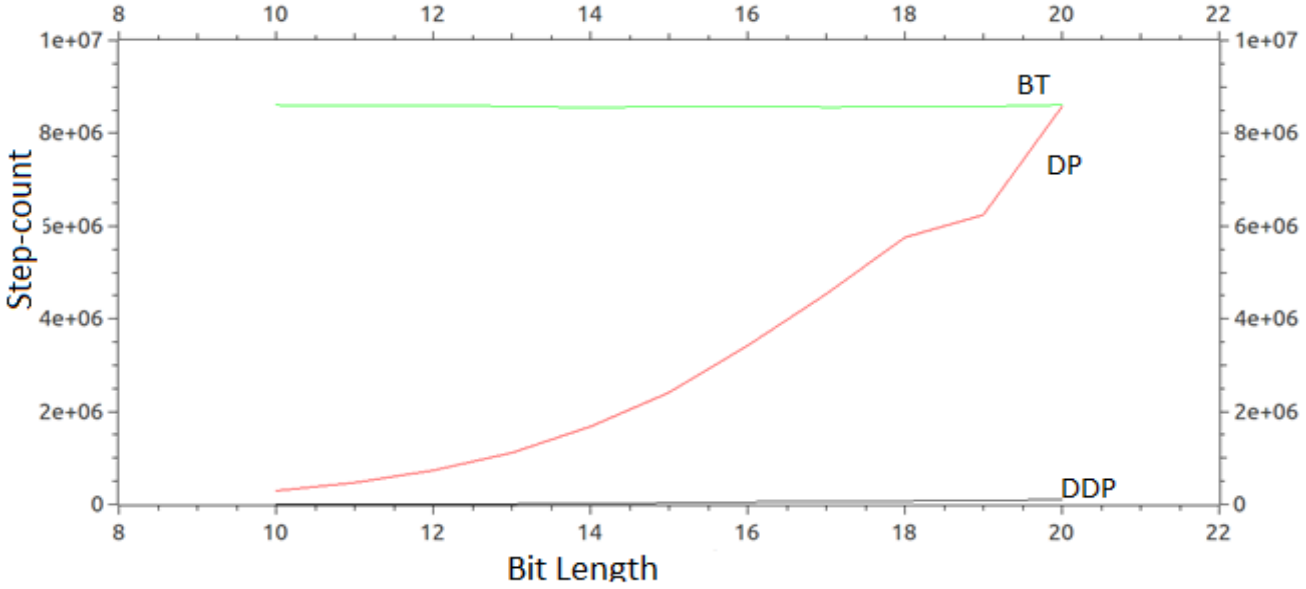

Figure 12: Step-counts for n = 23, m = 10..20

Case10: For n = 24

Table 13: Step-counts for n = 24, m = 10..20

m	Step-counts BT	Step-counts DP	Step-counts DDP
10	17159120	357947	2738
11	17158788	579034	3732
12	17149302	938669	5204
13	17110091	1449497	11909
14	17106312	2241199	16907
15	17158586	3376867	27108
16	17184825	4982786	54467
17	17159312	6943242	64501
18	17146444	9279666	79906
19	17091372	11470264	91597
20	17126665	13887013	112573

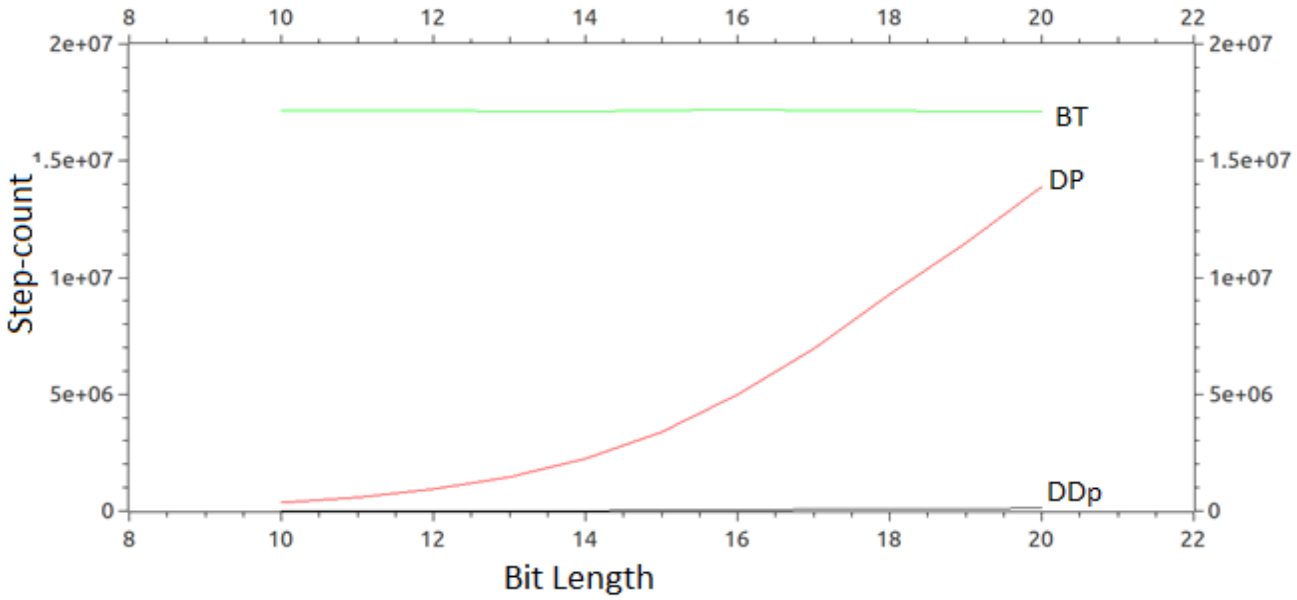

Figure 13: Step-counts for n = 24, m = 10..20

Case11: For n = 25

Table 14: Step-counts for n = 25, m = 10..20

m	Step-counts BT	Step-counts DP	Step-counts DDP
10	34222042	440450	2780
11	34185432	707408	4443
12	34320384	1167212	7554
13	34215419	1882590	13221
14	34094609	2950559	21428
15	34285386	4576428	36509
16	34219673	6782869	56570
17	34337625	9973402	78239
18	34367879	13983540	92775
19	34164170	18356114	106693
20	34139709	22964581	123384

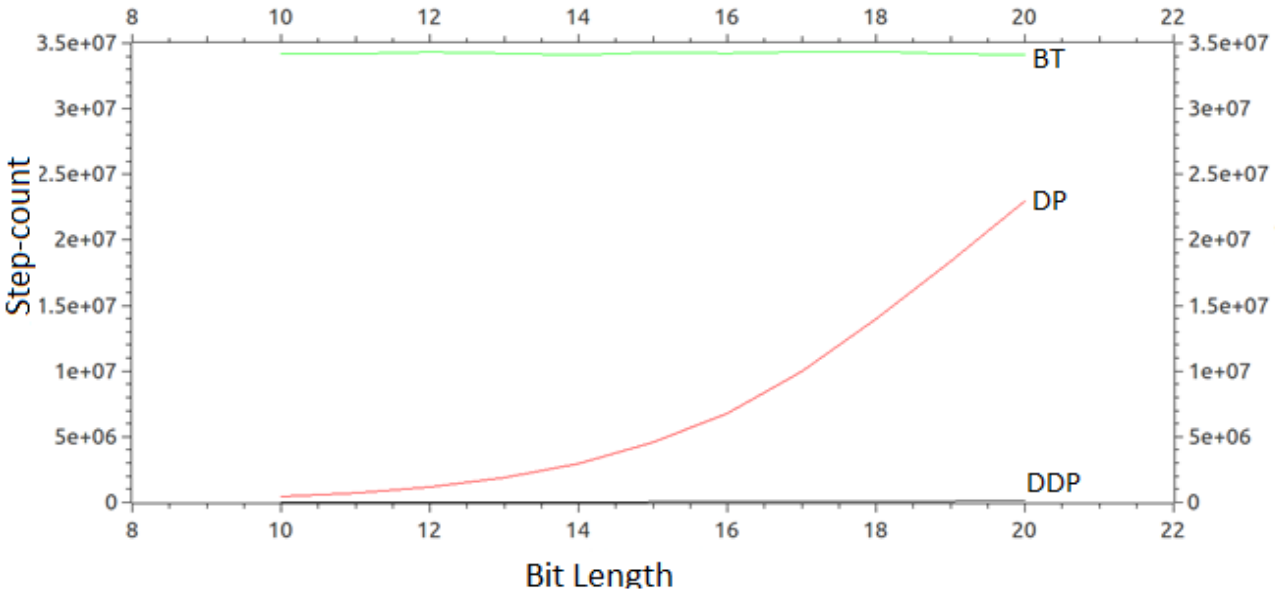

Figure 14: Step-counts for n = 25, m = 10..20

The above tabulations and graph plots show that the algorithms: DP and DDP have sub-exponential

complexity when the complexity parameter is the total bit-length m. The average sub-exponential growth rate in each case for DP and DDP can be tabulated as:

Table 15: DDP and DP growth rate when complexity parameter is m

Case	Sub-exponential growth rate(DP)	Sub-exponential growth rate(DDP)
1	1.13	1.28
2	1.13	1.23
3	1.22	1.31
4	1.21	1.45
5	1.26	1.4
6	1.3	1.45
7	1.36	1.48
8	1.42	1.56
9	1.49	1.49
10	1.53	1.35
11	1.57	1.56

BT seems steady or does not possess sub-exponential complexity when complexity parameter is total bit-length (m). That is, the response is invariant for m parameter. But, it has the exponential complexity when the decision parameter is the input size (n).

Analysis of BT in terms of number of elements, n

Table 16: BT analysis in terms of n

n\m	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
10	1087	1095	1089	1100	1048	1123	N	N	N	N	N
11	2149	2182	2174	N	N	2164	2159	N	N	N	N
12	4177	4314	4361	4328	4271	4226	N	N	N	4301	N
13	8638	8495	8528	8527	8659	8615	N	8345	N	N	N
14	17192	17122	17084	17082	17130	17102	17255	16769	N	N	N
15	34137	34189	34140	32945	33922	34149	34339	34103	33696	34030	34994

16	68186	68366	67871	68194	68140	67809	67501	68736	68597	67158	67880
17	135530	135948	135698	135344	135907	132621	135100	135015	135823	135631	136790
18	270000	270827	270388	270917	271570	270437	270204	271295	273189	270008	277357
19	540354	539243	539836	540370	540289	538544	531016	541319	537815	535592	533607
20	1077975	1076292	1078141	1079323	1077242	1102725	1077795	1077487	1075648	1077022	1073837
21	2154987	2146534	2159079	2153435	2151755	2158925	2154908	2153033	2149218	2206845	2147656
22	4287897	4306153	4291283	4300560	4298776	4291727	4295210	4316306	4301736	4290882	4286604
23	8606483	8584878	8602117	8578296	8553415	8575921	8583548	8561775	8570257	8573390	8610575
24	17159120	17158788	17149302	17110091	17106312	17158586	17184825	17159312	17146444	17091372	17126665
25	34222042	34185432	34320384	34215419	34094609	34285386	34219673	34337625	34367879	34164170	34139709

Note: N denotes the solution for the instance is not found

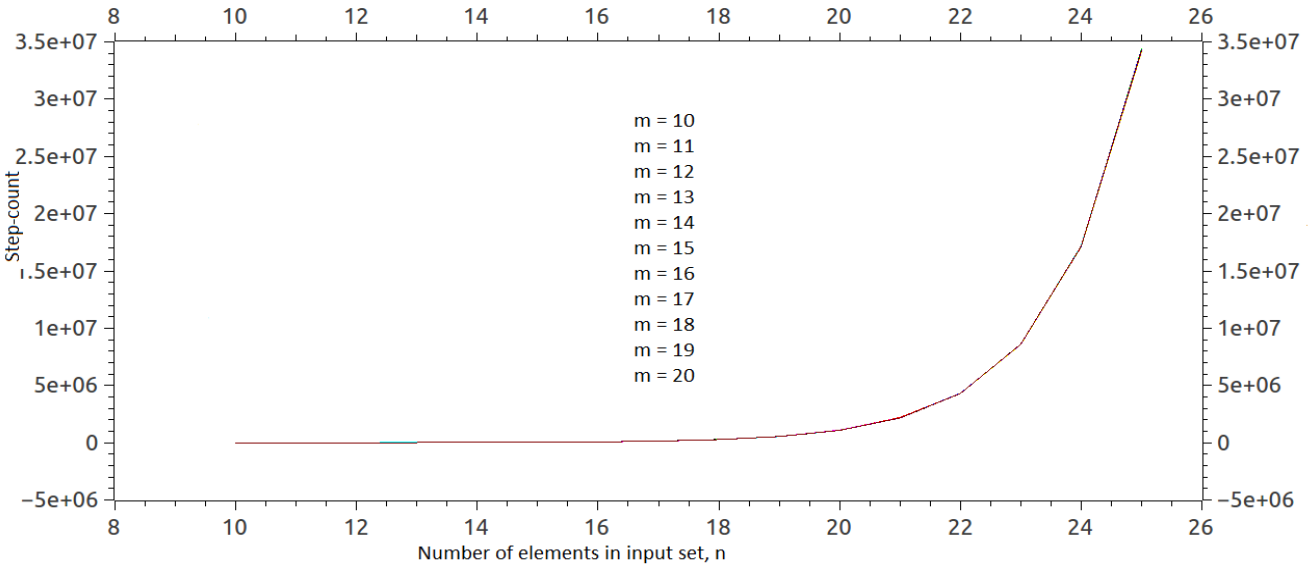

Figure 15: BT analysis in terms of n

This graph clearly shows that BT is not sensitive to the total bit length, m, but its step count grow exponentially when input parameter is the input size, n.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

The SSP is a well-known NP-Complete problem. The algorithms for solving SSP: DDP, DP, and BT have been proven to have exponential complexity when the decision parameter is the total number elements in the set, n. At the same time DDP have sub-exponential time complexity when it is evaluated in terms of total bit length, m. But Statistics of DP and BT was unknown in this regard. This dissertation has answered this question.

This dissertation work shows that time complexity of DP increases by 1.13 to 1.57 times when bit length, m, is increased by 1. This is clearly sub-exponential increment. At the same time BT is not sensitive to m and its time complexity increases by 1.98 to 2.01 times when number of inputs, n, is increased by 1.

From this observation, we can conclude the DP also have sub-exponential time complexity when input parameter is bit length, and it has no effect in BT algorithm for solving Subset Sum Problem (SSP).

5.2 Limitations and Future Work

The experiments were conducted on the limited set of numbers; the number of set (n) was varied from 15 to 25, total bit-length required to represent the set (m) was varied from 10 to 20, and the target was set only to half of the total-sum of the numbers in the set, $t = \Sigma S/2$. Further, a lot more experiments could be done by expanding the ranges of n, m and t.

References:

[1] L. Escudero, S. Martello, and P. Toth, A framework for tightening 0-1 programs based on extensions of pure 0-1 KP and SS problems. Lecture Notes in Computer Science, 920:110–123, 1995.

[2] C. Gu´eret and C. Prins. A new lower bound for the open-shop problem. Annals of Operations Research, 92:165–183, 1999.

[3] Thomas. E. O'Neil and Scott Kerlin, A Simple $2^{O(x)}$ Algorithm for Partition and Subset Sum, Proceedings of the 2010 International Conference on Foundations of Computer Science, 2010.

[4] Adarsh Kumar Verma, 'Ads' Algorithm for SSP, Student,Galgotias College Of Engineering and Technology,Greater Noida, G. B. Nagar, India 2013.

[5] Prof. Dr. Gautam Das, Lecturer. Saravanan, Advanced Computational Models and Algorithms, Lecture Notes For Subset Sum. Jan 28, 2010.

[6] Eric Allender, Michael C. Loui, Kenneth W. Regan, Complexity Classes, Algorithms and Theory of Computation, Rutgers University, University of Illinois at Urbana-Champaign, State University of New York at Buffalo, 1999.

[7] Thomas E. O'Neil, An Empirical Study of Algorithms for the Subset Sum Problem, Computer Science Department, University of North Dakota, 2013.

[8] D. Pisinger. An exact algorithm for large multiple knapsack problems. European Journal of Operational Research, 114:528–541, 1999.

[9] S. Martello and P. Toth, Approximation schemes for the subset-sum problem: Survey and experimental analysis, European Journal of Operational Research, 22,56-69, 1985.

[10] B. Dietrich and L. Escudero. Coefficient reduction for knapsack constraints in 0-1 programs with VUBs. Operations Research Letters, 9:9–14, 1990.

[11] Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, Computer Algorithms/C++, Second Edition, Universities Press, 2007.

[12] M. R. Garey& D. S. Johnson, Computers and Intractibility: A Guide to the theory of NP Completeness, W. H. Freeman and Company, New York, 1979.

[13] O. H. Ibarra and C. E. Kim, Fast approximation algorithms for knapsack and sum of subset problem, journal of the ACM, 1975.

[14] E. L. Lawler, Fast approximation algorithms for Knapsack problems, Mathematics of operation research, 4,339-356, 1979

[15] Gilles Brassard, Paul Bratley, Fundamentals of Algorithms, #p 306, Prentice Hall Englewood Cliffs, New Jersey 07632, 1996.

[16] Harsh Bhasin and NehaSingla, "Harnessing Cellular Automata and Genetic Algorithms to solve Travelling Salesman Problem", Conference: ICICT, New Delhi, 2012.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, #p 910, Second Edition, The MIT Press, Cambridge , Massachusetts London, England McGraw-Hill Book Company, 2001.

[18] Oded Goldreich, Introduction to Complexity Theory - Lecture Notes, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, ISRAEL, July 31, 1999.

[19] Luca Trevisan, Lecture Notes on Computational Complexity, Computer Science Division, U.C. Berkeley. Fall 2002, Revised May 2004.

[20] J. Hoogeveen, H. Oosterhout, and S. van de Velde. New lower and upper bounds for scheduling around a small common due date. Operations Research, 42:102–110, 1994.

[21] Thomas E. O'Neil, On Clustering in the Subset Sum Problem, University of North Dakota, Grand Forks, ND 58202-9015.

[22] A. Caprara and U. Pferschy. Packing bins with minimal slack. Technical report, University of Graz, 2002.

[23] R. Stearns and H. Hunt, "Power Indices and Easier Hard Problems", Mathematical Systems Theory 23, pp. 209-225, 1990.

[24] Yuli Yem, Priority Algorithms for the Subset-Sum Problem, Graduate Department of Computer Science, University of Toronto, 2006.

[25] Richard M. Karp, "Reducibility among combinatorial problems," in Complexity of Computer Computations, Raymond E. Miller & James W. Thatcher (eds.), Plenum Press, NY, 1972.

[26] Subexponential Time, http://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-5906-5_436, March 8th, 2015.

[27] Soumendra Nanda, Subset Sum Problem, CS 105: Algorithms (Grad), March 2, 2005

[28] T. E. O'Neil, "The Importance of Symmetric Representation," Proceedings of the 2009 International Conference on Foundations of Computer Science (FCS 2009), pp. 115-119, 2009.

[29] Sean A. Irvine, John G. Cleary, Ingrid Rinsma-Melchert, The Subset Sum Problem and Arithmetic Coding, Department of Computer Science, The University of Waikato, Hamilton New Zealand, Working Paper 95/7, March 1995.

[30] Jing Wang : The Subset Sum Problem: Reducing Time Complexity of NP-Completeness with Quantum Search, Bo Moon , University of South Florida 2012.

[31] S. Martello and P. Toth, Worst case analysis of greedy algorithms for the subset sum problem, Mathematical Programming, 28,198-205, 1984.

[32] A. Caprara and U. Pferschy. Worst-case analysis of the subset sum algorithm for bin packing. Operations Research Letters, 32:159–166, 2004.2006.

Bibliography:

- Marco ALmeida Rogerio Reis, Efficient Representation of Integer Sets, Faculdade De Ciencias Universidade Do Porto, 2006

- Mark Cieliebak, Institute of Theoretical Computer Science, On the complexity of variations of equal sum subsets, ETH Zurich, Switzerland, 2008

- Jun Kogure, Noboru Kunihiro, PAPER On the Hardness of Subset Sum Problem from Different intervals, Ieice Trans. Fundamentals, Vol. E95-A, 2012

- Jahanzeb Maqbool Hashmi, Solving Subset-Sum Problem by using Genetic Algorithm Approach, Department of Computer Engineering, Ajou University, 2012

- Claus Peter Schnorr and Taras.Shevchenko, Solving Subset Sum Problems of Density close to 1 by "randomized" BKZ-reduction, Fachbereich Informatik and Mathematik, Goethe-Universitat Frankfurt, Germany, 2012