Computational Complexity Analysis of Combinatorial Algorithms in the Context of the Knapsack Model

Bashar Bin Usman*, Ibrahim Abdullahi, Adamu A. Isah, A.E. Okeyinka
Department of Computer Science, Faculty of Natural Sciences,
Ibrahim Badamasi Babangida University,
Lapai, Niger State, Nigeria.
**Email: basharbinusman1@gmail.com*

***Abstract*-- The Knapsack problem is a complex optimization challenge, often impractical to solve exhaustively. Categorized as a Non-Deterministic Polynomial problem, it allows fast solution verification, but solution discovery time is unpredictable. The aim of this study is to compare the complexities of the knapsack model using greedy, genetic, and dynamic programming algorithms. Our findings, show that the greedy algorithm outperforms genetic, and dynamic programming algorithms in terms of time complexity, providing the most efficient solution for the knapsack model. Future study will delve into hybrid approaches, aiming to leverage the unique strengths of each algorithm and alleviate their limitations.**

keywords: knapsack, complexity, greedy, dynamic programming, genetic, heuristic and algorithm

I. BACKGROUND OF THE STUDY

The Knapsack problem poses a challenging combinatorial optimization issue, and finding its exact solution through exhaustive search methods is often considered impractical. This problem falls into the category of Non-Deterministic polynomial problems, for which solutions can be checked quickly, but the time taken to find a solution is uncertain. The question of whether NP problems can be solved in polynomial time remains one of the most significant unsolved problems in theoretical computer science [1]

The problem holds a distinguished status in the field of computational mathematics due to its wide-ranging practical applications in areas such as finance, logistics, container ship management, portfolio management, and aircraft cargo loading. Consider Mr. Bin Usman preparing for a journey. He has a backpack with a limited weight capacity c, and there are various items available for the journey, each with its weight wi and profit pi. The challenge is to select the items that will provide the most profit while ensuring that the total weight does not exceed the backpack's capacity. This problem is formally stated as follows:

$$\text{Max} \sum_{i=1}^{n} (x_i p_i)$$

subject to the constraints

$$\sum_{i=1}^{n} (x_i w_i) \leq c,$$

$$x_i \in (0,1)$$

Any class of problems in finance or network optimization, be it transportation, communication, production planning, and so forth, that bears resemblance to the above is considered a knapsack problem [2]. Two versions of the knapsack problem are the 0-1 knapsack problem and the fractional knapsack problem [3] To solve these problems for optimal or near-optimal solutions, various algorithms/heuristics have been developed.

This study aimed to effectively solve the knapsack problem by employing three different approaches, namely, greedy, genetic, and dynamic programming. To accomplish this, computational time measures of these three approximate algorithms were collected.

This paper is structured as follows: Section 2 discusses the Research Motivation. Section 3 provides an overview of related literature. In Section 4, we present the conceptual framework, including discussions on random number generation, Greedy, genetic, and Dynamic Programming algorithms. Section 5 presents the Result and findings, while Section 6 outlines the Conclusion.

II. RESEARCH MOTIVATION

Combinatorial problems are prevalent in computer science and real-world applications, necessitating the careful selection of the most suitable algorithms. Our study focuses on the knapsack model, wherein we compare the efficiency of the greedy, genetic, and dynamic programming methods, with a primary emphasis on time measurements. This comprehensive approach empowers informed decision-making in algorithm selection, furthering algorithm design and optimization, and ultimately enhancing computational solutions.

III. Related Works

The knapsack problem, central in combinatorial optimization, has wide practical applications in resource allocation, project scheduling, and portfolio optimization. This review examines its complexity and use of optimization algorithms. [4] compared dynamic programming and greedy algorithms for solving the 0/1 Knapsack Problem and fractional Knapsack problem with an input size of 5 numbers. The experiment evaluated algorithmic complexity using optimal profit and execution time. While both algorithms achieved similar profits, dynamic programming was faster. The research suggested dynamic programming as the more promising approach in terms of time efficiency. The study's scope was limited to input size five. [5] Greedy and dynamic programming algorithms were employed to tackle the Knapsack problem, and both were implemented in the same programming environment. To evaluate their performance, the complexity of the programs and, consequently, the algorithms were carefully assessed. The outcome of this comparative analysis indicates that the Greedy algorithm outperforms the dynamic programming approach in terms of efficiency when it comes to solving the Knapsack problem.

[6] A comparison between Dynamic Programming and Greedy Algorithm for solving the Knapsack Problem is explored. The aim is to maximize item benefits in a knapsack within its capacity. Dynamic programming is found to outperform the greedy algorithm in terms of time and total value generated. Unlike the greedy approach, dynamic programming exhibits parallelism. The study is confined to maximizing capacity and time considerations. [7] The effectiveness of Greedy and Dynamic Programming algorithms in solving the Knapsack problem was examined. Results from Java implementations demonstrated that Dynamic Programming yields superior optimal solutions, while Greedy is more efficient in terms of runtime and space. Java JDK 8.0 was used for implementation, with item weights generated using the JavaRandom.next() method. [8] Examined the use of greedy and dynamic programming algorithms for the knapsack problem. Chosen for their simplicity and efficiency, the greedy algorithm provides quick but sometimes suboptimal solutions, while dynamic programming ensures optimal solutions at a slower pace. The study compares their application properties and scope, ultimately favouring the superior efficiency of the greedy algorithm for the knapsack problem. The comparison focused solely on the time complexity of the two algorithms. [9] Compared the efficiency of three algorithms: greedy, dynamic programming, and branch-and-bound, implemented in C++. The findings revealed that the greedy approach served as the most efficient heuristic, closely followed by the dynamic programming method. In contrast, the branch-and-bound algorithm demonstrated the least efficiency among the three methods.

IV. Research Methodology

We have evaluated the complexities of the knapsack model using three distinct combinatorial optimization algorithms. We aimed to conduct a comparative analysis to assess the efficiency of these algorithms when utilized in the context of the knapsack model. To achieve this, we synthesized the complexities and subjected them to experimental runs. The overarching objective of this synthesis is to establish a more efficient metric for the knapsack model.

A. Knapsack Problem formulation

Suppose we are given a set of n items, each with a weight w*i* and a profit p*i*, determine the number of each item to include in a knapsack so that the total (weight) capacity c is less than or equal to a given limit and the total profit pi is as large as possible.

$$\text{Max} \sum_{i=1}^{n} (x_i p_i)$$

subject to the constraints

$$\sum_{=1}^{n} (x_i w_i) \leq c,$$

$$x_i \in \{0,1\} \quad \forall_i \in \{1,2,3\ldots., n\}$$

We formulate the knapsack problem (kp) by using the binary variable xi ($i = 1,\ldots,n$), where the outcome xi=1 indicates that item i must be included in the knapsack and xi = 0 indicates that item i has to be left

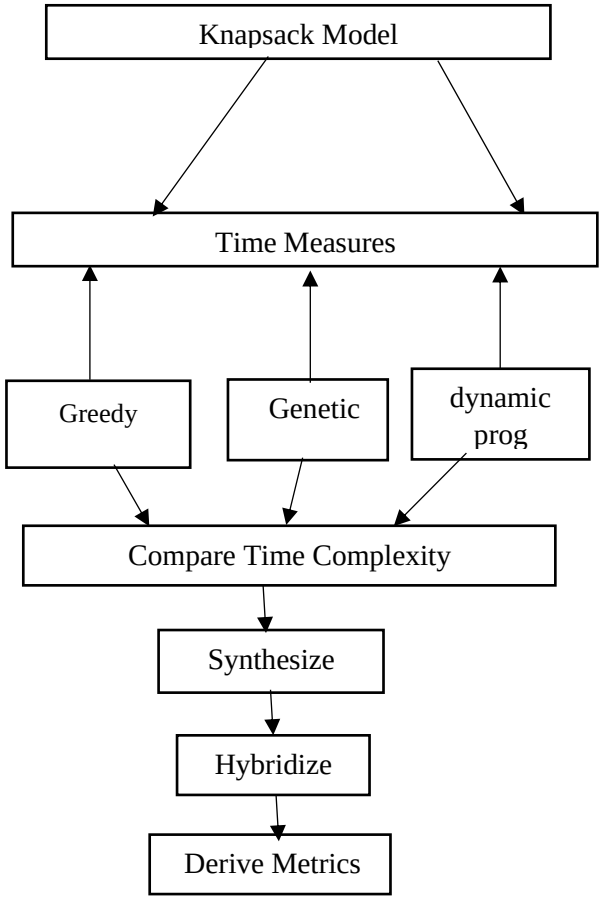

Figure1: Conceptual Framework

B. *Algorithms for Solving Knapsack Problem*

There were four algorithms in the overall study, these are:

i. Random number generation algorithm.
ii. Greedy algorithm.
iii. Genetic algorithm
iv. Dynamic programming algorithm

i. Random Number Generation Algorithm

```
1. import random
2. function generate_random_numbers(n,
 start_range, end_range):
3. generated_numbers = []
4.   for i from 1 to n:
5.           random_num =
 random.randint(start_range, end_range)
6. return generated_numbers  # Return
 the list of generated random numbers
7.     n = 10  # Number of random
 numbers to generate
8.    start_range = 1
9.    end_range = 100
10.         random_numbers =
 generate_random_numbers(n,
 start_range, end_range)
11.     print("Generated random
 numbers: random_numbers)
```

ii. Greedy approach implementationm

The greedy algorithm for the knapsack problem selects items based on a greedy criterion, prioritizing those with the highest value-to-weight ratio. It iteratively adds items to the knapsack until the weight limit is reached.

```
Greedy (p,w,c,i)
//objective: To obtain the maximum profit
of the knapsack
// input: list of items, each with a profit
pi and a weight wi
// the capacity of knapsack c
//  output: the maximum profit made by
filling the knapsack
      for i=1 to n do
         x=select (w)
      if  feasible (x) then
        solution= solution+x
      Endif
        Repeat
      Return (solution)
```

iii. Genetic approach implementation

A genetic algorithm for the knapsack problem is a heuristic approach inspired by natural selection. It involves representing potential solutions as chromosomes, combining them through crossover and mutating them to create new generations. The process includes evaluating fitness based on the knapsack constraints, selecting the fittest solutions for reproduction, and repeating the process until a satisfactory solution is found, aiming to optimize the selection of items to maximize value while respecting the capacity limit of the knapsack.1.

```
1. Initialize the population with random
chromosomes.
2. Evaluate the fitness of each chromosome
in the population.
3.  Repeat  for  a  fixed  number  of
generations:
     a. Select parent chromosomes based on
fitness (e.g., roulette wheel selection,
tournament selection).
     b. Perform crossover to create new
offspring chromosomes.
     c. Perform mutation on the offspring
chromosomes.
     d. Evaluate the fitness of the new
chromosomes.
     e. Replace the old population with the
new population.
4. Return the chromosome with the best
fitness as the solution.
```

Fitness function:

```
Evaluate each chromosome by calculating the
total value of the selected items in the
knapsack.
If the total weight of the selected items
exceeds the maximum weight, the fitness is
set to 0.
```

Crossover:

```
Select two parent chromosomes and choose a
crossover point.
Create  new  offspring  by  combining  the
substrings from the parents.
```

Mutation:

```
Randomly flip bits in the chromosome with
a low probability.
```

iv. Dynamic Programming approach implementation

Is an optimization method that efficiently solves the knapsack problem by breaking it into smaller subproblems. It constructs a table to store the maximum value that can be achieved at each capacity, considering previous subproblem solutions. By iteratively filling the table, it ensures optimal solutions for each subproblem, leading to the determination of the overall maximum value.

```
Dynamicalg(p,w,c,i,j,n,t)
//Objective: To obtain the maximum profit
of the knapsack
// Input: list of items, each with a profit
pi and a weight wi
```

```
  The capacity of knapsack c
// Output: the maximum profit made by
filling the knapsack
For (int i=0; i<=n, i++)
  {
For (int j=0; j<=c, j++)
  {
If (i=0, && j=o)
  t[i,j]=0;
  Elseif
     (wi > j)
t(i,j)=max(t[i-1,j], pi+t[i-1,j]-wi)
Else t[i, j]=t[1-i, j]
 }
}
End
Return [n,c]
```

V. Results and Findings

All algorithms discussed in this section were implemented using the Python programming language. We varied the array size (n) while keeping the capacity size constant. Each algorithm was tested five times, and the average execution time was recorded. Before the comprehensive testing, we initially verified the functionality and correctness of each algorithm using small-sized arrays.

The experimental data presented in the tables and figures below illustrate the execution times in milliseconds for the greedy, genetic, and dynamic programming algorithms in the implementation of the knapsack problem with varying dataset sizes (n).

Upon comparison, it shows that the greedy algorithm outperforms other algorithms in terms of time complexity, providing the most efficient solution for the knapsack problem. It is followed by the genetic heuristic, which is comparatively efficient. However, the algorithm with the least favorable time complexity is the dynamic programming approach, as its complexity grows when dealing with large data input compared to the remaining two heuristics.

Based on this comparison, we can conclude that the greedy algorithm is more time-efficient than the other two algorithms when dealing with the knapsack model.

Table 1: Comparing Greedy, Genetic and Dynamic Programming Algorithms

N	Greedy Alg.	Genetic Alg.	D.P Alg.
5	0.025034	0.041716	0.000506
10	0.028076	0.046286	0.001427
15	0.044823	0.05813	0.004249
20	0.052929	0.060524	0.008023
25	0.055075	0.065419	0.01293
30	0.057697	0.071643	0.017568
35	0.063181	0.076825	0.02314
50	0.049805	0.10016	0.039341
100	0.126953	0.162349	0.173023
250	0.210205	0.279513	0.674624
500	0.313721	0.424456	0.82383
Average	0.093409	0.126093	0.161696

Graphical Representations of Algorithm Efficiency

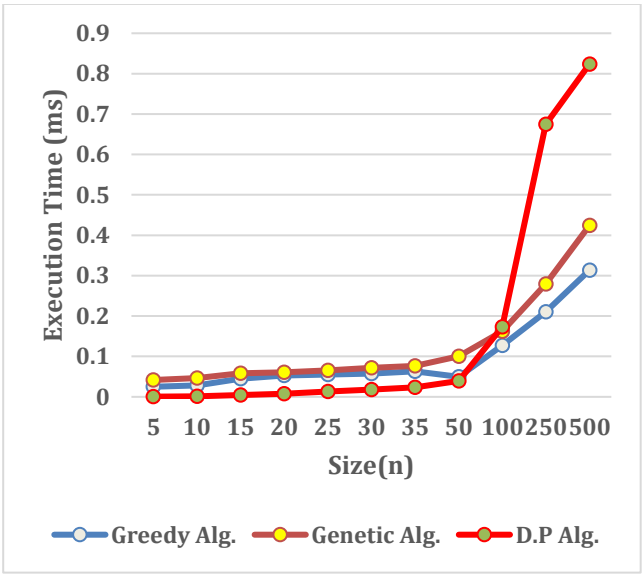

Figure 1: Comparison between Greedy, Genetic and Dynamic Programming Algorithms.

VI. Conclusion

Our research enhances the comprehension of the complex nature of the knapsack model and the effectiveness of different optimization algorithms. The findings provide valuable guidance for selecting the most suitable algorithm based on specific problem attributes, offering practical assistance to professionals dealing with real-world knapsack scenarios. Future investigations will delve into hybrid approaches, aiming to leverage the unique strengths of each algorithm and alleviate their limitations.

REFERENCES

[1] Wikipedia, "Non-Deterministic polynomial time," https://en.wikipedia.org/wiki/NP_(complexity), 2015.

[2] G. Ingargiola and J. Korsh, "A reduction algorithm for zero-one single knapsack problems," Management Science, vol. 20, no. 4, pp. 460-663, 1973.

[3] S. Goddard, "Dynamic programming 0-1 Knapsack problem," http://www.cse.uni.edu/-goddard/Courses/CSCE310J, 20015.

[4] J. Vala, D. Monaka, and J. Pandya, "Comparative Analysis of Dynamic and Greedy Approaches for Dynamic Programming," International Journal of Modern Trends in Engineering and Research, vol. 5, 2014.

[5] Oluyinka I. O. and Okeyinka A. E., "Comparative analysis of the greedy method and dynamic programming in solving the knapsack problem," Global Journal of Advanced Engineering Technologies and Sciences, 2015.

[6] San L. A., "Comparative Study of Dynamic Programming and Greedy Method," International Journal of Computer Applications Technology and Research, vol. 8, pp. 2319-8656, 2019.

[7] N. A. Al Etawi and F. T. Aburomman, "0/1 knapsack problem: greedy vs. dynamic programming," International Journal of Advanced Engineering and Management Research, vol. 5, no. 2, 2020.

[8] X. Chen, "A Comparison of Greedy Algorithm and Dynamic Programming Algorithm," University of Florida, 2022.

[9] B. B. Usman, I. Abdullahi, and A. E. Okeyinka, "Scientific research for smart, secure and sustainable future," in Proceedings of the Third Fonsa Conference, Ibrahim Badamasi Babangida University, Lapai, Niger State, 2023.