Optimal Paths and Profit: A Study of DP vs. Greedy Solutions in Staircase and Knapsack Problems

Shlok Kumbhar[1], Pranjali Mendhekar[2], Ram Yalmate[3], Prof. Dipti Pandit[4]
Department of Electronics and Telecommunication
Vishwakarma Institute of Technology Pune, India

Abstract: **Optimization problems, for example, the 0-1 Knapsack and Staircase Problems are part of the most important ideas in computer science; a solution has to be optimized to solve them. Optimistic strategies are needed when scenarios of resource allocation, logistics, and cost minimization are involved. The objective of this research work is to compare two important algorithms: Dynamic Programming (DP) and Greedy algorithms as implemented in the Knapsack and Staircase Problems. These algorithms, although fast because they make locally optimal choices, are likely to miss the global optimum. Conversely, DP assures the global optimums results based on cumulative costs. We implemented both approaches to each problem, so it could clearly highlight trade-offs between them. A close examination of our research demonstrates that Greedy is suited to speed-critical applications while DP is more suitable to precision-dependent applications especially for cost minimization and resource management.**

Keywords: **greedy algorithm, dynamic programming, 0-1 knapsack problem, staircase problem, performance comparison.**

I. INTRODUCTION

Advanced computational algorithms have advanced to such a level that they are now an inescapable requirement in handling huge volumes of data and solving a vast scale of real-world problems. Among these diverse algorithmic methods in computational optimization, dynamic programming and greedy algorithms are perhaps the most fundamental two design approaches. Each of these methods has different properties that make them suitable for use with different types of problems, and according to the nature of the data and specifics of the need of research, they often present dramatically disparate results. Especially this paper evaluates the performance in practical mode of two algorithms: The 0-1 knapsack problem and the cost minimization in a staircase problem. This work enlightens programmers about what best practice to apply on optimizing their functions through investigating the way they are implemented in real situations [5]. Any strategy, for the future still, must be optimal in all following states, regardless of the initial conditions.

This is the key feature of an ideal decision algorithm. Both the greedy algorithms and dynamic programming depend on this, and it is known as optimum substructure. Dynamic programming often cuts the problem into pieces that are smaller and more manageable-this can be termed as subproblems, determines a solution for each by recursively using the potential solutions and then recalls the solution to use later to improve the efficiency. Greedy algorithms, on the other hand, depend on achieving a global optimum from the choice of locally optimum solutions at each step. This calls for the need to fine-tune algorithms with better results. However, it has been reasoned that some problems may remain unsolved despite all the attempts and re-engineering with alternative solutions requiring study of different methods. Thus, analyzing the partial knapsack and also 0/1 knapsack issues, it also tosses out differences of algorithms between greedy and dynamic programming. In fact, 0/1 knapsack problem is a highly renowned example for how different types of algorithmic approaches could result in different kinds of outcomes, which would be determined by what approach one takes [5]. The staircase problem again shows that strategic planning is prime importance for the minimization of cost, since each step ensures a huge difference from the final outcome.

Instead, the interest of the paper lies in theoretic rationales behind such approaches and discusses these in relation to their uses in various domains such as project management, logistic, and resource allocation based on comparative analysis.

The paper is targeted towards equipping computer professionals with the skills they require in handling challenges related to the selection of algorithms. By realizing what are the advantages and disadvantages of both the greedy and dynamic programming techniques results appear in more effective and efficient problem-solving in a wide number of programming pursuits. In a nutshell, this paper adds to the already existing body of knowledge concerning optimization techniques, especially for problems of cost minimization and the 0-1 knapsack problem in particular. The results are not only useful in academic research but are helpful in practice with the presentation of each technique's pros and cons. With these details, readers will be able to build good solutions in overcoming difficult optimization problems in their professional fields.

II. LITERATURE REVIEW

There are two paradigms of algorithm design that come in handy in the solving of specific optimization problems-DP and Greedy Algorithms. DP is particularly suited for problems with overlapping subproblems, having optimal substructure, for which optimization requires methods such as memoization and tabulation [2]. It also has significant applications in problems such as the Knapsack Problem and the Longest Common Subsequence. In contrast, Greedy Algorithms work on the assumption that by making a sequence of locally optimum decisions they will somehow magically end up with some kind of global optimum. Very often it indeed leads to solutions faster and simpler in comparison, as with minimum spanning tree algorithms, for

instance. Yet this does not guarantee that the 0/1 Knapsack Problem does not have this optimality for all problems. For instance, 0/1 Knapsack Problem suffers from inefficiency.

In order to solve the Knapsack Problem and other optimization problems, the Greedy and DP algorithms are implemented and optimized in this paper. In actuality, there is a fascinating application for common issues with transportation, product pricing, and vending machines. The study also compares the two approaches used to solve the Minimizing Cost in a Staircase problem, concluding that the Greedy approach "always try to optimize immediate choices, and for DP "it always tries to evaluate cumulative choices at each step so that the global minimum is ensured." This also highlights how much the DP method saves in terms of the overall traversal cost. The study also provides Java pseudo-codes for the Greedy and DP approaches, and a graph shows the total expenses of each technique.

III. METHODOLOGY

A. Greedy Algorithm

1. *Introduction to Greedy Algorithms*

An approach known as a greedy algorithm selects the locally optimal option at each stage in the hopes that the sum of those local solutions will produce the globally optimal solution. The use of greedy algorithms, which have much simpler and more effective versions for a variety of problems, can occasionally result in a less-than-ideal solution [3] [5].

Basic characteristics of Greedy Algorithm:

- Locally optimal choice: The algorithm chooses the best available at each step without backing off previous choices.
- Non-look back: All decisions have been made and would not be reversed; no looking back.
- Efficiency: Greedy algorithms are usually more time-efficient than exhaustive or dynamic methods.

2. *Key Problems Solved by Greedy Algorithms*

- Using both the Kruskal and Prim algorithms, the Minimum Spanning Tree (MST) is calculated.
- The process of determining the shortest path in a graph with nonnegative weights is known as Dijkstra's Shortest Path Algorithm.
- Activity Selection Problem: Choose the maximum number of non-conflicting activities.
- Huffman Coding Huffman code is used to create prefix codes for optimal compression of data [3].

B. Dynamic Programming

1. *Introduction to Dynamic Programming (DP)*

Dynamic programming (DP) is an optimization technique that divides complicated issues into simple subproblems that overlap and only need to be solved once. Memorization is the practice of storing its outcomes for later use. DP is most effective when applied to substructure optimization and overlapping subproblems. [4, 5].

Dynamic Programming Characteristics:

- Optimal substructure: The subproblem's answer can be used to derive the problem's solution.
- Overlapping Subproblems: The problem is recursive and the same subproblems are solved more than once.
- Memoization: We save the solutions for solved subproblems so we would not solve them twice.
- Bottom-up or Top-down: DP can be implemented in two different ways - either top-down, i.e., using recursion and memoization or bottom-up using a table [4].

2. *Key Problems Solved by Dynamic Programming:*

- Fibonacci Sequence Using DP Avoid repetition Compute the nth Fibonacci number efficiently.
- knapsack problem: maximize value in the knapsack under its capacity.
- The Longest Common Subsequence (LCS) function yields the longest subsequence that two sequences have in common.
- Matrix Chain Multiplication: The number of scalar multiplications in matrix chain operations should be optimized.
- Modify the Distance Determine how few operations are required to change one string into another.

C. Key Problems

1. Minimizing Cost in a Staircase Problem: Greedy vs DP

a) Introduction:

A robot should move to a sequence of workstations. For every workstation, some cost is associated. In every step, the robot can proceed one or two steps in any direction. This problem is known as the Minimizing Cost in a Staircase. In this problem, it has been assigned the task to choose an ordered subset of a sequence with the minimum total traversal cost from the beginning to the end of the sequence. In this paper, we are considering two approaches namely, Greedy and Dynamic Programming (DP) [7].

Given the cost array:
cost = [10, 15, 20, 1, 5, 10]
Each element represents the cost of passing through that workstation.

b) Greedy Approach**:**

In the greedy approach, the robot selects the next move based on the immediate lowest cost between the next two workstations. This approach is straightforward but often leads to suboptimal overall results.
Stepwise Greedy Solution:

1. Start at cost[0] = 10.
2. Compare cost[1] = 15 and cost[2] = 20. Choose cost[1] = 15 (lower cost).

3. Compare cost[2] = 20 and cost[3] = 1. Choose cost[3] = 1.
4. Compare cost[4] = 5 and cost[5] = 10. Choose cost[4] = 5.
5. Finally, move to cost[5] = 10.

Greedy Path:
10 → 15 → 1 → 5 → 10

Total Cost: 10 + 15 + 1 + 5 + 10 = 41

Table 1. Step-by-Step Calculation of Total Cost in Greedy Algorithm for Staircase Problem [2]

Step	Current Position	Next Options	Choice	Total Cost
1	0(10)	15, 20	15	10 + 15 = 25
2	1(15)	20,1	1	25+1=26
3	3 (1)	5,10	5	26 + 5 = 31
4	4 (5)	10 (last step)	10	31+10=41

Table 1. shows the greedy approach results in a total cost of although it minimizes individual steps, it overlooks the cumulative cost of the overall path [7].

.

Greedy Approach Pseudocode in Java:

```
Function GreedyMinCost(cost):
   n = length of cost
   totalCost = cost[0]
   currentPosition = 0

   while currentPosition < n - 1:
      if currentPosition + 1 < n and currentPosition + 2 < n:
      if cost[currentPosition + 1] < cost[currentPosition + 2]:
            currentPosition = currentPosition + 1
         else:
            currentPosition = currentPosition + 2
      else if currentPosition + 1 < n:
         currentPosition = currentPosition + 1
      else:
         break
      totalCost = totalCost + cost[currentPosition]
   return totalCost
```

c) Dynamic Programming (DP) Approach**:**

The dynamic programming approach solves this problem by calculating the cumulative cost at each workstation, ensuring the minimum total cost. The recurrence relation is:
minCost[i] = cost[i] + min(minCost[i-1], minCost[i-2])
This relation ensures that at each step, the robot considers the minimum cost from the previous two steps to find the optimal solution.
Stepwise DP Solution:

1. Initialize:
   - minCost[0] = cost[0] = 10
   - minCost[1] = cost[1] = 15
2. For each subsequent step, compute:
   - minCost[2] = cost[2] + min(minCost[1], minCost[0]) = 20 + min(15, 10) = 30
   - minCost[3] = cost[3] + min(minCost[2], minCost[1]) = 1 + min(30, 15) = 16
   - minCost[4] = cost[4] + min(minCost[3], minCost[2]) = 5 + min(16, 30) = 21
   - minCost[5] = cost[5] + min(minCost[4], minCost[3]) = 10 + min(21, 16) = 26

Optimal DP Path:

10 → 15 → 1 → 10

Total Cost: 10 + 15 + 1 + 10 = 26

Table 2. Optimal Path Cost Breakdown Using Dynamic Programming for Staircase Climbing [2]

Step	Current Positon	Previous Cost	DP Calculation	minCost[i]
0	0 (10)	-	minCost[0] = 10	10
1	1 (15)	-	minCost[1] = 15	15
2	2 (20)	minCost[0] = 10, minCost[1] = 15	minCost[2] = 20 + 10 = 30	30
3	3 (1)	minCost[1] = 15, minCost[2] = 30	minCost[3] = 1 + 15 = 16	16
4	4 (5)	minCost[2] = 30, minCost[3] = 16	minCost[4] = 5 + 16 = 21	21
5	5 (10)	minCost[3] = 16, minCost[4] = 21	minCost[5] = 10 + 16 = 26	26

Table 2. shows the dynamic programming approach achieves a total cost of 26, which is significantly lower than the greedy approach. By considering cumulative costs at every step, DP ensures the global minimum is found.

Comparison of Results

- Greedy Approach: Total cost = 41, path: 10 → 15 → 1 → 5 → 10.
- DP Approach: Total cost = 26, optimal path: 10 → 15 → 1 → 10.

The greedy approach focuses on minimizing immediate costs, leading to a higher total cost. In contrast, the dynamic programming method considers the entire journey and calculates the minimal overall cost, producing the optimal solution. In problems requiring cost minimization, the dynamic programming approach clearly outperforms the greedy approach by considering cumulative costs and providing an optimal solution. The comparison highlights how DP achieves a significant reduction in total cost, proving its efficiency in this scenario.

Dynamic Programming Approach Pseudocode in Java:

```
Function DPMinCost(cost):
   n = len(cost)
   if n = 0:
      return 0
   if n = 1:
      return cost[0]
   minCost = list(range(n))
   minCost[0] = cost[0]
   minCost[1]

for i = 2 to n - 1:
   minCost[i] = cost[i] + min(minCost[i - 1], minCost[i - 2])
   return minCost[n - 1]
```

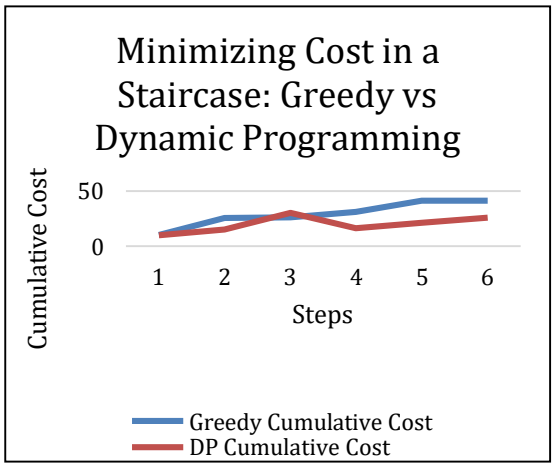

Figure 1: Minimizing Cost in a Staircase: Greedy vs Dynamic Programming [1].

In the Minimizing Cost in a Staircase Problem, this figure shows the total costs of the Greedy and Dynamic Programming (DP) approaches. The Y-axis shows the total expenditures, and the X-axis shows the number of stages from 1 to 6.

Blue line is the Greedy approach, which climbs steeply to a total cost of 41 by step 5, showing a tendency for the Greedy approach to take the cheapest costs in the immediate step. In contrast, the orange line is the DP approach, a smooth increase in cost and results in a total cost of 26 by step 5. This comparison shows that where Greedy concentrates on saving in the short term, DP optimally minimizes overall expenses.

2. Knapsack Problem: Greedy vs DP

a) Introduction:

In the Knapsack Problem, there is a thief with a knapsack and he needs to pick items for maximizing the value without the weight limit. Every object has a specific weight and worth. Dynamic Programming (DP) is the best method for solving the 0/1 knapsack problem, while greedy programming is best suited for the fractional knapsack problem. We go over both strategies in this section with an example [1] [6].

Problem Setting:

Item values and weights are given as follows:

Values: [60,100,120]

Weights: [10,20,30]

Knapsack Capacity: 50 units

b) Greedy Method (Fractional Knapsack:

For the Fractional Knapsack Problem, the Greedy technique can be applied. Items in this issue can be divided into fractions. To maximize the value as quickly as possible before the capacity is full, the strategy is to choose goods with the highest value-to-weight ratio [1].

Step-by-Step Greedy Solution:

1. Value-to-Weight Ratio Calculation:

   First, 60/10 = 6.0;

   second, 100/20 = 5.0;

   third, 120/30 = 4.0

2. Order Items by Ratio (Highest to Lowest)

   Items 1 (6.0), 2 (5.0), and 3 (4.0)

3. Select Items:

   Take all of Item 1 (10 units, 60 value), with 40 units remaining capacity.

   Take all of Item 2 (20 units, 100 value), with 20 units of capacity remaining.

   20/30 of Item 3 (Proportional value: 20/30 × 120 = 80). Take 20 units out of 30.

4. Total Value:

   60+100+80=240

Greedy Approach Pseudocode:

```
Function GreedyKnapsack(values, weights, capacity):
        n = number of values
        ratio = selection of value-to-weight ratio
sort items by ratio in descending order
        totalValue = 0
for each item in sorted_items:
        if capacity ≥ weights[item]
            totalValue += values[item];
            capacity -= weights[item]
        else:
            totalvalue += (values[item] / weights[item]) *
           capacity
       break
       return totalValue
```

Result for Greedy Approach:
Greedy Approach:
Question 1 (10 items, 60 points)
Article 2 (20 marks, 100 words)
20/30 Item 3 20 Units, 80 Value
Total Total 240 (including fractional items)

Dynamic Programming (DP) Approach (0/1 Knapsack):

Since all items must be taken whole numbers, the 0/1 Knapsack problem forces items to be included or not included entirely as compared to the DP method wherein it computes the best result involving all possible combinations of products [1] [4].
DP Solution Solves:
Given a knapsack capacity of w and the first i items, let dp[i][w] be the highest value that can be obtained [6].

1. Introduction
        dp[0][w] = 0 for all w (nothing = no value).
        dp[i][0] = 0 for all i (no capacity = no value).
2. Recurrence Relation:
        For each object i, for each weight w, update DP table
        dp[i][w] =
        max(dp[i−1][w],dp[i−1][w−weight[i−1]]
                +value[i−1])
        If the weight of the item exceeds the capacity w, then reject the item
                dp[i][w] = dp[i−1][w]

Table 3. Knapsack Optimization Table: Maximum Value achieved for Varying Weights and Items [5]

Item → \ Weight ↓	0	10	20	30	40	50
0 Items	0	0	0	0	0	0
Item 1	0	60	60	60	60	60
Item 2	0	60	100	160	160	160
Item 3	0	60	100	160	180	220

Items with values [60,100,120], weights [10,20,30], and a knapsack capacity of 50 are displayed in Table 3.

- Item 1: For capacities 10 or more, take Item 1 (60 value).
- Item 2: For capacities 20 or more, take Item 2 (100 value). If combined with Item 1, we can achieve 160 value.
- Item 3: For capacities 30 or more, the optimal solution is taking Item 1 and Item 3 for a total value of 220.

DP Approach Pseudocode:

```
Function DPKnapsack(values, weights, capacity):
    n = length of values
    dp = a two dimensional array of size (n+1) x (capacity+1)

      for i = 0 to n:
        for w = 0 to capacity:
            if i == 0 or w == 0:
                dp[i][w] = 0
            else if weights[i-1] <= w:
dp[i][w] = dp[i-1][w] or dp[i-1][w-Weight[i-1]] +
                        Value[i-1]
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][capacity]
```

Result for DP Approach:

- DP Path:
    - Item 1 (10 units, 60 value)
    - Item 2 (20 units, 100 value)
    - Item 3 (30 units, 120 value)
- Total Value: 220 (with whole items only)

Result & Discussion:

1. Minimizing Cost in a Staircase Problem

Using the cost array cost = [10, 15, 20, 1, 5, 10], we evaluated both the Greedy and Dynamic Programming (DP) approaches:

- Greedy Approach:

    Path: 10 → 15 → 1 → 5 → 10
    Total Cost: 41

The Greedy approach focuses on immediate costs, resulting in a higher total.

- Dynamic Programming Approach:

    Path: 10 → 15 → 1 → 10
    Total Cost: 26

The DP approach considers cumulative costs, yielding a lower total cost.

Table 3. Comparison for Staircase Problem [1]

Approach	Time Complexity	Space Complexity	Explanation
Greedy	O(n)	O(1)	Single pass with minimal memory usage.
Dynamic Programming	O(n)	O(n)	Single pass with an additional array for costs.

2. Knapsack Problem

Concerning the 0/1 Knapsack Problem parts, the items have been set to values and weights. Let values = [60, 100, 120] and weights = [10, 20, 30] and capacity = 50 [6].

- Greedy Approach (Fractional Knapsack): Total Value: 240 (using a fractional approach)

The Greedy method optimizes based on value-to-weight ratio but only applies to fractional items.

- Dynamic Programming Approach (0/1 Knapsack): Total Value: 220 (taking whole items)

The DP approach calculates maximum value considering the weight constraints.

Table 4. Comparison for Knapsack Problem [5]

Approach	Time Complexity	Space Complexity	Explanation
Greedy (Fractional)	O(nlogn)	O(1)	Sorting items based on ratio, then a linear pass.
Dynamic Programming	O(n×W)	O(n×W)	2D array storing maximum values for each weight.

In the Staircase Problem, the Greedy approach is faster but leads to suboptimal costs. Although it uses more memory, the DP method offers the best result. In contrast to the DP strategy, which ensures the highest result for specified weights, the Greedy solution for the Knapsack Problem performs well in fractional contexts but less well for 0/1 constraints. The significance of choosing the appropriate algorithm based on the problem type, the optimality of the intended solution, and the available resources is demonstrated by this comparison analysis.

Performance Comparison:

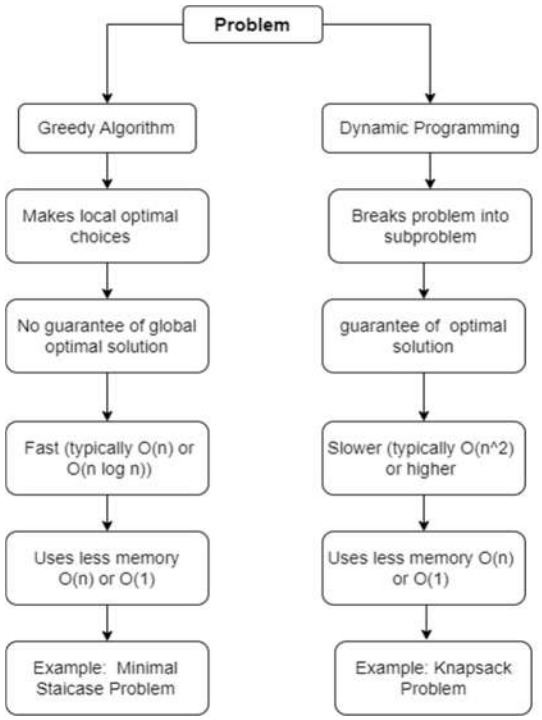

Figure 2. Comparative Flowchart on DP and Greedy [2]

Figure 2. shows flowchart diagram of a comparison of DP and Greedy algorithms It is showing their decision making processes, time and space complexities, as well as examples of when to use them, basing on nature of problem [2].

IV. CONCLUSION

A comparative study of Dynamic Programming (DP) and Greedy Algorithms reveals that both these methodologies play significant roles in solving optimization problems, yet each with its strengths and weaknesses [2]. Dynamic Programming is ideal for scenarios showing the existence of optimal substructure and overlapping subproblems-an approach that ensures optimum solutions through detailed exploration and the storage of results [5]. In contrast, Greedy Algorithms give a very efficient, intuitive approach with locally optimal choice making in such a manner that even though global optimality is not guaranteed, the solutions obtained become very sufficient in practical applications.

The understanding of the intricacies of each method enables researchers and practitioners to be able to choose the best algorithm for their task, taking into account the trade-offs between computational efficiency and optimality. It has been shown that if the problem under consideration is a cost minimization problem, then the Dynamic Programming approach outperforms the Greedy method in deciding based on cumulative costs, hence an optimal solution can be given. According to the explicit results, the Dynamic Programming Algorithm outperforms the Greedy Algorithm and random selection in resolving resource allocation issues. This conclusion is based on the outcomes of a number of computations that used the Dynamic Programming approach

to solve complex optimization problems effectively and efficiently.

V. REFERENCES

[1] X. C., "Comparative analysis of Greedy Algorithm and Dynamic Programming Algorithm" published in the SHS Web Conf, (2022).

[2] S.L. Aung "Comparative Study of Dynamic Programming and Greedy Method. *Recently Published*". International Journal of Computer Applications Technology and Research,, (2019).

[3] Y. Wang, "Review on Greedy Algorithm," Proceedings of the 3rd International Conference on Computing Innovation and Applied Physics,, (2023).

[4] C. E. J a R. R. Dohrmann, "A Dynamic Programming Approach to Rocket Guidance Problems.", American Institute of Aeronautics and Astronautics, Inc, 2016.

[5] F. &. L. I. &. S. M. Furini presented "An Effective Dynamic Programming Algorithm for the Minimum-Cost Maximal Knapsack Packing Problem.", European Journal of Operational Research, (2017).

[6] G. I. S. E. &. A. Sampurno, "An Analysis of the DP Algorithm and GA on Integer Knapsack Problem in the Freight Transportation". Scientific Journal of Informatics, Volume 6, Issue 2, December 2018.

[7] Y. Wu Compare Dynamic Programming and Greedy Algorithms and the way to solve 0-1 Knapsack Problem.