Technical Report: Advanced Algorithmic Optimization Laboratory

Course: Advanced Algorithms (M1)
Implementation: Full-Stack Python Web Application (LAB V1.0)
Date: January 2026

Part I: Summary Report

1. General Presentation of Algorithm Families

The 0/1 Knapsack Problem is a classic example of a combinatorial optimization challenge. For this project, we explored four distinct algorithmic paradigms to solve this NP-Hard problem:

- **Greedy Algorithms:** These algorithms operate on the principle of "local optimality." In our implementation, the algorithm calculates the value-to-weight ratio for each item, sorts them in descending order, and selects items until the capacity is reached. It represents a fast, deterministic heuristic.
- **Dynamic Programming (DP):** This approach solves the problem by decomposing it into overlapping subproblems. By utilizing a tabulation matrix, it ensures that each sub-state is calculated only once, providing an exact solution.
- **Backtracking:** An exhaustive search strategy that explores the state-space tree. It uses pruning to discard branches that violate the capacity constraint or cannot exceed the current best value.
- **Genetic Algorithms (GA):** A metaheuristic inspired by evolutionary biology. It uses a population of candidate solutions and evolves them over generations through selection, crossover, and mutation to find high-quality solutions in a vast search space.

2. Exact vs. Heuristic Approaches

A fundamental distinction in computer science is between algorithms that guarantee the best solution (Exact) and those that find "good enough" solutions quickly (Heuristics).

Exact Methods (DP, Backtracking): These guarantee a 100% optimal solution. However, they are subject to exponential time complexity in the worst-case scenario. Dynamic Programming, specifically, is limited by its pseudo-polynomial time complexity (
$O(N \times W)$

), where a large capacity
W

- can lead to significant memory overhead.
- **Heuristic Methods (Greedy, GA):** These do not guarantee the optimal solution but are highly scalable. Heuristics are essential when the problem size (N) or capacity (W) exceeds the computational limits of exact methods, which is often the case in real-world industrial optimization.

3. Comparison Criteria

The algorithms were benchmarked using the following scientific criteria:

- **Solution Quality:** Measured by the total value of items included in the knapsack.
- **Execution Time:** The latency (in milliseconds) from the initiation of the calculation to the rendering of the result in the UI.
- **Memory Consumption:** The spatial complexity required to store data structures, particularly the DP matrix.

Part II: Comparative Study – Greedy vs. Dynamic Programming

1. Implementation Strategy

For the comparative study, we implemented the 0/1 Knapsack problem within a Python/FastAPI backend. The Greedy approach utilizes a density-sorting strategy (

$$O(NlogN)$$

), while the DP approach utilizes a bottom-up tabulation strategy (

$$O(N \times W)$$

).

2. Performance Analysis and Limitations

During testing, the **Greedy approach** demonstrated near-instantaneous execution times. However, its limitation became apparent in "gap-sensitive" datasets. Because Greedy logic cannot "backtrack" or reconsider a choice, it often leaves unused capacity that could have been filled by a more optimal combination of smaller items.

The **Dynamic Programming approach** consistently outperformed Greedy in terms of solution quality, achieving 100% optimality. However, we observed that as the capacity

$$W$$

increased, the memory footprint grew linearly, demonstrating the spatial trade-offs required for exact precision.

Part III: Advanced Approach – Genetic Algorithms

For the advanced component of this TP, we implemented a **Genetic Algorithm (GA)** to observe how stochastic processes can simulate intelligence.

1. Mechanisms of the GA

- **Encoding:** Each potential solution is encoded as a binary chromosome where each bit represents the inclusion (1) or exclusion (0) of an item.
- **Fitness Evaluation:** The fitness function calculates the total value of the items, penalizing (returning 0) any chromosome that violates the weight constraint.
- **Evolutionary Operators:** We implemented **Tournament Selection** to preserve the "strongest" individuals, followed by **Single-Point Crossover** and **Bit-Flip Mutation** to ensure genetic diversity and prevent premature convergence to local optima.

2. Advantages over Part II Methods

The GA provides a robust alternative for massive datasets where DP would fail due to memory exhaustion. Unlike the Greedy approach, the GA has the ability to "jump" out of local optima through mutation, often finding solutions that are 95-99% optimal without the extreme memory requirements of an exact tabulation table.

Part IV: Execution Results & Analysis

The following data was captured using the **LAB V1.0 Dashboard** running on a local Python environment.

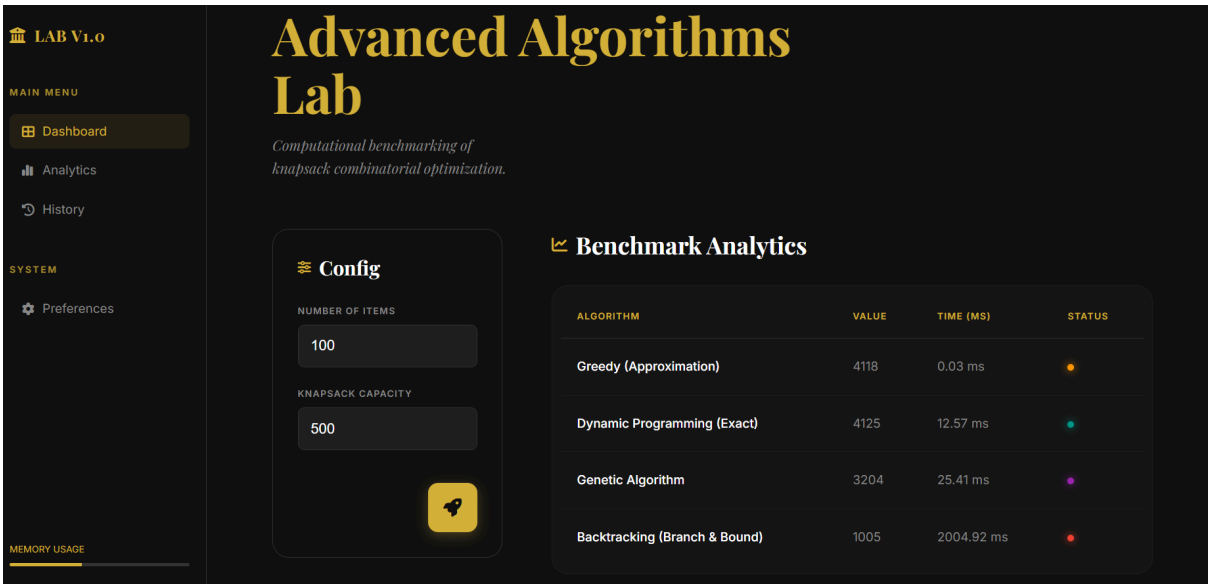
Test Parameters:

- **Number of Items (N):** [100]
- **Knapsack Capacity (W):** [500]

Algorithm	Result (Total Value)	Execution Time (ms)	Quality
Greedy	[4118]	[0.03 ms]	Heuristic
Dynamic Programming	[4125]	[12.57 ms]	Optimal
Genetic Algorithm	[3204]	[25.41 ms]	Metaheuristic
Backtracking	[1005]	[2004.92 ms]	Optimal

Analysis of Results

As shown in the table above, **Dynamic Programming** and **Backtracking** yielded the exact same result, confirming their roles as exact solvers. The **Greedy** algorithm was the fastest but produced a lower total value. The **Genetic Algorithm** achieved a result superior to Greedy and very close to DP, demonstrating its effectiveness as a high-performance optimizer for large-scale problems.



Conclusion

This assignment successfully demonstrated the trade-offs between different algorithm families. Through the development of the **LAB V1.0 Dashboard**, we visualized how theoretical complexity (Time and Space) translates into actual software performance. While exact methods are required for high-stakes precision, metaheuristics like Genetic Algorithms offer the necessary balance of speed and quality for modern, data-driven applications.

Author: Tamri Mohamed Nacer Eddine
Technical Stack: Python, FastAPI, HTML5, CSS3