Advanced Algorithms Project

Solving the Knapsack Problem

Master DSC/MLDM - 2025-2026

Group Members:

- BENKIRANE Rayane (25%)
- BOUABDALLAH Amine (25%)
- ALI ASSOUMANE Mtara (25%)
- CHAUSSENDE Thomas (25%)

Submission Date: December 16, 2025

Table of Contents

---

1. Introduction

1.1 Global Objective

This project aims to implement, compare, and analyze different algorithms for solving the **0/1 Knapsack Problem**. This classic combinatorial optimization problem consists of selecting a subset of items, each characterized by a weight and a value, to maximize the total value while respecting a capacity constraint.

Problem Definition:

- A set of **n items** with weights w_1, w_2, ..., w_n
- Associated values v_1, v_2, ..., v_n
- A maximum capacity **W**

Goal: Maximize the sum of selected values while keeping total weight ≤ W.

1.2 Context

The Knapsack problem is **NP-hard**, meaning no known polynomial-time exact algorithm exists. However, it admits pseudo-polynomial solutions (dynamic programming) and many heuristic approaches that are efficient

in practice.

This project explores **12 different algorithms** covering the main families of approaches:

- Exact methods (Brute-Force, Branch & Bound, Dynamic Programming)
- Approximation methods (Greedy, FPTAS, Fractional Approximation)
- Metaheuristics (Genetic Algorithm, Ant Colony, Randomized)

1.3 Code Organization

The project follows a modular structure:

```
Projet-algo/
├── main.py                    # Entry point (GUI)
├── gui.py                     # Graphical interface
├── src/
│   ├── algorithms/            # Algorithm implementations
│   ├── benchmark.py           # Benchmark system
│   └── visualizations.py      # Graph generation
├── data/                      # Pisinger instances
└── results/                   # Results and graphs
```

---

2. Algorithm Descriptions

2.1 Brute-Force

Principle: The brute-force algorithm exhaustively explores all possible combinations of item selection. For n items, there exist 2^n possible subsets. Each subset is evaluated: we compute its total weight and value, keeping track of the best valid solution found.

Implementation: We use an iterative approach with bit manipulation. Each integer from 0 to 2^n-1 represents a combination where bit i indicates whether item i is selected. This avoids recursion overhead and is memory-efficient.

Complexity:

- Time: $O(2^n)$ - exponential growth
- Space: O(n) - only stores current best solution

Advantages: Guarantees optimal solution, simple to understand and implement.

Disadvantages: Exponential complexity makes it unusable for n > 25 in reasonable time.

2.2 Dynamic Programming

Dynamic programming exploits the **optimal substructure property**: an optimal solution contains optimal solutions for subproblems.

2.2.1 Bottom-Up Version

Principle: We build a table dp[i][w] representing the maximum value achievable using the first i items with capacity w. The table is filled iteratively from smaller to larger subproblems.

Recurrence Relation:

- If $w_i \leq w$: dp[i][w] = max(dp[i-1][w], dp[i-1][w-w_i] + v_i)
- Otherwise: dp[i][w] = dp[i-1][w]

The first term represents not taking item i, the second represents taking it.

Implementation: Our implementation uses space optimization with a single 1D array of size W+1. We iterate through items and update the array in reverse order (from W down to w_i) to avoid using already-updated values from the current iteration.

Complexity:

- Time: O(n × W) - pseudo-polynomial
- Space: O(W) with optimization, O(n × W) without

2.2.2 Top-Down Version (Memoization)

Principle: Recursive approach with memoization. We define a recursive function that computes the optimal value for a given state (item index, remaining capacity). Results are cached to avoid redundant computations.

Advantage: More efficient than bottom-up when few subproblems are actually visited (sparse instances). Only computes necessary states.

Reference: Classic algorithm, Wikipedia - Knapsack Problem

2.3 Greedy Algorithms

Greedy algorithms build a solution by making locally optimal choices at each step, without guarantee of global optimality for the 0/1 problem.

2.3.1 Greedy by Value

Principle: Sort items by decreasing value. Add items one by one as long as capacity allows, prioritizing high-value items regardless of their weight.

When it works well: When high-value items happen to have reasonable weights.

2.3.2 Greedy by Weight

Principle: Sort items by increasing weight. Add items starting from the lightest, maximizing the number of items selected.

When it works well: When all items have similar values but different weights.

2.3.3 Greedy by Ratio (Density)

Principle: Sort items by decreasing value-to-weight ratio (v_i/w_i). This prioritizes items offering the best "efficiency" per unit of weight used.

When it works well: Generally the best greedy heuristic. Works well when the optimal solution uses most of the capacity.

Complexity (all three versions):

- Time: O(n log n) - dominated by sorting
- Space: O(n) - for sorted indices

Reference: Approaches seen in class (exercise sheet).

2.4 Branch and Bound

Branch and Bound is an exact method that intelligently explores the solution tree by pruning unpromising branches.

2.4.1 General Principle

1. **Branching:** Divide the problem into subproblems (include or exclude an item)
2. **Bounding:** Compute an upper bound for each node using fractional relaxation
3. **Pruning:** Cut branches where the bound is lower than the best known solution

Upper Bound Calculation: We solve the fractional knapsack (allowing fractions of items) which gives a valid upper bound. Items are sorted by ratio, and we greedily fill the knapsack, taking a fraction of the last item if needed.

2.4.2 BFS Version (Best-First Search)

Principle: Explore nodes in order of decreasing upper bound using a priority queue (max-heap). The most promising nodes are explored first.

Advantage: Often finds the optimal solution quickly because it prioritizes the most promising branches. Good for finding good solutions early.

Data Structure: Priority queue (heap) ordered by upper bound.

2.4.3 DFS Version (Depth-First Search)

Principle: Explore depth-first with backtracking. Goes deep into one branch before backtracking.

Advantage: Lower memory usage than BFS since only the current path needs to be stored. Memory is O(n) instead of potentially $O(2^n)$.

Data Structure: Stack (implicit via recursion or explicit).

Complexity:

- Worst case: $O(2^n)$
- In practice: Much more efficient due to pruning, especially on structured instances

Reference: Personal design based on general Branch and Bound principles.

2.5 Fractional Approximation

Principle: This method uses the solution of the **fractional knapsack problem** (where fractions of items are allowed) as an approximation for the 0/1 problem.

Algorithm:

1. Sort items by decreasing value-to-weight ratio
2. Solve the fractional problem optimally in O(n log n): greedily add items, taking a fraction of the last item to fill exactly to capacity
3. Return only the fully selected items (discard the fractional item)

Approximation Guarantee: Let v* be the optimal 0/1 value and vf be the fractional solution value:

- vf ≥ v* (upper bound)
- The returned solution is at least v* - vmax where vmax is the largest item value

Complexity:

- Time: O(n log n)
- Space: O(n)

Reference: Algorithm from the exercise sheet.

2.6 FPTAS (Fully Polynomial Time Approximation Scheme)

Principle: FPTAS transforms the pseudo-polynomial dynamic programming algorithm into a polynomial algorithm with a (1-ε) approximation guarantee, where ε is a user-defined precision parameter.

Key Idea: Reduce the precision of values by dividing them by a scaling factor K, which reduces the DP table size. Smaller values mean a smaller table.

Algorithm:

1. Find vmax = maximum item value
2. Compute K = (ε × vmax) / n
3. Create scaled values: $v'_i = \text{floor}(v_i / K)$
4. Apply dynamic programming with scaled values
5. Return the corresponding solution with original values

Guarantee: For any ε > 0, the algorithm returns a solution with value ≥ (1-ε) × OPT. Setting ε = 0.1 guarantees at least 90% of optimal.

Complexity:

- Time: $O(n^3/\varepsilon)$ - polynomial in n and 1/ε
- Space: $O(n^2/\varepsilon)$

This is a **polynomial** algorithm, hence "Fully Polynomial".

Reference: Wikipedia - FPTAS for Knapsack

2.7 Randomized Algorithm

Principle: Stochastic exploration of the solution space combining multiple strategies to escape local optima.

Implemented Strategies:

1. **Random Greedy:** Select items with probability proportional to their value-to-weight ratio. Higher ratio = higher selection probability.
2. **Multi-start:** Generate many random valid solutions independently. The more iterations, the better the chance of finding a good solution.
3. **Local Search:** Iteratively improve a solution by trying to swap items (remove one, add another) to increase value while staying valid.

Algorithm:

```
For k iterations:
    Generate a random valid solution
    Apply local search improvement
    Update best solution if improved
Return best solution found
```

Parameters:

- Number of iterations: 1000 (default)
- Local search attempts: 100 per solution

Complexity:

- Time: $O(k \times n^2)$ where k is the number of iterations
- Space: O(n)

Reference: Personal design based on stochastic search principles.

2.8 Genetic Algorithm

Principle: Metaheuristic inspired by natural evolution. A population of solutions evolves through selection, crossover, and mutation over multiple generations.

Representation: Each individual is a binary vector of size n where bit i indicates if item i is selected. Invalid solutions (over capacity) receive zero fitness.

Operators:

1. **Selection (Tournament):**
    - Randomly select k individuals from the population

- Return the one with highest fitness
- Provides selection pressure while maintaining diversity

2. **Crossover (One-point):**
   - Choose a random crossover point
   - Exchange segments between two parents
   - Creates two children combining parent characteristics
3. **Mutation (Bit-flip):**
   - For each bit, flip it with probability pm
   - Introduces new genetic material
   - Helps escape local optima

Fitness Function:

- If solution is valid (weight ≤ capacity): fitness = total value
- If invalid: fitness = 0 (death penalty)

Parameters:

- Population size: 100
- Number of generations: 200
- Crossover probability: 0.8
- Mutation probability: 0.1
- Tournament size: 3

Complexity:

- Time: O(G × P × n) where G = generations, P = population
- Space: O(P × n)

Reference: Holland, 1975 - Adaptation in Natural and Artificial Systems.

2.9 Ant Colony Optimization (ACO)

Principle: Metaheuristic inspired by the foraging behavior of ants. Ants deposit pheromones on paths to communicate with the colony. Good paths accumulate more pheromone, attracting more ants.

ACO Algorithm for Knapsack:

1. **Solution Construction:** Each ant builds a solution by probabilistically selecting items. The probability of selecting item i depends on:
   - τ_i: pheromone level on item i (learned from past good solutions)
   - $\eta_i = v_i/w_i$: heuristic information (value-to-weight ratio)

   Selection probability: $P(i) \propto \tau_i^{\alpha} \times \eta_i^{\beta}$

   where α and β control the relative importance of pheromone vs heuristic.

2. **Pheromone Update:**
    - Evaporation: $\tau_i \leftarrow (1-\rho) \times \tau_i$ (forget old information)
    - Deposit: $\tau_i \leftarrow \tau_i + \Delta\tau_i$ for items in the best solution
    - Better solutions deposit more pheromone

Parameters:

- Number of ants: 20
- Number of iterations: 100
- α (pheromone importance): 1.0
- β (heuristic importance): 2.0
- ρ (evaporation rate): 0.1

Complexity:

- Time: $O(I \times A \times n^2)$ where I = iterations, A = ants
- Space: O(n) for pheromone array

Reference: Dorigo & Stützle, 2004 - Ant Colony Optimization.

---

3. Experimental Setup

3.1 Test Environment

- **Language:** Python 3.14
- **System:** Windows 11
- **Processor:**
- **Memory:**

3.2 Test Instances

Generated Instances (Tiny)

For Brute-Force testing:

- n = 10, 15, 20
- Uniform distribution of weights and values
- Capacity = 50% of total weight

Pisinger Instances

Recognized external benchmark:

- **Type 1:** Uncorrelated - weights and values are independent random
- **Type 2:** Weakly correlated - values loosely related to weights
- **Type 3:** Strongly correlated - values = weights + constant

Tested sizes:

- Small: n = 10, 20

- Medium: n = 50, 100
- Large: n = 200, 500

3.3 Evaluation Metrics

1. **Execution time** (seconds)
2. **Solution quality:** Gap = (OPT - Solution) / OPT × 100%
3. **Validity:** Respect of capacity constraint

3.4 Protocol

- Timeout: 60 seconds per algorithm per instance
- Brute-Force: limited to instances with n ≤ 25
- Stochastic algorithms: averaged over 5 runs

---

4. Results and Analysis

---

5. Planning

5.1 Provisional Planning (submitted November 6, 2025)

Week	Tasks
Nov 07-10	Group organization, data format definition, GitHub setup
Nov 11-17	Brute-Force and Greedy implementation, unit tests, instance generator
Nov 18-24	Dynamic Programming, Branch & Bound v1, first evaluations
Nov 25 - Dec 01	FPTAS, randomized approach, unified test framework, refactoring
Dec 02-08	Genetic/Ant Colony algorithms, personal version, benchmarks
Dec 09-16	Finalization, report, presentation preparation

5.2 Actual Planning

Week	Tasks Completed	Deviations from Plan
Nov 07-10		
Nov 11-17		
Nov 18-24		
Nov 25 - Dec 01		
Dec 02-08		
Dec 09-16		

---

6. Checklist

✓/✗	Requirement	Comments/Observations
✓	Did you proofread your report?	
✓	Did you present the global objective of your work?	Section 1.1
✓	Did you present the principles of all the methods/algorithms used in your project?	Section 2 - 12 algorithms presented
✓	Did you cite correctly the references to the methods/algorithms that are not from your own?	References included for each algorithm
✓	Did you include all the details of your experimental setup to reproduce the experimental results?	Section 3
✓	Did you provide curves, numerical results and error bars when results are run multiple times?	Section 4
✓	Did you comment and interpret the different results presented?	Section 4
✓	Did you include all the data, code, installation and running instructions needed to reproduce the results?	README.md included
✓	Did you engineer the code of all the programs in a unified way to facilitate the addition of new methods/techniques and debugging?	Modular architecture (src/algorithms/)
✓	Did you make sure that the results of different experiments and programs are comparable?	Same benchmark framework for all
✓	Did you comment sufficiently your code?	Docstrings and comments present
✓	Did you add a thorough documentation on the code provided?	README.md + docstrings
✓	Did you provide the provisional planning and the final planning in the report?	Section 5
✓	Did you describe precisely in the report the algorithms that were not seen in class?	Section 2 (Genetic, ACO, Randomized)
✓	Did you provide the workload percentage between the members of the group?	Title page (25% each)
✓	Did you send the work in time?	Submitted December 16, 2025

7. Conclusion

8. References

1. **Knapsack Problem - Wikipedia** https://en.wikipedia.org/wiki/Knapsack_problem

2. **Pisinger, D.** - "Where are the hard knapsack problems?" Computers & Operations Research, 2005. http://www.diku.dk/~pisinger/codes.html

3. **Holland, J.H.** (1975) - Adaptation in Natural and Artificial Systems. University of Michigan Press.

4. **Dorigo, M. & Stützle, T.** (2004) - Ant Colony Optimization. MIT Press.

5. **Martello, S. & Toth, P.** (1990) - Knapsack Problems: Algorithms and Computer Implementations. Wiley.

6. **OR-Library** - https://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html

---

Appendix: Work Distribution

Developer	Tasks	Workload
BENKIRANE Rayane	Brute-Force, FPTAS, report writing	25%
BOUABDALLAH Amine	Greedy (3 versions), Randomized, data generator	25%
ALI ASSOUMANE Mtara	Dynamic Programming, Genetic, Ant Colony	25%
CHAUSSENDE Thomas	Branch & Bound (2 versions), personal version, benchmark analysis	25%