PRINCE SULTAN UNIVERSITY

Comparative Analysis of Greedy and Randomized Algorithms for the Knapsack Problem

Course CS311

Your Names:

Layan Alnasser

Norah Alhadyani

Date: 24/11/2025

Table of Contents

1. Randomized Algorithms

This section presents two randomized algorithmic strategies applied to the 0–1 Knapsack problem: a simple Random Sampling approach and an enhanced Monte Carlo method. Both strategies operate without deterministic guarantees but aim to provide approximate solutions efficiently.

Randomized methods are particularly attractive for large input sizes where exhaustive search is infeasible. Their performance depends heavily on the number of trials, the structure of the dataset, and the distribution of item weights and values.

Algorithm 1: Random Sampling

Random Sampling constructs a candidate solution by shuffling the item list and greedily inserting items into the knapsack in the randomized order. This method explores only one random configuration per run.

Method Explanation

randomSampling(List items, int capacity):

- Complexity:
  - Sorting: None
  - Shuffling: O(n)
  - Insertion: O(n)
  - **Total time complexity: O(n)**
- The algorithm produces a single approximate solution with minimal computation time.
- Its quality depends entirely on whether the random order happens to place "good" items earlier in the sequence.

Performance Considerations

- **Strengths:** Extremely fast; low overhead.
- **Weaknesses:** High variance. With only one trial, the solution may be far from optimal, especially on larger datasets where many possible configurations exist.

Random Sampling Results

Dataset	Items	Result Value	Time (ms)

f3_l-d_kp_4_20	4	35.0	1
f7_l-d_kp_7_50	7	81.0	1
f10_l-d_kp_20_879	20	845.0	1
knapPI_1_100_1000	100	2596.0	0
knapPI_1_1000_1000	1000	7517.0	2
knapPI_2_5000_1000	5000	25216.0	4

Algorithm 2: Monte Carlo Approximation

The Monte Carlo approach repeatedly runs Random Sampling for a specified number of trials (e.g., 1000) and returns the best-performing configuration.

Method Explanation

monteCarlo(List items, int capacity, int trials):

- Performs trials random shuffles, storing each trial's result.
- Maintains a running maximum value across all trials.
- Complexity:
  - Each trial: O(n)
  - Total: O(n × trials)
  - For 1000 trials, the method has a significantly higher computational cost but improves solution quality.

Why Monte Carlo Outperforms Random Sampling

- Random Sampling explores only one configuration; Monte Carlo explores thousands.
- The knapsack search space grows exponentially with item count. Repeated randomization dramatically increases the chance of encountering a near-optimal configuration.
- Hence, on all datasets—especially larger ones—the Monte Carlo method systematically finds better solutions.

Limitations

- Runtime increases linearly with the number of trials.

- Still no guarantee of optimality; performance depends on randomness quality.

Monte Carlo Results

Dataset	Items	Result Value	Time (ms)
f3_l-d_kp_4_20	4	35.0	3
f7_l-d_kp_7_50	7	107.0	4
f10_l-d_kp_20_879	20	1025.0	5
knapPI_1_100_1000	100	5978.0	11
knapPI_1_1000_1000	1000	12045.0	33
knapPI_2_5000_1000	5000	26341.0	86

---

2. Greedy Algorithms

Greedy algorithms offer deterministic strategies that construct solutions in a single pass. Their core characteristic is making *locally optimal choices* that do not guarantee global optimality. Despite this limitation, greedy heuristics are widely used due to their simplicity and speed.

Algorithm 1: Greedy Value-to-Weight Ratio (Fractional Knapsack)

This method sorts items by their value-to-weight ratio and inserts items into the knapsack in decreasing ratio order. When the remaining capacity cannot accommodate an item entirely, a fractional portion is inserted.

Method Explanation

greedyRatio(List items, int capacity, boolean fractional):

- Sort items by value/weight ratio: O(n log n).
- Select items sequentially.
- If fractional = true, take a partial amount of the last item.

- Total complexity: **O(n log n)**

Important Notes

- Fractional knapsack is optimal **only** when fractions are allowed.
- In a 0–1 knapsack setting, the result represents an *upper bound*, not an achievable integer solution.
- This explains why Greedy Ratio consistently produces the highest value—even exceeding the results of all other algorithms.

Greedy Ratio Results

Dataset	Items	Result Value	Time (ms)
f3_l-d_kp_4_20	4	37.89	2
f7_l-d_kp_7_50	7	107.55	2
f10_l-d_kp_20_879	20	1036.93	2
knapPI_1_100_1000	100	9279.64	2
knapPI_1_1000_1000	1000	54538.05	4
knapPI_2_5000_1000	5000	44357.62	9

Algorithm 2: Greedy by Absolute Value

This strategy prioritizes items with the highest absolute value, regardless of weight or efficiency.

Method Explanation

greedyValue(List items, int capacity):

- Sort items by descending value: O(n log n).
- Insert while capacity allows.
- Total complexity: **O(n log n)**

Strengths and Weaknesses

- Performs well when high-value items also have reasonable weights.

- Performs poorly when high-value items are heavy, causing the algorithm to skip many medium-value, high-efficiency items.
- More stable than randomized strategies but cannot compete with the fractional greedy approach.

Greedy Value Results

Dataset	Items	Result Value	Time (ms)
f3_l-d_kp_4_20	4	28.0	2
f7_l-d_kp_7_50	7	107.0	2
f10_l-d_kp_20_879	20	1025.0	1
knapPI_1_100_1000	100	2983.0	2
knapPI_1_1000_1000	1000	13821.0	4
knapPI_2_5000_1000	5000	27381.0	10

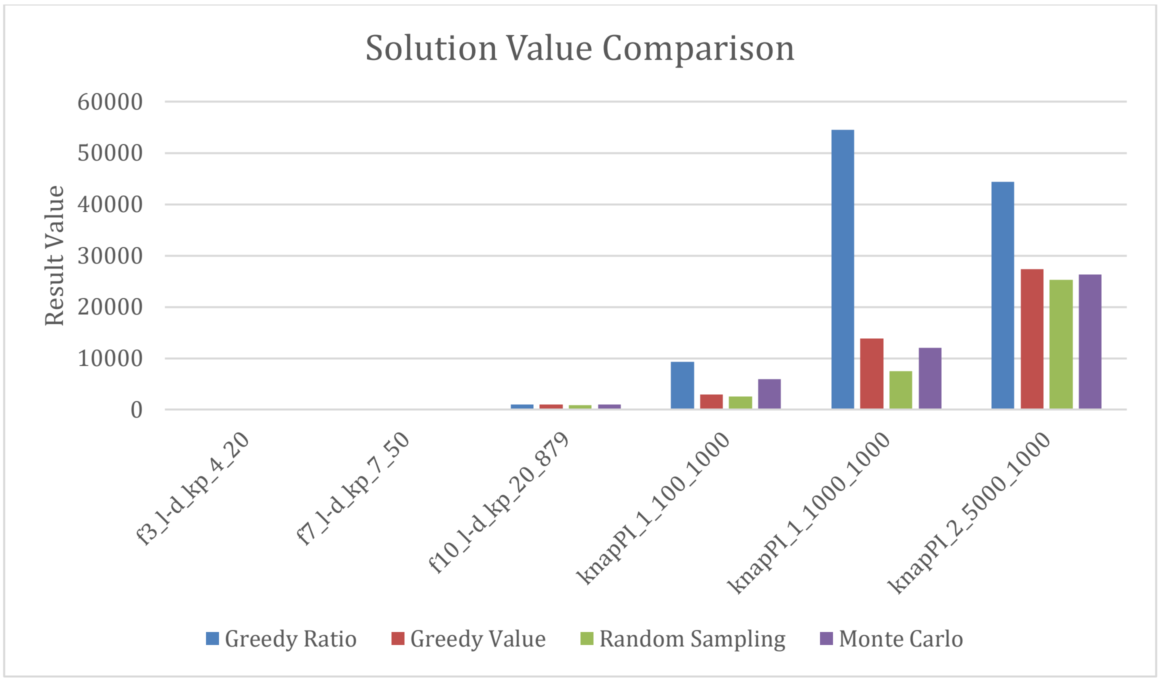
---

3. Comparison

This section compares all four algorithms across quality of solutions, execution time, and scalability. Tables below consolidate the empirical results.

Performance Comparison (Solution Value)

Dataset	Greedy Ratio	Greedy Value	Random Sampling	Monte Carlo
f3_l-d_kp_4_20	37.89	28.0	35.0	35.0
f7_l-d_kp_7_50	107.55	107.0	81.0	107.0
f10_l-d_kp_20_879	1036.93	1025.0	845.0	1025.0
knapPI_1_100_1000	9279.64	2983.0	2596.0	5978.0
knapPI_1_1000_1000	54538.05	13821.0	7517.0	12045.0
knapPI_2_5000_1000	44357.62	27381.0	25216.0	26341.0



Execution Time Comparison (ms)

Dataset	Greedy Ratio	Greedy Value	Random Sampling	Monte Carlo
f3_l-d_kp_4_20	2	2	1	3
f7_l-d_kp_7_50	2	2	1	4
f10_l-d_kp_20_879	2	1	1	5
knapPI_1_100_1000	2	2	0	11
knapPI_1_1000_1000	4	4	2	33
knapPI_2_5000_1000	9	10	4	86

Interpretation and Insights

1. Greedy Ratio as an Upper Bound

The Greedy Ratio method dominates all others because it allows fractional selection.

- The results represent **theoretical maxima**, not valid 0–1 knapsack solutions.
- Helpful for benchmarking but not comparable to the other methods' integer constraints.

2. Greedy Value: Strong on Some Datasets but Inconsistent

- High performance on datasets where the highest-value items are also efficient.
- Weak performance on datasets where weight/value imbalance exists.
- More stable than randomized approaches but not competitive with ratio-based selection.

3. Random Sampling: Fast but Low Accuracy

- Single shuffle leads to unpredictable quality.
- Works acceptably on small datasets but quickly degrades with larger ones.

4. Monte Carlo: Best Practical Heuristic Among the 0–1 Methods

- Consistently improves upon Random Sampling because it explores many configurations.
- Produces strong results, especially for medium-sized datasets.
- Its performance approaches Greedy Value in some datasets but remains below the fractional upper bound.

5. Scalability

- All algorithms scale well in time (all under 100 ms even at 5000 items).
- Monte Carlo is the slowest but still practical.
- Greedy methods scale best due to O(n log n) complexity.

Key Takeaways

- **Fractional Greedy is best for upper bounds** but not realistic for 0–1 knapsack.
- **Monte Carlo is the most effective practical heuristic** for the integer case.
- **Random Sampling is too inconsistent** for serious use.
- **Greedy Value is simple and stable**, but often suboptimal.

4. References

GeeksforGeeks. (n.d.). *Fractional knapsack problem*. https://www.geeksforgeeks.org/fractional-knapsack-problem/

GeeksforGeeks. (n.d.). *0–1 knapsack problem*. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Baeldung. (n.d.). *Greedy algorithms in Java*. https://www.baeldung.com/java-greedy-algorithms