Optimizing knapsack using heuristics: A Comparative Study

Nitin Kankanala

Department Of Comp. Science

Georgia State University

nkankanala1@student.gsu.edu

Murali Ram Ravipati

Department Of Comp. Science

Georgia State University

mravipati2@student.gsu.edu

Sai Vaishnavi Koganti

Department Of Comp. Science

Georgia State University

skoganti3@student.gsu.edu

Abstract: The Knapsack Problem is a well-known combinatorial optimization problem. It involves determining the number of each weighted item to be included in a hypothetical knapsack, so that the total weight is less than or equal to the required weight while maximizing the total value of the items. Metaheuristic algorithms have been widely used to solve such NP-hard problems, as they offer effective and efficient solutions. This paper compares four metaheuristic algorithms, namely Tabu Search, Scatter Search and Local Search, and Genetic Algorithm for solving the Knapsack Problem. The algorithms are evaluated based on three metrics: execution time, solution quality, and relative difference to the best-known quality solution.

Keywords- Metaheuristic algorithms, Knapsack Problem, Local Search, Tabu Search, Scatter Search, Genetic Algorithm.

1. Introduction

Optimization problems involve obtaining estimates of factors that maximize or minimize a target function while fulfilling objectives or constraints. These problems are prevalent in various areas of work, including manufacturing systems, engineering designs, economics, and finance. Optimization problems can be classified into different types such as multimodal optimization, multi-objective optimization, and combinatorial optimization. In this paper, we focus on the Knapsack problem as a Combinatorial Optimization Problem (COP) to analyze and compare different metaheuristic algorithms for solving an NP-hard problem.

The Knapsack problem is a combinatorial optimization problem that involves finding the number of each weighted item to include in a hypothetical knapsack such that the total weight is less than or equal to the required weight. The problem has many practical applications, including resource allocation, finance, and production planning. Metaheuristic algorithms are heuristic optimization algorithms that explore the search space in a non-exhaustive manner. Metaheuristic algorithms aim to provide a close-to-optimal solution in reduced time and cost compared to the exhaustive search approach. In this paper, we compare and analyze four different metaheuristic algorithms, including Tabu Search, Scatter Search, Local Search and Genetic Algorithm.

We compare the efficiency and accuracy of each of the algorithms based on certain parameters, including execution time, solution quality, and relative difference to the best-known quality. The experiment is conducted on a set of standard instances of the Knapsack problem. The chosen algorithms were tested on three types of problem instances: uncorrelated data instances, weakly correlated instances, and strongly correlated instances. Uncorrelated data instances refer to problem instances where there is no relationship between the item weights and values. Weakly correlated instances, on the other hand, have a slight correlation between item weights and values, while strongly correlated instances have a strong correlation.

Analyzing the performance of optimization algorithms on different types of problem instances is important to evaluate the robustness and versatility of these algorithms. Using uncorrelated data instances is a standard way of testing optimization algorithms because these instances do not contain any pattern or structure, and the optimization algorithm must explore the entire search space to find the optimal solution.

However, in real-world scenarios, there may be correlations between item weights and values. For example, in a shopping scenario, items that are heavy may be more expensive. Thus, testing the algorithms on weakly and strongly correlated instances can help to understand their performance in more realistic scenarios.

2. Problem Formulation

The knapsack problem is a well-known optimization problem that involves filling a pack or a bag with a group of elements of known weights and values while staying within the pack's restricted weight limit. This problem has real-life applications in various fields, such as cutting stock problems, project selection problems, loading problems, capital budgeting problems, parcel and outline of electronic circuits, and even the selection of crew for a flight.

ILP stands for Integer Linear Programming, which is a mathematical optimization technique that involves a linear objective function and linear inequality constraints with integer decision variables. In the context of the knapsack problem, an ILP formulation would involve defining decision variables to represent whether each item is included or not in the knapsack, and then formulating constraints that ensure the weight capacity of the knapsack is not exceeded. The objective function would then aim to maximize the total value of the items selected.

A general formulation of the 0-1 knapsack problem as an ILP is as follows:

Maximize:

$\sum v_i x_i$ (i=1 to n)

Subject to:

$\sum w_i x_i \leq W$ (i=1 to n)

$x_i \in \{0, 1\}$ (i=1 to n)

where:

n = number of items

v_i = value of item i

w_i = weight of item i

W = weight capacity of the knapsack

x_i = decision variable representing whether item i is selected (1) or not (0).

Metaheuristic algorithms are iterative processes that control subordinate heuristics by combining unique ideas to explore and exploit the search space to find an optimal solution. These algorithms are commonly used to solve complex problems, and some of the well-known ones include Genetic Algorithm, Tabu Search, Local Search, Ant Colony Optimization, Scatter Search and Genetic Algorithm. To compare the solutions of the Knapsack Problem, this paper proposes the use of three Meta-Heuristic algorithms, namely Tabu Search, Scatter Search, Local Search and Genetic Algorithm.

Tabu Search is a metaheuristic algorithm that uses a tabu list to keep track of previously visited solutions and to avoid searching close to those solutions. This algorithm also uses aspiration criteria to incorporate previously unvisited good quality solutions.

Scatter Search, on the other hand, consolidates all the other solutions to acquire a solution by investigating the feasible region and working on sets of solutions called 'Reference Sets.'

Local Search is an algorithm that searches for a locally optimal solution by repeatedly improving a starting solution through subroutines until a locally optimal solution is reached.

Genetic Algorithm is a population-based optimization algorithm that uses the concepts of natural selection and genetics to find the optimal solution.

To compare the performance of these four algorithms, the paper uses three types of data instances, namely Uncorrelated data, Weakly correlated instances, and Strongly correlated instances. The results of the comparative analysis can help determine which algorithm performs better for a given type of data instance.

3. Algorithms

Tabu Search is a metaheuristic algorithm that is commonly used to solve optimization problems, including the knapsack problem. In this algorithm, a search is initiated from an arbitrary initial solution, and the search progresses through the neighboring solutions. Tabu search utilizes a tabu list, which acts as a short-term memory to store the solutions that have already been searched and to prevent revisiting them. This list also contains prohibited moves. The main advantage of this list is that the algorithm never gets stuck in local maxima. The algorithm uses aspiration criteria, which allows previously unvisited good quality solutions to be incorporated into the search. In the context of the knapsack problem, the tabu search algorithm can be used to search for the optimal combination of items that can be packed into the knapsack without exceeding its weight limit.

The tabu search algorithm for the knapsack problem typically starts with a randomly generated solution, and then proceeds to the neighboring solutions by making one or more changes to the current solution. The algorithm evaluates each neighboring solution to determine whether it is better than the current solution, and if so, it updates the current solution accordingly. The algorithm then adds the current solution to the tabu list to prevent it from being revisited in the near future. This process is repeated until a stopping criterion is met, such as reaching a maximum number of iterations or not improving the solution for a certain number of iterations. The final solution is the best solution found by the algorithm during the search.

Local search is a heuristic optimization technique that works by iteratively improving a candidate solution to a problem. It starts with an initial solution and then tries to find a better solution by searching through the space of neighboring solutions. The goal is to find a solution that is locally optimal, meaning that no other feasible solution can be found in the immediate vicinity that is better than the current one. The process of finding the best neighboring solution and making it the new current solution is repeated until no further improvement can be made. In the context of the knapsack problem, local search can be used to find a solution that maximizes the value of the items in the knapsack subject to the weight constraint. The search begins with an initial solution, which could be a randomly generated solution or some other heuristic solution. Then, the algorithm explores the space of neighboring solutions by making small changes to the current solution, such as adding or removing an item from the knapsack. If a new solution is found that is better than the current solution, it becomes the new current solution. The process continues until no further improvement can be made.

One of the main advantages of local search is its simplicity and efficiency, especially for small to medium-sized problems. However, it can get trapped in local optima, meaning that it may find a solution that is optimal in its immediate neighborhood but not globally optimal. To address this issue, various modifications to the basic local search algorithm have been proposed, such as the use of random restarts or the inclusion of stochastic elements to allow for exploration of a wider search space.

Scatter search is a metaheuristic algorithm that is used for solving optimization problems. It was first introduced by Fred Glover in 1977. The main idea behind scatter search is to explore the solution space by generating and combining different subsets of solutions. The algorithm starts by generating a set of initial solutions using some heuristic method. It then uses these solutions to create a set of reference solutions. A reference set is a collection of solutions that are diverse and of high quality. Next, the algorithm searches for new solutions by combining different subsets of solutions from the reference set. These new solutions are evaluated and added to the reference set if they are of high quality and diverse.

The process of generating new solutions and updating the reference set continues until a stopping criterion is met. This criterion could be a maximum number of iterations, a time limit, or reaching a certain level of quality in the solutions. Scatter search is known for its ability to converge to high-quality solutions and its flexibility in incorporating problem-specific knowledge. However, it may require many function evaluations, and the generation of diverse solutions can be challenging for some problems.

Genetic Algorithm (GA) is a metaheuristic algorithm inspired by the process of natural selection and genetics in biology. It is a population-based optimization technique that mimics the process of evolution in nature, where the fittest individuals are selected for reproduction and their offspring inherit their characteristics. In the context of optimization problems, the individuals in the population represent candidate solutions, and the fitness of each solution is evaluated based on the objective function. The GA process starts with an initial population of candidate solutions, which are randomly generated. The next step is to evaluate the fitness of each solution, which is typically done by computing the objective function. Then, the best solutions are selected for reproduction, based on their fitness, and the offspring are generated by applying genetic operators such as crossover and mutation. Crossover involves swapping parts of the selected solutions to create new ones, while mutation involves introducing small random changes to the solutions.

The offspring are then added to the population, and the process is repeated for a number of generations. The algorithm terminates when a stopping criterion is met, such as reaching a maximum number of generations or achieving a desired level of convergence.

In the context of the knapsack problem, a GA would represent candidate solutions as strings of binary digits, where each digit represents whether a particular item is included in the knapsack. The fitness of each solution would be computed as the total value of the items in the knapsack, subject to the weight constraint. The genetic operators would be designed to preserve the feasibility of the solutions, i.e., to ensure that the weight constraint is not violated.

4. Implementation

The code implements four metaheuristic algorithms - Local Search, Scatter Search, Tabu Search, and Genetic Algorithm - for solving the Knapsack problem. The problem data is read from a file that contains multiple instances, and each instance is solved separately using each of the algorithms.

Local Search: For each instance, the algorithm first initializes a random solution and evaluates its fitness using a Knapsack fitness function. The fitness function calculates the total value of the selected items in the solution and checks if the total weight of the selected items does not exceed the given capacity. If the total weight exceeds the capacity, the fitness is set to zero.

The algorithm then iteratively explores the neighborhood of the current solution by generating all possible neighbors and selecting the best neighbor that has the highest fitness. The fitness of each neighbor is evaluated using the Knapsack fitness function. The algorithm updates the current solution with the best neighbor and repeats the process until a fixed number of iterations is reached.

Finally, the algorithm outputs the best solution found, along with its fitness, the total weight and value of the selected items, the capacity of the Knapsack problem, and the time taken to run the algorithm.

Tabu Search: The algorithm starts with a random initial solution and iteratively improves it by exploring its neighborhood solutions. The neighborhood of a solution is defined by flipping the value of a single element in the solution vector. To avoid getting stuck in local optima, the algorithm maintains a tabu list, which keeps track of the recent solutions that have been visited and prevents the algorithm from revisiting them for a certain number of iterations. The length of the tabu tenure, which specifies the number of iterations for which a solution is considered tabu, is a parameter of the algorithm.

At each iteration, the algorithm evaluates the fitness of the current solution and its neighbors. The fitness of a solution is the total value of the items included in the knapsack, subject to the capacity constraint. The algorithm selects the best non-tabu neighbor solution and updates the tabu list accordingly. If the fitness of the selected neighbor solution is better than the current solution, it becomes the new current solution.

The algorithm terminates after a fixed number of iterations, and the best solution found during the search is outputted. The output includes the objective value, which is the total value of the items in the selected subset, the selected items, their total weight, and the time taken by the algorithm to find the solution.

Scatter Search: The algorithm starts by defining the problem data, including the weights and values of each item and the knapsack capacity. It then generates an initial population of random solutions, where each solution is a binary string indicating whether an item is included in the knapsack or not. The algorithm runs for a fixed number of iterations, during which it generates a set of reference sets, each containing a fixed number of candidate solutions. The candidates are generated randomly and added to the reference set. The reference sets are then merged, and the resulting candidates are sorted by their fitness (i.e., the total value of the items in the solution). The best candidates are selected to form the new population for the next iteration.

After the iterations are complete, the best solution is selected from the final population based on its fitness. The solution's objective value and the items included in the knapsack are then output. Overall, the algorithm uses a scatter search approach to explore the solution space by generating and merging different sets of candidate solutions. By iteratively selecting the best candidates, the algorithm converges towards a high-quality solution to the Knapsack problem.

Genetic Algorithm: The given code is an implementation of a genetic algorithm to solve the Knapsack problem for multiple instances. The

Knapsack problem is a combinatorial optimization problem where given a set of items with corresponding weights and values, the goal is to maximize the total value of items that can be fit into a knapsack of a given capacity.

The code reads instance data from a file in DIMACS format, where each instance contains the number of items, knapsack capacity, and weight-value pairs for each item. It then applies the genetic algorithm to each instance and outputs the best solution found, along with its objective value, weight, and execution time.

The genetic algorithm involves the following steps:

Initialization: create an initial population of candidate solutions, where each solution is a binary string representing which items to include in the knapsack.

Selection: select parents from the population based on their fitness, which is the objective value of their corresponding solution.

Crossover: create offspring solutions by combining parts of the parent solutions.

Mutation: randomly flip some bits in the offspring solutions to introduce diversity.

Evaluation: evaluate the fitness of the offspring solutions.

Replacement: replace the population with the offspring solutions.

Repeat steps Selection - replacement for a fixed number of generations. The genetic algorithm is repeated for each instance in the input file, and the best solution found for each instance is outputted along with some relevant information.

5. Simulations

As described earlier in implementation we have written Python code for all four algorithms i.e., Local Search, Tabu Search, Scatter Search and Genetic Algorithm. The data is generated for them accordingly basing on one primary constraint i.e., correlation between profits and weights on items in each problem instance.

The dimensionality of the data is controlled by the number of items (n) and the capacity of the knapsack (cap). For Low Dimensionality Uncorrelated Data (LD-UC) instances, n and cap are randomly chosen between 10-25 and 10-1000, respectively. For High Dimensionality Uncorrelated Data (HD-UC), High Dimensionality Weakly Correlated Data (HD-WC), and High Dimensionality Strongly Correlated Data (HD-SC) instances, the same range of n and cap is used, but n is multiplied by 10 to increase the dimensionality of the problem.

The correlation between the item values and weights is controlled by the corr parameter. For LD-UC instances, corr is set to 0, indicating no correlation. For HD-UC, HD-WC, and HD-SC instances, corr is set to 1, 2, and 3, respectively, indicating increasing levels of correlation. The correlation is introduced by adding a random perturbation to the item weights, with the strength of the perturbation depending on the value of corr. The generated instances are saved in a text file, with each instance being identified by a unique filename based on its characteristics (e.g., knapPI_1_100_1000_1 for a high dimensional uncorrelated instance with n=100 and cap=1000). Each instance is represented by a list of item values and weights, followed by the capacity of the knapsack.

The generated instances can be used to evaluate the performance of different Knapsack algorithms on different types of data. For example, LD-UC instances are expected to be easier to solve than HD-UC, HD-WC, and HD-SC instances, because the problem dimensionality is lower and there is no correlation between the item values and weights. On the other hand, HD-UC instances are expected to be easier to solve than HD-WC and HD-SC instances, because the correlation between the item values and weights is weaker. Finally, HD-SC instances are expected to be the most difficult to solve, because the correlation between the item values and weights is the strongest.

We have a python main file which represents the 4 algorithms implementation and visualization in the same file. There is another file that is used for generating the above mentioned Data in specified format for the 4 classes of data we are working on.

6. Results

On the low dimensionality data we have represented some graphs which indicate that the scatter search is best in terms of runtime but has 2nd best optimal value calculation among the remaining. Local Search is the least performing in finding the optimal solution with slow convergence speed and obtaining near optimal solution. Below are the visualizations for the algorithms that were run on low dimensionality data.

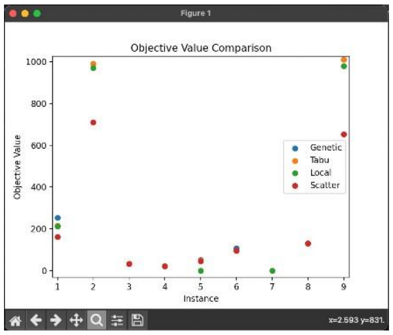
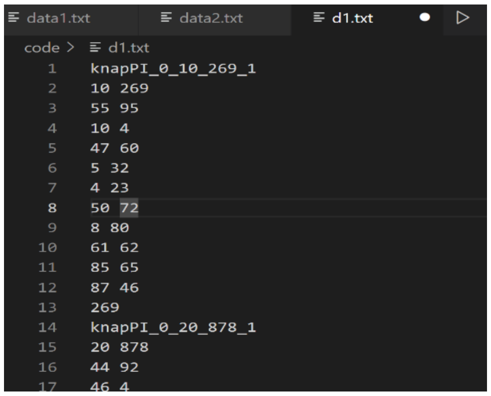


Fig 1. Objective Value Comparison

These results are obtained on by applying the Low D data a sample data is represented in below images.



```
knapPI_0_10_269_1
10 269
55 95
10 4
47 60
5 32
4 23
50 72
8 80
61 62
85 65
87 46
269
knapPI_0_20_878_1
20 878
44 92
46 4
```

Fig 2. Sample Data

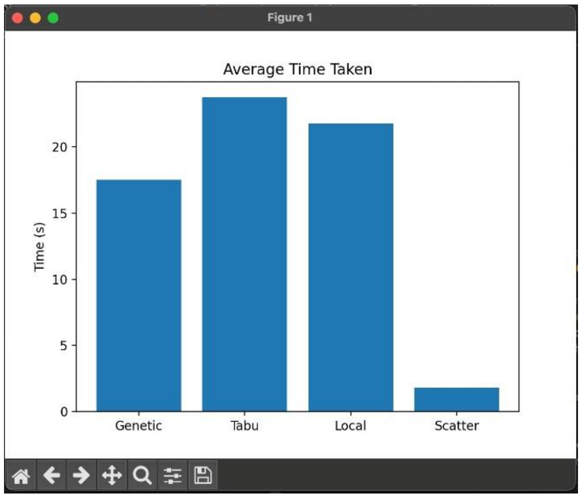


Fig 3. Average time taken for each graph

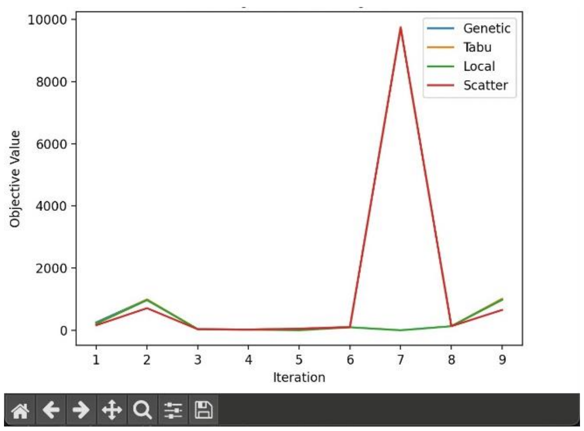


Fig 4. Convergence plot

These Above conclusions were drawn basing on the instance parameters of sizes

Genetic Algorithm: (POPULATION_SIZE = 100, NUM_GENERATIONS = 1000)

Scatter Search Algorithm: (POPULATION_SIZE = 150, NUM_GENERATIONS = 1000)

Local Search Algorithm: (NUM_ITERATIONS = 1000)

Tabu Search Algorithm: (TABU_TENURE = 10, NUM_ITERATIONS = 1000)

Similarly we obtained the results of the HD-UC, HD-WC, HD-SC we have taken 7 instances of each with no. of items in each instance varying in [100, 200, 500, 1000, 2000, 5000, 10000] and found out that overall Scatter Search Algorithm has better convergence rate and moderate runtime where as Genetic Algorithm has the near Optimal Solution.

Interestingly Tabu Search is able to find best Solution for HD-WC as well as HD-SC data but with more runtime.

The observations we made are purely based on random data generated. To avoid the high randomness we have consider correlation measure 30(min), 70(max)%.

7. Conclusions

With the data recorded, it can be concluded that the least time complexity (least execution time) for the performed experiment is recorded for Local Search. For Low Dimension data but Scatter is best for High Dimension. The algorithm which shows least deviation from the best known quality of the knapsack is Tabu Search. In future, more metaheuristic algorithms – like firefly, ant colony, GRASP can be used to compare the Knapsack Problem using more metrics, and different sample sizes. Various statistical tests can be applied to ascertain the results obtained. This research study is conducted with a primary aim of laying the groundwork for future research in the domain of combinatorial optimization.

8. References

[1] A Comparative Study of Meta-Heuristic Optimization Algorithms for 0 – 1 Knapsack Problem: Some Initial Results - Absalom E. Ezugwu, Verosha Pillay, Divyan Hirasen, Kershen Sivanarain, Melvin Govende.

[2] Solving the 0-1 Knapsack Problem with Genetic Algorithms Maya Hristakeva, Dipti Shrestha.

[3] A new class of hard problem instances for the 0–1 knapsack problem Jorik Jooken, Pieter Leyman, Patrick De Causmaecker.