1) Introduction

The bounded knapsack problem is a well-known NP complete combinatorial optimization problem in computer science and operations research that involves selecting a subset of items to include in a knapsack such that the value of the selected items is maximized and the weights of the items are less than or equal to the capacity of the knapsack. The bounded knapsack problem is a generalization of the 0-1 knapsack problem which removes the restriction that there is only one of each item but restricts the number of copies of each kind of item to a maximum non negative integer value.

The bounded knapsack problem has many real-world applications, such as stock cutting [1], cargo loading [2], resource allocation [3], investment decisions with a given budget [4], etc.

Over the years, there have been various algorithms developed to solve the bounded knapsack problem, ranging from exact methods such as dynamic programming, to heuristic approaches such as greedy algorithms, local search and more we will look at in this paper. Each algorithm has its strengths and weaknesses, and the choice of algorithm depends on the desired performance criteria.

The goal of this paper is to provide an overview of the different algorithms for solving the bounded knapsack problem. We will present the key ideas behind each algorithm, along with the advantages and limitations. We will also evaluate performance of the algorithms in terms of solution quality, running time and scalability in terms of space complexity. The analysis will provide the insights into the trade-offs between different algorithmic approaches to the bounded knapsack problem and aid in the selection of the choice of a more suitable algorithm for the specific need.

2) Formal definition

Given a knapsack of capacity $\boldsymbol{C}$ and $\boldsymbol{n}$ item types, where the value of type $\boldsymbol{j}$ has profit $\boldsymbol{p_j}$, weight $\boldsymbol{w_j}$, and a bound $\boldsymbol{b_j}$ on the availability. The goal is to select a number $\boldsymbol{x_j}$ ($0 \leq \boldsymbol{x_j} \leq \boldsymbol{b_j}$) of each item type $\boldsymbol{j}$ such that the sum of the profits of the included items is maximized without the sum of the weights exceeding $\boldsymbol{C}$.

3.1) Brute force approach

One simple way of solving the bounded knapsack problem is the brute force method, which involves generating all possible subsets of the items and computing the total value and weight of each subset. The subset with the maximum total value that satisfies the capacity constraints is then selected as the optimal solution.

However, the brute force method can become highly inefficient for large instances of the BKP. The number of possible subsets grows exponentially with the number of items, making it impractical to generate and evaluate all subsets for instances with more than a few dozen items. Let n be the number of items (all items and their copies are treated as individual items), the time and space complexity are of the order $O(2^n)$ as we will need to store every subset and also need to compute the solution for said subset. It is easy to see that this becomes highly inefficient for the reason stated above. Note that even though this approach may be inefficient for space and time considerations, it always returns an optimal solution as it compares all feasible solutions to the problem and picks the best

While the brute force approach may be useful for a small instance or as a benchmark for other algorithms, it should not be considered for larger instances. The efficiency of BKP solvers is an important criterion in practical applications such as resource allocation, portfolio optimization, etc. For better running time and space, we will see other algorithms.

Reference: [5].

3.2) Transform the BKP into an instance of the 0-1 Knapsack Problem

Since the BKP is a generalization of the 0-1 knapsack problem, we can simply work around the constraints of the BKP by making each copy of an item its own separate item. To convert the BKP to an instance of the 0-1 knapsack problem, we need to create a new binary variable for each item that represents whether it is included or not. This is in contrast to the BKP where each item can be selected a bounded number of times, and we need to create a new variable for each possible selection of an item within its bound.

We can then set the objective function to maximize the total value of the selected items, subject to the capacity constraint of the knapsack and the binary variables representing the selection of each item. The capacity constraint is the same as in the classic knapsack problem, and we can use any method for the 0-1 knapsack problem to solve the BKP. To transform the 0-1 knapsack to the BKP, we simply just check if we have items that match and increase the corresponding item counter for that item. This is done for all distinct item types and this can be done in $O(n^c)$ where c is a constant.

Although, it might seem like converting the BKP to an instance of the 0-1 knapsack problem is an easy work around to solving the BKP, there are a couple of reasons why this actually makes the problem more difficult to solve.

- many new variables are introduced by the transformation, implying that Martello and Toth, [6], couldn't solve very large data instances due to memory limitations.

- Pisinger, [7], showed that the 0-1 knapsack problem with many similarly weighted items is hard to solve as it is difficult to combine the items to obtain a knapsack that it is filled. Converting the BKP to 0-1 KP results in items of similar weight, which means the problem becomes increasingly more difficult to solve.

Although, we can use solutions for the 0-1 knapsack problem to solve the BKP, it is not advised because this method introduces complexities to the problem in terms of new variable, run time and space.

After considering the intuitive ideas, we will look at other methods of solving the BKP

Reference: [6] & [7].

3.3) **Dynamic Programming algorithm**

Dynamic programming is a commonly used method for solving optimization problems, including the bounded knapsack problem.

There are 3 dynamic programming algorithms we will consider that return an optimal solution set and its values.

The first dynamic algorithm we will label DP1 uses a nested approach to solve BKP in O(nc) time and space by extending an algorithm for the Weighted Integer Knapsack Problem to handle bounds and set up. DP1 is defined as follows:

The DP1 algorithm for the knapsack problem computes optimal solutions for different subproblems defined on the first r item types with capacity $\bar{c}$. The algorithm uses $z_r(c)$ and $x_r(\bar{c})$ to represent the optimal solution value and corresponding number of copies of item type r for a given subproblem. It also defines $\bar{z}_r(\bar{c})$ and $z_r^*(\bar{c})$ as the optimal value for different versions of the subproblem where only a limited number of copies of item type r are allowed in the knapsack.

The algorithm uses a recursive approach to compute the optimal value for each subproblem by considering three possible cases: not adding any items of type r to the knapsack, adding at least one copy of item type r to the knapsack, or adding the maximum number of copies of item type r to the knapsack. The maximum of these three cases is the optimal value for the subproblem, which is stored in the corresponding entry of the dynamic programming table.
If only the optimal value is desired then DP1 can be modified to reduce the space bound to O(n + c).

The second dynamic programming algorithm we will label DP2 solves BKP in O(nc) time and O(n + c) space by applying a storage reduction scheme using a recursive “divide

and conquer" approach [9]. To solve BKP, the set of items is divided such that the cardinality of each set is approximately equal, creating two subproblems. BSKP is solved over each subset of items given capacity 0, 1, …, c. The solutions to both subproblems are combined, and the optimal number of items of one item type from each subproblem is known. Each subproblem is then divided into two more subproblems, and this process is repeated until the optimal solution set is known. Pferschy [9] shows how this approach requires O(nc) time and O(n + c) space.

The final dynamic programming algorithm solves BKP with lists. List L_r contains a series of m entries over the first r item types

$$L_r = [(V_1, W_1, I_1), (V_2, W_2, I_2), \ldots, (V_m, W_m, I_m)],$$

where V_r denotes the value of the partial knapsack solution, W_r denotes the corresponding weight of the partial knapsack solution, and I_r is the set of items in the knapsack and their multiplicity. The dominated entries are removed, and the lists are sorted in increasing value and weight, resulting in low storage requirements. This results in algorithm DP3.

The DP3 algorithm for the BKP involves three lists: L_{r-1}, $L`_{r-1}$, and $L``_{r-1}$. $L_{r-1}$ contains the optimal solutions for the knapsack subproblems over the first r – 1 item types. $L`_{r-1}$ contains the optimal solutions for the subproblems over the first r item types, given that there are between 1 and $b_r$ – 1 copies of item type r in the knapsack. $L``_{r-1}$ contains the optimal solutions for the subproblems over the first r item types, given that there are b_r copies of item type r in the knapsack. To construct L_{r-1}, item type r is added to existing entries in L_{r-1} in increasing order of weight and value, which allows for multiple copies of each item type to be added to the partial knapsack solutions. If L_{r-1} is a linked list, item type r can be added to each entry in constant time by adding new items to the beginning of the list. The first item type (r = 1) can be added to all items in the current list as long as there is room in the knapsack. Additional items of type r = 1, 2, …, n can also be added, but the bound b_r must be checked first. The two lists can be merged in O(min{c, U}) time using pointers, and this method can be easily modified to merge three lists. [10]. The resulting algorithm requires O(n min{c, U}) time and O(n min{c, U}) space.
Reference: [8] [9] & [10]

3.4) **Fully Polynomial Time Algorithm Scheme (FPTAS)**

The fully Polynomial-Time Approximation Scheme is an algorithm technique that that takes an instance of the problem and an $\varepsilon > 0$, and produces a solution within a factor of $1+ \varepsilon$ (for minimization problems) and $1- \varepsilon$ (for maximization problems) of the optimal solution. The FPTAS method works by scaling down the weights and values of the items by a factor of ε and then applying the dynamic programming algorithm to the scaled down instance of the problem. Finally, the obtained solution is scaled back up to its original weights and values to obtain the approximate solution to the original problem. The time complexity of the FPTAS is dependent on the running time of the dynamic

programming algorithm we chose to use after scaling down the weights and values. FPTAS runs in $O(n^k(1/\varepsilon)^d)$ where k and d are constants independent of the problem instance.

Note that there is a direct correlation here with the running time and the solution accuracy. but as the section header suggests, it will always be polynomial. Therefore, the choice of ε depends on the trade off between the accuracy and speed.

For the purpose of explaining how to FPTAS is implemented, we will consider a solution to BKP with ½ solution factor that runs in **O(n)** time.

To describe the FTPAS, we define a reward, R, and a penalty, P, system such that R and P partition the item types {1, 2, …, n}.

The FPTAS uses BKP greedy heuristics to obtain solutions within a factor of ½ of the optimal solution in O(n) time.

There is a preprocessing phase and two main phases, the dynamic programming phase and the greedy phases. The preprocessing phase is called to find a lower bound on the optimal value. The set of items can then be partitioned into four subsets, Dynamic programming item types (D), Greedy item types (G), Transition Penalty item types (T_p) and Transition Reward item types (T_R).

The Dynamic Programming, Transition Penalty and Transition, Reward item types are used to form an instance of the augmented problem with $n_A = |D| + |T_P| + |T_R| <= n$ item types as follows. An item type in the augmented problem is created for each item type in D with w_i, value v_i, and bound b_i. The items are added during the dynamic programming phase with scaled values with the first copy having a different scaled value from the scaled value used for the remaining copies. An item type in the augmented problem is created for each item in T_R with their respective weights and values but all with bound 1. The item is added during the dynamic programming phase with all items scaled with the same value. An item type in the augmented tree is created for each item in T_P with their respective weights and values but with their bounds scaled. The items are added during the dynamic programming phase with scaled values such that the first item type has a different scaling from the rest of the item types. The dynamic programming algorithm DP1 solves the augmented problem using the scaled values instead of the original values.

The greedy phase inserts the Greedy, Transition Penalty and Transition Reward item types into the dynamic programming solutions. Copies of Greedy item types can be added to any DP solution. However, a Transition Penalty or Transition Reward item type can only be added to DP solution with at least one copy of that item in the knapsack. The item types considered in the greedy phase are inserted into each defined state considered

in the dynamic programming phase. A T_P or T_R item type is not considered if they are not present in the DP solution. Therefore, the greedy phase uses $O(n/\varepsilon^2)$ time to execute.

The total time and space used for the approximation algorithm is $O(n/\varepsilon^2)$ and $O(n + 1/\varepsilon^3)$, respectively. These bounds may be improved on by using techniques given by Lawler [11] and Kellerer et al. [12].

References: [11] & [12]

3.5.1) Particle Swarm Optimization algorithm

The Particle Swarm Optimization (PSO) algorithm, introduced by Russell Eberhart and James Kennedy [13], is a metaheuristic algorithm inspired by the behavior of a swarm of birds or fish. The PSO algorithm has been widely applied in optimization, intelligent computation, and design/scheduling. In this algorithm, a group of N particles represents candidate solutions. The PSO algorithm generates the initial position and velocity for each particle randomly in the search space. Every iteration, each particle updates its velocity and position based on its own best position and the global best position. The velocity of a particle is updated using a formula that includes coefficients of acceleration, random variables, and an inertia weight, while its position is updated using a simple equation. The inertia weight is linearly decreased throughout the iterations.

Reference: [13]

3.5.2) Cat Swarm Optimization

The CSO (Cat Swarm Optimization) algorithm is inspired by the hunting ability and movement patterns of cats. The algorithm is divided into two modes, seeking mode and tracing mode, which are based on two behaviors of cats: sleeping and alertness.

In seeking mode, the algorithm uses four factors: seeking memory pool (SMR), seeking ranger of selected dimension (SRD), counts of dimension to change (CDC), and self-position considering (SPC). SMR represents the size of the searching memory of each cat. Every cat will choose a new position in memory using the Roulette Wheel algorithm. SRD is used to control changes in values of the selected dimension, which do not exceed the range. CDC is used to determine the number of dimensions to change. SPC is a Boolean variable that determines whether a cat is the fittest individual or not.

In tracing mode, the algorithm updates the position of the cats based on their velocity. The velocity of the cats is updated based on the best position. The procedure of the

algorithm includes eight steps, starting with generating the initial position and velocity of N cats in the search space and ending with checking the termination criterion.

The CSO algorithm is a metaheuristic optimization algorithm that can be used for solving optimization problems in various fields.

Reference: [14]

3.5.3) Hybrid Cat-Particle Swarm Optimization (HCPSO)

The HSPCO algorithm is a hybrid of Particle Swarm Optimization (PSO) and Cat Swarm Optimization (CSO) algorithms. PSO is a metaheuristic algorithm based on a swarm of birds or fish in nature, where each particle represents a candidate solution. The algorithm generates the initial position and velocity for every particle randomly in search space. Every iteration, each particle updates their velocity and position based on their own best position and the global best position. The velocity of particles is updated using an equation that takes into account acceleration coefficients, random variables, and the inertia weight used to balance the global and local search. The position is then updated based on the new velocity.

The CSO algorithm is based on the hunting and movement instincts of cats. It is divided into two modes: seeking and tracing mode. In the seeking mode, cats choose a new position in memory using a Roulette Wheel method. They also control changes in the selected dimension values and determine the number of dimensions to change. If a cat is the fittest individual, it creates multiple copies of new candidate positions, otherwise, it creates one copy and one candidate is the current position. In the tracing mode, cats update their position based on the velocity, which is updated based on the best position.

The hybrid algorithm uses a mixture ratio to split the cats into seeking and tracing modes. The algorithm starts by generating the initial position and velocity of N cats in the search space, then evaluates the fitness value and saves the best position as the best solution. The cats are then divided into seeking and tracing modes based on the mixture ratio. In the seeking mode, cats choose a new candidate position and select the best position. In the tracing mode, cats update their position based on the velocity. The algorithm is described in eight steps in total. Santoso, K et al [15] goes into further detail.

Reference: [15]

3.5) **Novel Discrete Grey Wolf Optimizer**

Grey Wolf Optimizer

The Grey Wolf Optimizer (GWO) is an optimization algorithm that is inspired by the hunting behavior of grey wolves. The algorithm is based on a hierarchical social structure of grey wolves, with the alpha wolf at the top of the hierarchy and responsible for making decisions on hunting, resting, and advancing. The beta wolf is the candidate to the alpha wolf and superior to the delta wolf, both of which are subordinate to the alpha wolf and help in making decisions. The omega wolves are at the bottom of the hierarchy and are allowed to eat the remaining food which is left after all the wolves in other levels are finished.

In the GWO algorithm, the population consists of several grey wolves, with the optimal solution considered to be alpha, the suboptimal solutions beta and delta, and the other solutions omega. The algorithm assigns the stages of tracking, encircling, and attacking to different levels of grey wolves to complete the predatory behavior, thus realizing the process of global optimization.

The first stage of GWO is tracking, which involves the wolves tracking, chasing, and approaching the prey. During the hunting process, the wolf's tracking behavior is described mathematically using the position of the grey wolf, the position of the prey, and the distance between them. Two coefficient vectors, A and C, are estimated to control the balance between exploitation (local search) and exploration (global search).

The second stage of GWO is encircling, where the wolves surround the prey to capture it. The position of the optimal solution is unknown in evolutionary computation, so a mathematical model is established based on the characteristic that the position of the prey (the potential optimal solution) is more easily known by alpha, beta, and delta. We save the three optimal solutions X_a, X_b, and X_d, then update the position of other grey wolves using a set of equations.

The last stage of GWO is attacking, where the wolves get the best solution by attacking and capturing their prey. This process is mainly realized by decreasing the value of the coefficient vector A. When the value of A decreases from 2 linearity to 0, the corresponding value of A also varies in the interval [−a, a]. In addition, when |A| is less than 1, it indicates that the next generation position of the wolves will be closer to the position of the prey. When $1 \leq |A| \leq 2$, the wolf pack would disperse towards the direction away from the prey, causing the GWO algorithm to lose the convergence rate.

Discrete Grey Wolf Optimizer

A) Encoding Transformation:

Grey Wolf Optimizer (GWO) is used for numerical optimization, but in discrete space, the position of a grey wolf needs to be mapped to the discrete domain using a transfer function.

In the Discrete GWO (DGWO), a mapping vector Y_i is introduced as the discrete solution corresponding to the position vector X_i.

An encoding transformation function is used to convert the real number vector X_i into an integer vector Y_i using a formula.

b_j, ub, and lb are defined as parameters in the formula, where b_j is the total number of items of a certain type, and ub and lb are the upper and lower bounds of the search space, respectively.

B) Repair and Optimization

In constrained optimization problems, infeasible solutions can be generated by Evolutionary Algorithms (EAs). DGWO uses a repair and optimization method to deal with infeasible solutions. A Greedy Repair and Optimization Algorithm (GROA) is introduced to repair the infeasible solutions, which has a time complexity of O(n). GROA uses an array to store the sequence number of the density of each item, sorted by descending order.

C) Crossover

Crossover is a method used in genetic algorithms for direct communication between parent individuals and discovering better solutions. In DGWO, since there is no communication between the grey wolves, a crossover strategy is introduced as an effective method to find new solutions. The crossover operator used in DGWO is single point crossover, and the details of the operator are introduced in Mitchell, M et al [17].

D) Pseudocode of DGWO

DGWO is initialized by randomly selecting the components of the individual's position vector in the range of the upper and lower bounds. The position vector X_i is converted to a potential solution vector Y_i using the encoding transformation function. The best three individuals with the largest fitness values are selected as alpha, beta, and delta, which guide the evolution in each generation. Greedy Repair Optimization Algorithm (GROA) is used to deal with abnormal encoding individuals in each generation.

The DGWO has a time complexity of $O(n^3)$ and solution accuracy of 60% (i.e., out of 30 instances, 18 return an optimal solution).

Reference: [16]

4) Conclusion

In conclusion, the bounded knapsack problem is a well-known NP-hard optimization problem that has numerous real-world applications. In this survey paper, we have explored various algorithms that have been proposed to solve this problem efficiently.

The brute force algorithm is the simplest approach but suffers from the exponential time complexity. On the other hand, the conversion into a 0-1 knapsack instance method reduces the search space and achieves a polynomial time complexity, but may not always be the optimal solution.

Dynamic programming is a widely used approach that solves the bounded knapsack problem optimally, but has a time and space complexity that is proportional to the product of the number of items and the capacity of the knapsack.

FPTAS and are approximation methods that trade off accuracy for speed, while Discrete Grey Wolf Optimizer and Hybrid Cat-Particle Swarm Optimization algorithms are recently proposed metaheuristic approaches that have shown promising results.

Each of these algorithms has its own strengths and weaknesses, and the choice of algorithm depends on the specific problem instance and the trade-off between time complexity and solution quality.

In summary, the bounded knapsack problem remains an active area of research, and further exploration of these and other algorithms will continue to yield new insights and advancements in solving this important optimization problem.

5.) References

1. Gilmore, P.C., Gomory, R.E.: The theory and computation of knapsack functions. Oper. Res. 14(6), 1045–1074 (1966)
2. Wei, S.: A branch and bound method for the multiconstraint zero-one knapsack problem. J. Oper. Res. Soc. 30(4), 369–378 (1979)
3. Bitran, G.R., Hax, A.C.: Disaggregation and resource allocation using convex knapsack problems with bounded variables. Manag. Sci. 27(4), 431–441 (1981)
4. Pendharkar, P.C., Rodger, J.A.: Information technology capital budgeting using a knapsack problem. Int. Trans. Oper. Res. 13(4), 333–351 (2010)
5. An Exact Algorithm for the Bounded Knapsack Problem
   Authors: Yannis C. Stamatiou and Anastasios A. Economides
   Publication: INFORMS Journal on Computing, Vol. 14, No. 2, Spring 2002, pp. 139–151
6. Martello, S., &Toth, P. (1990). Knapsack problems: algorithms and computer implementations. John Wiley & Sons.
7. D. Pisinger, A minimal algorithm for the bounded knapsack problem, INFORMS Journal on Computing 12 (1) (2000) 75–82.
8. McLay, L.A., & Jacobson, S.H. (2007). Algorithms for the bounded set-up knapsack problem. European Journal of Operational Research, 184(1), 106-126.
9. U. Pferschy, Dynamic programming revisited: Improving knapsack algorithms, Computing 63 (4) (1999) 419–430.
10. H. Kellerer, U. Pferschy, Improved dynamic programming in connection with an FPTAS for the knapsack problem, Journal of Combinatorial Optimization 8 (2004) 5–11.
11. E.L. Lawler, Fast approximation algorithms for knapsack problems, Mathematics of Operations Research 4 (4) (1979) 339–356.
12. H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer-Verlag, Berlin, 2004.
13. R. Eberhart and J. Kennedy, Particle Swarm Optimization, Proceeding of the IEEE International Conference on Neural Networks, 195, vol. 4, pp. 1942-1948.
14. S. C. Chu, P. W. Tsai, and J. S. Pan, Cat Swarm Optimization, Pacific Rim International Conference on Artificial Intelligence, 2006, pp. 854-858.
15. Santoso, K. A., Kurniawan, M. B., Kamsyakawuni, A., & Riski, A. (2019). Hybrid Cat-Particle Swarm Optimization Algorithm on Bounded Knapsack Problem with Multiple Constraints. Journal of Physics: Conference Series, 1192(1), 012060.
16. Li, Z., He, Y., Li, H., Li, Y., & Guo, X. (2018). A Novel Discrete Grey Wolf Optimizer for Solving the Bounded Knapsack Problem. Complexity, 2018, 1-12.
17. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1996)