A Survey on Design Paradigms to solve 0/1 Knapsack Problem

P.Vickram, Dr. A. Sri Krishna and V.Sesha Srinivas

Abstract-This paper analyses various algorithm design strategies to solve the 0/1 Knapsack Problem.The Knapsack problem which is a maximization problem where one has to maximize the profit of objects in a knapsack without exceeding the capacity of the knapsack.The paper compares different algorithms in terms of the time, memory requirements and programming efforts.Comparison shows that the Dynamic Programming and Genetic Algorithms Strategies are more efficient comparatively with other strategies. The paper performs in depth analysis on these strategies and specifies the limitations and advantages.

Index Terms - *Multi-objective problem, C*ombinatorial Optimization, Genetic Algorithm, Evolutionary Algorithm, NP Completeness.

1. INTRODUCTION

Due to practical importance the 0/1 Knapsack Problem [1] is widely used. In last few years the generalization of this problem has been studied and many algorithms have been proposed. Evolutionary approach for solving the multi-objective 0/1 Knapsack Problem is one of them, many real worked papers found in the literature about multi-objective Knapsack Problem and about the algorithms introduced for solving them([2], [3], [4], [5]).

The knapsack problem [4] [5] is a problem in combinatorial optimization. Given a set of items, each with a cost and a value, determine the number of each item to include in a collection so that the total cost is least than a given limit and the total value is as large as possible. Consider n kind of items. 1 through n, each item i has a value P_i and a weight W_i. The maximum weight that carries the knapsack is W.

- P Vickram is currently pursuing bachelor's degree program in Computer Science engineering in RVR&JC College of Eng., Guntur, Andhra Pradesh India, PH:+91 94415 77577.
E-mail: vickrampentyala@gmail.com.
- Dr.A.Srikrishna, Professor, Dept of Information Technology, R.V.R. & J.C. College of Engineering, Guntur, Andhra Pradesh, India, PH- +91 94415 77577.
Email: atlurisrikrishna@gmail.com.
- V. Sesha Srinivas , Asst.Professor, Dept of Information Technology, R.V.R. & J.C. College of Engineering, Guntur, Andhra Pradesh, India, PH- +91 7382323001.
Email: vangipuramseshu@gmail.com

The 0/1 knapsack is a special case of the original knapsack problem in which each item of input cannot be subdivided to fill a container in which that input partially fits. The 0/1 knapsack problem restricts the number of each kind of item x_j to zero or one. Many optimization problems in decision-making can be presented as the 0-1 Knapsack Problem (KP). The 0-1 Knapsack Problem consists of loading objects in to a knapsack in such a way that the obtained total profit of all objects included in the knapsack is maximum and the sum of the weights of all packed objects does not exceed the total knapsack load capacity. Each object can be loaded or not loaded into the knapsack; this is the 0-1 decision concerning object loading. There are also other versions of this problem such as the Multi-dimensional 0-1 Knapsack Problem [6, 7–9] or the Multiple 0-1 Knapsack Problem [7–10, 11, 12]. The 0-1 Knapsack Problem does not allow the user to put multiple copies of the same items in their knapsack.

The Knapsack Problem has applications in areas such as operations research and finance. It is used in areas such as cargo packing in the airline and shipping industry. It has been referred to in various contexts as the "bin packing problem". The Knapsack problem is also well known in computer science, as it belongs to a class of problems which are NP Complete. When a problem is "NP-Complete" there is no known algorithm to solve the problem in polynomial time. It also means that if there were a polynomial time solution, all other NP problems could be reduced to the knapsack problem in polynomial time, and therefore be solved in polynomial time themselves. Hence, this type of problem has been studied carefully because of its relationship to other important problems in computer science.

THE KNAPSACK PROBLEM (KP)
The Knapsack Problem is an optimization problem. A Knapsack will have a capacity W, there will be N distinct objects(i) with weight(W_i) and profits(P_i) are to be placed into the knapsack such that the total profit is maximum and the total weights of the objects should be lesser than or equal to the capacity of the knapsack.

Xi takes the values 0 or 1. 1 specifies that the object i is considered into the Knapsack and 0 specifies that the object i is not considered into the Knapsack.

$$Maximize \sum_{i=1}^{N} P_i X_i \qquad (1)$$

Subjected to constraints:

$$\sum_{i=1}^{N} W_i X_i \leq W \qquad (2)$$

And $X_i = 0$ or 1

The organization of the paper is as follows: Section 1 discusses Knapsack problem and applications, section 2 deals with design paradigms to solve Knapsack Problem, Discussions regarding implementation of different design paradigms in section 3 and finally conclusion is given in section 4.

2.DESIGN PARADIGMS

2.1 GREEDY ALGORITHM

Greedy is an algorithm design strategy which makes a local optimal choice at each stage with the expectation of finding the global optimal solution. The heuristic search for solving the Knapsack problem is find the sorted order of objects with respect to profit weight ratio. Then consider the objects in that order by considering the constraint i.e., sum of the weights of the considered objects should be lesser than or equal to the capacity of the knapsack.

```
ALGORITHM GreedyAlgorithmKnap(Weights [1 … N], Values [1 … N])
// Input:  Array Weights is the weights of all items, Array Values is the values of all items
// Output: Array Solution indicates the items are included in the knapsack ('1') or not ('0') Integer cum Weight
Compute the value-to-weight ratios $r_i = v_i / w_i$, i = 1… N, for the items given
Sort the items in decreasing order of the value-to-weight ratios
for all items do
        if the current item on the list fits into the knapsack then
                place it in the knapsack
        else
                proceed to the next one
```

Complexity

1. Sorting by any advanced algorithm is O(NlogN)
2. $\sum_{i=1}^{n} 1$ = [1+1+1….1] (N times) = N ≈O(N)

From (1) and (2), the complexity of the greedy algorithm is, O(NlogN) + O(N)≈*O(NlogN)*. In terms of memory, this algorithm only requires a one dimensional array to record the solution string that is O(N).

2.2 DYNAMIC PROGRAMMING

Dynamic Programming is a design strategy for solving problems whose solutions satisfy recurrence relations with overlapping subproblems. Dynamic Programming simplifies a complicated problem by breaking it down into simpler sub problems in a recursive manner. It follows Principle of Optimality. The definition of Principle of Optimality is given by Richard Bellman as " An optimal solution has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal solution with regard to the state resulting from the first decision. The classical dynamic programming approach works bottom-up [15].

To design a dynamic programming algorithm for the 0/1 Knapsack problem, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller instances.

Consider an instance of the problem defined by the first *i* items, $1 \leq i \leq N$, with weights $w_1,\ldots,w_i$,values $p_1, \ldots , p_i$, and knapsack capacity j, $1 \leq j \leq$ Capacity.
Let Table [i, j] be the optimal solution of this instance (i.e. the value of the most valuable subsets of the first i items that fit into the knapsack capacity of j). Divide all the subsets of the first i items that fit the knapsack of capacity j into two categories that do not include the ith

item and subsets that include the ith item. This leads to the following recurrence

If $j < w_i$ then
Table [i, j] ← Table [i-1, j] cannot *fit* the ith item
Else
Table [i, j] ←maximum{Table [i-1, j]
Do not use the ithitem AND v_i + Table[i-1, j – v_i,]
Use the ith item

The goal is to find Table [N, Capacity] the maximal value of a subset of the knapsack. The two boundary conditions for the KP are: The knapsack has no value when there no items included in it (i.e. i = 0) Table [0, j] = 0 for j≥0

1. The knapsack has no value when its capacity is zero (i.e. j = 0), because no items can be included in it. Table [i, 0] = 0 for i≥ 0

ALGORITHM DynamicProgrammingKnap(Weights [1 … N], Values [1 … N], Table [0 ... N, 0 … Capacity])

```
// Input:        Array Weights is the weights of all items, Array Values is the values of all items, Array Table is initialized with 0s; it is used to store the results from the dynamic programming algorithm.
// Output: The last value of array Table ( Table [N, Capacity]) gives the optimal solution of the problem for a given Capacity
for i = 0 to N do
    for j = 0 to Capacity
        if j< Weights[i]   then
            Table [i, j] ← Table[i-1, j];
        else
    Table[i, j] ←maximum {Table[i-1, j] AND Values[i] + Table [i-1, j – Weights[i]];
return Table[N, Capacity];
```

The algorithm uses one array Items of type items, where each item is a structure with two fields: weight and value.
Optimal solution, is obtained using the following algorithm:

```
n ← N
    c←Capacity
    Start at position Table[n, c]
While the remaining capacity is greater than 0 do
        If Table[n, c] = Table[n-1, c] then
            Item n has not been included in the optimal solution
        Else
            Item n has been included in the optimal solution
            Process Item n
            Move one row up to n-1
            Move to column c – weight(n)
```

Complexity

$$\sum_{i=0}^{N}\sum_{j=0}^{Capacity} 1 = \sum_{i=0}^{N}[1+1+1+..+1](\text{Capacity times})$$

= Capacity * [1+1+1+……+1] (N times)
= Capacity * N
= O (N*Capacity)

Thus, the complexity of the Dynamic Programming algorithm is *O (N*Capacity).* In terms of memory, Dynamic Programming requires a two dimensional array with rows equal to the number of items and columns equal to the capacity of the knapsack. This algorithm is easiest to implement because it does not require the use of any additional structures.

2.3 BACKTRACKING

A backtracking algorithm is a recursive method for finding feasible solutions to a combinatorial optimization problem. A backtracking algorithm is an exhaustive search thatfinds all feasible solutions to find the optimal solution. Pruning methods can be used to avoid some feasible solutions that are not optimal. In a knapsack problem a problem instance consists of a list of values, P = [p_1,...,p_n]; a list of weights, W = [w_1,...,w_n]; and a capacity, W. These are all positive integers. We have to find the maximum value of $p_i x_i$ subject to $w_i x_i \leq W$ and $x_i \in \{0, 1\}$ for all i. An n-tuple [x_1, x_2,...,x_n] of 0's and 1's is a feasible solution if $w_i x_i \leq W$. One way to solve this problem is to try all 2^n possible n-tuples of 0's and 1's. Backtracking provides a simple method for generating all possible n-tuples. After each n-tuple is generated it is checked for feasibility. If it is feasible, then its profit is compared to the current best solution found to that point. The solution is updated each time when a better feasible solution is obtained.

To implement backtracking constructing a tree is a better choices known as "State Space Tree". The root is an initial state before the search for the solution begins. The nodes of the first level in the tree represent the choices for the first component and the nodes of a second level represent the choices for the second component and so on.
A node in the state space tree is promising if it corresponds to the partials constructed solution that may lead to the complete solution otherwise the nodes are called non-promising. Leaves of the tree represent either the non- promising dead end or complete solution found by the algorithm. The backtracking solution is given by a recursive

procedure named BTKnap, where the parameter "lev" shall denote the current co-ordinate (corresponding to the depth of a node in a tree) being chosen. OPTX and OPTP are passed by reference. Start with lev = OPTX = OPTP = 0.

ALGORITHM BackTrackingKnap (lev: depth of a node, OPTP: optimal profit, OPTX: optimal weight)

```
// Input: Array Weights contains the weights of all items, Array Values contains the values of all items
// Output: A state space tree with all possible
 solution nodes

Begin
    if (lev = n+1)then
        if ( P $w_i x_i \le W$ AND $\sum p_i x_i >$ OPTP )then
                OPTP = $\sum p_i x_i$;
                OPTX = X;
            end
        end
        else if
            xlev = 1 ;
            knap (lev+1, OPTP, OPTX);
        else
            xlev = 0 ;
            knap (lev+1, OPTP, OPTX);
        end
    end
```

This procedure generates all 2^n possible n-tuples in reverse lexicographic order. The complexity is O ($n \cdot 2^n$), since the second if-statement takes O (n). The recursive calls generated by Knap produce a binary tree, called the state space tree of the given problem instance.

2.4 BRANCH AND BOUND

Optimization problems like knapsack problem can be solved using Branch and Bound. It is an improvement over exhaustive search. In Branch and bound, candidate solutions one component at a time are generated and evaluates the partly constructed solutions. If no potential values of the remaining components to find a solution, the remaining components are not generated. This approach solves some large instances of difficult combinatorial problems, though, in the worst case, it still has an exponential complexity.

Branch and Bound constructs a 'state space tree'. In the context of the Knapsack problem, if there are N possible items to choose from, then the kth level represents the state where it has been decided which of the first k items have or have not been included in the knapsack. In this case, there are 2^k nodes on the kth level and the state space tree's leaves are all on level N [15]. Breadth First or Best First methods are used to traverse the state space tree in Branch and Bound. Both breath-first and best-first stop searching in a particular sub-tree, when they find the search would give no optimal solution.

Breadth First uses a regular queue whereas the Best First uses a priority queue, where both queues keep track of all currently known promising nodes.

In the state space tree, a branch going to the left indicates the inclusion of the next item while a branch to the right indicates its exclusion. In each node of the state space tree, we record the following information:

level- indicates which level is the node at, *cumValue* – the cumulative value of all items that have been selected on this branch,

cumWeight – the cumulative weight of all items that have been selected on this branch, *nodeBound*– used as a key for the priority queue.

The upper bound on the value of any subset is computed by adding the cumulative value of the items already selected in the subset, *v*, and the product of the remaining capacity of the knapsack (Capacity minus the cumulative weight of the items already selected in the subset, *w*,) and the best per unit payoff among the remaining items, which is v_{i+1} / w_{i+1} [15].

$$Upper\ Bound = v + (Capacity - w)*(v_{i+1} / w_{i+1})$$

ALGORITHM BestFirstBranchAndBoundKnap (Weights [1 … N], Values [1 … N)

```
// Input:   Array Weights contains the values of all items, Array Values contains the values of all items
// Output: An array that contains the best solution and its MaxValue
// Precondition: The items are sorted according to their value-to-weight ratios
PriorityQueue<nodeType> PQ
nodeType current, temp
Initialize the root
PQ.enqueue(the root)
MaxValue = value(root)
while (PQ is not empty)
current = PQ.GetMax()
if (current.nodeBound>MaxValue) then
    Set the left child of the current node to include the next item
if (the left child has value greater than MaxValue) then
    MaxValue = value (left child)
    Update Best Solution
if (left child bound better than MaxValue) then
    PQ.enqueue(left child)
    Set the right child of the current node not to
include the next item
if (right child bound better than MaxValue) then
    PQ.enqueue(right child)
return the best solution and it's maximum value
```

In the worst case, the branch and bound algorithm will generate all intermediate stages and all leaves. Therefore, the tree will be complete and will have $2^{n-1} – 1$ nodes, i.e. will have an exponential complexity. However, it is still better than the brute force algorithm because on average it will not generate all possible nodes (solutions). The required memory depends on the length of the priority queue.

2.5 GENETIC ALGORITHM

A genetic algorithm is a computer algorithm that searches for best solutions to a problem from among a large number of possible solutions. GAs begin with a set of solutions called population and each solution is called chromosome. From solutions of current population, a new population is generated in hope of getting a better solution.The new population is generated by applying the GA operations- Mutation, Cross-Over and Selection [16]. This process is repeated until some condition is satisfied [17]. From the population, find the best chromosome with high optimality as the result in that generation. Results of the generations are compared and if the difference in the results of any two consecutive generations is lesser than a pre specified threshold, then stop the procedure of generating new populations and can declare the chromosome with high optimality as result.

Outline of basic Genetic Algorithms are given here.

1. Start: Randomly generate a population of N chromosomes.
2. Fitness: Calculate the fitness of all chromosomes.
3. Create a new population:
    a. Selection: Randomly 2 chromosomes are selected from the population.
    b. Crossover: Perform crossover on the 2 chromosomes which are selected.
    c. Mutation: Perform mutation on the chromosomes which are obtained after performing crossover.
4. Replace: Replace the current population with the new population.
5. Test: Test whether the end condition is satisfied. If so, stop. If not, return the best solution in current population and go to step 2.

Each iteration of this process is called generation. The entire set of generations is called a run [3].

Complexity .The complexity of the genetic algorithm depends on the number of items (N) and the number of chromosomes in each generation (Size). It is *O(Size*N)*.

3. DISCUSSION

Implementation of Greedy algorithm is simple, will get only feasible solution and no guarantee that will get optimal solution its time complexity is linear with best sorting algorithm i.e. O(n) where n is the number of objects.

Implementation of Dynamic Programming, Backtracking, Branch and Bound and Genetic algorithms is little bit difficult but will get the optimal solution. Dynamic Programming results only one solution whereas the other methods result all possible optimal solutions. The time complexity of Dynamic Programming algorithm is polynomial i.e. O(n*W) where n is the number of objects and the W is the capacity of the bag. The time complexity of the other methods is exponential i.e. O (2^n) where n is the number of objects. Dynamic Programming, Branch and Bound and Genetic algorithm can be implemented as a parallel algorithm with linear complexity.

4. CONCLUSION

The comparison of the greedy, dynamic programming, backtracking, branch and bound and genetic algorithms shows that the complexities of some these algorithms are exponential, but may be applied to more suitable problems than others. The best approximation approaches for the 0/1 Knapsack Problem are dynamic programming and genetic algorithm. As, the choice between the two depends on the capacity of the knapsack and the size of the population. The dynamic programming yields only one optimal solution and GA results all the possible optimal solutions. The dynamic programming is easy and straight forward to code than genetic algorithm but the genetic algorithm can be programmed in linear time in parallel environment.

REFERENCES

[1] Gossett, Eric. Discreet Mathematics with Proof. New Jersey: Pearson Education Inc., 2003.

[2] Martello, S. and Toth, P. Knapsack problems: Algorithms and computer implementation, J. Willey and Sons, Chichester, 1990;

[3] Penn, M., Hasson, D., Avriel, M. Solving the 0/1 proportional Knapsack problem by sampling, J. optim.Theory Appl.80, 261-272, 1994;

[4] Sahni, S. Approximate algorithms for the 0/1 knapsack problem, Journal of ACM 22, 115-124, 1975;

[5] Vasquez, M., Hao, J. K. A hybrid approach for the 0/1 multidimensional knapsack problem. Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI 01) 1, 328-333, 2001;

[6] Alaya I., Solnon C., Gheira K., Ant algorithm for the multi-dimensional knapsack problem, International Conference on Bioinspired Optimization Methods and their Applications, (BIOMA 2004), 2004, 63-72.

[7] Fidanova S., Ant Colony Optimization for Multiple Knapsack Problem and Heuristic Model, Kluwer Academic Publishers, 2004.

[8] Fidanova S., Ant Colony Optimization for Multiple Knapsack Problem and Model Bias, [in:] Margenov S., Vulkov L.G., Wasniewski J. (Eds.), Numerical Analysis and Its Applications, LNCS, Vol. 3401, Springer, Berlin Heidelberg, 2005, 280-287.

[9] Fidanova S., Probabilistic Model of Ant Colony Optimization for Multiple Knapsack Problem, In Lirkov I., Margenov S., Wasniewski J. (Eds.), LSSC 2007, LNCS 4818, Berlin 2008, 545-552.

[10] Ke L., Feng Z., Ren Z., Wei X., An ant colony optimization approach for the multi-dimensional knapsack problem, Journal of Heuristics, Vol. 16, No. 1, 2010, 65-83.

[11] Shahrear I., Faizul B., Sohel R., Solving the Multidimensional Multi-choice Knapsack Problem with the Help of Ants, M. Dorigo et al. (Eds.), ANTS 2010, LNCS 6234, Berlin 2010, 312-323.

[12] Ji J., Huang Z., Liu C., Liu X., Zhong N., An Ant Colony Optimization Algorithm for Solving the Multidimensional Knapsack Problems, [in:] Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IEEE Computer Society, Los Alamitos, 2007, 10-16.

[13] Fidanova S., Heuristic for the multiple knapsack problems, IADIS International Conference on Applied Computing, 2005, 255-260.

[14] Boryczka U., Ants and Multiple Knapsack Problem, 6th International Conference on Computer Information Systems and Industrial Management Applications (CISIM'07), 2007, IEEE Computer Society, No. P2894, 2007, 149-154.

[15] Levitin, Anany. The Design and Analysis of Algorithms. New Jersey: Pearson Education Inc., 2003.

[16] Mitchell, Melanie. An Introduction to Genetic Algorithms. Massachusettss: The MIT Press, 1998.

[17]Obitko, Marek. "Basic Description." IV. Genetic Algorithm. Czech Technical University (CTU). http://cs.felk.cvut.cz/~xobitko/ga/gaintro.htmlHristakeva, Maya and DiptiShrestha. "Solving the 0/1 Knapsack Problem with Genetic Algorithms." MICS 2004 Proceedings.