Comparing between different approaches to solve the 0/1 Knapsack problem

Ameen Shaheen† and Azzam Sleit††

University of Jordan Computer Science Department, Amman, Jordan

Summary
Knapsack problem is a surely understood class of optimization problems, which tries to expand the profit of items in a knapsack without surpassing its capacity, Knapsack can be solved by several algorithms such like Greedy, dynamic programming, Branch & bound etc….
In this paper we will exhibit a relative investigation of the Greedy, dynamic programming, B&B and Genetic algorithms regarding of the complexity of time requirements, and the required programming efforts and compare the total value for each of them.
Greedy and Genetic algorithms can be used to solve the 0-1 Knapsack problem within a reasonable time complexity. The worst-case time complexity (Big-O) of both algorithms is O(N). Nevertheless, these algorithms cannot find the exact solution to the problem; they are helpful in finding a local optimal result only. Our main contribution here is to test both algorithms against well-known benchmark data sets and to measure the accuracy of the results provided by each algorithm. In other words, we will compare the best local result produced by the algorithm against the real exact optimal result.

Key words:
0/1 Knapsack, Algorithm, Greedy algorithm, dynamic programming.

1. Introduction

The 0-1 Knapsack Problem is vastly studied in importance of the real world applications that build depend it discovering the minimum inefficient approach to cut crude materials seating challenge of speculations and portfolios seating challenge of benefits for resource supported securitization, A few years ago the generalization of knapsack problem has been studied and many algorithms have been suggested [1]. Advancement Approach for settling the multi-objective0-1 Knapsack Problem is one of them, and there is numerous genuine worked papers established in the writing around 0-1 Knapsack Problem and about the algorithms for solving them.
The 0-1 KP is extremely well known and it shows up in the real life worlds with distinctive application. The solution of the 0-1 KP can be viewed as the result of a sequence of decisions [2]. 0-1 KP is NP problem (nondeterministic polynomial time) - complete and it also speculation of the 0 - 1 Knapsack problem in which numerous Knapsack are considered.
The KP is a: given an arrangement of items, each with weight and a value, decide the number of each item to include in a capacity so that the total weight is little than a given capacity and the total value must as large as possible [3].
We have n of items. Each of them has a value Vi and a weight Wi. The most extreme weight that we can convey the knapsack is C. The 0 – 1 KP is an uncommon case of the original KP problem in which each item can't be Sub separated to fill a holder in which that input part fits. The 0 – 1 KP confines the quantity of each kind of item xj to 0 or 1. Mathematically the 0 – 1 KP can be formulated as:
Maximize$\sum_{i=1}^{n}$ **Pi Xi** Subject to $\sum_{i=1}^{n}$ **Wi Xi** $\leq C$
Example:
Assume 7 numbers of items arrive as shown in table 1. We need to choose such items so that it will satisfy our two goals as follows:

1) Fill it to get the greatest benefit.
2) Knapsack holds a most extreme of 22 pounds. So the aggregate weight of the chose items not surpasses our greatest limit.

In this research, a 0/1 KP is presented. As a solution of the 0/1 knapsack problem, greedy algorithm, dynamic programming algorithm, B&B algorithm, and Genetic algorithm are applied and evaluated both analytically and experimentally in terms of time and the total value for each of them, Moreover, a comparative study of the greedy ,dynamic programming, branch and bound, and Genetic algorithms is presented.

Table 1: Knapsack Example

Items	1	2	3	4	5	6	7
Profit	10	8	9	15	7	7.2	5.5
Weight	12	8	6	16	4	5	8

The rest of this paper is organized as follows: in Section 2, gives a general view of background of knapsack problem, also presents the previous related work of the 0-1 KP and the algorithms that are used to solve it. All algorithms illustrated in Section 3. While in Section 4, analytical view of algorithm results will be presented. Moreover, the analysis involves the estimation of several performance metrics, including: the worst case time complexity. In

Manuscript received July 5, 2016
Manuscript revised July 20, 2016

Section 5, a comparison of the experimental results between the four algorithms will be shown. Finally, the conclusions will be discussed in Section 6.

2. Background and Related Work

In this part, we will introduce the 0-1 knapsack problem, and then we will present the related research work of the algorithms used to solve the knapsack problem and the comparisons done to demonstrate the differences between them. and finally the 0-1 knapsack applications.

2.1 0/1 Knapsack problem (0/1 KP)

The first appears of knapsack problem was in 1957, in two publications. The first was a paper by George Dantzig (1957); He is a creator of the field of Operations Research and a developer of linear programming. He demonstrated that the persistent of the KP, The second paper is flawlessly maximized by selecting items by bang-for-buck. [4]

KP is a well-known optimization problem, which has restriction of the value either 0 (leave it) or 1 (take it), for a given collection of items, where each has a weight and a value, that to determine the items to be included in a sets, then the total cost is less or equal to a given capacity and the total profit is as max as possible. Obviously, the items are indivisible, accordingly the problem is been called "0-1 Knapsack Problem", that because you can't derive, that mean take all value of the item or leave it.

2.2 Greedy algorithm and how to solve the problem

A greedy algorithm is a straight forward design technique, which can be used in much kind of problems. Mainly, a greedy algorithm is used to make a greedy decision, which leads to a feasible solution that is maybe an optimal solution. Clearly, a greedy algorithm can be applied on problems those have 'N' number of inputs and we have to choose a subset of these input values those satisfy some preconditions. While, this selection is been taken as a greedy decision which is hopefully leads to an optimal solution from the inputs list. Where, the next input will be chosen if it is the most input that satisfies the preconditions with minimizes or maximizes the value needed in the preconditions [5, 17].

KP can be solved by many algorithms like Greedy algorithm by select the option that look like the best at the moment and its trust the local optimal solution will lead to a global optimal solution[17], Greedy are used for optimization problems. Its typically use some heuristic knowledge to create a pool of sub optimal that hope converges to an optimum solution [6].

2.3 Dynamic programming algorithm and how to solve the problem

Dynamic algorithm is an algorithm design method, which can be used, when the problem breaks down into simpler sub problems; it solves problems that display the properties of overlapping sub problems. In general, to solve a problem, it's solved each sub problems individually, then join all of the sub solutions to get an optimal solution [13, 15].

The dynamic algorithm solve each sub problem individually, once the solution to a given sub problem has been computed, it will be stored in the memory, since the next time the same solution is needed, it's simply looked up. Distinctly, a Dynamic algorithm guarantees an optimal solution.

Here are two key traits that the problem must have all together for dynamic programming to apply: the first one is an overlapping sub problems and the second is an optimal substructure. The Overlapping sub problems is mean: the space of sub problems should be small, that is any recursive algorithm solving the problem should solve the same sub problems recursively [18], rather than creating new sub problems, and the optimal substructures mean: the solution to a given optimization problem can be acquired by the mix of optimal solutions to its sub problems.

Dynamic Programming algorithm was created by Richard Bellman which whose the term dynamic programming in 1957 [12], the authors were solves problems by consolidating the solutions for problems that contain sub-problems but notice that there are a distinction between Dynamic programming and Divide &Conquer, Divide & Conquer are solving sub-sub-problems many times but DP it solve the each sub - problem one time and store the solution's in a table [6].

Also [7] has solve the problem by two new algorithms recently proved to outperform all previous methods for the exact solution of the 0-1 Knapsack Problem by Dynamic Programming and Strong Bounds algorithms.

2.4 Branch & bound algorithm

In fact, Branch & bound is a well-known technique that is mainly used to solve the problem which categorized as optimization problems [14]. Actually, it is an improvement over exhaustive search that because B&B builds applicant solutions as one part at a time and assesses the built arrangements as unmistakable parts. From other side, in the event that there are no potential estimations of the remaining parts, which can give the solution, as a result, the remaining parts will not be created at all. Despite the fact that, in the worst case still has an exponential complexity, but it is may use to solve a large cases of difficult mixed problems. In [19] they use A

Branch & Bound algorithm for the KP by exhibited which can acquire either optimal or inexact solutions. A few attributes of the algorithm are talked about and computational experience is introduced. And the B& B is a credulous way to deal the 0–1 KP is to consider thusly all the 2n possible solutions X, figuring the benefit every time and monitoring the most elevated benefit discovered and the relating vector.[9]

2.5 Genetic Algorithm

In [3] they utilize the Genetic Algorithms which is computer algorithm that look for good solution for a problem from among huge arrangements of possible solutions. Also they proposed and created in the 1960s by John Holland, his understudies, and his colleagues at the University of Michigan. These computational standards were enlivened by the mechanics of characteristic advancement, including survival of the fittest, generation, and transformation. These mechanics are appropriate to determine an assortment of commonsense problems, including computational problems, in numerous fields [15, 16]. A few utilization's of Genetic Algorithms are streamlining, economics, machine learning, and social framework.

2.6 0/1 Knapsack applications

There are many applications for 0-1 Knapsack like cryptography for public key encryption. Different spaces where the problem shows up are: budget control, network flow, journals for a library A good overview of the early applications is located in [2].

3. Algorithms for Solving 0/1 knapsack problem

In this section the Greedy, dynamic programming, B&B and Genetic algorithms will be presented.

3.1Greedy Algorithm

Greedy Algorithm for solving 0-1 knapsack problem is calculate the ratio, where a ratio between the inputs values and the inputs weights will be calculated and according to this value the next input will be chosen to fill the knapsack in a proper way. A greedy algorithm mainly tests all inputs according to some preconditions then arranges them in a proper order to maximize or minimize the value of the required solution. Next, it starts to choose the most appropriate input that will lead to an optimal solution. (See Figure 1). The following is an illustration of the greedy programming for 0/1 knapsack problem supported with an example that applied the algorithm on knapsack the solution of knapsack problem is achieved according to the following steps:

Step 1 calculates the ratio for n inputs between weight and Benefit as (Benefit /weight).

Step 2 arranges the items in a non-diminishing order as per of the ratio value.

Step 3 picks the biggest ratio of the item that its weight is not exactly or equivalent to the W (knapsack limit) to add it to the arrangement vector.

Where n the number of input items, W the knapsack capacity, Weight [] is an array holds the weights of the items, Benefit [] holds the benefit values of the items, Ratio [] holds the value comes from dividing the items value by it weight and finally, an array that holds the solution vector. As an example of how the array Weight [] and Benefit [] is constructed shown in Figure 1, will make the algorithm running clearer, considering a 4 input items and the knapsack capacity is 50.

1) Computing the Ratio[i] of all items (lines 5-7 in Figure 1) by dividing the Benefit[i] by the weight[i].
2) Sorting the items according to the Ratio value is done in (lines 8-13).
3) Pick the largest value of the ratio that its Weight[] does not exceed the knapsack capacity (50 in this example)
4) Experience every one of the items and check which item can be fit in the knapsack to get the most extreme benefit but the total weights are not exactly or equivalent to the knapsack limit.

Based on an example of knapsack, greedy algorithm Solution steps will be as follows:

1) Calculate the ratio between the items values and weights by dividing the item’s value by the item’s weight (Table 2).
2) Arranges the items in a non-diminishing order according to their ratio (Table 3) .
3) Pick up the largest value, which stands at the top of the array. Thus, the array will be as Table 4.

Now, the first item will be chosen which has a weight as 3 which is less than the available knapsack capacity 5, so still there is another chance to get more items. In the next iteration, the capacity will be 2 instead of 5 because the first item filled it with its weight which equals 3.

4) Another time the first item which has the largest ratio and its weight is equal to the capacity reminds from the first iteration will be chosen and added to the knapsack. Then, the knapsack is full with a value equals to 9.
5) Finally,
   a) The knapsack will be filled by two items
   b) The knapsack value is 9
   c) The array will hold the rest of the items as Table 5.

Table 2: Ratio between the items

Item.NO(i)	***Weight of (i)***	***Value of (i)***	***Value(i)***
1	2	3	1
2	3	7	2
3	4	2	0
4	5	9	1

Table 3: Arranges the items

Item.NO(i)	***Weight of (i)***	***Value of (i)***	***Value(i)***
1	3	7	2
2	2	3	1
3	5	9	1
4	4	2	0

Table 4: Pick up largest value

Item.NO(i)	***Weight of (i)***	***Value of (i)***	***Value(i)***
1	2	3	1
2	5	9	1
3	4	2	0

Table 5: rest of the items

Item.NO(i)	***Weight of (i)***	***Value of (i)***	***Value(i)***
2	5	9	1
3	4	2	0

3.2 Dynamic programming algorithm

The DB is an algorithm for solving the problems that categorize as optimization problems, the main idea is to calculate the solutions to the sub-problems for one time and store the solutions in a table, so that it can be reused in future like: (See Figure 2)

1) Characterize the structure of an optimal solution by derive the problem into small problems, and look about a connection between the structure of the optimal solution of the first problem(Original) and the solutions of the smaller problems.
2) Define the optimal solution Recursively by express the solution of the first (original) problem in terms of optimal solutions for smaller problems.
3) Calculate the value of an optimal solution in a bottom-up approach by using a table.
4) Construct an optimal solution from computed information.

As an example about how DB solves the KP, suppose there is an item with weight and value as Table 6 and with capacity (W) of 10.DP algorithm will generate a matrix that holds all solutions as table 7. And the final output it will be V (4, 10) = 90.

Table 6: Knapsack Example in DP

Item	01	02	03	04
Profit	10	40	30	50
Weight	5	4	6	3

Table 7: Dynamic Programming Matrix

V(i,w)	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Greedy algorithm for solving knapsack problem (Weight [] and Benefit [])
Input:
1. Array for Weight, which holds the weight of all items.
2. Array for Benefit, which holds the Benefit of all items.
3. Capacity.
Output: Array Solution, which holds the items that its weight does not surpass the knapsack and it had the maximum amount of value

A. Calculate the ratio[1…n]
1. n is the number of input items;
2. W is the knapsack capacity;
3. Weight [i] holds the weight of i^{th} item;
4. Benefit [i] holds the Benefit of the i^{th} item;
5. Calculate the Ratio value for each item :
6. for all input items do
7. Ratio[i] = Benefit[i]/weight[i];

B. Sort the items in a non-decreasing order according the items ratio value

8. $for(i = n - 2; i >= 0; i - -)$
9. $for(j = 0; j <= i; j + +)$
10. $if(Benefit[j] < Benefit[j + 1])$
11. Swap (Benefit [j+1] , Benefit [j]) ;
12. Swap (Weight[j+1] , Weight [j]) ;
13. Swap (Ratio[j+1] , Ratio [j]) ;

C. Knapsack algorithm (find the Knapsack solution)
14. $for\ i = 1;\ i < n;\ i + +\ )$
15. $if\ (Weight[i] < W)\ then$
16. $amount = Weight[i]$;
17. $max\_value = max\_value + Benefit[i]$;
18. $Solution[i] = amount$;
19. else
20. $amount = knapsack$;
21. $solution[j + +] = amount$;
22. break;
23. $W = W - amount$;

Fig. 1 Greedy Algorithm for 0/1 KP

3.3Branch & Bound algorithm

This section presents the branch & bound Algorithm for solving the 0-1 knapsack problem .branch & bound is a technique that is used to solve the problems that categorized as optimization problems.

```
Dynamic Programing for solving knapsack problem
Input:
1. Array of Value (v).
2. Array of Weights (w).
3. Number of items(n)
4. capacity(W)

DP(w.v.W){
for i = 0 to W do
   m[0,i] = 0
end for

for i = 1 to n do
   for j = 0 to W do
      if w[i] ≤ j then
        m[i,j] = max (m[i-1,j],m[i-1,j-w[i]] + v[i])
      else
        m[i,j] = m[i-1,j]
      end if
   end for
end for
}
Return Max Value
```

Fig. 2 Dynamic Programming for 0/1 KP.

```
Branch & Bound Pseudo code
Input:
Array of Weights and array of values
Output:
Max Value
Note: Items are sorted according to value/weight ratios
Queue Q
Node Type: current, temporary
*Create the root
Q.enqueue(root)
Max Value = value
While (Q is not empty)
   current = PQ.GetMax()
 if (current >MaxValue)
    Then Set the left child of the current node to include
the next item8
If child.Left value is greater than MaxValue
MaxValue = Value of the Left Child
End if
If child.left bound better than MaxValue
Q.enqueue(Left Child)
End if
If child.Right bound better than MaxValue
Q.enqueue(Right Child)
End if
Return Best solution
```

Fig. 3 Branch and Bound Algorithm for 0/1 KP.

It is a changeover comprehensive search, on the grounds that not at all like it, branch & bound builds hopeful arrangements one part at a time and assesses the somewhat developed solutions. On the off chance that no potential estimations of the remaining parts can prompt a solution [8], the remaining segments are not created. This methodology makes it conceivable to settle some huge occasions of troublesome difficult combinatorial problems, however, in the most pessimistic scenario; regardless it has an exponential complexity.

B & B is based on state space tree. The state space tree is a root of the tree where every level represent to a decision in the solution space that relies on the upper level and any conceivable solution is represented to by a few ways beginning at the root and finishing with a leaf.

The root stayed in level 0 and represents the state where no incomplete solution has been made. A leaf has no youngsters and represents the state where all decisions making up an answer have been made.

The most well-known way, branch & bound uses to cross the state space tree, are best first traversal. This quits looking in a specific sub-tree when it is clear that to seek further down is pointless and it utilizes a customary line. In the state space tree, a branch is heading off to one side demonstrates the consideration of the following thing while a branch to the privilege shows its avoidance.

In every node of the tree, we were record the accompanying data:

Level ,cum Value , cum Weight, node Bound and the upper bound on the estimation of any subset by including the aggregate estimation of the items officially chose in the subset, v, and the result of the remaining limit of the knapsack and the best per unit result among the remaining items, which is vi+1/wi+1.(See Figure 3)

Upper Bound = v + (vi +1 / wi+1)*(C – w)

In the worst scenario, B&B algorithm will create every moderate stage and all leaves. Hence, the tree will be finished and will have 2 n-1 – 1 node.

Example:

The operation of the algorithm is illustrated with the following example. Consider a problem with seven items whose weight and values are given as table 8.

The total allowable weight in the load W = 100. A preliminary test reveals that the problem possesses a nonempty feasible solution and is not trivial, and $\sum$ wi> 100. We compute the ratios vi/wi and reorder the items. They are given below with the new indexing as Table 9.

Table 8: Knapsack Example in B&B

Item No	*Weight*	*Value*
1	40	40
2	50	60
3	30	10
4	10	10
5	10	3
6	40	20
7	30	60

Table 9: compute the ratios with new Indexing

New Index	*Item No*	*Weight*	*Value*	*Ratio*
1	7	30	60	2
2	2	50	60	6/5
3	1	40	40	1
4	4	10	10	1
5	6	40	20	½
6	3	30	10	1/3
7	5	10	3	3/10

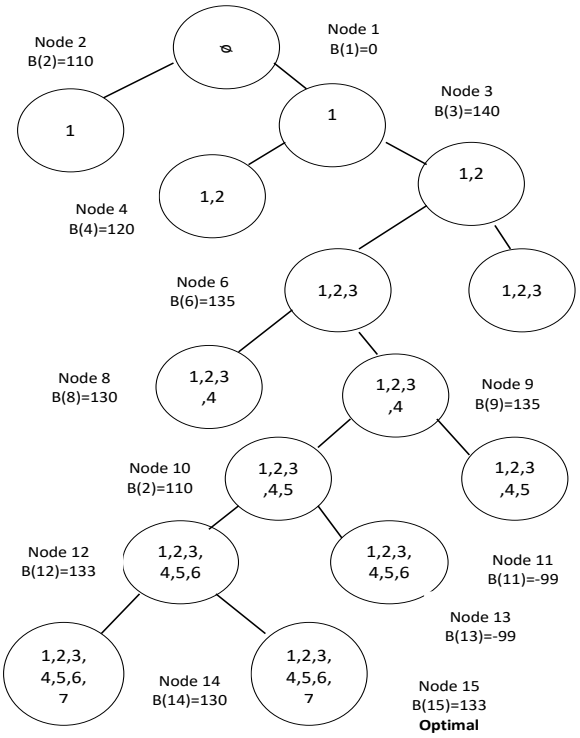

Fig. 4 Branch and Bound Tree for 0/1 KP.

The first node shown in Figure 2 is that including all possible solutions. The first branching uses index 1 as the first pivot and at node 2 where this index is excluded from the solution the upper bound is computed by:

B (2) = v2 + v3 + v4 = 110

While at node 3 where this index is included, we obtain

B (3) = v1 + v2 + ½ v3 = 140

As B (3) is the maximum upper bound, the next branching is made at node 3 and index 2 is selected as the pivot. The results of the repeated application of the algorithm are given in Figure 3. The optimum is reached at node 15. The total value being 133 and is attained by loading items 7, 2, 4, and 5. (See Figure 4)

3.4 Genetic Algorithms

Genetic Algorithms which is computer algorithm that looks for good solution for a problem from among huge arrangements of possible solutions. They were proposed and developed in the 1960s by John Holland, his students, and his colleagues at the University of Michigan. These computational paradigms were inspired by the mechanics of natural evolution, including survival of the fittest, reproduction, and mutation. These mechanics are well suited to resolve a variety of practical problems, including computational problems, in many fields. Some applications of GAs are optimization, automatic programming, machine learning, economics, immune systems, population genetic, and social system [1].

The main idea of Gas an arrangement of applicant solutions (chromosomes) called population. A new population is generated from an old population in any expectation of getting a better Solution. Solutions which were selected to form new solutions (offspring) are chosen according to their fitness. The more suitable the solutions are the greater chances they need to replicate. This procedure is rehashed until some condition is fulfilled [19]. Most GAs methods are based on the following elements, populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring.

The function that introduces the array chromosomes has an O (N). Crossover, fitness and mutation functions have also O (N). The two selection functions and the function that checks for the terminating condition do not depend on N and they have constant times of running O (1).and the aggregate complexity of the program is O(N).

Example:

Utilize an information structure, called cell, with two fields benefit and volume to represent all items. At that point we utilize an array of sort cell to store all items in it, which looks as table 10.

A chromosome can be represented in the knapsack by (‘1’) or not (‘0’). For instance, look to table 11, that Indicates the first and fourth item are included in the knapsack.

At that point we ascertain the fitness of every chromosome by summing up the profit of the things that are incorporated into the knapsack, while ensuring that the limit of the knapsack is not surpassed. On the off chance that the volume of the chromosome is more noteworthy than the limit of the backpack then one of the bits in the chromosome whose worth is "1" is transformed and the chromosome is checked once more.

For the usage of the group selection method, they utilize another array indexes size (table 12), where we put the indexes of the elements in the array fitness Size.

Now, Sort the array in slipping request as indicated by the fitness of the relating elements in the array fitness (table13). Thus, the indexes of the chromosomes with grater fitness values will be at the beginning of the array indexes, and the ones with little fitness will be towards the end of the array.

Now, divide the array into four groups:

1) 0 – 2 (0 ……. Size / 4)
2) 3 – 5 (Size / 4 ……. Size / 2)
3) 6 – 8 (Size / 2 …… 3* Size / 4)
4) 9 – 11 (3* Size / 4 …… Size)

Arbitrarily pick an item from the 1st group with 50%, from the 2ed group with 30%, from the 3ed group gathering with 15%, and from the last group gathering with 5%. In this way, the fitter a chromosome is the more risk it must be decided for a guardian in the next generation as table 14.

Table 10: Sort cell to store all items

Items	**0**	**1**	**2**	**3**
Benefit|vol	**20 | 30**	**5 | 10**	**10 | 20**	**40 | 50**

Table 11: Chromosome

items	**0**	**1**	**2**	**3**
chromosome	**1**	**0**	**0**	**1**

4. Analytical Modeling

The analytical study will be presented in this section,

Table 12: Selection Method

items	0	1	2	3	4	5	6	7	8	9	10	11
Chr fitness	40	20	5	1	9	7	38	27	16	19	11	3
indexes	0	1	2	3	4	5	6	7	8	9	10	11

Table 13: Fitness Array

items	0	1	2	3	4	5	6	7	8	9	10	11
indexes	0	6	7	1	9	8	10	4	5	2	11	3

Table 14: Generation Method

Population Size	**Group Selection Method**		
	No of Gen	**Max fit found**	**Items chosen**
100	39	3825	1,2,3,4,5,7,9,12
200	51	4310	1,2,3,4,5,6,7,8,11
300	53	4315	1,2,3,4,5,6,7,8,10
400	49	4320	1,2,3,4,5,6,7,8,9
500	65	4320	1,2,3,4,5,6,7,8,9
750	45	4320	1,2,3,4,5,6,7,8,9
1000	53	4320	1,2,3,4,5,6,7,8,9

the most vital metrics to evaluate the efficiency of The greedy, dynamic programming, branch and bound, and Genetic algorithms for solving the 0-1 knapsack problem, which will be used to show their effect on the 0/1 knapsack problem. These analyses include the following parameters: execution time and their efficiency to get the max benefit into the knapsack.

Execution time:

The execution time metric measures to what extent do the algorithm take to be finished. Time complexity analyses to get an estimated of the time required in the worst case to solve the 0/1 knapsack problem as a function of input data size. The execution time assumes a huge part in enhancing the systems performance. In this manner, the target of any algorithm solving KP is to perform productive efficient solution in the minimum possible time.

1) Greedy algorithm

The complexity time for greedy algorithm execution time will be as:

1. Sorting by Merge sort algorithm is O(NlogN)
2. $\sum_{i=0}^{n} 1 = n - 0$ is O(N)

From 1 and 2, the total complexity is O (NlogN) + O(n) which approximately equal O(NlogN).

2) Dynamic programming algorithm

The worst case time complexity of the dynamic programming algorithm used to solve the 0-1 KP is O $(W * n)$.

3) Branch and bound algorithm

In the worst case, the B&B algorithm will generate all intermediate stages and all leaves. Therefore, the tree will be complete then the Time complexity = O (2n).

4) Genetic algorithm

The function for introduces the array chromosomes has an O (N). Crossover, fitness and mutation functions have O (N). The two selection functions have O (1).The function that checks for the terminating condition has O (1). Then the total complexity of the program is O (N).

Table 15 shows the time complexity for the four algorithms.

Table 15: Time Complexity

Metric	Greedy	DP	B&B	Genetic
Ex.Time	O(NlogN)	O(W*N)	$O(2^n)$	O(N)

4. Experimental Results

The version of all algorithms presented in Section 3 (Greedy, dynamic programming, Branch and bound and Genetic Algorithm) has been coded in C++, We are test all of them using different array size but with the same Capacity size on a Core i5 1.70 GHz and 4GB Ram laptop and, we are run each Algorithms 40 time and we get the average time, we read the data set from files that we are generate with values between 1-1000 with deferent sizes, but at the beginning we test each of them in small size array to check that its work fine and gives a correct results. The experimental time that appears in table 16 is the execution time for Greedy, DP, B&B and Genetic Algorithm with different size where K mean thousand which 100K is mean 100,000 items.

Since the Branch and bound $O(2^n)$ then its need more space and in the device that we are test on, it does not work for 100000 array size so we test it just to the max number(60000) and Capacity size(100) that what we can and the experimental time for the four algorithms is shown in table 17.

From the result we can see that all of the experimental time for each algorithm are expected depend on the analytical model ,we can see that the minimal time is for genetic algorithm then Greedy, DB and B&B respectively.

The dynamic programming algorithm are always give the optimal result but the greedy and genetic algorithms are given the local optimal result, for that we are implement each of them on the same data set to compare which one that give the best local optimal result.

From the two tables (18 and 19) we can see that the genetic algorithm gives better local optimal results than the greedy algorithm not always but as an the average we can notify that the genetic local result is best than the greedy local result.

Table 16: Experimental Time

Size	Greedy	DP	B&B	Genetic
100K	0.8623	7.7177	NA	0.7325
200K	1.7758	13.9441	NA	1.4669
200K	2.8489	21.0783	NA	2.1652
400K	3.8071	28.1098	NA	2.8336
500K	4.9852	34.5011	NA	3.6795

Table 17: Experimental Time

Size	Greedy	DP	B&B	Genetic
20000	0.1325	0.2513	1.365	0.1683
30000	0.1687	0.3718	3.057	0.2251
40000	0.2041	0.4893	7.725	0.3112
50000	0.2549	0.6363	18.418	0.3552
60000	0.2943	0.7738	32.131	0.4316

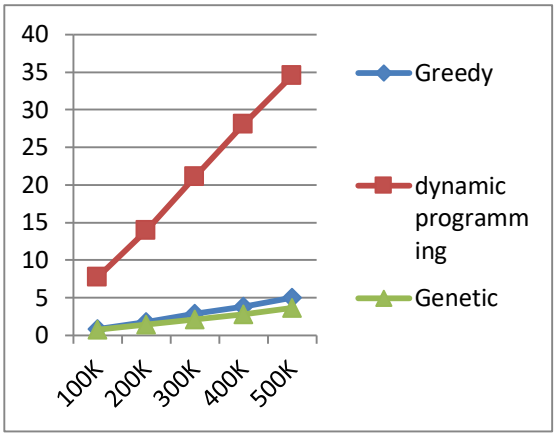

Fig. 5 . Execution time for Greedy,DP and Genetic Algos

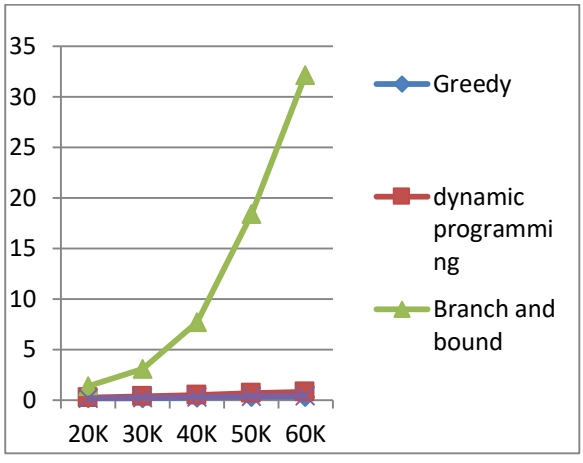

Fig. 6 . Execution time for Greedy,DP and Genetic Algos

5. Conclusion

The greedy, dynamic programming, branch and bound and genetic algorithms have been presented. The performed analysis and the conducted comparisons have been presented, and compared to the experiment results obtained from applying these algorithms on 0/1 knapsack problem.

Table 18: Greedy algorithm V.s Genetic algorithm

Data Size	Greedy	Genetic	Dynamic programming
100	224	239	239
200	328	328	334
300	417	416	417
400	477	477	477
500	510	510	510
600	529	544	544
700	555	555	563
800	555	555	563
900	572	572	581
1000	572	572	581

Table 19: Greedy algorithm V.s Genetic algorithm

Data Size	Greedy	Genetic	Dynamic programming
100	379	387	387
200	496	510	512
300	682	682	682
400	761	771	771
500	805	816	816
600	881	887	887
700	952	945	952
800	952	952	952
900	1002	1002	1002
1000	1009	1009	1015

The results demonstrate the effectiveness of the algorithms, in terms of execution time. We can conclude that the branch and bound and dynamic programming algorithms outperform the greedy and genetic algorithm in term of the total value it generated.

We used both, the greedy and the genetic algorithms in finding a local optimal result. From our experiments, it can be shown that genetic algorithms provide better results in terms of how close the results are to the real exact ones.

This is mainly because genetic algorithms allow for diversity in generating alternative solutions and they measure the fitness of these solutions at each step. In general, two factors affect the genetic algorithms accuracy. First, the possibility of representing the problem in a manner suitable for genetic algorithms evaluation and second the accuracy of the fitness function designed for the problem. In our research, we study the 0-1 Knapsack problem, which can be easily mapped to the genetic algorithm context. Also, the better parameters used (such as the number of chromosomes, crossover, mutation, and other population characteristics etc…), a more accurate output can be assumed.

The worst execution time is suffered by the branch and bound algorithm, since its complexity grows exponentially. Also if we increase the capacity of knapsack over the input items the execution time needed by dynamic greater than greedy algorithm.

The best execution time is suffered by genetic and Greedy algorithms since its complexity grows is O (n).

References

[1] S. Mohanty, R. Satapathy, “An evolutionary multiobjective genetic algorithm to solve 0/1 Knapsack Problem,” IEEE Transl. Beijing, vol. 2, pp. 397–399, August 2009.

[2] M. Babaiof, M. Babaiof, D. Kempe “A Knapsack Secretary Problem with Applications,” Springer, NJ. USA, vol. 7, pp. 16–28, August 2007.

[3] M. Hristakeva, D. Shrestha, “Solving the 0-1 Knapsack Problem with Genetic Algorithms,” IEEE Transl. Beijing, Science & Math Undergraduate Research Symposium, Indianola, Iowa, Simpson College June 2004.

[4] J. Bartholdi, “The Knapsack Problem,” Springer. Ch2, Georgia Institute of Technology, 2010.

[5] A. Sleit, S. Abusharkh, R. Etoom, Y. Khero “An enhanced semi-blind DWT–SVD-based watermarking technique for digital images,” The Imaging Science Journal, vol. 60, pp. 29–38, Novemper 2013.

[6] M. Hristakeva, D. Shrestha, “Different Approaches to Solve the 0/1 Knapsack Problem,” Proc. of 38th Midwest Instruction and Computing Symposium, Apr. 2005.

[7] S. Martello, D. Pisinger, P. Toth “Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem,” Management Science, vol. 45, pp. 414–424, 1999.

[8] A. Sleit, F. Fotouhi, S. Hasan, “The SB+-Tree: An Efficient Index Structure for Joining Spatial Relations,” International Journal of Geographical Information Science., vol, 2pp. 163–182, 1997.

[9] M.Lagoudakis, “The 0–1 knapsack problem—an introductory survey citeseer.nj.nec.com/151553.html, 1996.

[10] D. Pisinger,”algorithms for knapsack probmem” University of Copenhagen, P.h.D thesis, 1995.

[11] S. Martello, P. Toth, “Upper bound and algorithms for hard 0-1 knapsack problems,” Oper. Res, vol. 45, pp. 768–778, 1997.

[12] A. Kleywegt, D.Papastavrou, “The Dynamic and Stochastic knapsack Problem,” Opns. Res, pp. 17–35, 1998.

[13] Bellman, R.E. & Dreyfus, “Applied dynamic programming,” Princeton University Press, pp. 27–31, August 1962.

[14] E. Ignall, L. Schrage, “Application of the Branch and Bound Techniques to Some Flow Shop Scheduling Problems,” Operations Research, vol.13, pp. 400–412, 1965.

[15] B. Hammo, A. Sleit, R. Etoom, “effectiveness of query expansion in searching the holy quran,” The Second International Conference on Arabic Language Processing CITALA'07, Rabat, Morocco, pp. 1-10. 2007.

[16] M. Melanie, “An Introduction to Genetic Algorithms,” Massachusetts, MIT Press, 1999.

[17] G. Zäpfel, "The Knapsack Problem and Straightforward Optimization Methods," Springer. Chapter 2, Georgia Institute of Technology, 2010.
[18] A.Sleit, M. Al-Akhras, I. Juma, M. Alian"Applying Ordinal Association Rules for Cleansing Data with Missing Values," Journal of American Science. Marsland Press, pp. 52–62, 2009.
[19] P.KOLESAR, "A branch and bound algorithm for the knapsack problem. Manage," Sci, pp. 723-735, May 1967.