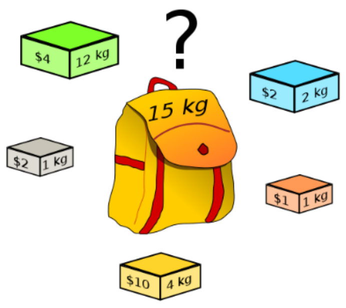
Knapsack Problem



A Project Report

Presented to
Professor Katerina Potika

Department of Computer Science
San José State University

In Partial Fulfillment

Of the Requirements for the
Class CS 255

By

Rekha Rani, Siddartha Thentu

April 2020

TABLE OF CONTENTS

I. INTRODUCTION

Decision making is a cognitive process of deciding something among many possibilities. We make a decision based on various factors like value, weight, preferences and beliefs. The result of this process is a final choice of the decision maker [1].

We will try to solve one such decision-making problem known as 0-1 knapsack problem. The 0-1 knapsack problem is a combination optimization problem to maximize the profit. This problem has various real-world applications like resource allocation with financial constraints (present Covid-19 situation). Given fixed resources and budget, how do one decide what things one should buy? Everybody want to get maximum benefit.

II. PROBLEM STATEMENT

The knapsack Problem (KP) is an NP complete problem in combinatorial optimization.

Input: Set of n items, with their weight and value

Objective: Select a subset of these items to maximize the benefit but the weight of the included items should be less than or equal to knapsack capacity [4].

0/1 knapsack problem:

- Items are indivisible; you either take an item or not.
- Number of copies of each type restricted to zero or one.
- Each Item has some weight and benefit value associated with it.

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0,1\}.$$

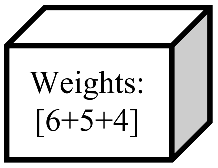
- The below example describes the knapsack problem

Items	**1**	**2**	**3**	**4**	**5**
Weight	10	5	7	4	6
Value	15	15	14	12	24

Knapsack capacity = 15

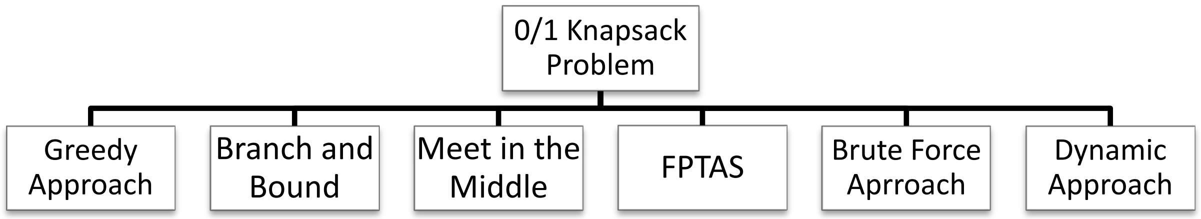
Aim: To include those weights so that total value is maximum and included weights do not exceed the knapsack capacity which is 15.

Optimal Solution:



III. TECHNICAL APPROACH/METHODOLOGY

We analyzed the following algorithm design paradigms for the 0/1 Knapsack Problem and compared them in terms of memory and time requirements.



1. Greedy Algorithm

Greedy algorithm is a simple decision technique that is based on the greedy choice. The solution obtained may or may not be the optimal solution.

Greedy Algorithm:

Input:

- Array containing the items with their weights and values

Output:

- Total weight, Value, index of items in the knapsack after including the items.

Steps:

```
1. Create a helper function to read the input file.
        a. Store the weights in an array (Weight []), store the values in an array (Value []).
        b. W is the knapsack capacity
        c. Ratio [i] = Value [i] / Weight [i]
2. Sort the Ratio array in decreasing order in an array sorted [], such that first element has
   highest value/weight ratio and last element has least value/weight ratio.
3. Find the knapsack solution:
    a. Initialize the loop variable, i= 0
    b. Loop While (included Weight < Knapsack Capacity) and i < n
        • Maximum ratio= sorted[i]
        • Find index of the maximum ratio in Ratio[] -> j
        • Find the weight and value associated with this ratio.
        • If (weight[j] + Total_weight) < Capacity
            ⬩ Total_weight = knapsack_weight + Weight [j]
            ⬩ Total_value =  Total_value + Value [j]
            ⬩ i = i+1
        • Else
            ⬩ Break
    c. Return Knapsack items, Total_weights, Total_values
```

2. Branch and Bound

It is a technique to solve mixed integer programming problems and it is based on tree search. In the worst case, we need to traverse and do calculate whole tree. But in the best case, we only need to calculate one path of tree and prune the remaining.

```
Branch & Bound Algorithm:
Input: Array containing the items with their weights and values
Output: Maximum value
Steps:
1.  Sort the items in decreasing order of value/weight ratio.
2.  Initialize Maximum_Value = 0
3.  Create a queue q → queue.Queue()
4.  Create a Root node
        Root.weight = 0
        Root.value = 0
        Insert Root to q
5.  Do while queue q is not empty:
        a.  startNode = q.get ()
        b.  if startNode.level = -1                // if it is fist node, make its level 0
                startNode.level = 0
        c.  currentNode = startNode +1.            // if the node is not the last node:
        d.  Add current level's weight and value to start node weight and value
        e.  if (currentNode.value] > Maximum_Value) & currentNode.weight <=Capacity
                Maximum_Value = currentNode.value
        f.  if (currentnode.Bound)> Maximum Value
                insert currentNode to q
        g.  Not considering currentNode as part of the solution:
                Repeat step d
        h.  if (currentnode.Bound)> Maximum Value
                insert currentNode to q
6.  Return Maximum Value
```

3. Brute force approach

If there are total n items, then there will be 2^n possible combinations of items for the knapsack.

```
Brute Force Algorithm:
Input: Array containing the items with their weights and values
Output: Maximum value
Steps:
    1.  Define BruteKnapsack()
        For each item i:
          If (Total Weight > knapsack capacity):
                Don't include the item
          Else:
        •   Create a new subset which includes item, recursively process the remaining capacity
            and item
        •   Create a new set without item i, and recursively process the remaining items
        Find maximum:
                Max(values[n] + BruteKnapsack(Capacity-weights[n] , weights , values , n-1),
                BruteKnapsack(Capacity , weights , values , n-1)
    2.  Return Maximum Value
```

4. Dynamic programming

Dynamic Programming is an optimization method that solves a problem by dividing it into smaller subproblems. Each subproblems are solved only once and the results are recorded.

```
Input: Array containing the items with their weights and values
Output: Maximum value
Steps:
    1. Identify the subproblem:
        • Items: {1…..N} , w: weight for each subset
        • subproblem: compute B[k,w] i.e best value for each subproblem
        • If weight[Nth item] > Capacity : Don't include it
        • Calculate the maximum:
          [1] Nth item: Not included
              • Max value of N-1 items and weight W.
          [2] Nth item: included
              • Value[Nth item] + Max value obtained by (N-1) items and W-weight[Nth]
    2. For (j = 0 to W):
       B[0, j] =0
    3. for (i =0 to n):
       B[i, 0] =0
            for w = 0 to W:
                if weightOfItem <= currentCapacity then: // item can be part of the
                solution
                if (B[i] + Table[i-1,w-w[i]] > Table[i-1,w])
                Table[i,w] = b[i] + Table[i-1,w- w[i]]
                Else
                Table[i,w] = Table[i-1,w]
                else Table[i,w] = Table[i-1,w] // wi > w
    4. Return Maximum value
```

5. FPTAS (Fully Polynomial Time Approximation Scheme)

When the profit values grow very big, Dynamic Programming solutions become infeasible. It becomes necessary to scale down the values while compromising for accuracy. If we round off the least significant bits of values/benefit in a given knapsack problem, then they are bounded by a polynomial and 1/ ε. **ε : bound on accuracy of result of knapsack.** So, it rounds off all the values so that they lie in smaller range. FPTAS returns optimal solution in rounded instance. FPTAS is the strongest possible result that we can derive for an NP-hard problem.

```
Input: Array containing the items with their weights and values
Output: Maximum value
Steps:
    5. Calculate the maximum value
            • maxVal = max(values)
    6. k = (maxVal * epsilon)/numOfItems
    7. for i =0 to n do:
    1. values'[i] = (values[i]/k)
    8. Return Maximum value by the use of values'[i] in the dynamic program(4th).
```

6. Verifying Optimality of Solutions

To consider the optimality of solution, we used a very small dataset. Table II contains a total of three items, and we have to maximize the benefit. **Knapsack Capacity = 5**

Items	**1**	**2**	**3**	**4**
Weight(kg)	2	1	3	2
Value($)	12	10	20	15
Value/Weight	6	10	6.6	7.5

Table II

1. Greedy Solution:

- Sort the items in descending order of ratio
- Solution: Items included= {2,4}, Total weights = 3, Total value = 25
- Optimal solution: Items included= {1,2,4}, Total weights = 5, Total value = 37

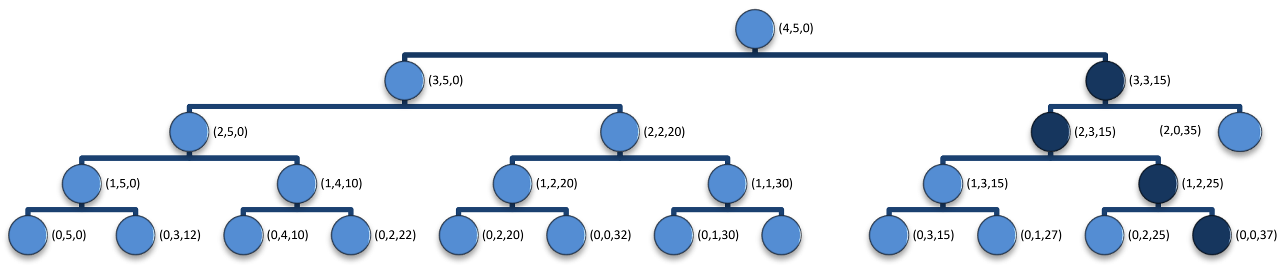
greedy algorithm fails to give optimal solution to 0/1 knapsack problem.

Items	**2**	**4**	**3**	**1**
Weight	1	2	3	2
Value	10	15	20	12
Value/Weight	10	7.5	6.6	6

2. Brute Force Solution:

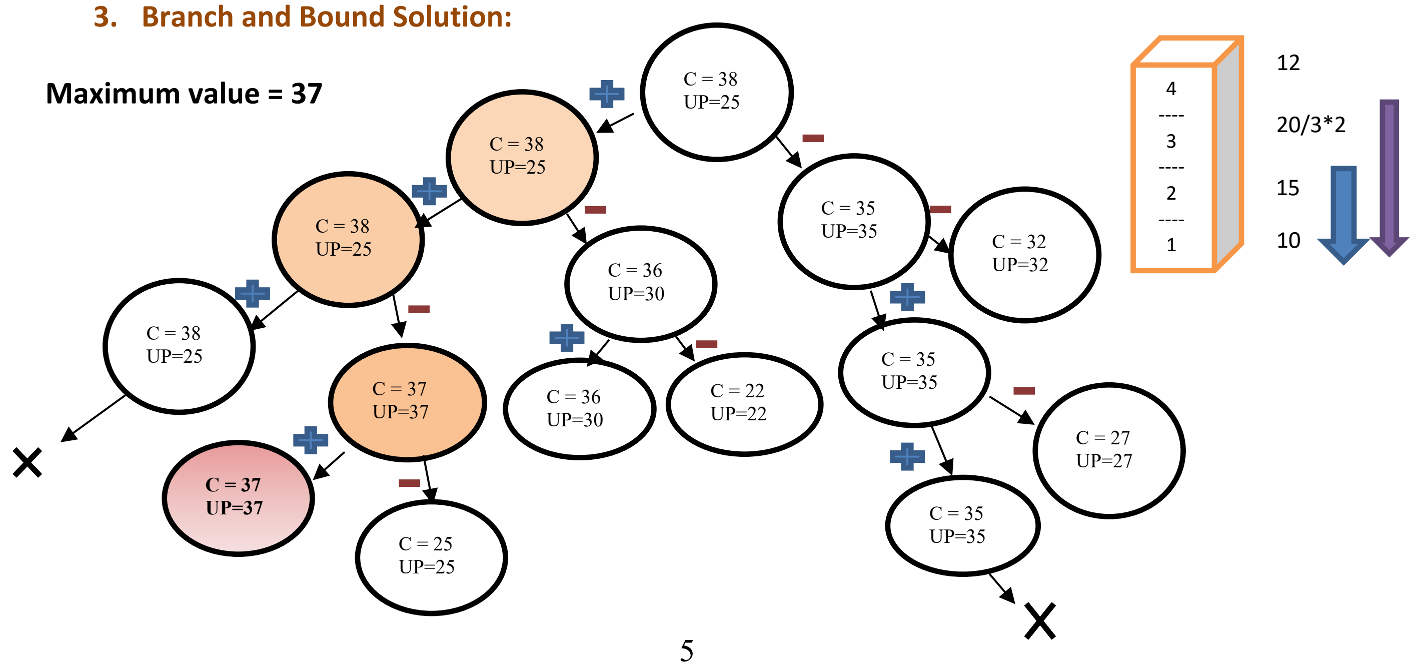
Total possibilities = $2^4 = 16$

Optimal solution: Maximum Value = 37



3. Branch and Bound Solution:

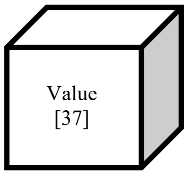
Maximum value = 37



4. Dynamic Solution:

- Create a 2D array of n + 1 rows and W + 1 columns

	0	**1**	**2**	**3**	**4**	**5**
Φ	0	0	0	0	0	0
{1}	0	0	12	12	12	12
{1,2}	0	10	12	22	22	22
{1,2,3}	0	10	12	22	30	32
{1,2,3,4}	0	10	15	25	30	37



- Total weight = 2+1+2 = 5 kg
- Total value = 12+10+15= $ 37 **Which is an optimal solution**

5. FPTAS

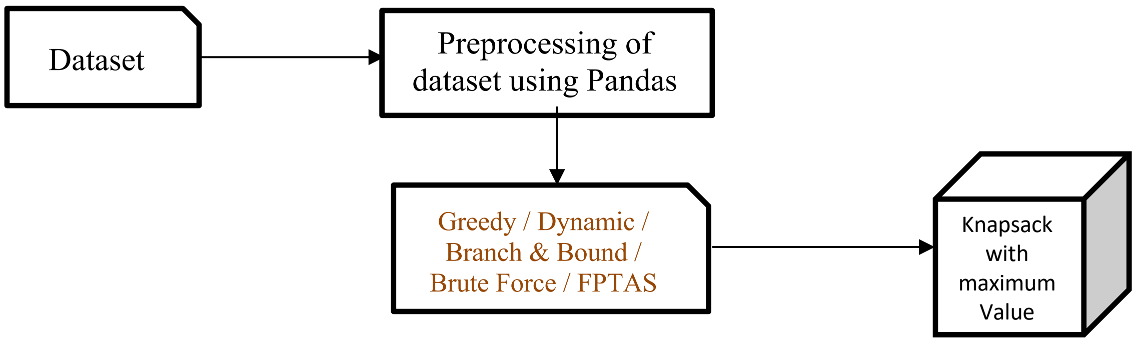
- With epsilon considered 0.5, the scaling factor for the above example becomes (20*0.5/4) = 2.5. After scaling down the values, the table becomes

Items	**1**	**2**	**3**	**4**
Weight(kg)	2	1	3	2
Value($)	4.8	4	8	6
Value/Weight	6	10	6.6	7.5

- Running a dynamic programming to the above problem yields a value of 14.8
- Returning the scaling factor (2.5) * value(14.8) yields **37** which is the optimal value.

IV. IMPLEMENTATION DETAILS

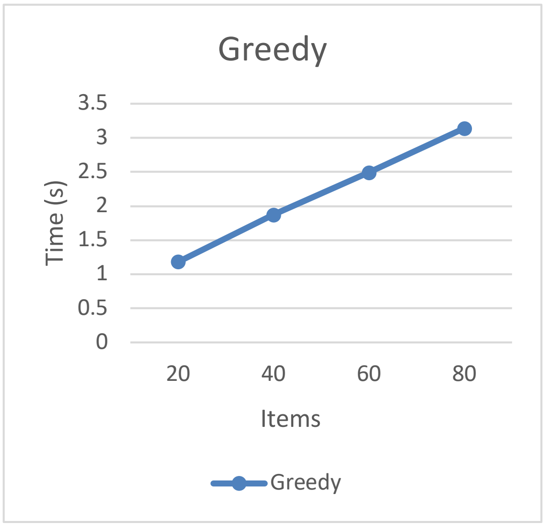
This 0/1 Knapsack problem project is implemented in python. We used OS module of python which provides the functionality for interacting with the operating system to read the file. We used the sorted.list() function of python that uses an algorithm called Timsort. Timsort is derived from merge sort and insertion sort. The execution of our implemented algorithms is in the following manner.


Dataset
Preprocessing of dataset using Pandas
Greedy / Dynamic / Branch & Bound / Brute Force / FPTAS
Knapsack with maximum Value

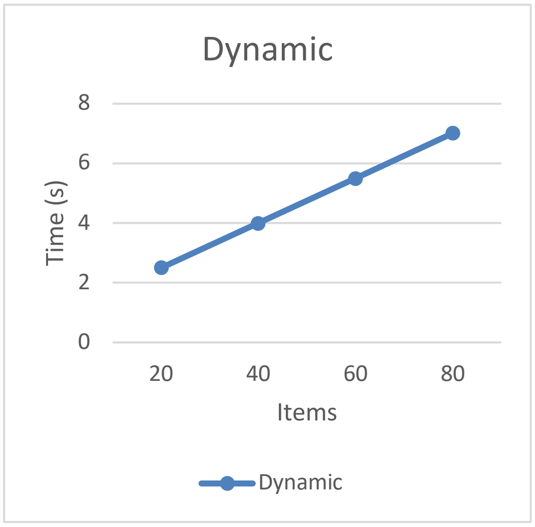
V. Experimental Results

Time: X axis, Records: Y Axis

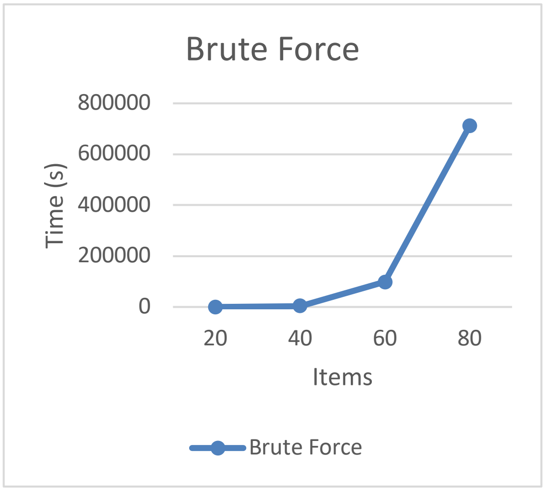
Time: t – e05 seconds



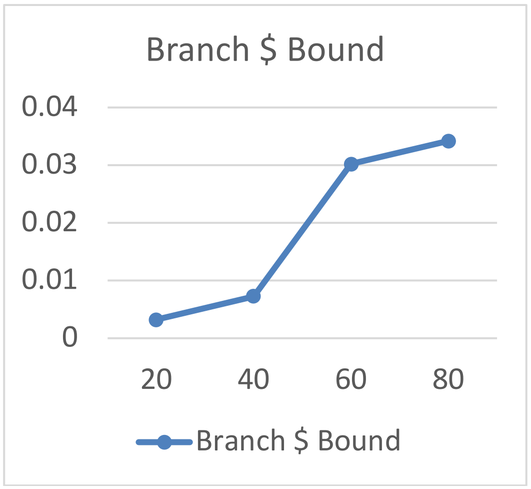
Time: t second



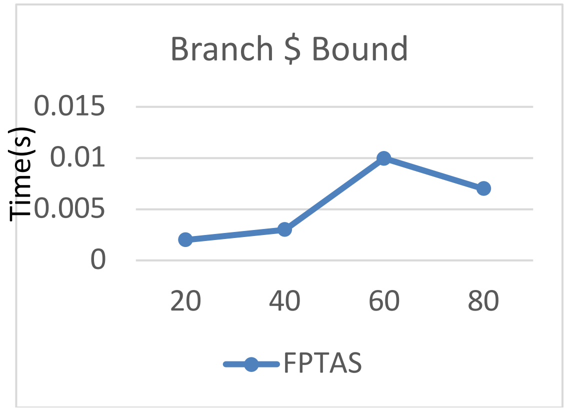
Time: t second



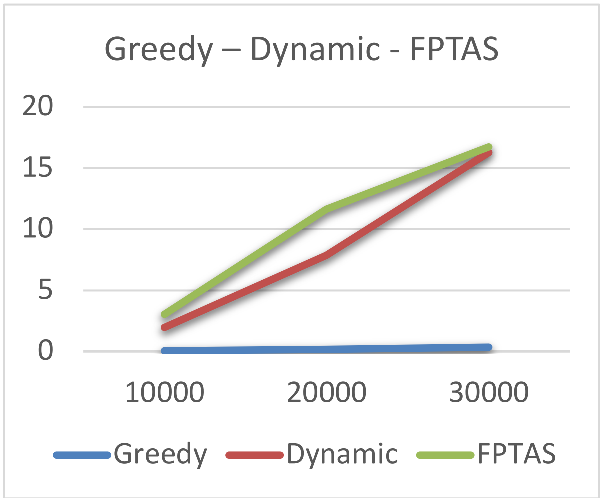
Time: t second



Time: t seconds



Time: t seconds, large records



Number of records

Number of records

0/1 KNAPSACK PROBLEM

Approach	Number of Records	Knapsack Capacity	Time Taken (s)	Values
Greedy	1. 20 2. 40 3. 60 4. 80	1. 200 2. 200 3. 200 4. 200	1. 1.18e-05 2. 1.87e-05 3. 2.49e-05 4. 3.14e-05	1. 811 2. 1220 3. 1256 4. 1400
Branch & Bound	1. 20 2. 40 3. 60 4. 80	1. 200 2. 200 3. 200 4. 200	1. 0.00323 2. 0.00726 3. 0.03022 4. 0.0342	1. 872 2. 1238 3. 1353 4. 1489
Brute Force	1. 20 2. 40 3. 60 4. 80	1. 200 2. 200 3. 200 4. 200	1. 12.229 2. 3887.27 3. 97719.62 4. 711916.61	1. 872 2. 1238 3. 1353 4. 1489
Dynamic	1. 20 2. 40 3. 60 4. 80	1. 200 2. 200 3. 200 4. 200	1. 2.534 2. 4.006 3. 5.5 4. 7.01	1. 872 2. 1238 3. 1353 4. 1489
FPTAS	1. 20 2. 40 3. 60 4. 80	1. 200 2. 200 3. 200 4. 200	1. 0.00199 2. 0.00299 3. 0.00997 4. 0.00701	1. 857.75 2. 1229.25 3. 1347.208 4. 1482.55

Approach	Number of Records	Knapsack Capacity	Time Taken	Values
Greedy	1. 10000 2. 20000 3. 30000	1. 400 2. 700 3. 1000	1. 0.0419 2. 0.1596 3. 0.3388	1. 22866 2. 42944 3. 62261
Brute Force & Branch & Bound	1. 10000 2. 20000 3. 30000	1. 400 2. 700 3. 1000	N/A	N/A
Dynamic	1. 10000 2. 20000 3. 30000	1. 400 2. 700 3. 1000	1. 1.95 2. 7.87 3. 16.27	1. 22934 2. 42981 3. 62298
FPTAS	1. 10000 2. 20000 3. 30000	1. 400 2. 700 3. 1000	1. 3.025 2. 11.606 3. 16.73	1. 22933.23 2. 42980.3 3. 62297.25

VI. TIME COMPLEXITY ANALYSIS

APPROACH	ANALYSIS	TIME COMPLEXITY
GREEDY	• Sorting – O(NLogN), Traversing Ratio Array – O(N) • Total = O(NlogN) +O(N)	**O(NLOGN)**
BRANCH $ BOUND	• In the worst case: All intermediate states generated • Complete Tree nodes = 2^{n-1}-1	**O(2^N)**
BRUTE FORCE	• A recursion tree with sub problems will have a depth of N where each problem is divided into two sub problems.	**O(2^N)**
DYNAMIC	• Nested Loops – O(W*n) • comparisons, lookups: constant	**O(W*N)**
FPTAS	• Scale the values by a scaling factor and reduce the dependency on the values.	**O(POLY(N,1/E)**

VII. Dataset Inference

To analyze the pattern of graph in dynamic and FPTAS approach, we need a large dataset. So, dataset is generated from online random number generator websites. Our dataset contains 50,000 items and each item has weight and value associated with it.

VIII. Future Research

All the algorithms discussed to solve the 0/1 knapsack problem are efficient and we are able to obtain optimal solutions from all the approaches except greedy. There are various shortcomings which needs to be resolved and other approaches like knowledge discovery algorithm should be used to along with the algorithms which we discussed to solve the knapsack problem.

IX. Conclusion

We presented the greedy, brute force, Branch & Bound, dynamic approach, and Fully Polynomial Time Approximation Scheme to solve the 0/1 knapsack problem and analyzed the time complexity for all these approaches. It was concluded that greedy algorithm running time is fastest, but greedy algorithm was not showing the optimal solution. All other approaches outperformed the greedy algorithm when we compared the total generated benefit. Brute force and Branch & bound have the worst-case time complexity because they perform an exhaustive search to get optimal solution. The dynamic programming algorithm is the best choice because we always got optimal solution. FPTAS can be used when the profit values grow very big since Dynamic Programming solutions become infeasible in that case. FPTAS scales down the values but the accuracy of solution is compromised.

X. Progression Timeline

Week 1: Feb. 17 – Feb. 23	Research about Knapsack Problem
Week 2: Feb. 24 – Mar. 1	Research about 0/1 Knapsack Problem
Week 3: Mar. 2 – Mar. 8	Deliverable #1: Project proposal
Week 4: Mar. 9 – Mar. 15	Set up programming environment
Week 5: Mar. 16 – Mar. 22	Dataset search
Week 6: Mar. 22 – Mar. 29	Implemented the brute force approach and greedy approach
Week 7: Mar 30. – Apr. 5	Stimulate the algorithm to test multiple files from dataset
Week 8: Apr. 6 – Apr. 12	Implemented dynamic approach
Week 9: Apr. 13 – Apr. 19	Time and space complexity analysis
Week 10: Apr. 20 – Apr. 26	Deliverable #2: Project Demo
Week 11: Apr. 27 – May. 4	Deliverable #3: Final presentation and report

REFERENCES

[1] https://en.wikipedia.org/wiki/Decision-making

[2] https://en.wikipedia.org/wiki/Dynamic_programming

[3] https://iopscience.iop.org/article/10.1088/1742-6596/1069/1/012024/pdf

[4] https://en.wikipedia.org/wiki/Knapsack_problem

[5] http://www.micsymposium.org/mics_2005/papers/paper102.pdf

[6] Pan, Xiaohui & Zhang, Tao. (2018). Comparison and Analysis of Algorithms for the 0/1

[7] Knapsack Problem. Journal of Physics: Conference Series. 1069. 012024. 10.1088/1742-6596/1069/1/012024.

[8] http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/

[9] M. Assi and R. A. Haraty, “A Survey of the Knapsack Problem,” *2018 International Arab Conference on Information Technology (ACIT)*, 2018.