

DSA PROJECT

MULTIDIMENSIONAL KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

Group Members :

Niharika Bhatia (17BCI0116)

Ikreet Brar (17BCI0142)

Sandhya Ananthan (17BCI0099)

ABSTRACT

This project is taken up to show that dynamic programming is a useful technique of solving certain kind of problems. The multidimensional knapsack problem can be solved with a dynamic programming approach. When the solution can be

recursively described in terms of partial solutions, we can store these partial solutions and reuse them as necessary (memorization).

Dynamic programming is both a mathematical optimization method and a computer programming method. ... Likewise, in computer science, a problem that can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems is said to have optimal substructure.

Dynamic programming has algorithm that finds solutions to sub problems and stores them in memory for later use. It is more efficient than “brute-force methods”, which solve the same sub problems over and over again.

INTRODUCTION

Knapsack problem-Given some items, pack the knapsack to get the maximum total value. Each item has some weight, volume and some value. Total weight that we can carry is no more than some fixed number W and volume V . So we must consider weights of items, volume of the items as well as their value. How to **pack** the knapsack to achieve maximum total value of packed items?

There are two versions of this problem. We will try to solve the multidimensional knapsack problem using dynamic programming where indivisible items are considered i.e. you either take an item or not.

The Multidimensional knapsack problem (MKP) has varied applications in various fields, for example economy: Consider a set of projects (variables) ($j = 1, 2, 3, \dots, n$) and a set of resources (constraints) ($i = 1, 2, 3, \dots, m$). Each project has assigned a profit c_j and resource consumption values a_{ij} . The problem is to find a subset of all projects leading to the highest possible profit and not exceeding given resource limits b_i and the volume v_i .

The MKP is a well known NP-Hard combinatorial optimization problem which can be formulated as follows

Maximize

$$\max Z = f(x_1, x_2, \dots, x_n) = \sum_{j=1}^n c_j x_j \quad (2.1)$$

subject to the constraints

$$\sum_{j=1}^n x_j \leq n$$

$$a_{ij}x_j \leq b_j, \quad i = 1, 2, \dots, m \quad a_{ij}x_j \leq v_j, \quad i = 1, 2, \dots, m \quad (2.2)$$

$$\sum_{j=1}^n$$

$$x_j \in \{0, 1\}, \quad j = 1, 2, 3, \dots, n \quad (2.3)$$

$c_j > 0, a_{ij} \geq 0, b_j \geq 0$ (2.4) The objective function $Z = f(x_1, x_2, \dots, x_n)$ should be maximized subject to the constraints given by (2.2). Here in a MKP, it is necessary that a_{ij} are positive. This necessary condition paves way for better heuristics to obtain optimal or near optimal solutions.

The popularity of knapsack problems stems from the fact that it has attracted researchers from both camps: the theoreticians as well as the practitioners enjoy the fact that these simple structured problems can be used as sub problems to solve more complicated ones. Practitioners on the other hand, enjoy the fact that these problems can model many industrial opportunities such as cutting stock, cargo loading, and processor allocation in distributed systems.

The special case of MKP with $m = 1$ is the classical knapsack problem

(01KP), whose usual statement is the following. Given a knapsack of capacity b and n objects, each being associated a profit c_j a volume occupation v_j , one wants to select k ($k \leq n$ and k not fixed) objects such that the total profit is maximized and the capacity of the knapsack is not exceeded. It is well known that 01KP is strongly NP-Hard because there are no polynomial approximation algorithms to solve it. This is not the case for the general MKP. Various algorithms to obtain exact

solutions of such problems were designed and well documented in literature.

ABOUT METHODOLOGY

Dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler sub problems, solving each of those sub problems just once, and storing their solutions – ideally, using a memory-based data structure. The next time the same sub problem occurs, instead of re computing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space.

Dynamic programming is an optimization approach that transforms a complex problem into a sequence of simpler problems; its essential characteristic is the multistage nature of the optimization procedure.

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved sub problems and will combine their solutions to give the best solution for the given problem.

ALGORITHM OF DYNAMIC PROGRAMMING

```
// WL = backpack weight limit
```

```
// SL = backpack size limit
```

```
for w = 0, 1, ..., WL // weight
```

```
for s = 0, 1, ..., SL // size
```

```
    A[0, w, s] = INFINITY
```

```
A[0, 0, 0] = 0
```

```
for i = 1, 2, ..., n // items
```

for $w = 0, 1, \dots, WL$ // weight

for $s = 0, 1, \dots, SL$ // size

// w_i, s_i and c_i are the weight, size and cost of the item respectively

// $A[i-1, w-w_i, s-s_i] + c_i = 0$ when $w_i > w$ or $s_i > s$

$A[i, w, s] = \min(A[i-1, w, s], A[i-1, w-w_i, s-s_i] + c_i)$

PROBLEM STATEMENT

A robber has robbed a museum. He has 5 different artefacts that have different values. The artefacts have more than 1 copies. The robber has to load these artefacts in a trolley which can have a maximum weight of 7 units and a maximum volume of 7 units.

What all artefacts can he carry to steal the most valuable artefacts.

ARTEFACT 1

VALUE: 1 WEIGHT: 2 VOLUME: 3

ARTEFACT 2

VALUE: 2 WEIGHT: 1 VOLUME: 4

ARTEFACT 3

VALUE: 3 WEIGHT: 3 VOLUME: 1

ARTEFACT 4

VALUE: 4 WEIGHT: 5 VOLUME: 5

ARTEFACT 5

VALUE: 5 WEIGHT: 4 VOLUME: 2

Item	I ₁	I ₂	I ₃	I ₄	I ₅
Value	1	2	3	4	5
Weight	2	1	3	5	4
Volume	3	4	1	5	2

Table :

Value	Item no.	Weight	No. of items = i weight \rightarrow	value = V	Volume = O		
			1	2	3	4	5
1	I_1	2	$V = 0$ $O = 0$ $i = 0$	$V = 1$ $O = 3$ $i = i_1$	$V = 1$ $O = 3$ $i = i_1$	$V = 2$ $O = 6$ $i = (i_1, i_1)$	$V = 2$ $O = 6$ $i = (i_1, i_1)$
2	I_2	1	$V = 2$ $O = 4$ $i = i_2$	$V = 2$ $O = 4$ $i = i_2$	$V = 3$ $O = 7$ $i = (i_1, i_2)$	$V = 3$ $O = 7$ $i = (i_1, i_2)$	$V = 3$ $O = 7$ $i = (i_1, i_2)$
3	I_3	3	$V = 2$ $O = 4$ $i = i_2$	$V = 2$ $O = 4$ $i = i_2$	$V = 3$ $O = 7$ $i = (i_1, i_2)$	$V = 5$ $O = 5$ $i = (i_2, i_3)$	$V = 6$ $O = 2$ $i = i_3, i_3$
4	I_4	5	$V = 2$ $O = 4$ $i = i_2$	$V = 2$ $O = 4$ $i = i_2$	$V = 3$ $O = 7$ $i = (i_1, i_2)$	$V = 5$ $O = 5$ $i = (i_2, i_3)$	$V = 6$ $O = 2$ $i = (i_3, i_3)$
5	I_5	4	$V = 2$ $O = 4$ $i = i_2$	$V = 2$ $O = 4$ $i = i_2$	$V = 3$ $O = 7$ $i = (i_1, i_2)$	$V = 5$ $O = 5$ $i = (i_2, i_3)$	$V = 7$ $O = 6$ $i = (i_2, i_3)$

CODE:

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
typedef struct {
```

```
char *name;
```

```
double value;
```

```
double weight;
```

```
double volume;
```

```
} itemconstraints;
```

```
itemconstraints items[] = {
```

```

{"artefact1",1,2,3},
{"artefact2",2,1,4},
{"artefact3",3,3,1},
{"artefact4",4,5,5},
{"artefact5",5,4,2},
};

int n = sizeof (items) / sizeof (itemconstraints);

int *count;

int *best;

doublebest_value;

void knapsack (inti, double value, double weight, double volume) {

int j, m1, m2, m;

if (i == n) {

if (value >best_value) {

best_value = value;

for (j = 0; j < n; j++) {

best[j] = count[j];

}

}

return;

}

m1 = weight / items[i].weight;

```



```

    m2 = volume / items[i].volume;

    m = m1 < m2 ? m1 : m2;

    for (count[i] = m; count[i] >= 0; count[i]--) {

        knapsack(

            i + 1,

            value + count[i] * items[i].value,

            weight - count[i] * items[i].weight,

            volume - count[i] * items[i].volume

        );

    }

}

int main () {

    count = malloc(n * sizeof (int));

    best = malloc(n * sizeof (int));

    best_value = 0;

    float wt, vol;

    printf("enter the max weight that can be carried");

    scanf("%f", &wt);

    printf("enter the max volume of trolley");

    scanf("%f", &vol);

```

```
knapsack(0, 0.0, wt, vol);

inti;

for (i = 0; i < n; i++) {

printf("%d %s\n", best[i], items[i].name);

}

printf("best value: %.0f\n", best_value);

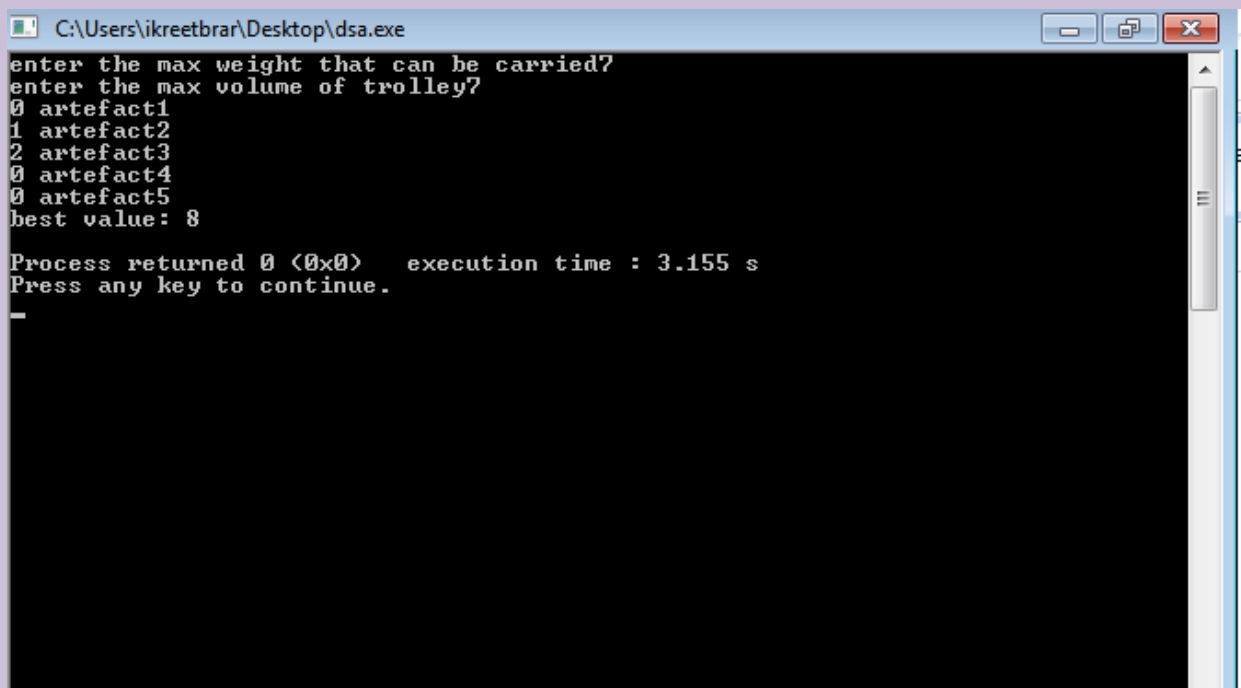
free(count);

free(best);

return 0;

}
```

OUTPUT:



```
C:\Users\ikreetbrar\Desktop\dsa.exe
enter the max weight that can be carried?
enter the max volume of trolley?
0 artefact1
1 artefact2
2 artefact3
0 artefact4
0 artefact5
best value: 8

Process returned 0 (0x0)   execution time : 3.155 s
Press any key to continue.
-
```

REAL WORLD APPLICATIONS

1. Transportation problem

Z = maximum cost savings

A_i = transportation cost saved for a given city

B_i = max no. of units that can be transported from Kanpur at once C_i = max no. of units that can be transported from Nainital at once

ANALOGY WITH KNAPSACK:

Z = maximize the profit

A_i = value

B_i = weight

C_i = volume

CODE:

```
#include<iostream>
#include<stdlib.h>
using namespace std;
typedef struct
{
    char *name;
    double cost;
    double kanpur;
    double nainital;
}constraints;
```

```
constraints items[] = {"Delhi", 1000.0, 8, 12}, {"Kashmir", 3000.0, 18,
25}, {"Chennai", 1000.0, 5, 20}, {"Allahabaad", 3000.0, 16, 11}, {"Abohar",
2000.0, 15, 11}};
```

```

int n = sizeof (items) / sizeof (constraints);
int *count;
int *max;
double max_cost;

void knapsack (int i, double cost, double kanpur, double nainital)
{
    int j, m1, m2, m;
    if (i == n)
    {
        if (cost > max_cost)
        {
            max_cost = cost;
            for (j = 0; j < n; j++)
            {
                max[j] = count[j];
            }
        }
        return;
    }
    m1 = kanpur / items[i].kanpur;
    m2 = nainital / items[i].nainital;
    m = m1 < m2 ? m1 : m2;
    for (count[i] = m; count[i] >= 0; count[i]--)
    {
        knapsack( i + 1, cost + count[i] * items[i].cost, kanpur - count[i] *
            items[i].kanpur, nainital - count[i] * items[i].nainital);
    }
}

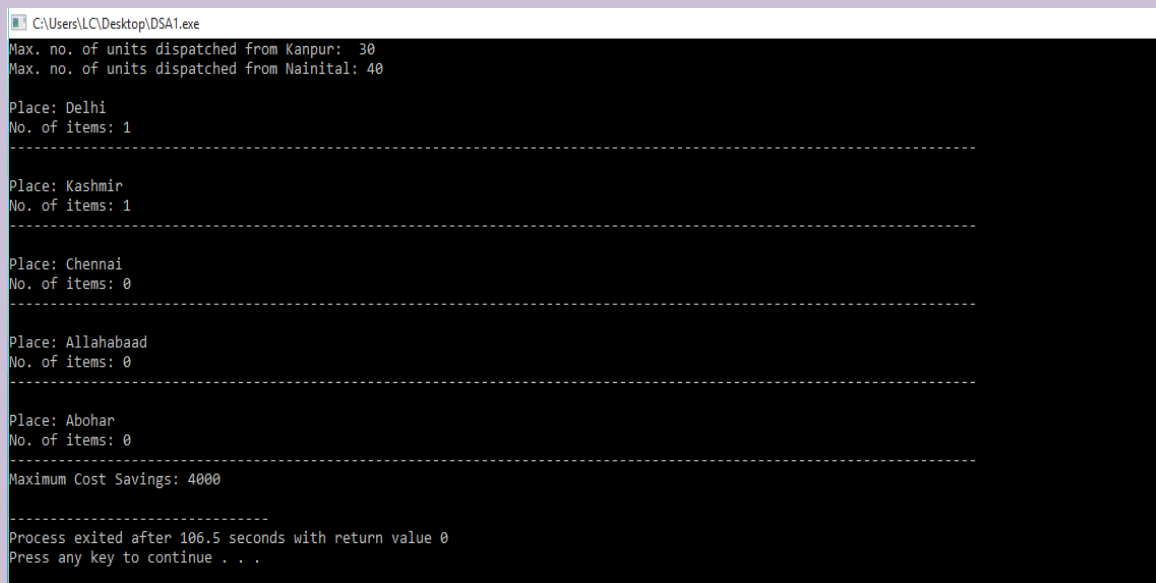
int main ()
{

```

```

count = (struct constraints*)malloc(n * sizeof (int));
max = (struct constraints*)malloc(n * sizeof (int));
max_cost = 0;
float kp, nain;
cout<<"Max. no. of units dispatched from Kanpur: ";
cin>>kp;
cout<<"Max. no. of units dispatched from Nainital: ";
cin>>nain;
knapsack(0, 0.0, kp, nain);
int i;
for (i = 0; i < n; i++)
{
cout<<"\nPlace: "<<items[i].name<<"\nNo. of items: "<<max[i]<<endl;
cout<<"-----\n";
}
cout<<"Maximum Cost Savings: "<< max_cost<<endl;
free(count);
free(max);
return 0;
}

```



```

C:\Users\LC\Desktop\DSA1.exe
Max. no. of units dispatched from Kanpur: 30
Max. no. of units dispatched from Nainital: 40

Place: Delhi
No. of items: 1
-----

Place: Kashmir
No. of items: 1
-----

Place: Chennai
No. of items: 0
-----

Place: Allahabaad
No. of items: 0
-----

Place: Abohar
No. of items: 0
-----

Maximum Cost Savings: 4000

-----
Process exited after 106.5 seconds with return value 0
Press any key to continue . . .

```

2) MANUFACTURING PROFIT

VARIABLES:

Z= Maximum profit form the pen production

Ai=Cost of the ith pen

Bi= Plastic requirement for the ith pen

Ci= Ink requirement for the ith pen

ANALOGY WITH KNAPSACK:

Z=maximize the profit

Ai= value

Bi=weight

Ci= volume

CODE:

```
#include<stdio.h>
#include<stdlib.h>
typedef struct {
    char *name;
    float cost;
    int plastic;
    int ink;
} itemconstraints;
itemconstraints items[] = {
    {"gel pen", 15.5, 30, 115 },
    {"ball pen", 7.5 , 50, 85},
    {"roller", 50.0, 100, 74},
    {"sketch pen", 60, 7.5, 134},
    {"marker", 35,75, 173},
};
```

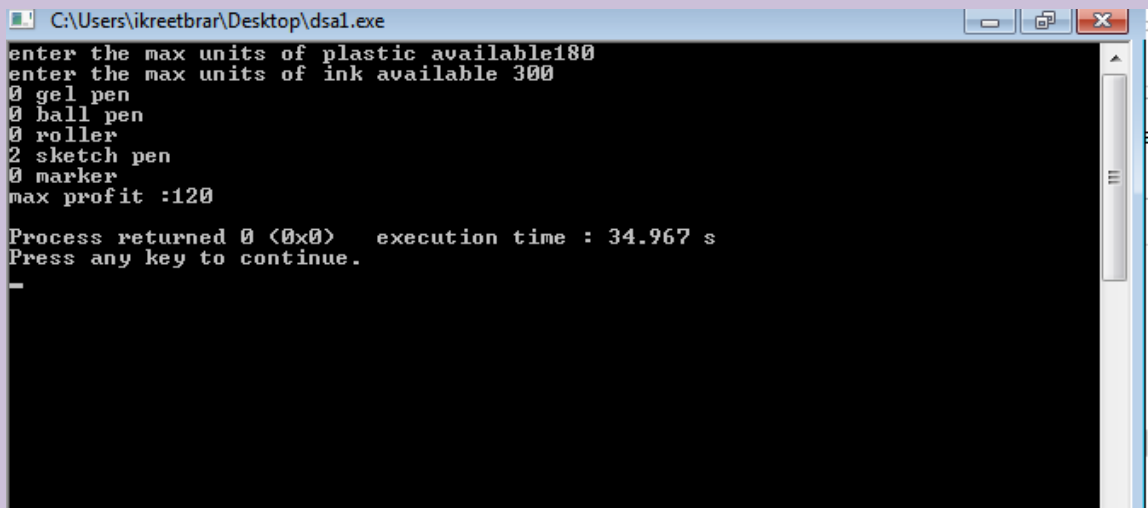
```

int n = sizeof (items) / sizeof (itemconstraints);
int *count;
int *best;
double best_cost;
void knapsack (int i, float cost, int plastic, int ink) {
int j, m1, m2, m;
if (i == n) {
if (cost > best_cost) {
best_cost = cost;
for (j = 0; j < n; j++) {
best[j] = count[j];
}
}
return;
}
m1 = plastic/ items[i].plastic;
m2 = ink/ items[i].ink;
m = m1 < m2 ? m1 : m2;
for (count[i] = m; count[i] >= 0; count[i]--) {
knapsack(
i + 1,
cost + count[i] * items[i].cost,
plastic - count[i] * items[i].plastic,
ink- count[i] * items[i].ink
);
}
}
int main () {
count = malloc(n * sizeof (int));
best = malloc(n * sizeof (int));
best_cost = 0;
float pr,ar;
printf("enter the max units of plastic available");

```

```
scanf("%f",&pr);
printf("enter the max units of ink available ");
scanf("%f",&ar);
knapsack(0, 0.0, pr, ar);
int i;
for (i = 0; i < n; i++) {
printf("%d %s\n", best[i], items[i].name);
}
printf("max profit :%.0f\n", best_cost);
free(count); free(best);
return 0;
}
```

OUTPUT:



```
C:\Users\ikreetbrar\Desktop\dsa1.exe
enter the max units of plastic available180
enter the max units of ink available 300
0 gel pen
0 ball pen
0 roller
2 sketch pen
0 marker
max profit :120
Process returned 0 (0x0)   execution time : 34.967 s
Press any key to continue.
-
```

ANALYSIS

The analysis of multidimensional knapsack problem is done considering two different algorithms to solve the same multidimensional knapsack algorithm by considering both space and time complexity of both the algorithms.

Let us assume we are considering two algorithms greedy and dynamic programming.

We know that the time complexity of dynamic programming is $O(n)$.

Comparing the state values given as solution when we solve multidimensional knapsack using greedy and dynamic programming

Time Complexity:

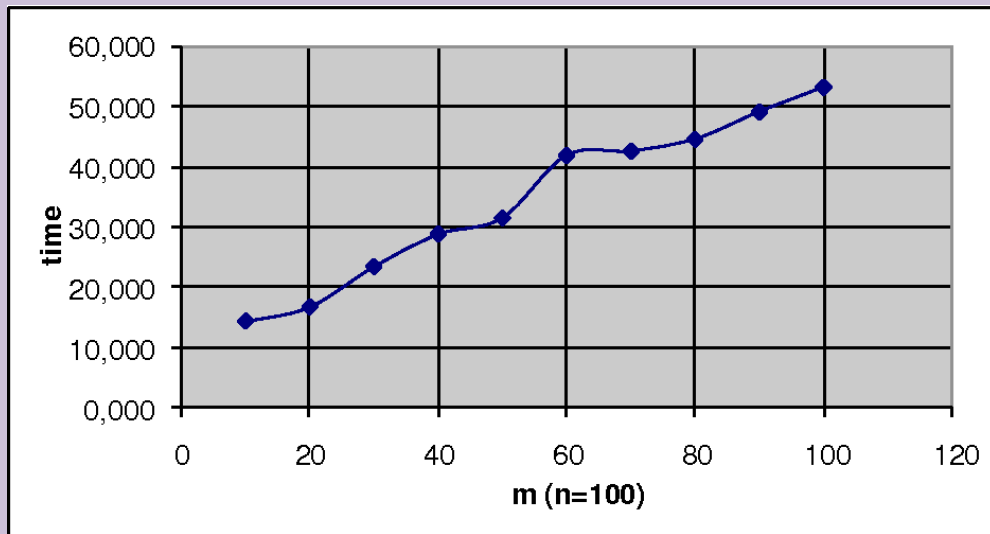
The time complexity of an algorithm is calculated by the number of processes it does before computing the results, so when we give different number of inputs the number of processes in different algorithm varies as follows:

We vary the number of inputs keeping the capacity constant. Let us assume =200

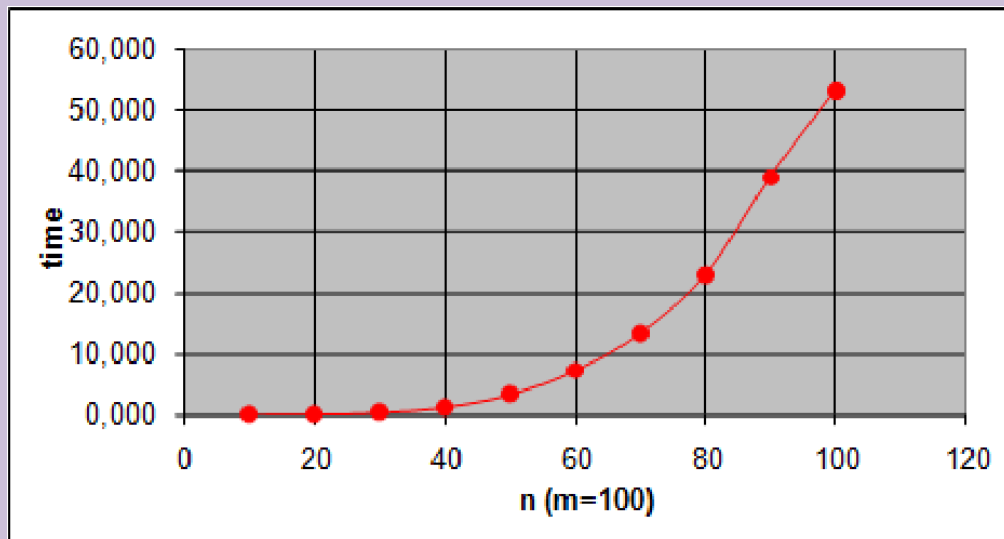
Number of inputs	Dynamic programming	Greedy algorithm
20	1067	45
30	2445	56
40	2879	78
50	3456	89
60	4564	98
70	5675	108

The graph becomes:

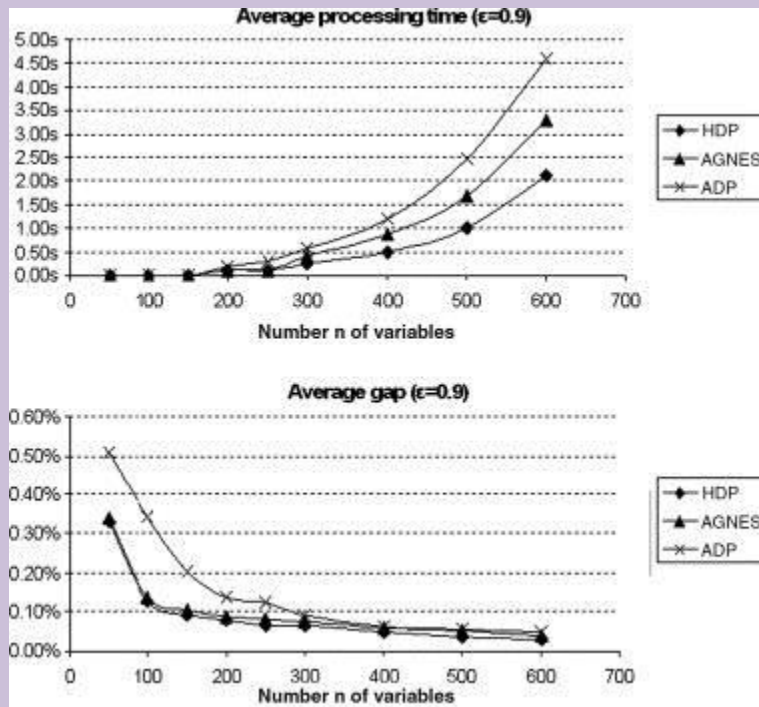
For dynamic programming



For greedy algorithm:



If we consider other algorithms, our graph becomes:



Space complexity:

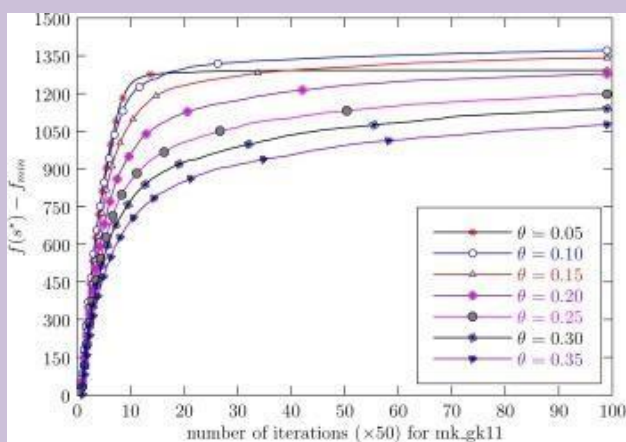
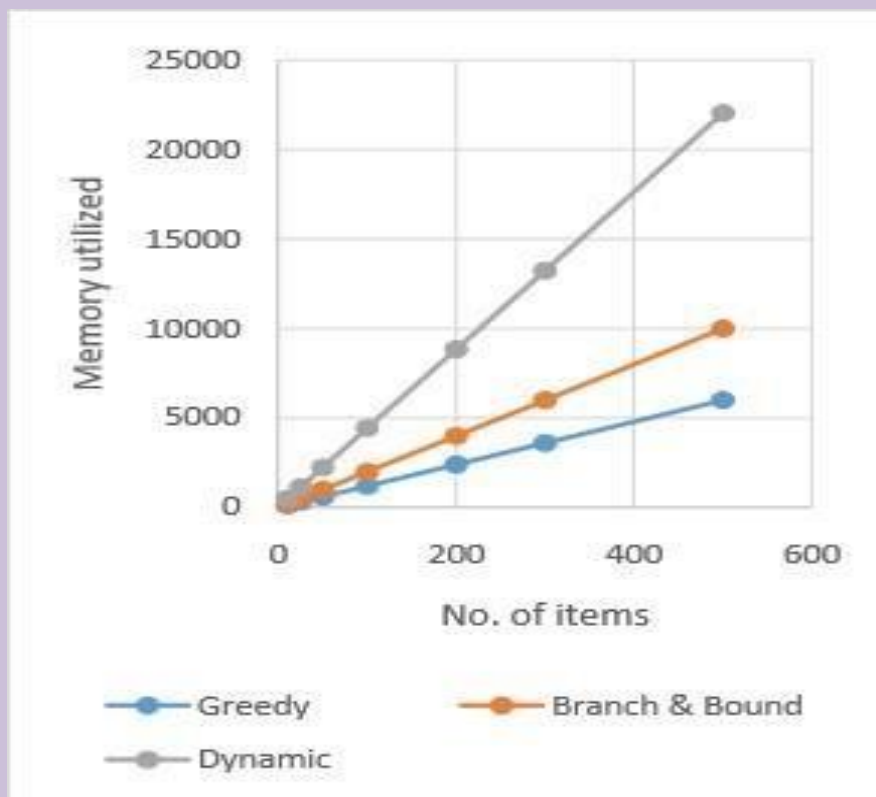
Space complexity of an algorithm is given when we vary the capacity and the total number of items.

Here the total memory consumption I bits is calculate for both the algorithms:

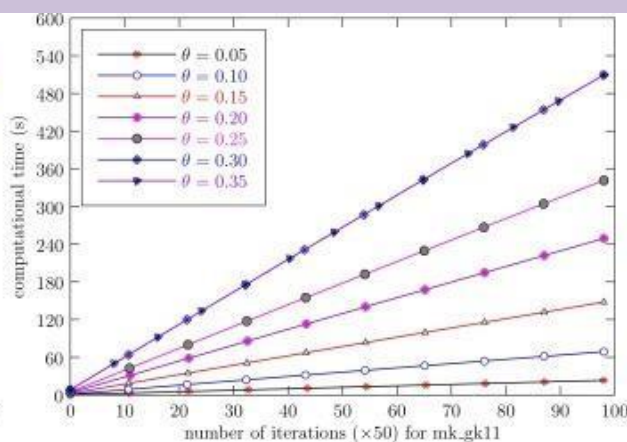
The table becomes:

Number of inputs	Dynamic programing	Greedy algorithm
20	450	289
30	678	345
40	987	476
50	1456	576
60	2557	698
70	4568	789
80	6544	934

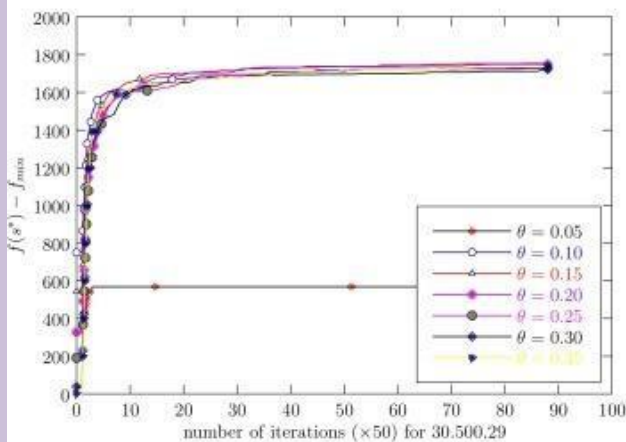
The graph becomes:



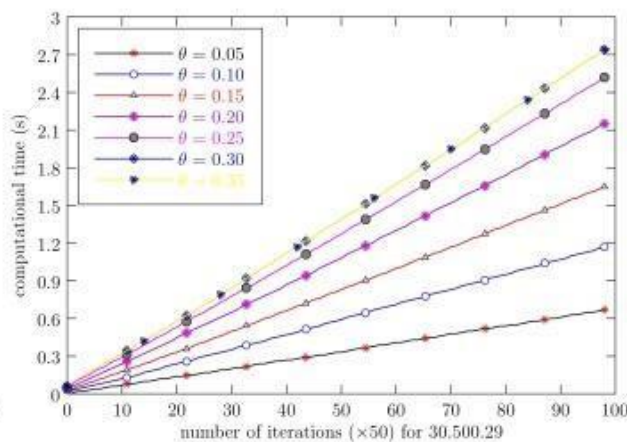
(a) Evolution of the best objective value



(b) Evolution of run time



(c) Evolution of the best objective value



(d) Evolution of run time

The above graph represents the number of inputs and accordingly the comparison between the time and number of iterations , which gives us both the time and space complexity of our algorithms.

Conclusion:

The most efficient technique is the Greedy Algorithm, but it is inappropriate under certain conditions since it does not result in the optimal solution.

The Dynamic programming technique proves to be very efficient in terms of number of computations for lesser capacities, but as the capacity of the knapsack increases, this technique proves to be inefficient. The memory utilized by this technique is also the highest among the three approaches considered.

For future work, genetic algorithms could be applied for the given problem, and a comparative analysis of the performance of the original algorithms .and the modified algorithms could be implemented

GRAPH ANALYSIS

Time vs number of items:

Number of items: 5

Time:

```
C:\Users\sandhya\Desktop\c\Untitled1202420.exe
enter the max weight that can be carried:27
enter the max volume of trolley:56
0 artefact1
12 artefact2
1 artefact3
0 artefact4
3 artefact5
best value: 42

Process returned 0 (0x0)   execution time : 3.342 s
Press any key to continue.
```

Number of items: 10
Time:

```
C:\Users\sandhya\Desktop\c\Untitled1202420.exe
enter the max weight that can be carried:27
enter the max volume of trolley:56
0 artefact1
0 artefact2
2 artefact3
0 artefact4
3 artefact5
0 artefact6
0 artefact7
0 artefact8
8 artefact9
0 artefact10
best value: 93

Process returned 0 (0x0)   execution time : 4.893 s
Press any key to continue.
```

Number of items: 15
Time:

```
C:\Users\sandhya\Desktop\c\Untitled1202420.exe
enter the max weight that can be carried:27
enter the max volume of trolley:56
0 artefact1
0 artefact2
2 artefact3
0 artefact4
3 artefact5
0 artefact6
0 artefact7
0 artefact8
8 artefact9
0 artefact10
0 artefact11
0 artefact12
0 artefact13
0 artefact14
0 artefact15
best value: 93

Process returned 0 (0x0)   execution time : 7.031 s
Press any key to continue.
```

Number of items: 20

Time:

```
C:\Users\sandhya\Desktop\c\Untitled1202420.exe
enter the max weight that can be carried:27
enter the max volume of trolley:56
0 artefact1
0 artefact2
2 artefact3
0 artefact4
3 artefact5
0 artefact6
0 artefact7
0 artefact8
8 artefact9
0 artefact10
0 artefact11
0 artefact12
0 artefact13
0 artefact14
0 artefact15
0 artefact16
0 artefact17
0 artefact18
0 artefact19
0 artefact20
best value: 93

Process returned 0 (0x0)   execution time : 4.248 s
Press any key to continue.
```

Number of items: 40

Time:

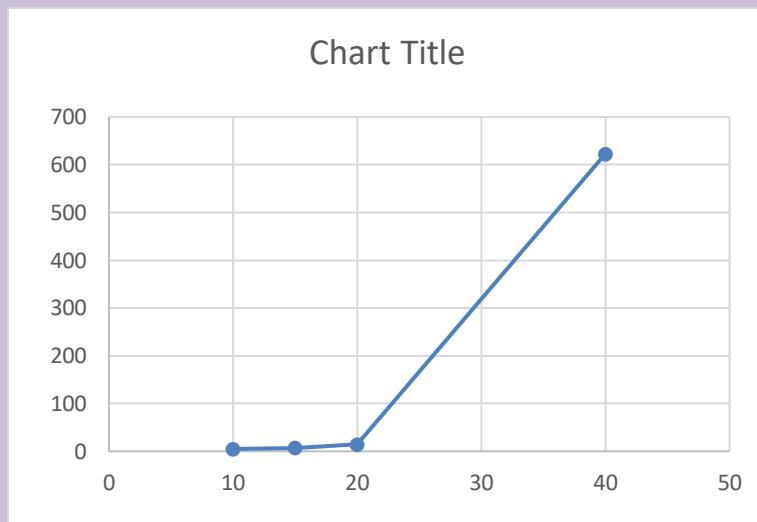
```
C:\Users\sandhya\Desktop\c\Untitled1202420.exe
0 artefact16
0 artefact17
0 artefact18
0 artefact19
0 artefact20
0 artefact21
0 artefact22
0 artefact23
0 artefact24
0 artefact25
0 artefact26
0 artefact27
0 artefact28
0 artefact29
0 artefact30
0 artefact31
0 artefact32
0 artefact33
0 artefact34
0 artefact35
0 artefact36
0 artefact37
0 artefact38
0 artefact39
0 artefact40
best value: 93

Process returned 0 (0x0)   execution time : 623.053 s
Press any key to continue.
```

Number of inputs:80
Time:

S:NO	No of inputs	Time taken
2	10	4.893
3	15	7.031
4	20	14.248
5	40	623.053
6	80	

GRAPH:



REFERENCES:

<https://github.com/rdok/Data-Structures>

https://en.wikipedia.org/wiki/Knapsack_problem <https://www.sanfoundry.com/>

<https://www.geeksforgeeks.org/knapsack-problem/>

<https://www.researchgate.net/publication/220669373> The Multidimensional Knapsack Problem Structure and Algorithms

<https://www.hackerearth.com/practice/notes/the-knapsack-problem/>

[https://www.tutorialspoint.com/design_and_analysis_of_algorithms_01_knapsack.html](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_01_knapsack.html)

THANK YOU