Empirical Study of three different approaches to solve 0-1 knapsack problem

Authors

Charchita Kuppa
Ruikang Mao

Abstract

The purpose of this paper is to present a comparative study of the performances of three different algorithms to solve 0-1 knapsack problem. To solve the 0-1 knapsack problem, we implemented 3 algorithms (brute force algorithm, bottom-up dynamic programming and top-down dynamic programming) in two different languages (python and java). After analyzing the time complexity of each algorithm. We generated test cases to compare the performance of these algorithms in terms of runtime complexity with varying n(no of items with values[1,2,..,n] and weights[1,2,..,n]) and W(knapsack capacity).

Introduction

The 0-1 knapsack problem is an optimization problem where we need to choose a subset of items (in a set of items) to be placed in a knapsack, whose total value(sum of values of items in subset) is maximized without exceeding the capacity of the knapsack. In this project Brute force , top down and bottom up dynamic programming algorithms are implemented to solve the 0-1 Knapsack Problem, which can help us learn more about dynamic programming and designing an experimental study to compare algorithms.

Algorithm Implementation

0-1 Knapsack Problem Description

There are n distinct items that may potentially be placed in the knapsack. Each item *i* to be placed in a knapsack has a positive integer weight w_i and positive integer value v_i .Given values v[1,2,….n] and weights[1,2,…,n], of items we need to find a subset of items that has

Maximum $\sum_{i=0}^{n} v_i$ provided $\sum_{i=0}^{n} w_i$ ≤ W(capacity of knapsack)

Brute Force Approach

Given n items with values v[1,2,…,n] and weights w[1,2,…n] there are 2^n possibilities in choosing items. An item *i* is either in the knapsack or not, this represented by an array of 0's and 1's ,0 when item *i* is not chosen and 1 when item *i* is not chosen. Total values and total weights for all 2^n subsets are calculated and subsets of items with maximum values that do not exceed capacity of knapsack are chosen and indices of these items in each subset are returned as an array.

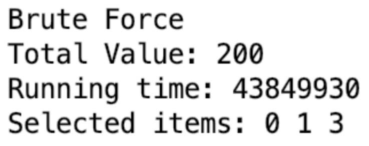
Input: values[1,2,…,n], weights[1,2,…,n], capacity
Output: array of indices of subset items with maximum v$alue$ and total weight ≤ capacity

Algorithm:
```
 arrayOfSubsets ⟵ [ ]
 maxValue ⟵ 0
 for i ⟵ 1 to $2^n$ do
       index ⟵ 0
       totalValueOfSubset ⟵ 0
       totalWeightOfSubset ⟵ 0
       while arrayOfSubsets[index] != 0 and index < length-1 do
              arrayOfSubsets[index] ⟵ 0
              index ⟵ index + 1
              arrayOfSubsets[index] ⟵ 1
       for j ⟵ 1 to n do
              if arrayOfSubsets[j] = 1 then
              totalValueOfSubset ⟵ totalValueOfSubset + values[j]
              totalWeightOfSubset ⟵totalWeightOfSubset + weights[j]
       if totalValueOfSubset > maxValue and totalWeightOfSubset <= capacity then
       maxValue ⟵totalValueOfSubset
              arrayOfIndices ⟵ [ ]
              for k ⟵ 1 to n do
              if arrayOfSubsets[k] = 1 then
              arrayOfIndices [k] ⟵ k

       return maxValue, arrayOfIndices
```

Time Complexity: As mentioned above, there are 2^n possibilities in choosing a subset from n items. Therefore, it would take O(2^n) time to iterate all subsets. For each subset, we need to iterate all

items in this subset, which would cost O(n) time. For each item we update the total value and weight, which would cost constant time. Therefore, the running time of brute force algorithm is O($n * 2^n$).
For the specific input value ={40, 100, 50, 60}, weight = {20, 10, 40, 30} and W = 60, the output is shown below.



```
Brute Force
Total Value: 200
Running time: 43849930
Selected items: 0 1 3
```

Dynamic Programming

Dynamic Programming solves each of the smaller subproblems only once and stores the results rather than solving overlapping subproblems over and over again. These solutions to the subproblems are then used to obtain a solution to the original problem. After we populate all maximum values in the dp array, we need to use backtracking to find all items that are selected. Here is the backtracking algorithm, whose running time is O(n), n is the number of items.

Algorithm:

```
i ⟵ length
w ⟵ capacity
arrayOfIndices ⟵ [ ]
while i > 0 and w > 0  do
        index ⟵ i - 1
        if  weights[index] <= w  then
                if  values[index] + V[i-1][w-weights[index]]  > V[i-1][w]  then
                        arrayOfIndices[index] ⟵ index
                        i ⟵ index
                        w ⟵ w - weights[index]
                else:
                        i ⟵ index
        else:
                i ⟵ i - 1
return arrayOfIndices
```

Bottom-up Approach

Suppose there is optimal solution OPT(I_k , w) with maximum value and maximum available weight w for subset I_i of first i ({1,2,..,i}) items in the set of items {1,2,….n} that are chosen to be placed in a knapsack. Then the optimal solution for subset I_{i+1} with first (i+1) items

OPT(I_{i+1}, w) = max(OPT(I_i , w), v_{i+1} + OPT(I_i , w- w_{i+1}))

We can divide all the subsets of the first i + 1 items that fit the knapsack of capacity W into two categories subsets that do not include the $i + 1^{th}$ item and subsets that include the $i + 1^{th}$ item.

If the item i+1 is not in the optimal set then

OPT(I_{i+1}, w) = OPT(I_i, w)

If item i+1 is in the optimal set then

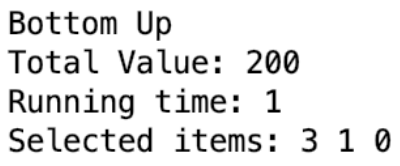
OPT(I_{i+1}, w) = v_{i+1} + OPT(I_i, w-w_{i+1}))

Input: values[1,2,…,n], weights[1,2,…,n], capacity
Output: array of indices of subset items with maximum v$alue$ and total weight $\le$ capacity

Algorithm:

```
V ←[ ][ ]
for i ← 0 to n do
    V[i][0] ← 0
for w ← 0 to W do
    V[0][w] ← 0
for i ← 1 to n do
    index ← i - 1
    for w ← 0 to W do
        if weights[index] > w then
            V[i][w] ← V[i-1][w]
        else
        V[i][w] ← max(V[i-1][w], values[index] + V[i-1][w-weights[index]])
return backTrack(V, values, weights, length, capacity)
```

Time Complexity: In this algorithm, we iterate all items from 0 to n and for each item we iterate capacity from 0 to W. Inside the inner loop, we just update the dp array, which costs constant time. Therefore, the running time of bottom up dynamic programming is O(n * W). For the specific input value = {40, 100, 50, 60}, weight = {20, 10, 40, 30} and W = 60, the output is shown below.



```
Bottom Up
Total Value: 200
Running time: 1
Selected items: 3 1 0
```

Top Down Approach

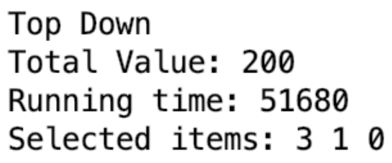
Dynamic programming Top down approach uses memorization technique which solves only needed subproblems

Input: values[1,2,…,n], weights[1,2,…,n], capacity, V[1,2,…,n][1,2,…W], currentIndex
Output: array of indices of subset items with maximum v$alue$ and total weight $\le$ capacity

Algorithm:

```
if currentIndex <= 0 or capacity < 0 then
    return 0
if weights[currentIndex -1] > w then
    V[i][w] ← knapsack_recursion(values, weights, V, currentIndex-1, W)
else
    V[i][w] ← max( knapsack_recursion(values, weights, V, currentIndex-1, W),
            values[currentIndex - 1] + knapsack_recursion(values, weights,V, currentIndex-1,
            W-weights[currentIndex - 1]) )
return backTrack(V, values, weights, length, capacity)
```

Time Complexity: In this algorithm, we use a 2-D array to store a particular state (n, w) if we get it the first time. Now if we come across the same state (n, w) again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time. Therefore, the running time of bottom up dynamic programming is O(n * W). For the specific input value = {40, 100, 50, 60}, weight = {20, 10, 40, 30} and W = 60, the output is shown below.



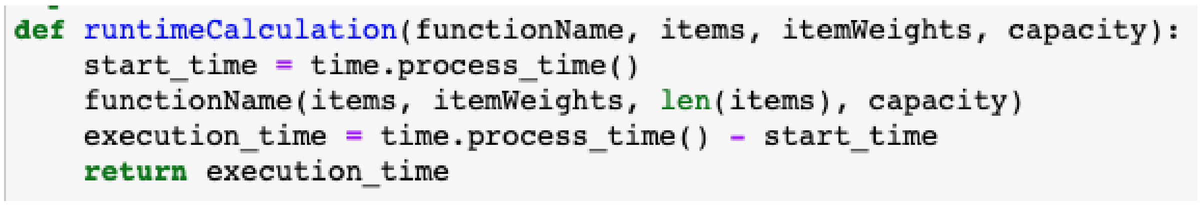
```
Top Down
Total Value: 200
Running time: 51680
Selected items: 3 1 0
```

Performance Comparison and Analysis

Test setup

Implementation:
Technologies used: python and Java
Operating system: macOS Catalina version 10.15.7

For the testing of the different algorithms, we randomly generated integers representing the weights and values of items. Two types of Datasets are generated for both values and weights of items - Data sets with duplicates and Data sets without duplicates (random integers with duplicates and without duplicates).The timer is started before each call to each algorithm in every iteration of testing and stopped after the call is completed. The difference in start and stop times is considered as execution time.



```python
def runtimeCalculation(functionName, items, itemWeights, capacity):
    start_time = time.process_time()
    functionName(items, itemWeights, len(items), capacity)
    execution_time = time.process_time() - start_time
    return execution_time
```

- Two phases of testing are performed. In the first phase of testing, we were increasing the number of items to be considered for the knapsack, while holding the capacity of the knapsack constant.

- During the second phase of testing, we were increasing the capacity of the knapsack, while fixing the number of items.

- We could only vary the datasets size till 20 for brute force approach . So a separate analysis for the other two approaches is performed with *n* up to 5000.

First phase:

- The performance of all three algorithms were tested on *n,* varying from 5 to 20, incrementing n by 5 in each iteration of testing and capacity = 15(constant) for both Data sets with duplicates and without duplicates.
- The performance of dynamic programming bottom up and top down approaches were tested on *n* varying from 500 to 5000, incrementing *n* by 500 in each iteration of testing and capacity = 50(constant) for both Data sets with duplicates and without duplicates.

Data set without duplicates:

W = 15	Execution time in seconds			
	n=5	n=10	n=15	n=20
Brute Force	0.00024	0.0056	0.197	8.1
Bottom Up	0.00012	0.00022	0.00032	0.00062
Top Down	0.00005	0.00011	0.00035	0.00075

Data set with duplicates:

W = 15	Execution time in seconds			
	n=5	n=10	n=15	n=20
Brute Force	0.00022	0.0080	0.305	12.50
Bottom Up	0.00017	0.00041	0.00081	0.00091
Top Down	0.00006	0.00024	0.00094	0.0014

Data set with duplicates:

W = 50	Execution time in seconds									
	n 500	n 1000	n 1500	n 2000	n 2500	n 3000	n 3500	n 4000	n 4500	n 5000
Bottom Up	0.023	0.047	0.0742	0.10	0.13	0.15	0.18	0.21	0.24	0.27
Top Down	0.012	0.039	0.112	1.17	6.48	8.71	12.57	17.41	20.71	26.71

Data set without duplicates:

W = 50	Execution time in seconds									
	n 500	n 1000	n 1500	n 2000	n 2500	n 3000	n 3500	n 4000	n 4500	n 5000
Bottom Up	0.016	0.031	0.046	0.063	0.081	0.099	0.112	0.130	0.146	0.163
Top Down	0.0081	0.028	0.074	0.21	0.74	1.56	6.94	8.42	22.27	33.01

Second phase:

- The performance of all three algorithms were tested on *W* (capacity) = varying from 10 to 100, incrementing W by 10 in each iteration of testing and n = 20 (constant) for both Data sets with duplicates and without duplicates.

Data set with duplicates:

n = 20	Execution time in seconds									
	W= 10	W= 20	W= 30	W= 40	W= 50	W= 60	W= 70	W= 80	W= 90	W= 100
Brute Force	12.13	14.43	14.96	14.97	15.75	14.75	14.75	14.75	14.71	14.71
Bottom Up	0.0004	0.0013	0.0022	0.0030	0.0041	0.0051	0.0058	0.0072	0.0079	0.0081
Top Down	0.0005	0.0027	0.012	0.041	0.12	0.25	0.46	0.81	1.24	1.45

Data set without duplicates

n = 20	Execution time in seconds									
	W= 10	W= 20	W= 30	W= 40	W= 50	W= 60	W= 70	W= 80	W= 90	W= 100
Brute Force	7.51	8.94	9.20	9.23	9.19	9.17	9.23	9.26	9.26	9.26
Botto m Up	0.00 018	0.000 46	0.000 92	0.001 1	0.001 4	0.001 7	0.002 1	0.002 24	0.002 7	0.002 9
Top Down	0.00 034	0.003 1	0.008 1	0.025	0.064	0.14	0.25	0.42	0.16	0.82

Analysis of test Results

First phase:

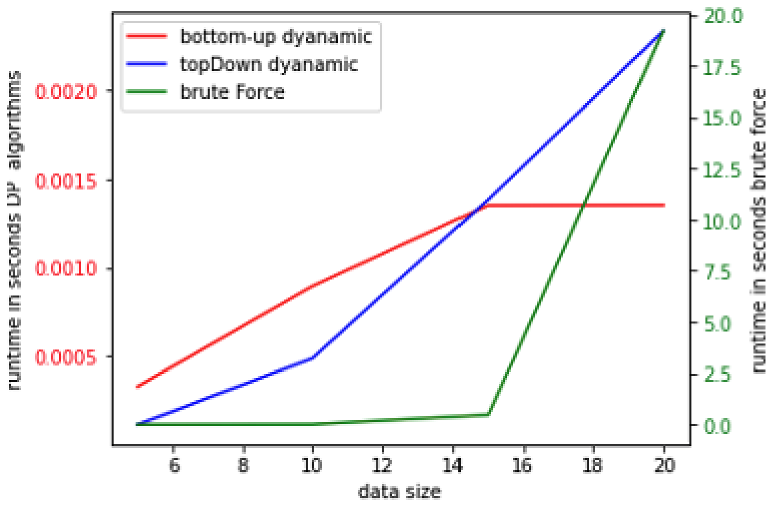


Figure 1 Performance of 3 algorithms with W = 15 (without duplicates in dataset)

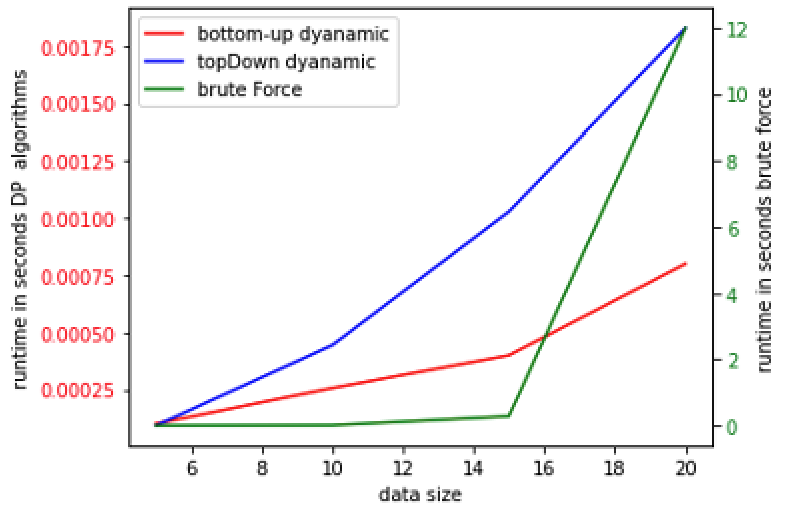


Figure 2 Performance of 3 algorithms with W = 15 (with duplicates in dataset)

The scale on the y-axis in “red” indicates the runtime for dynamic programming bottom up approach and dynamic programming top down approach .The scale on the y-axis in “green” indicates the runtime for brute force approach.
It can be observed that the rate of change in execution time is exponential for brute force as data size increases from 15 to 20.There is no change in performance for brute force with respect to duplicity in the dataset.
The rate of change in runtime for dynamic programming top down approach is linear with respect to rate of change in data size . There is no change in performance for top down approach with respect to duplicity in the dataset.
The rate of change in runtime for dynamic programming bottom up approach is linear as data size varies from 5 to 15 and becomes constant after that for data set with the duplicates while for data sets without duplicates the rate of change in execution time is linear with respect to change in dataset (although the change in runtime is not as high when compared to top down approach)

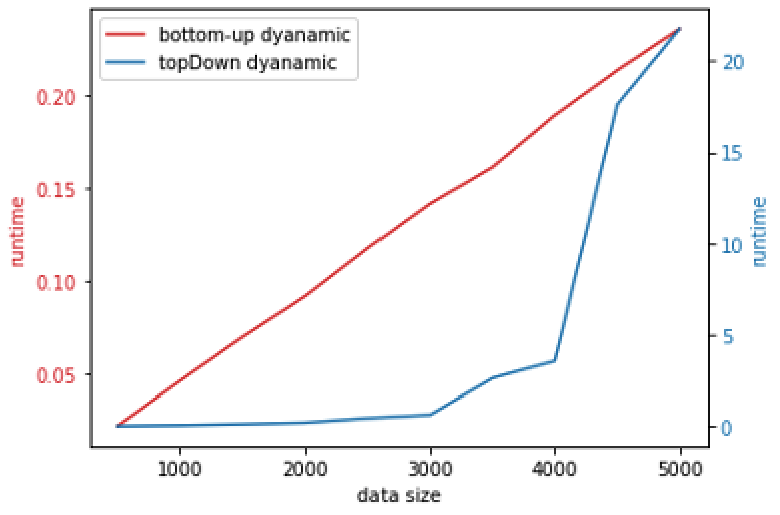


Figure 3 Performance of bottom-up dp and top-down dp with W = 50 (without duplicates in dataset)

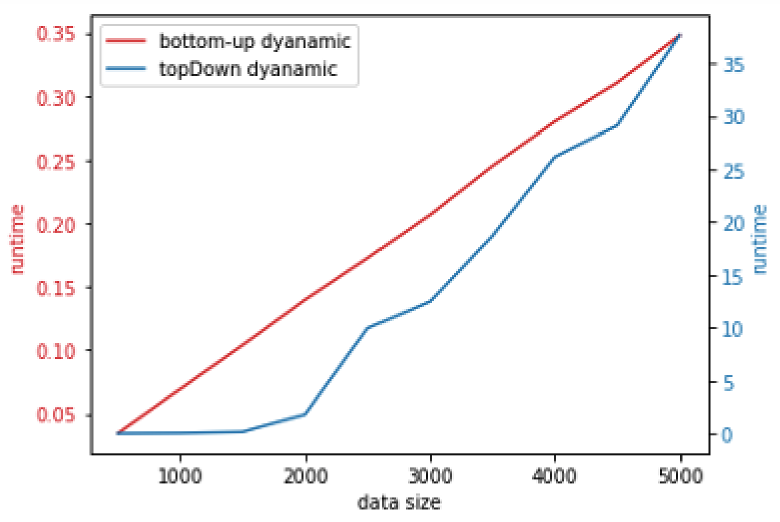


Figure 4 Performance of bottom-up dp and top-down dp with W = 50 (with duplicates in dataset)

The scale on the y-axis in "red" indicates the runtime for dynamic programming bottom up approach and the scale on the y-axis in "blue" indicates the runtime for dynamic programming top down approach. It can be observed that the execution times (in seconds)for dynamic programming bottom up approach (for n from 500 to 5000) are much smaller when compared to the execution times for top down approach.

Dynamic top down approach:
For the data sets with duplicates the rate of change in runtime is incremental with respect to rate of change in data size. For the data sets without duplicates execution time is constant as the data size varies till 3000 And there is a steep change in the curve as the data size changes from 4000 to 5000

Dynamic bottom up approach:
The rate of change of execution time with respect to data size is linear for dynamic programming bottom up approach. There is no effect on performance of dynamic bottom up approach with respect to duplicity.

Second phase:

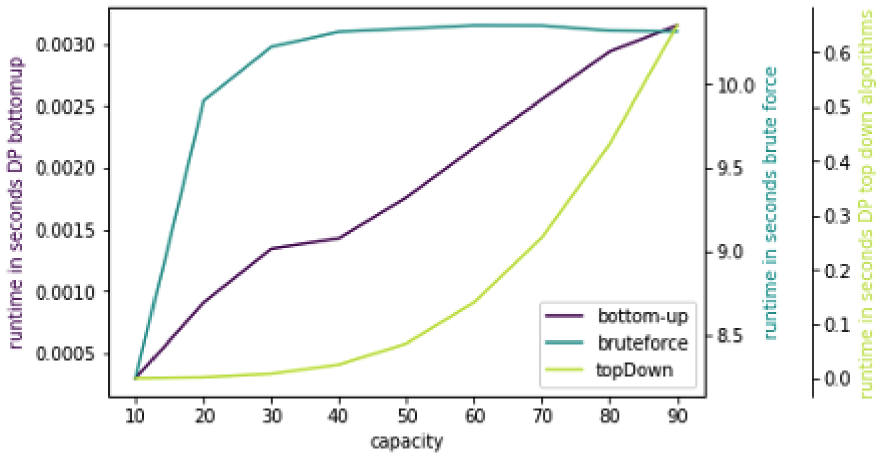


Figure 5 Performance of 3 algorithm with n = 20 (without duplicates in dataset)

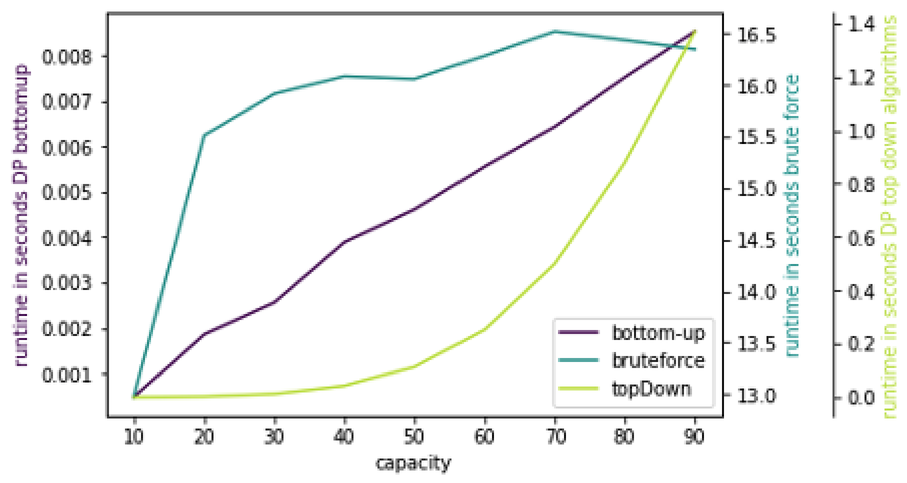


Figure 6 Performance of 3 algorithm with n = 20 (with duplicates in dataset)

The rate of change in execution time for brute force approach is almost constant (after W =10) with respect to change in capacity from 10 to 100.

As the capacity increases (W from 10 to 100), the rate of increase in runtime for dynamic bottom up approach with respect to capacity is slower compared to the rate of increase in runtime for dynamic top down approach when data size is constant(n = 20).

There is no effect in performance (in terms of execution time) of both bottom up and top down approach with respect to duplicity.

Conclusion

After analyzing the performance of those three algorithms, we can draw the conclusion that bottom-down dynamic programming is the most efficient algorithm among those three to solve the 0-1 knapsack problem. Although both bottom-up dynamic programming and top-down dynamic programming run in O(n * W) time, it seems that bottom-up dynamic programming runs faster. The reason could be that when we use top-down dynamic programming, there are a lot of recursive calls and return statements.