0-1 Knapsack Problem Analysis

Nate Tomlin

Eastern Mennonite University

I. Abstract

The purpose of this paper was to create and analyze three algorithms that solve the classic dynamic programing 0-1 knapsack problem. The three algorithms that were written to solve the problem are the top down dynamic programing approach, the bottom up dynamic programing approach and a brute force approach. After the codes for the algorithms were written they were tested against one another based on their runtimes for a certain data size. Those runtimes were compared to see what the fastest algorithms are and how they compare to one another.

II. Background

The 0-1 knapsack problem is a considered to be a combinatorial optimization problem. Its goal is to find the highest total value of the subset of items with given values and weights with a total weight constraint. It is also said that the 0-1 knapsack problem falls into the complexity of non-deterministic polynomial-time, which means that there are no know efficient algorithm solutions whose time complexity is in polynomial time [1]. With that said it is known that greedy algorithms to not work in solving this problem. This leaves dynamic programing and brute force algorithms to solve the problem. Two dynamic programing algorithms that have been used in the past to solve the problem have been the top down approach and the bottom up approach.

III. Methods

To solve this problem, I first started researching existing algorithms that achieved top down, bottom up and brute force solution to the problem. I then took what I knew from those codes and modified those algorithms to each of what I need to achieve. My top down algorithm goes through the items and checks to see if the score of the current item or the previous item is higher and will output the best possible score using recursion. My bottom up algorithm uses a two dimensional array to hold the weights and number of items and then compares the weight of the last item going back to the first item in a loop to the max weight. This then uses recursion to find the best possible score for the items. My brute force algorithm uses a list to hold both the items values and individual weights. It then takes the power set of the that lists and searches through all of the possible combinations of data in a loop to find the most optimal score that is under the max weight constraint of the knapsack. The code for each of my algorithms can be seen in the appendix.

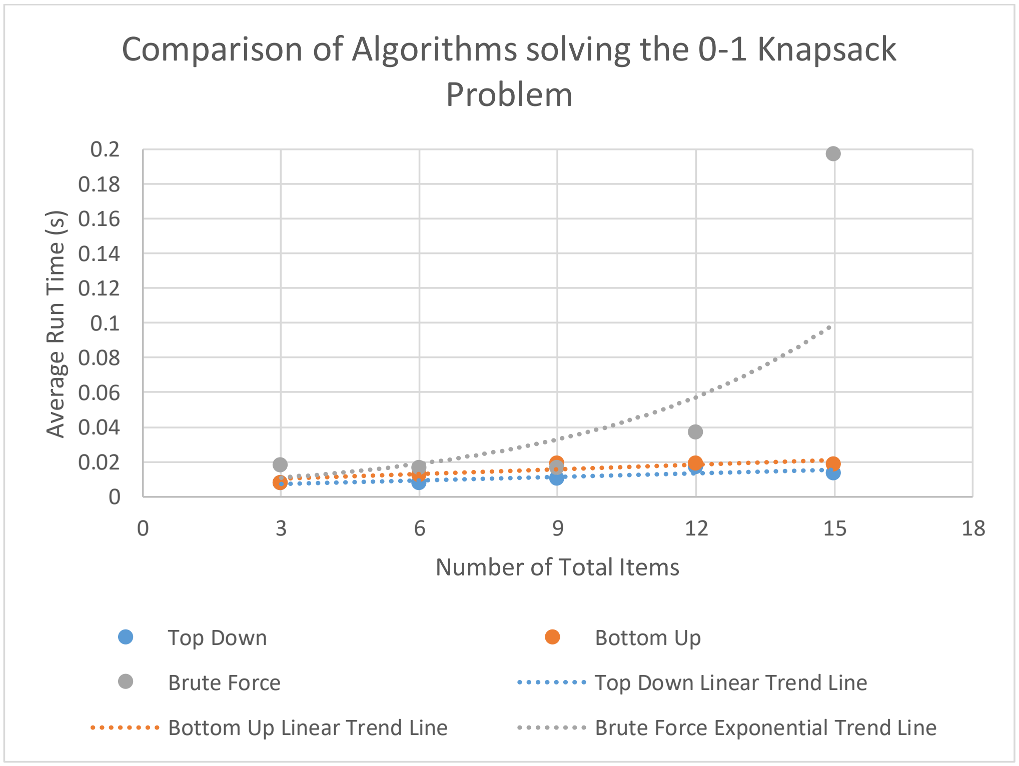
The experiments I will run will be to analyze the different run times of the above algorithms with a different number of items and compare those times to each other. I set up a loop after the core of the algorithms that will allow my algorithms to run a certain amount of times and then put the run time into a list. After the loop I created a statement to print the average of all of the times in the list. For the sake of time but still getting a good average of times I chose to run the algorithm 1000 times. I chose to run the algorithms at an increasing item size of multiples of 3 until the it would take an unrealistically long time to run the algorithm for 1000 times.

IV. Complexity

Time complexity is a formula that can describes how fast a program or algorithm runs. This formula is typically displayed in Big O notation which denotes only the upper bound, and not the true growth rate of the program or algorithm [2]. The top-down algorithm and the bottom-up algorithm are designed using the dynamic programming technique. The other algorithm is brute force. The explanations of these algorithms can be seen above and with those explanations in mind it makes sense that the dynamic programming algorithms are faster than the brute force algorithm based on how many times the function is called and the list of values and weights are looked at.
With that said the time complexity for the dynamically programed algorithms is O(n*W). The *n* in this formula is the number of total items being evaluated is and *W* is the weight capacity of the of the knapsack. This time complexity is backed up by Tardos in his Subset-Sum dynamically programmed algorithm where he states, "the Subset-Sum (n, W) algorithm correctly computes the optimal value of the problem, and runs in O(nW) time" [2]. The brute force algorithms time complexity is $O(2^n)$. This is very similar to the brute force algorithm time complexity of Hristakeva and Shrestha of $O(n2^n)$ [3]. The brute force algorithm is slower than the dynamic programing because as seen in the brute force code, the algorithm has to look through every items value and weight and compare every one of those to every other one which takes much longer than the dynamic programming approach of recursive calls.

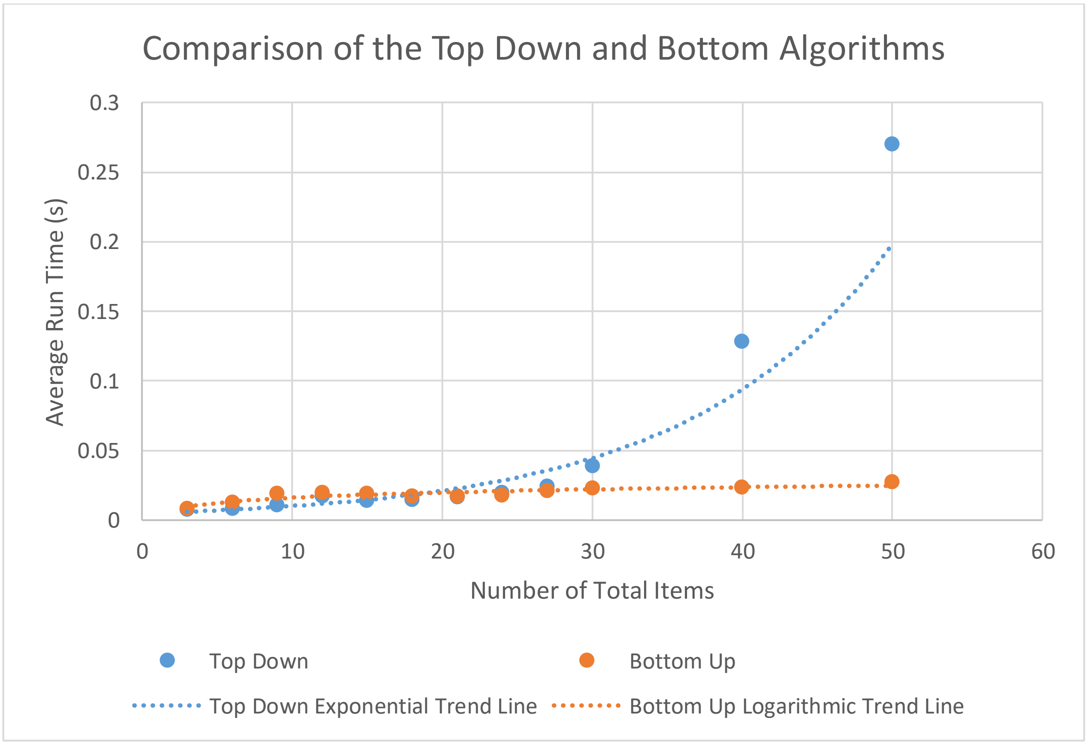
V. Experiments

I ran the tests as stated above in the methods section. I set my number of iterations to conduct the algorithm to 1000 so that the algorithm would run 1000 times and then spit out an average run time that I could enter as a data point of my graph. I conducted the tests with all three of my algorithms and the results can be found in the graph below:



This graph shows that as my number of total possible items to put in the knapsack increased the time it took to run the algorithms all increased. The brute force algorithm though increased in its run times a lot quicker as the number of total items increased as proven by the exponential trend line. The points aligning close to an exponential curve verifies that the time complexity of the brute force algorithm is indeed O(2^n). Tests were also run with a total item size of 18 but it was not put on the graph for the sake of being able to visually see the graph points of the dynamically programed algorithms as the brute force data point was so extreme. The tests were also run with a total item size of 21 but I was not able to completely run the brute force algorithm as it was started to take about two and a half hours to complete the test. With that said I just stopped the testing of the brute force algorithm with a total item value of 15 as seen in the graph above.

The top down and bottom up algorithms were still tested on higher total item values to see which one of those algorithms ran faster. Those test can be seen below:



This graph shows that the top down algorithm runs significantly slower than the bottom up algorithm as the total number of items is increased. It is also seen that neither of the algorithms show a linear trend line as seen in the other graph. The slower top down algorithm shows an exponential trend line as compared to faster bottom up trend line which is a logarithmic trend line. These tests could have been continued out further but after a total item size of 50 it is pretty easy to see that the top down algorithm is much slower than the bottom up as the number of total items is increased.

VI. Conclusion

After analyzing the run times for each of the algorithms in the graph above I can conclude that the dynamic programing approaches of top down and bottom up solve the problem much quicker especially as the size of the total number of items increases. The time complexity of the brute force algorithm of being exponential can be seen even in the limited amount of testing that was conducted as there is an exponential trend line that best fit the data points. Between the two so called quicker algorithms it is also seen that the bottom up algorithm is much quicker than the top down algorithm. This can be seen on the graph entitled, “Comparison of the Top Down and Bottom Algorithms”, as the bottom up algorithms data fits with a logarithmic trend line which is much faster than the exponential trend line that is seen fitting the top down algorithms data. With all this said it makes sense that brute force is the slowest algorithm out of the three because it has to go through every possible solution and then pick the best solution. Also, the top down algorithm is slower than its other dynamically programed bottom up algorithm because of the top downs algorithms heavy use of recursion which tends to slow programs down as the size of the problem gets too big. In the end the bottom up algorithm is the fastest in the 0-1 knapsack problem because a table of the data is formed and is just getting looped around without using recursion which makes it faster over larger data sets.

VII. References

[1] Murawski, C., & Bossaerts, P. (2016, October 07). How Humans Solve Complex Problems: The Case of the Knapsack Problem. Retrieved from https://www.nature.com/articles/srep34851

[2] Kleinberg, J., & Tardos, E. (2006). Algorithm design. Boston: Pearson/Addison-Wesley.

[3] Hristakeva, M., & Shrestha, D. (n.d.). Different Approaches to Solve the 0/1 Knapsack Problem. Retrieved 2019, from http://www.micsymposium.org/mics_2005/papers/paper102.pdf

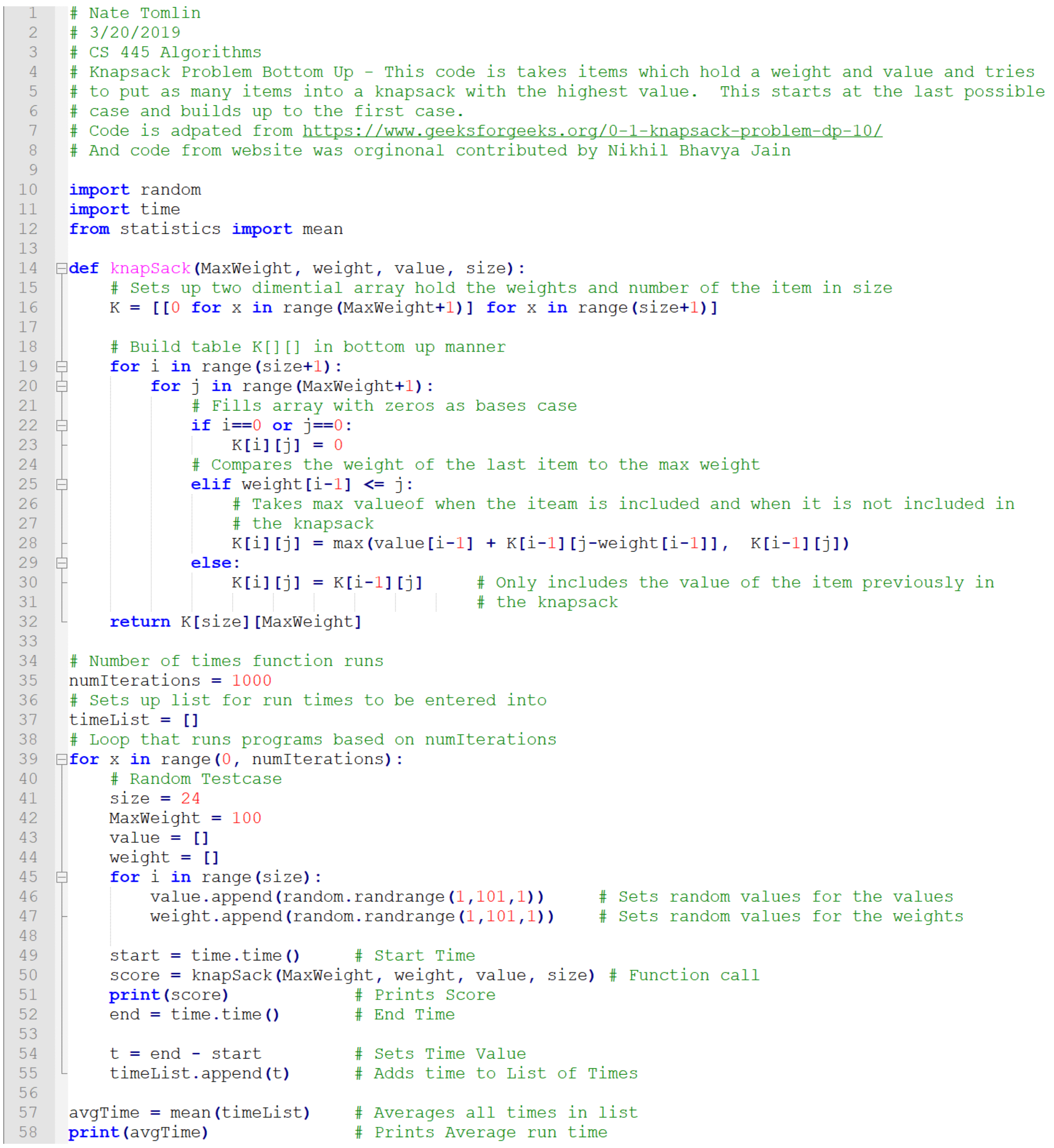
[4] 0-1 Knapsack Problem | DP-10. (2018, September 28). Retrieved from https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

VIII. Appendices

A. Top-Down Algorithm

```
# Nate Tomlin
# 3/20/2019
# CS 445 Algorithms
# Knapsack Problem Top Down - This code is takes items which hold a weight and value and tries
# to put as many items into a knapsack with the highest value.
# Code is adpated from https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
# And code for website was orginonal contributed by Nikhil Kumar Singh

import random
import time
from statistics import mean

def knapSack(MaxWeight, weight, value, size):
    # Base Case when size or weight are 0
    if size == 0 or MaxWeight == 0 :
        return 0
    # Compares the weight of the nth value to the max weight and if the nth value is more than
    # the weight we call the function without the nth value
    if (weight[size-1] > MaxWeight):
        return knapSack(MaxWeight, weight, value , size-1)
    else:
        # Key Point: Returns the max of either the item included or not included
        return max(value[size-1] + knapSack(MaxWeight - weight[size - 1], weight, value, size-1),
        knapSack(MaxWeight, weight, value, size-1))

# Number of times function runs
numIterations = 1000
# Sets up list for run times to be entered into
timeList = []
# Loop that runs programs based on numIterations
for x in range(0, numIterations):
    # Random Testcase
    size = 24
    MaxWeight = 100
    value = []
    weight = []
    for i in range(size):
        value.append(random.randrange(1,101,1))     # Sets random values for the values
        weight.append(random.randrange(1,101,1))    # Sets random values for the weights

    start = time.time()     # Start Time
    score = knapSack(MaxWeight , weight , value , size) # Function call
    print(score)            # Prints Score
    end = time.time()       # End Time

    t = end - start         # Sets Time Value
    timeList.append(t)      # Adds time to List of Times

avgTime = mean(timeList)    # Averages all times in list
print(avgTime)              # Prints Average run time
```

B. Bottom-Up Algorithm



```python
# Nate Tomlin
# 3/20/2019
# CS 445 Algorithms
# Knapsack Problem Bottom Up - This code is takes items which hold a weight and value and tries
# to put as many items into a knapsack with the highest value.  This starts at the last possible
# case and builds up to the first case.
# Code is adpated from https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
# And code from website was orginonal contributed by Nikhil Bhavya Jain

import random
import time
from statistics import mean

def knapSack(MaxWeight, weight, value, size):
    # Sets up two dimential array hold the weights and number of the item in size
    K = [[0 for x in range(MaxWeight+1)] for x in range(size+1)]

    # Build table K[][] in bottom up manner
    for i in range(size+1):
        for j in range(MaxWeight+1):
            # Fills array with zeros as bases case
            if i==0 or j==0:
                K[i][j] = 0
            # Compares the weight of the last item to the max weight
            elif weight[i-1] <= j:
                # Takes max valueof when the iteam is included and when it is not included in
                # the knapsack
                K[i][j] = max(value[i-1] + K[i-1][j-weight[i-1]],  K[i-1][j])
            else:
                K[i][j] = K[i-1][j]     # Only includes the value of the item previously in
                                        # the knapsack
    return K[size][MaxWeight]

# Number of times function runs
numIterations = 1000
# Sets up list for run times to be entered into
timeList = []
# Loop that runs programs based on numIterations
for x in range(0, numIterations):
    # Random Testcase
    size = 24
    MaxWeight = 100
    value = []
    weight = []
    for i in range(size):
        value.append(random.randrange(1,101,1))     # Sets random values for the values
        weight.append(random.randrange(1,101,1))    # Sets random values for the weights

    start = time.time()     # Start Time
    score = knapSack(MaxWeight, weight, value, size) # Function call
    print(score)            # Prints Score
    end = time.time()       # End Time

    t = end - start         # Sets Time Value
    timeList.append(t)      # Adds time to List of Times

avgTime = mean(timeList)    # Averages all times in list
print(avgTime)              # Prints Average run time
```

C. Brute Force Algorithm

```python
# Nate Tomlin
# 3/20/2019
# CS 445 Algorithms
# Knapsack Problem Brute Force - This code is takes items which hold a weight and value and tries
# to put as many items into a knapsack with the highest value.
# Function randomWV() and knapsack() is adapted from https://www.youtube.com/watch?v=EdLw7hjAyNU
# Function powerSet() is adapted from https://rosettacode.org/wiki/Power_set#Python

import random
import time
from statistics import mean

# This function sets up the the weights and values for each item in a list
def randomWV(size):
    WVList = []
    for i in range(size):
        #Sets random weights and values
        WVList.append((int(random.randrange(1,101,1)),int(random.randrange(1,101,1))))
    return WVList

# Creates a power set of all the possible combinations of the weights and values
def powerSet(WVList):
    result = [[]]
    for x in WVList:
        result.extend([subset + [x] for subset in result])
    return result

# Main function that compares all of the weights and values and returns the best value
def knapsack(VWList, MaxWeight):
    knapsack = []
    bestWeight = 0
    bestValue = 0
    for i in powerSet(WVList):
        # Adds all of the weights and all of the values
        sumWeight = sum([x[0] for x in i])
        sumValue = sum([x[1] for x in i])
        # Compares the running sum value to the best existing value while making sure that
        # it is under the max weight
        if sumValue > bestValue and sumWeight <= MaxWeight:
            bestValue = sumValue
            #bestWeight = sumWeight
            #knapsack = i
    return bestValue

# Number of times function runs
numIterations = 1
# Sets up list for run times to be entered into
timeList = []
# Loop that runs programs based on numIterations
for x in range(0, numIterations):
    size = 21
    MaxWeight = 100
    WVList = randomWV(size)

    start = time.time()                    # Start Time
    score = knapsack(WVList, MaxWeight)  # Function Call
    print(score)                           # Prints Score
    end = time.time()                      # End Time

    t = end - start                        # Sets Time Value
    timeList.append(t)                     # Adds time to List of Times

avgTime = mean(timeList)                   # Averages all times in list
print(avgTime)                             # Prints Average run time
```