Dept. of Computer Science, UC Davis. ECS122b June 6, 2016
Alex Torres|Chris Ellis|Georgy Golubev|Dan Manuylov|Ernesto Lomeli

Program #2 Report (Enumeration)

1 Introduction

In this report, we will observe the two enumeration implementations we discussed in class: subset enumeration and n-tuple enumeration. The first step was to create a program that generates data for the generalized knapsack problem. The generalized knapsack problem considers items *0, 1, 2, ..., n* with weights $w_0, w_1, w_2, ..., w_n$ and values $v_0, v_1, v_2, ...v_n$, as well as copies $c_0, c_1, c_2, ...c_n$. These items are to be stored into a knapsack with a certain weight capacity. The goal is to optimally store the items into the knapsack without going over the capacity and achieving the highest total value possible. Both of these algorithms use brute-force.

With the subset enumeration implementation, we treat each copy of an item as a distinct item, even though they will have the same weight and value. This algorithm takes each value and using brute-force, finds subsets that fit the capacity of the knapsack where the subset with the highest total value is the solution to the knapsack problem. The pseudo-code is as follows:

```
int curBestValue = 0;
    int curBestWeight = 0;
    int curBestPerm[totalNumItems] = 0;
    bool incomplete = true;
    while(incomplete != false)
        //add one to binary rep of permutations
        incomplete = addOne(curPerm, totalNumItems);
        //calculate this perms weight
        int tempWeight = sumVal(curPerm, weight, totalNumItems);
        //if calc'd weight exceeds limit
        if(tempWeight > MAXWEIGHT)
            continue;
        else //acceptable weight, is it better then our best?
            int tempValue = sumVal(curPerm, value, totalNumItems);
            if(tempValue > curBestValue)
```

```
                    curBestValue = tempValue;
                    curBestWeight = tempWeight;
                    int t = sizeof curBestPerm;
                    memcpy(curBestPerm, curPerm, sizeof curBestPerm);
                end if
            end if
        end while
```

With the n-tuple enumeration, we use the copies of each item to our advantage by taking the copies and mapping them onto the n-tuple. This is similar to the Add One Mixed Radix, except the bases are substituted for the number of copies of each item. The algorithm's checklist is to first enumerate all of the n-tuples, see if the n-tuple is "carryable", and then keep track of the most valuable solution. The pseudo-code is as follows:

```
//A is a 0'd out array of the same size as B, B holds max amount of items
Array NTuple(A, B, weight, value, maxWeight) :
    Array bestPerm;
    int bestWeight = 0;
    int bestValue = 0;
    int maxIterations = 1;
    for(i = 0; i < B.size; i++)
        maxIterations *= B[i];
    for(i = 0; i < maxIterations; i++)
        //takes perm vector and value or weight vector and
        //finds the corresponding weight or value.
        if(sum(A, weight) > maxWeight)
            continue;
        if(sum(A, Value) > bestValue)
            bestValue = sum(A, Value);
            bestWeight = sum(A, Weight);
            bestPerm = A;
        //this function increments the permutation
        addOneMixedRadix(A,B);
return bestPerm;
```

2 Empirical Studies

The data collected during each iteration in this study included the following: number of items, maximum weight of the knapsack, the number of unique items, the maximum number of individual copies, the time for subset enumeration to find the optimal solution, and the time for n-tuple enumeration to find the optimal solution. The range of the number of items allowed was initially 3-5, but was increased to 6 in order to gather

more meaningful data. For each of these cases, the maximum number of copies allowed for each unique item was initially varied between just 1,1 or 2, 1-3, 1-4, but was also increased to include 1-5, and 1-6 to improve the final data collected. The number of copies within the above ranges was chosen at random, with each number in the range having equal probability of being chosen. These ranges were decided upon through experimentation with the algorithms. It was found that on average, anything larger than this would take an unreasonable amount of time to complete. To gather our data, we used a data gathering bash script. The first set of data that we analyzed was a comparison between the maximum weight of the knapsack and the time it took each algorithm to find the optimal solution through brute force.

3 Empirical Studies

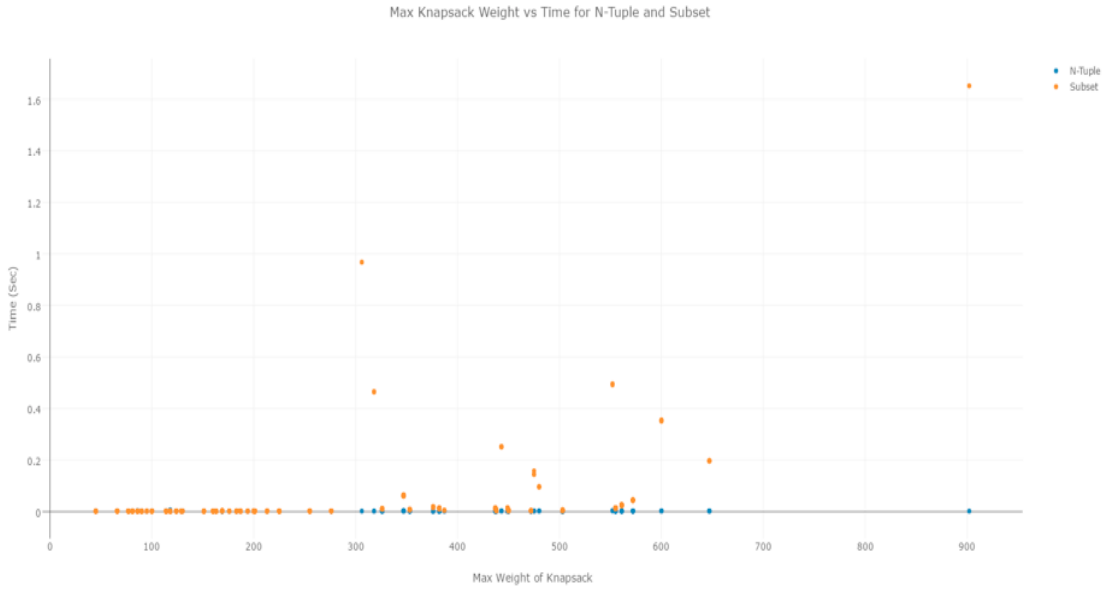

Figure 3.1: Max Knapsack Weight vs.Time for N-Tuple and Subset

For the n-tuple data, our data showed that for our test values, n-tuple enumeration would take about 0.002 seconds to find calculate the optimal solution through brute force, regardless of the maximum size of the knapsack. This, however, was due to the small set of items that were being tested. Conversely, in the case of the subset enumeration, the data shows that as the maximum weight of the knapsack increases, the time needed to find the optimal solution also starts to increase. In our data, once the size of the knapsack started to exceed 300, the time needed for subset enumeration also started to increase significantly. This is more apparent in the edge case with a max weight of 902. For the n-tuple enumeration, the time was still relatively fast around 0.002 seconds.

However, for the subset enumeration, the time was 1.652 second which meant that the subset enumeration algorithm was over 800 times slower than the n-tuple algorithm for this specific case.

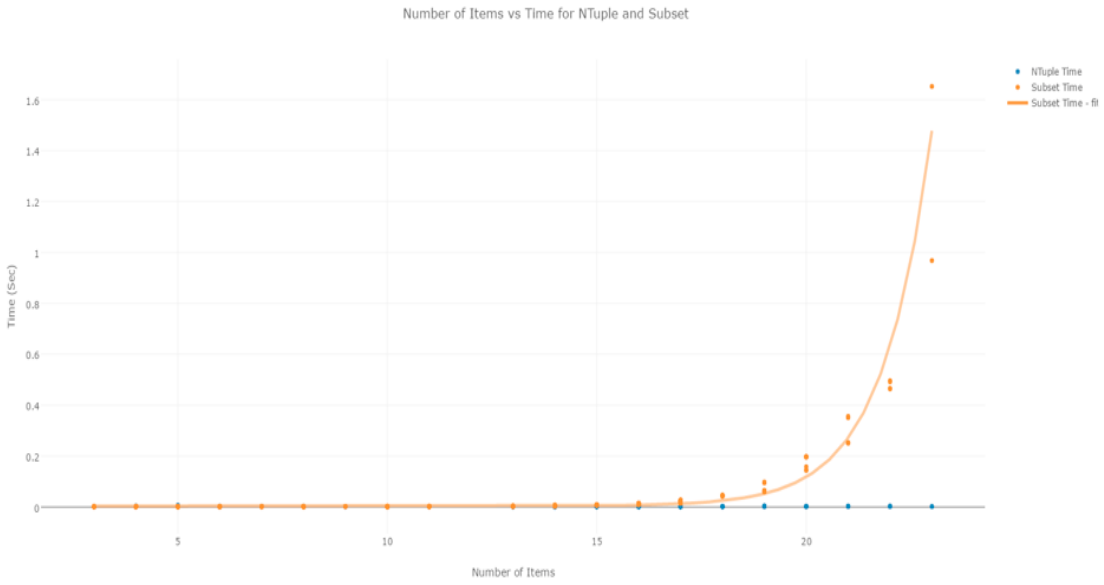

Figure 3.2: Number of Items vs. Time for N-Tuple and Subset

For this set of data, the performance difference between n-tuple enumeration and subset enumeration become much clearer. Even as the total number of items increased, the time for the n-tuple enumeration to find the solution still remained relatively low at around 0.002 seconds. However, as the graph confirms, for the subset enumeration algorithm, as the number of items increases, the time needed to find the optimal solution increases exponentially. Once the number of items reaches 17, the subset enumeration algorithm begins to take noticeably longer to complete compared to the n-tuple algorithm and by 20 the difference in the time is significant. For example, for the case with 23 item total, the n-tuple cases both took 0.002 seconds to complete, while the subset enumeration cases took 0.968 seconds and 1.652 seconds to complete.

To understand why the subset enumeration algorithm is significantly slower than the n-tuple enumeration algorithm, it is necessary to understand the difference in the total number of permutations each algorithm has to process. Consider the following set of data:

Item A with max copies of 3
Item B with max copies of 4
Item C with max copies of 5

In the case of the n-tuple algorithm, the maximum allowed copies of each unique item is a single entry in an array. For the given data, the array of maximum values would be the following:

$$[\ 4 \mid 5 \mid 6\ ]$$

Where index 0 corresponds to the maximum number of items allowed for item A + 1, etc. The reason for incrementing by one is to simplify the generation of each permutation in the algorithm. With this representation, the number of unique permutations is given by:

$$\textit{N-Tuple Enumeration Permutation Total} = 3 \times 4 \times 5 = 60$$

Conversely, in the case of subset enumeration, each individual item must be represented by its own index in the array. Therefore, for the given data, the array of maximum values would be the following:

$$[\ 1 \mid 1 \mid 1 \mid 1 \mid 1 \mid 1 \mid 1 \mid 1 \mid 1 \mid 1 \mid 1 \mid 1\ ]$$

In this representation, index 0 would correspond to item A1, index 1 is item A2, ... , index 3 is item B1, ... , index 6 is item B5, index 7 is item C1, ... , and index 11 is item C6. Therefore, in this form, the total number of permutation that must be examined is given by:

$$\textit{Subset Enumeration Permutation Total} = 2^{12} = 4096$$

Therefore, for this specific example, subset enumeration has to examine over 60 times as many permutations as n-tuple enumeration. As the number of total items increases, so difference in the number of permutations each algorithm analyzes will grow significantly.

The cause of this is due to the how inefficiently subset enumeration generates all possible permutations. Subset enumeration is inefficient because it treats each item as unique rather than the same. For example, consider item A in the above example data and consider all the cases consisting of just having 1 copy of item A. In subset enumeration, having only item A1 in the set is considered different then having only item A2 or A3. However, in the case of the n-tuple enumeration, the permutation only considers having one of item A in the set. It doesn‘t matter which specific copy of A it is because they are all the same. This inefficiency results in subset enumeration wasting time comparing permutations it has already done, but is unable to notice because it is treating each item as unique.

Conclusion: Our results show that for small sets of data, n-tuple enumeration tends to performs significantly better than subset enumeration. This is primarily due to n-tuple enumeration being able to filter out comparisons of permutations that it has already examined. However, it is important to realize that despite these results, they are only valid for the small sets of data that were examined here. In reality, if the test values included larger numbers of unique items with larger number of copies for each unique

item, the computation times of the n-tuple algorithm would drastically increase. This is due to the NP nature of the knapsack problem. In reality, both algorithms are brute force methods that are performing exhaustive searches to find a solution. There will be a point for both algorithms in which the number of items in the data set results in the algorithm requiring an increasingly larger amount of time find the optimal solution.