Due Date: 31.05.2019

CSE 2046.1 ANALYSIS OF ALGORITHM

HOMEWORK #3

COURSE INSTRUCTOR*: Assistant Prof. Ömer Korçak*

COURSE T.A *: Muhammed Nur Avcil*

GROUP MEMBERS

150116065 - Deniz Arda Gürzihin
150117509 - Mustafa Abdullah Hakkoz

1.0) *What is Knapsack problem?*

The knapsack problem or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

2.0) *0 – 1 Knapsack Problem*

The most common problem being solved is the **0-1 knapsack problem**, we are given a knapsack with finite capacity and a set of n items, where each item *i* is associated with a weight w_i and a value v_i. The objective of the KP is to select a subset of items such that the total value z is maximized, while the total weight b does not exceed the capacity of the knapsack.

2.1) Algorithm Explanation

During our preperation phase for this project, we found many approaches to get optimum value, which are three common methods to do it, **dynamic programming[1]**, **branch & bound[2]** and some approximation algorithms like **greedy aproximation[3]** and **polynomial time approximation scheme[3].** In additionaly, we figured some further methods like **genetic algorithms[4]** and some local search methods like **one-bit-flip[5]**, **tabu search[5][6]** and **memetic algorithms[5]**.

When we compare each one of their performance, based on some research papers[4], we saw that branch&bound method stays behind since it is just an improvement over exhaustive search with O(n^2) complexity. So we choosed it's rival, dynamic programming (O(W*n)) to produce exact solutions as our control group and greedy aproximation (O(nlogn)) to create our main algorithm.

- First we did our experiments with basic greedy algorithm (with **Question1-basicGreedy.c** which produces 1 file *output-basicGreedy-filename.dat*) as follows;

Step 1 calculates the ratio for n inputs between weight and values as (density=value /weight).

Step 2 re-aarrange the array with the items in a non-diminishing order as per of the ratio value. (with built-in quick sort function: qsort())

Step 3 picks the biggest ratio of the item that its weight is not exactly or equivalent to the W (knapsack limit) to add it to the solution set with it's original index (which is stored in item structure).

Step 4 repeat **Step 3** for all items and return optimum value of the sack.

- Then we tried some modification on basic greedy algorithm inspired from some local search methods pointed out above (with **Question1-modifiedGreedy.c** which produces 2 files *output- filename.dat* for modified greedy aproximation and *output-DP-filename.dat* for dynamic programming) as follows;

In addition to same 4 steps of basic greedy algorithm;

Step 5 find the element with biggest ratio which is not in solution set (i.e. arr[x]→remark that solution set has x elements).

Step 6 swap it with the first element of the array (arr[0]<->arr[x]) and calculate a new solution set. If new solution set has better value then the original one(which is calculated on **Step 4**), return it.

Step 7 shift the beggining of the array one element to right to discard first element (arr[0]) and calculate a new solution set. If new solution set has better value then the original one, return it.

Step 8 repeat **Step 7** by shifting j element for j=0,1,..,x (x is size of original solution step).

Step 9 repeat Step 6, 7, 8 by incrementing i while swapping i*th* element and arr[x] (arr[0+i]<->arr[x]).

Step 10 set time limit for Step 5-10 (i.e t_{limit} = runtime_of _dynamicprogramming/10)

Briefly, we combined 2 modifications. First we swap the elements in solution set with an alternative candidate(arr[x]). Second we discard the elements in solution set one by one to try out new optimum values.

2.2) Test Outputs

	Dynamic Programming	***Basic Greedy Approximation***	***Modified Greedy Approximation***
test1a.dat	2697	2649	2649
test1b.dat	18051	18038	18038
test1c.dat	146919	146888	146888
test1d.dat	-	4241499	4241499

We can see our results (Greedy aproximation) are so close to exact values (Dynamic Programming) so we can say that we succeed on this part of experiment. But we didn't have the chance to observe difference between our modified version with basic greedy. Maybe for some other test inputs they may differ.

Additionaly we did not calculate dynamic progrraming outputs for tes1d since it has big integers. We provided special version of our modified greedy code (**Question1-modifiedGreedy-bigint.c**) which excludes dynammic programming part.

2.3) Running Times and Complexity Analysis

	Dynamic Programming		***Basic Greedy Approximation***		***Modified Greedy Approximation*** *(t_{limit}=dynamic/10)*	
(secs)	*Only algorithm*	*Whole code*	*Only algorithm*	*Whole code*	*Only algorithm*	*Whole code*
test1a.dat (n=200)	0.003	0.049	0.001	0.003	0.001	0.038
test1b.dat(n=2000)	0.369	0.419	0.001	0.089	0.130	0.195
test1c.dat(n=10000)	13.949	19.273	0.005	0.130	3.471	3.976

test1d.dat(n=2000 but big integers)	-	-	0.001	0.043	0.001	0.004

Again we did not calculate dynamic progrraming outputs for tes1d since it has big integers. We used special version of our modified greedy code (**Question1-modifiedGreedy-bigint.c**) instead of (**Question1-modifiedGreedy.c).**

We can observe that our modified version has better runtime than dynamic programming algorithm, but a little bit more than basic greedy algorithm.

Time complexity of **Step1**->O(n), **Step2**->O(nlogn), **Step4**->O(n). Thus general complexity of basic greedy algorithm is O(n)+O(nlogn)+ O(n) = O(nlogn)

On our modified algorithm, we increased complexity of basic greedy algorithm on **Step 8** (x times) and **Step 9** (x times) with 2 nested loops. So our general complexity is $O(x^2 n\log n)$ for x=size of original solution set. In most of cases, for small values of x, complexity can be considered as O(nlogn). For big values of x, it tends to become $O(n^2\log n)$ but another constrait of our code, time limit, moves in and restrict it to not exceed pre-defined running time value (in our code one-tenth of running time of dynamic programming).

3.0) Multiple Knapsack Problem

The Multiple Knapsack Problem (MKP) is the problem of assigning a subset of *n* items to *m* distinct knapsacks, such that the total profit sum of the selected items is maximized, without exceeding the capacity of each of the knapsacks.

3.1) Algorithm Explanation

For this problem, we have started with making research on the Internet and we have not found any implementation that could show us the way so we have coded our algoritm. Our algorithm is not so complicated.

Lets start with the important variables we have used on coding part.

- int numberOfItems for to se how many values and weights has been given,
- int num_Knapsack[] for how many knapsack will be used on solution,
- int values[] for storing the values in txt file,
- int weights[] for storing the weights part in txt file and additionally
- double Fractional[] for store the Fractional importance with calculating from the values[n]/weights[n]
- int solSet[][] for which element we have been used

Our algorithm is based on Fractional importance so we have created a while loop in order to test all possible elements where the condition is “any Fractional[] value is not equal to zero.” Then we have called the findMax() function which is finding the biggest value in the array return the index of this element and we have initialized this as an integer y value. After this, we have created a for loop the test all elements where the increment value is the number of Knapsack in the text file, we have started with findMax() again in order to find the index in for loop, then we have checked that is there enough space in Knapsack, if there is not with for loop we have checked the next Knapsack with array index, if there is then we have decreased the net capacity of the index by the weights of the element we have found, increased the net value on Knapsack by the value of the element, initialize the solSet[][] to show we have used this element and finally initialize the Fractional[y] value as 0.0. That was the end of the for loop. After for loop, we have initialized the Fractional[y] value as 0.0 again because if we want to continue in while loop we want to be sure about the value of Fractional and if we don't make this assignment while we can not find any elements to fit in Knapsack in for loop, first while loop cannot finish and we will enter the infinite loop. And in the last, we have found again and index with findMax() function as integer x value and this value is needed for the while loop condition to check Fractional[x] value is equal to zero or not. If the assignment of x becomes zero; the while loop has ended.

FindMax() is a function that finds the biggest value in array and returns the index of it. With O(n) time complexity.

Step By Step Description

Step 1 -> Read the values from the .dat file and make the valid assingment
Step 2 -> Then call the multiple Knapsack function with variables from step 1
Step 3 -> Initialize the array value knapsack[] by zero
Step 4 -> Find the maximum fraction in the .dat file
Step 5 -> Until Fraction[] become zero, find the maximum fraction again and create a for loop with going zero to number of knapsack times while doing this initiliaze an integer variable as the findMax() function to find the index of the maximum fraction, then check is there enough capacity in knapsack or, if not then make Fraction that we have used as 0.0 and find the maximum fraction index; if yes decrease te total capacity of the knapsack by weights[] at that index, increment the total value of knapsack by value[] at that index, make solset[][] as 1 to show we have used that element and finally assign the Fraction[] at that index to zero.
Step 6 -> Return the Total Value Of Knapsack from multiple_Knapsack function and assign this to an integer
Step 7 -> Print the total value and solution set into a dat file

3.2) Outputs and Time Complexity

Input File/Outputs	Total_Value	First Timer(s)	Second Timer(s)
test2a.dat	295	0.00400	0.00200
test2b.dat	423	0.00600	0.00300
test2c.dat	369	0.00800	0.00600
test2d.dat	326	0.00700	0.00500
test2e.dat	146485	2.888	2.557

Time complexity of our algorithm for multiple knapsack is O(n* m) where n is the number of items and m is the total number of knapsacks.

REFERENCES:

[1] Geeksforgeeks, Printing Items in 0/1 Knapsack, 31.05.2019:

https://www.geeksforgeeks.org/printing-items-01-knapsack/

[2] Geeksforgeeks, 0/1 Knapsack using Branch and Bound, 31.05.2019:

https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/

[3] Yaron Singer, AM 221: Advanced Optimization lecture notes, 30.03.2016:

https://people.seas.harvard.edu/~yaron/AM221-S16/lecture_notes/AM221_lecture17.pdf

[4] Ameen Shaheen and Azzam Sleit, Comparing between different approaches to solve the 0/1 Knapsack problem, July 2016:

https://www.researchgate.net/publication/306021086_Comparing_between_different_approaches_to_solve_the_01_Knapsack_problem

[5] Suwan Runggeratigul, Local Search and Memetic Algorithms for Knapsack Problems, MIC2003:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.5572&rep=rep1&type=pdf

[6] Mhand Hifi, Slim Sadfi, Abdelkader Sbihi, An Efficient Algorithm for the Knapsack Sharing Problem, 20.04.2006:

https://hal.archives-ouvertes.fr/hal-00023189/document

[7] Silvano Martello, Paolo Toth, Knapsack Problems, John Wiley & Sons, 1990

http://www.or.deis.unibo.it/kp/KnapsackProblems.pdf