Algorithmic Operation Research
Homework 6

Theodora Panagea - 1115201400135
Anna-Aikaterini Kavvada - 1115201500050

23-1-2020

Exercise 1

Define the 0-1 knapsack problem.

To begin with, in $0-1$ knapsack problem we have the primal restriction of the number x_i of copies of each item to zero or one, since we cannot intersect the items available and get just part of them.

Now, consider a set of n items, numbered from 1 up to n. Each item has a weight w_i and a value v_i. Also, for the problem, consider our knapsack has a maximum weight capacity W. Our goal, is to maximize the the sum of the item values in the knapsack, so that the weights is less than or equal to the maximum capacity W. Thus, we get the following problem expression:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \quad \text{and} \quad 0 \leq x_i \leq 1$$

Exercise 2

Give a dynamic programming solution to the 0-1 knapsack problem.

In order to solve the 0-1 knapsack problem with dynamic programming, we have to come up with an optimal substructure and overlapping subproblems, both of which are indeed present in the 0-1 knapsack problem.

So, let's suppose we have a knapsack that can hold w weight units. We have a total of n items to choose from, each one of them having its own value and weight. As a reminder, this is the 0-1 knapsack problem, so we either include or exclude an item in our knapsack.

Now that we have defined the problem, a solution - algorithm could be the following:

Step 1: Firstly, we create a 2-dimensional array, with $n + 1$ rows and $w + 1$ columns.

A row number i represents the set of all the items 1 to i. For example, the values in row 3, assumes that we have only the items 1,2 and 3.

A column number j represents the weight capacity of our knapsack. Therefore, the values in column 3, assumes that our knapsack can hold up to 3 weight units.

Step 2: We start filling the base cases in our table, for which the solution is trivial. For instance, at row 0, we have no items to pick from, so the maximum value in any knapsack is 0. Similarly, at column 0, the knapsack can hold 0 weight units, so there is no maximum value to begin with, and we put 0, since where are no massless valuable items.

Step 3: Here, at each step, we will make use of our solutions to the previous sub-problems. The relation between the rows, columns and the previous sub-problems is as follows:

The element on the i-th row and j-th column, is the maximum value of a sub-problem consisting of items $1, 2, \ldots, i$ with a knapsack of j capacity. In each sub-problem, we can either include item i or not. Therefore, each time, we compare the maximum value we can obtain with item i and without it.

Of course, the latter can be found at row $i-1$, column j. But what about the first option?

To calculate that, firstly we compare the weight of the i-th item with

the knapsack's weight capacity. Obviously, if item i weighs more than the knapsack's capacity, it is easily excluded, and we don't have to make any further calculation. In that case, we fill the maximum value with the one we can obtain wiwthout item i ($1 - i$ row, same column). When however. the item i weighs less, we have the option to include it, if it potentially increases the maximum obtainable value. The maximum obtainable value by including item i is consequently:= the value of item i + the maximum value that can be obtained with the remaining capacity of the knapsack. We obviously want to make full use of the capacity of our knapsack. Therefore, at row i and column j (which represents the maximum value we can obtain there), we would pick either the maximum value that we can obtain without item i, or the maximum value that we can obtain with item i, whichever is larger.

Step 4: This is the final solution. Once the 2-dimensional array has been filled, the solution can be found at the last row and last column, which represents the maximum value obtainable with all the items and the full capacity of the knapsack.

Exercise 3

Give real world applications of knapsack problem.

The most simple and obvious real life application of the knapsack problem is the following. Consider being a traveller/nomad in todays world. This means that the total weight of all your wordly possesions must fall under airline cabin baggage weight limits - usually 10kg. On some smaller airlines, however, this weight limit drops to 7kg. Occasionally, you have to decide not to bring something with you to adjust to the smaller weight limit.

So you have to decide what to leave behind (or get rid of altogether), which entails laying out all your belongings and choosing which ones to keep. That decision is based on the item's usefulness to you (/cost) and its weight, so that i can pack as more items as possible in the limited space you get.

The knapsack problem is an example of a combinational optimization problem, a topic in mathematics and computer science about finding the optimal object among a set of objects. This is a problem that has been studied for more than a century and is a commonly used example problem in combinatorial optimization, where there is a need for an optimal object or finite solution where an exhaustive search is not possible. The problem can be found real-world scenarios like resource allocation in financial constraints or even in selecting investments and portfolios. It also can be found in fields such as applied mathematics, complexity theory, cryptography, combinatorics and computer science. It is easily the most important problem in logistics.

In the knapsack problem, the given items have two attributes at minimum - an item's value, which affects its importance, and an item's weight or volume, which is its limitation aspect. Since an exhaustive search is not possible, one can break the problems into smaller sub-problems and run it recursively. This is called an optimal sub-structure. This deals with only one item at a time and the current weight still available in the knapsack. The problem solver only needs to decide whether to take the item or not based on the weight that can still be accepted. However, if it is a program, re-computation is not independent and would cause problems. This is where dynamic programming techniques can be applied. Solutions to each sub-problem are stored so that the computation would only need to happen once. Some other applications of the knapsack algorithm can be found in

1. Home Energy Management

2. Cognitive Radio Networks
3. Resource management in software
4. Large-scale multi-period precedence constrained knapsack problem: A mining application
5. Relay selection in secure cooperative wireless communication
6. Power allocation management
7. Selecting adverts garden city radio
8. Solve the production planning problem
9. Cognitive radio networks
10. 5G mobile edge computing
11. Selection of renovation actions
12. Solve the production planning problem
13. Sensor Selection in Distributed Multiple-Radar
14. Architectures for Localization
15. Appliance Scheduling Optimization for Demand Response
16. Adaptive Variable Density Sampling
17. Secure Cooperative Wireless Communication
18. Optimizing Power Allocation to Electrical Appliances
19. Computation Offloading in Wireless Multi-Hop Networks
20. Formulation and Solution Method of Tour Conducting
21. Plastic Bags Waste Management Using the Knapsack Model
22. Workflow mapping using CUDA
23. Optimization of Content Delivery Networks

24. Network Selection for Mobile Nodes

Another application is the one described below.

Mapping and running jobs on suitable resources are the core tasks in grid computing. In the algorithm to map light communication Grid-based workflow within the SLA context, there is an operation of resolving the conflict period which is exact a 0/1 knapsack problem. When the size of the workflow is large such as in the case of mapping a group of workflows, the time to solve this problem is long and thus, makes the whole mapping process long. In this paper, we describe a way to solve this problem by exploiting the parallel computing power of graphic processing unit (GPU) with compute unified device architecture (CUDA). The experiment shows that the approach is very efficient with huge problem.

Link to paper: *Light Communication SLA-Based Workflow Mapping Using CUDA*

Exercise 4

Define the Subset sum problem and give a dynamic programming solution for it. Write down the difference between the subset sum and the knapsack problem.

Subset Sum Problem: Given a set of positive integers, and a value sum S, find out if there exists a subset in array whose sum is equal to given sum S.

0-1 Knapsack and Subset Sum are two well-known NP-complete, closely related problems. Apparently, the methods for solving Knapsack are generally the same as for Subset Sum. So, we can say that Subset Sum is a special case for Knapsack. Both problems have the same complexity, when the complexity parameter is n, but Knapsack appears to be easier when the complexity parameter is x, the total bit length of the input.

In implementation, compared to the knapsack's approach (Exercise 2), the Subset Sum problem needs a boolean 2-dimensional array, being filled with in a bottom up manner. An element at row i and column j denotes if there is a subset of sum j with element at index $i-1$ as the last element, therefore that value will be true if there is a subset of set$[0,\ldots,j-1]$ with sum equal to i, otherwise it will be false. Of course, all elements on column 0 are true, because the sum of no elements is 0, and the elements on row 0 are false, because with no elements we get no sum. If an element at index i is greater than j, then the value of the subproblem is false, as we cannot get a subset of positive numbers with the latter element as a member. Again, the solution is the element at the last row and column.