COMP26120
Academic Session: 2018-19

Lab Exercise 12: The 0/1 Knapsack Problem

Duration: 3 lab sessions. **WARNING: You will find it difficult to complete this exercise if you do not work for all three weeks.**

For this lab exercise you should do all your work in your COMP26120/ex12 directory.

Learning Objectives

By the end of this lab you should be able to:

- To explain the 0/1 Knapsack problem and Fractional Knapsack problem.
- To implement a number of exact techniques for solving the 0/1 Knapsack problem.
- To implement one inexact technique - or heuristic - for finding good but not necessarily optimal solutions to the 0/1 Knapsack problem.
- To compare the time complexity and running times of these techniques.

Introduction

In this section we introduce two related 'Knapsack' problems.

The 0/1 Knapsack Problem and Logistics

Suppose an airline cargo company has 1 aeroplane which it flies from the UK to the US on a daily basis to transport some cargo. In advance of a flight, it receives bids for deliveries from (many) customers.

Customers state

- the weight of the cargo item they would like to be delivered;

- the amount they are prepared to pay.

The company must choose a subset of the packages (bids) to carry in order to make the maximum possible profit, given the total weight limit that the plane is allowed to carry.

In mathematical form the problem is: Given a set of N items each with weight w_i and value v_i, for $i = 1$ to N, choose a subset of items (e.g. to carry in a knapsack, or in this case an aeroplane) so that the total value carried is **maximised**, and the total weight carried is less than or equal to a given carrying capacity, C. As we are maximising a value given some constraints this is an *optimisation* problem.

This kind of problem is known as a **0/1 Knapsack problem**. A Knapsack problem is any problem that involves packing things into limited space or a limited weight capacity. The problem above is "0/1" because we either do carry an item: "1"; or we don't: "0". Other problems allow that we can take more than 1 or less than 1 (a fraction) of an item. Below is a description of a fractional problem.

See the description in Goodrich and Roberto, p. 498. or the briefer description in Cormen, Leiserson, Rivest and Stein, p. 425.

The Fractional Knapsack Problem

Now consider that the cargo company introduces a new kind of flight transporting expensive liquids e.g. petrol or aftershave. Now customers state the amount they are prepared to pay for a certain weight to be transported but the airline company can choose to carry a fraction of this. In this setup we ignore the overhead of carrying different kinds of liquid on the same flight.

In mathematical form this is: Given a set of N items each with weight w_i and value v_i, for indexes i=1 to N, choose the amounts x_i, of each item to carry, where x_i is **any real number** in the range 0 to w_i, so that the total value carried is maximised, and the total weight carried is less than or equal to a given carrying capacity, C.

An Enumeration Method for solving 0/1 Knapsack

A straightforward method for solving any 0/1 Knapsack problem is to try out all possible ways of packing/leaving out the items. We can then choose the most valuable packing that is within the weight limit.

For example, consider the following knapsack problem instance:

Sample Input
3
1 5 4
2 12 10
3 8 5
11

The first line gives the number of items; the last line gives the capacity of the knapsack; the remaining lines give the index, value and weight of each item e.g. item 2 has value 12 and weight 10.

The full enumeration of possible packings would be as follows:

Items Packed	Value	Weight	Feasible?	
000	0	0	Yes	
001	8	5	Yes	
010	12	10	Yes	
011	20	15	No	
100	5	4	Yes	
101	13	9	Yes	OPTIMAL
110	17	14	No	
111	25	19	No	

The items packed column represents the packings as a binary string, where "1" in position i means pack item i, and 0 means do not pack it. Every combination of 0s and 1s has been tried. The one which is best is 101 (take items 1 and 3), which has weight 9 (so less than $C = 11$) and value 13.

Some vocabulary

- **A solution:** Any binary string of length N is referred to as a packing or a solution; This only means it is a correctly formatted instruction of what items to pack.
- **A feasible solution:** A solution that also has weight less than the capacity C of the knapsack.
- **An optimal solution:** The best possible feasible solution (in terms of value).
- **An approximate solution:** Only a high value solution, but not necessarily optimal.

In this lab we will investigate some efficient ways of finding optimal solutions and approximate solutions.

Branch and Bound

We now briefly describe the branch and bound approach. This was briefly mentioned in lectures. You should also see pages 521-524 of Goodrich and Roberto.

Let's consider how we might, as people, solve the 0/1 knapsack problem. We would try the 'best' item first (e.g. the one with the highest value-to-weight ratio) and see if we can fit the best items in. This is also a reasonable approach for a computer e.g. to sort the items in descending order of value-to-weight ratio, and add items in that order until the knapsack is full. However, we might realise that putting that really big item in the bag means there is left over space and putting two smaller items in would have been better i.e. we **back-track** by removing an item and trying something else. We get computers to do the same thing when systematically exploring the search space.

In order to prevent us backtracking through every possible solution, we can "prune" off parts of the set of the solutions we know cannot contain a feasible/optimal solution. If we know that a particular subset of items is heavier than the capacity, we do not need to consider any solutions that use that subset. Similarly, if we know that not including a particular item (e.g. the first item) the best we can do is value v_{upper}, and we have already found a

solution better than v_{upper}, then we no longer have to consider any solution that does not contain that crucial item.

Description

This lab asks you to implement four different solutions to the 1/0 Knapsack problem over three weeks. We have provided partial solutions and it is your job to complete them.

Task 1a: Full Enumeration

Take a look at the files.

- *Enum.c* implements the full enumeration of all possible solutions as described above.
- *knapsack-util.c* provides some functions to read in a knapsack instance, print the instance details, print out a solution, and evaluate the values and weights of a solution. There's no need to edit *knapsack-util.c* during this lab, but you may do so if you wish.

Make enum and run it on *easy.20.1.txt*, which is a 0/1 knapsack problem instance with 20 items to pack:

```
make enum
./enum easy20.1.txt
```

The program enumerates the value, weight and feasibility of every solution and prints them to the screen. However, it does not "remember" the best (highest value) feasible solution or display it at the end.

1. Adapt the code so that it does that.
2. It would also be useful to display how much of the enumeration has been done - like a progress bar. Add code to the enumeration loop to print out the fraction of the enumeration that is complete.

You may change the value of the QUIET variable to suppress output to the screen and make the code run faster.

Task 1b: Dynamic Programming

Before you implement your solution by dynamic programming let us check that it satisfies the requirements. In a file `dp.txt` write a description as to why the 0/1 Knapsack Problem has the three necessary properties (see p. 278 of Goodrich and Roberto):

1. Simple Subproblems
2. Subproblem Optimality
3. Subproblem Overlap

Now complete the program that solves the 0/1 Knapsack Problem by dynamic programming. Use the files *knapsack-util.c* and *dp.c* we provide for you. Further instructions are in *dp.c*. The suggestion is to follow the pseudocode found on slide 17 of http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf but you can take your own approach. There is a detailed explanation with examples on p 278-281 of Goodrich and Roberto.

Test your code on the given instance file, *easy.20.1.txt*. You should get the same answer as with the enum program.

You should be here by the end of week 1 of this lab.

Task 2a: Fractional Knapsack Bound

Imagine we have decided we definitely want to pack some particular items, and definitely don't want to pack some particular other ones. For the remaining items we are not sure yet. We can represent this situation as

001110*****

where 0 means definitely won't take the item, 1 means definitely will, and * means don't know. We call these **partial solutions**.

We can now calculate an estimate of the best value of all possible ways of replacing the *s by 0s and 1s. This will be an overestimate. We call it an upper bound.

To calculate the estimate we take the * items in decreasing order of value-to-weight ratio and add them to the knapsack, until we go over the capacity. For the last item added, we take it out again and only add that fraction of its weight that would fit, and adding the same fraction of its value to the total value of the knapsack. This is a kind of cheating; however, we are only interested in making an estimate.

In the *bnb.c* code, we have already provided an almost complete function $frac\_bound()$ which accepts a partial solution of the form 01101****** as input and does the following things:

1. Checks the feasibility of the partial solution and sets the solution value to -1 if it is infeasible, and returns.
2. If the partial solution is feasible then its value is calculated, i.e. the value of the items already packed, and the value is updated
3. If the partial solution is feasible then its upper bound is also computed and updated.

Make sure you understand this $frac\_bound()$ function and complete it by filling in the two missing lines.

Task 2b: Branch-and-Bound

You must now use the $frac\_bound()$ function to complete the branch-and-bound implementation. The outline of the algorithm can be given as follows (see Goodrich and Roberto for more details).

1. Sort the items by decreasing value-to-weight ratio.
2. Compute the upper bound of the solution ****...
3. Compute the current values of each of the two solutions 0***... and 1***... (i.e. the total value of all the 1s in each string), also their upper bound values, and check they are feasible.
4. If they are feasible we place them on a priority queue.
5. In the next and all subsequent iterations, we remove the item with best bound value off the priority queue and again consider appending a 0 and a 1 to it.
6. The algorithm stops when the queue is empty or a solution (a complete solution with no stars in it) with value equal to the current upper bound is found.

Complete the *branch_and_bound* function given in *bnb.c*. This will call the fractional knapsack bound function $frac\_bound()$ and use the priority queue functions provided. More instructions are given in the code provided.

You should be here by the end of week 2 of this lab.

Task 3a: Greedy Algorithm

The greedy algorithm is very simple. It sorts the items in decreasing value-to-weight ratio. Then it adds them in one by one in that order, skipping over any items that cannot fit in the knapsack, but continuing to add items that do fit until the last item is considered. There is no backtracking to be done.

Write your own greedy algorithm in *greedy.c*.

A greedy algorithm is not optimal for the 0/1 Knapsack Problem but what about the Fractional Knapsack problem? In `greedy.txt` explain why a greedy approach is not necessarily optimal for 0/1 and state with explanation whether it is optimal for the fractional case.

Task 3b: Testing and Comparing Methods

You should answer the following questions in `answers.txt`. You should make notes in your lab book of your answers to the following questions. The lab demonstrator will ask you these questions during marking.

You should now have four methods for solving a knapsack problem: enumeration, dynamic programming, branch-and-bound and greedy-heuristic.

1. For large instances, you cannot use enumeration. Why? How large an instance do you think you can solve on the lab PCs using enumeration? (An accurate answer is not needed, so you can assume that one evaluation of a solution takes 1 microsecond, you do not need to run anything, this question is a thought exercise)
2. Run the other three algorithms on the following knapsack problem instances and note what happens.

easy.200.4.txt
hard1.200.11.txt
hard1.2000.1.txt

Which instances does greedy solve optimally? Does dynamic programming work on all instances, and why/why not? Does branch-and-bound come to a stop on all instances?

3. Can you explain WHY the hard1 instances are easy or hard (cause problems) for (i) greedy, (ii) branch-and-bound and (iii) dynamic programming? This question is quite tough. Do not attempt it if you are running out of time.

4. The airline has problems of size 500-2000 of similar type to the hard1 instances. Which algorithm(s) do you recommend using and why? What should they do in case the algorithm runs out of time?

Marking Scheme

Part 1A. Is the Full Enumeration solution correctly completed? (next question addresses progress bar)	
The code is correctly completed and the student can explain and demonstrate how it works. The student can also explainhow a solution is encoded in the given code and how the give code enumerates through these e.g. what the order of enumeration is.	(3)
The code is corect as above but some of the explanation demonstrates areas of misunderstanding, the student can explain how a solution is encoded.	(1.5)
The code is correct as above but the student cannot explain how a solution is encoded.	(1)
There are some mistakes in the solution e.g. it may not always produce the correct result.	(1)
No attempt has been made	(0)

Part 1A. The progress bar has been implemented correctly	
The progress bar has been fully implemented correctly	(1)
There is a minor mistake in the implementation of the progress bar	(0.5)
No attempt has been made or the implementation does not work	(0)

Part 1B. The justification for being able to use dynamic programming (in terms of satisfying the necessary requirements) is reasonable.	
All three areas are addressed with sensible justifications. The student may have used an example. Answers should be in full sentences and demonstrate some thought has been given to the questions.	(2)
An attempt is made to address all three areas but the answers might be brief and may contain some misconceptions.	(1.8)
At least one point is addressed reasonably	(1)
An attempt has been made	(0.5)
No attempt has been made	(0)

Part 1B. The dynamic programming solution has been correctly implemented (understanding checked next)	
The solution works as expected and has evidently been tested on many input files (the student should demonstrate this).	(2)
There are minor mistakes that lead to the wrong solution in some edge cases but in general the solution is correct	(1.8)
There are major mistakes that lead to the wrong solution in many cases	(1)
An attempt has been made	(0.5)
No attempt has been made	(0)

Part 1B. The student understand the dynamic programming solution	
The student can explain how the dynamic programming solution work conceptually (not at a low-level algorithmic level but a conceptual problem level). For example, the student can explain why V[i][w] is given the value it is given.	(2)
The student has a reasonable understanding of the dynamic programming solution but not at the level of detail required above e.g. the understanding might be quite low-level.	(1.5)
The student has a broad understanding of the idea behind the dynamic programming solution but cannot explain the details	(1)
The student can explain what the idea is behind dynamic programming but not how it relates to this problem	(0.5)
No attempt has been made	(0)

Part 2A. The frac_bound function is understood and completed	
The student has completed the two missing lines in frac_bound() and can explain what the function is computing. The student can explain what a feasible solution is and what the value of a solution is and how both concepts apply to partial solutions.	(2)
As above (function correctly completed) but some understanding is missing	(1.8)
The solution has a minor flaw	(1)
An attempt is made but it does not work and is not understood	(0.5)
No attempt has been made	(0)

Part 2B. The branch and bound code is completed correctly	
The code works as expected and can be demonstrated correct on multiple input problems.	(4)
There is a minor mistake in the implementation that leads to an incorrect result in some edge cases	(3.5)
The solution is complete but there is a reasonably large mistake in the implementation that leads to an incorrect result in many cases	(3)
The solution is incomplete but what has been implemented seems correct	(2)
An attempt has been made	(1)
No attempt has been made	(0)

Part 2B. The student can explain the branch and bound solution	
The student can explain the general method and how the frac_bound() function is used. The student may sketch the search tree being explored and explain how this might evolve and how it relates to the encoding of solutions. The student can explain the role of the priority queue. The student can justify why the method works. The student can relate this all to the code that they have written.	(3)
The student can explain the general method, including the role of the frac_bound() function, and how it relates to the code that they have written.	(2.5)
As above but with some areas of misunderstanding	(2)
The student can explain most but not all of their code	(1)
The student can explain some but not most of their code	(0.5)
No attempt has been made	(0)

Part 3A. The greedy solution is correct and can be explained	
The solution works as expected and has been tested on multiple inputs. The student can explain what the idea is behind greedy algorithms and how this is acheived in their code.	(3)
The solution works but the explanation is lacking in clarity.	(2.5)
There are minor flaws in the implementation	(2)
There are major flaws in the implementation	(1)
An attempt has been made	(0.5)
No attempt has been made	(0)

Part 3A. The discusion of the optimality of the greedy solution shows understanding	
The analysis is correct and well-thought through. Justification for the non-optimality for the 0/1 case need not be formal, it may be by example.	(2)
As above but the answer is brief and lacking in detail.	(1.8)
An attempt has been made but there are some mistakes	(1)
An attempt has been made but there are many mistakes	(0.5)
No attempt has been made	(0)

Part 3B. The Questions are answered well. Score 1 point for a good answer, 0.5 for a satisfactory answer and allow 0.5 extra credit for an exemplary answer (up to the given maximum).
Marks awarded in 0.5 increments up to 4.5 maximum

Overall code quality throughout	
Code quality excellent	(1.5)
Code quality good	(1)
Code quality okay	(0.5)
Code quality not good enough	(0)