COMP26120 Lab Exercise 10: 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.**

This is an examinable lab. This means the Summer Examination will ask a question on material closely related to this lab. In the exam, you must answer at least one of the two examinable lab questions. The other one will be on Testing the Small World Hypothesis (Lab 13).

For this assignment, **you MAY use code taken from other sources**. We are providing most of the code you need, so in the end it may be difficult to use code from other sources and adapt it to fit in. However, if you choose to do that you will NOT LOSE MARKS, provided you fully credit the sources used in your code, and check that it works correctly.

Copyright Notice

This lab makes use of material taken from http://www.es.ele.tue.nl/education/5MC10/.
In particular, the dynamic programming pseudocode we advise you to follow is available at http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf as well as locally.

Aims

To appreciate what discrete optimization problems are, and that they can be difficult to solve. To learn some of the algorithmic concepts used to solve them exactly (optimally) or approximately.

Learning Outcomes

On successful completion of this exercise, a student will:

1. Understand the 0/1 Knapsack Problem and Fractional Knapsack Problem. Appreciate how these have practical significance in logistics and other domains.
2. Have implemented a number of exact techniques for solving the 0/1 Knapsack Problem.
3. Have implemented one inexact technique - or heuristic - for finding good but not necessarily optimal solutions to the 0/1 Knapsack Problem.
4. Have compared the time complexity and running times of these techniques.
5. Have recommended an algorithm to a fictitious airline cargo company for the large knapsack problem instances that the company faces.

Summary

The task is to write a series of programs for solving the 0/1 Knapsack Problem and related problems. The programs will be compared for their running time and solution accuracy. You will then recommend one of them.

1. In week 1, you will look at the *full enumeration* procedure for solving the 0/1 Knapsack Problem, and implement a much more efficient *Dynamic Programming* method. The dynamic programming is the more significant task.
2. In week 2, you will write a *Branch-and-Bound* program for solving 0/1 Knapsack.
3. In week 3, you will make a simple greedy method based on the fractional knapsack method from week 2. You will then compare all the algorithms' complexities. You will try some large cargo packing problems, and recommend an algorithm. The significant work here is the experiments trying out the different algorithms.

Deadline

You should note that you will find it very difficult to complete this exercise if you do not start it in the FIRST week of the Lab and work consistently.

The standard deadline for this exercise is the end of the third lab session. An extended deadline is automatic for those who attend the final lab, and will be the beginning of the next lab session.

You cannot get any further extension for this lab (except for e.g. medical problems)!

Description

The 0/1 Knapsack Problem and Logistics

Transportation companies such as TNT and Royal Mail face daily problems in *logistics*. Consider the following simple logistics problem, which you will solve:

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 delivered, and the amount they are prepared to pay. The airline is *constrained* by the total amount of weight the plane is allowed to carry. The company must choose a subset of the packages (bids) to carry in order to make the maximum possible profit, given the weight limit that they must respect.

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) so that the total value carried is maximized, and the total weight carried is less than or equal to a given carrying capacity, C.

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 the coursebook, p. 498. You do not have to understand Theorem 17.15, as complexity theory is not part of this lab. (However, you may want to find out about this.)

The Fractional Knapsack Problem

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 maximized, 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. For each way, it is easy to calculate the total weight carried and the total value carried. We can then choose the most valuable packing that is within the weight limit.

For example, consider the following knapsack problem *instance*:

```
3
1   5   4
2  12  10
3   8   5
11
```

Where: 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.

(We will always use that way of representing knapsack problem instances)

The full enumeration of possible packings would be as follows 0

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: 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). If the solution also has weight less than the capacity *C* of the knapsack, then it is a *feasible solution*. If it is the best possible feasible solution (in terms of value) then it is an *optimal solution*. If it is only a high value solution, but not optimal, then it is an *approximate solution*. We will investigate some efficient ways of finding optimal solutions and approximate solutions.

Task 1a: Full Enumeration

Take a look at the files,

`enum.c`

and knapsack-util.c, as well as the makefile. Enum.c should be straightforward; it 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. (You should not 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. **(i) Adapt the code so that it does that.**

It would also be useful to display how much of the enumeration has been done - like a progress bar. **(ii) 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

Now write a program that solves the 0/1 Knapsack Problem by Dynamic Programming. We strongly recommend you follow the algorithm on page 17 of this handout*. (You may need to read the whole handout in order to understand it fully, you are advised do that afterwards.) Use the files knapsack-util.c and dp.c we provide for you in /opt/info/courses/COMP26120/problems/ex10. Further instructions are in dp.c. Test your code on the given instance file, easy.20.1.txt. You should get the same answer as with the enum program.

*From: http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

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

Introducing Branch and Bound

See pages 521-524 of the course textbook.

A good way to often get quite good solutions to the knapsack problem is to sort the items in descending order of value-to-weight ratio, and add items in that order until the knapsack is full.

Why will that often work well?

To do even better, we might have to take out some of the items we have put in and replace them by some other ones. In other words, we may need to *back-track*. This idea of back-tracking in an organized fashion is half the idea behind branch-and-bound.

The other half is that in order to prevent us backtracking through every possible solution, we can "prune" off parts of the search we must do. 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.

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 value of the best of all possible ways of replacing the *s by 0s and 1s. The estimate will not necessarily be accurate, it will be an *overestimate*. We call this overestimate 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 - but we are only interested in making an estimate. Why does this method guarantee not to underestimate the best possible value?

In the /opt/info/courses/COMP26120/problems/ex10/bnb.c code, we have already provided a 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.

Task 2b: Branch-and-Bound

Algorithm description (see book for more details).

In the branch-and-bound approach to solve the 0/1 Knapsack we first sort the items by decreasing value-to-weight ratio. We then begin by computing the upper bound of the solution ****... We then consider the two solutions 0***... and 1***... We compute the current values of each of them (i.e. the total value of all the 1s in each string) and their upper bound values, and check they are feasible. If they are feasible we place them on a priority queue. 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. Infeasible solutions are never placed on the queue. Feasible solutions are. 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 the 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.

Note: if the result you get does not look correct, it could be that when you set a bit in `solution[.]`, you used `solution[temp_index[i]]` when you should have used `solution[i]`. The function `check_evaluate_and_print_solution()` assumes that you will be referring to the items in their sorted order.

Task 3b: Testing and Comparing Methods

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.

Q1. 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.) (1 mark)

Q2. 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? (2 marks)

Q3. 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. (1 mark)

Q4. 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? (2 marks)

Marking Process

You should have in the directory COMP26120/ex10 the following SIX files: enum.c, dp.c, knapsack-util.c, bnb.c, greedy.c and makefile. Both labprint and submit will look for these six files.

The marks are awarded as follows:

```
Task 1:
4 marks: adapting the enumeration algorithm and its correct
operation on the example instance.
8 marks: implementation of the dynamic programming algorithm and
its correct operation on the example instance.
Task 2:
8 marks: implementation of the branch-and-bound algorithm and
operation on the example instance.
Task 3:
4 marks: implementation of the greedy algorithm and its correct
operation on the example instance.
6 marks: Answers to the questions about the testing of the
algorithms on the large instances.
Total 30
```