All exercises for dat119 (dat119-0)

`Revision: 1.2`

Exercises marked with **P** (for programming) ask you to write and/or run computer programs. The other exercises can be solved with pen and paper only. Some of the Java programs printed in the exercises have been typographically enhanced to increase readability. The symbol $\equiv$ has been used to denote Java's equality operator `==`.

Exercises marked with † have appeared in previous exams (minor details may have changed, and several bugs are removed). Square brackets surround the number of points that were given for different parts of the exercise, summing to 10. These values do not necessarily reflect the difficulty of the question.

1. An *ad hoc* algorithm for shortest (simple) path might start at the source, and at each vertex choose the shortest outgoing edge *from which the sink can be reached without touching the already constructed path.*

a) Construct an instance where the ad hoc algorithm above finds the shortest path. Construct an instance where the same algorithm happens to find the *longest path.*

b) What happens if we delete the part in italics from the above algorithm? Is the new algorithm guaranteed to find a solution? If yes, prove this, if not, construct an instance where it doesn't.

2. Exhibit an optimisation problem for which F is infinite.

3. Consider the following definition:

> A *maximisation* problem is a pair (F, c), where F is any set, the domain of feasible points; c is the cost function, a mapping
>
> $$c : F \to R.$$
>
> The problem is to find an $f \in F$ for which
>
> $$c(f) \geq c(y) \quad \text{for all } y \in F.$$

Formulate such a maximisation problem (F, c) as a *minimisation* problem according to Definition 1.1 in [PS]. What is the value of the optimal solution?

4. Formulate the following as an optimisation problem instance, giving the domain of feasible functions F and the cost function c:

- Find the longest path between two nodes in a graph with edge weights representing distance.

5. We will cast *sorting* as an optimisation problem. Recall that the input to the sorting problem consists of n integers $k_1, \ldots, k_n$, for simplicity we assume that they are all distinct. The problem is to find a reordering of the indices $s_1, \ldots, s_n$ such that $k_{s_1} > k_{s_2} > \cdots > k_{s_n}$ (note that we want to sort into decreasing order, this makes things slightly simpler).

a) Formulate the sorting problem as an optimisation problem, i.e., define the set F of feasible solutions and a cost function c.

The *2-change* neighbourhood N_2 of a sequence $s = \langle s_1, \ldots, s_n \rangle$ is defined as the set of sequences that differ from the original sequence on only 2 adjacent points. More precisely, if $t \in N_2(s)$ then there is a position i such that $s_i = t_{i+1}$ and $s_{i+1} = t_i$; on all other positions, s and t agree.

b) As an answer to the first question, Professor X— proposes the cost function given by

$$c(s) = \big|\{\, i \mid k_{s_i} < k_{s_{i+1}} \,\}\big|$$

Show that for this cost function, N_2 is not exact.

c) Propose a different cost function and show that N_2 is exact for it.

Thus the *2-opt* algorithm in Example 1.8 of [PS] finds an optimal solution (read 'sequence' for 'tour'). In other words, it is a sorting algorithm.

d) What is the running time of this sorting algorithm? And by what name is it commonly known?

Your answers should be short and completely formal.

6. P Write a class for minimum spanning forests. Recall that the greedy algorithm for this problem behaves as follows:

```
for all edges $e$ { insert $e$ into a priority queue }
while the queue is not empty
      { $e$:= shortest edge in the queue
        if $e$ does not induce a cycle in $F$
           insert $e$ into $F$.
      }
```

Your solution should extend the skeleton in Fig. 1. Run the program with

```
java MSF miles 128
```

the resulting forest should have size 16598 (the size is written in the first line of output).

7.P Implement the *cheapest insertion* heuristic for TSP on a complete graph. The heuristic extends a path node by node until the path is a Hamiltonian cycle. At each stage, the next node to be inserted is the one closest to the path's head.

You must implement a method

public static Path cheapestInsertion(**Graph** G, **Vertex** w)

that constructs such a path, starting from vertex w.

Report the shortest path you can find in Miles(128), iterating over all start vertices.

8.P Write a method that finds a *1-tree* lower bound on the TSP. Given a complete graph $G = (V, E)$ a 1-tree based on $v \in V$ is a spanning tree on the node set $V \setminus \{v\}$ together with two edges incident on v. You must implement the static method

```
import lu.cs.co.graph.*;
import lu.cs.co.util.PriorityQueue;
class MSF extends Forest
 {
   MSF(Graph G)
    { super(G);
      PriorityQueue Q= new PriorityQueue(G.m);
      // Your code goes here!
    }
   public static void main(String[] args)
    { Graph G= Graph.parseGraph(args);
      MSF F= new MSF(G);
      System.out.println(F);
      new Viewer(F);
    }
 }
```

Figure 1: Code skeleton for ex. 6. The main method reads in a graph G from the command line, calculates a minimum spanning forest F, writes it to standard output, and starts a Viewer (from lu.cs.co.graph) to show F on screen.

public static int oneTreeBound(**Graph** G, **Vertex** v)

that returns the length of the cheapest 1-tree based on v – you may assume that v's degree is at least 2. Also, implement the method

public static int oneTreeBound(**Graph** G)

finds the best 1-tree bound by iterating over all v. (See [PS, Example 18.4] for background and references on 1-trees).

9.P Extend the local search algorithm from Ex. 22 to start with a cheapest insertion tour from Ex. 7.

10.P Perform a well-documented, small scale experiment on the upper and lower bound heuristics constructed in Ex. 8 and 9. You should at least run them on a number of instances of different sizes (graphs with 10, 20, 30, . . . , nodes) and present the resulting bounds as a table or curve.

11.P Write an algorithm to find a TSP tour in the graph constructed by Miles() in lu.cs.co.graph. Your answer must consist of

1. the length of your tour
2. a string describing this tour, this string must pass the certificate checking algorithm you wrote for Ex. 34.
3. your answer to Ex. 34.
4. a brief description of how you found the tour.

Otherwise there are no rules—you may use any programming language and any algorithm you want, or solve the problem by hand.

12. P Write a class TSP with a static method

public static Path byTotalEnumeration(**Graph** G)

that finds a travelling salesman tour in G by trying all possible Hamiltonian cycles in G.

13.P Extend the class TSP from Ex. 12 with a static method

public static Path byBranchAndBound(**Graph** G)

that finds a travelling salesman tour in G by branch-and-bound (see [PS, Figure 18-5] for a code skeleton). How much larger instances can you solve than with total enumeration?

Comments:

1. for a good initial solution and upper bound take the tour constructed in Ex. 7.

2. for a good lower bound modify Ex. 8 to calculate the size of the cheapest 1-tree including a given path (see [PS, Example 18.4]).

14. A manufacturer uses resources of material (m) and labour (l) to make up to four possible items (a to d). The requirements for these, and the resulting profits are given by the following table:

item	resources needed	profit
a	$4m + 2l$	£5
b	$m + 5l$	£8
c	$2m + l$	£3
d	$2m + 3l$	£4

There are available up to 30 units of material and 50 units of labour per day. Assuming that theses resources are fully used and neglecting integrality constraints,

a) express the problem of finding a manufacturing schedule for maximum profit as an LP problem in standard form,

b) show that the policy of manufacturing only the two highest profit items yields a bfs which is not optimal

c) evaluate the schedule in which equal amounts of each item are manufactured

d) find the optimal schedule

Source: [1].

15. Consider the linear program

$$\begin{array}{rl} \text{minimise} & -4x_1 - 2x_2 - 2x_3 \\ \text{subject to} & 3x_1 + x_2 + x_3 = 12 \\ & x_1 - x_2 + x_3 = -8 \\ & x \geq 0. \end{array}$$

Verify that the choice $B(1) = 1, B(2) = 2$ gives a bfs, and hence solve the problem. *Source: [1].*

16. Consider the LP problem from the last exercise and replace the cost function by

$$-4x_1 - 2x_3.$$

Show that the basic feasible solutions derived from the basis $B(1) = 1, B(2) = 2$ and from $B(1) = 2, B(2) = 3$ are both optimal, and that any convex combination of these solutions is also a feasible solution. *Source: [1].*

17. Sketch the set of feasible solutions of the following inequalities:

$$\begin{aligned} x_1 + 2x_2 &\leq 4 \\ -x_1 + x_2 &\leq 1 \\ x_1 + x_2 &\leq 3 \\ x &\geq 0. \end{aligned}$$

At which points of this set does the function $x_1 - 2x_2$ take (a) its maximum and (b) its minimum value? *Source: [1].*

18. Consider the problem

$$\begin{array}{ll} \text{maximise} & x_2 \\ \text{subject to} & 2x_1 + 3x_2 \leq 9 \\ & |x_1 - 2| \leq 1 \\ & x \geq 0. \end{array}$$

a) Solve the problem graphically.

b) Formulate the problem as an LP problem in standard form.

Source: [1].

19. Use the tableau form of the simplex method to solve the LP problem

$$\begin{array}{ll} \text{minimise} & 5x_1 - 8x_2 - 3x_3 \\ \text{subject to} & 2x_1 + 5x_2 - x_3 \leq 1 \\ & -3x_1 - 8x_2 + 2x_3 \leq 4 \\ & -2x_1 - 12x_2 + 3x_3 \leq 9 \\ & x \geq 0 \end{array}$$

after reducing it to standard form. *Source: [1].*

20. † Let μ be a rational number and consider the following linear programme:

$$\begin{aligned} \min \mu x_1 - x_2 & \\ x_1 + 2x_2 &\leq 4, \\ 6x_1 + 2x_2 &\leq 9, \\ x_1, \quad x_2 &\geq 0. \end{aligned}$$

a) [1] Write the above linear programme in *standard form* by introducing slack variables.

b) [2] Write the programme as a *tableau* and use the tableau to find the basic feasible solution corresponding to the basis given by columns 1 and 2. (*Hint:* the correct answer is $(1, \frac{3}{2})$.)

c) [4] Assume $\mu = -1$. Use the simplex method to decide if the basic feasible solution from Question b is optimal.

d) [3] Prove that for all values of μ in the range $-3 < \mu < -\frac{1}{2}$ the optimum for our linear programme is achieved at the same point.

Source: [1].

21. † Consider the following linear programme:

$$\begin{array}{rrrrrrrrl}
\max & x_1 & + & 2x_2 & + & 3x_3 & + & 4x_4 & \\
 & x_1 & + & x_2 & + & x_3 & + & x_4 & = 1, \\
 & x_1 & & & + & 2x_3 & - & x_4 & = \frac{1}{2}, \\
 & x_1 & , & x_2 & , & x_3 & , & x_4 & \geq 0.
\end{array}$$

a) [1] Formulate the above programme as an equivalent *minimisation* problem.

b) [1] Write the minimisation problem on *tableau form.*

There are 6 basic solutions to the above problem, corresponding to the 6 choices of basis $\{A_1, A_2\}$, $\{A_1, A_3\}$, $\{A_1, A_4\}$, $\{A_2, A_3\}$, $\{A_2, A_4\}$, and $\{A_3, A_4\}$, where as usual A_i denotes the programme's ith column. Some friendly person has already constructed the tableaux corresponding to 5 of these bases:

basis $\{A_1, A_2\}$:

$\frac{3}{2}$	0	0	-3	-1
$\frac{1}{2}$	0	1	-1	2
$\frac{1}{2}$	1	0	2	-1

basis $\{A_1, A_3\}$:

0	0	-3	0	7
$-\frac{1}{2}$	0	-1	1	-2
$\frac{3}{2}$	1	2	0	3

basis $\{A_1, A_4\}$:

$\frac{7}{4}$	0	$\frac{1}{2}$	$-\frac{7}{2}$	0
$\frac{1}{4}$	0	$\frac{1}{2}$	$-\frac{1}{2}$	1
$\frac{3}{4}$	1	$\frac{1}{2}$	$\frac{3}{2}$	0

basis $\{A_2, A_3\}$:

$\frac{9}{4}$	$\frac{3}{2}$	0	0	$-\frac{5}{2}$
$\frac{3}{4}$	$\frac{1}{2}$	1	0	$\frac{3}{2}$
$\frac{1}{4}$	$\frac{1}{2}$	0	1	$-\frac{1}{2}$

basis $\{A_2, A_4\}$:

1	1	0	5	0
$-\frac{3}{2}$	2	1	3	0
$-\frac{1}{2}$	-1	0	-2	1

c) [2] Construct the tableau corresponding to the remaining basis $\{A_3, A_4\}$.

d) [1] List all basic solutions to the problem. Which of these are feasible?

e) [3] From basis $\{A_1, A_2\}$, on which elements can the simplex algorithm pivot? What are the corresponding new bases after a single pivot?

The basic feasible solutions define a digraph as follows: The vertices are the basic feasible solutions. There is an arc (i.e., a directed edge) from a bfs x to another bfs x' if the simplex algorithm can pivot from x to x'. (This is the same graph as the neighbourhood graph described in [PS. p. 62].)

f) [2] Draw this graph for our example programme. What is the worst case number of pivot operations the simplex algorithm can execute for our programme before it finds the optimum?

```
$Q$:= $P$:= some Hamiltonian cycle
do
  for $1 \leq i \neq j \leq n$ do
      exchange the $i$th and $j$th vertex in $P$
      if  $P$ was improved then $Q$:= $P$.clone()
      else  change the vertices back
while there was an improvement
return $Q$
```

Figure 2: A local search algorithm for TSP

22.P Write a 2-OPT algorithm for the Traveling Salesman problem similar to the algorithm sketched in Example 1.8 of [PS].

You should use the neighbourhood defined as follows. The set of feasible solutions F is the set of Hamiltonian cycles in the graph (recall that a simple path is a Hamiltonian cycle it is closed and contains all vertices exactly once). If P is in F then the neighbours of P are given by $N_2(P) = \{P' : P' \in F$ and P' is obtained by exchanging two vertices on the cycle$\}$. The **Path** class of lu.cs.co.graph contains an *exchange* method that does exactly that.

Test your algorithm on the graph constructed by 'Miles(i)'. What is the shortest TSP you can find?

Comments:

1. You have to construct an initial element in F (that is, a Hamiltonian cycle) to begin with, the constructor

   Path(**Graph** G, **int**[], **boolean**)

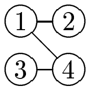
   in lu.cs.co.graph probably is the quickest way to do this.

2. There is no need to follow the pseudocode in Example 1.8. Especially, you may not want to write an *improve* procedure as suggested there. Figure 2 contains is a more efficient alternative and is easier to code.

3. While the algorithm is running, you can watch what happens. Construct a new **Viewer** based on Q. Whenever Q is improved you update the Viewer by calling its *setSubgraph* method with the new Q.

23. † The *minimum graph bisection* problem (MGB) is given a graph $G = (V, E)$ to find a partition of V into two equal sized sets A, B with $A \cup B = V$, $A \cap B = \emptyset$, and $|A| = |B|$ such that the number of edges between A and B (so-called *crossing edges*) is minimised.

In this exercise, we will write a partition as a list of length $|V|$ of letters 'A' and 'B'. The list [A,B,A,B] means that the first and third vertex of V are in A, while the second and fourth vertex are in B. In the example graph



,

the number of crossing edges in the partition [A,B,A,B] is 3.

a) [2] List all feasible solutions to MGB for the above graph and find the optimal solution.

The *2-exchange neighbourhood* N_2 for MGB is given as follows: The neighbours of a partition (A, B) are constructed by moving one node from A to B and moving one node from B to A. More formally, (A', B') is a neighbour of (A, B) if and only if there exist $a \in A$ and $b \in B$ such that

$$A' = A \cup \{b\} \setminus \{a\}, B' = B \cup \{a\} \setminus \{b\}.$$

b) [4] Write a local search algorithm for MGB that finds a local optimum with respect to N_2. The algorithm takes as input a graph and returns an array of characters 'A' and 'B' describing a partition as defined above. (Your algorithm does not have to be efficient.)

c) [4] Prove that N_2 is not exact.

24. † The *bin packing* problem is defined as follows: We are given n objects of *size* $s_1, \ldots, s_n$, the sizes satisfy $0 < s_i < 1$. The object is to pack these objects into as few *bins* as possible, each bin can hold a number of object whose total size is at most 1.

More formally, the problem is to minimise k such that $s_1, \ldots, s_n$ can be partitioned into k sets $B_1, \ldots, B_k$ such that

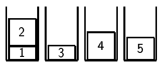
$$\sum_{s_i \in B_j} s_i \leq 1$$

for all $1 \leq j \leq k$. We will use size(B_j) to denote the left hand side of the above expression.

For example, consider 5 objects $\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}$ whose sizes are

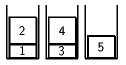
$$s_1 = \tfrac{1}{4}, s_2 = \tfrac{1}{2}, s_3 = \tfrac{1}{4}, s_4 = \tfrac{1}{2}, s_5 = \tfrac{2}{5}.$$

The following partition packs them into 4 bins:

$$B_1 = \{s_1, s_2\}, B_2 = \{s_3\}, B_3 = \{s_4\}, B_4 = \{s_5\},$$ or 

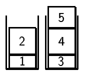
.

A better partition of the same elements, using only 3 bins, is given by

$$B_1 = \{s_1, s_2\}, B_2 = \{s_3, s_4\}, B_3 = \{s_5\},$$ or 

.

Finally, an infeasible partition is given by

$$B_1 = \{s_1, s_2\}, B_2 = \{s_3, s_4, s_5\},$$ or 

,

which is small but illegal, since size$(B_2) = \frac{1}{4} + \frac{1}{2} + \frac{2}{5} > 1$.

a) [1] Find an optimal bin packing for the above example.

b) [1] Describe (briefly) an algorithm that always finds an optimal solution and state its running time. (Your algorithm need not be efficient).

In this exercise we consider local search algorithms for bin packing. To fix some notation, we assume that our programming language supports operations on *sets* like

adding an element: $B_j := B_j \cup \{s_i\}$ adds element s_i to set B_j,

removing an element: $B_j := B_j \setminus \{s_i\}$ removes element s_i from set B_j (if it exists),

membership: $s_i \in B_j$ returns **true** if element s_i is in set B_j,

cardinality: $|B_j|$ returns the number of elements of B_j.

total size: $\text{size}(B_j)$ returns the total size of all elements in B_j,

iteration: **for** $s_i \in B_j$ **do** runs through all elements in B_j in some order.

Consider the following scheme for local search:

```
1 $[B_1,\dots,B_k]$:= $\langle$some initial partition$\rangle$;
2 while $improve([B_1,\dots,B_k]) \neq$ null do
3         $[B_1,\dots,B_k]$:= $improve([B_1,\dots,B_k])$;
4 $[A_1,\dots,A_r]$:= $[B_1,\dots,B_k]$ without the empty bins;
5 return $[A_1,\dots,A_r]$;
```

(We are not very precise about how to implement the 4th line, which can be done using standard list operations in time $O(k)$.)

The idea is that *improve* produces a partition with more empty bins than its input by moving a single element from one bucket to another. For example we can improve

$$[\{\tfrac{1}{4},\tfrac{1}{2}\},\{\tfrac{1}{4}\},\{\tfrac{1}{2}\},\{\tfrac{1}{2}\}] \quad \text{to} \quad [\{\tfrac{1}{4},\tfrac{1}{2}\},\{\tfrac{1}{4},\tfrac{1}{2}\},\emptyset,\{\tfrac{1}{2}\}]$$

by moving an element from the 3rd to the 2nd bucket, emptying the 3rd bucket. In local search terminology, the cost of a partition is given by the number of nonempty bins

$$c([B_1,\dots,B_k]) = |\{\, B_i \neq \emptyset \,\}|$$

and the neighbourhood N_1 of $[B_1,\dots,B_k]$ consists of all partitions of the form

$$[B_1,\dots,B_i \cup \{s\},\dots,B_j \setminus \{s\},\dots,B_k].$$

for some element s and bins B_i and B_j ($1 \le i \le k$ and $1 \le j \le k$).

c) [1] Complete line 1 of the above code by presenting a feasible initial partition.

d) [3] Write, in pseudocode, an algorithm for the function *improve* with respect to c and N_1 given above. State the running time.

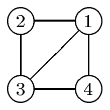
e) [2] Is N_1 exact? If yes, give a proof, if no, give a counterexample.

f) [2] Present a different cost function than c. Your cost function should lead to the optimum from partitions like $[\{\frac{3}{4}\},\{\frac{3}{4}\},\{\frac{3}{4}\},\{\frac{1}{4},\frac{1}{4},\frac{1}{4}\}]$, or generally

$$\left[\left\{\frac{n-1}{n}\right\},\dots,\left\{\frac{n-1}{n}\right\},\left\{\frac{1}{n},\dots,\frac{1}{n}\right\}\right].$$

25. †

A colouring of the vertices of a graph is *feasible* if no two adjacent nodes have the same colour. Consider for example a graph and two different colourings, χ and χ':



$\chi(1) =$	blue	$\chi'(1) =$	blue
$\chi(2) =$	red	$\chi'(2) =$	yellow
$\chi(3) =$	blue	$\chi'(3) =$	red
$\chi(4) =$	yellow	$\chi'(4) =$	yellow

The colouring χ is infeasible, since Vertex 1 and Vertex 3 are adjacent and both blue. The colouring χ' is feasible and uses 3 colours.

We will view a colouring as a function

$$\chi \colon V \to C_k$$

where C_k is a set of k colours. In the example, both colourings χ and χ' have $k = 3$ and $C_k = \{\text{blue}, \text{red}, \text{yellow}\}$.

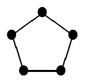
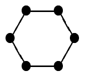
The *vertex colouring* problem is to assign a minimal number of colours to the vertices of a graph so that the resulting colouring is feasible:

Vertex colouring (evaluation)

INSTANCE: An undirected graph $G = (V, E)$, $|V| = n$.

QUESTION: Minimise k such that there exists set C_k of size k and a mapping $\chi \colon V \to C_k$ such that
$$\chi(i) \neq \chi(j) \quad \text{for all } [i, j] \in E.$$

a) [1] Solve the vertex colouring problem for the 5- and 6-cycle, i.e., the graphs

 and 

and describe a minimal feasible colouring for each.

b) [1] Describe a linear time algorithm that finds a feasible colouring for *any* graph using n colours.

c) [1] Describe a graph that does not admit a feasible colouring with fewer than n colours.

We now define a neighbourhood N of a colouring. Two colourings are neighbours if they assign the same colour to every vertex except for possibly one. Formally, $\chi' \in N(\chi)$ if

$$\bigl|\{v \in V \mid \chi(v) \neq \chi'(v)\}\bigr| \leq 1.$$

The *cost* of a colouring is the number of colours it uses,

$$c(\chi) = \Bigl|\bigcup_{v \in V} \chi(v)\Bigr|.$$

d) [3] Write a local search algorithm for vertex colouring with respect to N and c. (Your algorithm does not need to be efficient nor does it have to find the optimal solution.)

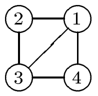
e) [4] Is N exact? If yes, give a proof, if no, give a counterexample.

26. † This exercise studies the combinatorial optimisation problem defined in Exercise 25, the *vertex colouring* problem.

The idea behind the *greedy* algorithm for vertex colouring is to colour the vertices one by one, using a new colour whenever necessary:

```
for ($i$= 1; $i \le n$; $i$= $i + 1$)
 { $\chi(v_i)$= ⟨colour with lowest number that is not used by any neighbour of $v_i$⟩ }
```

For example, if the colours are 'colour 1,' 'colour 2,' and so forth, then the graph



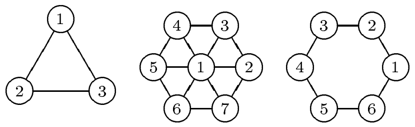
is coloured

$\chi(1) =$	colour 1
$\chi(2) =$	colour 2
$\chi(3) =$	colour 3
$\chi(4) =$	colour 2

,

which happens to be optimal.

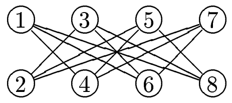
a) [1] For which of the following graph does the greedy algorithm find an optimal colouring:



b) [2] Consider the bipartite graph $G_n = (V, E)$ with $|V| = n$ even given by

$$E = \{\, [i,j] \mid i \neq j-1,\ i = 1,3,\ldots,n-1,\ j = 2,4,\ldots n \,\}$$

For example, G_8 looks like this:



What is the size of the smallest feasible colouring for G_n?

c) [2] What is the size of the colouring for G_n found by the greedy algorithm?

d) [5] Show that the greedy algorithm is *not* an $\frac{1}{5}n$-approximation algorithm for vertex colouring.

27. † This exercise studies the combinatorial optimisation problem defined in Exercise 24, the *bin packing* problem.

A simple algorithm for the bin packing problem is given below, known as the *first-fit heuristic.* The idea is to take each object in turn and place it into the first bin that can accommodate it, using a new bin whenever necessary.

```
$k$=1
for $i = 1$ to $n$
    { packed= false;
      for  $j = 1$ to $k$
```

```
            if size($B_j$)+$s_i \leq 1$ then { $B_j$= $B_j \cup \{s_i\}$; packed= true; }
        if not packed then { $k = k + 1$; $B_k = \{s_i\}$; }
    }
    return $B_1, \ldots, B_k$;
```

a) [1] Does the first-fit heuristic always find an optimal solution?

b) [2] What is the worst-case running time of first-fit?

We will now analyse the behaviour of first-fit as an approximation algorithm. We say that the jth bin is less than half full if size(B_j)$< \frac{1}{2}$.

c) [2] How many bins are left less than half full by first fit in the worst case? Give an example of this.

d) [2] Show that first-fit uses no more than $\lceil 2t \rceil$ bins, where

$$t = s_1 + \cdots + s_n$$

denotes the total size of the inputs. (*Hint*: Use Question c)

e) [3] Using Question d, show that the approximation ratio of first-fit is 1. (*Hint*: Can any algorithm use fewer than $\lceil t \rceil$ bins?)

Source: [2]

28. Recall that an $s-t$ Hamiltonian path in G is a path from s to t that includes every vertex of G exactly once. Let P be a path starting in s. Suggest an easily computed lower bound on the length of the shortest $s-t$ Hamiltonian path extending P. *Hint*: a Hamiltonian path is a spanning tree.

29.P Write a branch-and-bound algorithm for finding the shortest Hamiltonian path between two nodes in a complete graph.
Comments:

1. You should extend the class lu.cs.co.util.BranchAndBoundSkeleton, which implements the basic algorithm in [PS, Figure 18-5].

2. Study the example code in lu/cs/co/demo, which finds a shortest path as in [PS, Example 18.2]. You have to change the methods *isCompleteSolution* and *bound*.

3. For a lower bound on the length of the shortest Hamiltonian path extending a given path see Ex. 28.

4. Don't expect miracles from this nave exercise—your algorithm is probably very, very inefficient. Run it on Miles(n) for moderate values of n.

How many spanning trees (as a function of the input size) are computed before your algorithm finds the first complete solution?

30. † Consider a class **Path** whose members P support the following methods:

clone() return a copy of P of class **Path**,

insert(e) extend P with the edge e,

head() return the last node on P,

len() return the total length of the edges in P,

card() return the number of nodes on P,

contains(u) return ‘true’ if and only if vertex u is on P.

The constructor **new Path**(u) yields a new path consisting only of the vertex u

We also consider a class **PriorityQueue** supporting:

insert(P) insert the path P,

deleteMin() return the shortest path in the queue and remove it from the queue.

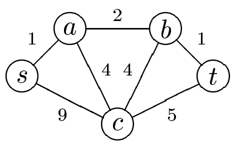
We let $N(u)$ denote the nodes incident to u in E, i.e.,

$$N(u) = \{\, v \in V \mid [u,v] \in E \,\}.$$

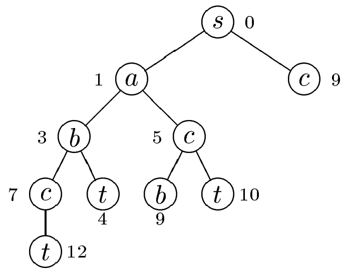
The algorithm below finds a Hamiltonian path from vertex s to vertex t in a graph with n vertices.

```
PriorityQueue $Q$= new PriorityQueue();
$Q$.insert(new Path($s$));
while ($Q \neq \emptyset$) { Path $P$= $Q$.deleteMin();
                    Vertex $u$= $P$.head();
                    if ($u = t \wedge P$.card() $= n$)
                      { print($P$);
                        stop;
                      }
                    elseif ($u \neq t \wedge P$.card() $\neq n$)
                        for ($v \in N(u)$)
                            if (!$P$.contains($v$))
                              $Q$.insert($P$.clone().insert([$u, v$]));
                  }
```

Consider the following example graph:



The order in which nodes are visited by the above algorithm is given by the following search tree:



Next to each node, we state the total length of the path corresponding to that node.

a) [2] Modify the above algorithm such that it finds the *shortest* Hamiltonian path. (Your algorithm does not have to be efficient.)

b) [1] Draw the search tree corresponding to your algorithm.

We now restrict our attention to graphs without negative edge lengths (like the example graph above). In that case, the total length of an intermediate solution can never decrease by inserting more edges. Thus an intermediate solution like $\textcircled{s}-\textcircled{c}-\textcircled{a}$ (of length 13) need not be extended, if we already have found a shorter, complete solution.

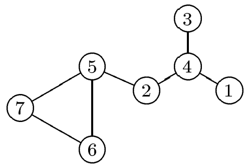
c) [5] Modify the above algorithm so that it uses branch-and-bound and present the resulting search tree for the example graph.

d) [2] Show that the nonnegativity assumption is necessary by presenting a graph with negative weights where your algorithm fails.

31. A subset of vertices $C \subseteq V$ of a graph $G = (V, E)$ is a *clique* if it is totally connected, i.e., $[u, v] \in E$ for all $u, v \in C$. Consider the following problem:

Maximum clique (evaluation)
INSTANCE: An undirected graph $G = (V, E)$,
QUESTION: Find the size of the largest subset of nodes $C \subseteq V$ such that $[u, v] \in E$ for all $u, v \in C$.

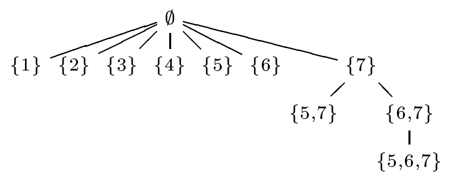
In this exercise we will refer the following example graph:



The following algorithm for the clique problem simply constructs all subsets of V, starting with the one-element sets $\{v_1\}, \{v_2\}, \ldots, \{v_n\}$ (which are cliques of size 1) and adding vertices if possible. We use a stack S to keep track of all the subsets:

```
$S = \emptyset$;
$max = 0$;
for ($v \in V$) { $S.push(\{v\})$;  }
while ($S \neq \emptyset$) { $U = S.pop$;
                    if ($U.size > max$) { $max = U.size$;  }
                    for ($v \in V - U$)
                     { if $\langle v$ is connected to all vertices in $U\rangle$
                        { $S.push(U \cup \{v\})$;  }
                     }
                  }
return $max$
```

When we run this algorithm on the example graph, the search tree describing the evolution of S looks like this after a while:



a) [1] What is the size of the largest clique in the example graph?

b) [2] Finish the search tree describing the evolution of S.

c) [2] Let $\delta(v)$ denote the *degree* (number of neighbours) of a vertex. Show that if vertex v belongs to a clique C then $|C| \leq \delta(v) + 1$.

d) [4] Modify the above algorithm into a *branch-and-bound* algorithm using the lower bound from the last question.

e) [1] Draw the entire search tree corresponding to your branch-and-bound algorithm.

32. Draw the entire tree of partial solutions enumerated by a total enumeration algorithm (say, the one in lu.cs.co.demo.TotalEnumeration) on the instance in [PS, Fig. 18–6(a)]. In general, give an upper bound on the size of such a tree for a graph with n nodes.

33. Let P be a path in a directed graph $G = (V, E)$ (recall that a path is a sequence of vertices $(P_1, \ldots, P_k)$ such that $(P_i, P_{i+1}) \in E$ and no vertex appears twice). We write $e \in P$ if edge e belongs to P in the sense that for $e = (P_i, P_{i+1})$ for some i. Let $\operatorname{len}(P)$ denote the length of path P, i.e.,

$$\operatorname{len}(P) = \sum_{e \in P} \operatorname{len}(e),$$

where $\operatorname{len}(e)$ denotes the length of edge e.

We will consider paths from $s \in V$ to $t \in V$. Let $P = (P_1, \ldots, P_k)$ be a path starting in s but not necessarily ending in t. Let S denote the set of paths P' that extend P to reach t, i.e., if $P' = (P'_1, \ldots, P'_r)$ we have $P_i = P'_i$ for $i = 1, \ldots, k$ and $P'_r = t$.

We now derive an upper bound on the length of every path extending P:

$$\text{for all } P' \in S\colon\ \operatorname{len}(P') \leq \operatorname{len}(P) + \sum_{e \in E'} \operatorname{len}(e),$$

where

$$E' = \{\, (u, v) \in E \mid u \neq t, u \neq P_i, v \neq P_i, (i = 1, \ldots, k-1) \,\}.$$

Give a *completely formal* proof of this fact.

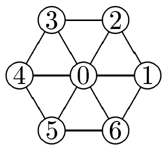
34.P Write a *certificate checking algorithm* for TSP. Your algorithm must consist of a static method

public static boolean checkTSP(**Graph** G, **String** S, **int** l)

that checks if

1. the string S describes a Hamiltonian cycle in G
2. its total length is no larger than l.

You have to decide how your algorithm expects the cycle to be encoded. For example if

$G =$ 

then a Hamiltonian cycle could be encoded by

$S =$ `"v4 v3 v2 v0 v1 v6 v5"`

or by

$S =$ `"[4,3], [3,2], [2,0], [0,1], [1,6], [6,5], [5,4]"`

Your answer must include

1. the implementation of the above method
2. a detailed description of the encoding expected by your algorithm
3. documentation for a number of tests
4. analysis of the running time of your algorithm.

The aim of this exercise is to identify and implement all the necessary checks yourself. For that reason you can *not* use the lu.cs.co.graph.Path class or the methods therein.

35.P Write certificate checking algorithms for one or both of the following problems:

Clique Given a graph G and an integer l, does G have an l-clique [see PS, Example 15.5]? Your algorithm must contain a static method

public static boolean checkClique(**Graph** G, **String** S, **int** l)

such that there is an input S for which the method returns true if and only if there is a k-clique in G.

Evenclique Given a graph G, is the largest clique of G of even size? Your algorithm must contain a static method

public static boolean evenClique(**Graph** G, **String** S)

such that there is an input S for which the method returns true if and only if the largest clique in G has even size.

36. None, some, or all of the following problems are in NP. Decide which.

1. Given a complete graph G with weighted edges and an integer B. Does G have a path from node 1 to node 2 of length less than B?
2. Given a complete graph G with weighted edges and an integer B. Does G have a spanning tree total length less than B?
3. Given a complete graph G with weighted edges and an integer B. Does G have a Hamiltonian path from node 1 to node 2 of length less than B?
4. Given a complete graph G with weighted edges and an integer B. Does G have a Hamiltonian cycle (a TSP tour) of total length less than B?
5. Given a complete graph G with weighted edges and an integer B. Does G have two different Hamiltonian cycles of total length less than B?
6. Given a complete graph G with weighted edges and an integer B. Is the number of Hamiltonian cycles of total length less than B in G even?
7. Given a complete graph G with weighted edges and an integer B. Do all Hamiltonian cycles in G have total length less than B?
8. Given two complete graphs G_1, G_2 on the same set of nodes with weighted edges and an integer B. Do G_1 and G_2 have a common Hamiltonian cycle whose length is less than B in both graphs?
9. Given two complete graphs G_1, G_2 on the same set of nodes with weighted edges and an integer B. Do all Hamiltonian cycles whose length is less than B in G_1 also have length less than B in G_2?

37. Repeat the last exercise for P (the class of problems solvable in polynomial time) and EXP (the class of problems solvable in exponential time)

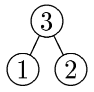
38. A *colouring with k colours* of a graph $G = (V, E)$ is a mapping:

$$\chi : V \to 1, 2, \ldots, k$$

such that $[u, v] \in E$ implies $\chi(u) \neq \chi(v)$.

The *graph colouring problem* is: given a graph, what is the smallest number of colours needed to colour it.

1. What is the smallest number of colours needed to colour the following graphs:



   Same question for a complete graph? A bipartite graph? A cycle?

2. Assume the graph G is encoded in the alphabet

$$\Sigma = \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}, \texttt{[}, \texttt{]}, \texttt{,}\},$$

   by listing the number of nodes followed by a list of edge pairs. For example, the above graph is represented by "`3,[1,3],[2,3]`". Suggest an encoding for a colouring χ.

3. Sketch an algorithm that solves the graph colouring problem (your algorithm may take exponential time).

4. Formulate the *recognition version* of the graph colouring problem.

5. Show that graph colouring is in NP by constructing a certificate checking algorithm for it.

39. † This exercise studies the combinatorial optimisation problem defined in Exercise 25, the *vertex colouring* problem.

a) [2] Formulate the *recognition* version of the vertex colouring problem.

b) [4] Show that the recognition version of vertex colouring is in NP by presenting a certificate checking algorithm for it. Be precise about what the inputs to the algorithm are, how the certificate is encoded, and what its length is. State the algorithm's running time.

Consider the 3-colouring problem:

Three-colouring
INSTANCE: An undirected graph $G = (V, E)$.

QUESTION: Does G admit a feasible colouring with only three colours?

c) [4] Show that the recognition version of vertex colouring is NP-complete. You may use the fact that three-colouring is known to be NP-complete. Be precise about what the inputs to your algorithm are.

40. Translate into Swedish the newspaper article presented in [PS] Exercise 15.19. Remove all errors and clarify all misunderstandings.

41. At the nearest beach, a computer scientist and a biologist are asked to light a fire using wet drift-wood. The available tools are (i) a bucket, (ii) a red herring and (iii) some matches. After a while, both succeed (the computer scientist finishes last, having examined all tools in order). In the next round, the task again is to light a fire, but this time using dry branches from the nearby forest. What happens?

42. Is there a winning strategy for the game of Noughts and Crosses (Three in a Row)? How would you prove such a claim?

43. Nine cards $\boxed{1}$, $\boxed{2}$, up to $\boxed{9}$ are placed face-up on the table. Alice and Bob take turns picking cards. The winner is the player who can form the sum 15 with exactly three of his or her cards. For example, if Alice holds $\boxed{1}$ $\boxed{5}$ $\boxed{6}$ and $\boxed{9}$ she wins because 1+5+9=15.
Is there a winning strategy for this game?

44. The *longest path* problem is: given a graph $G = (V, E)$ and a distance matrix D where d_{ij} denotes the length of edge $[i, j] \in E$, and two nodes $s, t \in V$, find the longest (simple) path from s to t in G. Formulate the recognition version of this problem and show that it is NP-complete. (Hint: use Corollary 1 on p. 370 in [PS].)

45. † Two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on the same set of nodes are *isomorphic* if there exists a permutation π on V such that

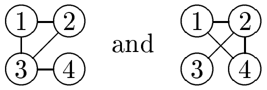
$$E_2 = \bigcup_{[u,v] \in E_1} \{ [\pi(u), \pi(v)] \}.$$

In other words G_1 is the same as G_2 up to a renumbering of the nodes.

In this exercise we will write a permutation as a list of $|V|$ numbers. For example, the list $[4, 1, 2, 3]$ stands for the permutation given by the following table:

u	1	2	3	4
$\pi(u)$	4	1	2	3

The two graphs



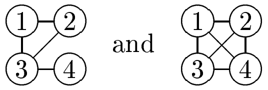
are isomorphic because of the permutation $[4, 1, 2, 3]$.

The *graph isomorphism* (GI) problem is, given two graphs G_1 and G_2 answer 'yes' if and only if G_1 is isomorphic to G_2.

a) [2] Draw two graphs with 4 vertices and 4 edges that are not isomorphic.

b) [4] Show that GI is in NP by writing a certificate checking algorithm for it. Be precise about what the inputs to the algorithm are, and state its running time.

The *subgraph isomorphism* problem is, given two graphs G_1 and G_2 answer 'yes' if and only if G_1 is isomorphic to some *subgraph* of G_2. For example, given



,

the answer is 'yes' because the left graph is isomorphic to a subgraph of the right (delete $[1, 3]$ and $[3, 4]$ from the right graph).

c) [4] Prove that *subgraph isomorphism* is NP-complete. *Hint:* Use the fact that the Hamiltonian path problem is NP-complete.

46. † This exercise studies the combinatorial optimisation problem defined in Exercise 24, the *bin packing* problem.

In this exercise we consider the *recognition version* of bin packing: Given n objects of size $s_1, \ldots, s_n$ and an integer k, do the objects fit into k bins?

a) [4] Show that the recognition version of bin packing is in NP by presenting a certificate checking algorithm for it. Be precise about what the inputs to the algorithm are, how the certificate is encoded, and what its length is. State the algorithm's running time.

The *partition* problem is defined as follows: given a set $X = \{x_1, \ldots, x_n\}$ of positive integers, is there a subset $S \subseteq X$ that adds up to exactly half the total sum? For example, if the set is

$$X = \{1, 3, 5, 6, 7, 10\}$$

then the answer is 'yes' because we can take $S = \{1, 3, 5, 7\}$, which sums to 16.

b) [1] What is the answer to the partition problem for $X = \{3, 4, 8, 11, 64, 78\}$?

Assume we have an algorithm for bin packing. To be precise, assume that we have an efficient implementation for the function

boolean binPacking(**rational**`[]` S, **int** k)

that takes an array of rational numbers S and an integer k and returns **true** if and only if S fits into k bins.

c) [2] Write an efficient algorithm for partition using binPacking. To be precise, you must implement the function

boolean partition(**int**`[]` X)

that takes as input an array X of integers and returns **true** if and only if X can be partitioned into two equal size parts. You may (and should) use binPacking as a subroutine.

d) [3] Show that bin packing is NP-complete. You may use the fact that partition is NP-complete.

47. Consider the local improvement algorithm of [PS, example 1.8].

a) Show that the algorithm *terminates* (i.e., halts within a finite number of steps) if $|F| < \infty$.

b) Exhibit an optimisation problem for which the running is infinite.

48. The *knapsack problem* is to pack some of n items into a knapsack such that their total weight does not exceed the knapsack's capacity, and the value of the items is maximised.

For example, consider a knapsack of capacity 10 and the items in Table 1. An example of a feasible solution (called a *packing* of the knapsack) contains the gold watch and the sweater. A better packing contains the gold watch and both books. The knapsack cannot hold the bicycle. We can formulate the problem as an integer linear program as follows:

Table 1

Item	weight	value
Gold watch	3	10
Red book	3	2
Blue book	3	3
Sweater	5	1
Compass	2	2
Bicycle	14	10

$$\begin{array}{ll} \text{maximise} & 10x_1 + 2x_2 + 3x_3 + x_4 + 2x_5 + 10x_6 \\ \text{such that} & 3x_1 + 3x_2 + 3x_3 + 5x_4 + 2x_5 + 14x_6 \leq 10 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \in \{0, 1\} \end{array}$$

More generally, the knapsack problem is given as follows:

Knapsack (optimisation)

INSTANCE: Integers $v_1, \dots v_n, w_1, \dots, w_n, K \geq 0$

QUESTION:

$$\begin{aligned} &\text{maximise} \sum_{i=1}^{n} v_i x_i \\ &\text{such that} \sum_{i=1}^{n} w_i x_i \leq K \\ &x_1, \dots, x_n \in \{0, 1\} \end{aligned}$$

In this exercise we will consider a simple instance with only two items:

	weight	value
Item 1	5	3
Item 2	4	4

The capacity of our knapsack is 6.
All the following questions relate to this instance.

a) [1] Write down the instance as an integer programme.

b) [1] List all feasible solutions and identify the optimum.

c) [1] Rewrite the problem as a minimisation problem.

d) [3] By replacing the integrality constraints with inequalities and introducing slack variables, write down the linear programming *relaxation* of the problem on standard form. Briefly explain each newly introduced equation and variable. *Hint*: The solution uses 5 variables.

e) [4] Use the simplex method to find the optimal solution to the LP relaxation. If you haven't solved Exercise c, you may instead optimise the following programme:

$$\begin{array}{lrcrcrcrcrcl} \text{minimise} & -4x_1 & - & 3x_2 & & & & & & & & \\ & 6x_1 & + & 7x_2 & + & x_3 & + & x_4 & + & x_5 & = & 3, \\ & x_1 & & & & & + & x_4 & & & = & 1, \\ & & & x_2 & & & & & + & x_5 & = & 1, \\ & x_1 & & x_2 & , & x_3 & , & x_4 & , & x_5 & \geq & 0. \end{array}$$

49. This exercise studies the same problem as Ex. 48, the knapsack problem.

A feasible solution to the knapsack problem will be represented by an array x of booleans such that $x[i] =$ **true** if an only if Item i is in the knapsack $(0 \leq i < n)$. For example, a function w to compute the weight of a solution x looks as follows:

```
int w(boolean [] x)
 { int s= 0;
   for (int i= 0; i < x.length; i= i + 1) { if (x[i])   s= s + w_i;   }
   return s;
 }
```

A function v to compute the value of a solution can be implemented similarly.

Define a solution's neighbourhood $N(x)$ by $y \in N(x)$ if and only if

$$\left|\{\, 1 \leq i \leq n \mid x[i] \neq y[i] \,\}\right| = 1,$$

i.e., the arrays differ in exactly one position.

a) [1] Compute $|N(x)|$, the neighbourhood's size.

b) [4] Is N exact? If yes, give a proof. If no, give a counterexample.

We now consider a local search algorithm with respect to N. Our strategy is the following:

- if possible, add the most expensive item not already in the knapsack,
- otherwise, remove the cheapest item from the knapsack.

A quick way to implement this rule is to observe that the difference in value δ will be $\delta = v_i$ for adding Item i and $\delta = -v_i$ for removing Item i. Our local search algorithm simply moves to the neighbour with largest δ. To be quite precise, our (not very useful) local search algorithm looks like this:

```
 1 **boolean** [ ]$optimise()$
 2  { **boolean** [ ]$x$= ⟨initial feasible solution⟩;
 3    **boolean** [ ]$x^\star$= $x.clone()$;
 4    **int** $counter$= 100;
 5    **do** { **int** $i^\star$= $-1$; **int** $\delta$= $-\infty$;
 6           **for** (**int** $i$= 0; $i < x.length$; $i$= $i+1$)
 7           {
 8             **if** ($x[i] \wedge -v_i > \delta$);                 Want to remove Item $i$
 9               { $\delta$= $-v_i$; $i^\star$= $i$; }
10             **if** ($\neg x[i] \wedge v_i > \delta \wedge w(x) + w_i \le K$)    Want to add Item $i$
11               { $\delta$= $v_i$; $i^\star$= $i$; }
12           }
13           **if** ($i^\star \neq -1$)   $x[i^\star]$= $1 - x[i^\star]$;         Move to best neighbour
14           **if** ($v(x) > v(x^\star)$)   $x^\star$= $x.clone()$;           New optimum found
15           $counter$= $counter - 1$;
16        }
17    **while** ($counter > 0$);
18    **return** $x^\star$;
19  }
```

c) [1] Write code to implement Line 2.

d) [1] Run the algorithm on the following instance: The knapsack has capacity 2, Item 1 has weight 2 and value 3, Items 2 and 3 both have weight 1 and value 2.

The idea behind taboo search is to prevent an algorithm to move from solution x to solution $y \in N(x)$ if this move is *taboo*. We will modify the above algorithm with this idea in mind, using the following rule:

- at item that has been removed cannot enter the knapsack in the next step.

e) [3] Rewrite the above algorithm to use the above rule. Run your algorithm on the instance from question d.

50. This exercise studies the same problem as Ex. 48, the knapsack problem.

A famous algorithm for knapsack uses the *dynamic programming* idea. Let a table T of dimension $n \times K$ be defined so that $T(w, i)$ is the largest value attainable by choosing items among the first i such that their total weight is at exactly w. For the instance in Table 1 of Exercise 1, we show part of T to the right. The entry $(2, 4)$ is 0 because there is no way to choose items among the first 4 to obtain weight 2. The entry $(5, 6)$ is 12 because among the 6 items we can choose the compass and the gold watch to obtain weight 5 and value 12.

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	0	0	0	0	2	2
3	10	10				
4	10					
5						12

To compute $T(w, i+1)$ in general we can either

- not include Item $(i+1)$, in which case $T(w, i+1) = T(w, i)$, or
- include Item $(i+1)$, in which case the value is that of the new item, v_{i+1}, plus $T(w - w_{i+1}, i)$, the optimal packing of the remaining space in the knapsack.

In summary,

$$T(w, i+1) = \max\{T(w, i),\, T(w - w_{i+1}, i) + v_{i+1}\}.$$

a) [3] Use the above rule to write an algorithm for Knapsack that runs in time $O(nK)$.

If K is large (e.g., $K = 2^n$), the running time of the above algorithm becomes unacceptable. The idea behind the following algorithm is to order the items with respect to the ratio v_i/w_i and pack them in that order until the knapsack is full. We give up as soon as we encounter an item that does not fit.

```
 1  **boolean** []*greedyKnapsack*(**int** $K$, **int** []$w$, **int** []$c$)
 2  { ⟨order $w$ and $v$ so that $\frac{v[i]}{w[i]} \geq \frac{v[i+1]}{w[i+1]}$   $(0 \leq i < n-1)$⟩
 3    **boolean** [] $x$= **new boolean** $x[w.length]$;
 4    **for** (**int** $i$= 0; $i < x.length$; $i$= $i+1$)   $x[i]$= **false**;
 5    **boolean** *full*= **false**;
 6    **int** $i$= 0;
 7    **do** { **if** $(w(x) + w[i] \leq K)$   $x[i]$= **true**;
 8             **else** *full*= **true**;
 9             $i$= $i+1$;
10         }
11    **while** ($\neg$*full*);
12    **return** $x$;
13  }
```

The function w used in Line 7 was defined in Exercise 2.

b) [1] Run greedyKnapsack on the Instance from Table 1 in Exercise 1. How many iterations are performed?

c) [2] Show that greedyKnapsack is a 1-approximation algorithm.

d) [1] Consider the following two-element instance: Item 1 has weight 1 and value K, Item 2 has weight K and value $K(K-1)$. Which solution does the algorithm find, and what is the optimum?

e) [3] Use the above question to show that greedyKnapsack is not an ϵ-approximation algorithm for any $\epsilon < 1$.

51. This exercise studies the same problem as Ex. 48, the knapsack problem.

Below are two suggestions for a recognition version of the knapsack problem. The instances are a knapsack problem and a bound B.

Knapsack (recognition) 1

INSTANCE: Integers $v_1, \ldots v_n, w_1, \ldots, w_n, K, B \geq 0$,

QUESTION: Do all $x_1, \ldots, x_n \in \{0,1\}$ such that

$$\sum_{i=1}^{n} w_i x_i \leq K$$

satisfy the bound

$$\sum_{i=1}^{n} v_i x_i \leq B$$

Knapsack (recognition) 2

INSTANCE: As above.

QUESTION: Does there exist $x_1, \ldots, x_n \{0,1\}$ such that

$$\sum_{i=1}^{n} w_i x_i \leq K$$

that satisfies the bound

$$\sum_{i=1}^{n} v_i x_i \geq B$$

a) [4] (At least) one of the above problems is in NP. Decide which and show that it is in NP by presenting a certificate checking algorithm for it. Be precise about what the inputs to the algorithm are, how the certificate is encoded, and what its length is. State the algorithm's running time.

The *partition* problem is defined as follows: given a set $S = \{s_1, \ldots, s_n\}$ of positive integers, is there a subset $R \subseteq S$ that adds up to exactly half the total sum? More formally,

Partition

INSTANCE: Set of integers $S = \{s_1, \ldots, s_n\}$.

QUESTION: Is there a subset $R \subseteq S$ such that

$$\sum_{s \in R} s = \tfrac{1}{2} \sum_{s \in S} s.$$

Assume that we have an algorithm for the recognition version of the Knapsack problem. To be precise, assume that we have an efficient implementation for the function

boolean *knapsack*(**int** K, **int** $[\,]w$, **int** $[\,]v$, **int** B)

that takes a capacity, and two arrays of weights and values, respectively, and returns **true** if and only if the items can be packed into the knapsack such that their total value is at least B.

b) [3] Write an efficient algorithm for partition using knapsack. To be precise, you must implement the function

boolean *partition*(**int** [ ]S)

that takes as input an array S of integers and returns **true** if and only if S can be partitioned into two equal size parts. You may (and should) use knapsack as a subroutine. *Hint*: Construct items whose value is the same as their weight.

c) [3] Show that the recognition version of Knapsack is NP-complete. You may use the fact that partition is NP-complete.

52. A basis $\mathcal{B}'$ a *neighbour* of basis $\mathcal{B}$ if it is obtained from $\mathcal{B}$ by a single pivoting step (see [PS, Thm. 2.7]). Define the neighbourhood $N(x)$ of a bfs x corresponding to $\mathcal{B}$ as the set of bfs x' that correspond to neighbours $\mathcal{B}'$ of $\mathcal{B}$. Show that N is not exact for linear programming.

Hint: Consider the LP given by $c = (0, 0, 1, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})$,

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix},$$

and the bfs corresponding to $\{A_4, A_5, A_6\}$.

References

[1] Fletcher: Practical methods of optimization, Wiley 1987. Most exercises have been slightly modified for notational consistency.

[2] Cormen, Leiserson, Rivest: Introduction to algorithms, MIT Press, 1989.