COMP2001 (AIM) Project 2022

SOLVING THE 0-1 KNAPSACK PROBLEM

1 Description of the problem

The 0-1 *knapsack problem* is a well-known and well-studied NP-hard combinatorial optimisation problem.

A prize-winning shopper, given a knapsack, enters the store finding n items; the j^{th} item is worth p_j GBPs, and weights w_j pounds. The shopper is allowed to take any load but to carry at most W pounds in the knapsack. So, the shopper wants to take as profitable a load as possible. Which items should the shopper take for the maximum profit?

More formally, given a set of n items, each with a weight $w_j > 0$ (positive value) and a profit $p_j > 0$ (positive value), where $j \in N = \{1, \dots, n\}$, and a *knapsack* with maximum weight capacity $W > 0$ (positive value), the problem is to select a subset of items to

$$\text{maximise} \quad f(x_1, \dots, x_n) = \sum_{j=1}^{n} p_j x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq W \tag{2}$$

where $x_j = \begin{cases} 1, & \text{if item } j \text{ is selected} \\ 0, & otherwise \end{cases}$

This is called the 0-1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once. The knapsack problem is an abstraction of many real-world problems from combinatorial auctions and capital budgeting to frequency allocation in cognitive radio networks, cutting across the fields of logistics, informatics, finance, engineering and more.

There are different greedy heuristic methods for solving the problem. However, they often do not provide the optimal solution to a given instance. You can find below 3 different constructive heuristics and examples of how they operate.

Greedy Heuristic#1

Steal the items with the largest profit.

Example: Given $n = 3$, $W = 30$, and items sorted with respect to the profit from max. to min. $\{p_1 = 10, p_2 = 9, p_3 = 9\}$ with the associated weights $\{w_1 = 25, w_2 = 10, w_3 = 10\}$, place an item one at a time in the given order into the knapsack starting from the first item to the last one as long as it can be accommodated, otherwise skip the item. So, once we place the first item with profit 10 and weight 25, we cannot place another item, so we have the solution <100>, that is $\{x_1 = 1, x_2 = 0, x_3 = 0\}$ which yields the profit of 10, although the optimal profit is 18 if the second and third items were selected (<011>).

Greedy Heuristic#2

Steal the items with the smallest weight.

Example: Given $n = 3$, $W = 30$, and the items sorted with respect to weight from min. to max. $\{w_1 = 5, w_2 = 10, w_3 = 20\}$ with the associated profits $\{p_1 = 50, p_2 = 60, p_3 = 140\}$, place an item one at a time in the given order into the knapsack starting from the first item to the last one as long as it can be accommodated, otherwise skip the item. So, we place the first item with profit 50 and weight 5, and we have remaining capacity. Then, we place the next item with profit 60 and weight 10, but we cannot place the last item as we would exceed the capacity of the knapsack. So, we halt with the solution <110>, that is $\{x_1 = 0, x_2 = 1, x_3 = 1\}$ which yields the profit of <110>, although the optimal profit is 200 if the second and third items were selected (<011>).

Greedy Heuristic#3

Steal the items with the largest profit per unit weight.

Example: Given $n = 3$ with the associated profits $\{p_1 = 50, p_2 = 60, p_3 = 140\}$ and weights $\{w_1 = 5, w_2 = 10, w_3 = 20\}$, $W = 30$, and the items sorted with respect to profit/weight (profit per unit weight) from min. to max. $\{p_1/w_1 = 10,\ p_3/w_3 = 7, p_2/w_2 = 6\}$, place an item one at a time in the given order into the knapsack starting from the first item to the last one as long as it can be accommodated, otherwise skip the item. So, we place the first item with profit 50 and weight 5, and we have remaining capacity. Then, we place the next (third) item with profit 140 and weight 20, but we cannot place the last (second) item as we would exceed the capacity of the knapsack. So, we halt with the solution <101>, that is $\{x_1 = 1, x_2 = 0, x_3 = 1\}$ which yields the profit of 190, although the optimal profit is 200 if the second and third items were selected (<011>).

2. Task

Design and implement a (meta/hyper-)heuristic-based optimisation approach from scratch, that is you should not make use of any APIs that are not included in the standard Java libraries, for solving the 0-1 Knapsack Problem using Java. You are fully free to choose and implement the solution approach and its algorithmic components as you see fit, however, depending on the difficulty level of this choice, the maximum achievable mark from the project is fixed using the following weights.

Optimisation Approach	**Difficulty Level**	**Weight (Max. Mark)**
A Learning Hyper-heuristic* or a Multimeme Memetic Algorithm embedding multiple operators for each type	**1**	1.00 (100)
Memetic Algorithm	**2**	0.70 (70)
Genetic Algorithm	**3**	0.60 (60)
Single point-based metaheuristics using hill climbing (e.g., Iterated Local Search using Simulated Annealing)	**4**	0.50 (50)

*This hyper-heuristic should not be using random choice for heuristic selection.

All projects will be assessed out of 100 and the overall final mark for a project will be computed by multiplying this grade with the associated weight considering which optimisation approach is implemented as illustrated in the table above.

Example: Assuming that a student implements and submits a Genetic Algorithm as the solution approach to the problem, then after grading based on the assessment criteria, that mark will be multiplied by 0.6 to compute the overall final mark for the project. So, if the submission receives 80, then 48 will be returned as the final mark for the project.

2.1. Initial test instances

The following low dimensional 0-1 knapsack problem benchmark instances with known optima are provided for testing purposes. Please check Moodle for download.

Instance ID	Optimum
test1_4_20	35
test2_10_269	295
test3_20_879	1025

2.2. Instance reader – Input File Format for the Problem Instances

You should implement an instance reader code to input a given problem instance. All problem instances will be provided in the same format.

File name: instanceName_n_W.txt

instanceName: name of the instance of 0-1 knapsack problem

n: number of items

W: knapsack capacity

Example

n W

p1 w1 // profit and weight for item#1

p2 w2 // profit and weight for item#2

: :

pj wj // profit and weight for item#j

: :

pn wn // profit and weight for item#n

Warning: Please do not rely on the name of the file for extracting the number of items and knapsack capacity for a given instance. You should be using the content of the file instead. The filename identifies a problem instance.

2.3. Output

Your code should output the following information and files when executed:

- Display the best solution as a binary string after its objective value (profit) separated by a newline via the standard output for each trial.

  E.g.: Assuming a problem instance with *n*=5, and running the algorithm for 2 trials, the output could be as follows

  Trial#1: // this indicates the ID of the trial – that is the result from the first trial

  23 // this is the objective value found after running the algorithm till termination

10010 // this is the binary solution indicating that first and fourth items are selected

Trial#2:

55

11010

- You should define a parameter (e.g., *numberOf*) indicating how many pairs of objective values you will be recording when you run your algorithm for multiple trials for output. If you are using a single-point-based search method, record the best and current objective values at each step and print it into a textual file for the first *numberOf* iterations for each trial.
  If you are using a population-based search method record the best and worst (as the current) objective value in the population at each generation and print it into a textual file for the first *numberOf* generations for each trial.

  E.g.: instanceName_n_W_trialID_output.txt file contains the following

  Best-1 current-1

  Best-2 current-2

  : :

  Best-k current-k // best and current (worst) profit values at the kth step (generation)

  : :

  best-maxNoOfEntries current-maxNoOfEntries

2.4. Representation

Use any suitable encoding scheme in your implementation. Normally, a binary array/string is used to represent candidate solutions, assuming that perturbative heuristics/operators will be implemented.

2.5. Solution Initialisation

You can randomly initialise a solution or use a constructive heuristic of your choice (e.g., greedy heuristic#3 above).

2.6. Heuristics/Operators

Appropriate set of heuristics/operators based on mutation (e.g., bit flip), hill climbing (e.g., steepest/best gradient, next/first gradient, Davis's bit hill climbing) and crossover (e.g., one point crossover, uniform crossover) should be implemented depending on the choice of your solution approach. All mutation operators should be parameterised using intensity of mutation (IOM), and the hill climbing operators using depth of search (DOS). There is no need parameterising crossover operators. The parameter settings for standard operators are discussed below. If you design a non-standard operator, it is up to you to decide on how the parameter setting is used to control the operator.

2.6.1. Intensity of Mutation (IOM)

The intensity of mutation setting should be used to control the number of $times$ that a mutation (e.g. bit flip) or crossover operation (e.g., one point crossover) is performed using the following mapping.

1. $0.0 \leq IOM < 0.2 \Rightarrow times = 1$
2. $0.2 \leq IOM < 0.4 \Rightarrow times = 2$

3. 0.4 ≤ IOM < 0.6 ⇒ $times$ = 3
4. 0.6 ≤ IOM < 0.8 ⇒ $times$ = 4
5. 0.8 ≤ IOM < 1.0 ⇒ $times$ = 5
6. IOM = 1.0 ⇒ $times$ = 6

Example: If you implement random *bitFlip* as the basis mutation operator which chooses a bit randomly in a given candidate solution and flips its value, and assuming $IOM = 0.25$, then *bitFlip* should be invoked 2 $times$, successively.

2.6.2. Depth of Search (DOS)

The depth of search setting should be used to control how many times a complete pass on a solution or a move considered for an accept/reject decision should be repeated should be carried out in a hill climbing operator using the following mapping.

1. 0.0 ≤ DOS < 0.2 ⇒ $times$ = 1
2. 0.2 ≤ DOS < 0.4 ⇒ $times$ = 2
3. 0.4 ≤ DOS < 0.6 ⇒ $times$ = 3
4. 0.6 ≤ DOS < 0.8 ⇒ $times$ = 4
5. 0.8 ≤ DOS < 1.0 ⇒ $times$ = 5
6. DOS = 1.0 ⇒ $times$ = 6

Random mutation hill climbing: A move considered for an accept/reject decision should be repeated for ($times$ x *n*) times.

Steepest/best gradient, Next/first gradient, Davis's hill climbing: $times$ represents the number of complete passes over a solution. If there is no improvement in a pass, the algorithm should be terminated.

2.7. Objective Function and Delta Evaluation

You can use the Equation (1) as a part of the maximising objective function in your approach for evaluating a candidate solution, if you can ensure that any generated new solution is **feasible**, that is the maximum capacity is not exceeded – every solution satisfies the inequality in Equation (2).

For example, given $n = 3$ with the associated profits $\{p_1 = 50, p_2 = 60, p_3 = 140\}$ and weights $\{w_1 = 15, w_2 = 10, w_3 = 20\}$, $W = 30$, the solutions <101> and <111> are *infeasible* solutions as the items would weigh 35 and 45, respectively, exceeding the capacity of 30 for this problem instance. The objective values of 190 and 250, respectively, for those solutions are not valid and Equation (2) cannot be used to assess any infeasible solution. However, <011> is a *feasible* solution with the objective value of 200, for which Equation (2) can be used as the objective function.

If you plan to allow infeasible solutions during the search process in your approach, then design a modified maximising objective function which can evaluate infeasible solutions producing, perhaps a negative profit if there is any infeasibility, and taking into account the degree of infeasibility of a solution. So, <101> should have a "better" negative objective value than <111> based on your maximising objective function, since it is closer to the maximum capacity, that is $W = 30$.

Delta evaluation should be implemented to make the objective value calculations faster whenever a heuristic/operator is invoked. Delta evaluation exploits that fact that including an excluded item in the knapsack (e.g., x_j becoming 1 from 0), increases the overall profit/objective value (by p_j), or excluding an included item in the knapsack (e.g., x_j becoming 0 from 1), decreases the profit (by p_j).

Note that when delta evaluation is used for crossover heuristics, the calculation time can be slower. So, it would be acceptable if the delta evaluation is implemented for mutation and hill climbing heuristics only.

2.8. Termination Criterion

The user should be able to identify the maximum number of evaluations for termination. Any other additional criteria are up to you.

3. Project Completion

The project assessment constitutes 30% of the overall as a part of the coursework.

In order to complete your AIM project, please follow the subsequent steps referring to Moodle for further details:

1) run your code on the released test as well as hidden instance(s) to be released in March,
2) complete the project report based on the results using the provided template,
3) submit your code and report following the instructions provided in Moodle by the submission deadline, and
4) attend your online demonstration.

The project demonstrations are a compulsory component of the assessment which allows us to scrutinise your work. You must attend the online demonstrations in order to obtain marks for your implementation. Failure to attend the online demonstration will result in a **zero mark** for the **entire project**.

PLAGIARISM WARNING: This project should be completed individually. Under no circumstances should you work with fellow students (you may discuss the project but never copy each other). The work must be your own and your own only. If plagiarism is suspected, ALL submissions will be cross-checked using plagiarism detection tools, and those offending students will face severe penalties per the university's strict plagiarism rules!