Binary Knapsack Problem resolution with Branch and Bound

Author: Gabriele Boscarini 2063145

Resolution strategy

I adopted a branch and bound algorithm written in Python code. I've created two files: **data_generator.py** and **main.py**. The description of the classes and methods adopted in the two files is in the following.

File 1: data_generator.py

Methods: **generate_data**

This file contains a method for generating inputs data as requested by the project description. The method generates 5 instances for every combination of the parameters and it writes them in a .txt file, separating the instances with a blank line for better legibility.

File 2: main.py

Classes:

- **Node**
- **item**

Methods:

- **dot_product**
- **relaxer**
- **relaxation solver**
- **branch_and_bound**

This file imports data from the input file and writes solutions on the output file.
The format of the output file is the same as the input file.

The item class:

parameters:

- **profit** = profit of the item
- **weight** = weight of the item
- **index** = index of the items in the items_list

attributes:

- **profit**
- **weight**
- **index**
- **relative profit** = ratio of profit and weight of the item

This class describes the components to be placed in the knapsack. Every component has a profit, a weight and an index to identify it in the items list. Relative profit is an attribute that needs to compute an optimal solution.

The Node class:

parameters:

- **level** = Number of the node, needs to identify it in the binary tree.
- **constraints** = constraints to be respected when computing optimal solution
- **w** = the capacity of the knapsack associated with the node
- **items_list** = list of the items of the knapsack associated with the node

attributes:

- **level**
- **x_opt** = optimal solution associated with the node
- **upper_bound** = profit obtained with the optimal solution
- **relaxation index** = index of the critical element of the optimal solution

This class describes the nodes of the branch and bound binary tree. A Node object embeds as an attribute the optimal solution (x_opt) and the profit (upper_bound) of the problem associated with it. The solution is computed by making use of the function **relaxation_solver.**

relaxation_solver Method:

parameters:

- **items_list** = list of the knapsack items
- **w** = capacity of the knapsack
- **x_opt** = constraints

output:

- **x_opt** = optimal solution (integer or fractional)
- **upper_bound** = profit obtained with the optimal solution
- **relaxation index** = index of the critical element of the optimal solution

This method solves with a greedy algorithm the LP relaxation of binary knapsack problem. It copies in a list indices of items that respect the constraints (there is a 1 at its index in x_opt parameter list) and then it follows on this list the method explained in Fischetti, Ch 8.0.3. Then, because every item object is identified with an index in items_list, the method modifies by reference only the involved elements in items_list to build the optimal solution.

relaxer Method:

parameters:

- **direction** = 0 if we want to relax critical element to zero, 1 otherwise
- **relaxation_index** = index of the critical element to relax
- **x_opt** = optimal fractional solution, to relax

output:

- **new_constraints** = optimal fractional solution relaxed to zero or to one

This function makes a copy of the solution to relax and then outputs the relaxed solution. If x_opt is integer, no relaxation is computed.

branch_and_bound Method:

parameters:

- **items_list** = list of the knapsack items
- **w** = capacity of the knapsack

output:

- **x_opt** = optimal integer solution found
- **profit** = max profit of the knapsack binary problem
- **number_of_nodes** = number of nodes created by algorithm
- **Time** = runtime of an instance
- **optimality gap** = optimality gap if time limit reached
- **solved** = True if problem solved within time limit, False otherwise

This function implements branch and bound algorithm. First, the root is generated. The binary tree is implemented with a queue (list). While the queue is not empty, at each iteration of the algorithm, a node to process is chosen by Depth First Search. Then two child nodes are generated, with the relaxed solution (one relaxed to 0, the other to 1) of the father node as a parameter in input. if the optimal solution of the current child is not integer and its upper_bound is greater than the actual max profit, then the child is inserted in the queue of active nodes. Otherwise, if the optimal solution of the current child is integer and the upper_bound is greater than the current max profit, current optimal solution and current max profit are updated and all the nodes that have an upper_bound lower than the current max profit are erased from the queue, simulating a recursive inspection of the binary tree.

Computational results

Evaluating the algorithm on input data, it came out that all instances have been solved within the time limit of 5 minutes. Indeed the optimality gap for each instance is zero. Results of each instance are shown in the following tab:

instances	n_items	nodes	Time	opt gap	solved
1	50	55	0.0016	0	True
2	50	49	0.0015	0	True
3	50	49	0.0014	0	True
4	50	53	0.0015	0	True
5	50	51	0.0016	0	True
1	60	61	0.0021	0	True
2	60	61	0.0021	0	True
3	60	63	0.0021	0	True
4	60	67	0.0022	0	True
5	60	59	0.0020	0	True
1	70	71	0.0028	0	True
2	70	69	0.0027	0	True
3	70	71	0.0028	0	True
4	70	73	0.0028	0	True
5	70	61	0.0025	0	True
1	80	79	0.0035	0	True
2	80	89	0.0039	0	True
3	80	93	0.0038	0	True
4	80	83	0.0036	0	True
5	80	89	0.0037	0	True
1	90	105	0.0048	0	True
2	90	89	0.0042	0	True
3	90	91	0.0043	0	True
4	90	93	0.0043	0	True
5	90	93	0.0044	0	True
1	100	103	0.0053	0	True
2	100	101	0.0053	0	True
3	100	105	0.0054	0	True
4	100	105	0.0054	0	True
5	100	105	0.0054	0	True