-General description of the Quadratic Knapsack problem:

It is an optimization problem in which we have to pack a subset of items (from a set of items) in a knapsack, so that we can maximize the total profit given by the items in the knapsack.

The knapsack has a limited capacity (constraint), and each object has its own weight, its own individual coefficient and a set of 0 to n-1 ('n' being the cardinality of the set of items) quadratic coefficients (a quadratic coefficient is an added profit to the item that owns that coefficient if it is packed with another specific object in the knapsack).

The mathematical model is:

$$\text{maximize} \quad \sum_{i \in N} \sum_{j \in N} p_{ij} x_i x_j$$

$$\text{subject to} \quad \sum_{j \in N} w_j x_j \leqslant c,$$

$$x_j \in \{0, 1\}, \quad j \in N.$$

Being

$P_{ij,}$ the quadratic profit when objects *i* and *j* are packed

$x_{i,}$ a discrete variable with value 1 if object *i* is packed, 0 otherwise

$x_{j,}$ a discrete variable with value 1 if object *j* is packed, 0 otherwise

$w_{j,}$ the weight of object *j*

N, the total number of objects in the set

c, the capacity of the knapsack

Description and pseudocode of the multi-start heuristic

In my multi-start heuristic I combined the k_best approach with the local_search ‘on top’ of it to try to get a better solution than when applying just my constructive heuristic method to the list of items where I’m working on, the pseudocode is the following:

```
S=Constructive_heuristic to the item's list
Repeat 10 times
    S'=Randomized_Constructive_Heuristic applied to the item's list
    S*= local_search_first_improvement to S'
    IF profit(S*)>profit(S)
    S=S*
return S
```

Constructive heuristic description:

In my method to solve the QKP I used, to calculate the decision variable of each object in the set, the following formula:

(Profit+Quad_profits_average)/weight

And, after introducing the items in the knapsack from greatest to lowest decision variable value, the remaining (when the next object could not 'fit' because of the weight) are introduced from lowest to highest weight, until there is no 'room' for the next one.

Concise description of the code (made with the help of ChatGPT):

1. Data Structures Initialization:

- Define a class `ij` to store the quadratic index of items, where `j` represents the index of the item it points to, and `value` is the quadratic profit associated.

- Create a class `Item` with attributes such as `c_i` (linear coefficient), `c_ij` (list of quadratic coefficients represented by instances of `ij`), `ev_func` (evaluation function), `weight`, and `packed` (a flag indicating if the item is packed or not).

2. Evaluation Function Calculation:

- Create a function `calc_ev_func(List_of_Items, index)` to calculate the evaluation function of a given item in the list.

- Iterate through the quadratic coefficients of the item, checking if the items it points to are packed, and accumulate those profits.

- Calculate the average of the accumulated values and add the item's linear coefficient to get the final evaluation function.

3. Heuristic Function:

- Develop a heuristic function `heuristic(Items, Knapsack_capacity)` that initializes a copy of the items list (to modify the copy and not the original list)

- Iterate through unpacked items, calculating their evaluation functions and selecting the one with the highest value that also fits into the knapsack.

- Pack the selected item, update the remaining capacity, and repeat until no more items can be packed.

- Return the modified list of items.

4. Local Search (First Improvement):

- Implement a function `local_search_first_improvement(S, Knapsack_capacity)` to perform a local search on the given solution ‘S’.

- Initialize variables to keep track of the current solution, its profit, remaining capacity, etc.

- Iteratively remove the heaviest packed item, attempt to replace it with unpacked and fitting items having a higher evaluation function, and update the solution if an improvement is found.

- Continue until no further improvement is possible.

-Return the obtained list, in the worst case the profit ‘returned’will be the ame as the one of the original solution

5. Utility Functions:

- Define utility functions like `A_belongs_to_B(Num, List_of_Num)` to check if a number belongs to a list.

- Create `weight_consumed(List_of_Items)` to calculate the total weight consumed by a solution.

- Develop `calculate_profit_given_packed_items(List_of_Items)` to calculate the profit of a solution based on packed items.

- Implement `quicksort_by_attribute(List_of_Items, attribute)` to sort a list of items based on a specified attribute.

6. Instance Reader:

- Develop `Instance_reader(file_name)` to read an instance file, extracting information such as the number of items, linear and quadratic coefficients, knapsack capacity, and weights.

- Return a list containing the number of items, a list of items, and the knapsack capacity.

7. Feasibility Check:

- Implement `is_solution_feasible(List_of_Items, Knap_capacity)` to check if a solution is feasible by comparing the consumed knapsack capacity with the given capacity.

8. Randomized Constructive Heuristic:

- Create `Randomized_Constructive_Heuristic(Items, Knapsack_capacity)` to construct a solution by randomly selecting unpacked items with high evaluation functions and updating the solution until no more items can be added.

-Steps:

Create List_of_Items as a copy of the original item list.

Initialize Capacity_verif to knapsack capacity.

Set k (number of items to select) and initialize other variables.

While a valid item with a positive evaluation function is found:

Select the top k unpacked items with the highest evaluation functions.

Randomly choose one item from the selected subset.

Pack the chosen item and update the remaining capacity.

Return the modified List_of_Items.

9. Main Calculation and Output:

- Develop `Calculate(Instance_Path, optimal)` to orchestrate the entire process.

- Read the instance, apply the heuristic, perform local search, measure computing time, and return relevant results including profit, computing time, and optimality gap.

10. Execution:

- Execute the `Calculate` function with appropriate instance paths and optimal solutions to obtain results and compare them against optimality.

My implementation is finally ready, except for a problem that I have, related to the implementation of lists, the problem was that every movement I did to a list, it affected the main list of items, the one that I read directly from the file. I still don't know how to solve it, but I tried my best to correct the results and not let them be altered, for example: Create a copy of the received list of items so that the main list is left untouched. That did not work either, but time is limited in this case,so I´ll upload the code as it is now

Instance description:

The processed instances are randomly generated, can be found in https://cedric.cnam.fr/~soutif/QKP/QKP.html , and have the following format:

Each (QKP) instance correspond to a file containing the following informations:

- the reference of the instance (r_10_100_13 in the following example)
- the number of variables (*n*) (10 in the following example)
- the linear coefficients (*c_i*) of the objective function (91 78 22 4 48 85 46 81 3 26)
- the quadratic coefficients (*c_ij*) of the objective function
- a blank line
- 0 if the constraint is of type <= (i.e. always since we are considering (QKP) instances) and 1 if the constraint is an eglity constraint
- the capacity of the knapsack (145)
- the coefficients of the capacity constraints (weights, *a_i*) (34 33 12 3 43 26 10 2 48 39)
- some comments

Here is an example concerning a 10-variable instance:

```
r_10_100_13
10
91 78 22 4 48 85 46 81 3 26
55 23 35 44 5 91 95 26 40
92 11 20 43 71 83 27 65
7 57 33 38 57 63 82
100 87 91 83 44 48
69 57 79 89 21
9 40 22 26
50 6 7
71 52
17

0
145
34 33 12 3 43 26 10 2 48 39
```

Hardware description:

Item	Value
OS Name	Microsoft Windows 10 Home
Version	10.0.19045 Build 19045
[redacted]	
OS Manufacturer	Microsoft Corporation
[redacted]	
System Manufacturer	Dell Inc.
[redacted]	
System Type	x64-based PC
[redacted]	
Processor	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, 2592 Mhz, 6 ...
BIOS Version/Date	Dell Inc. 1.25.0, 6/6/2023
[redacted]	
BIOS Mode	UEFI
Installed Physical Mem...	16.0 GB
Total Physical Memory	15.8 GB
Available Physical Mem...	4.46 GB
Total Virtual Memory	18.1 GB
Available Virtual Memory	2.24 GB
Page File Space	2.38 GB

Programming language: Python

Results:

Instance name	Total profit obtained	Execution time	Optimal profit	Average percentage deviation of profit obtained with rand. const. heuristic and the appl. of Local search to that solution	Percentage deviation from optimal profit
200_100_1	894637	10.99975276	937149	1.076637787	4.5363
300_50_10	898959	1.796266079	996070	--------	9.7494
300_25_7	421094	1.767256498	485263	--------	13.2236
200_100_8	661026	0.536378622	782443	--------	15.5177
200_100_5	656932	0.589403152	786635	--------	16.4883
300_25_4	357122	1.706212521	444759	--------	19.7044
300_50_3	700342	1.685188055	875788	--------	20.0329
300_25_10	289504	1.68018508	383377	--------	24.4858
300_50_8	568292	5.719044447	767977	1.380263216	26.0014
300_50_6	529457	37.17521405	734053	1.072797224	27.8721
200_100_7	499595	10.57044792	701094	1.542449384	28.7407
300_50_9	537579	37.072124	761351	0.938652738	29.3914
300_50_5	476172	1.429000139	727820	--------	34.5756
300_25_2	172629	1.290922403	281990	--------	38.7819
200_100_9	382989	0.506357193	628992	--------	39.1107
300_25_6	144246	1.138803244	269782	--------	46.5324
300_25_9	123737	1.282887459	250761	--------	50.6554
300_50_1	228914	1.077776194	513379	--------	55.4103
200_100_10	143578	10.40633464	378442	2.768529998	62.0608
300_50_4	106030	0.713502884	307124	--------	65.4765
200_100_2	71968	0.208146572	303058	--------	76.2527
200_100_4	16521	0.118088245	100838	--------	83.6163
300_25_1	4595	27.06908536	29140	4.896626768	84.2313
300_50_2	15162	0.350264072	105543	--------	85.6343
200_100_6	4932	0.089064837	41171	--------	88.0207
200_100_3	2933	0.061055422	29367	--------	90.0126
300_50_7	3822	0.183129072	43595	--------	91.2329
300_25_5	1114	0.146115541	14988	--------	92.5674
300_25_8	681	0.127088785	9343	--------	92.7111

I finally got my results, they showed that:

-It seems that an instance´s “n” impacts more execution time than the instance´s density.

-Not in every case the randomized constructive heuristic returned better results than the constructive heuristic, but in those cases where it happened, the percentage deviation is, maximum, 4.89% and minimum 0.93%

-It seems that the local search method had 0 effects in my solutions.

-The average deviation of the result obtained and the optimal result in the resolution of a given instance is 48.7%, being the farthest (or worst) 92.7%, and the closest (or best) 4.5363%

-The average Execution time is 5.4s, ranging from 0.06s to 37.1s

-The time an instance takes to be solved is not directly proportional to the percentual difference of the obtained profit with respect to the optimal profit. An example of this is the instance 300_25_1, delaying 27.06 seconds, but not even achieving 20% of the optimal solution