LOG 734 Heuristics in Analytics

Assignment 2

Students:

191360	Alexandr Reznik
191363	Andrey Belski
191362	Ivan Terekh

1. Introduction

In this assignment, we should design and implement an improvement heuristic algorithm for the multidimensional knapsack problem (MKP). The problem is described in the section 2. Problem definition. As improvement heuristics needs a starting point we used several construction heuristics (described in section 3. Construction Heuristics) to create initial solutions for the problem. Two improvement heuristic were created (both described in 4. Improvement Heuristics) for the problem. Both of them are based on the jagged search using different neighborhoods. The main heuristics algorithm gives good results on the tests for this assignment (comparing to the results of previous years) and usually it is able to improve the starting solution. The second heuristics shows worse results for the test cases but it works faster and can be used for larger instances. Problem definition, pseudocode for common heuristic algorithms and inspiration are taken from LOG 734 lectures and lecturer.

Programing language used for the implementation is c++.

2. Problem definition

Multidimensional knapsack problem can be formulated as:

$$max \sum_{j \in V} c_j x_j$$

$$\sum_{j \in V} a_{ij} x_j \leq b_i, \qquad i \in M$$

$$x_j \in \{1,0\}, \qquad j \in V$$

Where $V = \{1, \dots, n\}$ is a set of different items.

$M = \{1, \dots, m\}$ is a set of dimensions.

Item $j \in V$ has a given size $a_{ij} \geq 0$ in dimension $i \in M$. (r[i][j] in the code)

There is a fixed capacity $b_i > 0, i \in M$.

Item $j \in V$ has a given profit $c_j \geq 0$. (p[j] in the code)

x_j denotes if item j is selected or not.

3. Construction heuristics

There are three construction heuristic algorithms used to create an initial solution. All of them are very similar and follow the same pseudocode. The only difference is line 4.

1: $V^{\#} := \{1, \dots, n\}$

2: $g_i := 0 \; \forall \; \boldsymbol{i} \in \{\mathbf{1}, \dots, \boldsymbol{m}\}$

3: while $V^{\#} \neq \emptyset$ do

4: calculate $w_j \; \forall \; \boldsymbol{j} \in \boldsymbol{V^{\#}}$

5: Choose a variable $j^* = \arg\max_{j \in V^{\#}} \{\frac{c_j}{w_j}\}$

6: **if** $\exists \; \boldsymbol{i} \in \{\mathbf{1}, \dots, \boldsymbol{m}\}: g_i + a_{ij^*} > b_i$ **then**

7: $x_{j^*} := 0$

8: **else**

9: $x_{j^*} := 1$

10: $g_i := g_i + a_{ij^*} \; \forall \; \boldsymbol{i} \in \{\mathbf{1}, \dots, \boldsymbol{m}\}$

11: **end if**

12: $V^{\#} := V^{\#} \setminus \{j^*\}$

13: **end while**

14: Output $f = \sum_{j \in \{1,\dots,n\}} c_j * x_j$

In the code each heuristic is presented as a function `solveConstruction(…);` with the different last argument.

Construction1:

The first heuristic (uses `calculateRateEasy(…)` as the last argument of `solveConstruction(…)`) is the simple heuristic algorithm. The line 4 is $w_j = \sum_{i \in \{1,\dots,m\}} \frac{a_{ij}}{b_i} \; \forall \; \boldsymbol{j} \in \boldsymbol{V^{\#}}$**.** The weights are static (the amount of used space g_i isn't used to calculate the weight). That means that calculations can be done outside the "while" cycle to improve performance. This can be implemented in future. But this change doesn't affect the asymptotic complexity of the whole heuristic algorithm because other construction heuristics calculate weight dynamically.

Construction2:

The second heuristic (uses `calculateRate(…)` as the last argument of `solveConstruction(…)`) is the heuristics from the assignment 1. The line 4 in pseudocode for this heuristic is $w_j = \sum_{i \in \{1,\dots,m\}} \frac{a_{ij}}{(b_i - g_i)} \; \forall \; \boldsymbol{j} \in \boldsymbol{V^{\#}}$.

Construction3:

The third heuristic (uses `calculateRateNorm(…)` as the last argument of `solveConstruction(…)`) is one of the heuristics used in the previous year. The line 4 there is the following:

$$w_j = \sqrt{\sum_{i \in \{1,\dots,m\}} \left(\frac{a_{ij}}{(b_i - g_i)}\right)^2} \quad \forall\, \boldsymbol{j} \in \boldsymbol{V}^{\#}.$$

To obtain better results, it was decided to use multiple different construction heuristics as different starting points. This helps the algorithm to find a better solution because using different starting points lets us start the local search from different parts of the solution space and find more than one local optimum. This is not worth than finding only one but could be better. The second and the third solutions are quite good but a bit similar to local optimem. The first one is added for diversity.

4. Improvement heuristics

The first attempt of improvement heuristics was a local search with double-flip neighborhood operator $N^2(x) = \{x' : \sum_{j=1}^{n} |x_j - x'_j| = 2\}$. The pseudocode of the local search is the following:

1: input: initial solution, x

2: input: neighborhood operator, N

3: **while** there is a $x' \in N(x)$ that is better than x **do**

4: Choose the best neighbor $x' \in N(x)$ that is better than x, and update $x := x'$.

5: **end while**

In the code see the function `solveImprovementFlip2(…)` with the argument `jagged` set to false.

We compare the solutions using the value of objective function. We don't have any infeasible solutions during construction or improvement and we don't have to deal somehow with the comparison of infeasible solutions.

The time used for any test instance was less than a second and we decided that we could use triple-flip neighborhood operator $N^{3*}(x) = \{x' : \sum_{j=1}^{n} |x_j - x'_j| = 3\}$. Afterwards it was changed to improve performance. The improved operator is:

$$N^3(x) = \{x' : \sum_{j=1}^{n} |x_j - x'_j| = 3\,, -3 < \sum_{j=1}^{n} (x_j - x'_j) < 3\}.$$

This change allow us to skip neighbor solutions where all 3 variables we try to change have the same value. If we try to remove 3 items the step won't be improving and we don't want to spend time calculating it. Adding 3 items doesn't make any sense as we have a greedy construction at the beginning, that means we can't only put item in a knapsack without removing anything.

In the code see the function `solveImprovementFlip3(…)` with the argument `jagged` set to false.

We made an analysis of using different improvement heuristics.

In the following table left column is for test case name. All tests were run using the best result of all construction heuristics as an initial solution. Column "0" contains the initial solution cost (with no improvement). Numbers in the head row are improvement heuristics and their order. 2 means local search with N^2 operator. 3 means local search with N^3 operator. 2 3 means local search with N^2 operator and then local search with N^3 operator for the result of first improvement. The best solution for each case is highlighted.

Test\Heuristics	0	2 3	2 3 2	3 2	3 2 3	3	2
100-5-01	23984	24192	24192	24285	24285	24160	24192
100-5-02	24274	24274	24274	24274	24274	24274	24274
100-5-03	23317	23449	23449	23476	23476	23476	23449
100-5-04	22893	23273	23273	23094	23094	23094	23040
100-5-05	23826	23826	23826	23826	23826	23826	23826
250-10-01	58393	58776	58776	58965	58965	58965	58776
250-10-02	58145	58145	58145	58145	58145	58145	58145

250-10-03	57267	57528	57528	57624	57624	57624	57408
250-10-04	60296	60624	60624	60623	60623	60593	60624
250-10-05	57643	57643	57643	57643	57643	57643	57643
500-30-01	114878	115038	115038	115038	115038	114878	115038
500-30-02	113760	114038	114038	114246	114246	114246	114038
500-30-03	115340	115576	115576	115853	115853	115853	115576
500-30-04	113807	114005	114005	114473	114473	114473	114005
500-30-05	115226	115532	115532	115928	115928	115928	115532

Table 1

We can see that there is profit from using one improvement after another. And it's better to use 3 2 than 2 3.

After that we decided that it's possible to use even quadruple-flip neighborhood local search for small instances ($n \leq 250$). Quadruple-flip operator is $N^{4*}(x) = \{x' : \sum_{j=1}^{n} |x_j - x'_j| = 4\}$ Afterwards was changed to improve performance. The improved operator is:

$$N^4(x) = \{x' : \sum_{j=1}^{n} |x_j - x'_j| = 4\,, -4 < \sum_{j=1}^{n} (x_j - x'_j) < 4\}.$$

In the code see the function `solveImprovementFlip4(…)` with the argument `jagged` set to false.

The results are shown in the following table:

	0	2 3	2 3 2	3 2	3 2 3	3	2	4	4 3 2 1
100-5-01	23984	24192	24192	24285	24285	24160	24192	24329	24329
100-5-02	24274	24274	24274	24274	24274	24274	24274	24274	24274
100-5-03	23317	23449	23449	23476	23476	23476	23449	23496	23496
100-5-04	22893	23273	23273	23094	23094	23094	23040	23338	23338
100-5-05	23826	23826	23826	23826	23826	23826	23826	23932	23932
250-10-01	58393	58776	58776	58965	58965	58965	58776	59021	59021
250-10-02	58145	58145	58145	58145	58145	58145	58145	58346	58346
250-10-03	57267	57528	57528	57624	57624	57624	57408	57511	57511
250-10-04	60296	60624	60624	60623	60623	60593	60624	60875	60875
250-10-05	57643	57643	57643	57643	57643	57643	57643	57726	57726

Table 2

It becomes obvious that quadruple-flip operator local search obtains better results for small instances.

After getting more inspiration from the lecture, we decided to implement jagged local search instead of usual that we had. Jagged local search makes a move if the new solution is better than the previous one (not the current). The pseudocode is the following:

1: input: initial solution, x

2: input: neighborhood operator, N

3: $x_{prev} := x$

3: **while** there is a $x' \in N(x)$ that is better than x_{prev} **do**

4: Choose the best neighbor $x' \in N(x)$ that is better than x, and update $x_{prev} = x, x := x'$.

5: **end while**

Making search jagged is achieved by making the argument `jagged` of the functions `solveImprovementFlip2(…),solveImprovementFlip3(…),solveImprovementFlip4(…)`set to false.

We decided that the jagged search is better than the usual one after looking at the following tables.

Test	2	2 jagged	dif
100-5-01	23393	23577	184
100-5-02	24225	24225	0
100-5-03	23320	23320	0
100-5-04	22306	22306	0
100-5-05	23328	23328	0
250-10-01	58341	58341	0
250-10-02	58320	58320	0
250-10-03	57271	57271	0
250-10-04	60538	60538	0
250-10-05	56710	57103	393
500-30-01	113299	113530	231
500-30-02	112924	112924	0
500-30-03	114007	114140	133
500-30-04	112744	112761	17
500-30-05	113335	113919	584

Table 3

Test	3	3 jagged	dif
100-5-01	23919	23919	0
100-5-02	24117	24117	0
100-5-03	23361	23361	0
100-5-04	23104	23104	0
100-5-05	23592	23592	0
250-10-01	58285	58285	0
250-10-02	58483	58483	0
250-10-03	57228	57228	0
250-10-04	60036	60388	352
250-10-05	56643	57208	565
500-30-01	113424	114594	1170
500-30-02	112733	113468	735
500-30-03	114679	115108	429
500-30-04	112907	113515	608
500-30-05	114088	114088	0

Table 4

Test	4	4 jagged	dif
100-5-01	24196	24196	0
100-5-02	24274	24274	0
100-5-03	23464	23464	0
100-5-04	23241	23241	0
100-5-05	23991	23991	0

250-10-01	58985	58985	0
250-10-02	58556	58556	0
250-10-03	57744	57744	0
250-10-04	60788	60788	0
250-10-05	57594	57681	87

Table 5

Column "Test" is used for the test name; 2, 3 and 4 for usual search; 2 jagged, 3 jagged and 4 jagged for jagged search; dif for difference between jagged and usual searches (jagged - usual).

Based on the results obtained above, it can be concluded that the pattern for improvement now is the sequential use of jagged local searches with operators in descending order of flips (it follows from *Table 2*): (4j, n<= 250)-3j-2j-1j – i.e. each operator is applied to the solution obtained in the previous step. Using exactly the sequence doesn't make the solution worse (this also follows from *Table 2*). The quadruple operator applies only when n<=250, otherwise the performance will suffer:

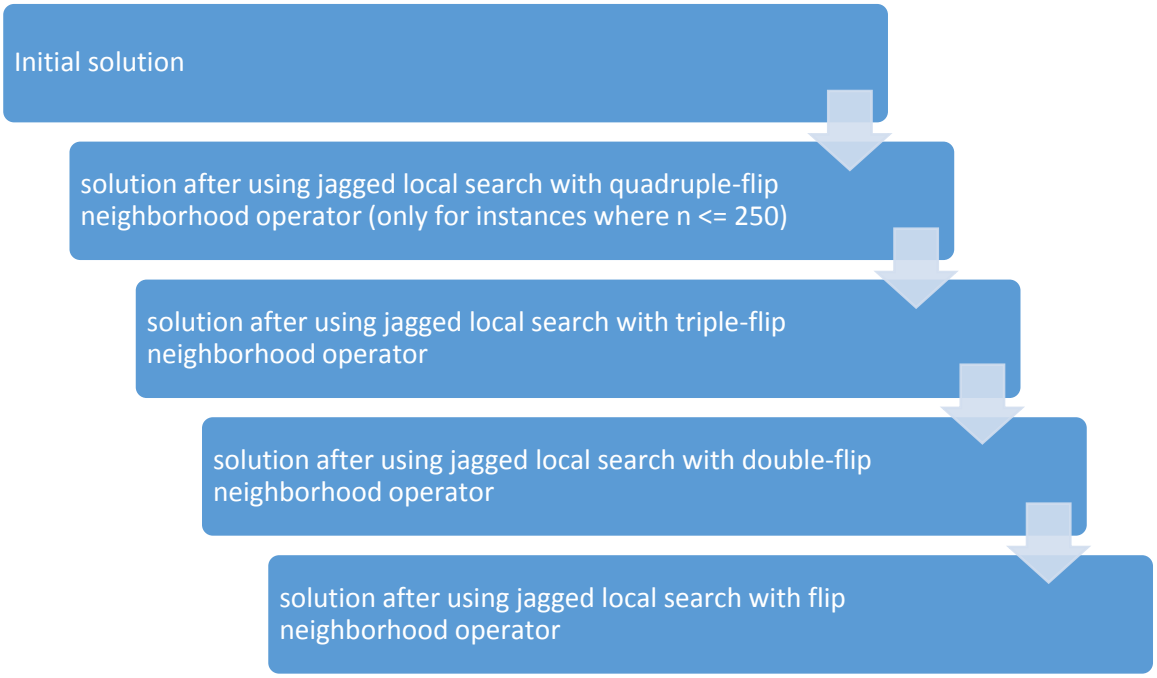

Chart 1

Where flip neighborhood operator is $N^1(x) = \{x' : \sum_{j=1}^{n} |x_j - x'_j| = 1\}$

Previously we used one initial solution to improve. By improving several a better result could be obtained. Here are the results of improving different initial solutions by the same improvement heuristic (4j <= 250)-3j-2j-1j.

Construction1	Construction2	Construction3
24196	24329	24381
24274	24274	24274
23480	23496	23494
23241	23389	23338
23991	23932	23959

58285	59021	58740
58483	58629	58145
57228	57821	57624
60501	60875	60241
57256	57801	57571
114594	115154	115038
113585	114199	114246
115108	116228	115853
113515	114645	114473
114498	115557	115928

Table 6

Even though improvement of severa; initial solutions leads to a better result, it is not a good idea to use it because this approach requires more computing time. The majority of time is spend on quadruple-flip search for the cases where n = 250. We cannot run quadruple-flip search for cases where n = 250 three times (for each of the construction heuristics). That is why we decided to use quadruple-flip search for cases where n = 250 only for one starting solution (which is obtained by our Construction2 because it shows better results according to the table).

The final algorithm is the following (it is called Good in results):

1: solution1 := Construction1

2: Improve solution1 using the method (4j <= 100)-3j-2j-1j.

3: solution2 := Construction2

4: Improve solution1 using the method (4j <= 250)-3j-2j-1j.

5: solution3 := Construction3

6: Improve solution1 using the method (4j <= 100)-3j-2j-1j.

7: result := the best solution among solution1, solution2, solution3.

Implemented as `solveBestImprovement()` function.

One more algorithm which is suitable for larger instances is using (3j <= 100)-2j-1j improvement heuristics for 3 different initial solutions. It is called Quick in results.

5. Results

We can look at the results of two improvement heuristics (Good and Quick) described in the previous section. The next plot shows how much the solution after using improvement heuristics is better than the best solution among obtained by the construction heuristics. We assume that it makes sense to compare results after improvement with the best solution among obtained by the construction heuristics not with our results from the first assignment because we added several construction heuristics and the initial solution can be better than the result in the first assignment. Anyway, if we compare with the results of the first assignment the difference would be larger and our heuristic will look even better than it is.

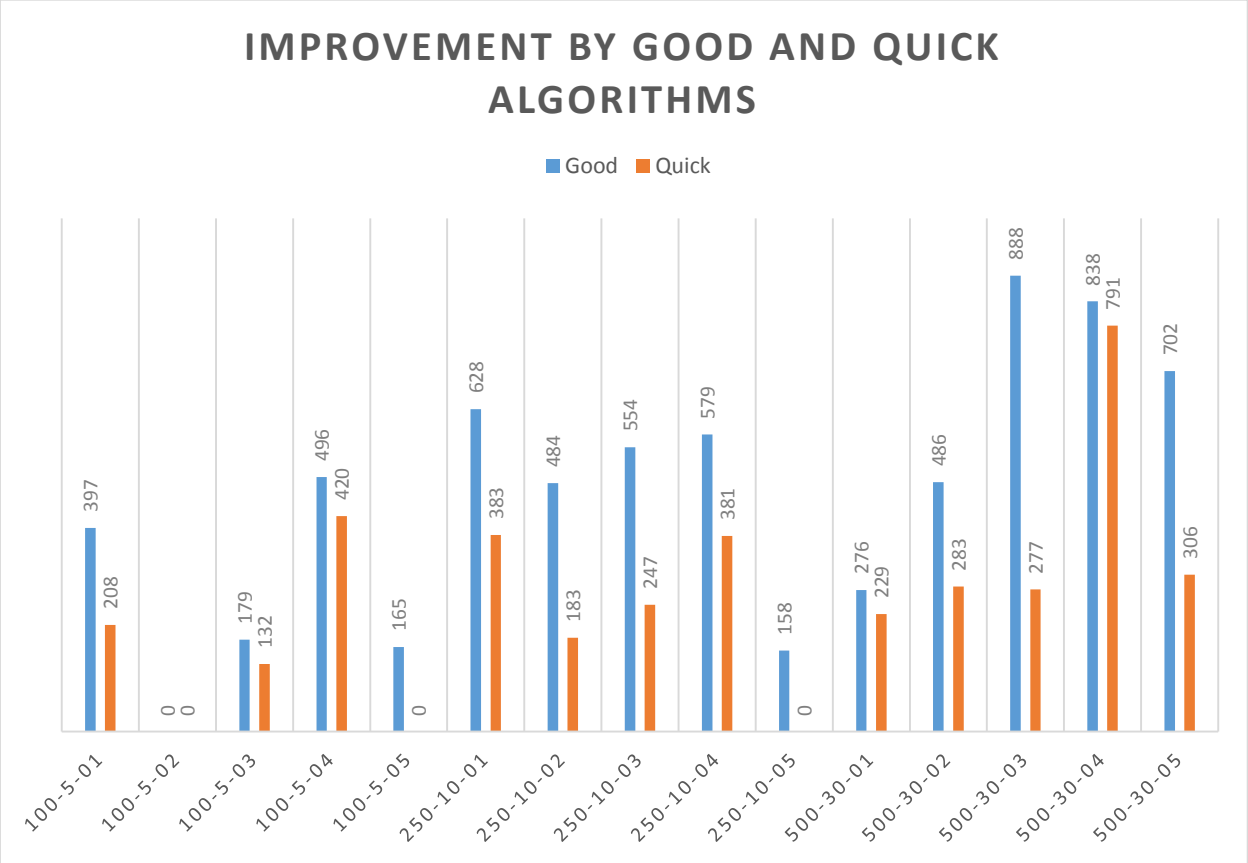

Chart 2

The results for Good improvement heuristics are in the following table:

Test	Result Good	Time(seconds)	Difference with 18-19 best result
100-5-01	24381	0,346	189
100-5-02	24274	0,213	49
100-5-03	23496	0,309	52
100-5-04	23389	0,417	127
100-5-05	23991	0,414	165
250-10-01	59021	8,919	149
250-10-02	58629	5,977	262
250-10-03	57821	7,446	-14
250-10-04	60875	5,818	251
250-10-05	57801	8,916	156
500-30-01	115154	7,016	-95
500-30-02	114246	3,548	228
500-30-03	116228	5,735	303
500-30-04	114645	4,302	97
500-30-05	115928	4,545	113

Table 7

The times in the table above are obtained using processor “Intel Core i7-4720HQ CPU @ 2.60 GHz”.

The running time is less than 10 second for any test case. Results are quite good comparing to the previous year’s results.

The next table is for Quick. Results are not that good but the running time is less than a second.

Result Quick	time	Difference with 18-19 best result
24192	0,007	0
24274	0,007	49
23449	0,007	5
23313	0,011	51
23826	0,009	0
58776	0,012	-96
58328	0,009	-39
57514	0,011	-321
60677	0,010	53
57643	0,013	-2
115107	0,135	-142
114043	0,111	25
115617	0,119	-308
114598	0,141	50
115532	0,140	-283

Table 8