Heuristics in Analytics

LOG 734 2021

Assignment 2

Improvement Heuristic

Vladislav Klyuev
Mark Drozd
Sviatlana Lapitskaya

Logistics Analytics
Høgskolen i Molde

March 2021

Introduction

With this report we experiment with improvement heuristics described in the lectures, namely with **FLIP**, **DOUBLE_FLIP** and **SWAP**. The definition of the problem is located in *Problem statement* part. Implementation language is C++.

Problem statement

Multidimensional knapsack problem can be formulated in the following way:

$$maximize \sum_{j=1}^{n} c_j x_j$$

$$s.t \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i = 1, ..., m$$
$$x_j \in 0, 1, \quad j = 1, ..., n$$

where the objective is to maximise the value of items we manage to fit into the knapsack. This knapsack is *multi-dimensional*, meaning it is limited by the numbers b_i in each of it's dimensions. The family of knapsack constraints represent the limitation of the knapsack volume. Thus, we strive to achieving the maximum value within a size-limited volume.

Heuristic description

An improvement heuristic tries to modify to find an already feasible solution (found by a construction heuristic in our case) by making perturbations in the solution structure.
The main motivation of the heuristic application arises directly from the name – we are aiming at achieving a better solution in terms of objective function. The nature of such a heuristic requires us to have one or more feasible solutions in a pool to build improvement upon. That is exactly why we first have to create an initial solution before utilizing the methods described below.
We first create an initial solution described in our construction heuristic proposal [1].
Then, the neighbourhood search is performed. The general pseudocode of it is described below.

Algorithm 1 Local Search

1: input : initial solution, x
2: input : neighbourhood operator, N
3: **while** there is a $x^{'} \in N(x)$ that is better than x **do**
4: Choose a neighbour $x^{'} \in N(x)$ that is better than x
5: **end while**
6: output : solution x

The idea of such algorithms is searching the neighbourhood for a feasible solution along which objective function strictly improves. We experiment with three neighbourhood operators:

- $N^1(x) = \{x^{'} : \sum_{j=1}^{n} |x_j - x_j^{'}| = 1\}$ - **flip** neighbourhood
- $N^2(x) = \{x^{'} : \sum_{j=1}^{n} |x_j - x_j^{'}| = 2\}$ - **double-flip** neighbourhood
- $N^{swap}(x) = \{x^{'} : \sum_{j=1}^{n} |x_j - x_j^{'}| = 2, \sum_{j=1}^{n} (x_j - x_j^{'}) = 0\} \subset N^2(x)$ – **swap** neighbourhood

Those definitions present the variants of the local search (Algorithm 1): step 4 of choosing a neighbour requires us to iterate through some set of both feasible and infeasible plans in search of one that is better than the starting point in terms of the objective. We choose to stay in the feasible region all along the way of the search (though of course it is not necessary – one might experiment leaving the feasible tetrahedron for a while hoping to jump to another hyperplane on the next iteration). Once we encounter a neighbourhood containing only solutions that are not better than where the current solution stands, we interrupt the process.

Implementation

Heuristic was implemented using C++ programming language based on the example presented in the tutorial and the code submitted in [1]. Instances were read from file and stored in vectors corresponding to the right hand side coefficients, the left hand side coefficients and the objective function coefficients. In addition to the functions implemented before, we added some more logic into our code: some micro-procedures such as *computeSolutionObjective* and *checkSolutionFeasibility* allow us to *quickly* perform necessary feasibility tests and obtain objective value from the solution. *Notice* that we now use the *bitset* class to follow the decision vector logic – this class is much faster than the regular boolean vector due to its machine-friendly implementation (it runs on machine words). Bitset allows us to iterate over *set* bits quickly, skipping all zeros between them (functions *_Find_next* that are understandable by GNU compilers), and therefore makes such operations as computing the objective much faster.
Additionally, we implemented a fast procedure *checkSolutionFeasibilityAfterPerturbing* that is responsible of checking feasibility of a solution after we flip or swap variables within it. This procedure does not iterate over entire constraint matrix to do this check, but instead utilizes row activities (values of rows) we precompute before perturbations and checks whether the new row values fit into the row bounds.
All heuristic logic of finding a neighbour is implemented within *yieldNeighbour* function – there we iterate through variables in a way dependent on the input method, and try to perturb solutions judging on improvement criteria and staying in the feasible region.
The main *localSearch* procedure repeats the process of finding the best neighbour until no such one is obtained. Then, it yields the one it managed to find. There is a posiibility to turn on the *first improvement heuristic* in the function's signature – up to now we only experimented with *best improvement local search*.
Notice that we included the *timeLimit* parameter into our method – the current time constraint is 60 sec, which is ludicrous compared to the actual efficiency of our method. By the time limit should be present nonetheless: it is a good practice with heuristic methods that run in quadratic time, and may become a crucial stopping criteria for much larger problems.

Results

The computations were done utilising the 2,3 GHz Dual-Core Intel Core i5 Processor with 8 GB 2133 MHz LPDDR3 Memory. The initial results of building improved solution upon the one obtained by our construction heuristic are presented in the following tables:

FLIP

Problem	Init Solution	Time, ms	Improved Solution	Time, ms
100-5-01	24049	0.135	N/A	0.004
100-5-02	23970	0.147	N/A	0.005
100-5-03	23317	0.097	N/A	0.004
100-5-04	22979	0.113	N/A	0.004
100-5-05	23826	0.088	N/A	0.004
250-10-01	58284	1.085	N/A	0.012
250-10-02	58293	1.223	N/A	0.012
250-10-03	57390	1.556	N/A	0.017
250-10-04	60467	0.986	N/A	0.012
250-10-05	57325	1.202	N/A	0.009
500-30-01	114839	13.63	N/A	0.035
500-30-02	113662	7.518	N/A	0.033
500-30-03	115805	8.932	N/A	0.034
500-30-04	114203	9.622	N/A	0.033
500-30-05	115471	8.964	N/A	0.03

DOUBLE_FLIP

Problem	Init Solution	Time, ms	Improved Solution	Time, ms
100-5-01	24049	0.133	24164	0.294
100-5-02	23970	0.128	24225	0.171
100-5-03	23317	0.132	N/A	0.109
100-5-04	22979	0.152	23294	0.204
100-5-05	23826	0.154	N/A	0.104
250-10-01	58284	1.011	58711	1.064
250-10-02	58293	0.618	58508	1.23
250-10-03	57390	1.191	57710	1.2
250-10-04	60467	1.265	60484	1.108
250-10-05	57325	1.235	57667	2.44
500-30-01	114839	14.96	115250	16.592
500-30-02	113662	11.49	114158	7.711
500-30-03	115805	11.576	115821	6.877
500-30-04	114203	13.585	114543	13.047
500-30-05	115471	10.349	115773	6.967

SWAP

Problem	Init Solution	Time, ms	Improved Solution	Time, ms
100-5-01	24049	0.14	N/A	0.027
100-5-02	23970	0.143	N/A	0.027
100-5-03	23317	0.153	N/A	0.029
100-5-04	22979	0.165	N/A	0.03
100-5-05	23826	0.102	N/A	0.02
250-10-01	58284	0.687	N/A	0.073
250-10-02	58293	0.652	N/A	0.081
250-10-03	57390	0.735	N/A	0.081
250-10-04	60467	0.639	N/A	0.07
250-10-05	57325	0.717	N/A	0.072
500-30-01	114839	9.139	N/A	0.328
500-30-02	113662	10.465	N/A	0.329
500-30-03	115805	11.334	N/A	0.356
500-30-04	114203	12.559	N/A	0.337
500-30-05	115471	10.493	N/A	0.322

As we can see, both **FLIP** and **SWAP** neighbourhood operators do not manage to help finding better solutions in the neighbourhood of solutions produced by our construction heuristic. Some row slacks appear to be too tight for these two heuristic methods of perturbing variables. But even if they manage to obtain another solution keeping variables in the solution domain, the quality of these solutions is not better, which is why the local search method returns a failure of success.

A better performance is yielded by **DOUBLE_FLIP** heuristic that flips two variables in the starting point. The quality of these solution is nevertheless worse than those obtained for the problem in previous years of development.

Thus, one might experiment with different combinations of these heuristics. Of course, there is no reason to run **DOUBLE_FLIP** after **FLIP** or **SWAP** since it is going to be the same as running the former purely. But our experiment of starting **FLIP** or **SWAP** *after* we manage to find solutions with **DOUBLE_FLIP** shows no change at all – **FLIP** and **SWAP** are still unsuccessful at finding any better solutions.

What if we experiment with a different construction heuristic instead? Here are the results of **DOUBLE_FLIP** improvement heuristic after constructing with Moraga's rule:

$$h_j = \frac{c_j}{\sum_{i=1}^{m} \frac{a_{ij}}{b_i - g_i}}$$

DOUBLE_FLIP from MORAGA'S RULE

Problem	Init Solution	Time, ms	Improved Solution	Time, ms
100-5-01	23984	0.04	24192	0.2
100-5-02	23970	0.1	24225	0.2
100-5-03	23317	0.1	23449	0.2
100-5-04	22884	0.1	23313	0.2
100-5-05	23826	0.1	N/A	0.09
250-10-01	58393	1	58776	1.5
250-10-02	57991	0.5	58328	1.6
250-10-03	56942	1	57710	1.2
250-10-04	60296	1	60624	1.08
250-10-05	57643	1	N/A	0.67
500-30-01	114188	12	115107	18
500-30-02	113570	7	114043	7
500-30-03	115328	5	115617	4
500-30-04	113592	7	114598	19
500-30-05	114730	10	115326	17

As we can see, the different construction heuristic manages to find better solution on one subset of problems while losing in quality on the other. The number of found solutions have not increased. **FLIP** and **SWAP** still perform bad with this problem.

We also tried normalizing, calculating dynamically, taking into account original capacity:

$$h_j = \frac{c_j}{\sum_{i=1}^{m} \frac{b_i a_{ij}}{b_i - g_i}}$$

All results are presented in the following table[1]:

Problem	Initial	FLIP	SWAP	DOUBLE_FLIP	DOUBLE_FLIP - Moraga	DOUBLE_FLIP - Norm
100-5-01	24049	N/A	N/A	24164	24192	N/A
100-5-02	23970	N/A	N/A	24225	24225	24225
100-5-03	23317	N/A	N/A	N/A	23449	23449
100-5-04	22979	N/A	N/A	23294	23313	23313
100-5-05	23826	N/A	N/A	N/A	N/A	N/A
250-10-01	58284	N/A	N/A	58711	58776	58768
250-10-02	58293	N/A	N/A	58508	58328	58581
250-10-03	57390	N/A	N/A	57710	57710	57669
250-10-04	60467	N/A	N/A	60484	60624	60564
250-10-05	57325	N/A	N/A	57667	N/A	57712
500-30-01	114839	N/A	N/A	115250	115107	114950
500-30-02	113662	N/A	N/A	114158	114043	114043
500-30-03	115805	N/A	N/A	115821	115617	115197
500-30-04	114203	N/A	N/A	114543	114598	114563
500-30-05	115471	N/A	N/A	115773	115326	115326

In our experiments, only the **DOUBLE_FLIP** neighbourhood operator proved effective in finding improved solutions. **FLIP** and **SWAP** heuristics do not manage to find any better solution. We also experimented with a couple of other construction heuristics. The final solutions seem to be better for smaller problems if we use them. Our construction heuristic helps finding better improved solutions for bigger problems.
Further development may be to depart from our choice of always staying in the feasible region – perturbing the solution, even if it goes infeasible, may prove worthy to further ascending to a region of an abundance of high-quality solutions.

References

[1] S. Lapitskaya V. Klyuev M. Drozd. “Assignment 1: Construction heuristic.” In: *LOG 734* (2021).

[1] Initial solution in the table was a foundation only for solutions in first three heuristics. Solutions yielded by Moraga and the dynamic rule are different