Mathematics 2 - Part 2 - Nelder Mead, Local search

Luka Žontar

Introduction

This is the report of the second homework of second part of the Mathematics 2 course, where we show several different optimization methods, more specifically the Nelder Mead method and the local search method. Local search method is implemented on a sample problem, maximal weight matching.

All the test executions that can be used to reproduce results that are shown in continuation can be found in notebook *Homework - Nelder Mead, Black Box, Local Search.ipynb*.

A documented Nelder Mead method implementation in Python can be found in *NelderMeadOptimizer.py* file.

A documented implementation of local search of the maximal weight matching problem in Python can be found in *MaximalWeightMatchingLocalSearch.py* file.

Nelder Mead method

In Table 1 results from Nelder Mead method execution on three different cost functions can be found. For all the results the execution was stopped if cost function values of the sample points were close enough or the maximum number of iterations were reached. Cost function values of the sample points are interpreted as close enough if the difference between best and worst point is smaller than $\varepsilon = 1e-6$. The maximum number of iterations was set to 10000 for all cases. Below you will find the starting points of the aforementioned cost functions:

- **Function 1**: $(-\frac{1}{6}, -\frac{11}{48}, \frac{1}{6})$
- **Function 2**: $(-1, 1.2, 1.2)$
- **Function 3**: $(4.5, 4.5)$

For each of the cost functions, we tested different diameters (i.e. diameters of sizes 1, 3 or 5) that are used to generate the initial tetrahedron. However, as can be seen in the Table 1 Nelder Mead method converges for all these diameters and the differences between true and estimated y are not significantly distinct. Note that time and number of steps are also quite similar.

As for comparison with HW2/5a,b,c, the Nelder Mead method outperforms all my implementations in previous homework, which might perform better than the Nelder Mead method.

	Diameter	Error (y)	Time (s)	Steps
Function 1	1	4.41e-7	1.6e-3	44
	3	4.05e-7	1.9e-3	55
	5	3.34e-7	2.3e-3	67
Function 2	1	3.01e-7	6.8e-3	192
	3	7.5e-7	7.4e-3	209
	5	2.14e-7	5.1e-3	146
Function 3	1	1.05-6	1.1e-3	34
	3	7.97e-8	1.2e-3	36
	5	3.72e-7	1.8e-3	50

Table 1. Nelder Mead method results: For all the tested functions optimized with different diameters when building the initial tetrahedron, we provide results for Nelder Mead method optimization. Results include the number of steps and time spent for execution. *Error (y)* represents the difference between the true minimum y and the one estimated with Nelder Mead which tells us whether or not the method converged to the correct minimum.

Black box optimization

In black box optimization assignment, we try to optimize functions for which we don't know how they look like or work. The black box accepts the student ID, function ID (there are three possible functions) and 3 sample points. We try to optimize all three functions providing the student ID, function ID and the initial points that are set to 0 in our test. Diameter of tetrahedron is set to 1.

As can be seen in table 2, the results of a commercial LP solver are as good as the results from our implementation of Nelder Mead method. We can safely say that performance of both methods does not differ significantly. In all cases both approaches converge.

Local search study

As mentioned above, the local search study in this assignment implements the optimization of the maximal weight matching problem. Local search at each point of iterative improvement

		Nelder Mead		L-BFGS-B
	$\lvert y_{LP} - y_{NM} \rvert$	Time (s)	Steps	Time (s)
Func. 1	6.04e-7	1550.81	76	304.66
Func. 2	3.79e-7	636.77	35	51.85
Func. 3	6.79e-7	732.26	38	51.93

Table 2. Black box optimization results: Table shows the difference of the performance of a commercial LP solver and our implementation of Nelder Mead method. It also shows the complexity of the methods by showing the time needed to optimize a function.

tries to find a better solution near the current position, that is, in the neighborhood. First, we explain some important terms. The implementation was tested on G_{20} 2D grid graph where in local search in every iteration the matching can potentially be improved and replaced with the k-adjacent matching. We tested $k = 1, 2, 3$

Maximal, perfect matching and greedy approach

Given a graph G with weights w, a matching M is maximal if $w(M)$ is **maximal** for G, where $w(M)$ is the sum of edge weights in M. **Perfect matching** is matching where the matching covers all the vertices in G. Note that graph can have a perfect matching if and only if the number of vertices is even. A greedy approach of finding the maximal weight matching might be to start with an empty graph and add an edge with minimal weight. Then at each point going forward we would select a point with maximal weight that persists the matching constraint of vertex incidence. We proceed with this process until we cannot add any other edge without breaking the matching constraint.

Jump moves can enhance the performance of local search by allowing a (possibly) bad move at the start of the execution. A sensible jump move may include promoting to add an edge with highest possible weight at each iterative improvement. That is, the choice of edges that will be removed or added in the current iteration depends on their weights. We include the results of this logic in the Table 3. However, since originally we sampled weights uniformly at random from interval $[1, 2]$, we transform the weights back to $[0, 1]$ interval. Using equations below, we calculate probability weights that are used in sampling. In Table 3 we see that the proposed improvement only improves 1 of the results, where $k = 2$ and that the execution time is significantly increased. Results were generated by making 10000 local search steps.

In Equation 1, w_i is the probability weight of picking a sample edge, $E(V)$ is the edge set of graph G with vertex set V and w is the weight vector of those edges. This equation represents probability weight of sampling the i-th edge to add to the matching M. We norm it by subtracting 1 from all the numbers so that the differences between weights are more significant.

$$w_i = \frac{w(e_i) - 1}{\sum_{e \in E(V)} (w(e) - 1)} \tag{1}$$

	Random edge selection		Weighted random edge selection	
k	Time (s)	Sum of weights	Time (s)	Sum of weights
1	14.34	205.24	451.33	203.89
2	20.06	192.65	918.94	207.14
3	25.29	193.37	1327.93	179.14

Table 3. Local search results: Table shows execution time and sum of weights of optimal matching return by our local search solver.

In Equation 3 we calculate the probability weight of picking a sample edge to remove from the matching M. The setup is similar except for $E(V)$ being the edge set of matching M. In normalization we now subtract the weight from 2, because we want to select edges with lower weights, thus inverting the probability weight from before.

$$w_i = \frac{2 - w(e_i)}{\sum_{e \in E(V)} (2 - w(e))} \tag{2}$$

Relax Maximal Weight Matching

To understand Relax Maximal Weight Matching problem, we first define the **fractional matching**. Given edge e and vector $x \in [0, 1]^E$, x is fractional matching if for every vertex v we have $\sum_{e \in E(v)} x_e \leq 1$. In other words, we have at most one edge that uses vertex v.

In Relax Maximal Weight Matching, we are looking for the fractional matching with maximal $\sum_{e \in E(G)} x(e) w(e)$ given a graph G with weights w.

We construct Relax Maximal Weight Matching problem as a linear problem that can be solved with a commercial LP solver. Given an incidence matrix we construct:

1. **Coefficient matrix** A that is constructed from the incidence matrix of the given graph and a positive and negative identity matrices for upper and lower bounds of the system.

2. **Constraint vector** b that is constructed from vector of 1s to bound the solver to solve for given graph, vector of 1s for the upper bound and vector of 0s for the lower bound of the fractional matching.

3. **Cost vector** c that is constructed from negative weights of the given graph. Weights are negative because the original problem is trying to maximize the sum of weights and we are using a minimizer to do so.

Using a commercial LP solver we obtained the system solution for G_{20}, where $x^* = 339.29$, which is also the optimal solution for the above example. Table 3 shows that we are far from the optimal solution.

Assuming that $\sum_{i=1}^{k} \alpha_i = 1, \alpha_i >= 0$, for all i and x_i is the i-th perfect matching, a fractional matching x can be expressed as a convex combination of perfect matchings:

$$x = \alpha_1 \cdot x_1 + \alpha_2 \cdot x_2 ... + \alpha_k \cdot x_k \tag{3}$$