Heuristic Opt. Techniques - Assignment 5+6 Report

Martin Blöschl and Cem Okulmus

1 Assignment 5

1.1 Implementation

In the previous exercise, we developed a genetic algorithm. As stated in the previous report, our algorithm provided quite good results by itself. However, we wanted to improve the results by combining the genetic algorithm with local search.

Our genetic algorithm was implemented to be as simple and fast as possible. This allowed us to have a large population and compute a lot of generations. By incorporating local search, we want to improve the already very diverse solutions by reaching optimality in a certain neighbourhood.

Our Metaheuristic works in the following way: First, we execute the genetic algorithm to compute solutions. The best solution will then be used as a starting solution for local search. We can use any local search neighbourhood and step function from the previous exercises. Local search will be executed and the best solution from the genetic algorithm will be improved further. The result of the local search will be a solution that (if first improvement or best improvement is used as step function) is as least as good as the solution from our genetic algorithm and also a local optimum in a certain neighbourhood.

1.2 Parameters of the Metaheuristic

Since this approach combines a genetic algorithm (GA) with a local search (LS), the configuration space becomes more complex. This becomes more relevant when one tries to find optimal or near optimal values for given problem instances. Here is a short description of each parameter our implementation uses.

1.2.1 Generations

-i #number

This indicates how many generations the GA will simulate. While ideally this number should be very high, there is an observable point for each instance, where the GA no longer produces better results. Therefore an ideal parameter has to be high enough to fully exploit the GA but not too high to waste time.

1.2.2 Population size

-p #number

This parameter describes how many instances should be in the population throughout the GA, since our approach keeps the number of instances constant. This parameter is ideally very high, to have large “gene pool” for the GA. At the same time it increases the memory usage of the GA quite a bit. So ideally a compromise between performance and effectiveness has to be found.

1.2.3 Neighbourhood

-n $\{0, 1, 2, 3, 4\}$

This determines which neighbourhood structure is used for the local search. Compared to the parameters for the GA, this one chooses one of five separate methods of doing local search, rather than increasing or decreasing a number. Our approach uses all the neighbourhoods we have developed so far.

0. *N Node Swap*
   N nodes are swapped in the current spine order, creating a new one.
1. *N Edge Flip*
   N Edges are flipped from their current assigned page to another
2. *M N Edge Flip Node Swap*
   Combination of the two above, using two parameters
3. *Node Neighbour Swap*
   As *N Node Swap*, but restricting to swapping two adjacent nodes in the spine order.
4. *Edge Neighbour Flip*
   As *N Edge Flip*, but restricted to flipping two edges assigned to adjacent pages.

1.2.4 Step-function

-s $\{0, 1, 2\}$

This determines which step function is used for the local search to find better instances.

1. First Improvement
   This chooses the first instance in the neighbourhood of a target, which has a lower crossing count.
2. Random Step
   This produces a completely random element in the neighbourhood (irrespective of any ordering) and picks it regardless of its crossing count (Leads to random search, with results only returned if better crossing eventually found)
3. Best Improvement
   Fully examines all elements of the neighbourhood and picks the one with the lowest crossing count, which is better then the target.

1.3 Experimental setup

We tested our implementation locally on a PC with the following specification: Intel i7-4500U processor, 8 GB of RAM, Ubuntu 16.10 64 bit. Note that we have only tested our implementation with one configuration. The reason for this is that we tested different parameter configurations for

Table 1: The results (crossings) and run times (RT) in seconds using default values of the algorithm.

Instance	Best	Mean	Std. Dev	Mean RT	Std. Dev RT
1	9	9,2	0,34	14,99	1,36
2	0	0	0	21,74	1,87
3	43	50,7	2,84	78,21	2,03
4	0	0,6	0,04	54,3	1,1
5	4	7,2	1,46	69,23	1,46
6	8416290	8441085,9	34687,3	900	0
7	82565	90530,5	6546,04	369,21	10,88
8	1107609	1112839,1	19450,45	900	0
9	1582522	1596328,5	90587,87	900	0
10	212878	215352,8	2154,52	900	0

the assignment 6 anyway. The results of this section are meant to be ”reasonable default values” that we can compare to the results from the last assignments. The configuration is the following:

Generations: 150
Population Size: 5000
Neighbourhood: *M N Edge Flip Node Swap* with **M:** 2, **N:** 2
Step-function: *First Improvement*

Figure 1: Our “untuned” configuration

1.4 Results + Runtimes

The results from the algorithm using the configuration in Figure 1 are displayed in Table 1.

1.5 Comparison

Looking at our results in Table 1 and the results from the last exercise, we can see that the number of crossings are very similar. We also see a slight increase of runtime, since our genetic algorithm was extended with a local search.

We will see if we can further improve our results by optimizing our parameters.

2 Assignment 6

For this we were asked to familiarize ourselves with the *irace* package in R. This is an implementation of the “iterated race” method introduced in the lecture to find good configurations for (meta-)heuristics, where a large number of parameters have to adapted to certain set of training instances.

We chose for these the ten instances (automatic-1 to automatic-10) that we used throughout the course. We also had to provide irace with a simple script that would execute our heuristic for a given instance and sets of parameters. This was all we needed to to do, as the process is fairly automated.

Table 2: The results of the four elite configurations.

I	A (Best)	A (Mean)	B (Best)	B (Mean)	C (Best)	C (Mean)	D (Mean)	D (Mean)
1	9	9,2	9	9.3	9	9.8	9	9
2	0	0	0	0	0	0.2	0	0.4
3	38	45.6	26	37.8	35	41.7	34	40
4	0	0.2	0	0	0	1.2	0	0.4
5	3	5.2	2	4.9	3	5.7	4	6
6	8466454	8472111.4	8481672	8497624.7	8464659	8476492.4	8459663	7633459
7	85074	97363.2	95260	109261.9	84718	97434	89119	95114.7
8	1107939	1115966.8	1135812	1140314	1111452	1115030.7	1124319	1129448.9
9	1589458	1601760.3	1621229	1633475	1590908	1600572.2	1617767	1626279.9
10	213863	215938.8	216214	217755.5	212445	215925.6	215868	217462.2

It produced, after 180 tests, four elite configurations which were found to perform the best by irace. These we named A, B, C and D and they are shown in the Figures below.

Generations: 282
Population Size: 5584
Neighbourhood: *M N Edge Flip Node Swap* with **M:** 4, **N:** 4
Step-function: *First Improvement*

Figure 2: Elite configuration A

Generations: 752
Population Size: 9886
Neighbourhood: *N Edge Flip* with **N:** 2
Step-function: *Random Step*

Figure 3: Elite configuration B

Generations: 896
Population Size: 7081
Neighbourhood: *M N Edge Flip Node Swap* with **M:** 2, **N:** 2
Step-function: *First Improvement*

Figure 4: Elite configuration C

Generations: 881
Population Size: 5516
Neighbourhood: *N Edge Flip* with **M:** 2, **N:** 2
Step-function: *Random Step*

Figure 5: Elite configuration D

3 Evaluation

To evaluate them, we first run for each configuration and each instance ten runs. From these we took the averages and as instructed run a “Wilcox Signed Rank Test”, which is included in the “MASS” package of R. The results are ....

4 Conclusion

Since none of the elite configurations proved to be statistically different, in the sense of being clearly from a different distribution, we conclude that our default configuration was already rather good, or vice-versa that *irace* failed to produce configurations which were clearly superior. Perhaps this could be fixed by going beyond the 180 tests, though these already proved to be rather time consuming on our local machine.