Programming Assignment 4

Heuristic Optimization Techniques

David Molnar, 1326891
Daniel Füvesi, 1326576
Gruppe 9

December 11, 2016

1 Population-based Methods

1.1 Genetic Algorithm

For this assignment we decided to implement a genetic algorithm. First the core algorithm and the operatos are described, followed by applied methods to improve performance and overall runtime. Previous implementations have been used, especially for generating random solutions with the construction heuristics. Below the pseudocode of the algorithm solely.

Algorithm 1 Genetic Algorithm for KPMP

```
1: Global variables
2:     population_size, elitism_size, mutation_rate, crossover_rate
3: end Global variables
Input: KPMP Instance
Output: KPMPSolution that is at least as good as the input
4: function IMPROVE( )
5:     population ← generatePopulation(size, instance)
6:     evaluatePopulation(population)
7:     iterations = 0
8:     while iterations < iterationsLimit && notTimedOut do
9:         nextGeneration = new Population
10:        addElitesToNextGeneration(elitism_size)
11:        while new generation size not full do
12:            mother, father = selection()                    ▷ select 2 parents
13:            children = crossover(mother, father, crossover_rate)   ▷ children with probability
14:            children = mutate(children, mutation_rate)      ▷ mutate with probability
15:            addChildrenToNewGeneration(children)
16:        population = nextGeneration
17:        evaluatePopulation(population)
       return best solution
```

1.2 Representations

Consistent to other assignments, a `KPMPSolution` contains a spine order, an edge partitioning and the number of pages. Additionally, for the genetic part there are so-called `Individual` and `Population` objects. An `Individual` consists of `genes` which is a `KPMPSolution` (the genes are the spine order and the edges), the number of crossings as the evaluation function and a fitness value. The fitness value of an individual solution is calculated by a simple fitness function that takes the maximum possible crossings and subtracts the current number of crossings of the individual. Here, the maximum possible crossings denote the calculated crossings before applying the optimization thus acting as an upper-bound. The fitness function is efficient (only a subtraction needed) and fulfills the conditions (> 0, the higher the better) at the same time. `Population` encapsulates a list of `Individuals`, the population size and the total fitness of it.

1.3 Selection

Just like in other parts of this assignment, many experiments have been concluded to find a suitable selection procedure. First we implemented a simple roulette wheel selection based on the fitness values of the candidates, but it didn't perform well. Therefore, we tried a roulette wheel with linear ranking. For comparison, the tournament selection has been implemented as well and despite its simplicity it provided the best results on the long run. The algorithm invokes this selection operator twice to obtain a mother and a father individual. Note that it is possible that the same individual is returned twice such that the use of terms "father" and "mother" would not fit - they are simply used to better explain the algorithm with descriptive words.

1.4 Recombination

Having both parent solutions selected, a recombination takes place with a random probability. If the random number is greater than the required probability, the parents skip the recombination and proceed to the mutation phase. In case of a recombination, 2 child individuals are generated using 1-point-crossover for both the spine order and the edge partitioning. 2 random numbers are picked for the crossover line. In case of the spine order, all vertices up to the line are copied from the mother to the first child. Visited vertices are removed from a father's copy so that only those vertices get added from father's side that have not been already added. The edge partitionings are sorted according to their `a` and `b` values since essentially the order of the edges does not have any influence. The sorted orted however makes it easier to copy two distinct subsets of edges divided by the random line from both parents. The second child is put together in the same way with flipped parent parts. Both children get evaluated immediately.

1.5 Mutation

In this phase a mutation happens - again - with a random probability for each of the 2 children. The inner mutation itself is randomized as well, meaning that there can be a single or multiple mutations at the same time. If a mutation needs to happen, a single edge is always moved to another random page. With another random probability, a second edge will get a new pagenumber as well. Furthermore, with a third random probability a node swapping takes place. This is because we could not manage to calculate only the difference in crossings when swapping 2 vertices. If the

algorithm swaps vertices on every mutation the performance loss outweighs the gain for better solutions. Consequently, the much faster edge-move is always performed, node swapping only occasionally.

2 Performance Improvements

Despite our big expectations towards the genetic algorithm the performance turned out the be poor in comparison to the GVNS from the last assignment. This is rather true for instances 6-10 (1-5 are ok, best solutions until now are found). Therefore we spent a good amount of time during development to increase the performance wherever we can. First, we introduced a a `needsEvaluation` flag in an `Individual`. This flag indicates whether the solution's crossing numbers (and thus fitness) need to be re-calculated or not. In case only edges are moved in a mutation, there is no need for a complete scan for crossings. This flag saves a lot of unnecessary calculation. We noticed that many times the children are significantly worse than their parents (because the recombination is solely random). For this, we added a `family_elitism_rate`: in case of a random probability the 2 best individuals of the "family" are passed onto the next generation. This way it is possible that a good mother and a good child or only the parents are returned.

2.1 Multithreaded Construction

Generating the randomized population takes up a lot of time in the larger instances, especially when the size of the population is set to high. In order to reduce execution time, we implemented a multithreaded construction process. The program generates as many threads as virtual cores are available (4/8). The population size is distributed evenly across the threads such that each thread generates the same amount of solutions. Having 8 threads (in our setups at least) can boost the construction significantly.

2.2 Multithreaded GA

The above method has also been applied to the genetic algorithm, because we simply could not stand having this poor performance. First the main algorithm adds the elites to the next generation, then it dispatches the threads using the same method from above (even distribution). A so-called `Slave` is responsible for selection, recombination and mutation as well. Each `Slave` receives only a part of the big population thus avoiding writing conflicts. The additional advantage beside the performance gain is that there is a distributed tournament selection so that the selection pressure can be set higher (there is a tournament in each of the 8 distinct chunks).

3 Experimental Setup

The algorithm has been implemented in Java without external libraries; tested on 2 Windows 10 PCs, both having 8GB RAM and a CPU of Intel Core i7 870 (4 cores, 2.93GHz) and Intel Core i7 3630QM (4 cores, 2.4GHz) respectively. All instances received an average CPU time limit of 20 minutes. This time varies depending on the size of the instance. In our opinion the construction time should not be considered, therefore we increase the limit with the time that has been spent on

generating the initial population. If the running time exceeds the limit (checked on each iteration), the current best solution is returned.

4 Results

As already mentioned, we weren't able to reach a good performance. We assume this is due to the randomized recombination (there is no deterministic method that aims to create a better solution from 2 parents, for that matter child solutions are almost always worse) and due to the lousy evaluation when mutating the spine order. Additionally, there are a lot of parameters that influence one another and they provide a big challenge to optimize. Beside the population and the elite size, we have probability parameters (each btw. 0 and 1) for mutation, crossover, family elitism, node swap and double edge move. Theoretically these 7 parameters should be optimized for every instance.

4.1 Results using Randomized Construction Heuristic

Instance	Avg. Crossings B4 GA	Best Crossings	Avg. Crossings	Time	Runs
automatic-1	18	9	9	520 ms	15
automatic-2	6	0	0	675 ms	15
automatic-3	285	28	40	118'487 ms	15
automatic-4	3	0	0	10856 ms	15
automatic-5	24	0	0.9	9166 ms	15
automatic-6	5'751'426	5'745'556	5'750'425	full time	5
automatic-7	6'710	4'976	5'985	full time	5
automatic-8	734'630	686'880	695'613	full time	5
automatic-9	147'709	131'267	139'318	full time	5
automatic-10	69'754	57'290	59'747	full time	5

4.2 Best paramter models

Instance	population size	elite size	mutation rate	crossover rate	family elitism rate	ns rate
automatic-1	120	8	0.7	0.7	0.6	0.2
automatic-2	80	8	0.8	0.7	0.6	0.2
automatic-3	4000	8	0.5	0.7	0.6	0.4
automatic-4	80	8	0.6	0.7	0.6	0.7
automatic-5	400	8	0.9	0.7	0.8	0.2
automatic-6	8	0	0.5	0.7	0.6	0.2
automatic-7	56	8	0.5	0.7	0.6	0.2
automatic-8	16	8	0.5	0.7	0.6	0.2
automatic-9	16	8	0.5	0.7	0.6	0.2
automatic-10	16	8	0.5	0.7	0.6	0.2