Programming Assignment 5

Heuristic Optimization Techniques

David Molnar, 1326891
Daniel Füvesi, 1326576
Gruppe 9

January 15, 2017

1 Hybrid Metaheuristic

1.1 Memetic Algorithm

In this assignment we implemented a combination of the previous genetic algorithm and a local search. The implementation consists of the steps similar to the presented ones in the lecture. After the population is generated, we perform a local search on the individuals. Only after that the population is evaluated and passed onto the genetic part of the algorithm. Here, no changes has been made - depending on the many probability parameters individuals are selected, crossed over and mutated. At the end of each iteration we perform a local search on the new generation as well.

Algorithm 1 Memetic Algorithm for KPMP

Global variables
population_size, *elitism_size*, *mutation_rate*, *crossover_rate*, *loca_search_rate*, *node_swap_rate*, $family\_elitism\_rate$
end Global variables
Input: KPMP Instance, Type of LocalSearch, Stepfunction
Output: KPMPSolution that is at least as good as the input
function IMPROVE( )
    population ←generatePopulation(*size*, *instance*)
    performLocalSearch(*population*)
    evaluatePopulation(*population*)
    while ... **do**
        // genetic part, obtain *nextGeneration*
        performLocalSearch(*nextGeneration*)
        evaluatePopulation(*nextGeneration*)
    return *best solution*

1.2 Local search

For the local search we decided to use an already existing one. For performance reasons we went with the *Single-Edge-Move* (S-E-M) neighbourhood along with the best-improvement stepfunction. Reminder: the S-E-M neighbourhood consists of all solutions where 2 solutions differ in the page number of an edge (i.e. an edge is moved to another page). In order to maintain a fair balance between diversity and accuracy, not necessarily every individual gets locally optimized. This means that in the *performLocalSearch()* method an individual is only improved by the S-E-M if a random number is less than **0.8** (adjustable parameter).

2 Performance Improvement

Beside the multithreaded construction of initial solutions and the genetic part we also implemented the local search using multithreading. Since individuals do not affect each other, it is a suitable case for parallelization. The population gets divided by the number of (virtual) cores so that every thread cares for a portion doing its local search. The crossing numbers are incrementally calculated during the local search so that a subsequent invocation of *evaluatePopulation()* in the genetic algorithm does not re-compute crossings unnecessarily.

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 resulting in an appr. 15 min time limit within the memetic algorithm. If the running time exceeds the limit (checked on each iteration), the current best solution is returned.

4 Results

We already realized in the last assignment that the population size (= diversity) is a huge trade-off and influences the result especially in our case having only an 8-core CPU. The goal of our memetic algorithm would be to take the best from both worlds: having a wide range of possible solutions that are being optimized on their own "hill". Light instances, where hundreds or even thousands of initial solutions can be quickly generated perfectly exploit our intended purpose. Unfortunately, we could not reach a significant improvement in the results for instances 6, 8, 9 and 10. A second reason for that (beside diversity-accuracy paradigm) is the unefficient way our algorithm handles node swaps. Swapping two nodes has a greater (positive or negative) impact on a solution than a relocationing of an edge. This means that there should be almost as many node swaps as edge moves. However, due to our inefficient calculation of node changes we are bound to restrict the occurrence of these node swaps.

4.1 Results using Randomized Construction Heuristic

Instance	Avg. Crossings B4 GA	Best Crossings	Avg. Crossings	Avg. Time	Runs
automatic-1	17	9	9	190 ms	15
automatic-2	7	0	0	799 ms	15
automatic-3	280	20	24	110'114 ms	15
automatic-4	3	0	0	4'631 ms	15
automatic-5	30	0	0.4	14'548 ms	15
automatic-6	5'749'611	5'665'556	5'700'145	full time	5
automatic-7	7'458	5'545	5'985	full time	5
automatic-8	744'648	684'987	694'012	full time	5
automatic-9	140'215	129'487	139'121	full time	5
automatic-10	69'245	52'265	58'879	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	0.5
automatic-2	80	8	0.8	0.7	0.6	0.5
automatic-3	2400	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	64	8	0.5	0.7	0.5	0.8
automatic-8	16	8	0.5	0.7	0.4	0.7
automatic-9	16	8	0.5	0.7	0.4	0.7
automatic-10	16	8	0.5	0.7	0.4	0.2