Programming Assignment 1

Heuristic Optimization Techniques

David Molnar, 1326891
Daniel Füvesi, 1326576
Gruppe 9

October 23, 2016

1 Construction Heuristic

1.1 Deterministic Construction Heuristic

Our construction heuristic consists of 2 separate steps, the spine odering and the edge partitioning. For the spine ordering we implemented a simple depth-first search algorithm. To satisfy determinism, the middle node of the initial spine is picked as the root of the new ordering. Neighbours with the lowest indices are prioritised and added to the spine. This new ordering is then used for the edge partitioning, namely the *CFL (Conflicting Edge Distribution)*. Instead of simply iterating through the edges, we first sort them by the number of conflicts they induce in an ascending order and remove the ones with 0 conflicts. The idea is, that we want to move edges that cause the most crossings to another page as soon as possible. We iterate through this set and each edge gets placed on a page where it leads to the least crossings amongst all pages.

Algorithm 1 KPMP

Input: KPMPM Problem Instance
Output: KPMPSolution

1: **function** SOLVE( )
2: *spine order* ← *DFS Spine Ordering*(adjacency list,⌊number of vertices/2⌋);
3: *edge partitioning* ← *CFL Edge Partitioning*(*calculated spine order*, edges);
return solution(*spine order, edge partitoning*);

4: **function** DFS SPINE ORDERING(adjacency list, root vertex index)
return *spine order* ← DFS(*spine order*, vertex index);

```
5: function CFL Edge Partitioning(spine order, edge list)
6:     organizedEdges ← removeEdgesWithZeroCrossings(edge list);
7:     organizedEdges ← sortByCrossings(edge list);
8:     while not iterated through all edges in edges do
9:         edge ← next edge in organizedEdges
10:        move edge in edge list to page where it leads to the least crossings amongst all pages
       return edge list
```

A valid solution is encapsuled in an object storing an arraylist of integers as the spine order, an arraylist of *PageEntry* as the edge distribution and a single integer for the number of pages. A so-called *SolutionChecker* is then used to calculate the crossings at the end.

In our opinion it makes more sense to separate the two steps for they can have completely independent implementations and they can be optimized perfectly because of the fine granularity. While the spine-ordering might be enhanced by a local search, the edge partitioning could implement a natural selection algorithm without direct influence on one and each other.

1.2 Randomized/Multi-Solution Construction Heuristic

As far as randomization goes we implemented it in both steps. Our approach is to generate k different solutions, evaluate them and return the best one. In case of the first step we let our *SpineOrderHeuristic* to generate k different spine orders where the DFS algorithm is used from above with the modification of picking a random root node. In order to avoid duplicates, only unique k spine orders are generated. For each of these spine orders the edge distribution is executed like in the deterministic variant. However edges are not selected from a predefined (sorted & reduced) list, instead they are picked randomly. Plus we let the edge partitioning loop to run multiple times in succession to further improve the result. Since we've adapted/extended our deterministic approach, the algorithm does not degenerate into random search. After conducting multiple test-runs the randomized version turned out to be performing better in the majority of cases. Despite the overall good results, the huge number of iterations (due to multiple solutions) act as a significant trade-off regarding execution time (and performance). This could be compensated by reducing the number of generated solutions and by applying local search, simulated annealing and/or evolutionary algorithms to improve the few existing ones.

2 Experimental Setup

The heuristics were tested on a Windows 10 PC with Intel Core i7 870 (4 cores, 2.93GHz) and 8GB RAM, implemented in Java without external libraries. All instances received a CPU time limit of 15 minutes. If the running time exceeds the limit (checked on each iteration), the current best solution is returned. The number of spine order solutions and iterations in the partitioning were allocated dynamically depending on the "weight" of the instance. Instances 1-5 always generated as many different spine order solutions as the number of vertices they had. Instance 6 only randomized 2 spine orders, while the rest (7-10) did 5 different ones. For fairness, we allowed the deterministic algorithm to make multiple iterations through the edges as well.

3 Results

3.1 Results of Deterministic Construction Heuristic

Instance	Crossings	Time
automatic-1	17	20 ms
automatic-2	10	33 ms
automatic-3	78	78 ms
automatic-4	6	36 ms
automatic-5	20	40 ms
automatic-6	3'708'167	450'273 ms (7.50455 min)
automatic-7	8806	2523 ms
automatic-8	208'775	297'682 ms (4.961 min)
automatic-9	31817	318'593 ms (5.309 min)
automatic-10	17057	72813 ms (1.213 min)

3.2 Results of Randomized Construction Heuristic

Instance	Best Cr.	Avg Cr.	Std. Dev. Cr.	Avg. Time	Std. Dev. Time	Runs
automatic-1	9	11.2	2.52	209.6 ms	70.80 ms	15
automatic-2	3	3.8	1.13	282.8 ms	12.6 ms	15
automatic-3	44	50.4	3.53	2678.83 ms	76.86 ms	15
automatic-4	2	3.9	2.38	1122.16 ms	63.26 ms	15
automatic-5	8	11	4.16	1695.83 ms	87.69 ms	15
automatic-6	2'021'124	2'859'051	371980.26	699130.16 ms (11.65 min)	24889.06 ms	15
automatic-7	4993	6162.5	1015.28	16667.5 ms	139.7 ms	15
automatic-8	198'957	203'191.87	4353.03	785092.5 ms (13.08 min)	11352.10 ms	15
automatic-9	31350	37752.77	3165.16	808543.66 ms (13.47 min)	12397.69 ms	15
automatic-10	15220	15873.27	321.92	491439.83 ms (8.19 min)	3254.12 ms	15