# Programming Assignment 1

Heuristic Optimization Techniques

David Molnar, 1326891
Daniel Füvesi, 1326576
Gruppe 9

October 22, 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. With the help of an efficient calculation for the result of a "move", the whole algorithm finishes under a minute.

**Algorithm 1** KPMP

```
function SOLVE(problem instance)
    calcuated spine order ← DFS Spine Ordering(adjacency list, number of vertices / 2)
    calcuated edge partitioning ← CFL Edge Partitioning(calculated spine order, edges)
     return solution(calculated spine order, calculated edge partitoning)

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

function CFL EDGE PARTITIONING(spine order, edge list)
    edges ← sortByCrossings(edge list)
    new edge list ← empty list

    while edges not empty do
     new edge list ← new edge list + edge
     move edge to page where it leads to the least crossings amongst all pages
     return new 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 $k$ spine orders where the DFS algorithm is used from above with the modification of picking a random root node (*Note:* Duplicate spine orders are possible in our current implementation. Ideally, the algorithm should return $k$ permutations - there is always room for improvement.). 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) list, instead they are picked randomly. 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

# 3 Results

## 3.1 Results of deterministic construction heuristic

## 3.2 Results of randomized construction heuristic