Programming Assignment 2

Heuristic Optimization Techniques

David Molnar, 1326891
Daniel Füvesi, 1326576
Gruppe 9

November 6, 2016

1 Local Search

1.1 Description

For this assignment we extended our current program with a local search strategy. Due to a solid software architecture that we have been carefully designing since the beginning, we only had to define a new interface and inject the proper implementation. An abstract class encapsulates the basic procedure of the local search:

Algorithm 1 KPMP Local Search

Input: Constructed KPMPSolution
Output: KPMPSolution that is at least as good as the input

```
1: function improveSolution(initialSolution, stepFunction)
2:     best solution ← initialSolution
3:     globalStepfunction ← stepFunction
       return performLocalSearch();

4: function performLocalSearch( )
5:     do
6:         neighbourSolution ← nextNeighbour(globalStepfunction)
7:         if neighbourSolution better than best solution then
8:             best solution ← neighbourSolution
9:     while isStoppingCriteriaSatisfied(neighbourSolution) ≠ true
10: return best solution
```

Subclasses only need to implement the *nextNeighbour()*, *randomNextNeighbour()* methods that are invoked depending on the type of the step function and 3 stopping criterias also driven by the actual step function.

1.2 Neighbourhood Structures

We implemented the following 3 neighbourhood structures:

1. **NodeSwap** This neighbourhood consists of all solutions, where 2 vertices of the spine order are swapped.
    - Objective function: Unfortunately all the crossings need to be calculated for each solution which obviously results in an overhead.
2. **Single-Edge-Move** This neighbourhood consists of all solutions, where a **single** edge is put on another page.
    - Objective function: Incremental evaluation; only calculate the difference of the crossings that this edge creates on both pages.
3. **Node-Edge-Move** This neighbourhood consists of all solutions, where **all** edges of a node are put on another page.
    - Objective function: Incremental evaluation; only calculate the difference of the crossings that those edges create on both pages.

1.3 Step functions

- **BestImprovement**: enumerate all the solutions in the neighbourhood and return the best
- **FirstImprovement**: return the first solution that is better than the original one
- **Random**: generate random solutions until a certain limit (time or number of iterations) is exceeded

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 14 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. In case of the random step function, the number of max itarations (besides the time limit) was always the number of edges multiplied by the number of vertices.

3 Results

First of all it needs to be said that due to the high number of combinations to test we we are not content with the amount of testing we have conducted. From the tables below it is quite obvious that the Single-Edge-Move neighbourhood structure returns the best solutions. It is a good structure and we could use incremental evaluation thus having a good overall performance. In most cases random initial solutions yield better end results, especially when either the instance is small enough for best improvement step function **or** the random step function is used. As far as the number of itaration goes, we could not set up a pattern when local optima are reached. Generally, a good number of iterations were needed to reach good (or best) solutions. For simplicity we only present the best solutions, otherwise the tables would be too big.

Abbreviations: (neighbourhood_structure, step_function)

- **NS** ... NodeSwap
- **S-E-M** ... Single-Edge-Move
- **N-E-M** ... Node-Edge-Move
- **b** ... best step function
- **f** ... first step function
- **r** ... random step function

3.1 Results with Deterministic Construction Heuristic

Instance	Best Crossings	Time
automatic-1	13 (S-E-M, **b**)	10 ms
automatic-2	8 (NS, **b**)	77 ms
automatic-3	66 (S-E-M, **b**)	44 ms
automatic-4	3 (NS, **b**)	250 ms
automatic-5	14 (S-E-M, **b**)	12 ms
automatic-6	5'015'298 (S-E-M, **r**)	285'736 ms (4.76 min)
automatic-7	8860 (N-E-M, **b**)	11564 ms
automatic-8	382'945 (S-E-M, **r**)	68796 ms
automatic-9	102'859 (S-E-M, **r**)	71616 ms
automatic-10	34392 (S-E-M, **r**)	22708 ms

3.2 Results with Randomized Construction Heuristic

Instance	Best Crossings	Avg. Time	Runs
automatic-1	12 (NS, **b**)	3 ms	5
automatic-2	4 (S-E-M, **b**)	35 ms	5
automatic-3	72 (S-E-M, **b**)	190 ms	5
automatic-4	8 (S-E-M, **b**)	50 ms	5
automatic-5	10 (N-E-M, **b**)	66 ms	5
automatic-6	4'199'545 (S-E-M, **r**)	271'586 ms	5
automatic-7	8738 (S-E-M, **b**)	166137 ms	5
automatic-8	292'692 (S-E-M, **r**)	68870 ms	5
automatic-9	117'712 (S-E-M, **r**)	69546 ms	5
automatic-10	26633 (S-E-M, **r**)	24711 ms	5