Heuristic Optimization Techniques

WS 2017 Assignment I Report

Peter Rjabcsenko & Roland Pajuste (Group 3)

**Description of the implemented algorithms:**

Greedy Construction:

* initialize the spine order in ascending vertex order (i.e. 1,2,3…)
* traverse edges in a particular order and check to which page it is the cheapest to add the next edge (which page assignment would result in the smallest increase of *f(x)*) and add it (use incremental evaluation)

Randomized Construction:

* initialize the spine order in ascending vertex order and then shuffle it, resulting in a random spine order
* pick a random edge and assign it to a random page (calculate *f(x)* increase on that page for incremental evaluation)

Local Search:

* **one-edge-neighbourhood** (1 edge is assigned to a different page). For all three step functions we use incremental evaluation in the following form: when an edge is selected for re-assigning, we calculate the number of crossings that the edge was involved in on its original page and the number of crossings it would create on its new page, we add the difference to the current *f(x)*
  + - **random-step-function**: pick a random edge and assign it to a random page, if this results in a lower *f(x)*, keep this solution (repeat this step # edge times)
    - **first-improvement-step-function**: traverse edges in a particular order and check if assigning an edge to a page decreases *f(x)*, return the first solution with a smaller *f(x)*
    - **best-improvement-step-function**: we use concurrency for this step. Each thread checks a portion of edges (rounded down), where *n* is the number of vertices
* **one-vertex-neighbourhood**(1 vertex is moved to a different position in the spine order). We do not use incremental evaluation for this neighbourhood, which (obviously) results in frustratingly high runtimes
  + - **random-step-function**: move a random vertex to a random position in the spine order (repeat this step # vertex times)
    - **first-improvement-step-function**: traverse vertices in the spine order and check if moving them to a different position lowers *f(x)*, accept first better solution
    - **best-improvement-step-function**: we use concurrency in this step, each thread is responsible for one vertex in the spine order and checks which new position for this vertex would lead to the lowest *f(x)*

VND:

* uses one-edge-neighbourhood and one-vertex-neighbourhood (in this order) with best-improvement step-function

GVNS

* uses VND with randomly generated initial solutions by shuffling the spine order of the current best solution (only uses 1 neighbourhood structure for “shaking” and repeats it for # vertex times)

**Experimental setup:**

* machine used: Lenovo laptop

processor: 2.60GHz Intel Core i5-4210M

RAM: 16GB

* we used an upper limit of 5 minutes runtime for each algorithm

**Best objective values and runtimes:**

Note: mean and std. dev. is calculated over 10 runs.

|  |  |  |
| --- | --- | --- |
| **Instance** | **Best objective value** | **Runtime (sec.)** |
| 01 | 21 | 0.003 |
| 02 | 92 | 0.003 |
| 03 | 147 | 0.007 |
| 04 | 119 | 0.004 |
| 05 | 114 | 0.005 |
| 06 | 1184 | 0.018 |
| 07 | 22583 | 0.14 |
| 08 | 9776 | 0.085 |
| 09 | 3770 | 0.077 |
| 10 | 7160 | 0.051 |
| 11 | 7524000 | 78.537 |
| 12 | 162968 | 1.36 |
| 13 | 1047118 | 39.626 |
| 14 | 1547842 | 47.673 |
| 15 | 201022 | 29.235 |

**Greedy Construction**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instance** | **Best objective value** | **Runtime (sec.)** | **Mean** | **Std. dev.** |
| 01 | 31 | 0.0 | 39.7 | 5.3 |
| 02 | 77 | 0.0 | 95.8 | 13.1 |
| 03 | 262 | 0.0 | 303.8 | 21.1 |
| 04 | 194 | 0.0 | 219.3 | 16.6 |
| 05 | 141 | 0.0 | 162.2 | 14.8 |
| 06 | 1418 | 0.0 | 1664.5 | 134.1 |
| 07 | 26021 | 0.062 | 27204.7 | 687.1 |
| 08 | 10309 | 0.062 | 10846.8 | 447.4 |
| 09 | 4970 | 0.031 | 5404.8 | 338 |
| 10 | 9334 | 0.015 | 9910.4 | 310.1 |
| 11 | 8080462 | 11.285 | 8085820.8 | 2617.5 |
| 12 | 173155 | 1.157 | 183266.8 | 4185.9 |
| 13 | 1106006 | 9.889 | 1128650.6 | 12818.1 |
| 14 | 1568565 | 20.16 | 1585111.6 | 17893 |
| 15 | 215714 | 12.688 | 219561.9 | 2466.7 |

**Randomized Construction**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instance** | **Best objective value** | **Runtime (sec.)** | **Mean** | **Std. dev.** | **Neighbour-hood / step-function** |
| 01 | 21 | 0.0 | ---- | ---- | edge/best |
| 02 | 8 | 0.141 | ---- | ---- | vertex/best |
| 03 | 77 | 0.031 | ---- | ---- | edge/first |
| 04 | 5 | 2.541 | ---- | ---- | vertex/first |
| 05 | 20 | 1.688 | ---- | ---- | vertex/first |
| 06 | 179 | 56.854 | ---- | ---- | vertex/best |
| 07 | 7753 | 224.279 | ---- | ---- | edge/first |
| 08 | 5425 | 21.075 | ---- | ---- | edge/best |
| 09 | 1414 | 21.791 | ---- | ---- | edge/best |
| 10 | 2316 | 17.656 | ---- | ---- | edge/best |
| 11 | 7524000 | 300.1(timeout) | ---- | ---- | edge/best |
| 12 | 149416 | 300.1(timeout) | ---- | ---- | edge/best |
| 13 | 1046789 | 300.262(timeout) | ---- | ---- | edge/best |
| 14 | 1545864 | 300.845(timeout) | ---- | ---- | edge/best |
| 15 | 200918 | 302.465(timeout) | ---- | ---- | edge/best |

**Local Search**

|  |  |  |
| --- | --- | --- |
| **Instance** | **Best objective value** | **Runtime (sec.)** |
| 01 | 21 | 0.015 |
| 02 | 3 | 0.188 |
| 03 | 51 | 0.407 |
| 04 | 8 | 1.031 |
| 05 | 12 | 1.187 |
| 06 | 227 | 60.375 |
| 07 | 8534 | 300.4 (timeout) |
| 08 | 5351 | 381.9 (timeout) |
| 09 | 1330 | 300.1(timeout) |
| 10 | 2268 | 53.412 |
| 11 | 7524000 | 300.1(timeout) |
| 12 | 149416 | 300.1(timeout) |
| 13 | 1046789 | 302.387(timeout) |
| 14 | 1545864 | 300.083(timeout) |
| 15 | 200918 | 304.192(timeout) |

**VND**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instance** | **Best objective value** | **Runtime (sec.)** | **Mean** | **Std. dev.** |
| 01 | 11 | 0.218 | ---- | ---- |
| 02 | 1 | 1.641 | ---- | ---- |
| 03 | 44 | 4.618 | ---- | ---- |
| 04 | 0 | 35.942 | ---- | ---- |
| 05 | 5 | 26.262 | ---- | ---- |
| 06 | 135 | 300.021(timeout) | ---- | ---- |
| 07 | 8534 | 300.5(timeout) | ---- | ---- |
| 08 | 5351 | 381.7(timeout) | ---- | ---- |
| 09 | 1330 | 300.1(timeout) | ---- | ---- |
| 10 | 2152 | 300.007(timeout) | ---- | ---- |
| 11 | 7524000 | 300.1(timeout) | ---- | ---- |
| 12 | 149416 | 300.1(timeout) | ---- | ---- |
| 13 | 1046789 | 300.4(timeout) | ---- | ---- |
| 14 | 1545864 | 300.1(timeout) | ---- | ---- |
| 15 | 200918 | 305.062(timeout) | ---- | ---- |

**GVNS**

**Further notes:**

* **Solution representation**:

We store the spine order in a list (similarly to the way it was provided in the parser framework) and the edges/pages in an adjacency matrix, where 2 vertices that form an edge correspond to their assigned page value in the matrix (0,1,2,etc.) while vertices that do not form an edge are represented by the value -1 in the matrix. In hindsight it might have been a better choice to store the edges in an adjacency list to avoid traversing all the edgeless vertex pairs of the matrix

* **Spine order and edge partitioning construction**:

We always construct the spine order first and then the edge partitioning for the sake of simplicity

* **Degrees of randomization**:

We use 2 construction heuristics, one of which is completely random. It might be a good idea to instead use 1 (GRASP style) construction heuristic that would allow the user to control the degree of randomness in the initial solution by manipulating the restricted candidate list. Furthermore we use a random step function for both neighbourhood structures and the GVNS “shaking” step we use generates a completely random spine order

* **Metaheuristic building blocks**:

Local Search consists of 1 of 2 neighbourhood structures, 1 of 3 step functions and predefined stopping criteria for the random step functions. VND is built from these 2 neighbourhood structures using the best improvement step function. GVNS uses VND with only 1 “shaking” neighbourhood structure (randomly shuffling the spine order) with a predefined stopping criteria

* **Covering all feasible solutions**:

Our neighbourhood structures do not cover all feasible solutions by themselves but when combined (e.g. in a VND) they can potentially generate any feasible solution

* **Incremental evaluation**:

We use incremental evaluation for the one-edge-neighbourhood by only calculating the number of crossings a potential edge would remove from a page and the number of crossings it would add to a new page and use this information to calculate the new objective value from the old one, however we do not use incremental evaluation for the one-vertex-neighbourhood and the difference in runtimes for these two neighbourhoods is tremendous

* **VND neighbourhood order and solution quality**:

the order of the neighbourhoods definitely influences the solution quality. For example

VND converges to value 227 on *instance-06* (it starts with 1-edge and continues with 1-vertex neighbourhood), while Local Search converges to the value 179 on the same instance using 1-vertex neighbourhood and best-improvement. So if VND switched its neighbourhood order, it would not get a solution worse than 179, which is already better than the final value 227 of the original version