Vienna University of Technology

192.137 Heuristic Optimization Techniques

TU Wien Algorithmics and Complexity Group

Programming Project 1

Authors:

Tobias Sloviak
01204691

Camilo Tello Fachin
12127084

Supervisors:

Prof. Dr. Günther Raidl

Dipl. Ing. Enrico Iurlano

Dipl. Ing. Laurenz Tomandl

December 1, 2023

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Contents

1 Introduction **2**

2 Questions **3**

3 Questions to Consider during developtment **4**

4 Real World application **5**

5 Deterministic Construction Heuristic **5**

6 Randomized Construction Heuristic **6**

6.1 Local improvement move operators 6

6.1.1 Fuse Operation 7

6.1.2 Swap Operation 8

6.1.3 Operation for taking our redundant edges 8

6.1.4 Shaking operation 8

6.1.5 Final thoughts 9

7 Neighborhood Structures and Step Functions **10**

8 Variable Neighborhood Descent (VNS) **11**

9 Greedy Randomized Adaptive Search Procedure (GRASP) **12**

10 General Variable Neighborhood Search (GVNS) on top of VND **13**

11 Manual Paramter Tuning **14**

12 Delta Evaluation **15**

13 Delta Evaluation **15**

14 Experimental Results **16**

15 Results and Conclusion **17**

References **17**

To Do's

- either make framework work or create new framework
- analyze locks in lecture slides and in paper:
- - Test-and-Set Lock
  - Test-and-Test-and-Set Lock
  - Ticket Lock
  - Array Lock
  - CLH Lock
  - MCS Lock
  - Hemlock (from paper..)
- in the lectureslides from the project it says:
- - Benchmark the following two under various scenarious, meaning low/high contention
  - Benchmark Throughput (probably locks-unlocks performed per time)
  - Benchmark Latency
  - Benchmark for fairness, we have to think of some number to measure fairness, maybe locks/unlocks achieved per thread

1 Introduction

2 Questions

9. Perform some manual tuning of relevant algorithmic parameters to find sensible parameter settings for the final experiments. Relevant parameters may be related to the degree of randomization, neighborhood structure sizes, probabilities for the random step function in composite neighborhood structures, the cooling schedule, the tabu list length and its variation, etc. Report the impact of a number of different settings on the solution quality of a selected meaningful subset of instances. 11. Run experiments and compare all your algorithms on the instances provided in TUWEL:

(a) deterministic and randomized construction heuristic and GRASP

(b) Use the solution of the deterministic construction heuristic to test the other implementations:

i. Local search for at least three selected (possibly composite) neighborhood structures using each of the three step functions (i.e., at least nine different algorithm variants).

ii. VND

iii. GVNS, SA, or TS

12. Write a report containing the description of your algorithms, the experimental results and what you conclude from them; see the general information document for more details.

3 Questions to Consider during developtment

- Ad 3 and 4: Does randomization and iterated application improve the generated solutions?
- What parameters do you use and which values do you chose for them?
- How does incremental evaluation work for your neighborhood structures?
- What is the time complexity to fully search one neighborhood of your neighborhood structures?
- VND: Does the order of your neighborhood structures affect the solution quality?

4 Real World application

A general s-plex application that is probably used is the identification of group or friend networks where one can make assumptions about group or friend networks. This is application is not directly derived from our programming assignment, but it is somewhat related since one still has to identify k-plexes in a undirected graph which with the amount of people using social media these days is not trivial. An application that is closer to the given s-plex programming assignment would be telecommunication networks. Where the distance between the antennas, towers or compute node could be correlated to the weight matrix and the s-plex relaxation of the clique's could be used to make crucial system more reduntant (low s-plexes) and other less crucial systems less reduntant (higher s-plexes).

5 Deterministic Construction Heuristic

Our first idea for a construction of solutions is the following: An empty graph (without edges) is always a 1-plex, since a single node without edges is a 1-plex. Removing all the edges gives a valid solution, however a very bad one. Our first idea for an algorithm that finds a good solution was to start with an empty graph and go through all the edges in A_0, check if adding them to A is legal and if so, do that. This algorithm improves the empty graph, however it can only find s-plexes of up to $|S| = s + 1$. The reason is that it will connect the first node v_1 with $\min(s, \deg_{A_0}(v_1))$ many other nodes, then the cluster of v_1 is an s-plex with size at most $s + 1$ and no more nodes can be added. Then, all the other nodes in the cluster might get connected among each other, but the s-plex will not get any bigger. Practically speaking, with this construction we got approximately $n/(s + 1)$ many clusters of size $s + 1$ which were mostly fully connected, so this construction results in small clusters.

deterministic construction heuristic where we explain how the construction heuristic is done. Maybe only do on the problem with 9 vertices and plot some results.

6 Randomized Construction Heuristic

For our random construction we randomly parse the entries of A_0 but do the same as above otherwise.

6.1 Local improvement move operators

After this point we found it difficult to think of the next move, because simply adding one edge in this state will always yield an illegal solution. In trying to understand the problem better, we wrote the adjacency matrices A_0 in files and looked at their structure. What we found was that there is a structure in most of the graphs, more precisely there are 3 types of graphs (see also figure 1):

1. locally strongly connected: 26/60 (Here, nodes that are close in numbering are also strongly connected in the graph)
2. seemingly randomly connected: 23/60
3. tentacle-like connected: 7/60 (Here, the Graphs have a semi-large, strongly connected cluster with "tentacles" reaching out. Tentacle-shaped clusters are bad for s-plexes, so we will have to seperate these "lines" into small "spheres".)

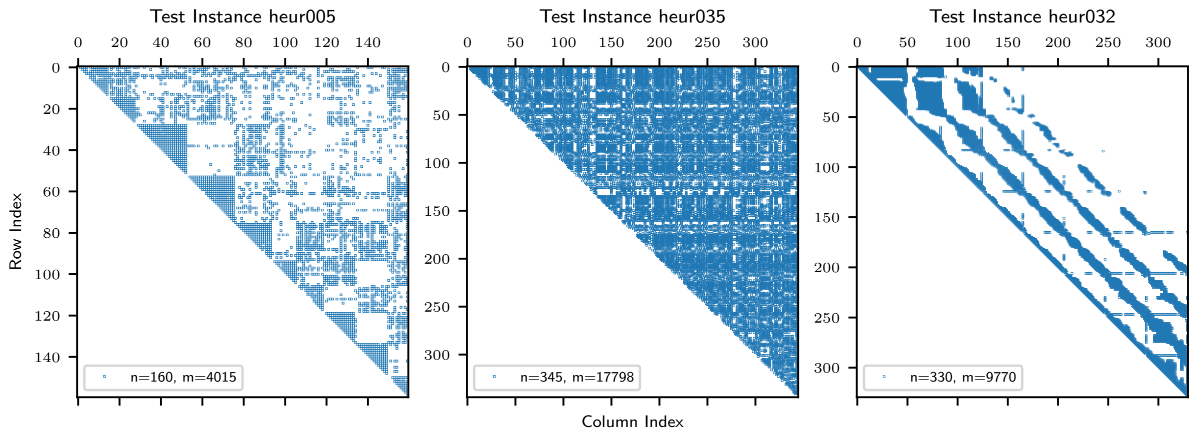
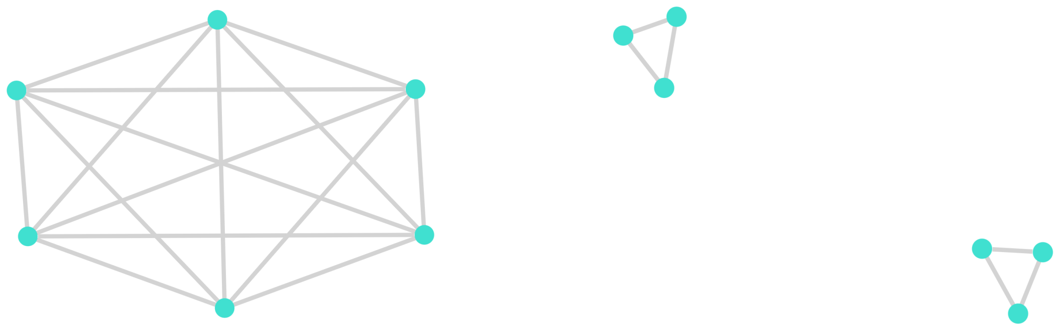


Figure 1: Instances of the different graph types. Left is a locally strongly connected graph, this is part of instance 005. In the middle is a seemingly randomly connected graph, which is part of instance 035. On the right is a tentacle-like connected graph, which is part of instance 032.

We will use this special structure of some instances to derive parts of our algorithms. Some might consider this cheating, but we think, that finding patterns in the instances is an important part of solving problems in real world applications so we find it to be okay. We will see later, that for the locally strongly connected instances we can find reasonably good solutions with very little computational effort. We will also see that the algorithms that were derived with the locally connected graphs in mind also work well on the other graphs.

6.1.1 Fuse Operation

The case of locally strongly connected graphs made it easier to think about the problem. The idea for our next move came from looking at the following situation. Let's say we are looking for 2-plexes and a certain section of our adjacency matrix is a clique, so A_0 in that part looks like



(a) Graph with A0 Adjacency Matrix

(b) Solution A based on A0

Figure 2: Caption for the entire figure.

To get from our constructed A to A_0 in this section we need to set to 1 all the entries between the 2 cliques. In terms of the graph, we fully connect the two found cliques. Exactly this is our idea for the first move, which is then also applicable for the randomly distributed graphs.

So our first move after construction is fully connecting existing clusters, this always yields a legal s-plex. We call this operation a **fuse** the corrsponding functions are called `fuse_first!` for first improvement and `fuse_best!` for best improvement. We implemented this operation in a way, that it can fuse any 2 clusters, not only neighbouring ones.

Especially for the locally connected instances we found this to work very well and after repeatedly fusing clusters, our found adjecency matrices A showed very similar structures to the original ones A_0.

After fusing two clusters one could also try to remove edges that are not there in A_0 until no more edges can be removed legally. We describe our strategy for this in section 6.1.3. To do this after every move, set the global variable SPARSEN to true in `move_ops.jl` (as we said above: the code is very raw). We found that setting this to true improves the results slightly but greatly decreases performance (but we should mention here that we also implemented this in a very inefficient way). We decided that for good results it is sufficient to let the fuse method fully connect it's clusters and only at the end of the algorithm take out the unnecessary edges (see section 6.1.3).

6.1.2 Swap Operation

Looking at our results we found that in some cases, single nodes did not land in the cluster they were supposed to be in, for example in the case of instance 051 we found the solution given in figure 3.

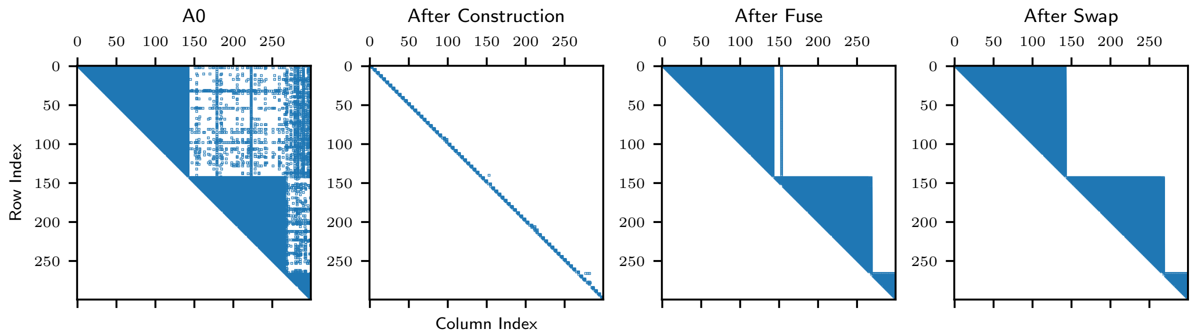


Figure 3: Instance 051 after fusing until no progress is made. One can see that node 154 seems misplaced. Indeed, when node 154 is swaped to the other cluster the cost is reduced by roughly 500.

We see that node 154 looks a little misplaced and should be in the first cluster note the second one. When we manually put it into the first one we found that this indeed reduced the cost from 8645 to 8153 which is a very significant improvement. So for our second move we think that it should be beneficial to try to swap nodes between clusters.

6.1.3 Operation for taking our redundant edges

All the discribed algorithms so far rather find cliques than s-plexes. In the fuse method, the clusters get fully connected, so any deviation from a clique must be in the original small clusters from the construction heuristic. Our algorithm finds clusters that are also clustered in A_0 but up till now it does not deal well with the edges in A_0 that are missing within a cluster. Generally, the most profitable edges in a cluster to remove will not always be in the small cluster from the construction heuristic. To find these most profitable edges to remove we implemented the **cliquify then sparsen** method. This method first fully connects the clusters within A and then removes as many edges as possible, starting from the most profitable edges. This does not really fit into the framwork of metaheuristics, that we saw in the lecture, since it is more of a ”cleanup” at the end of the search algorithm. One could also do one **cliquify then sparsen** after each however, this is again connected to more computational effort.

6.1.4 Shaking operation

For the shaking operation for GVNS we thought that randomly disconnecting n nodes totally might be a good idea. We implemented this an we will discuss it in the results section.

6.1.5 Final thoughts

In the case of the locally strongly connected graphs, our deterministic construction with a first improvement fuse strategy already works very well while being very efficient. The deterministic construction which goes through the nodes by their numbering and thereby clusters them mostly right, since nodes that are close in numbering are also "close" in the graph. The first improvement fuse operation will then first fuse the clusters which are closest in numbering, which also fits the structure of these graphs.
In the case of seemingly randomly connected graphs, the numbering is not correlated to how close nodes are in the graph, so here a random construction with GRASP, that rather randomly groups nodes in the beginning should perform better. For the same reason, a best improvement strategy for the fuse-operation should be better, however, this is very costly in terms of computation.
From our discussion above, we come to the conclusion, that first progressively fusing clusters until there is node more profit to make from fusing and then swapping the misplaced nodes until there is no more profit to make is a reasonable strategy here. Since there was no metaheuristic discussed in the lecture that exactly fits this procedure we will call it **sequential local search** (SNS) because we sequentially apply move operators. It is somewhat of a mix of local search and VND. We will also compare this metaheuristic to the other ones. Our expectation is that the results will be worse than for e.g. VND but that the computational performance will be significantly better.

7 Neighborhood Structures and Step Functions

Develop or make use of a framework for basic local search which is able to deal with:

- different neighborhood structures
- different step functions (first-improvement, best-improvement, random)

5. Develop a set of meaningful neighborhood structures to address the different aspects of the problem, i.e., related to the quality of the s-plexes, or the assignment of nodes to s-plexes.

8 Variable Neighborhood Descent (VNS)

text

9 Greedy Randomized Adaptive Search Procedure (GRASP)

10 General Variable Neighborhood Search (GVNS) on top of VND

we chose disconnect as shaking moves. It turns out not to work so well, most of the time no improvement. we found one improvement for one instance, here is the console output, took a long time to run.

```
GVNS for ../data/datasets/insttest/heur020n320m5905.txt
performing gvns for file ../data/datasets/insttest/heur020n320m5905.txt
copying solution
found value 4014 after first vnd
gvns iteration 1 out of 5
copying solution
gvns iteration 2 out of 5
gvns iteration 3 out of 5
gvns iteration 4 out of 5
gvns iteration 5 out of 5
found obj-fct value is: 4009
```

11 Manual Paramter Tuning

Perform some manual tuning of relevant algorithmic parameters to find sensible parameter settings for the final experiments. Relevant parameters may be related to the degree of randomization, neighborhood structure sizes, probabilities for the random step function in composite neighborhood structures, the cooling schedule, the tabu list length and its variation, etc. Report the impact of a number of different settings on the solution quality of a selected meaningful subset of instances.

12 Delta Evaluation

Use delta-evaluation. Explain which steps in your algorithm use delta-evaluation and describe why delta-evaluation results in better performance in this step. Are there other elements in your algorithm that could have also benefitted from delta-evaluation?

13 Delta Evaluation

We implemented delta evaluation, but we only did so for one case to have a working example with an estimate of the performance gain and a proof of correctness.
Since the structure of our functions did not really fit delta evaluation, we created new, adapted functions. The basic idea behind our delta evaluation is that for every changed bit in A we calculated the added cost similar to the following pseudocode:

```
changed = (G.A[i, j] == 1)
G.A[i, j] = 0
costreduced = (G.A0[i, j] == 0)
addedcost += changed * (-costreduced * 2 + 1) * G.W[i,j]
```

We adapted the move operators `fuse_first!` and `swap_first!`, you can find the adapted versions in `move_ops_delta.jl`. We made a new version of the VND which you can find in `metaheuristics.jl`, it is called `vnd_delta!`.

We only did a quick check on the performance and correctness of our implementation of delta evaluation in `delta_eval.jl`. If you run it you will see, that the improvement that delta evaluation made was (if at all existend) smaller than statistical fluctuations of the runtime. On our machine it takes 50 seconds to run, if you dont want to run it, the output of our machine is: (for one run, the time, of course, fluctuates statistically)

```
======
Demo of Delta Evaluation
======
running vnd and vnddelta for file ../data/datasets/instcompetition/heur051n300

Profiler of VND with fastest configuration with delta evaluation
calculated obj value of delta evaluation is: 6930
Actual obj value is: 6930
Total time for VND is 3.578387975692749
====


Profiler of VND with fastest configuration without delta evaluation
Actual obj value is: 6930
Total time for VND is 3.64827299118042
====
```

We know that this is not good practice in terms of experimentation but we think it is sufficient to show that delta evaluation is correct and for this problem not very useful. We also ran this for other files, the results were comparable.

14 Experimental Results

15 Results and Conclusion

Run experiments and compare all your algorithms on the instances provided in TUWEL:

(a) deterministic and randomized construction heuristic and GRASP

(b) Use the solution of the deterministic construction heuristic to test the other implementations:

i. Local search for at least three selected (possibly composite) neighborhood structures using each of the three step functions (i.e., at least nine different algorithm variants).

ii. VND

iii. GVNS, SA, or TS