Algorithm Engineering – Exercise 3

Team 02: Julian Fechner, Julio Cesar Perez Duran and Denis Koshelev

1. Implemented Features

We added missing bounds in each iteration, implemented eight reduction rules from the lecture, and wrote a program that shows the difference between graphs after reduction.

Zero-Degree rule: The initial graph does not contain zero-degree vertices per definition. We also added the rule in every iteration for the last exercise.

High-Degree and Buss rules: We iterate over all vertices $v \in V$. We delete all vertices where $|N(v)| > k$. Repeat this process until not possible and reduce k accordingly. After Zero- and High-Degree we always apply the Buss rule: either increment the bound before branching or return no solution, if it doesn't exist during the branching.

One-Degree Rule: We iterate over all vertices $v \in V$. If $|N(v)| = 1$ and delete $N(v) = \{u\}$ from graph.

Two-Degree Rule: We iterate over all vertices $v \in V$. If $|N(v)| = 2$, we delete+merge the vertices according to the lecture.

Domination Rule: As seen in the lecture.

Unconfined Rule: As seen in the lecture.

LP Rule: We implemented an ineffective version of the LP rule, which allowed us to reduce the number of recursive steps, but the time pro recursive step increased and caused worse performance overall (e.g. for `05-football.graph` with this rule 93 recursive steps with 5 seconds against 432 steps and 2 seconds without the rule). That's why we introduced a threshold so that we can use it for smaller graphs.

2. Data Structures

In our second submission we have introduced custom data structure `VertexDegreeOrder` that stored vertices sorted by degree and had a pointer for a maximum degree vertex. Also in our `Graph` class we stored some redundant information, such as array and hash map of edges. Writing reduction rules we realized that this code's bad maintainability complicates engineering, so we had to refactor our code. Unfortunately, our code became slower and could solve 12 instances less afterward, but we made this trade-off intentionally to be able to write more reduction rules for this exercise.

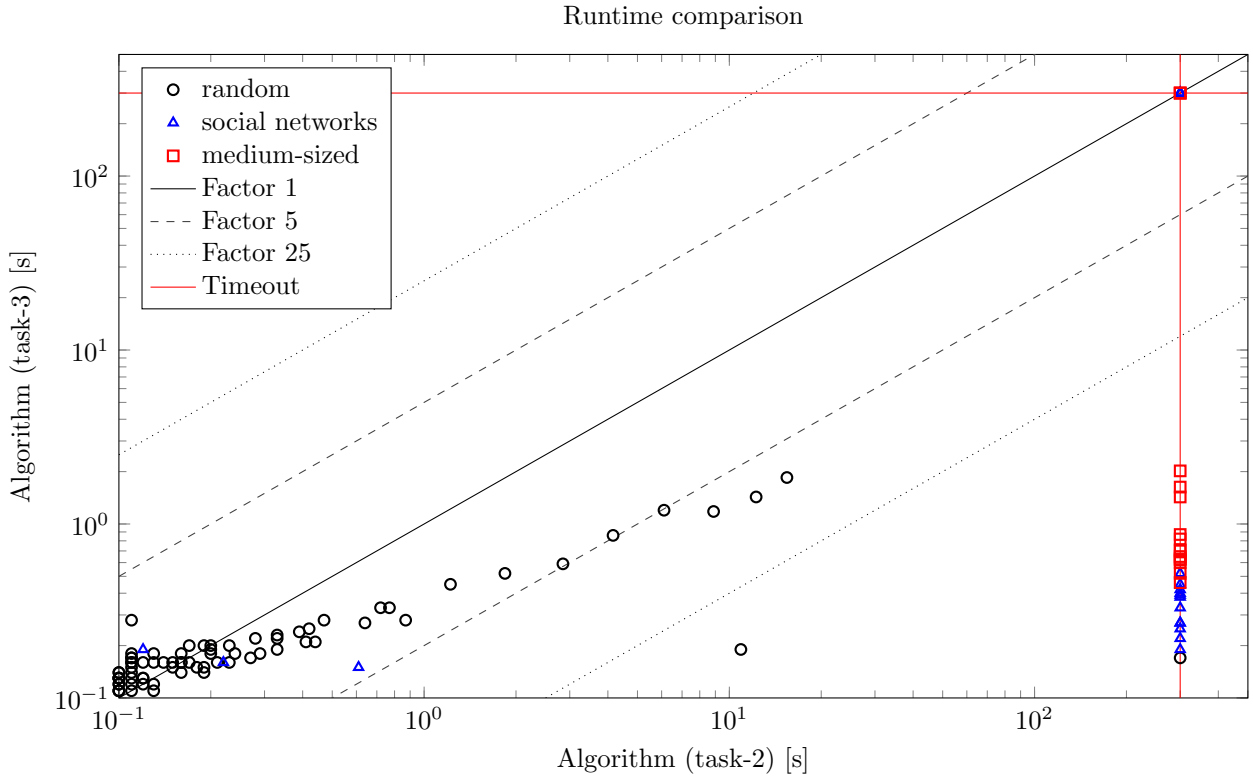
We also implemented `ReductionRules` class only before initializing our `Graph` class. This allowed us, to apply these rules on a single data structure, hash map, instead of multiple data structures in our `Graph` class. This resulted in a minor but still noticeable speed-up.

3. Highlights

- **Runtime comparison**: task-2 VS task-3 (Figure 1)
- **Rules comparison**: individual effects (Figure 1)

4. Experiments

- **Reduction before Bounds VS after Bounds**: As expected, this experiment has shown, that the number of solved instances has been significantly increased when applying reduction rules before computing bounds.
- **Reduction in n'th iteration**: In this experiment, we tested the effect of applying the reduction rules only in each 2nd and 3rd iteration. It has shown, that the execution time has not been reduced effectively, nevertheless, the number of recursive steps increased.
- **LP-Reduction Threshold**: Since we implemented a slow version of the LP-Reduction, we had to determine a threshold parameter for this rule. Experimenting with different thresholds has shown, that the execution time dramatically increases when the threshold parameter is too large (> 100 vertices).
- **Changing order of rules**: We also tested the impact of applying rules in a different order. It turned out, that the order did not make a notable difference compared to our already implemented order.
- **Sequential VS Parallel Rules**: Some rules (eg. 1-degree, 2-degree, and domination rule) can be computed in one run instead of applying them sequentially. There was no significant effect noticeable.



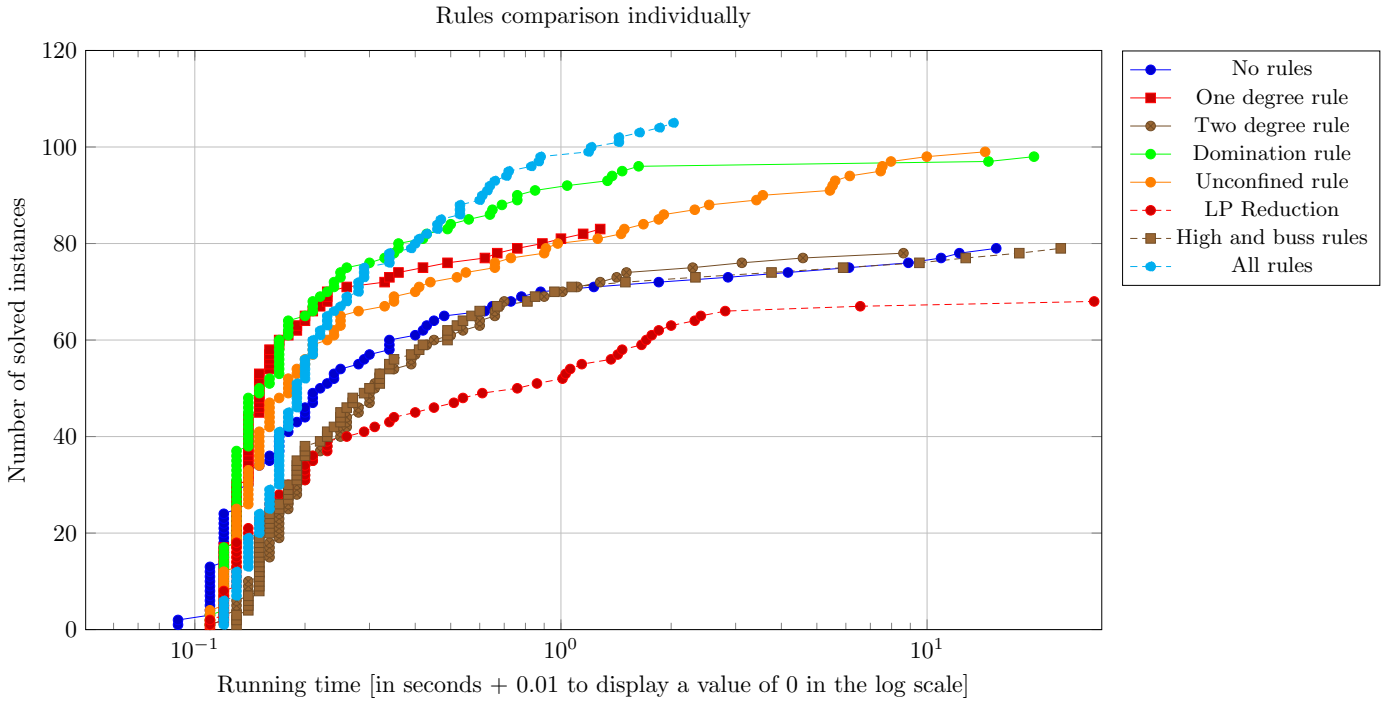


Figure 1: This plots compares all rules enabled separately, simultaneously at the beginning of the algorithm and in every iteration of the branching. Timeout was set to 30 and 3 instances were allowed to fail.