Algorithm Engineering – Exercise 3

Team 3 (Lisa Dörr, Dennis Weiss, Khiem That Ton)

1. Implemented Features

Instead of incrementing k linearly when searching for the minimum k, we now use exponential search to find an interval containing the minimum k with a base of 1.2 in $O(log_{1.2}(k))$ time. Afterwards binary search is employed.

We implemented the following data reduction rules from the lecture (given in order of application):

1. Weight$>k$ rule
2. Heavy non-edge
3. Heavy edge, single end
4. Heavy edge, both ends
5. Clique rule
6. Large neighborhood rule
7. Min-cut rule (Large neighborhood rule II)

Our min-cut rule was implemented by firstly determining the closed neighborhood of each vertex and then using the Nagamochi Ibaraki Algorithm [1] to find the min-cut cost of the closed neighborhoods. All except the min-cut rule are applied at every step exhaustively. The strategies for using the min-cut rule are described in Section 3.

2. Data Structures

We changed the underlying data structure compared to the version of milestone 2. We are now using an adjacency matrix for the edge weights, which are realized as 2-dimensional arrays in Java. Furthermore, we avoided uses of possible inefficiencies like Java Streams. As it can be seen in Fig. 1 our new implementation is much faster. For the heavy edge reductions we store for each vertex the sum of the (signed) weights to all other vertices and the sum of the absolute weights.

3. Experiments

In Fig. 2 we investigated the effect of the three heavy edge reduction rules. A clear improvement might not be immediately observable, especially as some instances were slower compared to without those reductions, but in Tab. 1 it becomes evident that the average running times could be reduced by 45.4%. This is caused by a reduction of the number of recursive steps to less than a half and only a slight increase (23.7%) in running time for each recursive step. Based on the observations in Tab. 2 it is notable that

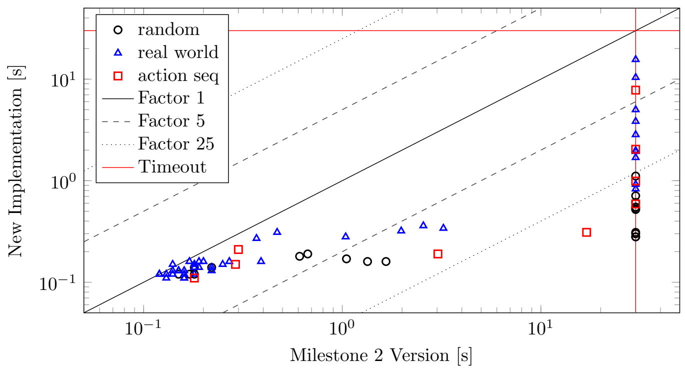


Figure 1: Comparison of running times of new implementation with an adjacency matrix and without Java Streams and with Arrays

the reductions showed to be effective for the real-world and action-seq instances, but not at all for the random instances. This observation is consistent with the work of [2], where it was observed that the random instances could rarely be reduced, even though the investigation was made for a different reduction rule.

Regarding the large neighborhood reduction we conclude that it is rather ineffective (Fig. 3). In our experiments we found that the reduction could only be applied on rare occasions. For the min-cut neighborhood rule we employed different strategies of applying this reduction rule. One strategy is to deterministicly apply the rule in every n^{th} recursion layer. The other strategy is to apply this rule in every recursive step with probability $\frac{1}{n}$. We call this factor of how often the rule is applied α. In Fig. 4 we plot the running times per instance, number of recursive steps per instance and running times per recursive step for different values of α. Overall it can be said that the number of recursive steps can be reduced slightly by employing the reduction rule (Fig. 4c and Fig. 4d). But on the other hand the time per recursive steps grows largely by computing the min-cuts as it can be seen in Fig 4e and Fig. 4f. Overall it can be concluded from Fig. 4a and Fig 4b, that using our current implementation of the min-cut rule slows our algorithm down.

References

[1] H. Nagamochi, T. Ono, T. Ibaraki, Implementing an efficient minimum capacity cut algorithm, Mathematical Programming 67 (1) (1994) 325–341.

[2] H. Schulz, On Efficient Cut-Based Data Reduction for Weighted Cluster Editing (2021).

	Without heavy edge data reductions	With heavy edge data reductions
Avg. time per instance [s]	7.3643	4.0238 (-45.4%)
Avg. recursive steps per instance	4161	1837 (-55.8%)
Avg time per recursive step [s]	$1.770 \cdot 10^{-3}$	$2.190 \cdot 10^{-3}$ ($+23.7\%$)

Table 1: Comparison of running times and number of recursive steps between version with and without heavy edge reduction rules

	random		real-world		action-seq	
	Without heavy edge data reductions	With heavy edge data reductions	Without heavy edge data reductions	With heavy edge data reductions	Without heavy edge data reductions	With heavy edge data reductions
Avg. time per instance [s]	6.4512	6.8886 ($+6.8\%$)	8.6306	2.5969 (-69.9%)	4.4642	2.0900 (-53.2%)
Avg. recursive steps per instance	1394	2320 ($+66.4\%$)	5064	1341 (-73.5%)	8087	2622 (-67.6%)
Avg time per recursive step [s]	$4.626 \cdot 10^{-3}$	$2.969 \cdot 10^{-3}$ s (-35.8%)	$1.704 \cdot 10^{-3}$	$1.937 \cdot 10^{-3}$ ($+13.6\%$)	$0.552 \cdot 10^{-3}$	$0.797 \cdot 10^{-3}$ ($+44.4\%$)

Table 2: Comparison of running times and number of recursive steps between version with and without heavy edge reduction rules for the 3 different instance categories

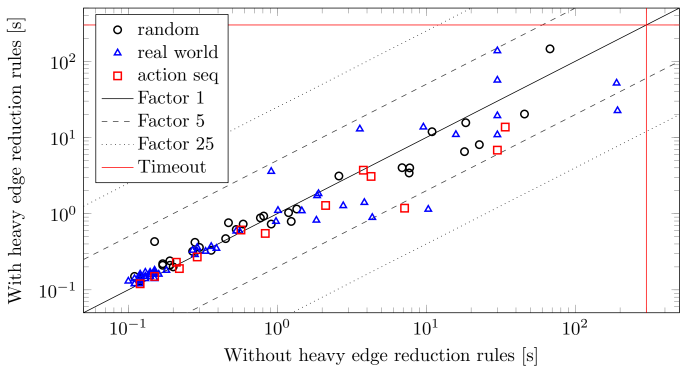


Figure 2: Comparison of running times of versions with and without the heavy edge reduction rules

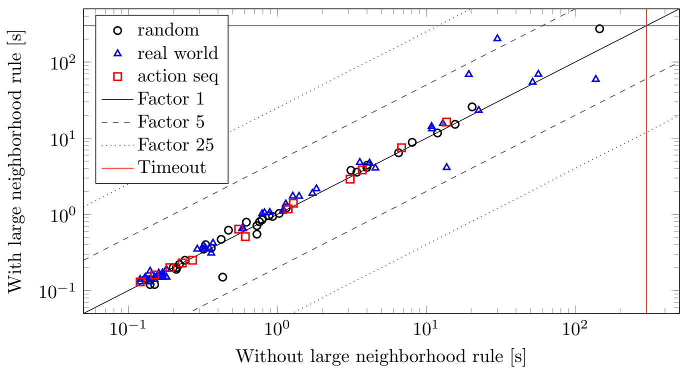
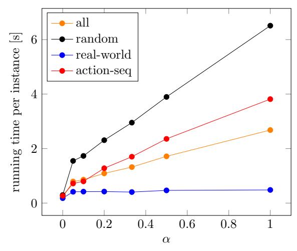
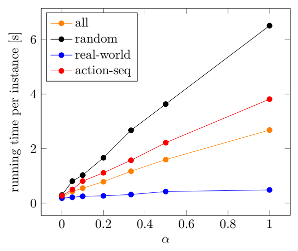


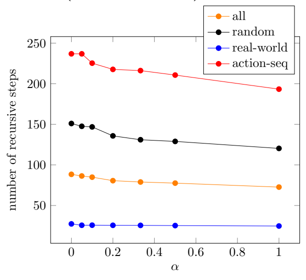
Figure 3: Comparison of running times of versions with and without the large neighborhood reduction rule



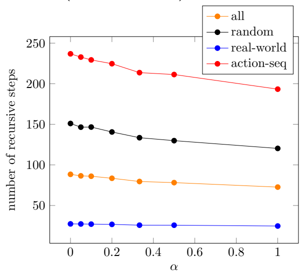
(a) Running times per instance for different rule application factors α (deterministic)



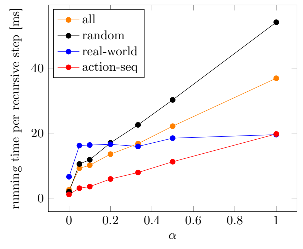
(b) Running times per instance for different rule application factors α (randomized)



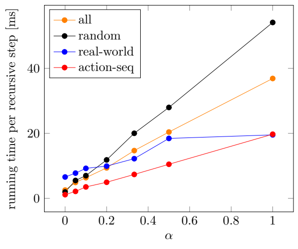
(c) Number of recursive steps per instance for different rule application factors α (deterministic)



(d) Number of recursive steps per instance for different rule application factors α (randomized)



(e) Running time per recursive step for different rule application factors α (deterministic)



(f) Running time per recursive step for different rule application factors α (randomized)

Figure 4: Analysis of running times and number of rec. steps for different employment strategies of the min-cut neighborhood rule