Algorithm Engineering – Exercise 2

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

1. Implemented features

- Our baseline version for this 2nd milestone consists of an implementation that merges two vertices if the weight of the edge between these two vertices is greater than the current value of k. After the recursive calls of *ceBranch* it is necessary to reconstruct the original graph from the merging operation. That is done by storing what vertices a merged vertex has been merged from and using their unmodified list of weights to all other vertices present before the merge.

- We additionally implemented merge branching which reduces the branching to two cases (delete an edge or merge its vertices). That results in reducing the size of the search tree from $\mathcal{O}(3^k)$ to $\mathcal{O}(2^k)$. Thereby, we also saved the forbidden edges and made sure to not edit them or an edge they merged into again.

- To store the adjacency list of a vertex we used hash maps that map from the vertex to the associated weight instead of a pure list. This results in a $\mathcal{O}(1)$ search instead of a $\mathcal{O}(n)$ search.

- Now, in each recursive step we search for all P_3's and choose the P_3 with the largest total absolute weight to branch on. Although there is some additional work in searching all P_3's instead of just searching for one P_3, the algorithm is significantly faster, as the number of recursive steps get reduced in most cases.

- We used the branch-and-bound paradigm with lower bounds 1 and 2 from the lecture.

  $$k \geqslant t \cdot \min_{u,v \in V} \frac{|s(u,v)|}{t(u,v)}, \qquad (1)$$

  with $s(u,v)$ being the weight of the edge between u and v, $t(u,v)$ being the number of P_3's the edge between u and v and t being the total number of P_3's of the graph with vertices V.

  $$k \geqslant \sum_{(u,v,w) \in P} \min(s(u,v), s(v,w), -s(u,w)), \qquad (2)$$

  with P being a list of edge-disjoint P_3's.

- An additional improvement has been made in running time of creating a list of edge-disjoint P_3's. Instead of searching through all triples of vertices to create a list of edge-disjoint P_3's, we pass the list of all P_3's to the computation of lower bound 2. With that strategy we could save up $\mathcal{O}(n^3)$ calls every time the lower bound computes a list of edge-disjoint P_3's.

2. Data Structures

In principal, we use an adjacency list to store the weights of a vertex to the $n-1$ other vertices. Using an adjacency list over an adjacency matrix makes the merging operation much more practical, as during a merging operation only two vertices in an adjacency list need to replaced by one merged vertex. For an adjacency matrix it would be necessary to change the size of the entire matrix, as the number of vertices changes with the merging operations.

Wherever we would use a linear search in a list, we changed that list data structure by a hash map that maps from some key to the object we search for. In the case of the adjacency list that contains objects with the neighbored vertex and the corresponding weight, the mash map maps from the vertex to that object. This allows for an on average $\mathcal{O}(1)$ search of a weight of an edge.

3. Experiments

We compared the running times of different versions of the algorithm. In figure 1 it can clearly be seen that the use of hash maps gave a clear advantage compared to the use of simple lists. In figure 2 we see that the use of lower bound 1 made the whole algorithm much slower, owed to the additional work required for computing the value of the lower bound. By comparing the number of recursive steps we observed that the number of recursive steps were not reduced significantly, which indicates that this is not a tight lower bound and thus not of great use. A significant gain in algorithm speed was achieved by employing the merged branching strategy as it can be seen in figure 3, It allowed us to solve instances within the time limit that could not be solved before. Figure 4 indicates that the version with lower bound 2 poses a slight deterioration of the algorithm, even though the number of recursive steps could be reduced in most cases. That implies that lower bound 2 is a tighter bound than lower bound 1. Eventually, we conclude that the current version of the algorithm clearly outperforms the algorithm from our first milestone, as figure 5 states.

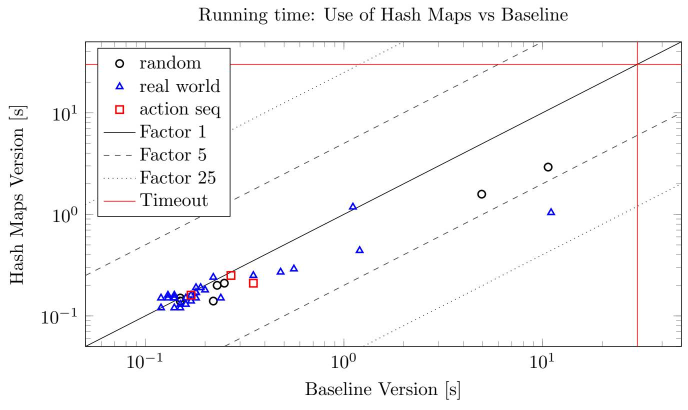

Figure 1: Comparison of running times of hash maps version with baseline version

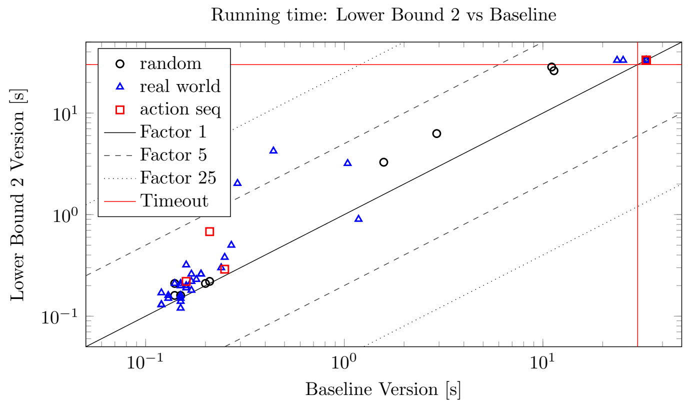

Figure 4: Comparison of running times of the lower bound 2 version with baseline

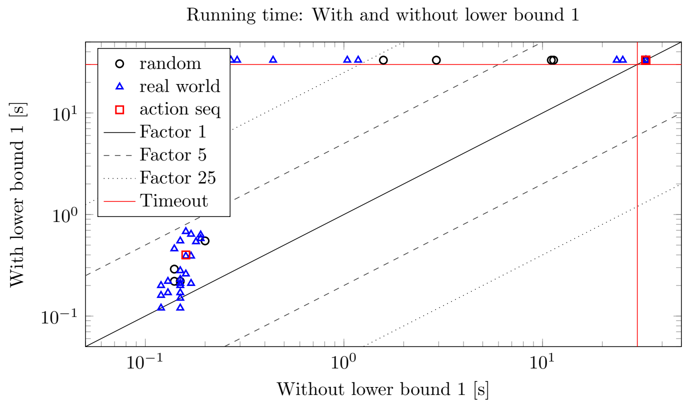

Figure 2: Comparison of running times between version with and without lower bound 1

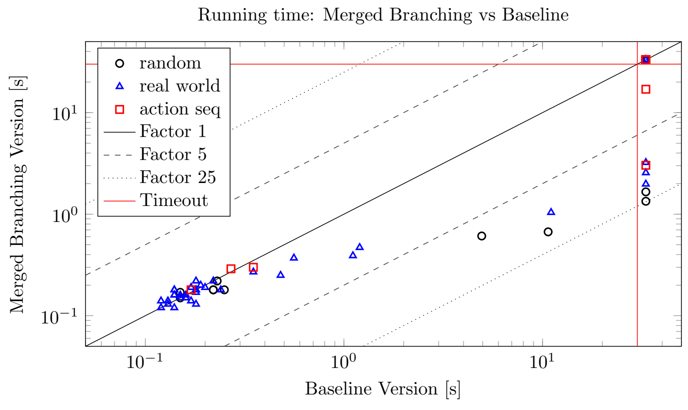

Figure 3: Comparison of running times of the merged branching version with the baseline version

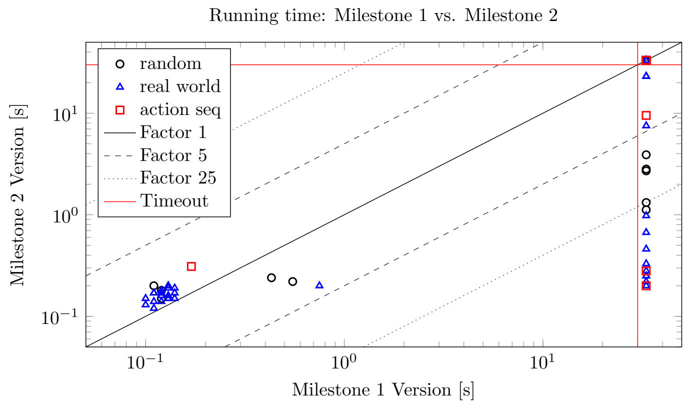

Figure 5: Comparison of running times of version of this milestone and the previous milestone