Algorithm Engineering – Exercise 1

Khiem That Ton, Lisa Dörr, Dennis Weiss

1. Implemented Features

The **main()** function reads the input and creates the graph, i.e. the adjacency matrix and the adjacency lists (see Data Structures). We implemented exactly the methods **ce(Graph)** and **ceBranch(Graph, weight)** from the lecture except the following: In order to not copy the data structures every time we edit them before branching (calling **editEdge(i, j)**) and revert the changes afterwards (again by **editEdge(i, j)**). The function **editEdge** does the following: If the edge exists, we remove it and if it does not exist, we add it. Therefore we just have to multiply the entry in the adjacency matrix by -1. For the faster **findP3** algorithm we also need to remove or add entries in the adjacency lists, respectively.

Our naive implementation of **findP3()** simply goes through all possible combination of three vertices and checks with the help of the adjacency matrix if they form a *P3* in any way.

Our second implementation of **findP3()** calls **findVerticesOfConnectedComponents()**, which returns exactly one vertex per connected component by performing a **depthFirstSearch(Vertex, visitedVertices)** that marks visited vertices and afterward starting again with one that was not already marked. Afterwards **findP3(Vertex u)** searches *P3* in the connected component represented by the Vertex by calling **findP3In2Closure(Vertex)**, which checks all adjacent vertices w of vertices v which are adjacent to u if they are also adjacent to u. It that is the case, u, v and w form a *P3*. If we cannot find any, u has an edge to all other vertices in its connected component. We check if any of u's neighboring vertices v has less edges: If yes, we call **findP3(Vertex v)** on it. If not, that connected component is a cluster.

2. Data Structures

We use an Adjacency Matrix with the weights in it mainly because that way it is very easy to store the weights and check if a specific edge exists or not.

For the second **findP3()** implementation we also use Adjacency Lists (one list per vertex containing all vertices it has an edge to.)

3. Highlights

Our second implementation of the **findP3()** is currently slower than the naive implementation. This is probably because operating on arrays is often very quick. And this might change when we are able to solve bigger instances, because the naive approach runs in $O(n^3)$ and the other in $O(n + m)$. Still that was surprising to us.

We have some ideas, how we could improve our second **findP3()** implementation:

- get rid of the adjacency matrix that is mainly used to store the weights
- improve **findVerticesOfConnectedComponents()** by updating it as well when **editEdge()** is called or merge it with **findP3In2Closure()**, which could already mark some vertices

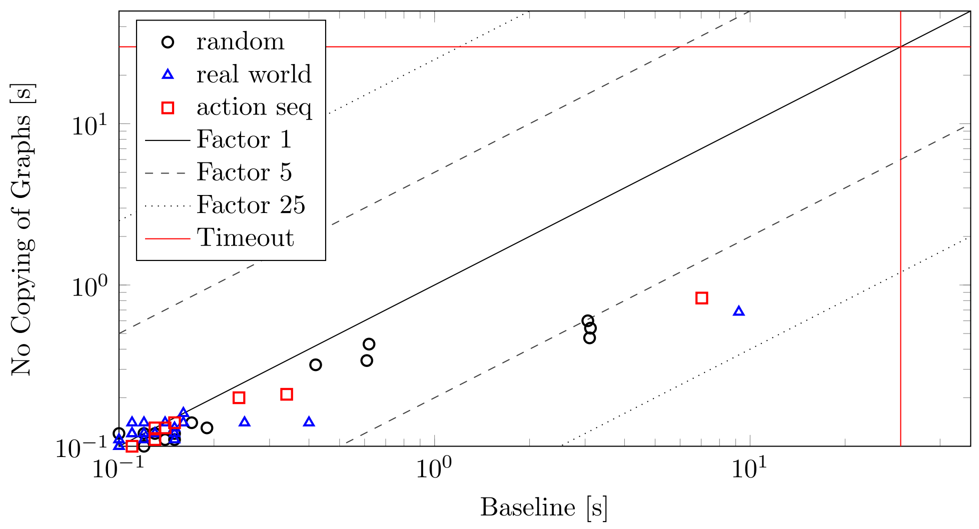
4. Experiments

We compared the running times of each of the mentioned optimizations with a baseline algorithm version, which does not contain any optimization, but the most-straightforward and naive implementation. In the following plots the running times for each input instance are depicted. Finally, we compared the baseline algorithm to our submission version which contains the copy optimization and branching optimization.

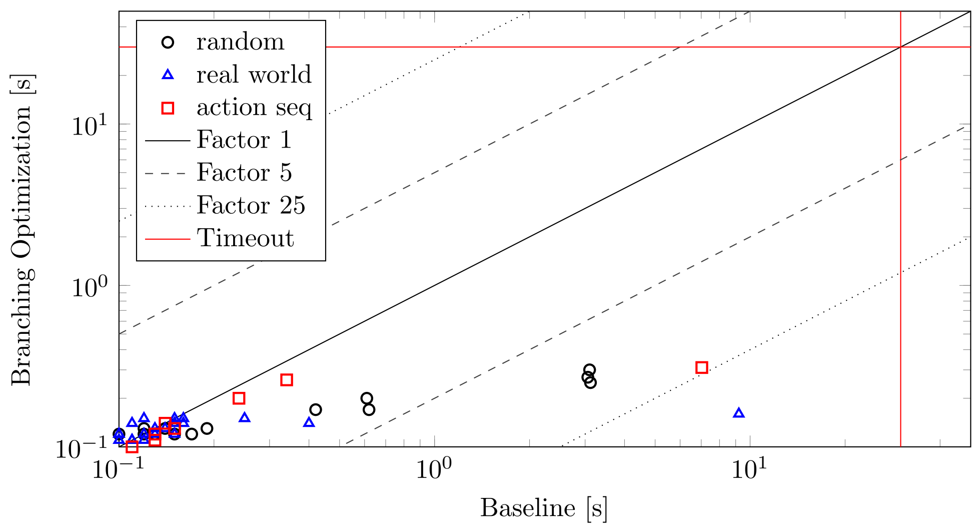
Our optimizations were:

1. Instead of copying the whole graph, we edited it before branching and reverted the changes afterwards.
   That improved the run time with the given test set by a factor up to 10.
2. We saved the edges we already edited and added a check to not branch on those again.
   That improved the run time with the given test set by a factor up to 25.
3. Second find P3 algorithm as described in section 1.
   That mostly worsened the run time with the given test set by a factor up to 4. Except for three cases where it improved the run time.
4. We branch on larger weights in the triangle first. (To find globally the largest weight of an P3 will be a future improvement.)
   That only slightly improved the run time with the given test set by a factor up to 1.1.

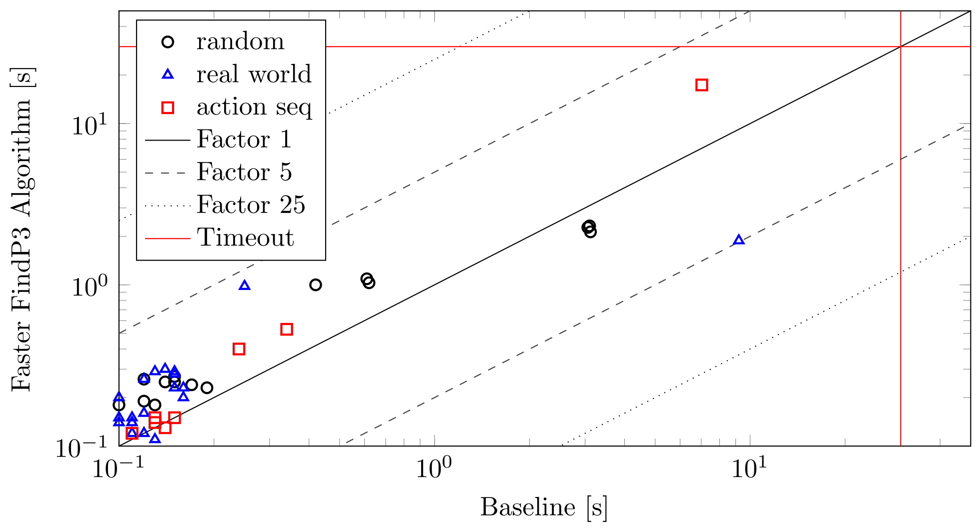
No Copying of Graphs



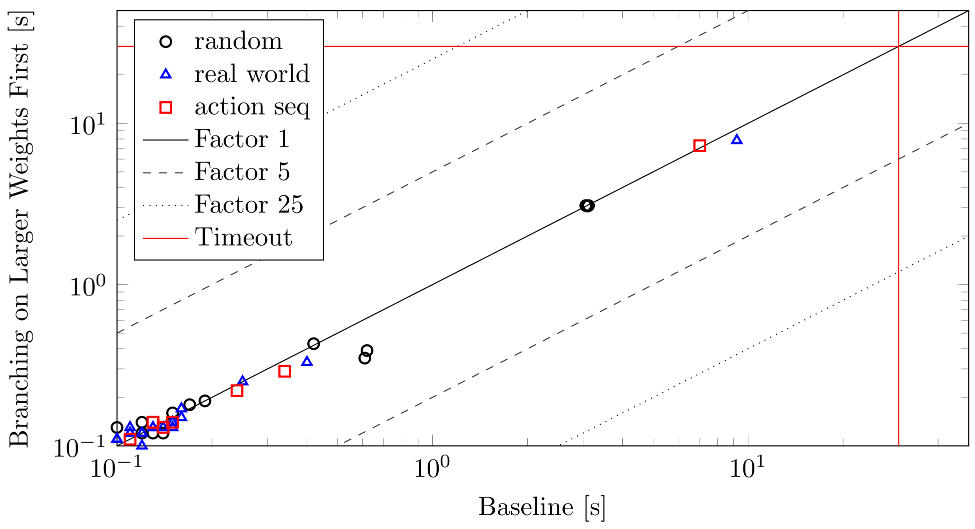
Branching Optimization vs Baseline



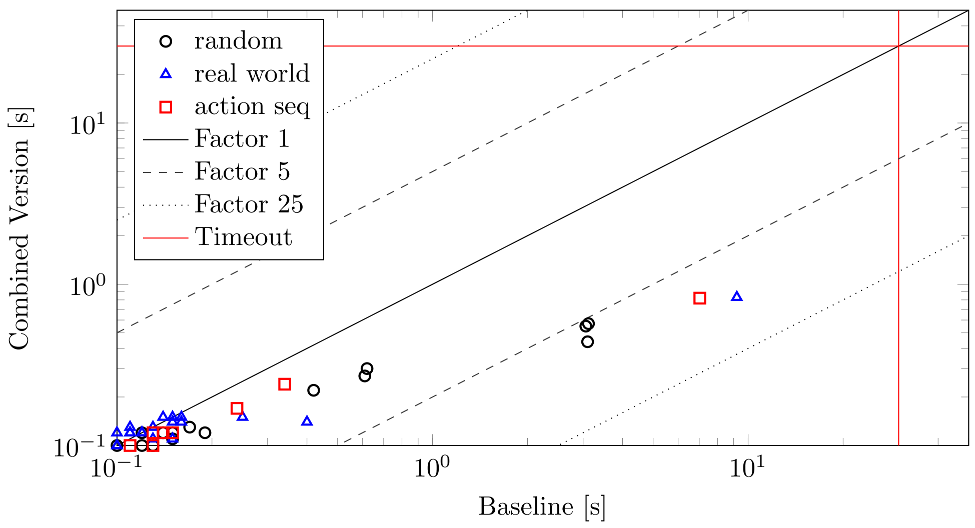
Faster FindP3 Algorithm vs Baseline



Branching on Larger Weights First vs Baseline



Combined Version vs Baseline


random
real world
action seq
Factor 1
Factor 5
Factor 25
Timeout
Combined Version [s]
Baseline [s]
10^{-1}
10^{0}
10^{1}