Algorithm Engineering – Exercise 2

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

1. Implemented Features

1.1. Clique bound

We implemented two different approaches to compute an approximate vertex clique cover. The first one is based on the degree of the vertices since it takes the vertex with the highest degree available in every recursion level and builds a clique with it, deleting the associated vertices from the graph. In the second approach, we randomly shuffle the list of vertices and construct cliques following the order of the vertices in the list. We do this ten times and choose the best result. We were expecting a higher lower bound with this one and, therefore, a lower running time than in the other approach, but this wasn't the case. See Figure 1 for a runtime comparison of these heuristics.

1.2. LP bound

Firstly, we construct a corresponding bipartite version with two disjoint and distant sets of vertices. As a result, we can compute a maximum matching for a new graph with **Hopcroft-Karp algorithm**: while there is an augmenting path, which is checked with breadth-first search, we mark all vertices that are not yet in the matching as chosen and vice versa. Finally, we halve the size of our maximum and this corresponds to the lower bound for our graph according to **Kőnig's theorem**.

1.3. Modified branching

We have implemented a modified branching strategy as described in the lecture. Therefore, we extract the vertex v of maximum degree and divide the Graph $G = (V, E)$ into two cases: once a graph without the vertex itself und k reduced by 1, and once without all neighbors and k reduced by the amount of them.

2. Data Structures

To extract the maximum degree vertex efficiently, we implemented a new data structure. It consists of a **hash map** with the degrees of the vertices as the keys and **hash sets** of vertices (having same degree) as values. Additionally, it contains a pointer to the maximum degree at the moment in the graph so that retrieving the next vertex with the highest degree is constant time possible. This data structure also stores keys in a sorted set to update the maximum degree faster when removing a vertex. We considered this data structure for the following reasons:

1. The data structure can be built while initiating the graph.
2. The space complexity is $O(n)$ with n being the number of vertices.
3. The time complexity of getting a vertex with the highest degree is constant.

Almost all the disadvantages stem from the fact that adding the degree of a vertex to the sorted set has a time complexity of $O(log(n))$, which increases the worst-case time complexity of adding, decreasing, and increasing the degree of a vertex from constant to logarithmic. But it was worth it to reduce the time complexity of removing a vertex by finding the next highest degree faster. See Figure 2 for the comparison between sorting the keys and not sorting them.

3. Highlights

- Using the new data structure (without sorted keys) instead of the heap built in Java does not provide the advantage we thought it would provide. See Figure 3 for the comparison. Our expectation of the customized data structure to be faster than the priority queue stems from the fact that every time a vertex changes its degree, one has to remove it and add it again, which is an O(n) operation, while it should theoretically be much faster with the customized data structure.

- The LP Bound is responsible for the highest jump in improvement (sizes of solutions achieved). Nevertheless, we believe the reason for that is that the LP bound is a bound for bipartite graphs and most random graphs are bipartite since we did not witness such an improvement for the other types.

4. Experiments

After comparing our improved algorithm with the first version, we observed great performance improvement. From Figure 4 could be seen that our algorithm solves 42 instances more than the last submitted version. Even though the algorithm didn't show great improvement for smaller graphs, since a lot of instances were solved faster but still within Factor 1, it could solve bigger instances much faster, on which the first algorithm timed out.

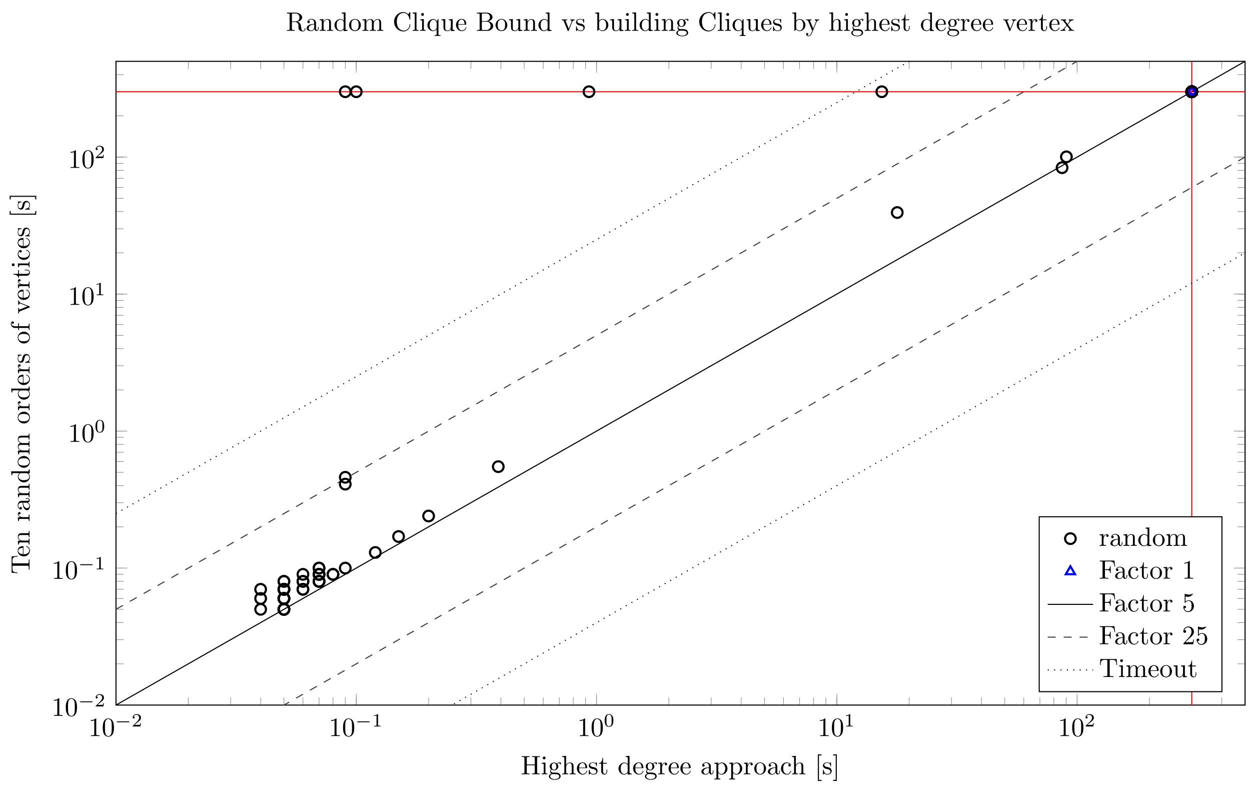

Figure 1: Comparison of the running times of the algorithm with the clique bound obtained with two different approaches,

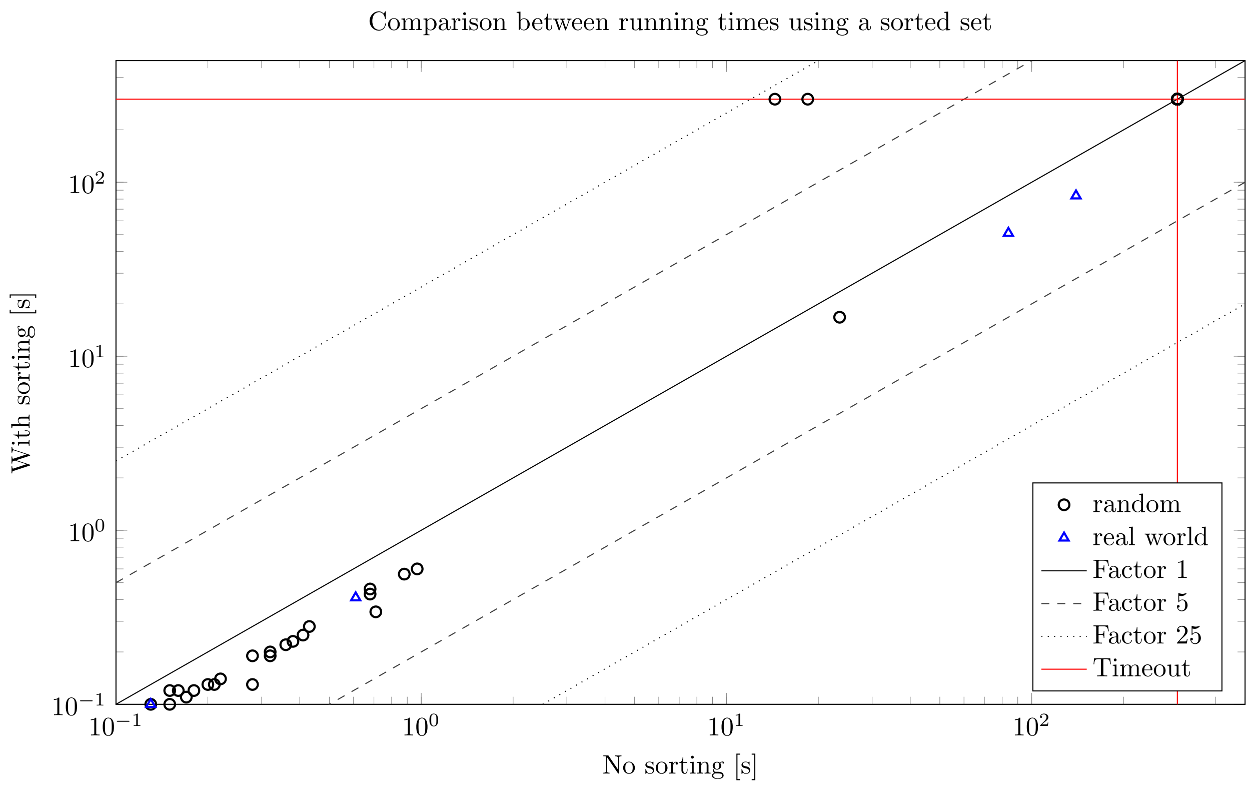

Figure 2: Comparison of the running times of the algorithm with and without sorted keys for the branching data structure

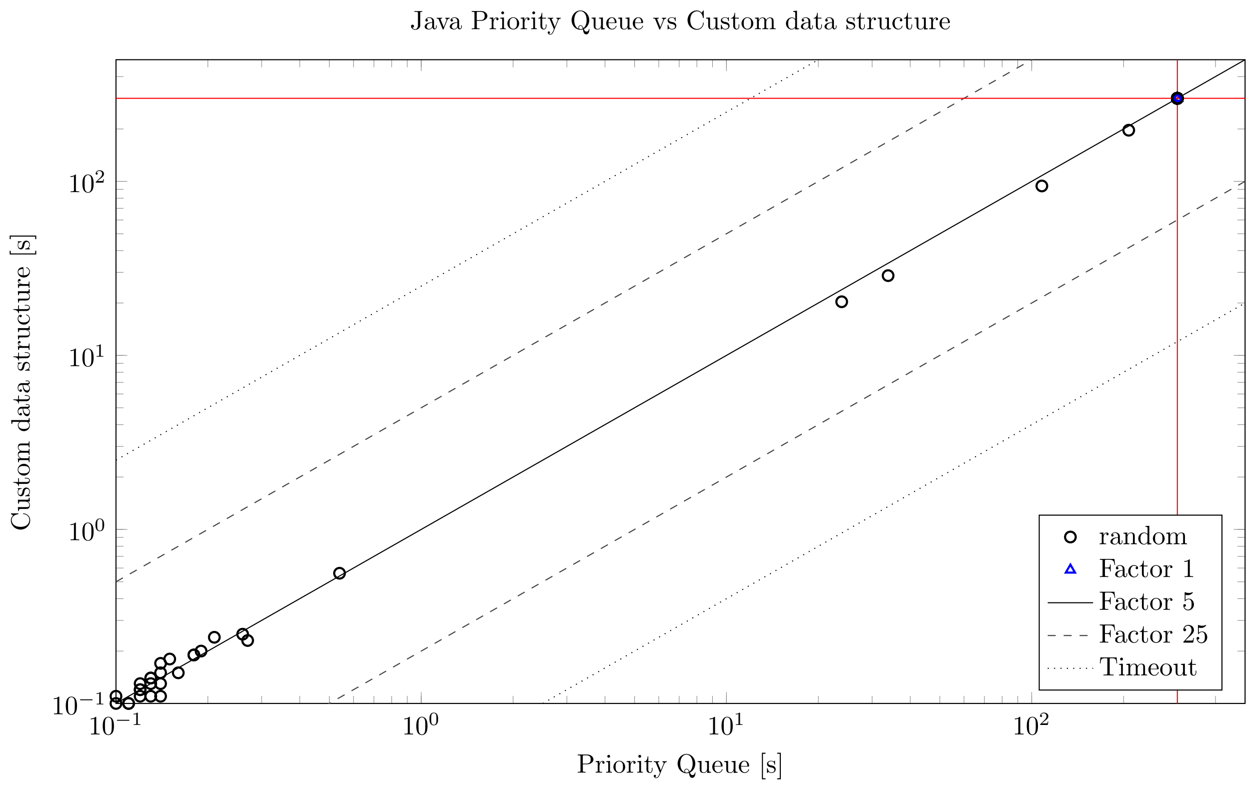

Figure 3: Comparison of the running times of the algorithm with the priority queue of java as a heap and a customized data structure (before adding sorted keys)

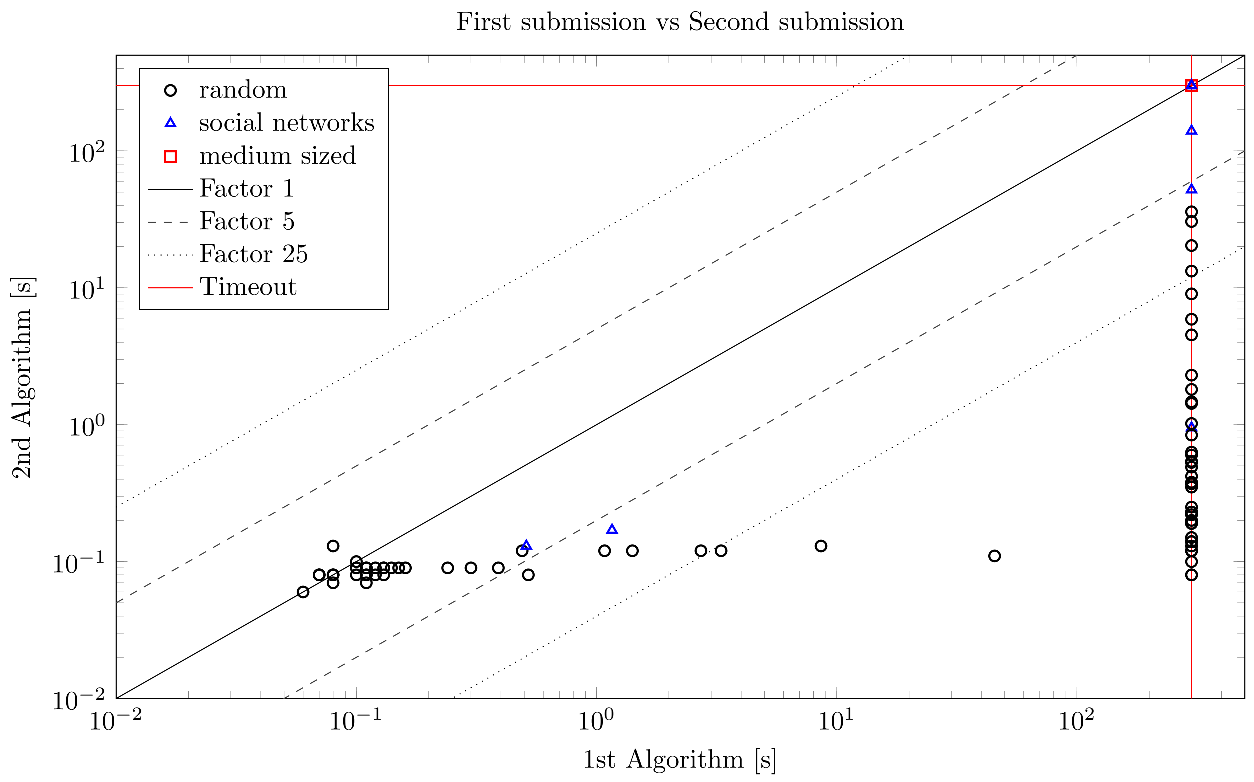

Figure 4: Comparison of the running times of algorithm from first and second submissions. As illustrated on the plot, last version of the algorithm could solve 42 instances more than the first one.