Beating the Worst Case
Winterterm 2026/2027
Max Göttlicher & Jean-Pierre von der Heydt

Karlsruhe Institute of Technology

Exercise Sheet 1: Entrance Exam

Published on 28.10.2025. Duration: 2 weeks

On our website[1] you will find a ZIP file containing 672 different graphs, which we have collected in a mysterious way. Your task is to implement two different algorithms and identify the graphs on which each algorithm performs particularly well.

To submit your solution, please provide a text file that lists the filenames of the identified graphs, one per line.

Prepare a short *5-minute presentation* with your results. Explain how you arrived at your solution and what insights or difficulties you encountered along the way.

```
0.gr
3.gr
8.gr
11.gr
14.gr
...
```

Figure 1: Excerpt from a possible submission file `solution1.txt`

Exercise 1: Warm-up Exercise

The framework[2] for this practical course provides basic functionality for working with graphs in C++. To get started, create your own fork of this repository. Read the framework's README file and follow the instructions to compile and run the example program `run`.

Exercise 2: Shortest st-Paths

To compute shortest paths in unweighted graphs, one can use the well-known breadth-first search (BFS). A common acceleration technique is to run the BFS *bidirectional* (bi-BFS). Here, when searching for an st-path, vertices are explored not only from s but also (in an alternating sequence) from t. This means that in addition to the state of the forward search (the "visited" markers and the queue of discovered vertices), the algorithm also maintains the state of the backward search. The algorithm alternates between the two directions after a layer has been completely explored (a

[1] `https://scale.iti.kit.edu/teaching/2025ws/beating_wc/start`

[2] `https://gitlab.kit.edu/kit/iti/scale/templates/praktikum_beating_the_worst_case_framework/`

layer consists of all vertices at the same distance from s or t). We consider the alternation strategy, which always continues with the direction for which the estimated cost of exploring the next layer is smaller. These costs are estimated as the sum of the vertex degrees in the previous layer.

The worst-case running time of bi-BFS is, as for classical BFS, $O(n + m)$. However, it has been observed that there exist graph classes on which the bidirectional search achieves an asymptotic speed-up compared to the unidirectional variant. That is, the factor by which Bi-BFS is faster grows with n.

Among the given graphs, there is a secret subset of graphs on which Bi-BFS exhibits *little* to *no* asymptotic speed-up. Implement the algorithm and evaluate it on the provided graphs to identify this secret subset.

Exercise 3: Vertex Cover

For a graph $G = (V, E)$, we call a subset $X \subseteq V$ a *vertex cover* if for every edge $\{u, v\} = e \in E$ it holds that $u \in X$ or $v \in X$. Determining whether there exists a vertex cover of a given size is NP-hard. However, there are polynomial-time reduction rules that can produce an equivalent but smaller instance. One such reduction rule is the *dominance rule*.

For two adjacent vertices $u, v \in V$, we say that u *dominates* v if $N[v] \subseteq N[u]$[3]. The dominance rule states that there exists a minimum vertex cover that includes u. Thus, the instance can be reduced by including u in the vertex cover and removing it from the graph. To assess the effectiveness of the dominance rule, we apply it *exhaustively*, i.e., until no dominant vertices remain. In addition, isolated vertices are removed. We denote by c the number of vertices in the largest connected component of the reduced instance and refer to this as the *kernel size*.

Among the given graphs, there is a secret subset of graphs on which this rule produces *particularly small* kernels. Implement the dominance rule and evaluate it on the given graphs to identify this secret subset.

[3]Here $N[v]$ denotes the closed neighborhood of v, i.e. $N[v] = N(v) \cup \{v\}$.