Algorithm Engineering **Summer 2026**

Dr. Phillip Keldenich
Dr. Dominik Krupke

Submission due: 07.05.2026 13:15

Homework 2

Work in teams of two. Homework is due every two weeks.

Submission format: Host your solution in a private Git repository on one of the following platforms and add Rouven as a collaborator: IBR GitLab (`kniep`), TU GitLab (`r.kniep`), GitHub (`rkniep`), or Codeberg (`rkniep`). If you want to use a different platform, please ask first.

Before the deadline, send the link to your submission commit by e-mail to `kniep@ibr.cs.tu-bs.de`.

Exercise 1 (Graph Data Structures): **(6+20+4 points)**

The `exercise01` subdirectory for this exercise contains a benchmark setup and an initial implementation for graph data structures that support the following operations:

`construct` Given n, the number of vertices, which will be from 0 to $n-1$, prepare the data structure.

`rebuild` Given an array A of undirected edges, i.e., pairs $\{v, w\}$ of vertices from 0 to $n-1$, erase all edges and insert the ones from A.

`query` Given a pair $\{v, w\}$ of vertices, determine whether it was present in the array A last given to `rebuild`; if it was, return the index of $\{v, w\}$ in A.

`iterate edges` Iterate over all edges, enumerating each undirected edge only once (i.e., only one call for vw and wv), reporting both endpoints and the edge's index in A for each edge.

`iterate neighbors` Given a vertex v, iterate over each edge incident on v, reporting both the neighbor w and the edge's index in A.

The existing benchmarking harness consists of four (sets of) graphs and measures the performance of each of the individual operations (except for construction) as well as the performance of a *combined load*, i.e., a certain combination of operations, across multiple different implementations of such a graph data structure. The graphs are generated differently: graphs 1 are generated uniformly at random (Erdős-Renyi $G(n, p)$ graph), which contains around 8 % of all possible edges, graphs 2 are the same but with fewer vertices and 30 % of possible edges. Graphs 3 have many more vertices but are considerably sparser with constant average degree, and graphs 4 are sparser but have a small subset of high-degree vertices. Unlike graphs 1 and 2, which decide whether to add each edge vw in a systematic order, graphs 3 and 4 generate edges at random until a target edge count is reached.

The existing implementations include structure-of-arrays (SoA) and array-of-structures (AoS) adjacency list implementations, using simple linear search to answer queries, adjacency lists in compressed sparse row (CSR) format using linear search to answer queries, sorted CSRs using binary search to answer queries, as well as a CSR that has a (replaceable) additional lookup data structure for queries; currently, the only version uses a `std::unordered_map` to answer queries.

The repository also contains a plot (in `benchmark-results.pdf`) that was produced by running the benchmark.

a) Why might the performance of the sorted CSR for iterating over all edges be significantly better than the other approaches on graphs 3 and 4? Write your answer in a Markdown file called `sorted_csr_analysis.md` in the `exercise01` directory.

b) Our goal is now to improve the performance of the data structure, in particular considering the `combined_load` benchmark.

   Since the best-looking approach in our benchmark seems to be binary search based, a colleague suggested further improving this approach. His idea was to take the CSR, but instead of sorting all adjacency lists, he proposes to only sort adjacency lists that have a length above some constant K. He then proposed to also bring the long, sorted lists into a more search-friendly layout, namely the so-called Eytzinger layout[1] or an implicit B-tree layout[2]. To achieve the memory alignments that improve the performance of these approaches, he suggests that a suitable number of *blank* entries can be inserted into the CSR before long adjacency lists to ensure they are aligned as required. Since this is only done for long lists, the memory overhead should be small; he considers this tweak to be optional. Whenever a query $\{u, v\}$ comes in, if the shorter of the two neighbor lists for u and v is below K, a linear search is performed on the shorter list; otherwise, a binary search is performed, making use of its memory layout.

   Implement one of the two memory layouts; you are free to choose the one you prefer. Add it to the benchmark, and use the benchmark to try to find a good value for K; you can filter the benchmarks that are actually executed by passing a regex like `--benchmark_filter=.*combined_load.*`.

c) Another colleague thinks the approach from b) is interesting, but way too much effort and ultimately the wrong approach. Upon seeing the usage of `std::unordered_map` with boost's hash function for pairs, he screams *hell, no!* and leaves the room, suggesting to use `boost::unordered_flat_map` as a drop-in replacement instead.

   Do as your colleague says and try the replacement hash table; add it to the benchmark. Which of the approaches performs best?

Exercise 2 (Scalable TSP Heuristic): (20 points)

The `exercise02` subdirectory contains a C++/Python project for the Traveling Salesman Problem (TSP). Given a set of 2D points, your task is to implement a heuristic in `_core.cpp` that computes a short tour — a closed cycle visiting every point exactly once.

[1] `https://algorithmica.org/en/eytzinger`
[2] `https://algorithmica.org/en/b-tree`

The project includes four benchmark instances from well-known TSP libraries (TSPLIB, Waterloo), ranging from 25 000 to 500 000 points. All instances are geometric: points lie in the Euclidean plane. For each instance, a baseline tour length is provided. Your implementation must produce a tour at least as good as the baseline within the time limit.

The test suite (`python -m pytest tests/ -v -x -s`) tests instances in order of increasing size and stops at the first failure. Points are awarded per instance:

Instance	**Points (n)**	**Baseline**	**Points**
`sw24978` (Sweden)	24 978	1 074 991	2
`ch71009` (China)	71 009	5 636 267	2
`mona-lisa100K`	100 000	6 886 143	4
`lra498378` (VLSI)	498 378	2 700 652	12

The baselines are not particularly tight — a straightforward approach with the right data structure will pass. The best-known tour lengths for these instances are significantly shorter; closing that gap requires more sophisticated techniques but is not required. If your tour significantly closes the gap to the best-known solution, the test script will let you know.

Think about which data structures from the lecture are suited for efficiently finding nearby points in 2D. An $O(n^2)$ approach will be too slow for the larger instances.