Exercise 2 (20)

submission date: **21. Nov (23:59)**

This exercise includes a number of graphs[1], represented by lists of edges. You can download them with the included script, or manually from `https://algo2.iti.kit.edu/download/efficient-parallel-cpp-ex2.zip`. Use the provided function to read them into your program.

Important: Do not add the data files to your repository, or you will quickly exceed your quota.

2.1 Breadth-First Search (11)

Even for the simplest graph algorithms, such as Breadth-First Search (BFS), quality of implementation matters. Your task is to implement a variety of graph representations, as well as BFS, efficiently, and evaluate their performance. For this task, you should disregard the edge weights.

Note: Keep the interface of the graph implementations and the BFSHelper intact.

a) Naive Explicit Graph (1)

Implement a graph using explicit, linked node objects by following the structure given in `node_graph.hpp`. You are allowed (and required) to add members, but must keep the pointer-based representation.

If you use a consistent interface, such as indicated in the class, you will have an easier time implementing a BFS that works for all your graph representations.

b) Adjacency List (2)

Implement a graph using an adjacency list. Use a vector-of-vectors, as given in `adj_list.hpp`.

c) Adjacency Array (2)

Implement a graph using an adjacency array. Use two vectors, as given in `adj_array.hpp`. One vector should contain all the edges, ordered by the tail vertex. The other vector should contain the index of the first edge for each vertex.

[1] Taken from the 10th DIMACS Challenge (`https://www.cc.gatech.edu/dimacs10/downloads.shtml`)

d) Breadth-First Search (2)

Write a breadth-first search for your graphs use the structure predefined in `bfs.hpp`. Specifically, given a source s and a destination t, return the distance between s and t using a BFS. Do not use any speed-up techniques or preprocessing (beyond possibly allocating and initializing memory), but do try to be as efficient as possible.

e) Evaluation (4)

Evaluate your implementations, and create meaningful plots, e.g., comparing the graph representations, or to show the improvement of significant optimizations you made. Here are some ideas:

- Create a number of queries (it's okay to simply select source and destination at random, but make sure that your "random" selection is reproducible), then run those queries with each of the three graph representations.
- Measure each graph more than once, and look at mean and standard deviation.
- Try using different data types, e.g., 64 vs. 32 bits, where applicable (this should be fairly easy by turning your classes into templates). If you use templates to do this, please set reasonable default parameters, such that we can instantiate the class without setting parameters.
- Also look at construction performance: How long does it take to build the graph from the list of edges? Can this be improved?

2.2 Dijkstra (9)

In this second task, you will use weighted graphs, and replace BFS with Dijsktra's shortest path algorithm.

a) Graph Representations (3)

Choose your best graph implementation and adapt it to use weighted edges. Try two variants: One where edge data is stored together (e.g., a `struct` containing both head and weight of the edge), and one where edge data is stored component-wise (e.g., two parallel vectors, one containing heads and one

containing weights of edges). Write your implementations in `weighted_graph_paired.hpp` and `weighted_graph_separated.hpp` respectively.

b) Dijsktra (3)

Implement Dijsktra's shortest path algorithm (into `dijkstra.hpp`). As with the BFS, you only need to compute the distance between s and t, not the actual path. You may use your priority queue from the first assignment, or you may use the one provided[2].

c) Evaluation (3)

Evaluate your implementations.

[2] After the deadline of exercise 1 has passed.