Task 5: Jarnik-Prim on Sample-MST with Filter

Lucas Alber, Noah Wahl

August 2, 2023

1 Implementation

Every step of our implementation was carefully evaluated using micro benchmarks. The time measurement code can be found in our git history, we removed it for our final benchmarks. The following subsections present our findings.

1.1 Sampling

Problem: We receive undirected edges as directed edges in both directions from the graph generator. However, the sampling would be far easier if these edges were only present in one direction, to prevent accidentally sampling duplicates.

Solution: We went through multiple sampling strategies:

1. `std::mt19937`: Select an edge `e if e.tail < e.head` with probability $p = n/\sqrt{n \times m} = \sqrt{n/m}$. This turned out to be very slow because branch miss-predictions on these conditions are very common and we have to iterate over the whole edge list.
2. `std::mt19937`: Select $\sqrt{n \times m}$ edges `edge_list[x_i]` where `x_i` is sampled uniformly from `[0, edge_list.size() - 1]`. This way we may sample edges twice, but we found this to not diminish the quality of the sample.
3. `XORShift128`: Same as 2. but with a faster random number generator. We could not detect any compromise in quality.
4. Deterministic Sampling: Select every `edge_list.size() /` $\sqrt{n \times m}$-th edge from the edge list. This was more than an order of magnitude faster than relying on random number generators at no disadvantage to the quality of the sample. So we settled for this strategy in our final implementation.

1.2 Graph Representation

We use single undirected edges where possible and only add reverse edges during the adjacency array buildup for the Jarnik-Prim subroutine. The edges in the adjacency array are reduced to weight-head pairs instead of carrying the tail as well.

1.3 Range Maximum Queries (RMQ)

We expand the levels to the next power of two allowing for efficient shift operations instead of multiplications. This also eliminates special cases in the buildup. We also flattened the levels into one single array.

To speed up the filter loop condition for including edges between components, we set the weight of the root node in each component to inf. In this regard we deviate from the paper where the weight of the root node is set to zero. This way, queries across boundaries return inf and the condition `e.weight < query(e)` evaluates to true.

1.4 Jarnik-Prim

Our Jarnik-Prim implementation is very straight forward. We use a specialized version for computing the minimum spanning forest for our sampled edges and combine data that is accessed together to minimize cache misses. For edge priorities we use an indexed priority queue implementation which turned out to be faster than `std::priority_queue`, especially for dense graphs.

1.5 Filter Loop

The filter loop turned out to be the main bottleneck because we need to check the filter condition for each edge. However, through the aforementioned techniques we reduced the number of computations to a minimum. Additionally, we split the query on the range maximum query data structure to allow for short-circuit evaluation because in this case cache misses (on the RMQ data structure) were more expensive than branch miss-predictions.

1.6 Exploit Simple Instances

Finally, our algorithm decides based on the expected runtime of Jarnik-Prim and IMaxFilter whether to run only a single Jarnik-Prim iteration or run the full IMaxFilter algorithm which can provide a speedup of two for small instances.

1.7 Parallel IMaxFilter

We added a simple parallel implementation of our IMaxFilter that uses OpenMP to speed up the filter loop.

2 Testing

To test our data structures and algorithms we added tests using the Google Test Framework.

3 Experiments

3.1 Setup

Our implementation was evaluated on a machine running Arch GNU/Linux 64bit on Linux 5.18.14 with an AMD Ryzen 5800X, 8 Cores (16 Threads) @ 3800 MHz and 4 × 8 GB DDR4 RAM @ 3400 MT/s. We used GCC 12.1.0 to compile the code and did not change any flags.

3.2 Evaluation

For our evaluation we varied the number of nodes, the distribution of weights and the density of the benchmark graphs.

Figure 1 shows the time per edge in relation to the number of nodes with a double-log scale. The Jarnik-Prim-based algorithms take more time per edge with an increasing number of nodes while the implementation of Kruskal's algorithm remains nearly constant (nearly, because the Union-Find data structure depends on the number of nodes). We observed a similar behavior for all other configurations and therefore omit these plots here. They can be found in the `/plots` directory.

Comparing Figure 3 and 5 points out the effects of increasing the range of different weights. Because Kruskal's algorithm sorts the edges on these weights, it experiences a noticeable slowdown when the range of weights is larger. The Jarnik-Prim-based algorithms do not sort the edges by weights and therefore do not suffer from this.

Lastly, we compared different graph densities by raising the average degree. All figures present a similar picture. For the numbers in this section we refer to Figure 3. The IMaxFilter algorithm was intended to be used on dense graphs and our evaluation shows that in these configurations it really does outperform the other algorithms. Because of memory limitations we could only evaluate high density graphs with up to 2^{16} nodes. Additionally, the graph generator was not able to produce high density graphs without running for a very long time. Therefore, our maximum evaluated density is $\frac{2^{13}2^{11}}{2^{13}(2^{13}-1)} \approx \frac{1}{4}$ for the graphs with the fewest nodes. All Figures show a similar picture: For very small densities (≤ 0.0005) Kruskal performs best, but then Jarnik-Prim-based algorithms pull ahead very quickly. Once IMaxFilter switches (see Section 1.6) from one simple Jarnik-Prim to the sampling algorithm it outperforms the traditional Jarnik-Prim algorithm. Our naive parallel implementation is up to an additional factor of 2 faster than our optimized sequential version for dense graphs.

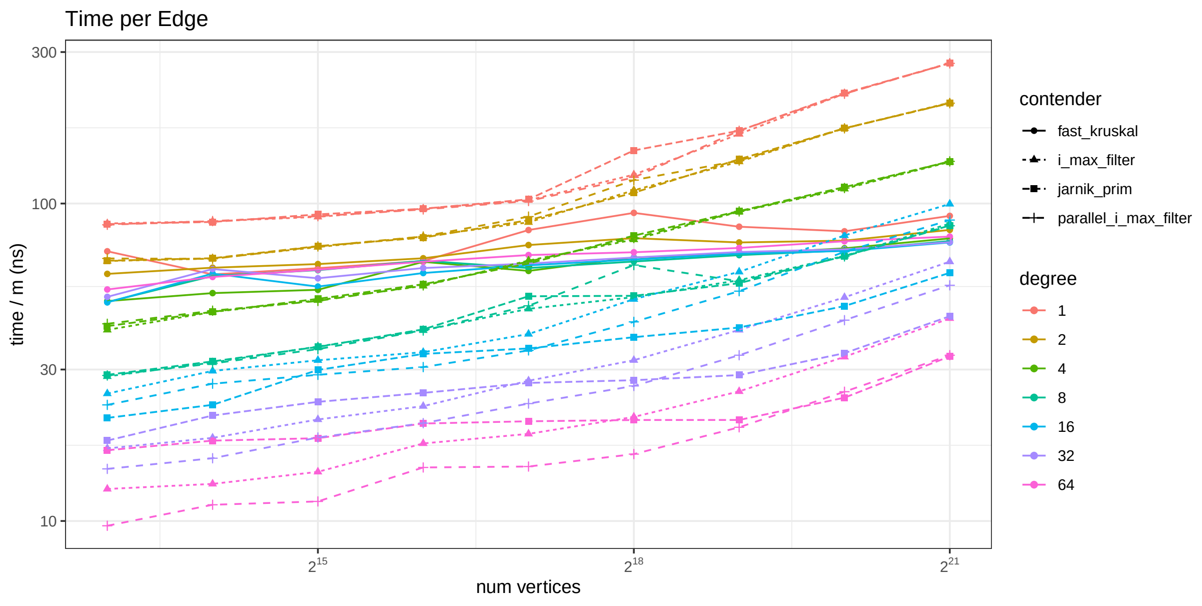

Figure 1: Time per edge for 2^{13} to 2^{21} vertices with an average degree of up to 64 and edge weights up to 2147483647

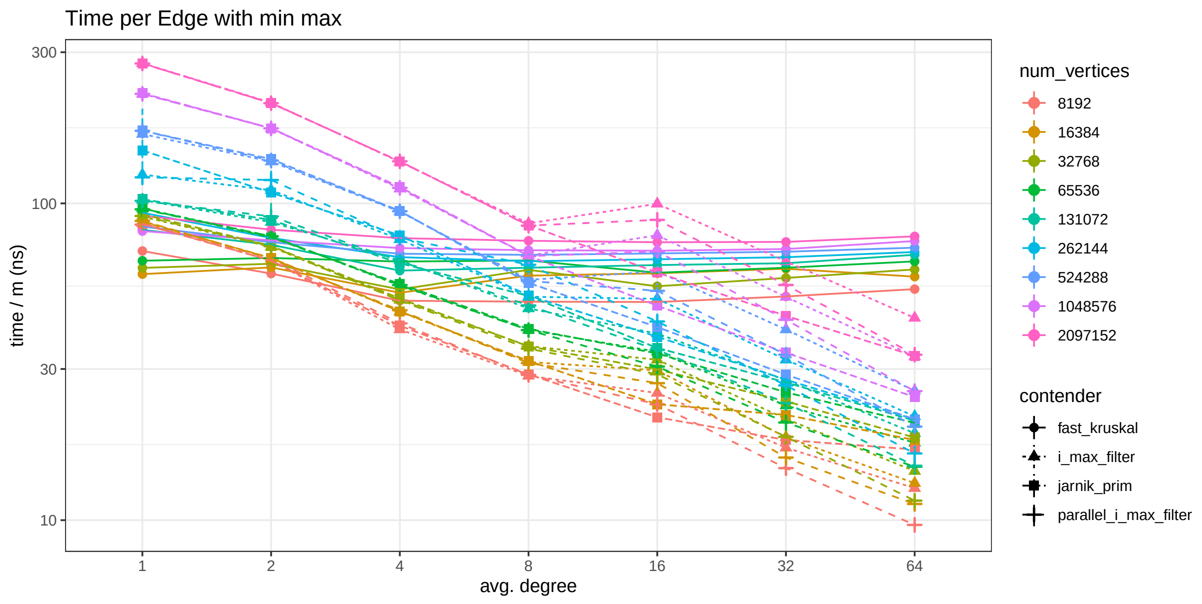

Figure 2: Time per edge for an average degree of up to 64 with 2^{13} to 2^{21} vertices and edge weights up to 2147483647

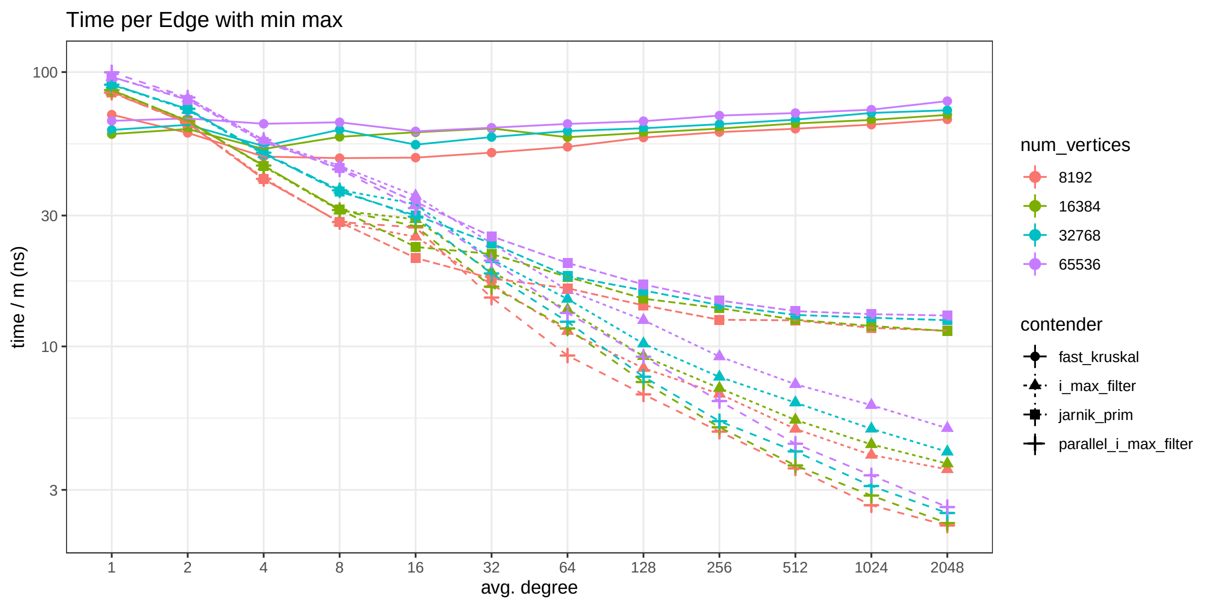

Figure 3: Time per edge for an average degree of up to 2048 with 2^{13} to 2^{16} vertices and edge weights up to 2147483647

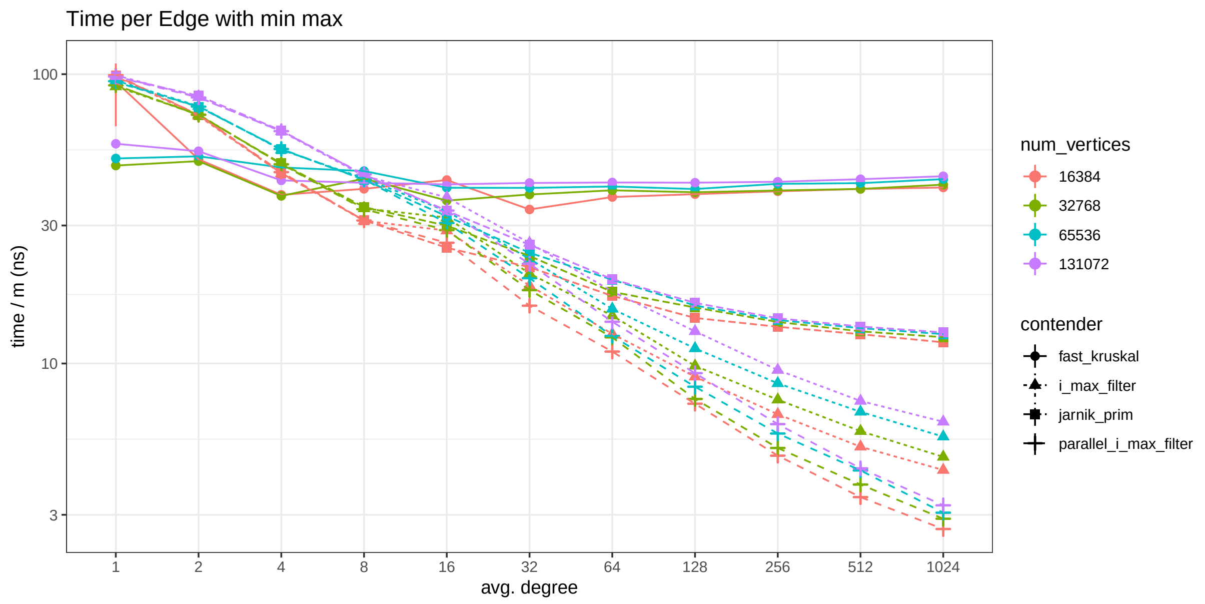

Figure 4: Time per edge for an average degree of up to 1024 with 2^{14} to 2^{17} vertices and edge weights up to 255

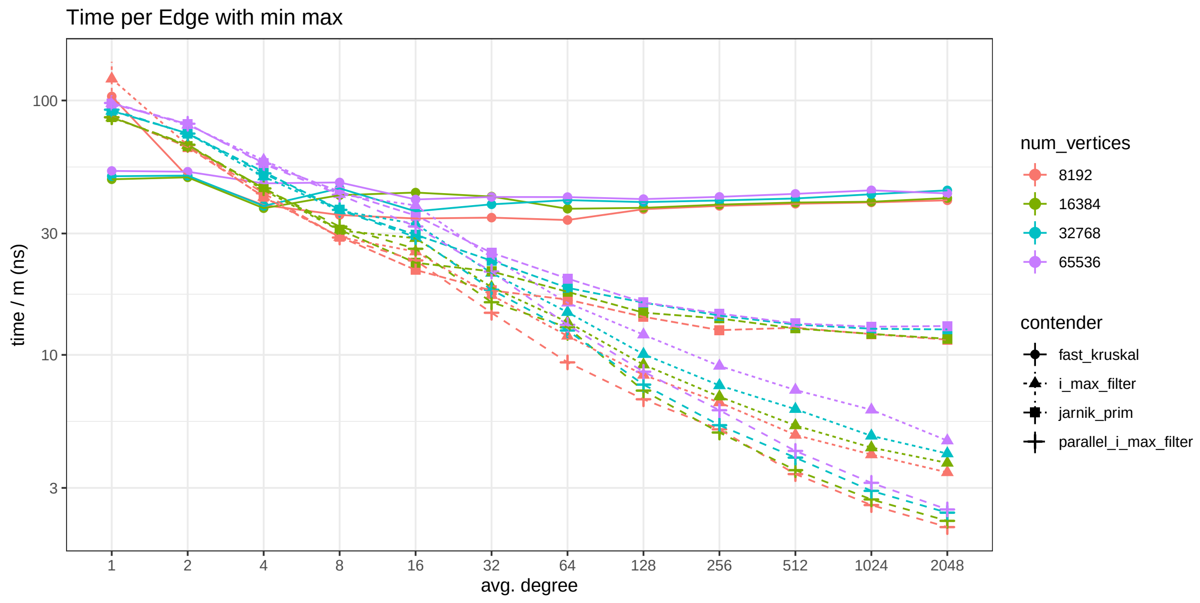

Figure 5: Time per edge for an average degree of up to 2048 with 2^{13} to 2^{16} vertices and edge weights up to 255