Algorithmic aspects of vertex elimination on graphs

Advanced Algorithms project - A.Y. 2020/21

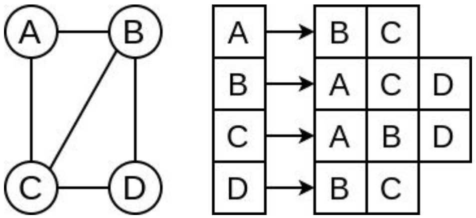
July 23, 2021

Marco Bonelli
10522665

Data structures

I used the [Boost Graph Library] to create a generic templated implementation of the algorithms that can work on different implementations of graphs.

Graphs: `boost::adjacency_list<vecS, vecS, undirectedS>`



This is the Graph implementation that I used, a vector of vectors: fast iteration over vertices and neighbors of a vertex.

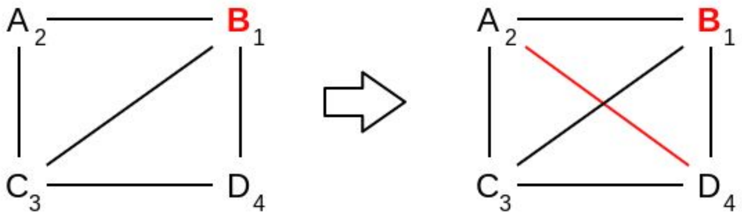
Vertex orderings: `std::vector<Graph::vertex_descriptor>`

Vector of vertex descriptors, which for `adjacency_list<vecS, vecS, …>` simply means `size_t`.

FILL algorithm

Computes the **chordal completion** of any simple, connected, undirected graph given an elimination order for its vertices. Expected complexity: *O(V+E)* time and space.

This means modifying the graph adding all necessary edges belonging to the *fill-in* of the graph. The *fill-in* is the union of the *deficiencies* of each vertex following the elimination order.



FILL algorithm

Define a mapping v ➔ set of successors of v. The successors of a vertex **v** are the vertices adjacent to **v** that come after **v** in the order.

- For each vertex **v** in order:
  - Find **m** as the successor of **v** with minimum index in the order
  - For each other successor **w** of **v** that is not a successor of **m**:
    - Connect **w** and **m** adding the edge (**w**, **m**) to the graph
    - Add **w** to the successors of **m**

FILL algorithm

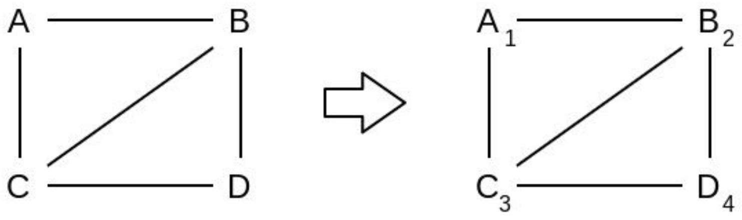
Used data structures:

- `unordered_map<Vertex, Index>` to keep track of the index in the order of each vertex
- `unordered_map<Vertex, unordered_set<Vertex>>` to keep track of the successors of each vertex (this is not necessarily needed but avoids checking all neighbors of each vertex every time)

LEX P algorithm

Computes a **perfect vertex elimination order** for a perfect elimination graph (i.e. a graph for which the existence of such an order is guaranteed, also known as *chordal* graph). Expected complexity: *O(V+E)* time and space.

A perfect elimination order results in an empty fill-in. In other words FILL would add no edges to the graph given this order.



LEX P algorithm

Create a linked list of labels (initially only one label), each label keeps track of which vertices have the label through a set. The head of the linked list has the highest label.

Repeat as many times as the number of vertices:

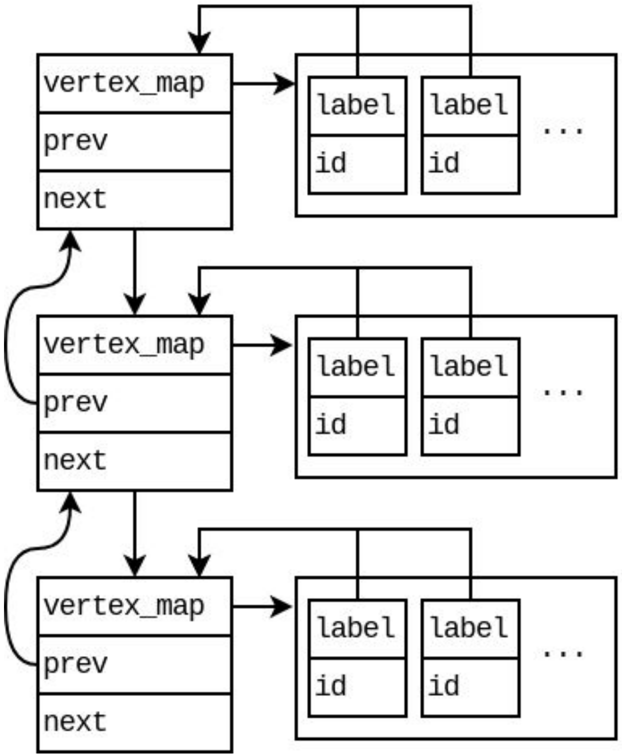
- Get **v** as any of the highest-labeled unnumbered vertices, add it as last in the order and mark it as numbered
- Increase the label of each unnumbered neighbor of **v** by creating a new label and inserting it right before the previous label. Vertices with the same old label need to end up with the same new label (use a map old -> new to keep track of this).

LEX P algorithm

Used data structures:

- `struct Label {vertex_map, prev, next}`
- `struct LabeledVertex {label, id}`
- `unordered_map<Vertex, LabeledVertex*>` for unnumbered vertices

- `unordered_map<Label*, Label*>` to temporary hold newly created labels before insertion



LEX M algorithm

Computes a **minimal vertex elimination order** of any simple, connected, undirected graph. Expected complexity: $O(VE)$ time, $O(V+E)$ space.

A minimal elimination order is such if there is no other elimination order that produces a fill-in that is a strict subset of the one produced by it.

LEX M algorithm

Repeat as many times as the number of vertices:

1. Get **v** as the highest-labeled unnumbered vertex, add it as last in the order and mark it as numbered
2. Increase the label of every unnumbered vertex **w** that is reachable from **v** *through a chain of unnumbered lower-labaled (than **w**) vertices*.

In step 2, the unnumbered vertices are explored in lexicographical breadth-first order of label by keeping one queue of vertices for each different label. Initially queues only hold the neighbors of **v**, and are then processed until empty in increasing label order.

LEX M algorithm

`Label`s are simply even unsigned integers from `0` to `2*n_unique_labels - 1`. Increasing a label (done at most once) means `+1`. To keep unnumbered vertices sorted according to their label a radix sort algorithm (LSD base 16) is used at the end of each step of the main loop, and then vertices are re-labeled to even numbers again.

Used data structures:

- `unordered_set<Vertex>` for unnumbered vertices and for reached vertices during lexicographic BFS
- `unordered_map<Vertex, Label>` to keep track of the label of each vertex
- `unordered_map<Label, std::deque<Vertex>>` queues for lexicographic BFS

LEX M algorithm

Errors/typos in the implementation of the paper:

- The loop should run in the opposite direction with `j` from `1` to `k`: the graph needs to be explored starting from lower labeled vertices.
- The `if l(z) < k` should be `if l(z) > j`: the label of vertex `z` should only be increased if we can reach it through a chain of lower-labeled vertices.
- All the `k`s that appear inside the loop should be `j`s.

```
search: for j:=k step -1 until 1 do
           while reach(k) ≠ φ do begin
              delete a vertex w from reach(k);
              for z ε adj(w) and z unreached do begin
                 mark z reached;
                 if ℓ(z) < k then begin
                    add z to reach(ℓ(z));
                    ℓ(z):=ℓ(z)+1/2;
                    mark {v,z} as an edge of G*α;
                 end else add z to reach(k);
           end end;
```

Random graph generation

For testing and benchmarking purposes we need to generate:

Random connected graphs (FILL, LEX M)

Use an Erdős–Rényi model to generate a random graph with a given number of vertices and a given probability of having edges between any possible pair of vertices.

Then, identify the connected components of the graph with the help of a union-find and add one edge at a time between two of them until the graph is completely connected.

Random chordal graphs (LEX P)

Implemented as described by the clique-tree algorithm in *Two methods for the generation of chordal graphs (Markenzon, Vernet, Araujo, 2008)*.

This algorithm generates a chordal graph that has a given fixed number of vertices and respects a given upper bound on the number of edges.

Testing

I wrote unit tests using the Boost Test library, gcov + Codecov for code coverage (continuous-integration through GitHub actions).

1. Random graph generation utilities: ensure graphs are simple, connected or chordal as needed, not complete unless needed.
2. Radix sort correctly sorts
3. **FILL**: chordal completion on known small graphs, empty fill-in for complete graphs given any elimination order, empty fill-in for chordal graphs given a perfect elimination order.
4. **LEX M**: ensure computed elimination order is minimal for connected graphs and perfect for chordal graphs.
5. **LEX P**: ensure computed elimination order is perfect for chordal graphs.

Testing

```
test/unit/test_lex_p.cc(11): Leaving test suite "LexP"; testing time: 256834us
test/unit/test_radix_sort.cc(10): Entering test suite "RadixSort"
test/unit/test_radix_sort.cc(22): Entering test case "radix_sort_works<unsigned char>"
test/unit/test_radix_sort.cc(22): Leaving test case "radix_sort_works<unsigned char>"; testing time: 2386us
test/unit/test_radix_sort.cc(22): Entering test case "radix_sort_works<unsigned short>"
test/unit/test_radix_sort.cc(22): Leaving test case "radix_sort_works<unsigned short>"; testing time: 2726us
test/unit/test_radix_sort.cc(22): Entering test case "radix_sort_works<unsigned int>"
test/unit/test_radix_sort.cc(22): Leaving test case "radix_sort_works<unsigned int>"; testing time: 3443us
test/unit/test_radix_sort.cc(22): Entering test case "radix_sort_works<unsigned long>"
test/unit/test_radix_sort.cc(22): Leaving test case "radix_sort_works<unsigned long>"; testing time: 4772us
test/unit/test_radix_sort.cc(10): Leaving test suite "RadixSort"; testing time: 13361us
test/unit/test_fill.cc(22): Entering test suite "Fill"
test/unit/test_fill.cc(52): Entering test case "known_graph"
test/unit/test_fill.cc(52): Leaving test case "known_graph"; testing time: 788us
test/unit/test_fill.cc(82): Entering test case "complete_graph_has_empty_fill_in_for_any_order"
test/unit/test_fill.cc(82): Leaving test case "complete_graph_has_empty_fill_in_for_any_order"; testing time: 2387882us
test/unit/test_fill.cc(110): Entering test case "complete_graph_has_empty_fill_in"
test/unit/test_fill.cc(110): Leaving test case "complete_graph_has_empty_fill_in"; testing time: 1056263us
test/unit/test_fill.cc(128): Entering test case "chordal_graph_has_empty_fill_in"
test/unit/test_fill.cc(128): Leaving test case "chordal_graph_has_empty_fill_in"; testing time: 69356us
test/unit/test_fill.cc(22): Leaving test suite "Fill"; testing time: 3514354us
Leaving test module "Main"; testing time: 32541260us

Test module "Main" has passed with:
  21 test cases out of 21 passed
  277273 assertions out of 277273 passed
```

Timing benchmarks

Done through the Google Benchmark library, results plotted through Python3 + matplotlib + scikit-learn.

Benchmarked all algorithms on random connected graphs with different edge densities, linearly increasing the number of vertices. For LEX P: generated a random connected graph, then used LEX M to find a minimal elimination order, then FILL to make it chordal.

Tested edge densities (***d***): {0.1, 0.25, 0.50, 0.66, 0.75, 1.0}.

Tested number of vertices (***V***) for each density: {100, 200, …, 1000}.

Timing benchmarks

```
template <unsigned num, unsigned div>
void lex_m_random_graph(benchmark::State& state) {
   const unsigned v = state.range(0);
   auto g = gen_random_connected_graph<Graph>(v,
(double)num/div);

   for (auto _ : state)
       benchmark::DoNotOptimize(lex_m(g));

   const auto n = boost::num_vertices(g) * boost::num_edges(g);
   state.counters["n"] = n;
   state.counters["v"] = boost::num_vertices(g);
   state.SetComplexityN(n);
}
```

```
BENCHMARK_TEMPLATE(lex_m_random_graph, 1, 10)->DenseRange(100, 1000, 100)->Complexity(benchmark::oN)->Unit(benchmark::kMillisecond);
BENCHMARK_TEMPLATE(lex_m_random_graph, 1,  4)->DenseRange(100, 1000, 100)->Complexity(benchmark::oN)->Unit(benchmark::kMillisecond);
BENCHMARK_TEMPLATE(lex_m_random_graph, 1,  2)->DenseRange(100, 1000, 100)->Complexity(benchmark::oN)->Unit(benchmark::kMillisecond);
BENCHMARK_TEMPLATE(lex_m_random_graph, 2,  3)->DenseRange(100, 1000, 100)->Complexity(benchmark::oN)->Unit(benchmark::kMillisecond);
BENCHMARK_TEMPLATE(lex_m_random_graph, 3,  4)->DenseRange(100, 1000, 100)->Complexity(benchmark::oN)->Unit(benchmark::kMillisecond);
BENCHMARK_TEMPLATE(lex_m_random_graph, 1,  1)->DenseRange(100, 1000, 100)->Complexity(benchmark::oN)->Unit(benchmark::kMillisecond);
```

Timing benchmarks

```
--------------------------------------------------------------------------------------------
Benchmark                                   Time             CPU   Iterations UserCounters...
--------------------------------------------------------------------------------------------
fill_in_random_graph<1, 10>/100         0.584 ms        0.583 ms         1266 n=583 v=100
fill_in_random_graph<1, 10>/200          2.78 ms         2.78 ms          250 n=2.214k v=200
fill_in_random_graph<1, 10>/300          7.12 ms         7.12 ms           99 n=4.788k v=300
fill_in_random_graph<1, 10>/400          15.8 ms         15.8 ms           50 n=8.378k v=400
fill_in_random_graph<1, 10>/500          23.8 ms         23.8 ms           30 n=12.81k v=500
fill_in_random_graph<1, 10>/600          41.9 ms         41.9 ms           16 n=18.457k v=600
fill_in_random_graph<1, 10>/700          65.0 ms         65.0 ms           10 n=25.352k v=700
fill_in_random_graph<1, 10>/800           103 ms          103 ms            7 n=33.084k v=800
fill_in_random_graph<1, 10>/900           134 ms          134 ms            5 n=41.238k v=900
fill_in_random_graph<1, 10>/1000          163 ms          163 ms            4 n=51.079k v=1000
fill_in_random_graph<1, 10>_BigO      3030.58 N       3029.99 N
fill_in_random_graph<1, 10>_RMS            17 %            17 %
fill_in_random_graph<1, 4>/100          0.728 ms        0.728 ms          929 n=1.346k v=100
fill_in_random_graph<1, 4>/200           3.11 ms         3.11 ms          226 n=5.177k v=200
fill_in_random_graph<1, 4>/300           7.41 ms         7.41 ms           99 n=11.588k v=300
fill_in_random_graph<1, 4>/400           13.8 ms         13.8 ms           47 n=20.601k v=400
fill_in_random_graph<1, 4>/500           24.6 ms         24.6 ms           30 n=31.469k v=500
fill_in_random_graph<1, 4>/600           41.9 ms         41.9 ms           13 n=45.535k v=600
fill_in_random_graph<1, 4>/700           58.0 ms         58.0 ms           12 n=61.964k v=700
fill_in_random_graph<1, 4>/800           77.1 ms         77.1 ms            8 n=80.824k v=800
fill_in_random_graph<1, 4>/900            122 ms          122 ms            6 n=102.209k v=900
fill_in_random_graph<1, 4>/1000           137 ms          137 ms            5 n=126.139k v=1000
fill_in_random_graph<1, 4>_BigO       1056.85 N       1056.70 N
fill_in_random_graph<1, 4>_RMS             15 %            15 %
fill_in_random_graph<1, 2>/100          0.732 ms        0.732 ms          887 n=2.56k v=100
```

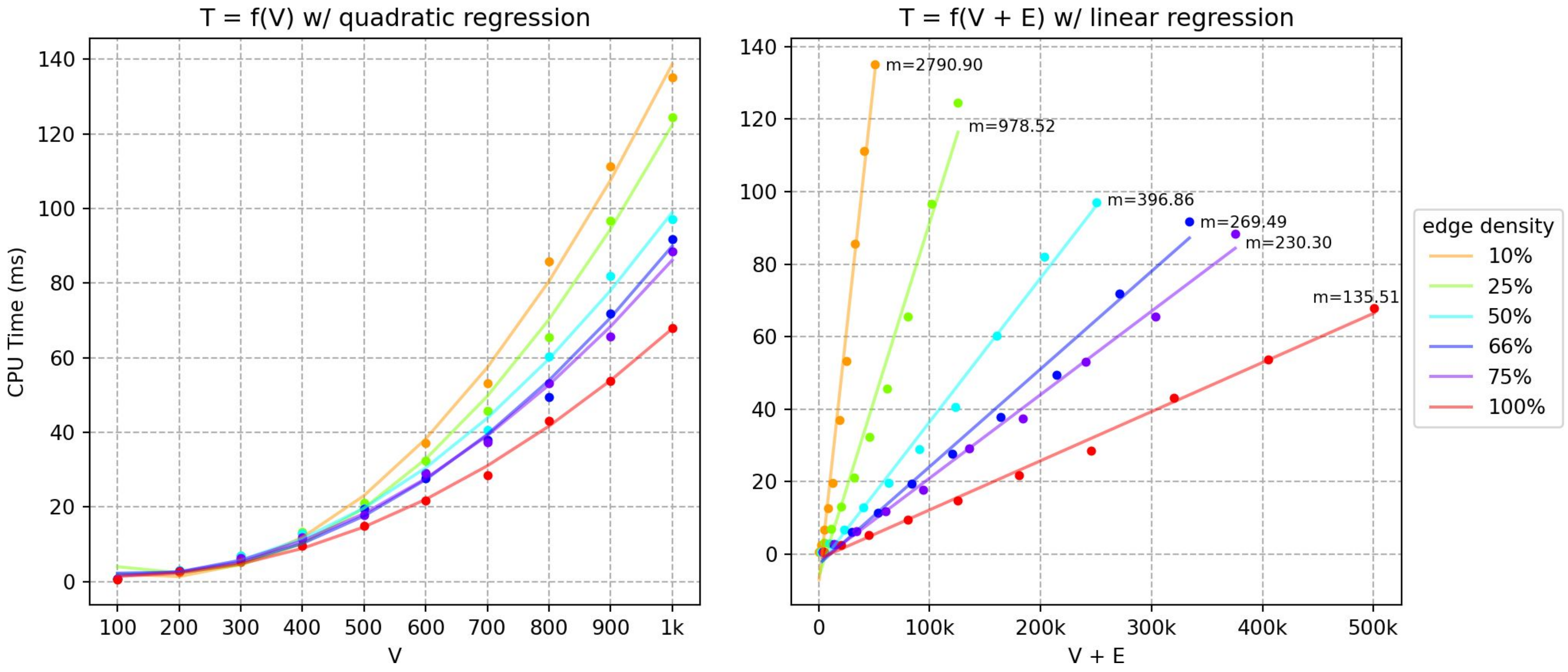
Time complexity

Number of edges (***E***) directly depends on the number of vertices and the edge density (***d***) (i.e. Erdős–Rényi probability parameter): $\boldsymbol{E \approx dV(V-1)/2 \approx dV^2}$.

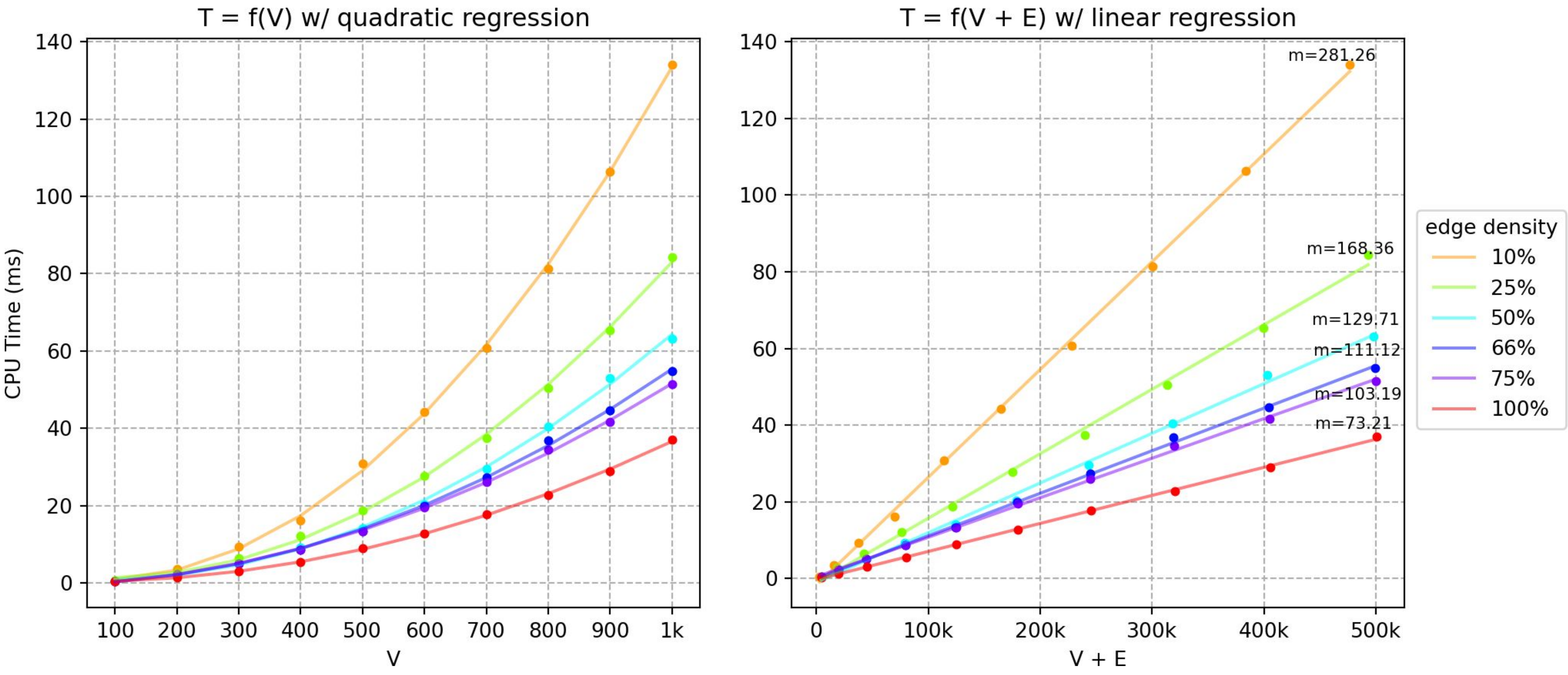
Time complexity:

FILL	$O(V + E) = O(V + dV(V - 1)/2) = O(V + V^2) = O(V^2)$
LEX P	$O(V + E) = O(V + dV(V - 1)/2) = O(V + V^2) = O(V^2)$
LEX M	$O(VE) = O(VdV(V - 1)/2) = O(V^3)$

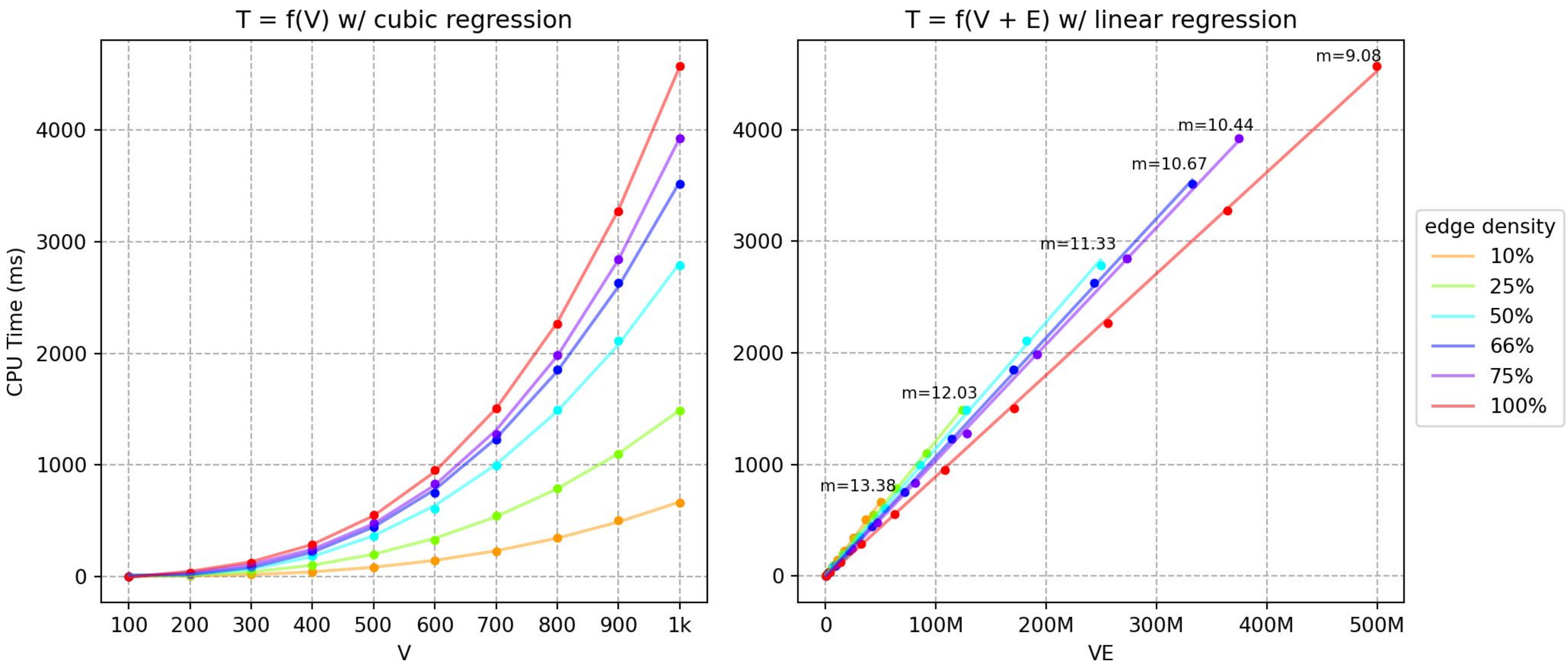
Time complexity: FILL



Time complexity: LEX P



Time complexity: LEX M



Memory benchmarks

Done through glibc's `__{malloc,realloc,free}_hook` in a very simple custom memory profiling library to keep track of maximum heap usage.

```
template <unsigned num, unsigned div>
void fill_random_graph(void) {
   char name[128];

   for (const auto v : vertices) {
     Graph g = gen_random_connected_graph<Graph>(v, (double)num/div);
     const unsigned n = boost::num_vertices(g) + boost::num_edges(g);

     sprintf(name, "fill_random_graph<%u,%u> v=%u n=%u", num, div, v, n);
     start_trace(name);
     do_not_optimize(lex_m(g));
     stop_trace();
   }
}
```

```
static void start_trace(const char *name) {
   __malloc_hook         = malloc_hook;
   __realloc_hook        = realloc_hook;
   __free_hook           = free_hook;
   cur_trace_name        = name;
   allocated_memory      = 0;
   max_allocated_memory = 0;
}

static void stop_trace(void) {
   __malloc_hook  = NULL;
   __realloc_hook = NULL;
   __free_hook    = NULL;

   std::cout << cur_trace_name
      << ": max " << max_allocated_memory
      << " bytes" << std::endl;
}
```

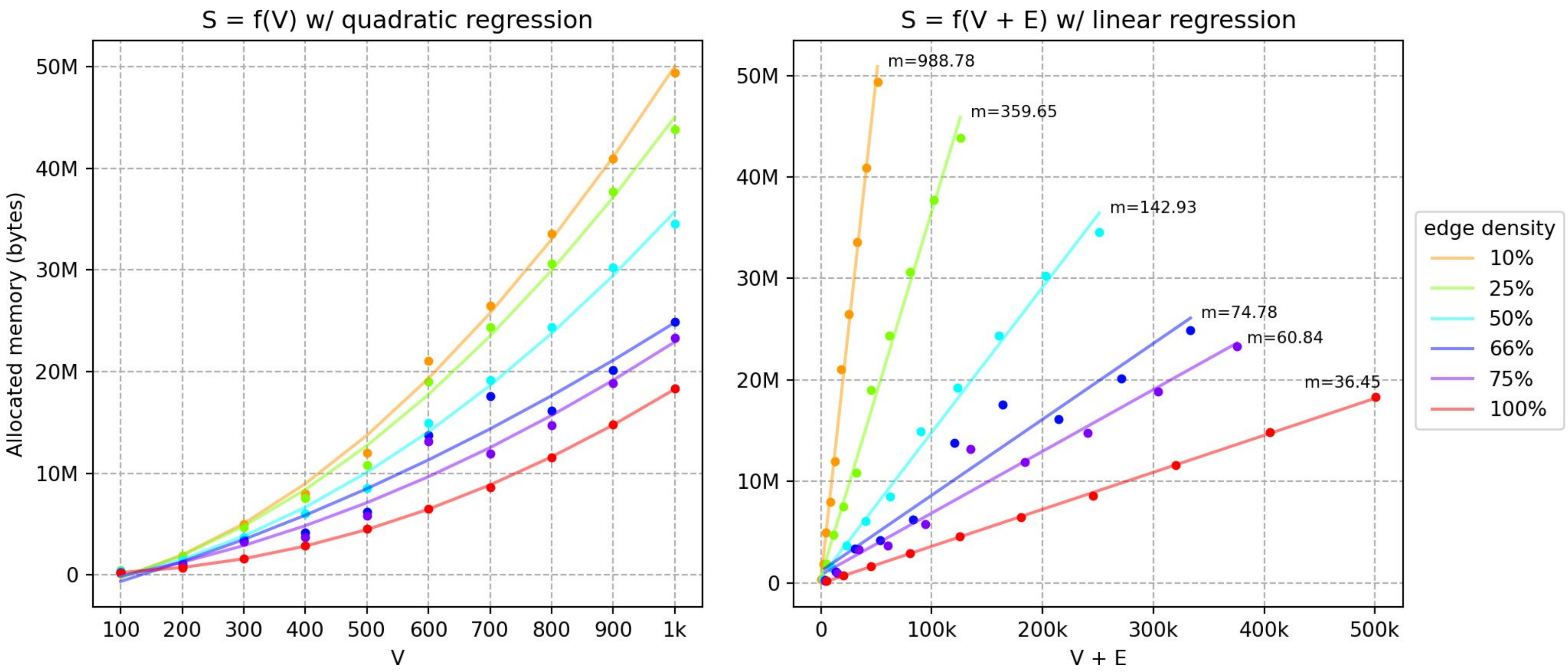
Space complexity

As in timing benchmarks, the Number of edges is still: $\boldsymbol{E \simeq dV(V-1)/2 \simeq dV^2}$.

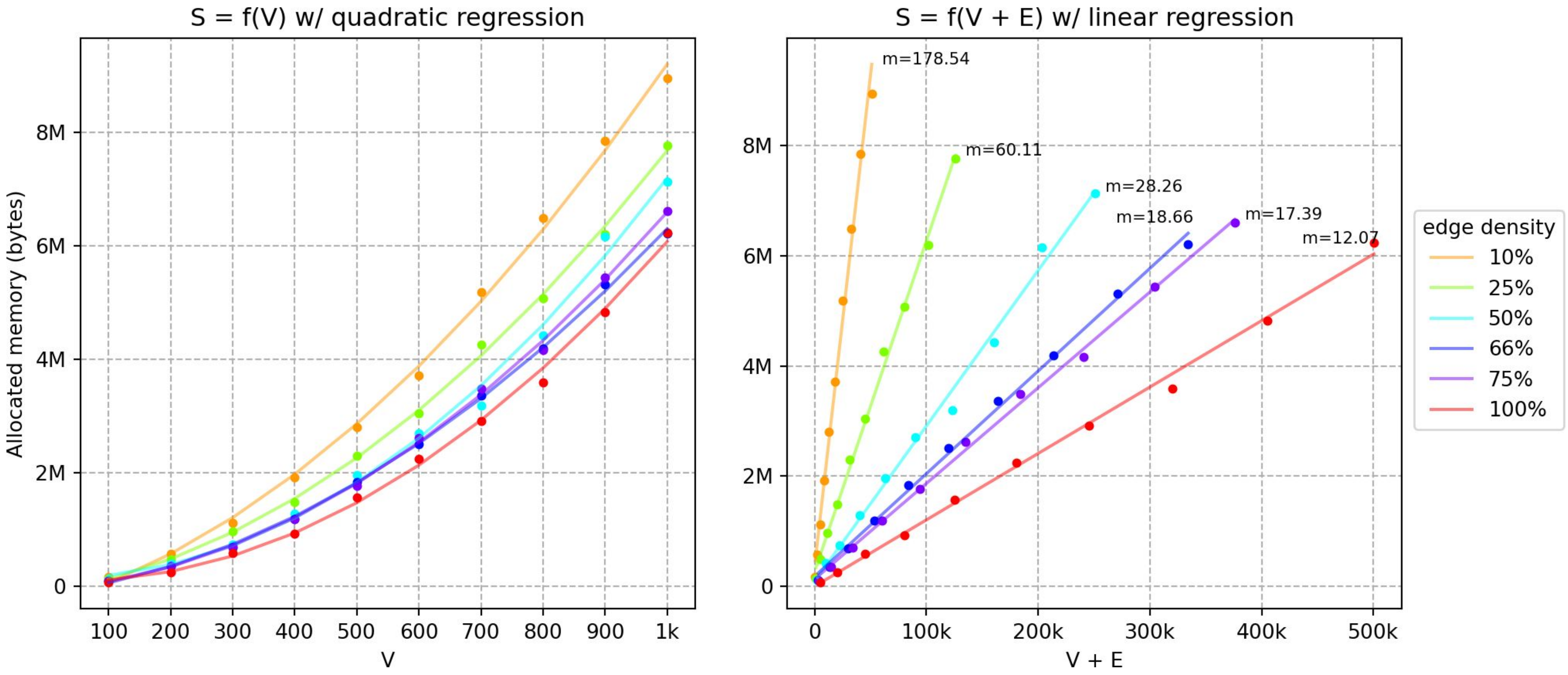
Space complexity (in terms of **additional** space required):

FILL	$O(V + E) = O(V + dV(V - 1)/2) = O(V + V^2) = O(V^2)$
LEX P	$O(V + E) = O(V + dV(V - 1)/2) = O(V + V^2) = O(V^2)$
LEX M	$O(V)$

Space complexity: FILL



Space complexity: LEX P



Space complexity: LEX M