Triangle Listing

Team 02

Outline

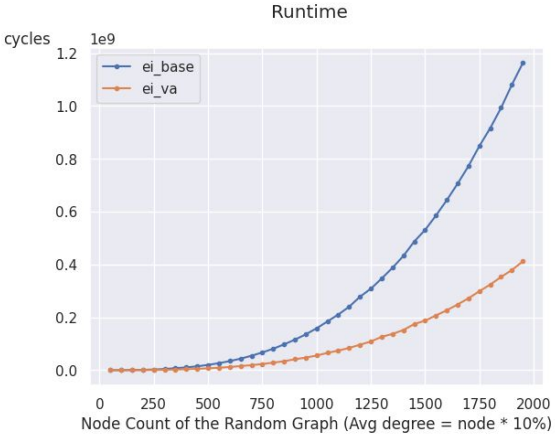
- Optimization by preprocessing the input graph
- Optimizations for sorting
- Optimizations for each algorithms
  - Edge iterator
  - Forward
  - Forward hashed
- Questions

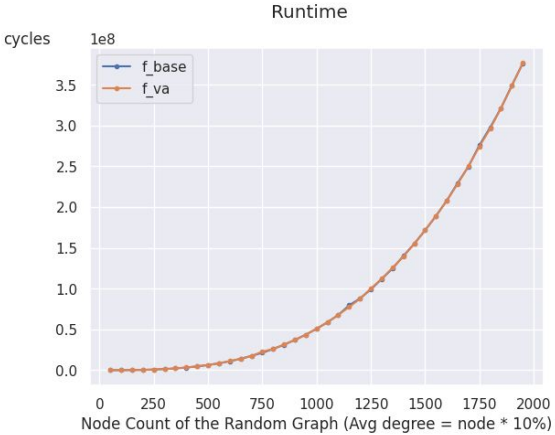
Reduce Branching with Preprocessing

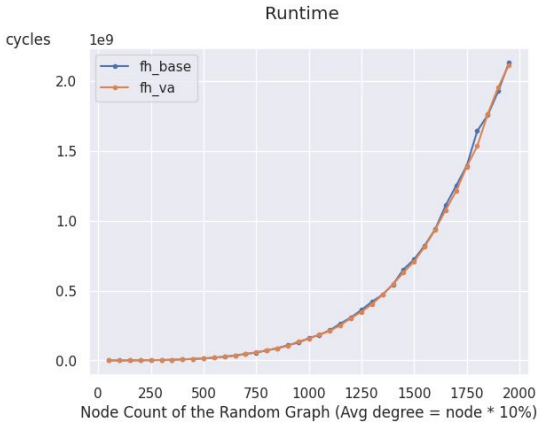
```
for s in G:
    for t in s.neighbor:
        if s < t:
            set intersection
```

- The input graph is undirected, but the algorithms only consider s → t where s < t.
- All three algorithms check s < t on the fly before set intersection to avoid duplicate triangles.
- Problem: redundant branching
  - Edge iterator: Same s < t comparison done multiple times because the result is not saved.
  - All three algos: Sorting already compares the nodes.
- Solution: embed s < t comparison in the pre-processing step
  - Cutting: do a (quicksort) partition on each adjacency list and ignore the smaller neighbors
- Benefits:
  - Remove all s < t branchings in the algorithm core ⇒ enable further optimizations
  - Reduce iterations through smaller neighbors in the adjacency list
  - Reduce sorting time
  - Don’t need extra space

Cutting vs No-Cutting: Results on Dense Graphs

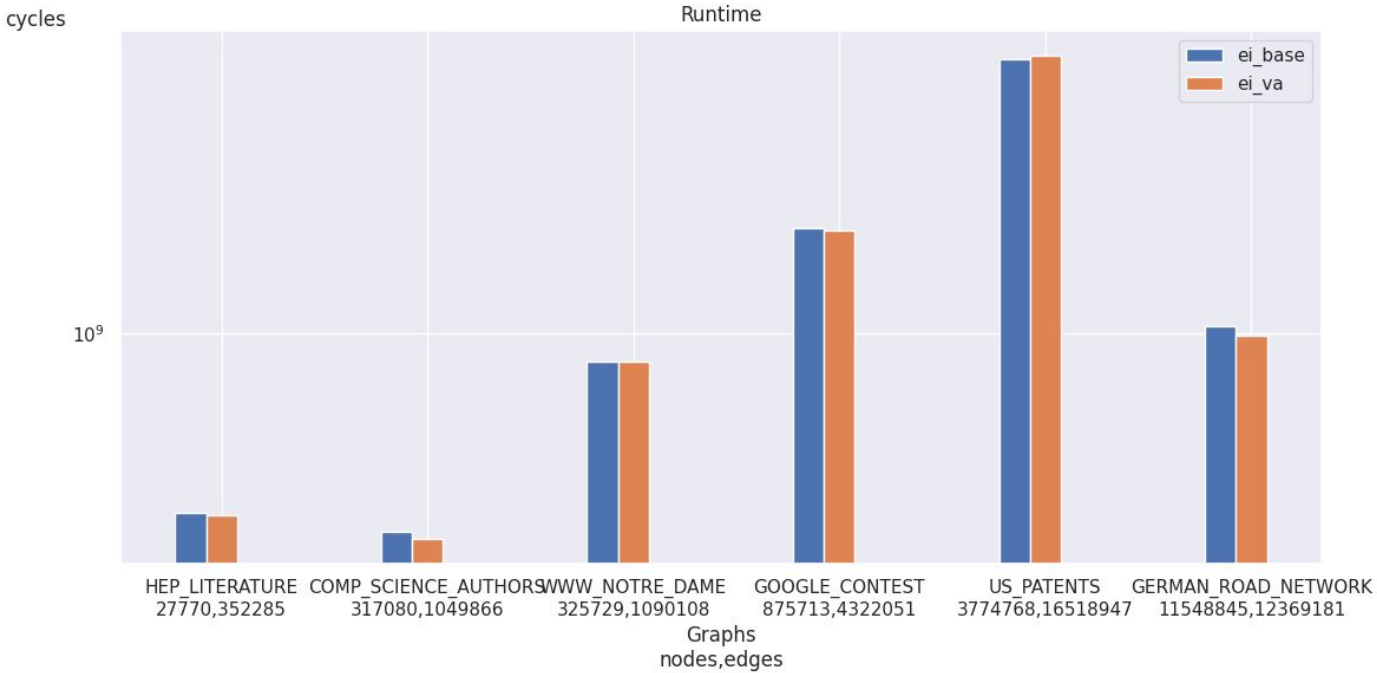






Cutting vs No-Cutting: Runtime on Real World Graphs

Other algorithms show similar results.



Sorting

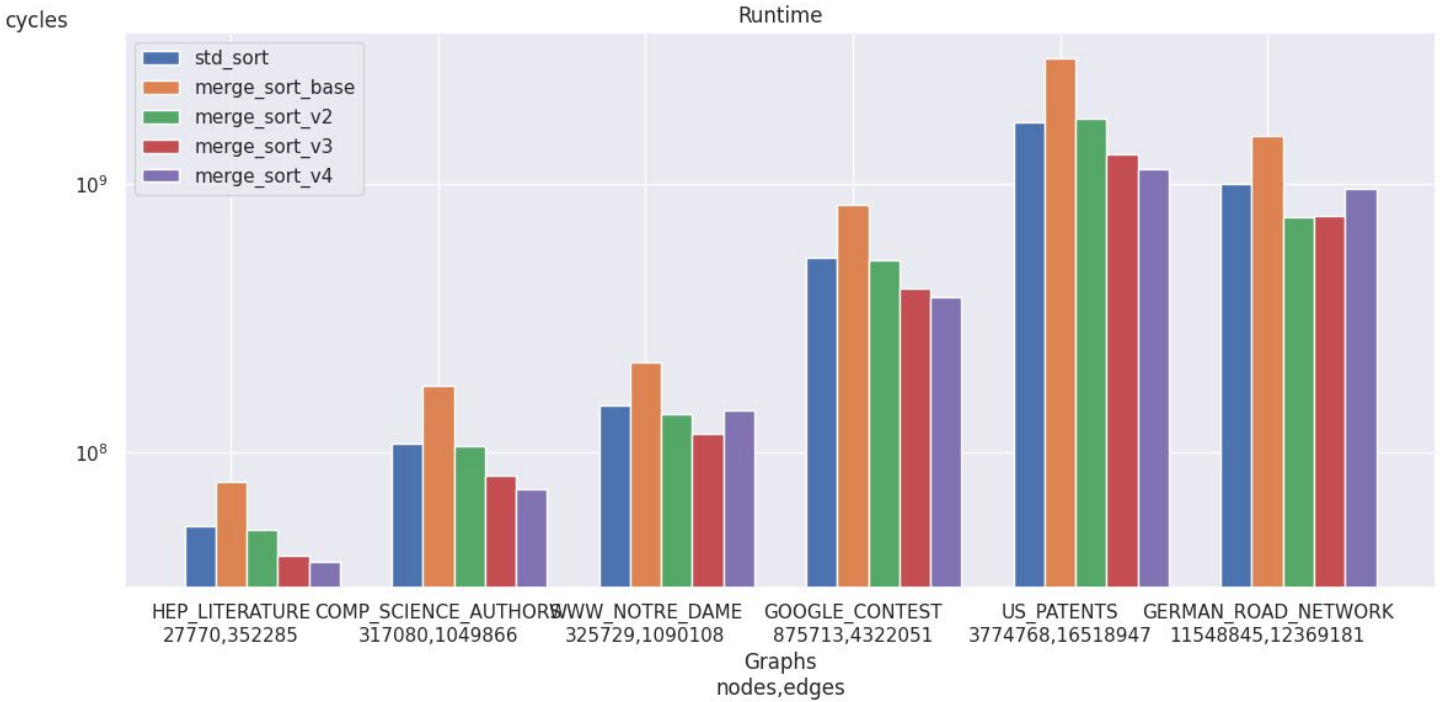
Quick Sort vs Merge Sort:

- Optimisation Potential

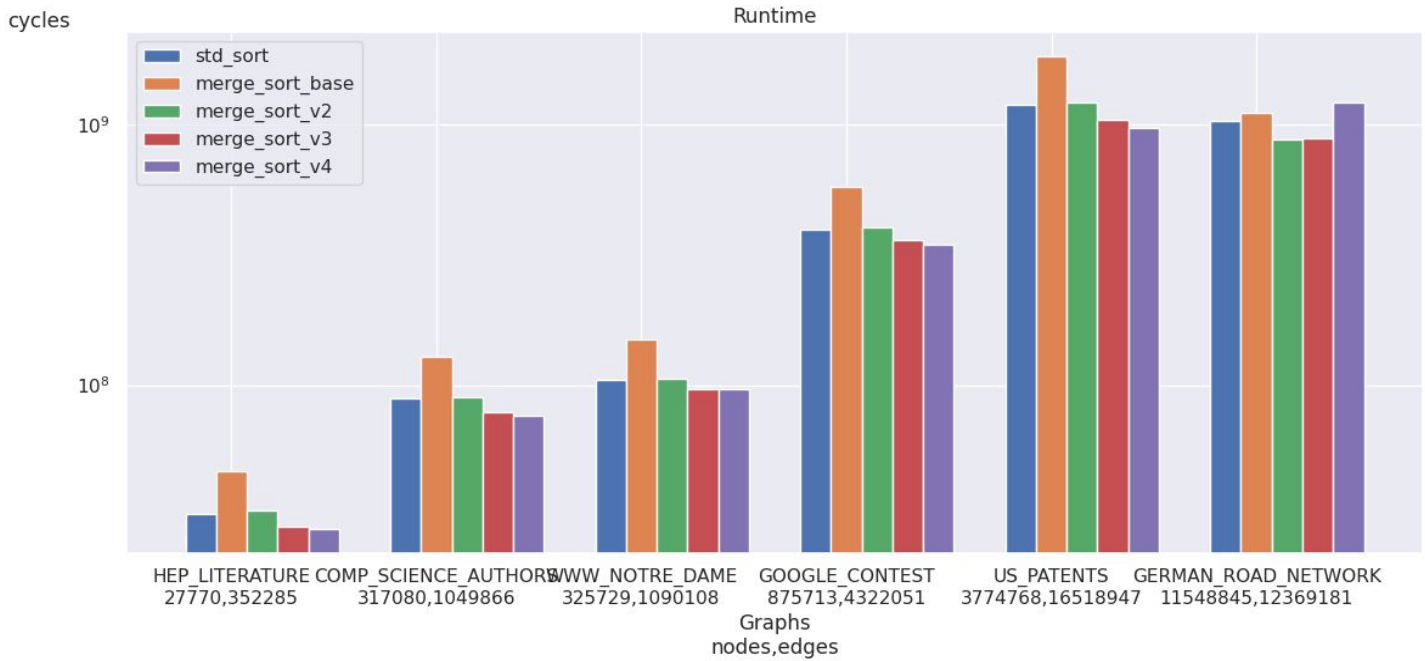
Merge Sort:

- Base Implementation
- Add insertion sort for small lists
- Add Vec Sort with min/ max and shuffle operations (Changes allocation).
- Add insertion sort for small lists again.

Sorting + No-Cutting: Runtime on Real World Graphs

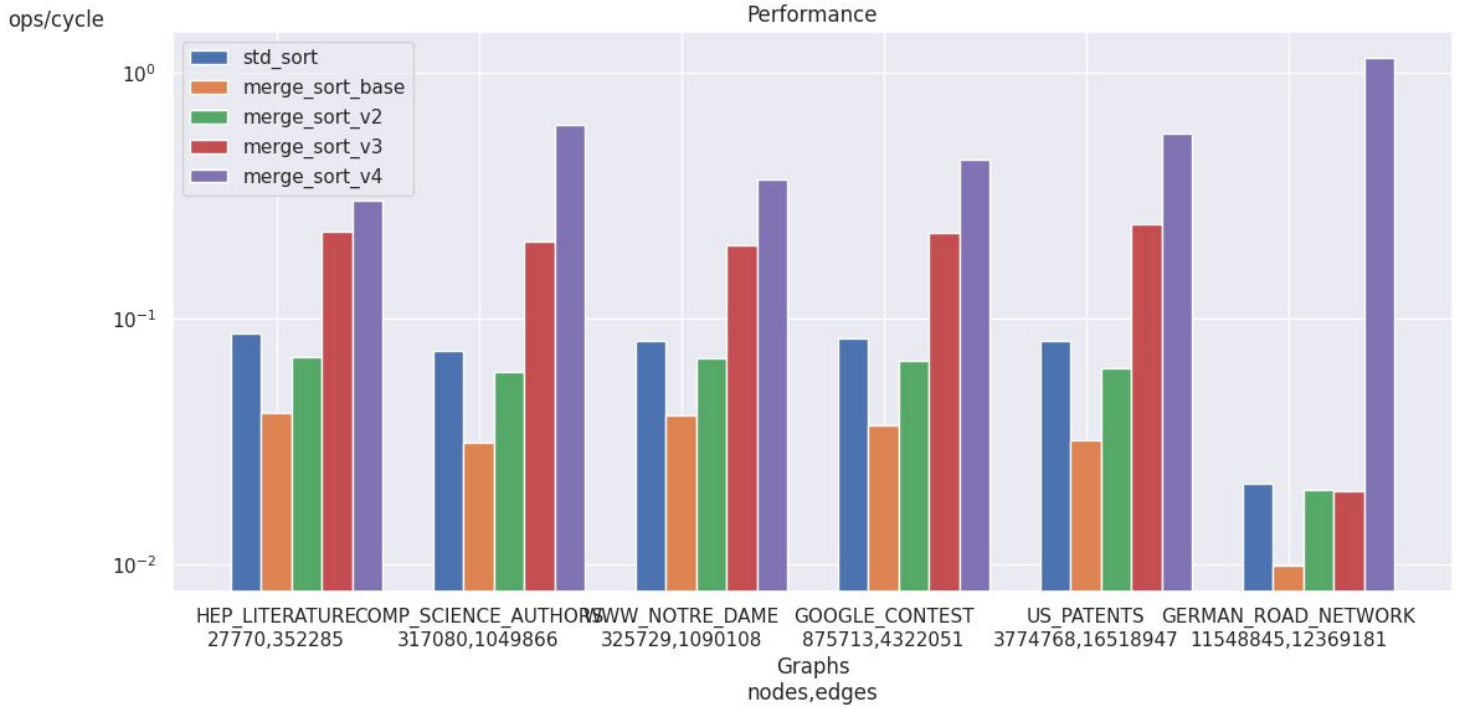


Sorting + Cutting: Runtime on Real World Graphs



Sorting + No-Cutting: Performance on Real World Graphs

With cutting shows similar results.



Optimisations Edge Iterator: Set Intersection

Problem: Iterate through both adjacency lists with hard-to-predict branches

The base implementation of the set intersection has to perform a smaller/greater-or-equal comparison whenever the currently compared elements don't match. As this is the common scenario many hard to predict branches need to be evaluated. Additionally the intersected, sorted lists are often of very different size leaving room for optimisations.

- V1: Exponential search on larger list (10x Difference)
  - Fewer ops when many elements can be skipped
- V2: Unrolling + difficult branch elimination on similar-sized lists
  - More ops and easy to predict == comparisons
  - Fewer difficult to predict <= comparisons
- V2: Eliminate t < u check that ensures triangle s-t-u -> s < t < u
  - Binary search the starting point for intersected lists to skip useless elements
  - Removes branch condition and overall number of ops

Optimisations Edge Iterator: Set Intersection

Problem: Iterate through both adjacency lists with hard to predict branches

The base implementation of the set intersection has to perform a smaller/greate-or-equal comparison whenever the currently compared elements don’t match. As this is the common scenario many hard to predict branches need to be evaluated. Additionally the intersected, sorted lists are often of very different size leaving room for optimisations.

- V3 (in work): vectorize == comparison
- U4: Remove hard to predict branches by incrementing using booleans
  - A different approach to solving the issue of many branches is to increment pointers using the boolean value of the comparison result instead of branching
  - Reduces branches
  - Not clear how/if possible to vectorize

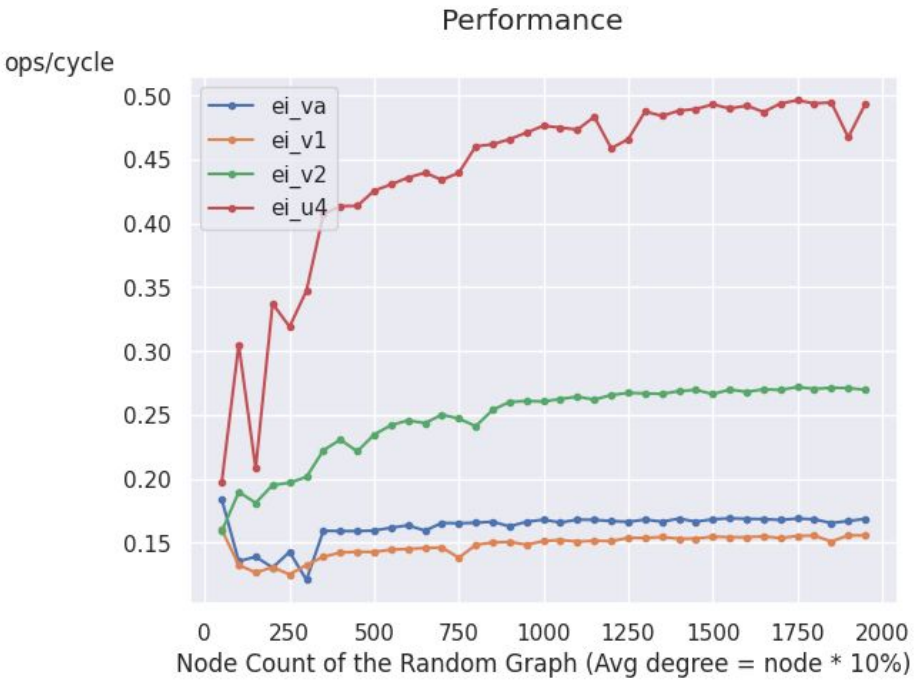
Optimisations Edge Iterator: Unrolling + Vectorization

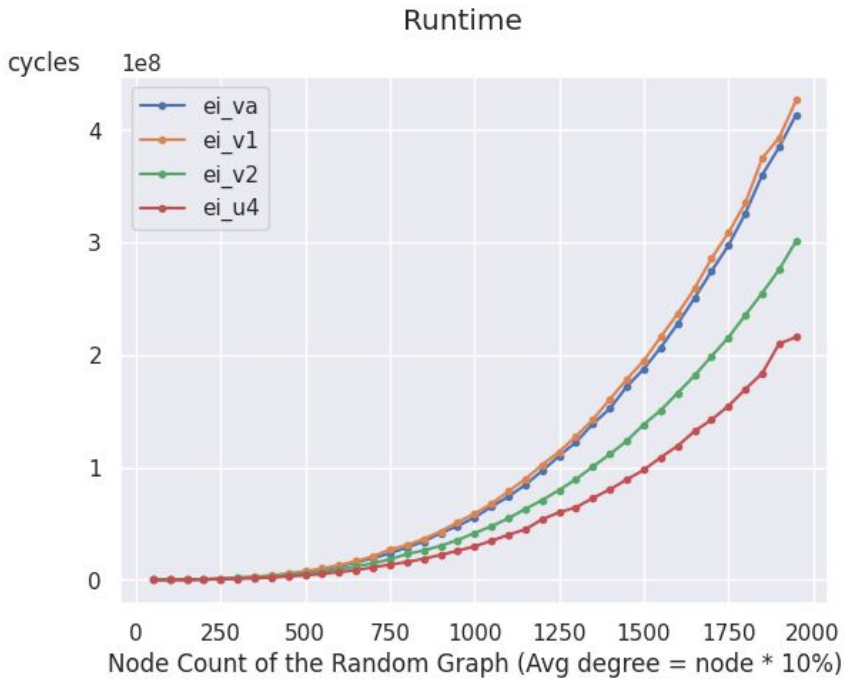
Problem: Set intersections are independent; can be interleaved/vectorized

For a given node s, we intersect its neighbourhood with every neighbourhood of every neighbour. These intersections are completely independent and only their results need to be accumulated in a shared list of triangles. However, as in comparison the number of triangles is usually low, the dependency is minimal

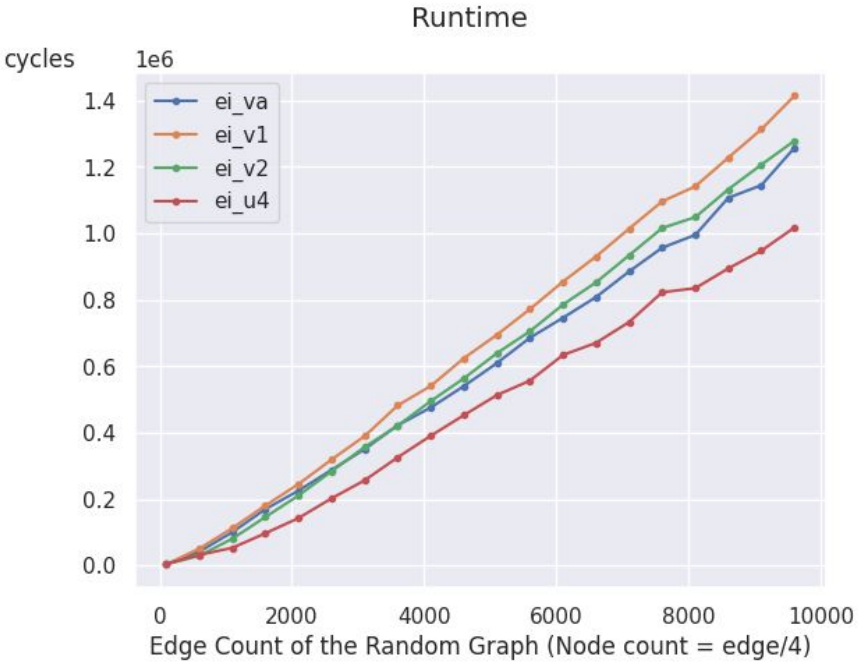
- U4: unroll the outer-loop to increase ILP
  - The iterations are largely independent, which is why we don’t expect much of an improvement, but it is a necessary preparation for vectorization
- U5 (in works): vectorize multiple set-intersection as in forward

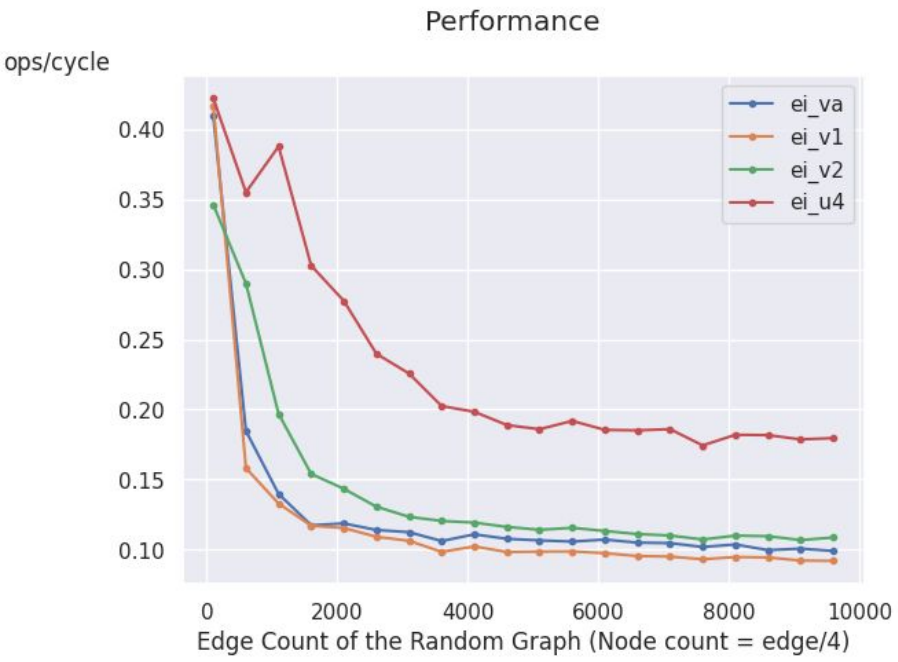
Edge Iterator: Result on Dense Graphs



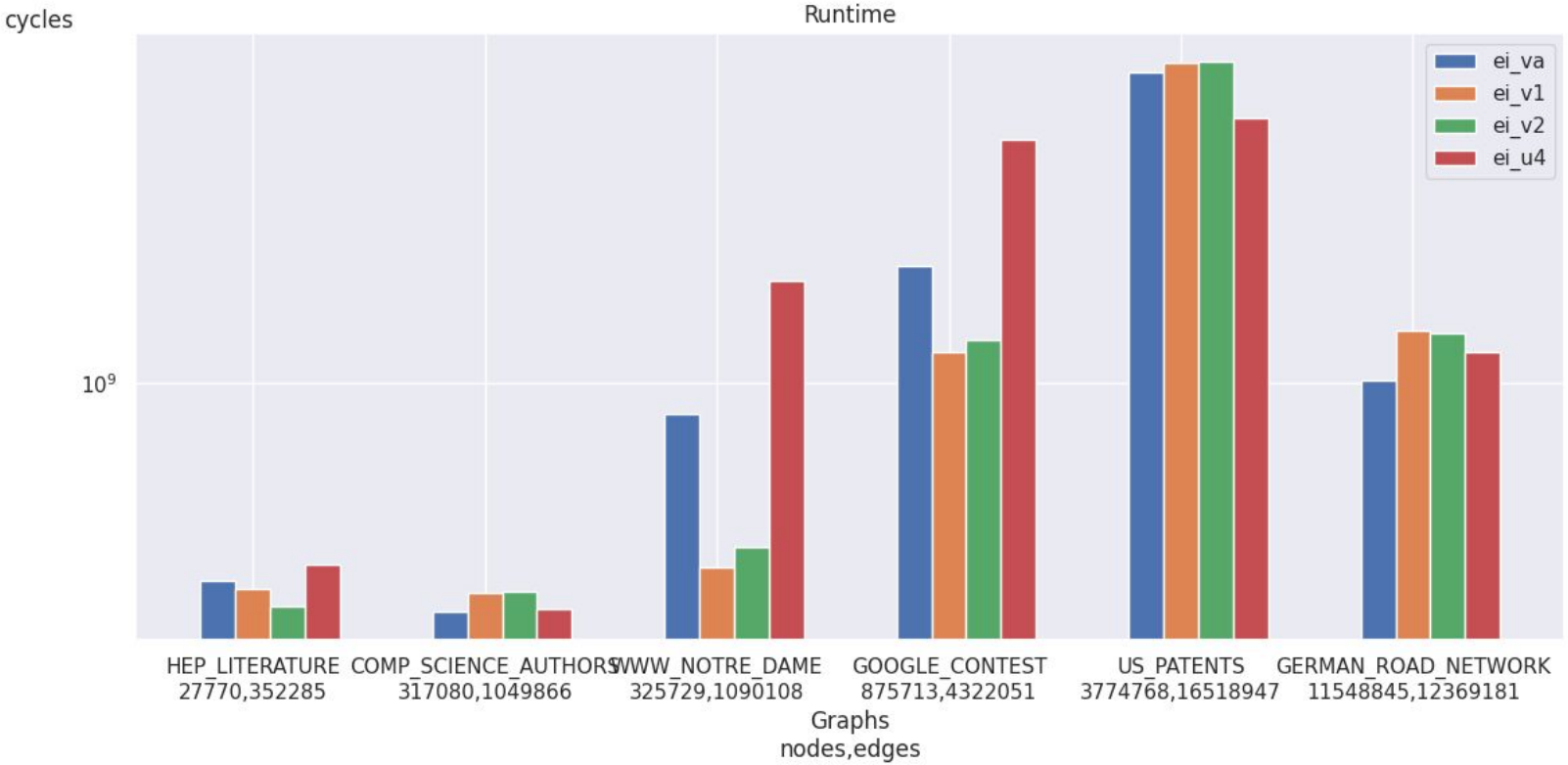


Edge Iterator: Result on Sparse Graphs

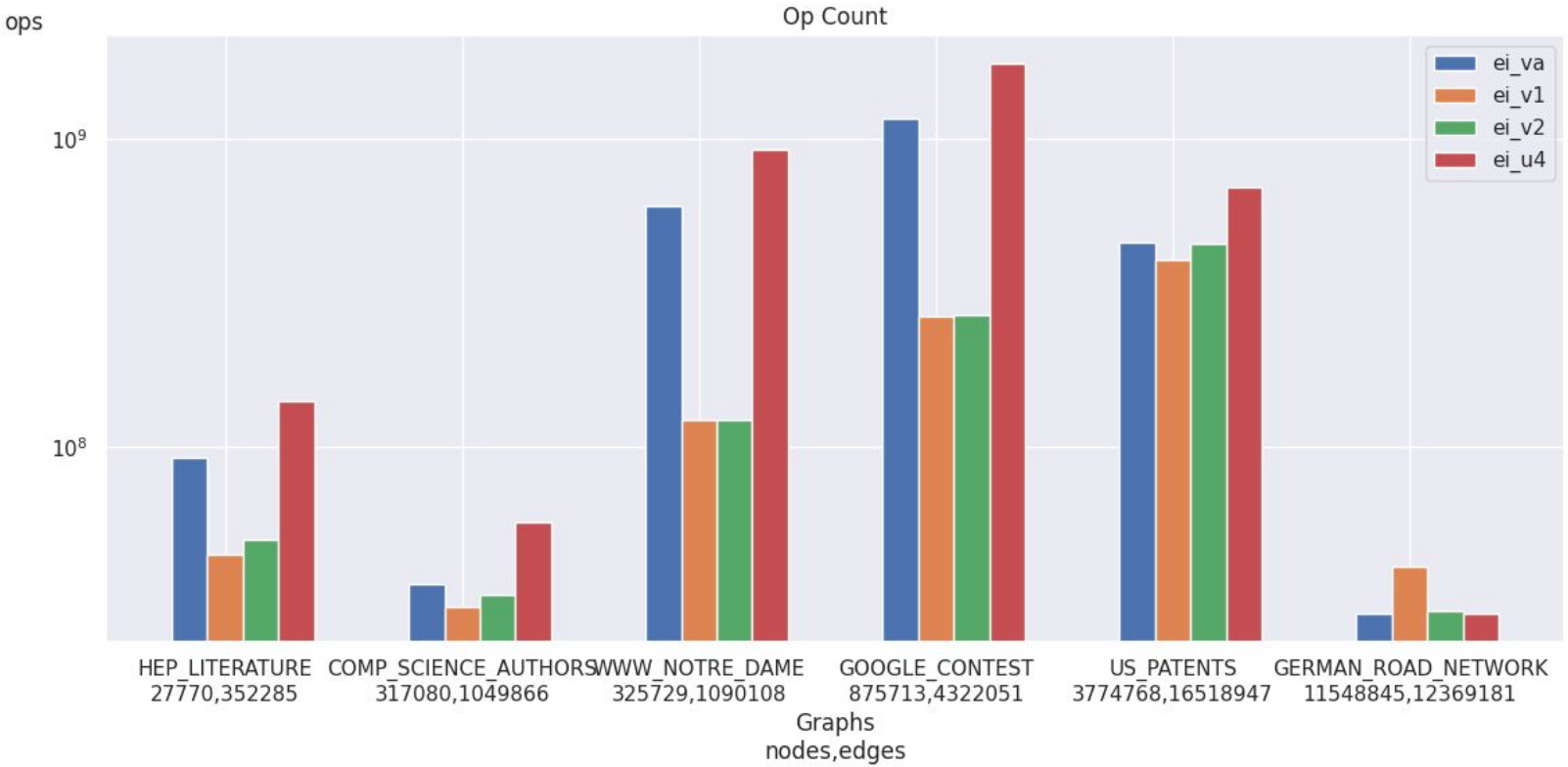




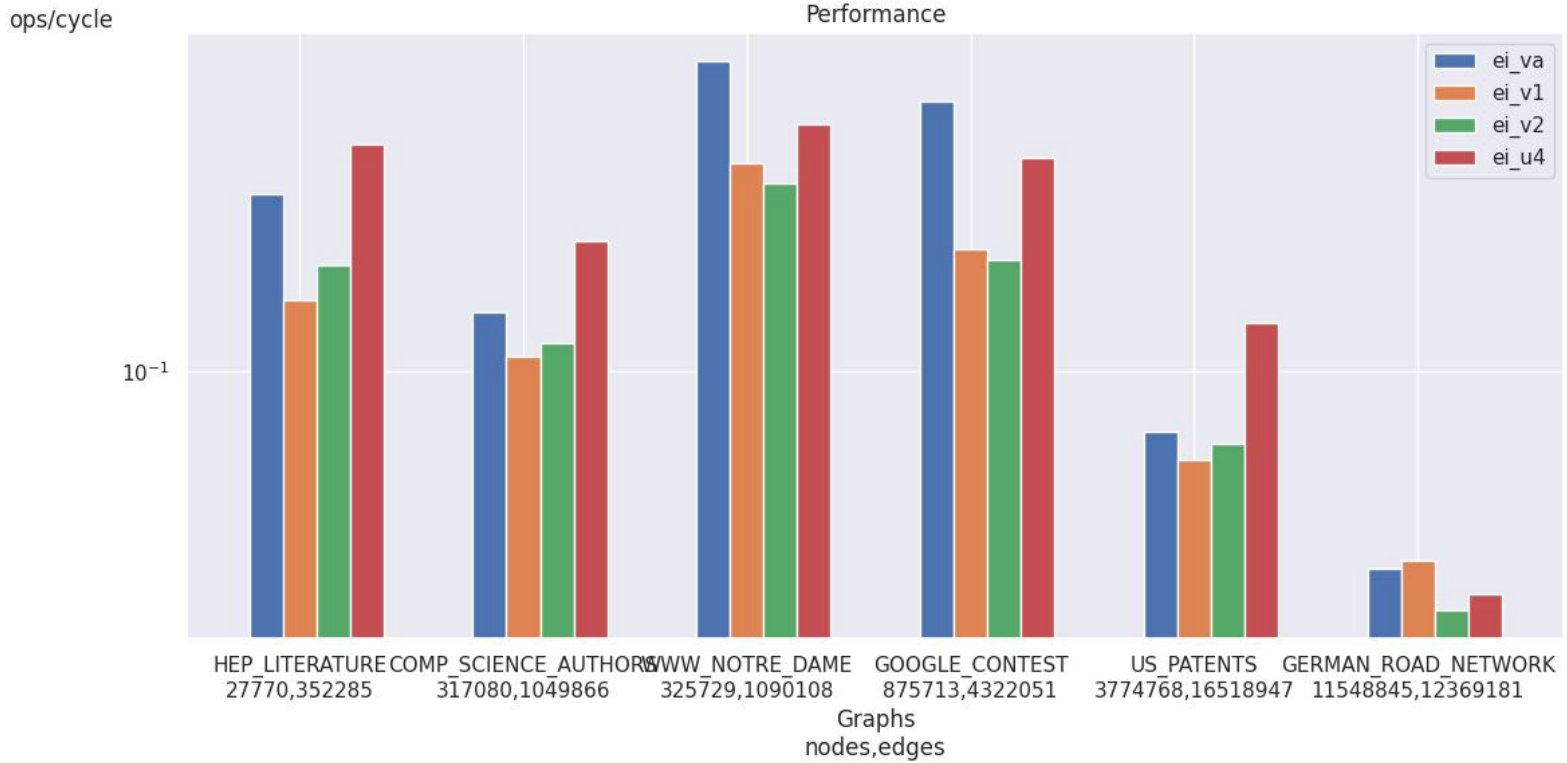
Edge Iterator: Runtime on Real World Graphs



Edge Iterator: Ops Real World Graphs



Edge Iterator: Performance on Real World Graphs



Forward Baseline

```
for s:
    for t:
        while loop for set intersection
            .. .. ..
        add s to A_t
```

Forward V1

```
for s:
    for multiple ts:
        while loop for set intersection
            .. ..  All branching removed
                   Doing for intersection in one loop body (ILP)
        add s to A_t
```

Forward V2 & V3 & V4

```
for s:
    for multiple ts:
        while loop for set intersection
            .. ..  Same As before but vectorized
        add s to A_t
```

- Vectorized with size 4
- Vectorized with size 8
- Vectorized with size 8 + 8
- All of these optimizations can also be applied to Edge Iterator.

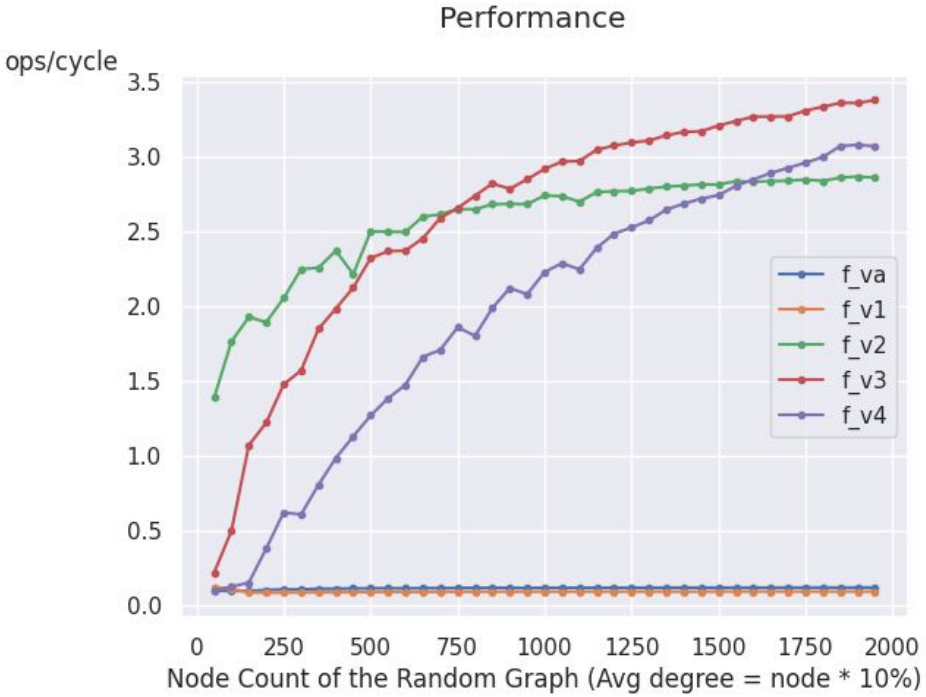
Optimisations Forward: Unrolling + Vectorization

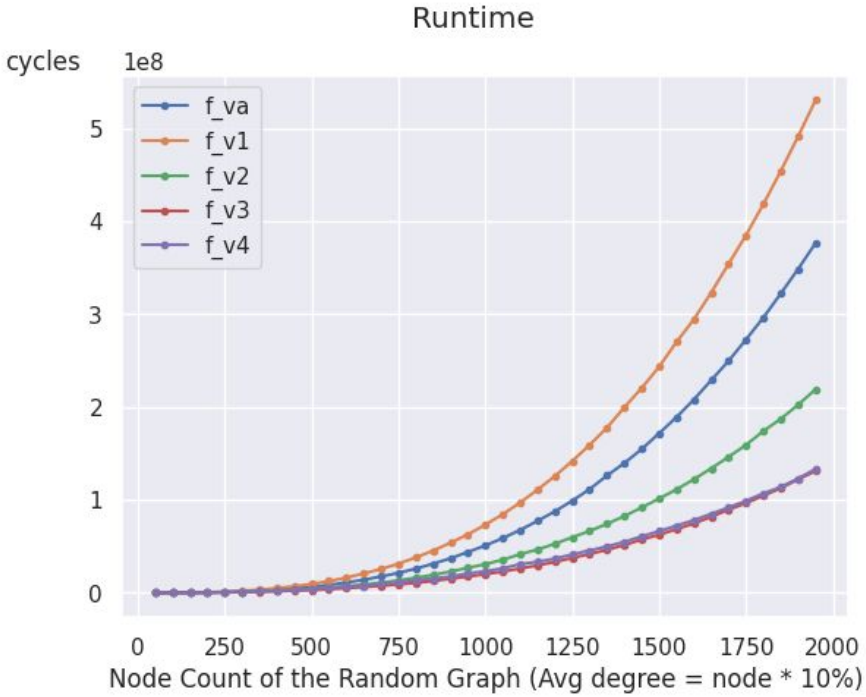
Problem: Set intersections are independent; can be interleaved/vectorized

For a given node s, we intersect its neighbourhood with every neighbourhood of every neighbour. These intersections are completely independent and only their results need to be accumulated in a shared list of triangles. However, as in comparison the number of triangles is usually low, the dependency is minimal

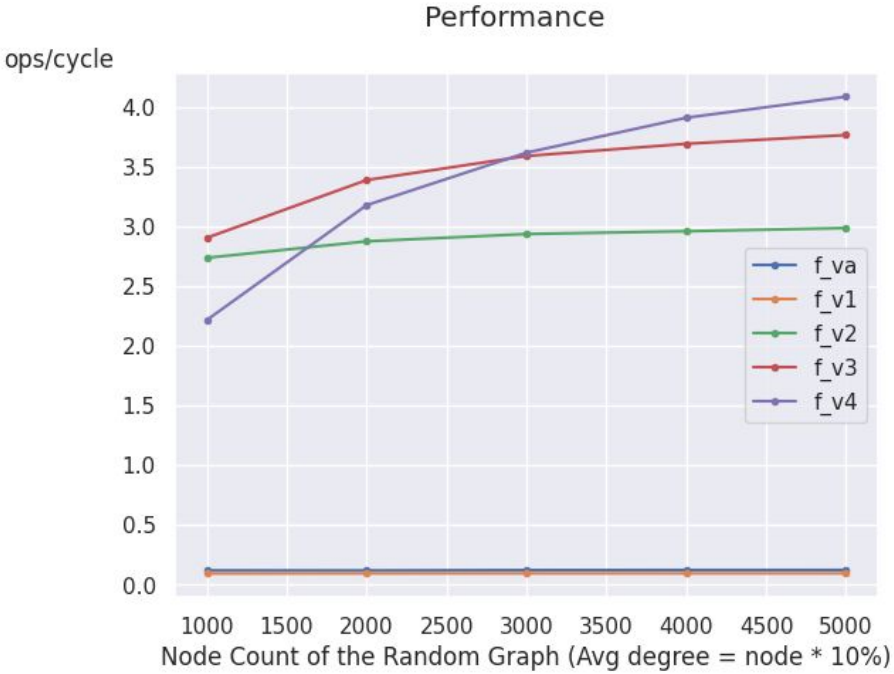
- V1: Unroll
- V2: 4
- V3: 8

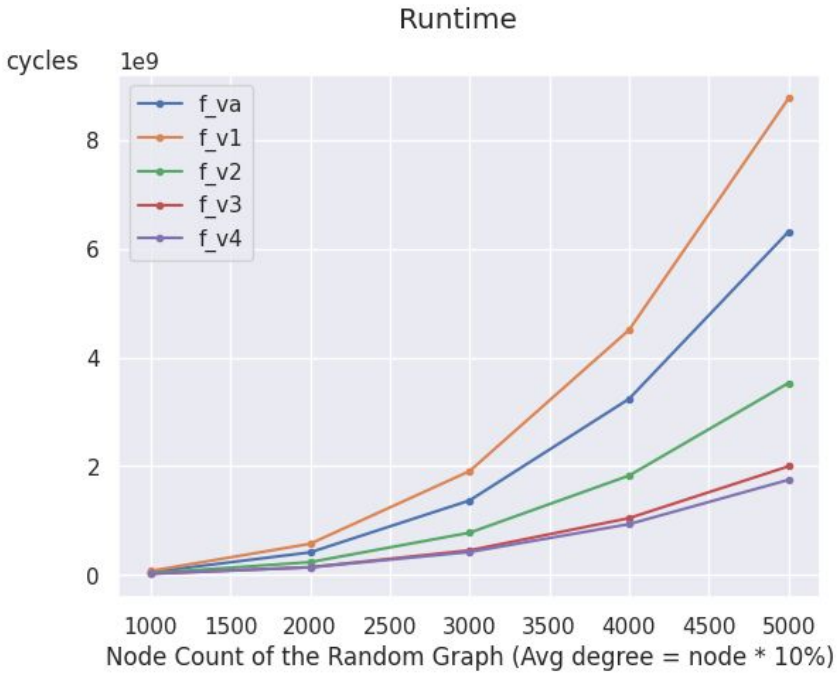
Forward Results: Dense Graphs



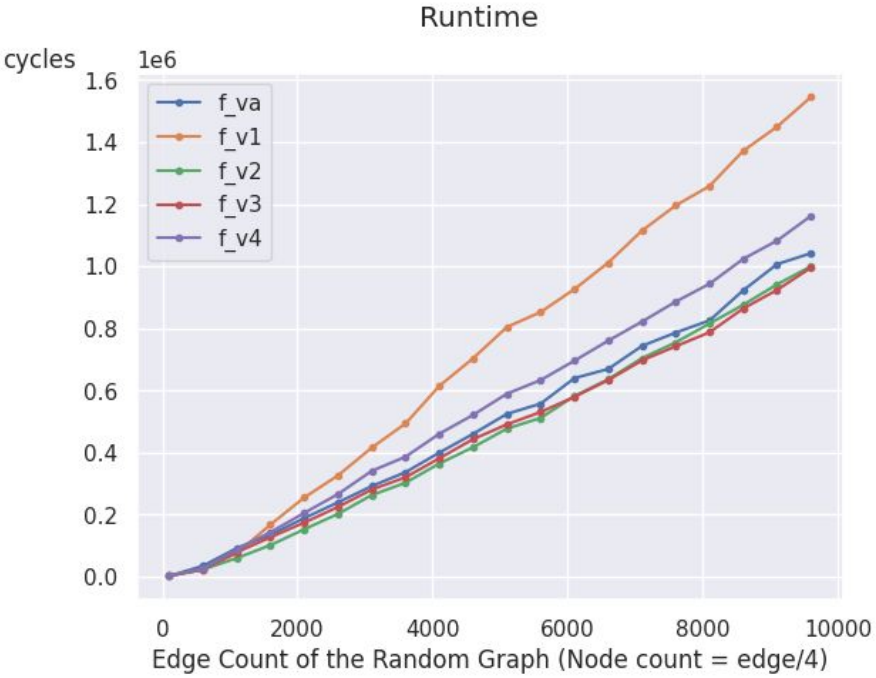


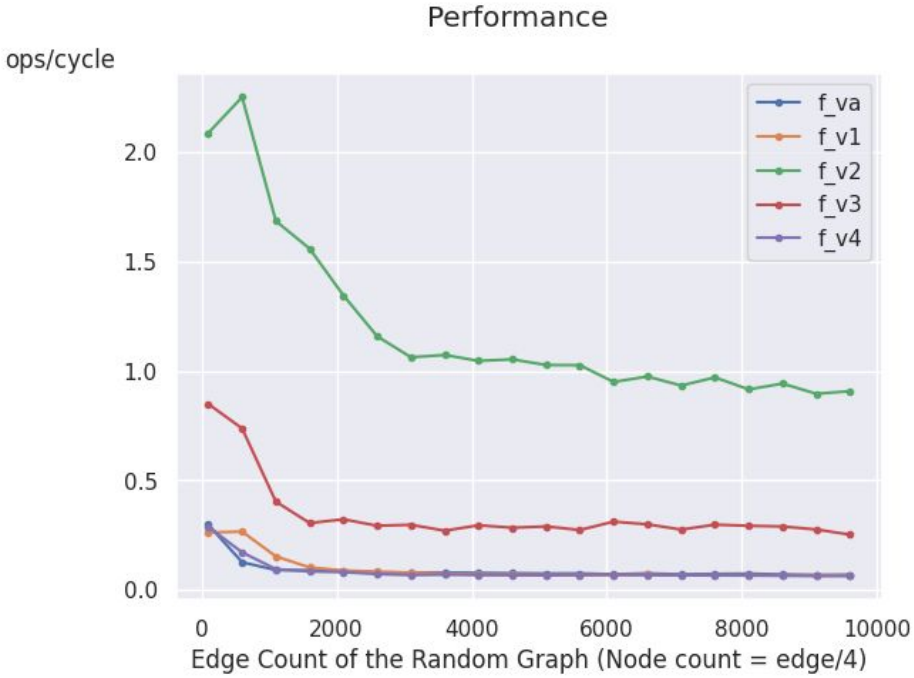
Forward Results: Dense Graphs



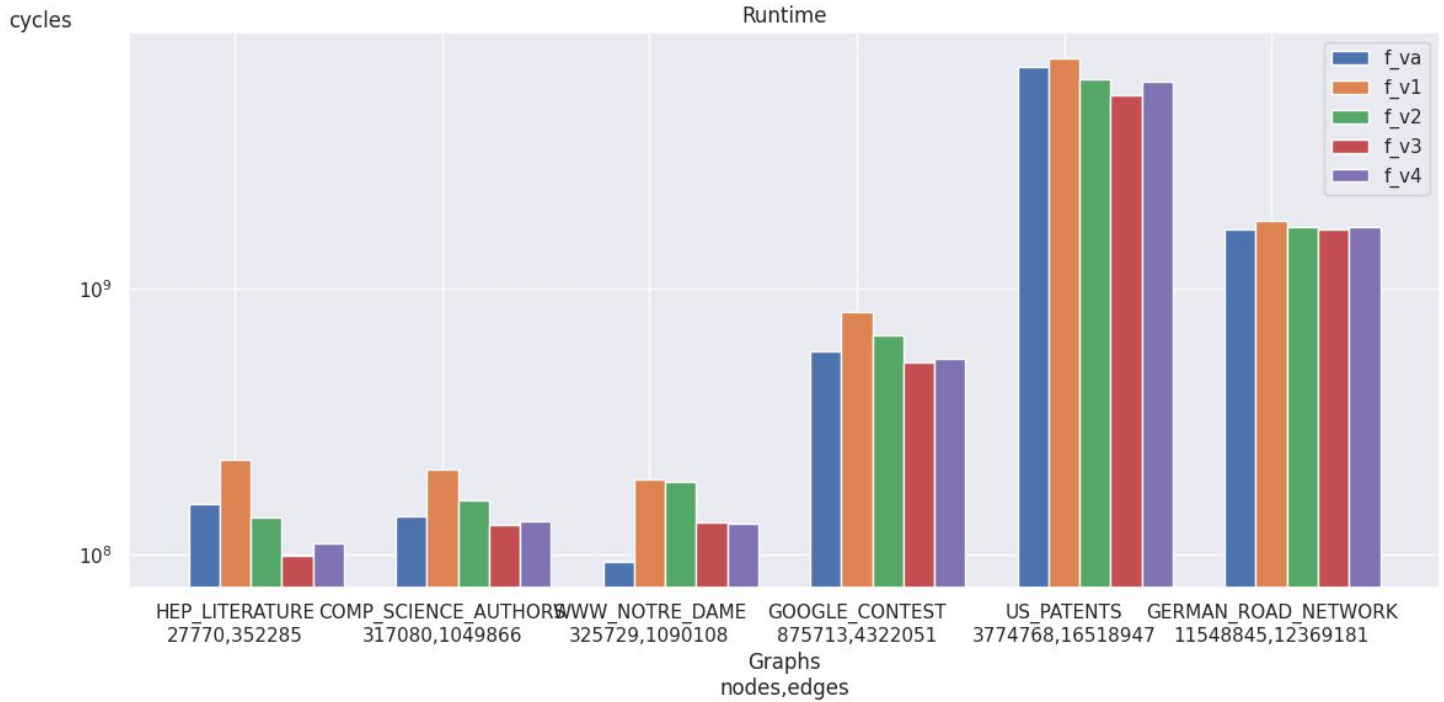


Forward: Result on Sparse Graphs

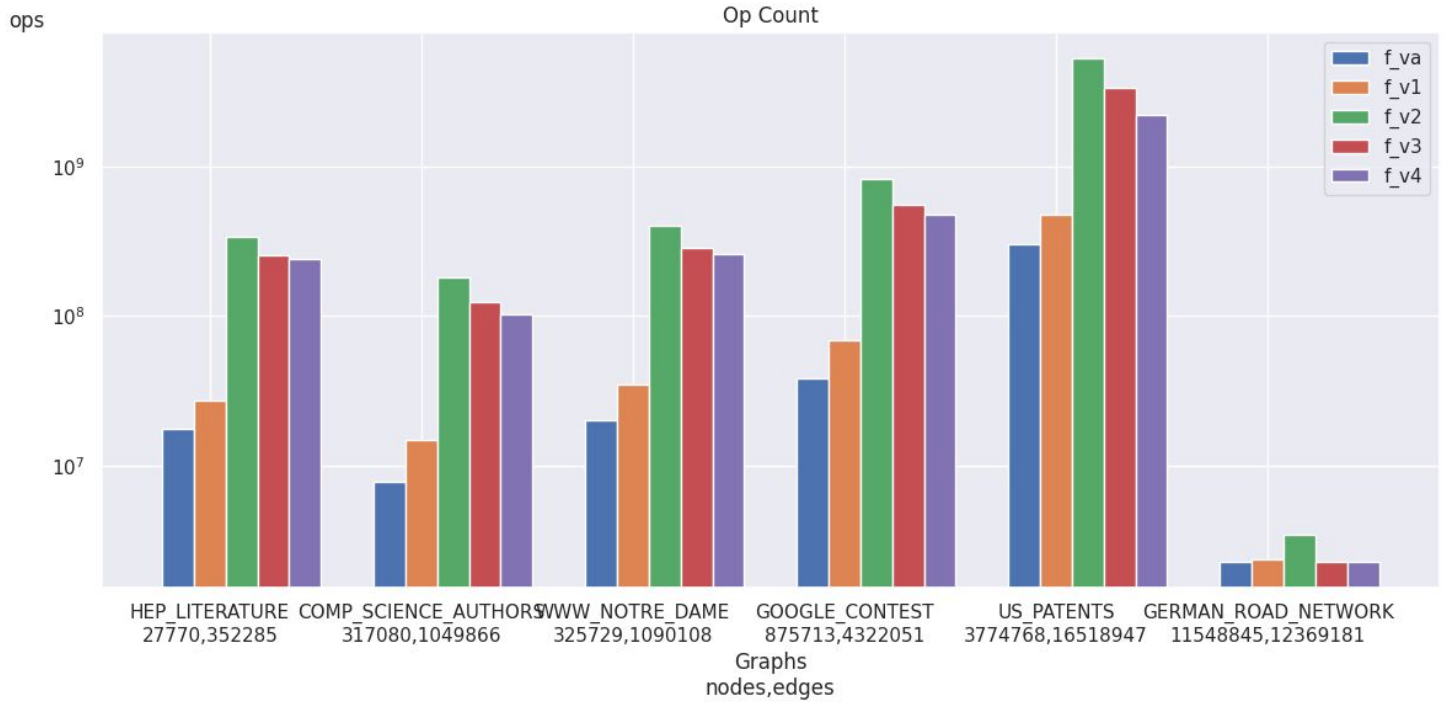




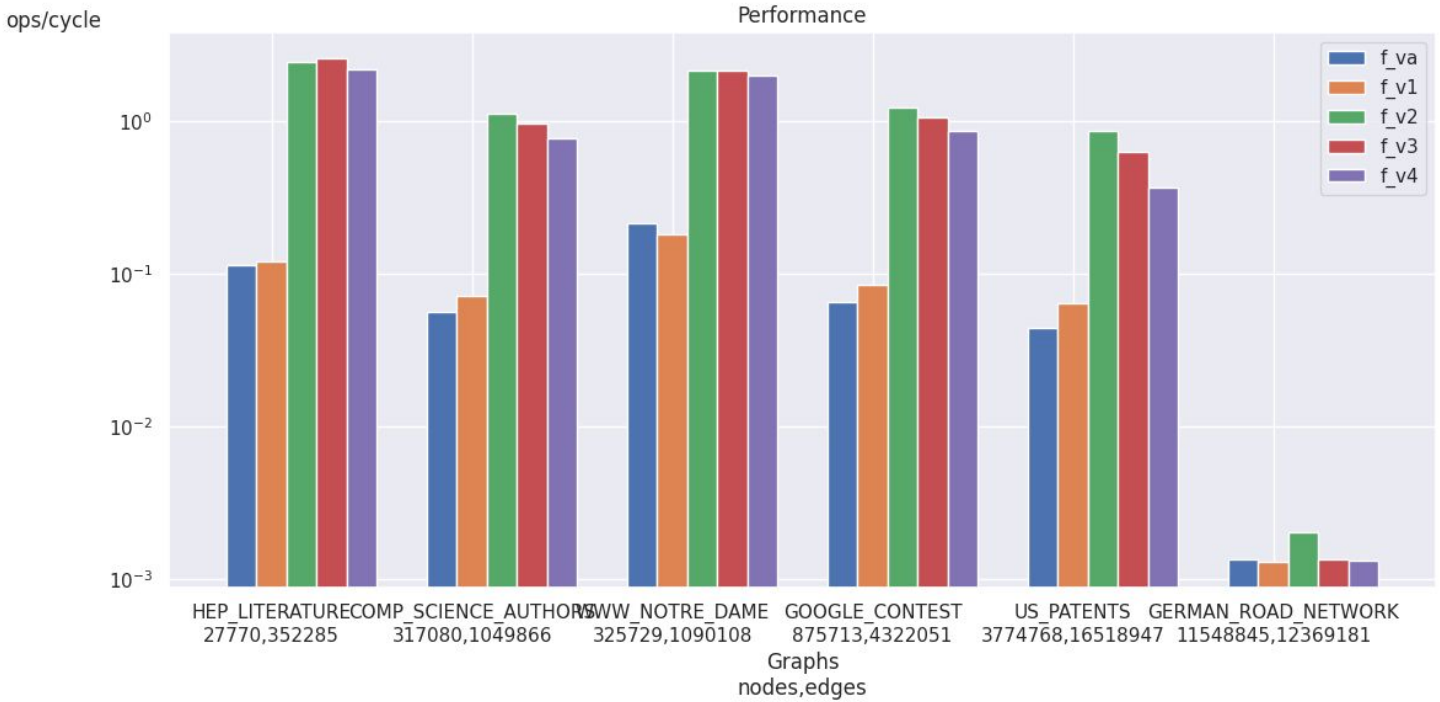
Forward Results: Real World Graphs



Forward Results: Real World Graphs

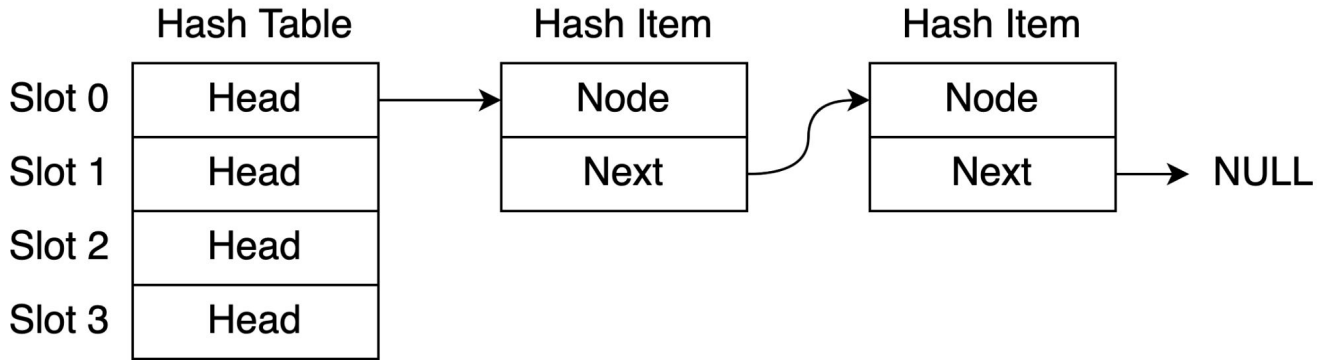


Forward Results: Real World Graphs



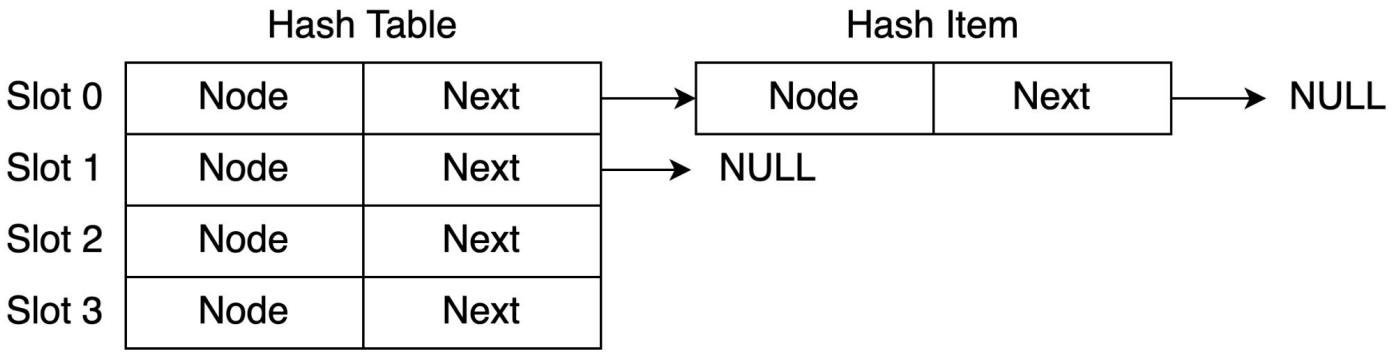
Forward Hashed Baseline

- Algorithm: find common nodes in the hash tables of neighbors
- Baseline: a lot of indirection with pointers



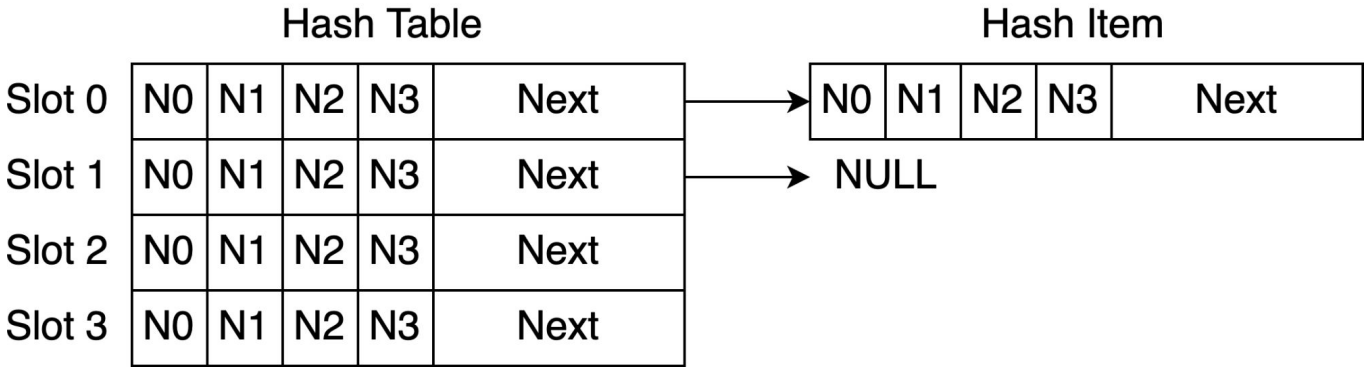
Forward Hashed V1

- Embed one Hash Item into the Hash Table

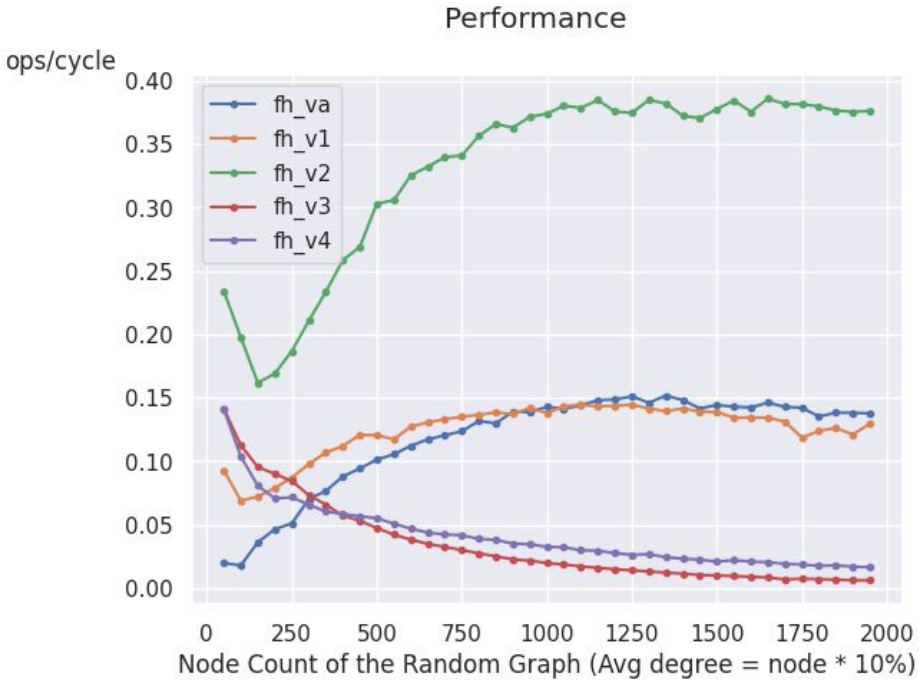


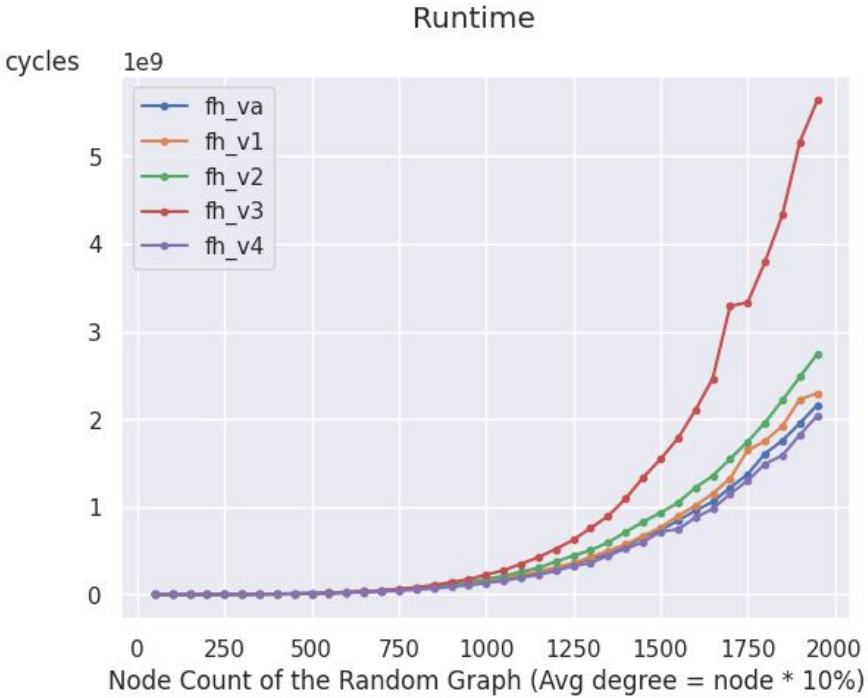
Forward Hashed V2, V3 and V4

- Multiple nodes per hash item to amortize the overhead of pointer tracing
- Vectorization: compare the node with N0-N3 all at once
- V3 & V4: different sizes of Hash Item

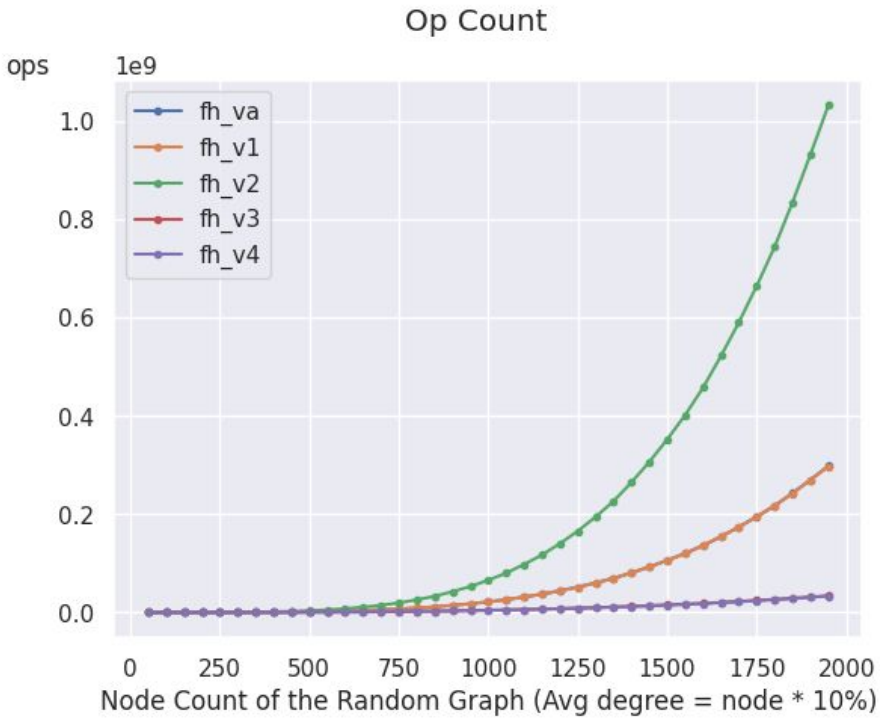


Forward Hashed Results: Dense Graphs

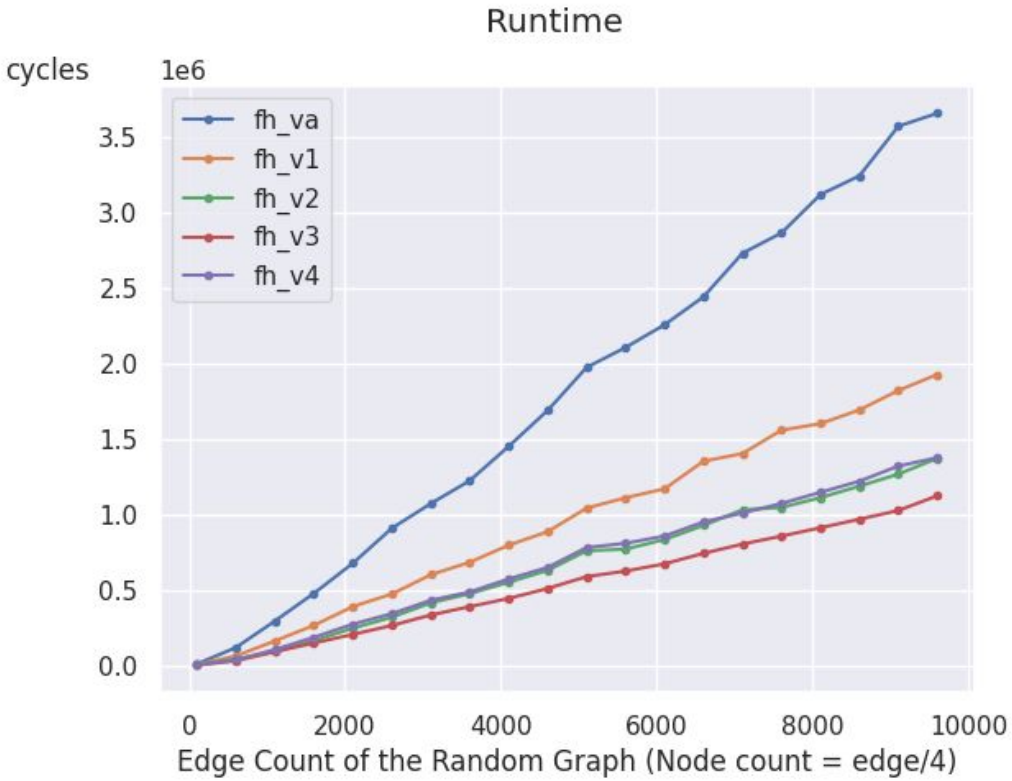


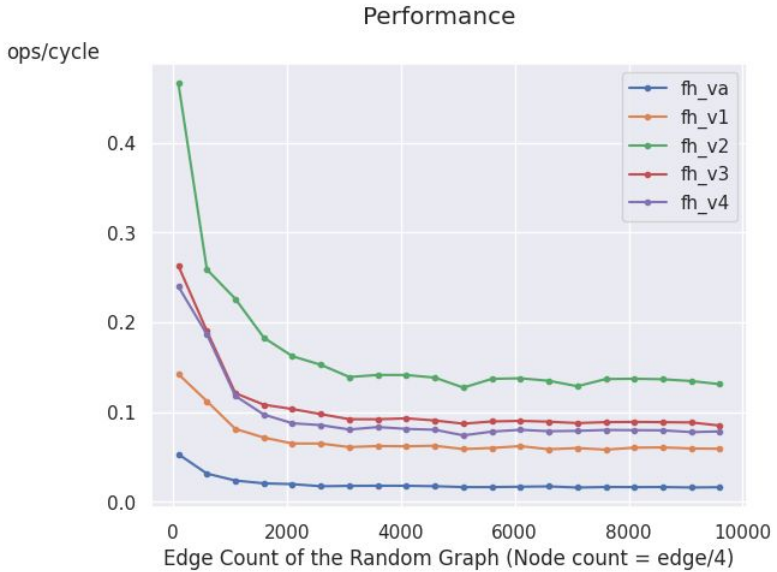


Forward Hashed Results: Dense Graphs

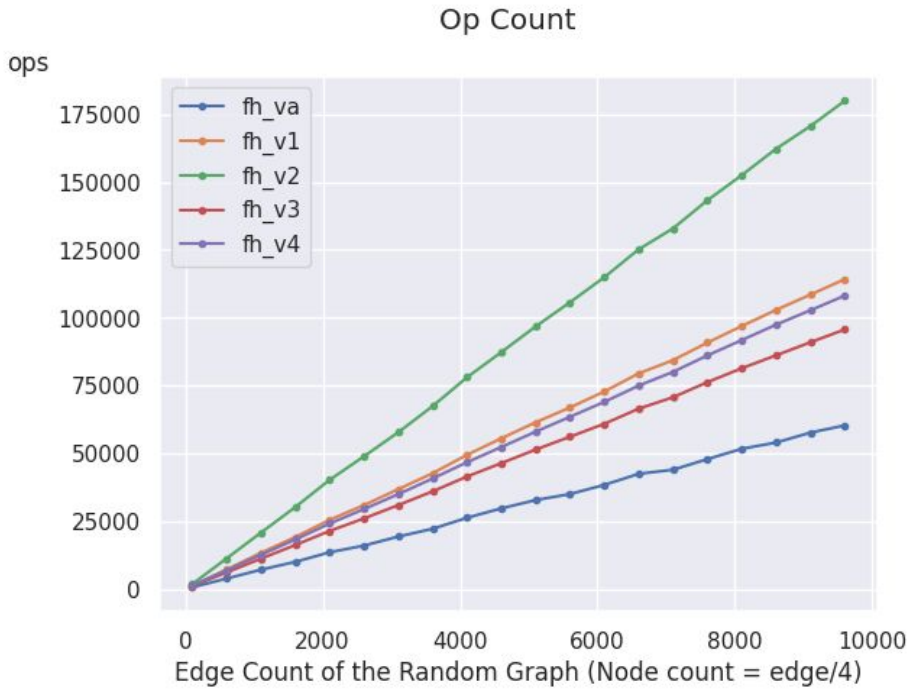


Forward Hashed Results: Sparse Graphs

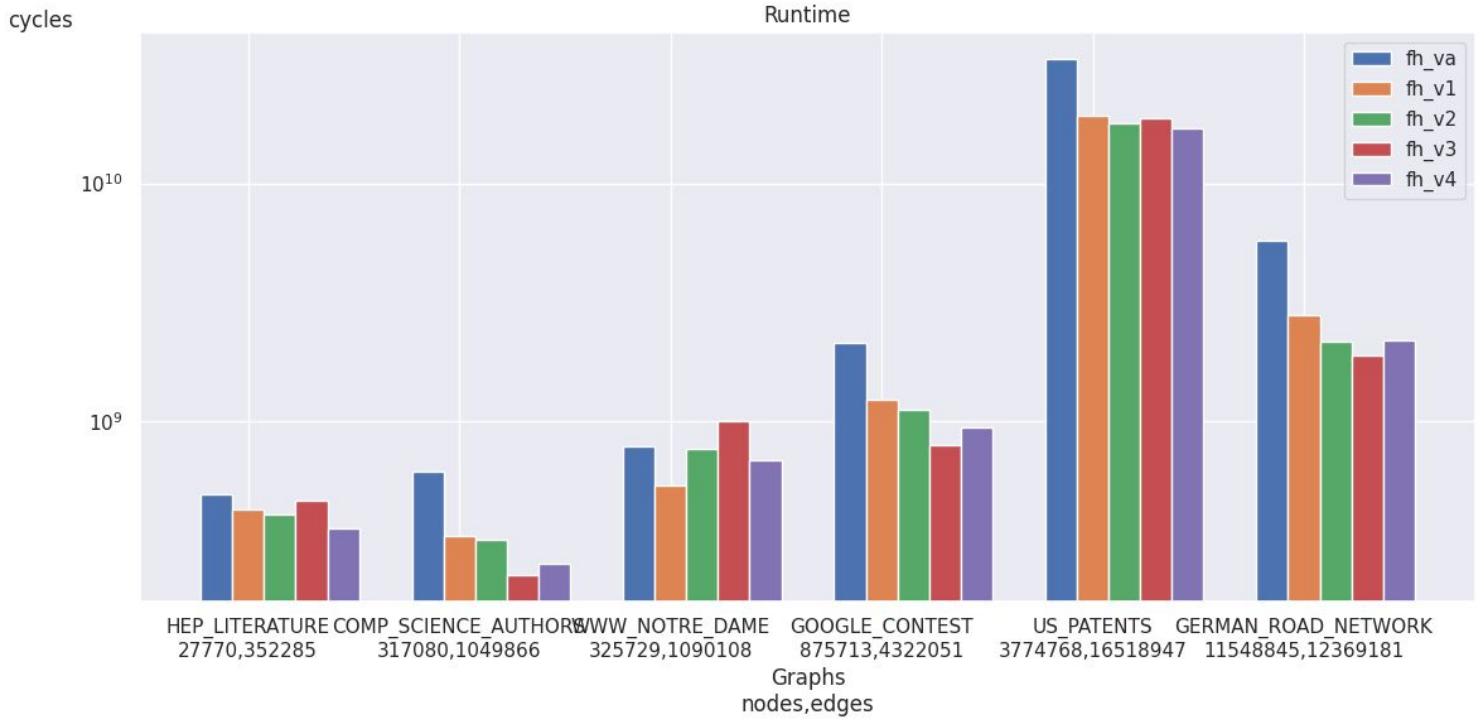




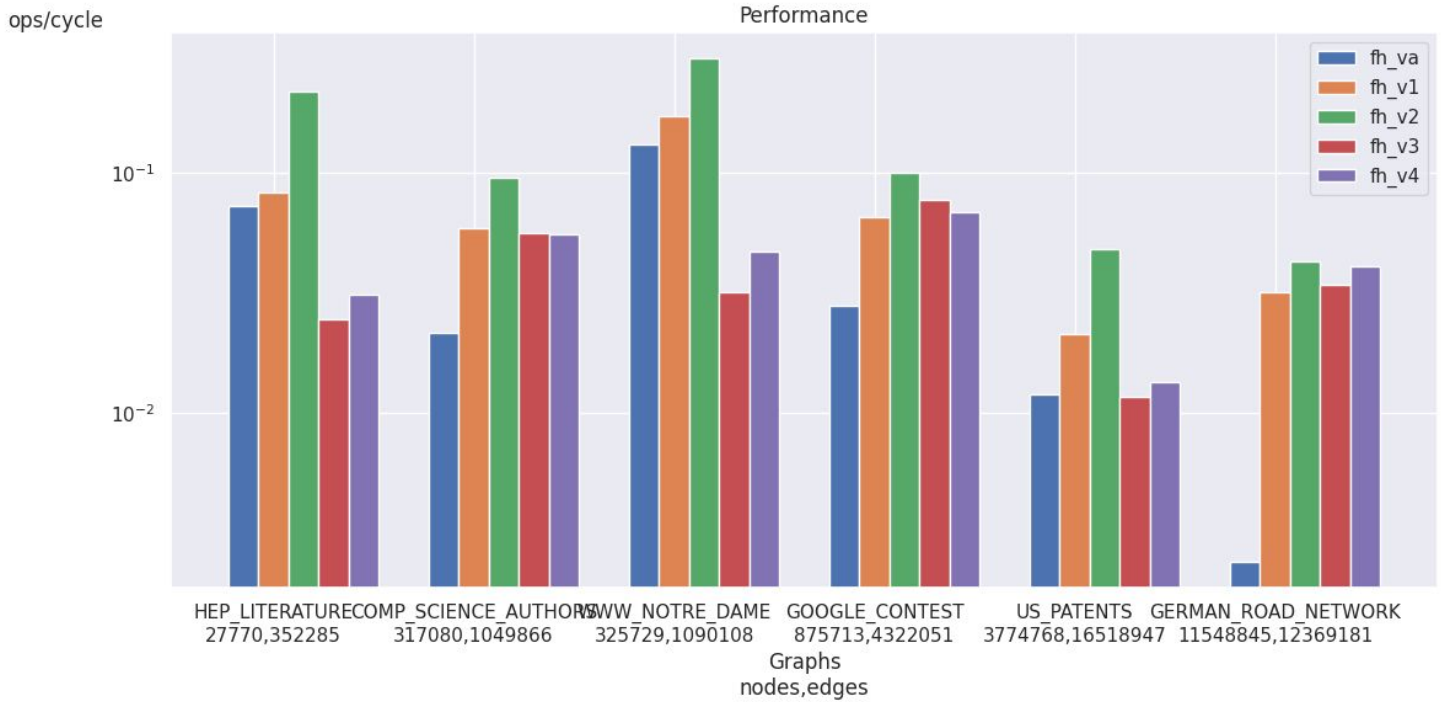
Forward Hashed Results: Sparse Graphs



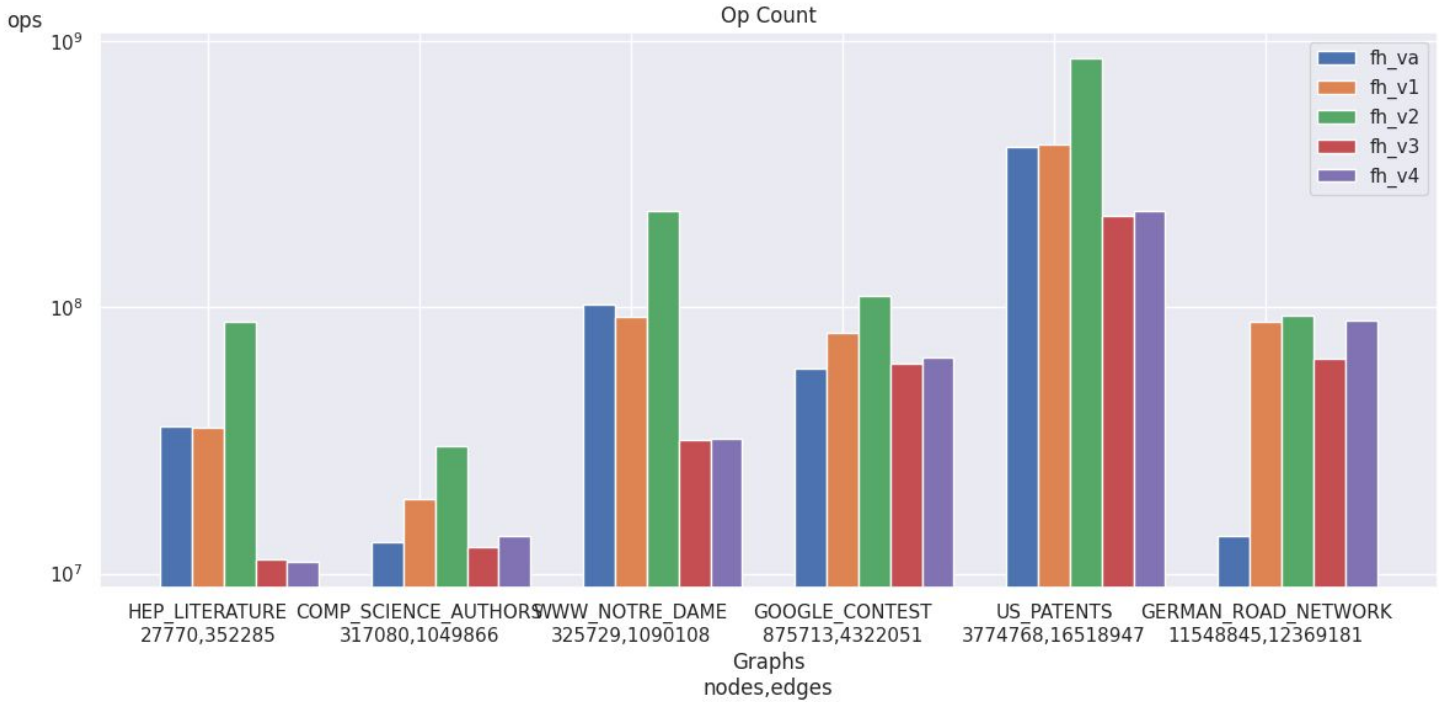
Forward Hashed: Real World Graphs



Forward Hashed: Real World Graphs



Forward Hashed: Real World Graphs



Questions

1. Sorting: how to optimize at a lower level? how much should we focus on it?
2. Triangle Counting or Listing? Do we have to write all the triangles to some array? Is it enough to only count?
3. What kind of graphs to experiment with? Dense/sparse?
4. Graphs seem to fit in cache? (95% L1 hit rate even for large graphs?)
5. Are we allowed to irrevesibly change the input graph, after it was received from the user?