Efficient Graph Processing

Spinellis, Diomidis

DOI
10.1109/MS.2024.3477013

Publication date
2025

Document Version
Final published version

Published in
IEEE Software

Editor: **Diomidis Spinellis**
dds@aueb.gr

Efficient Graph Processing

Diomidis Spinellis

SCIENCE TYPICALLY ADVANCES in small incremental steps, but in some rare instances it leaps forward. One discovery or invention can change how we see the world around us. Would it not be neat to be able to accurately pinpoint those moments of time in an objective way and thereby investigate science and technology's progress? In 2016, Russel Funk of the University of Minnesota's Carlson School of Management and Jason Owen-Smith from the University of Michigan published a measure for exactly this purpose.[1] Their so-called consolidation-disruption (CD) index quantifies the extent to which published findings affect the subsequent use of the knowledge on which those findings relied. Worryingly, a widely cited subsequent study applied this measure on patents and scientific publications, finding a slowdown in disruptive progress.[2] Thickening the plot, a later preprint attributed the finding to dataset artefacts.[3] These studies prompt the need for an efficient way to calculate the CD index on large amounts of openly available data.

The CD index for a given publication (or patent) is calculated by examining the citation graph associated with it: earlier citations cited by the publication and subsequent publications that cite the given publication and the publications it cites. When later publications cite both the examined one and the ones it cites [Figure 1(a)], it is considered a consolidating development. On the other hand, when subsequent publications do not seem to bother citing the ones the examined publication cites [Figure 1(b)], the publication indicates a destabilizing development because it marks a shift in the type of knowledge considered important. The CD metric formalizes this idea with a method normalizing citation ratios between the two extreme cases to a range from –1 (maximally consolidating) to 1 (maximally disrupting), with details for examining a specific time horizon (e.g., CD_5 for five years) and with a weight that considers a publication's importance. Commendably, Funk has released *cdindex*, an open source software Python package that calculates the CD index.[4] The calculation is performed by building a graph containing all publications linked by their citations as incoming and outgoing edges. After that, the offered `cdindex` function can form a set with all citations related to a publication and a specified time horizon, add them up according to their source and destination, and finally divide the sums by the set's cardinality to derive the CD index.

Calculating the CD index for all publications in a large dataset is expensive.

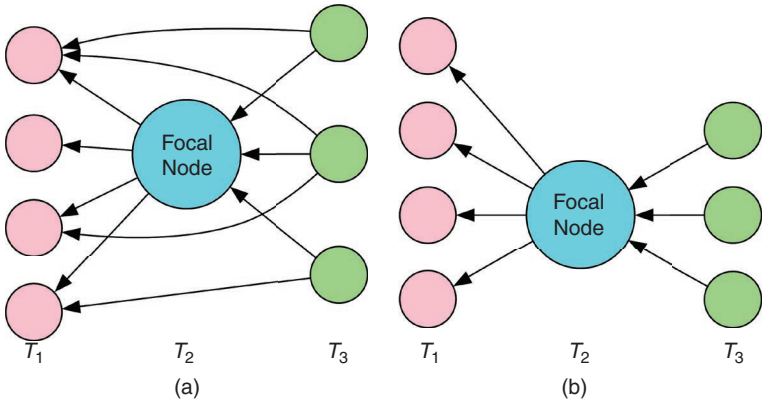

FIGURE 1. (a) A maximally consolidating and (b) a maximally destabilizing publication.

Digital Object Identifier 10.1109/MS.2024.3477013
Date of current version: 11 December 2024

Take as an example the publication and citation data released this year by Crossref, a scholarly research community nonprofit organization, which is the largest digital object identifier registration agency. The data contain details about 158 million publications linked through 2.3 billion citations: an immense graph. Based on the cost of processing just 3 million publications, I estimate it would take 22 days to calculate the CD index for all of them.

In the next sections I describe how I optimized the original *cdindex* package and its use to bring the figure down to a dozen hours. The optimized code and the steps I took to arrive to it are available on GitHub as a repository and its commit history.[5] In common with many performance optimization tasks, optimization came down to the employment of efficient data structures and algorithms, as well as parallelization.

Efficient Data Structures

As is the case with many of Python's scientific and machine learning libraries, the original CD index implementation is a Python module where some C code does the heavy lifting. This is an appropriate choice because the same code written in Python would use considerably more time and memory. The module works by representing publications and their citations as a graph, stored in a dynamically allocated array of vertices.

```
typedef struct Graph {
    Vertex *vs;
    long int vcount;
    long int ecount;
} Graph;
```

The index of each vertex in the array is used to represent its identifier, which allows accessing a vertex with a quick array lookup. The graph's edges are represented through an adjacency list. Each graph vertex maintains a list of its incoming and outgoing edges in dynamically allocated arrays of vertex identifiers as follows:

```
typedef struct Vertex {
    long int id;
    long int timestamp;
    long int *in_edges;
    long int *out_edges;
    long int in_degree;
    long int out_degree;
} Vertex;
```

This graph representation and its implementation offer several optimization opportunities.

First, the addition of each new edge reallocates space for the `in_edges` and `out_edges` arrays, potentially copying the existing array data to the larger newly allocated space. The cost of this operation is not constant: it depends on the number of stored edges. The same logic applies to the addition of each vertex. Consequently, this changes the complexity cost of constructing a graph with V vertices and E edges from $0(V+E)$ into $0(V^2+E^2)$. In general, implementing an algorithm with operations costing more than $0(1)$ increases the algorithm's complexity.

A second issue is that the representation of the graph's vertices as integer indices into the array of vertices burdens the CD index calculation with the cost of indirections through the array. In most modern languages this cost can be avoided by storing the edge data as pointers to the corresponding vertices rather than as integer identifiers of them. For example, rather than accessing the time stamp of a vertex v as `vs[v].timestamp`, we can access it as `v->timestamp`.

A third issue is that each vertex's in and out degrees are stored explicitly and also duplicated in the metadata of the dynamically allocated memory used for storing the corresponding edges. This wastes memory, which for millions of vertices adds up.

Finally, all these issues result in less predictable and more memory accesses, which, as a secondary effect, reduce the benefits of the CPU's caches, increasing the calculation's run time.

The reallocation cost can be easily reduced by doubling the allocated memory each time it exceeds the originally allocated space. However, it is counterproductive and error-prone to explicitly optimize memory allocations. Instead, it is best to reuse existing polished and tested libraries. In this case, an obvious choice is the C++ Standard Template Library (STL) `std::vector` data structure, which allows us to store each vertex as follows:

```
class Vertex {
private:
    timestamp_t timestamp;
    std::vector<Vertex *> in_edges;
    std::vector<Vertex *> out_edges;
}
```

Adding elements to a C++ `std::vector` via the `push_back` method is guaranteed to have an amortized constant time cost, thereby avoiding the algorithmic complexity increase of reallocating memory as each element is added. (Internally, the `std::vector` implementation is most likely using the doubling of allocated space method, but this is not something that we need to care about.) Furthermore, the two edge vectors store pointers to other vertices, thereby eliminating the cost of indirecting through vertex identifiers. Finally, the vertex's in and out degrees are obtained through methods that return the size of the underlying vectors, eliminating the cost of duplicating this information.

```
size_t get_in_degree() const {
    return in_edges.size();
}
size_t get_out_degree() const {
    return out_edges.size();
}
```

The size of the original vertex data structure is 48 bytes on an x86-64 CPU. To this, one must add the 24 byte overhead for maintaining each of the two dynamically allocated arrays, which gives a total empty vertex footprint of 96 bytes. In contrast, the size of the C++ `Vertex` implementation is 56 bytes, almost half of the original footprint.

One remaining issue regarding the new implementation is the representation of the C++ vertex pointers in Python, which (rightly) lacks support for unrestricted pointers. I addressed this by defining a `union` data type that stores a vertex identifier both as a pointer to the vertex data and as an (`unsigned long`) integer overlayed over the same area as the pointer.

```
typedef union {
    unsigned long int id;
    Vertex *v;
} vertex_id_t;
```

This allows Python's C interface to obtain and return the vertex identifiers through `id` as integers with the same bit representation as the original C++ pointers. Correspondingly, the C++ code processes the vertex identifiers through `v` as pointers.

Efficient Algorithms

The original implementation of the CD index calculation also offered opportunities to employ more efficient algorithms. In two loops and two doubly nested loops the calculation calls the function `in_int_array` to determine whether a specific vertex is connected to another. The function works by performing a linear search through the array's elements, which results in a quadratic blowup in the algorithm's polynomial complexity.

An alternative to this method might be storing the edges as an STL `std::set` data structure, whose lookup complexity is much more efficient: logarithmic rather than linear. However, this data structure requires additional space, which I wanted to avoid for the billions of edges that would get stored. I addressed this by capitalizing on the fact that the CD index calculation is performed by first building the complete graph and then processing its vertices without further modifying it. This allowed me to write a method that can be called at the end of the graph's construction to optimize the graph for further processing.

```
void Graph::prepare_for_searching() {
    for (auto i: versus) {
        i->shrink_to_fit();
        i->sort_out_edges();
}
}
```

The method does two things. First, it disposes unused space that was initially (over)allocated for the efficient addition of edges.

```
void Vertex::shrink_to_fit() {
    out_edges.shrink_to_fit();
    in_edges.shrink_to_fit();
}
```

Second, it sorts the vector containing the outgoing edges (which are the ones that are searched for the CD index calculation).

```
void Vertex::sort_out_edges() {
    std::sort(out_edges.begin(), out_edges.end());
}
```

The sorted vector allows a vertex's edges to be searched with an efficient $O(\log N)$ binary search rather than the original more expensive $O(N)$ one without any storage overhead.

A further similar optimization opportunity presents itself in the CD calculation at the point where it builds a set of vertices that are the incoming edges of the focal vertex's outgoing edges at a given time stamp. As this data structure is transient for each edge, it makes sense to build it as an STL `std::set` from the beginning and use its efficient $O(\log N)$ operations for adding unique elements to it and then iterating through them.

```
    std::set<Vertex *> it;
[…]
    it.insert(out_edge_i_in_edge_i);
[…]
    it.insert(in_edge_i);
[…]
    for (auto i: it) {
[…]
```

Careful performance measurements should always back any performed code optimizations. In this case, the vertex addition throughput actually dropped from 109 vertices per second to 91 vertices per second, but the edge addition throughout increased from 74 edges per second to 241 edges per second. More importantly, the CD index calculation throughput also increased from 84 values per second to a sprightly 672 values per second. For the 2024 Crossref publication dataset these numbers would bring down the estimated calculation duration from 22 days to fewer than three days.

And Parallelization

Three days is still a significant time for a job to run, so it makes sense to examine whether this can be further

reduced. An important insight is that once the graph is built, the CD index calculations can be performed independently from each other, making it a trivially parallelizable operation.

Unfortunately, achieving this parallelism in Python is close to impossible. The two alternatives involve using multiple threads or multiple processes. Multiple threads in Python, run, e.g., with `ThreadPoolExecutor`, are not helpful for CPU-bound tasks, such as the CD index calculation, because a global interpreter lock prevents more than one processing thread from being executed concurrently. In the case of multiple processes sharing the graph's data structure, another feature of Python kicks in to limit the method's potential. Because Python maintains a reference count for the objects it uses, when one of the multiple processes accesses the shared graph's data structure, the corresponding reference counts get incremented. This forces the operating system kernel's copy-on-write mechanism to create a private (nonshared) copy of the corresponding data, resulting in hugely expensive data copies and memory size increases. (The 2024 Crossref dataset citation graph occupies 62 GiB of RAM.)

Consequently, I decided to perform the CD index calculation entirely in C++ rather than in Python.[5] This mainly involved rewriting the Python driver code into C++. C++ has matured significantly over the past decades, becoming considerably more expressive. Its expressiveness allowed me to rewrite the 154 lines of Python code into just 241 lines of C++ code, which also utilized concurrency through the CPU's multiple processing cores.

I designed the concurrent processing based on STL's parallel algorithms, introduced in C++17. Specifically, I implemented the parallelism by constructing multiple batches of nodes that could be processed by a single worker. Batching together a few thousand nodes reduces the overhead of coordinating workers by amortizing it over a larger amount of work.

```
const int BATCH_SIZE = 10000;
vector <work_type> chunks;
auto pos = s2v.begin();
auto begin = pos;
size_t i;
for (i = 0; i < s2v.size(); i++, pos++)
   if (i > 0 && i % BATCH_SIZE == 0) {
      chunks.push_back(pair{begin, pos});
      begin = pos;
   }
```

With these chunks at hand I could then invoke the `std::for_each` algorithm, which applies a function to each element in a range.

```
for_each(execution::par_unseq, chunks.begin(),
chunks.end(), worker);
```

The specified `par_unseq` execution policy allows the elements to be executed in parallel in different threads, while `worker` is the function that calculates the CD index for each element in the chunk.

Running the CD_5 index calculation on the entire 2024 Crossref dataset on a 40-core CPU took 3.7 h of wall clock time against 79 h of CPU time, giving a speedup of close to 22 times compared with the sequential execution and more than 140 times compared with the original unoptimized CD index implementation.

ABOUT THE AUTHOR

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology, Athens University of Economics and Business, 104 34 Athens, Greece, and a professor of software analytics in the Department of Software Technology, Delft University of Technology, 2600 AA Delft, The Netherlands. He is a Senior Member of IEEE. Contact him at dds@aueb.gr.

REFERENCES

1. R. J. Funk and J. Owen-Smith, "A dynamic network measure of technological change," *Manage. Sci.*, vol. 63, no. 3, pp. 791–817, 2016, doi: 10.1287/mnsc.2015.2366.
2. M. Park, E. Leahey, and R. J. Funk, "Papers and patents are becoming less disruptive over time," *Nature*, vol. 613, no. 7942, pp. 138–144, 2023, doi: 10.1038/s41586-022-05543-x.
3. V. Holst, A. Algaba, F. Tori, S. Wenmackers, and V. Ginis, "Dataset artefacts are the hidden drivers of the declining disruptiveness in science," 2024, *arXiv:2402.14583*.
4. R. J. Funk. "cdinex." GitHub. Accessed: Oct. 17, 2024. [Online]. Available: https://github.com/russellfunk/cdindex
5. R. J. Funk and D. Spinellis. "fast-cdindex." GitHub. Accessed: Oct. 17, 2024. [Online]. Available: https://github.com/dspinellis/fast-cdindex/
6. D. Spinellis, "Open reproducible scientometric research with Alexandria3k," *PLoS One*, vol. 18, no. 11, Nov. 2023, Art. no. e0294946, doi: 10.1371/journal.pone.0294946.