# Compressing Inverted Indexes with Recursive Graph Bisection: A Reproducibility Study

Joel Mackenzie[1], Antonio Mallia[2], Matthias Petri[3], J. Shane Culpepper[1], and Torsten Suel[2]

[1] RMIT University, Melbourne, Australia
[2] New York University, New York, US
[3] The University of Melbourne, Melbourne, Australia

**Abstract.** Document reordering is an important but often overlooked preprocessing stage in index construction. Reordering document identifiers in graphs and inverted indexes has been shown to reduce storage costs and improve processing efficiency in the resulting indexes. However, surprisingly few document reordering algorithms are publicly available despite their importance. A new reordering algorithm derived from recursive graph bisection was recently proposed by Dhulipala et al., and shown to be highly effective and efficient when compared against other state-of-the-art reordering strategies. In this work, we present a reproducibility study of this new algorithm. We describe the implementation challenges encountered, and explore the performance characteristics of our clean-room reimplementation. We show that we are able to successfully reproduce the core results of the original paper, and show that the algorithm generalizes to other collections and indexing frameworks. Furthermore, we make our implementation publicly available to help promote further research in this space.

**Keywords:** Reordering · Compression · Efficiency · Reproducibility

## 1 Introduction

Scalable processing and storage of large data collections has been a longstanding problem in Information Retrieval (IR). The volume of data being indexed and retrieved continues to grow, and a wealth of academic research has focused on managing this new data. A key area of focus is how to better compress the data structures used in these storage applications; better compression results in lower storage costs, and improves the efficiency of accessing data. *Document reordering* is a widely used technique that improves the compression rate of many coding schemes at the cost of additional computation at indexing time. However, finding a favorable reordering is a challenging problem for IR-scale data collections. The problem has been extensively studied in academia, and making significant improvements that are both effective and practical is quite difficult.

Documents identifiers can be assigned in many ways, such as random ordering, based on document similarity or page relevance [21], or by just following a sorted URL ordering [23]. Further, it has been noted that document reordering

can also result in improved query processing efficiency [11, 15, 16], although a thorough understanding of this effect is still missing. Considering the advantages that reordering can yield, it is critical that tools for reordering are made publicly available, and that researchers describe the order of their index when conducting large scale efficiency studies (such as [20, 11, 8, 18, 16]).

Recently, Dhulipala et al. [10] proposed a new algorithm which aims to minimize an objective function directly related to the number of bits needed to store a graph or an index using a delta-encoding scheme. The authors experimented on both graphs and inverted indexes, obtaining notable improvements when compared to previous approaches. Their algorithm, based on *recursive graph bisection*, is currently the state-of-the-art algorithm for minimizing the compressed space used by an inverted index (or graph) through document/vertex reordering. An unfortunate aspect of this work is that the implementation was unable to be released "due to corporate restrictions," most likely because the work was done primarily at Facebook. In this paper, we perform a "clean-room" reimplementation of this algorithm, reproduce the results obtained by the original authors, and extend their original experiments to additional collections in order to confirm the effectiveness of the approach.

*Our Contributions* The key contributions of this paper are:

1. We implement and validate the algorithm originally presented by Dhulipala et al. [10]. We confirm both effectiveness in compression due to reordering and efficiency in terms of execution time and memory usage.
2. We extend the experimental analysis to other large collections. The original work focused primarily on reordering graphs, with experiments shown for just two standard text collections: Gov2 and ClueWeb09. With an extensive experimental analysis over four additional text collections, we strengthen the evidence of the generalizability of the approach.
3. We evaluate an additional compression technique with the reordered index, to further examine how well the approach generalizes.
4. We make our implementation publicly available in order to motivate future analysis and experimentation on the topic.

## 2 Overview of Document Reordering

Several previous studies have looked at the document reordering problem. In this section, we outline the problem of document identifier assignment, review the key techniques that have been proposed in the literature, and describe the recursive bisection algorithm that is the focus of this work.

### 2.1 Document Identifier Assignment

The document identifier assignment problem can be described informally as finding a function that maps document identifiers to new values with the aim of minimizing the cost of coding the document gaps. More formally, different approaches exist to reduce this problem to several classical NP-Hard problems such as TSP [22], and versions of the optimal linear arrangement problem [2, 7, 10].

The most intuitive formalization is the *bipartite minimum logarithmic arrangement* (BiMLogA) problem [10] which models an inverted index as a bipartite graph $G = (V, E)$ with $|E| = m$ and the vertex set $V$ consisting of a disjoint set of terms, $T$, and documents, $D$, $V = (T \cup D)$. Each edge $e \in E$ corresponds to an arc $(t, d)$ with $t \in T$ and $d \in D$ that implies that document $d$ contains term $t$. The BiMLogA problem seeks to find an ordering $\pi$ of the vertices in $D$ which minimizes the *LogGap* cost of storing the edges for each $t \in T$:

$$LogGap = \frac{1}{m} \sum_{t \in T} \sum_{i=0}^{\mathrm{d}_t} \log_2(\pi(u_{i+1}) - \pi(u_i))$$

where $\mathrm{d}_t$ is the degree of vertex $t \in T$, and $t$ has neighbors $\{u_1, \ldots, u_{\mathrm{d}_q}\}$ with $\pi(u_1) < \cdots < \pi(u_{\mathrm{d}_t})$ and $u_0 = 0$. Intuitively, *LogGap* corresponds to minimizing the average logarithmic difference between adjacent entries in postings lists of an inverted index and can generally be considered a lower bound on the storage cost (in bits per integer) of a posting in an inverted index.

### 2.2 Document Ordering Techniques

Although there are a wide range of document ordering techniques that have been proposed [5, 4, 3, 2, 12, 24, 22], we focus on a few that can be efficiently run on web-scale data sets while also offering significant compression benefits. In particular, we focus on a subset of techniques that were examined in the work that we are reproducing [23, 7, 10].

*Random Ordering* A `Random` document ordering corresponds to the case where identifiers are assigned randomly to documents with no notion of clustering or temporality. We reorder the document identifiers based on an arbitrary ordering specified by a pseudorandom number generator. This ordering represents the worst-case scenario (short of an adversarial case), and is used as a point-of-reference when comparing approaches.

*Natural Ordering* Text collections usually have some notion of a `Natural` ordering. Two common orderings that can be considered as natural are either the *crawl* order of the documents, which assigns identifiers in a monotonically increasing order as they are found by the crawler, or the *URL* ordering, which is based on lexicographically sorting the URLs of the indexed documents [23].

*Minhash Ordering* The `Minhash` (or shingle) ordering is a heuristic ordering that approximates the *Jaccard Similarity* of documents in order to cluster similar documents together [6, 7]. The main idea is to obtain a fingerprint of each document (or neighbors of a graph vertex) through hashing, and to position similar documents (or vertices) close to each other. The key observation is that this improves *clustering*, which aids compression.

### 2.3 Recursive Graph Bisection

The `BP` ordering is the primary focus for this reproducibility study [10]. `BP` was proven to run in $\mathcal{O}(m \log n + n \log^2 n)$ time, and shown experimentally to yield excellent arrangements in practice. Unlike the aforementioned approaches, which

implicitly try to cluster similar documents together (thus reducing the overall size of the delta encoding), the `BP` algorithm explicitly optimizes for an ordering by approximating the solution to the BiMLogA problem. The algorithm is described in Algorithm 1. On a high level, the algorithm recursively splits (bisects) the ordered set of document identifiers $D$ into smaller ordered subsets $D_1$ and $D_2$. At each level of the recursion documents are swapped between the two subsets if a swap improves the *LogGap* objective.

At each level an initial document ordering (such as `Random` or `Minhash`) of all document identifiers in the current subset $D$ is used to create two equally sized partitions $D_1$ and $D_2$ (line 2). Next, for a fixed set of iterations (*MaxIter*) the algorithm computes the *MoveGain* (described below) which results from moving documents in $D$ between partitions (lines $4-6$). In each iteration, the documents with the highest *MoveGain*, i.e., the documents for which swapping partitions reduces the *LogGap* the most, are exchanged as long as the overall gain of the swap is beneficial (lines $8-10$). The current level of the recursion finishes once *MaxIter* iterations have been performed, or no document identifier swaps have occurred in the current iteration (lines $11-12$). Next, the same procedure is recursively applied to $D_1$ and $D_2$ until the maximum recursion depth (*MaxDepth*) is reached (lines $13-15$). As the recursion unwinds, the ordered partitions are 'glued' back together to form the final ordering.

Computing the *MoveGain* for a specific document/node $v$ is shown in lines $17-24$. The function computes the average logarithmic gap length for all $t \in T$, for the parts of the adjacency lists (or postings lists) corresponding to documents in $D_a$ and $D_b$. Specifically, the *MoveGain* of a document in $D_a$ is defined as the *difference* in average logarithmic gap length between $v$ remaining in $D_a$ and $v$ moving to $D_b$ (and vice versa for a $v$ in $D_b$). This gain can be positive (i.e., it is beneficial to move $v$ to the other partition) or negative.

## 3 Reproducibility

Following other recent reproducibility studies in the field of IR [14], we adapt the following definition of reproducibility from the 2015 SIGIR Workshop on Reproducibility, Inexplicability, and Generalizability of Results [1]: *"Repeating a previous result under different but comparable conditions."* To this end, we are interested in reproducing improvements in the compression of textual indexes to the same degree as the improvements reported by Dhulipala et al. [10], using a full reimplementation of the methods as described in the paper.

### 3.1 Implementation Details

In this section we are going to present the choices we made in our implementation. Even though the basic algorithm is conceptually simple to understand and implement, important details on implementing the algorithm so that it is scalable and efficient were omitted in the original work.

The first step is to build a forward index, which, in our case, is compressed using VarintGB [9] to optimize memory consumption. This forward index can be considered a bipartite graph, and is the input used by the `BP` algorithm.

**Algorithm 1:** Graph Reordering via Recursive Graph Bisection

```
1  Function RecursiveBisection(T, D, d)
       In: Bipartite graph (T, D) with |D| = n vertices and recursion depth d
2      D_1, D_2 = SortAndSplitGraph(D)
3      for iter = 0 to MaxIter do
4          forall v in D_1 and u in D_2 do
5              gains_{D_1}[v] = ComputeMoveGain(T, v, D_1, D_2)
6              gains_{D_2}[u] = ComputeMoveGain(T, u, D_2, D_1)
7          SortDecreasing(gains_{D_1},gains_{D_2})
8          forall v in gains_{D_1} and u in gains_{D_2} do
9              if gains_{D_1}[v] + gains_{D_2}[u] > 0 then
10                 SwapNodes(v, u)
11         if No Swaps Occurred then
12             iter = MaxIter
13     if d < MaxDepth then
14         D_1 = RecursiveBisection(T, D_1, ⌊n/2⌋, d + 1)
15         D_2 = RecursiveBisection(T, D_2, ⌈n/2⌉, d + 1)
16     return Concat(D_1, D_2)

17 Function ComputeMoveGain(T, v, D_a, D_b)
       In: Bipartite Graphs (T, D_a), (T, D_b) with |D_a| = n_a, |D_b| = n_b and
           v ∈ D_a
18     gain = 0
19     forall t in T do
20         if t connected to v then
21             d_a, d_b = number of edges in from t to D_a and D_b
22             gain = gain + d_a log_2(n_a/(d_a+1)) + d_b log_2(n_b/(d_b+1))
23             gain = gain − (d_a − 1) log_2(n_a/d_a) + (d_b + 1) log_2(n_b/(d_b+2))
24     return gain
```

To minimize the number of memory moves required by the algorithm, we create a list of references to the documents in the collection so that only pointer swaps are required when exchanging two documents between partitions. Where possible, references are used to avoid expensive memory copy or move operations.

As described by Dhulipala et al. [10], the two different recursive calls of Algorithm 1 are independent and can be executed in parallel (Algorithm 1, line 14/15). For this reason, we also employ a fork-join computation model using the Intel TBB library[4]. A pool of threads is started and every recursion call is added into a pool of tasks, so that threads can assign jobs according to the TBB scheduling policy.

After splitting the document vector into two partitions, the algorithm computes the *term degree* for every partition. In order to do so, we precompute the degree for all the terms in each partition. Since every document contains distinct terms, we can again exploit parallelism though parallel_for loops. Given the

[4] https://www.threadingbuildingblocks.org/

size of the collections used and the fact that the degree computations can run simultaneously on different partitions, we use a custom array implementation, which requires a one-off initialization. In contrast to constant-time initialization arrays [13], our implementation uses a global counter, $C$, and two arrays of the same size, $V$ and $G$. The former of the two is used to store the actual values of the degrees and the latter to keep track of the validity of the data present in $V$ at the corresponding position. The following invariant is maintained:

$$V[i] \text{ is valid} \iff G[i] = C.$$

Once the vector is allocated, which happens only once since it is marked as thread_local in our implementation, the vector $G$ is initialized to 0. The counter $C$ is set to 1, which indicates that none of the variables in the array are valid. Thus, an increment of the counter corresponds to a clear operation of the values in the array. If a position of the array contains a non-valid value, a default value is returned. The intuition behind this arrangement is to allow easy parallelism, and to avoid reinitialization of large vectors.

Next, for a fixed number of iterations, *MaxIter*, the gain computation, sorting, and swapping of documents is repeated (Algorithm 1, line 3–12). Gain computation is particularly interesting because it can be very expensive, so we address this operation as follows. Since terms are typically shared among multiple documents, we adopt a cache to compute every term gain at most once. However, checking if a term cost has already been cached introduces a branch that is hard to predict by the CPU; intuitively, fewer documents are processed deeper in the recursion, which implies fewer terms shared and a lower probability for each of them to be in the cache. To avoid this misprediction, we develop two functions which only differ for the check performed in the cache, where both provide branch prediction information for whether the value is likely to be in the cache or not. Furthermore, to compute a single term cost, we use SIMD instructions, allowing four values to be processed in a single CPU instruction. Sorting is, again, done in parallel and, as for the parallel_for, we use the Intel Parallel STL[5] implementation. The swap function also updates the degrees of the two partitions, so recomputation is not needed for every iteration.

Finally, when the recursion reaches a segment of the document vectors which is considered small enough to terminate, a final sorting is applied to sort the otherwise unsorted leaves by their document identifiers.

## 4 Experiments

*Testing details* All of the algorithms are implemented in C++17 and compiled with GCC 7.2.0 using the highest optimization settings. Experiments are performed on a machine with two Intel Xeon Gold 6144 CPUs (3.50GHz), 512GiB of RAM, running Linux 4.13.0. The CPU is based on the Skylake microarchitecture, which supports the AVX-512 instruction set, though we did not optimize for such instructions. Each CPU has L1, L2, and L3 cache sizes of 32KiB, 1024KiB, and 24.75MiB, respectively. We make use of SIMD processor intrinsics in order to

[5] https://software.intel.com/en-us/get-started-with-pstl

| | | ≥ 4,096 | | Full Index | |
|---|---|---|---|---|---|
| Graph | $\|D\|$ | $\|T\|$ | $\|E\|$ | $\|T\|$ | $\|E\|$ |
| NYT | 1,855,658 | 10,191 | 457,883,999 | 2,970,013 | 501,568,918 |
| Wikipedia | 5,652,893 | 14,038 | 749,069,767 | 5,604,981 | 837,439,129 |
| Gov2 | 25,205,179 | 42,842 | 5,406,607,172 | 39,180,840 | 5,880,709,591 |
| ClueWeb09 | 50,220,423 | 101,676 | 15,237,650,447 | 90,471,982 | 16,253,057,031 |
| ClueWeb12 | 52,343,021 | 88,741 | 14,130,264,013 | 165,309,501 | 15,319,871,265 |
| CC-News | 43,530,315 | 76,488 | 19,691,656,440 | 43,844,574 | 20,150,335,440 |

**Table 1:** Properties of our datasets. We consider only postings lists with ≥ 4,096 elements when conducting the reordering, but apply compression on the entire index. For completeness, we show both groups of statistics here.

speed up computation. When multithreading is used, we allow our programs to utilize all 32 threads, and our experiments assume an otherwise idle system. The source code is available[6,7] for the reader interested in further implementation details or in replicating the experiments.

*Datasets* We performed our experiments using mostly standard datasets as summarized in Table 1. While most of these collections are readily available, we do note that the Wikipedia and CC-News collections are exceptions: both of these collections are temporal (and thus subject to change). To this end, we will make the raw collections available by request to any groups interested in repeating our experiments, following the best practices from Hasibi et al. [14].

- NYT corresponds to the New York Times news collection, which contains news articles between 1987 and 2007,
- Wikipedia is a crawl of English Wikipedia articles from the 22nd of May, 2018, using the Wikimedia dumps and the Wikiextractor tool[8],
- Gov2 is a crawl of `.gov` domains from 2004,
- ClueWeb09 and ClueWeb12 both correspond to the 'B' portion of the 2009 and 2012 ClueWeb crawls of the world wide web, respectively, and
- CC-News contains English news documents from the Common Crawl News[9] collection, from August 2016 to April 2018.

Postings lists were generated using Indri 5.11, with no stopword removal applied, and with Krovetz stemming. Furthermore, our results were tested for correctness by ensuring that the output of the reordered indexes matched the output of the original index for a set of test queries.

*Reordering parameters* For most of our collections, we apply the URL ordering on the index and consider this the `Natural` ordering. Two exceptions are the NYT and Wikipedia collections. For NYT, we apply crawl ordering, as all indexed

[6] https://github.com/pisa-engine/pisa
[7] https://github.com/pisa-engine/ecir19-bisection
[8] https://github.com/attardi/wikiextractor
[9] https://github.com/commoncrawl/news-crawl

sites have the same URL prefix. For Wikipedia, we use the ordering of the crawl as specified by the Wikipedia `curid`, which is a proxy for URL ordering (as these identifiers are monotonically increasing on the page titles, which are usually the same or similar to the long URLs). For the `Minhash` scheme, we follow Dhulipala et al. [10], and sort documents lexicographically based on 10 minwise hashes of the documents (or, adjacency sets). When running `BP`, we only consider posting lists of lengths $\geq$ 4,096 for computing the reordering, we run 20 iterations per recursion, and we run our algorithm to $MaxDepth = \log(n) - 5$ unless otherwise specified [10]. The file orderings are available for each collection for repeatability.

### 4.1 Compression Ratio

Our first experiment investigates whether we are able to reproduce the *relative compression improvements* that were reported in the work of Dhulipala et al. [10] for the `BP` algorithm, while also reproducing the baselines [23, 7]. Table 2 shows the effectiveness (in average bits per posting) of the various reordering techniques across each collection for a variety of state-of-the-art compression methods, including $\epsilon$-optimal Partitioned Elias-Fano (PEF) [20], Binary Interpolative Coding (BIC) [19], and Stream Variable Byte (SVByte) [17]. We also report the *LogGap* as described in Section 2.1. We find that the `BP` algorithm outperforms the baselines for every collection, with improvements over the closest competitor between 2 and 15% and up to around 50% against the `Random` permutation for PEF encoding. Similar improvements are observed for the other tested compression schemes. Our findings confirm that these reordering strategies generalize to collections outside of those tested experimentally in the original work, including Newswire data such as NYT and CC-News.

### 4.2 Efficiency

Next, we focus on the efficiency of our implementation. In the original work, the authors experimented with two implementations. One approach utilized a distributed implementation written in Java, which conducted the reordering across a cluster of "a few tens of machines." The other approach was a single-machine implementation, which used parallel processing across many cores. We opted to follow the single-machine approach as discussed in Section 3.1. We report the running time of `BP` for each dataset in Table 3.

Note that for these experiments, we used the `Natural` ordered index as input to `BP` (discussed further in Section 4.3). Clearly, our implementation of `BP` is very efficient, completing Gov2 in 28 minutes, and ClueWeb09 in 90 minutes. This is comparable to the timings reported in the original work, which reported Gov2 and ClueWeb09 taking 29 and 129 minutes, respectively. We must remark that our timings are not directly comparable to those from Dhulipala et al. for a few reasons. Firstly, our indexes were built using Indri, whereas they opted to use Apache Tika for their indexing, resulting in a different number of postings to process. Furthermore, subtle differences in servers such as clock speed and cache size can impact timings. In any case, we are confident that the `BP` algorithm can run efficiently over large collections, and can use whatever processing pipeline adopters have available. Another aspect of efficiency is memory consumption.

| Index | Algorithm | LogGap | PEF | BIC | SVByte |
|---|---|---|---|---|---|
| NYT | Random | 3.79 | 6.36 / 2.22 | 6.48 / 2.16 | 11.67 |
| | Natural | 3.50 | 6.31 / 2.20 | 6.23 / 2.13 | 11.62 |
| | Minhash | 3.18 | 5.91 / 2.19 | 5.79 / 2.11 | 11.51 |
| | BP | **2.61** | **5.24 / 2.13** | **5.06 / 2.04** | **11.33** |
| Wikipedia | Random | 5.12 | 8.03 / 2.20 | 8.01 / 1.98 | 12.45 |
| | Natural | 4.76 | 7.83 / 2.17 | 7.65 / 1.93 | 12.31 |
| | Minhash | 3.94 | 7.08 / 2.11 | 6.71 / 1.85 | 12.02 |
| | BP | **3.13** | **6.17 / 2.03** | **5.74 / 1.77** | **11.69** |
| Gov2 | Random | 5.05 | 7.96 / 2.97 | 7.93 / 2.53 | 12.47 |
| | Natural | 1.91 | 4.37 / 2.31 | 4.01 / 2.07 | 11.44 |
| | Minhash | 1.99 | 4.57 / 2.34 | 4.17 / 2.10 | 11.51 |
| | BP | **1.54** | **3.67 / 2.20** | **3.41 / 2.01** | **11.30** |
| ClueWeb09 | Random | 4.88 | 7.69 / 2.39 | 7.68 / 2.08 | 12.47 |
| | Natural | 2.71 | 6.12 / 2.20 | 5.36 / 1.84 | 11.71 |
| | Minhash | 3.00 | 6.46 / 2.23 | 5.77 / 1.87 | 11.79 |
| | BP | **2.38** | **5.49 / 2.12** | **4.84 / 1.79** | **11.52** |
| ClueWeb12 | Random | 5.08 | 7.99 / 2.39 | 7.95 / 2.09 | 12.91 |
| | Natural | 2.51 | 6.07 / 2.20 | 5.11 / 1.81 | 12.07 |
| | Minhash | 2.89 | 6.08 / 2.17 | 5.49 / 1.86 | 12.06 |
| | BP | **2.32** | **5.20 / 2.07** | **4.64 / 1.77** | **11.90** |
| CC-News | Random | 3.56 | 6.06 / 2.19 | 6.16 / 2.06 | 11.48 |
| | Natural | 1.49 | 3.38 / 1.91 | 3.26 / 1.73 | **10.92** |
| | Minhash | 1.95 | 4.49 / 2.02 | 4.12 / 1.82 | 11.08 |
| | BP | **1.39** | **3.31 / 1.90** | **3.11 / 1.72** | **10.92** |

**Table 2:** Reordered compression results for all six collections. We report the bits per edge for representing the DocIDs and the frequencies (DocID / Freq). Note that we omit the frequency data for SVByte as reordering does not impact the compression rate for this codec.

Dhulipala et al. report that their implementation "utilizes less than twice the space required to store the graph edges." While this is hard to interpret (how was the graph edge space consumption calculated?), we provide some intuition on our memory consumption as follows. On our largest collection, CC-News, the BP algorithm has a peak space consumption of 110GiB, which includes the graph representation of the dataset. Given that the compressed forward index for CC-News consumes 25GiB, it seems that we have a higher memory footprint than the original implementation (which would use up to 75GiB). It is important to note that this is due to our caching approach, which incurs a higher memory footprint for faster execution time. Of course, alternative caching strategies may allow for lower memory consumption at the cost of a slower run time.

### 4.3 Parameters and Initialization

First, we investigate the impact that the number of iterations has on the algorithm. Dhulipala et al. showed that while on the higher levels of the recursion

| NYT | Wikipedia | Gov2 | ClueWeb09 | ClueWeb12 | CC-News |
|---|---|---|---|---|---|
| 2 | 5 | 28 | 90 | 86 | 97 |

**Table 3:** Time taken to process each dataset with recursive graph bisection, in minutes.

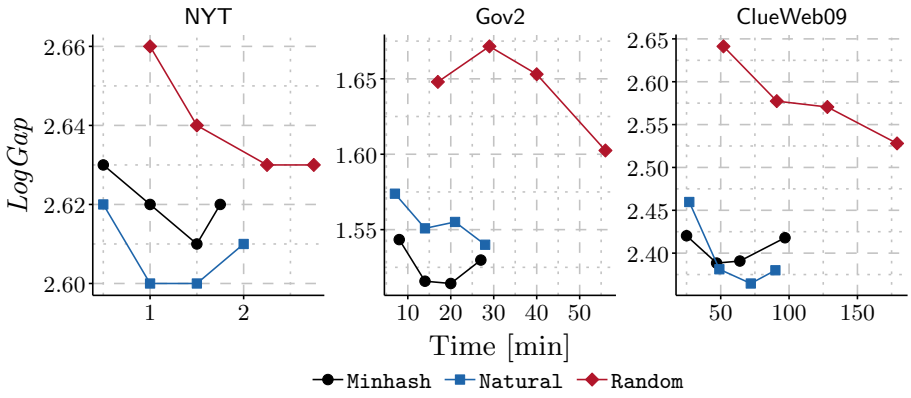


**Fig. 1:** *LogGap* cost of the three different input orderings after running BP as the number of iterations increases.

we approach convergence after just a few iterations, around $MaxIter = 20$ iterations are required at deeper levels. We are interested in understanding how effective the BP algorithm is at producing a good ordering when we do less iterations, as less iterations results in improved run-time efficiency. To measure this, we ran the algorithm across the collections setting the maximum number of iterations $MaxIter = \{5, 10, 15, 20\}$. Figure 1 shows the resulting trade-off in terms of *LogGap* and execution time. We can make a few observations from this figure. Firstly, the more optimal the input graph, the less iterations required to reach a reasonable ordering. This is intuitive, as a better input ordering implies that document clustering is already somewhat reasonable, meaning less work is required to further improve the clustering. Secondly, the quality of the input graph also seems to impact the run time of the algorithm. For example, examine Figure 1 (right), which shows the ClueWeb09 collection. Using either the `Natural` or `Minhash` inputs achieves competitive compression levels using only 10 iterations, which takes around 50 minutes. On the other hand, the `Random` input takes longer to process in each iteration, and results in a less effective final ordering. Similar results were found on all tested collections. `Random` orderings are slower to compute for two main reasons. Firstly, on each iteration, more vertices are moved, which takes longer to process. Secondly, it is less likely that the convergence property will be met using a `Random` ordering, which results in more iterations in total (Algorithm 1, Line 11 and 12). Therefore, we recommend using a `Natural` or `Minhash` ordered index as input to the BP algorithm where possible, and setting $MaxIter = 20$ for to ensure a good level of compression. If

the run time is critical, using smaller values of *MaxIter* allows trading off some compression effectiveness for a faster total processing time.

Our next experiment investigates the effect of initialization on the performance of the BP algorithm. Recall that in Algorithm 1, $D$ must be partitioned into sets $D_1$ and $D_2$. As discussed by Dhulipala et al., the initialization of these sets may impact the quality of the final ordering of the vertices. Our implementation uses a generic sort-by-identifier approach to do this partitioning, so the partition is made by first sorting the documents in $D$ by their identifiers, and then splitting it into two equal sized subgraphs. Therefore, the initialization approach used is the same as the ordering of the input collection. Based on the original work, we expect the initialization to have little impact on the final compression ratio, with no clear best practice (and negligible differences between the resultant compression levels). To test this, we ran BP using three different initialization orderings: Random, Natural, and Minhash. Our results confirm that the initialization order used to initialize $D_1$ and $D_2$ only has a very moderate impact on the efficacy of the bisection procedure. In particular, the largest differences in the *LogGap* for the resulting orderings was always within $\pm 5\%$. We found that that there was no consistently better approach, with each initialization yielding the best compression on at least one of the tested collections. This effect can be observed in Figure 2, by comparing each line at depth $\log(n) - 5$. For example, running BP with Random initialization results in the best ordering on the Wikipedia dataset, whereas Minhash initialization is the best on ClueWeb09.

Our final experiment examines both the depth of the recursion and the potential impact of the initialization of $D_1$ and $D_2$ on the convergence of the algorithm. For each collection, we run the BP algorithm for each recursion depth between 1 and 26 since the collection with the largest number of documents, ClueWeb12, has $\lceil \log(n) \rceil = 26$. Again, we use three varying approaches of initializing the sets $D_1$ and $D_2$, as the initialization may impact the depth at which the algorithm converges. Figure 2 shows the results for this experiment. We reiterate that the input ordering is the same as the initialization approach applied, hence the different starting points of each line. Interestingly, the ordering of the input/initialization approach does not impact the convergence level of the BP algorithm. We confirm that the recommended heuristic of using $\mathit{MaxDepth} = \log(n) - 5$ does indeed fit with our implementation and the additional collections we tested, with marginal (if any) gains following at further depths. Another interesting observation is that even in cases where the input ordering is very close to the compression level of the output ordering from BP (primarily in the case of the CC-News collection where the input is the Natural index), the BP algorithm still takes around 14 levels of recursion to begin improving upon the input ordering, reaching convergence at around level 19 or 20 as expected.

## 5 Conclusion and Discussion

During this reproducibility study, a lot of effort was spent on optimizing the implementation. In order to achieve the efficiency that is described in our experiments, considerable thought was put into various prototype algorithms and

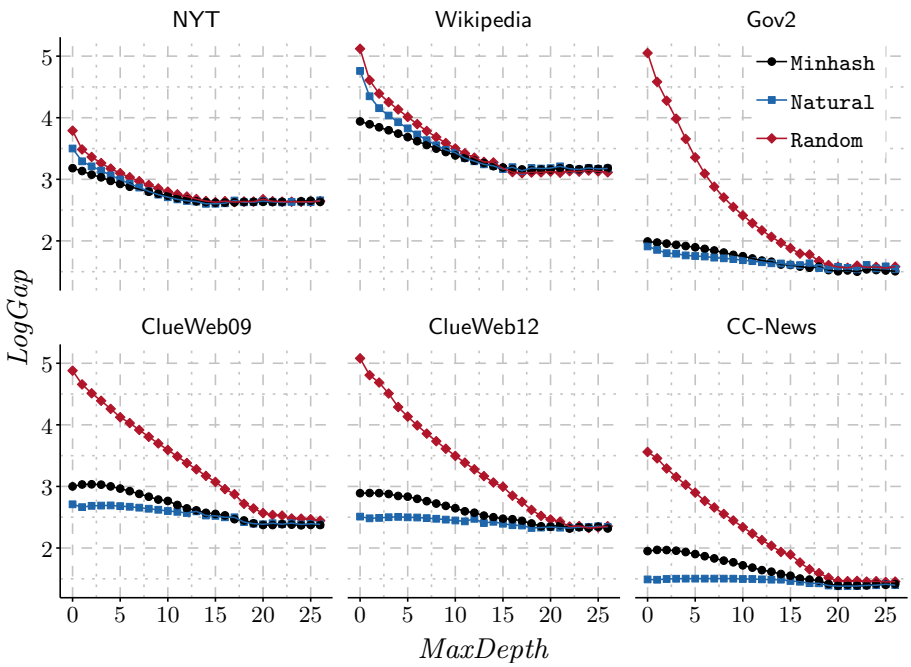


**Fig. 2:** *LogGap* cost of the three different input orderings as *MaxDepth* is increased. *LogGap* cost is reported up to a depth of 26 for all collections.

varying approaches before a final version was produced. While the original paper primarily focuses on the theoretical reasoning behind the BP algorithm, this leaves less room for explaining specific implementation details which are important in practice. Since the source code was not released in the original paper, this contributed to the difficulty of reproducibility. However, we are confident that the algorithm presented in the original work and the findings based on this algorithm are valid, as shown in this reproducibility study. By making our implementation available, we hope to stimulate further research in this interesting area of efficiency. Finally, we believe that future research should make the applied index ordering known, as is already done for other experimental factors such as stemming or stopping. This is of course important for the reproducibility of efficiency experiments conducted across inverted indexes, for both query processing speeds and compression numbers.

**Acknowledgments** This work was supported by the National Science Foundation (IIS-1718680), the Australian Research Council (DP170102231), and the Australian Government (RTP Scholarship).

## References


[1] J. Arguello, F. Diaz, J. Lin, and A. Trotman. SIGIR 2015 workshop on reproducibility, inexplicability, and generalizability of results (RIGOR). In *Proc. SIGIR*, pages 1147–1148, 2015.

[2] R. Blanco and Á. Barreiro. Document identifier reassignment through dimensionality reduction. In *Proc. ECIR*, pages 375–387, 2005.

[3] R. Blanco and Á. Barreiro. Characterization of a simple case of the reassignment of document identifiers as a pattern sequencing problem. In *Proc. SIGIR*, pages 587–588, 2005.

[4] R. Blanco and Á. Barreiro. TSP and cluster-based solutions to the reassignment of document identifiers. *Inf. Retr.*, 9(4):499–517, 2006.

[5] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. DCC*, pages 342–352, 2002.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[7] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proc. SIGKDD*, pages 219–228, 2009.

[8] M. Crane, J. S. Culpepper, J. Lin, J. Mackenzie, and A. Trotman. A comparison of Document-at-a-Time and Score-at-a-Time query evaluation. In *Proc. WSDM*, pages 201–210, 2017.

[9] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *Proc. WSDM*, pages 1–1, 2009.

[10] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proc. SIGKDD*, pages 1535–1544, 2016.

[11] S. Ding and T. Suel. Faster top-$k$ document retrieval using block-max indexes. In *Proc. SIGIR*, pages 993–1002, 2011.

[12] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *Proc. WWW*, pages 311–320, 2010.

[13] K. Fredriksson and P. Kilpeläinen. Practically efficient array initialization. *Soft. Prac. & Exp.*, 46(4):435–467, 2016.

[14] F. Hasibi, K. Balog, and S. E. Bratsberg. On the reproducibility of the TAGME entity linking system. In *Proc. ECIR*, pages 436–449, 2016.

[15] D. Hawking and T. Jones. Reordering an index to speed query processing without loss of effectiveness. In *Proc. ADCS*, pages 17–24, 2012.

[16] A. Kane and F. Wm. Tompa. Split-lists and initial thresholds for WAND-based search. In *Proc. SIGIR*, pages 877–880, 2018.

[17] D. Lemire, N. Kurz, and C. Rupp. Stream vbyte: Faster byte-oriented integer compression. *Inf. Proc. Letters*, 130:1–6, 2018.

[18] A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini. Faster BlockMax WAND with variable-sized blocks. In *Proc. SIGIR*, pages 625–634, 2017.

[19] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Inf. Retr.*, 3(1):25–47, 2000.

[20] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. SIGIR*, pages 273–282, 2014.

[21] M. Richardson, A. Prakash, and E. Brill. Beyond pagerank: Machine learning for static ranking. In *Proc. WWW*, pages 707–715, 2006.

[22] W-Y. Shieh, T-F. Chen, J. J-J. Shann, and C-P. Chung. Inverted file compression through document identifier reassignment. *Inf. Proc. and Man.*, 39(1):117–131, 2003.

[23] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. ECIR*, pages 101–112, 2007.

[24] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.