Fast and Efficient Bipartite Graph Compression

Jongjin Kim
Dept. of Computer Science
Seoul National University
j2kim99@snu.ac.kr

Kahyun Park
Dept. of Computer Science
Seoul National University
kahyun.park@snu.ac.kr

Jaemin Park
Dept. of Computer Science
Seoul National University
ytreqw97@snu.ac.kr

Jaeri Lee
Dept. of Computer Science
Seoul National University
jlunits2@snu.ac.kr

December 7, 2022

Abstract

How can we save a huge bipartite graph on storage? In this project, we propose IBSO to compress the adjacency matrix of the given bipartite graph. IBSO reorders vertex indices of the graph to reduce the number of nonzero blocks in the adjacency matrix.

1 Introduction

The problem we want to solve is the following:

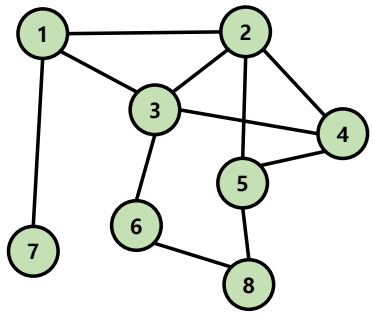
- GIVEN: a bipartite graph $G = (V, E)$
- FIND: a permutation of vertices P
- to MINIMIZE: the number of nonzero blocks of size $k \times k$ in the adjacency matrix

Graph compression through compression-friendly vertices reordering is a common strategy to handle large-scale real-world graphs. Figure 8 shows the importance of the vertices reordering. Figures 8(b) and 8(d) describe the same graph with different permutations of vertices. However, the number of nonzero blocks in Figure 8(d) is 10 while that in Figure 8(b) is 12. Thus, we can store the graph by keeping only 10 submatrixes instead of 12 through vertices reordering, which requires 16.7% reduced storage capacity. Hence, it is important to find an optimal permutation of vertices to efficiently store the matrix.

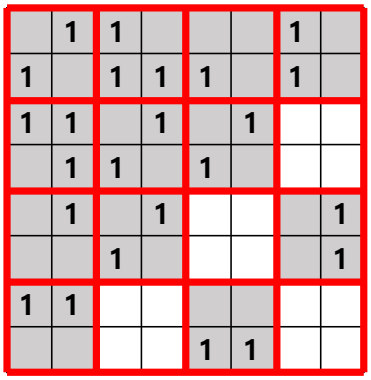
A lot of real-world data such as user-item purchase data or music-artist data form bipartite graphs. However, the vertices reordering of bipartite graphs differs from the normal

graph reordering. An adjacency matrix of a bipartite graph has a fundamental sparsity because of the definition of a bipartite graph. Figure 9 demonstrates the fundamental sparsity of a bipartite graph. Colored entries of Figure 9(b) are fixed to zero since vertices in the same partition cannot be connected. Thus, the density of the block cannot be maximized if the block covers the vertices of both partitions so the number of nonzero blocks increases to contain all edges of the matrix. However, previous vertices reordering methods do not consider this problem of bipartite graphs, which leads to unoptimized compression of the given bipartite graph.

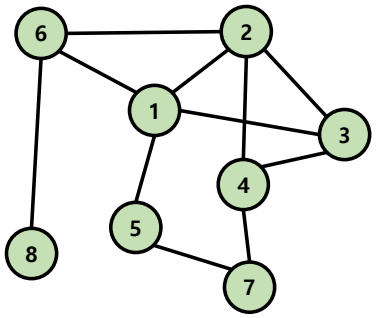
We will propose IBSO, a novel bipartite graph compression method that carefully considers the property of the bipartite graph while reordering vertices.



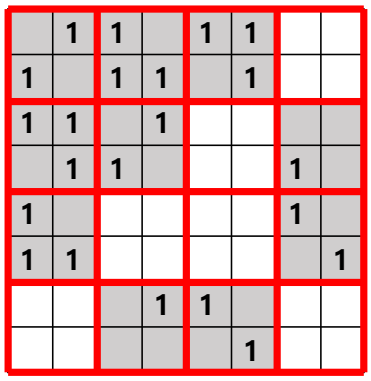
(a) Original graph



(b) Adjacency matrix

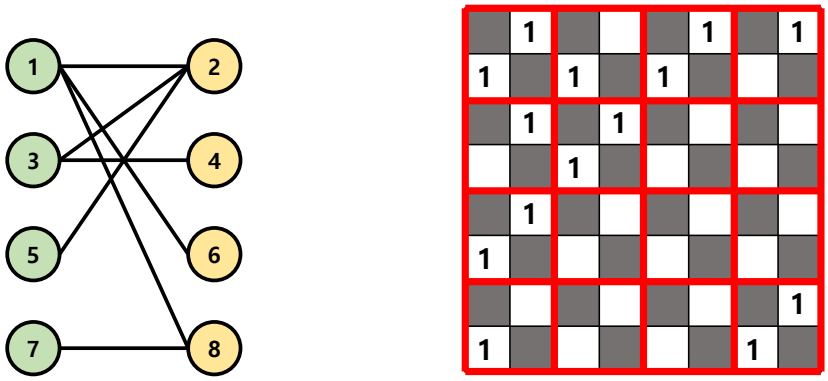


(c) Reordered graph



(d) Reordered adjacency matrix

Figure 1: Example of the compression-friendly reordering of the given graph. (b) and (d) are adjacency matrixes of (a) and (c), respectively. They define the same graph structure with different vertices permutations. However, (d) has fewer nonzero blocks than (b), so it requires less storage capacity.



(a) Bipartite graph (b) Adjacency matrix

Figure 2: Example of (a) a bipartite graph and (b) its adjacency matrix. Colored elements in the adjacency matrix are fixed to zero, which leads to the fundamental sparsity of nonzero blocks.

2 Survey

Next, we list the papers that each member read, along with their summary and critique.

2.1 Papers read by Jongjin Kim

The first paper introduces BP algorithm proposed by Dhulipala et al [8].

- *Main idea*: to compress the adjacency matrix with index distance encoding, it is important to solve the minimum logarithmic gap arrangement (MLogGapA) problem. The goal of this problem is to find a new permutation of vertices in the given graph such that the sum of logarithmic gaps between consecutive neighbor indices is minimized. BP solves this problem with a divide and conquer strategy motivated by the intuition from solving MLogGapA in a bipartite graph. Similar to quicksort algorithm, BP repeats bisecting given vertices into two groups such that minimizing gaps in each group. BP achieves the lowest logarithmic gap among competitors, which leads to compression friendly adjacency matrix.
- *Use for our project*: BP is one of our competitors since BP and our project both aim to reorder given vertices into compression-friendly order. The intuition behind their bisecting procedure could be useful since they also try to achieve certain properties in the adjacency matrix of the bipartite graph.
- *Shortcomings*: BP is designed to reorder vertices for a specific matrix encoding scheme, which may not suit some practical tasks such as slicing or matrix multiplication.

The second paper is the Rabbit Order paper by Arai et al [2].

- *Main idea*: this paper aims to improve the locality of the given graph by index reordering that makes dense non-zero blocks in an adjacency matrix. Rabbit Order algorithm

solves this problem by arranging vertices into groups of dense subgraphs. Rabbit Order repeats merging subgraphs which gain maximum density to build a hierarchical structure of communities. Then, Rabbit Order performs DFS on the hierarchical structure to obtain the permutation of vertices. The extensive evaluation shows that Rabbit Order improves compression performance compared to other existing reordering methods.

- *Use for our project*: Rabbit Order is a compression-friendly graph reordering algorithm that would be a baseline for our project. The iterative strategy to gain the maximum desired property seems helpful to find the ideal permutation of the partition of a given bipartite graph. This paper also presents an informative summary of research on graph reordering through a related work section.
- *Shortcomings*: Rabbit Order tries to make dense blocks on the adjacency matrix through community detection. However, communities of the bipartite graph have fundamental sparsity because of the property of the bipartite graph. Thus, the densities of blocks in Rabbit Order's adjacency matrix would be low.

The third paper is the shingle ordering paper by Chierichetti et al [6].

- *Main idea*: this paper formulates MLogA and MLogGapA problems, which define the compression-friendly permutation of the given graph. Shingle ordering is a heuristic using Jaccard similarity to find optimal compression-friendly order. If vertices are arranged in the shingles' ascending order, similar vertices would be joined since they would have the same shingle with a high probability. Thus, shingle ordering sorts vertices with their shingles to obtain compression-friendly permutation. Extensive experiments show that shingle ordering provides better permutation than other ordering schemes to compress real-world graphs.
- *Use for our project*: Shingle ordering provides a simple yet effective baseline for compression-friendly graph ordering. It also gives an idea to separate vertices of a bipartite graph into respective partitions, since vertices of different partitions cannot share the same shingle because of the property of bipartite graphs.
- *Shortcomings*: Shingle ordering is a loose heuristic highly dependent on the random seed. There are a lot of potential improvements to optimize the detailed order of vertices based on the intuition of shingle ordering.

2.2 Papers read by Kahyun Park

The first paper is the BFS graph compression by Apostokico and Drovandi [1].

- *Main idea*: previous works have focused on assigning indices considering the lexicographical ordering of the web URLs, thus usage of techniques worked only on the Web Graph. By compressing graphs using BFS, the topological structure of the Web Graph is used instead of the lexicographical ordering of the URLs. Compression by BFS performs breadth-first traversal of a given graph G and assigns indices to each node and its neighbors consecutively. Then, consecutive chucks of l nodes are compressed

separately, where l is the compression level. The adjacency list A_i of each node $v_i \in V$ in a chunk is encoded in increasing order. Encoding by BFS enables graph compression without prior knowledge of a graph with a high compression ratio.

- *Use for our project*: Our project focuses on graph compression-friendly reordering of bipartite(multipartite) graphs. BFS can encode bipartite graphs and it can be used on bipartite graph reordering. Graph compression by BFS can also be used as our baseline model.
- *Shortcomings*: Graph compression by BFS may cause 4 types of redundancy and it might not be suitable for large real-world bipartite graphs.

The second paper is the ORDER-INDEX paper by Blandford and Blelloch [3].

- *Main idea*: on search engines, lists of documents with difference coding in an inverted index are well compressed when lists have high locality. ORDER-INDEX creates locality in individual posting lists by permuting the document number. ORDER-INDEX uses the cosine measure to compute document similarity and construct a document-document similarity graph with BUILD-GRAPH. Then with the SPLIT-INDEX algorithm, ORDER-INDEX partition the graphs created with BUILD-GRAPH. Lastly, ORDER-CLUSTERS apply rotation to the clustering tree for optimization of the ordering.
- *Use for our project*: ORDER-INDEX used document reordering to improve index compression by increasing locality. Our project aims to find an ordering method to help compression in bipartite graphs. It gives a simple but effective intuition to our project.
- *Shortcomings*: increasing locality may not be of great help in compressing bipartite graphs. Also, hierarchical clustering in multipartite graphs is quite complicated

The third paper is the label propagation community detection paper by Raghavan et al [13].

- *Main idea*: given a network without a priori information about likely communities, optimization with a specific measure of community strength is generally used. The label propagation community detection uses only structural information on networks and does not optimize a chosen measure. When given a network, the label propagation algorithm initializes every node with unique labels and propagates labels. Then each node chooses to join the community where the maximum number of its neighbors. The algorithm performs the process iteratively, each node label is updated at every step. The process continues until the whole node labels are unchanged.
- *Use for our project*: label propagation community detection works well on bipartite graphs. By detecting community structure in a network, we can adopt another compression algorithm for each community.
- *Shortcomings*: the computation is still expensive for real-world graphs. Randomly given labels may affect the performance of the method.

2.3 Papers read by Jaemin Park

The first paper was 'The WebGraph Framework I : Compression Techniques' by Boldi. [4]

- *Main idea*: being one of the pioneering works for compressing large-scale networks, the paper proposes the use of lexicographical order of URLs for compressing web graphs. The proposed method relies on two assumptions, which are locality and similarity. Locality indicates that most links contained in a page lead to other pages within the same host, giving it a navigational nature. Similarity, on the other hand,
  will lead to pages if linked to the same host, whereas similarity indicates that pages with the same hosts will often share the same links.
- *Use for our project*: this paper points out some of the possible features that a network might possess, which are locality and similarity. Through empirical analysis of our problem, we will be able to diagnose whether our target network also possesses such features. If so, or if not, the idea proposed in the paper can be further extended to address such issues.
- *Shortcomings*: the paper is based on the robust assumption of links being navigational and having strong similarities. This eventually exerts limitations on the method, not allowing it to be applied to networks without such traits.

The second paper was 'MapReduce: Simplified Data Processing on Large Clusters' by Dean and Ghemawat. [7]

- *Main idea*: given a large dataset, the paper proposes designing a programming model that is automatically parallelized to process and generate data. The model consists of map and reduce functions. The map function generates intermediate key/value pair when given key/value pair. The reduce function takes in the output from map function as input and combines the key/value pair with the same key. MapReduce is useful in that it can process and generate large-scale data in parallel and distributed systems. It achieves high scalability for large-scale data and can generalize the model to process other real-world problems.
- *Use for our project*: although the efficient data processing method proposed by the paper is not directly related to our project, it can be inferred that an efficient data storage method can be derived from efficient data processing methods. MapReduce conveys an efficient key/value based data processing method that enables users to locate highly related pages in a short amount of time. Such a method could be applied to hashing data into different keys so that the hashed output shows a high representation of the original data.
- *Shortcomings*: since there are no specific conditions for data, the model might not be able to show high performance in other networks due to data-dependency. Moreover, the data used does not include streaming data, when streaming data is often considered important. The model could be further developed to operate in streaming settings as well.

The third paper was 'PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations' by Kang. [10]

- *Main idea*: the paper proposes a generalized matrix-vector multiplication (GIM-V) method which can represent many graph mining operations on map-reduce framework. The graph mining system PEGASUS was implemented on Hadoop platform, and the optimized method showed highly effective scale-up, linear running time, and 5 times faster performance than the nave version of GIM-V.
- *Use for our project*: the paper possesses novelty in that it can represent many graph mining algorithms in the form of matrix-vector multiplication. It generalized multiplication operation into GIM-V, and designed the method to work on map-reduce framework. If we were to use the map-reduce framework in organizing our data, PEGASUS could be used in our compression process.
- *Shortcomings*: despite the generalization capabilities of the model, the paper only addresses map-reduce framework as its distribution system. The paper would have had more generalization capability if it had mentioned the needed features of a distribution system suited for this model.

2.4 Papers read by Jaeri Lee

The first paper was the wavelet paper by W.Fan et al. [9]

- *Main idea*: query preserving compression method builds a smaller graph that has the information needed for answering queries. Given a class of queries, it computes the equivalence classes of nodes. The paper also studies incremental query preserving compression efficiently adapting to changes in the real-world graph. This allows us to perform the graph compression only once and easily update the compressed graph when there are changes.
- *Use for our project*: the authors point out that it is also important to consider the change in real-life graphs on graph compression. It may be meaningful to consider changes in graphs in our proposed method.
- *Shortcomings*: while the method may be effective for simple queries, it is less effective for complex queries. Also, the original graph is not reconstructible.

The second paper was the S-node paper by Raghavan and Gracia-Molina [12]

- *Main idea*: for space-efficient Web graph representation, S-Node partitions the Web graph and represents it as a set of smaller two-level directed graphs: A supernode graph, a top-level directed graph, is composed of "supernodes" and "superedges", which represent partitions of a Web graph and their connections, respectively, and an intranode graph which represents the interconnections between the pages that belong to a partition. S-node also identifies important partitions considering pages' adjacency list, domain, and host, for efficient local and global access. With S-node representation, in-memory processing of very large Web graphs is possible, and query execution time is noticeably shortened.

- *Use for our project*: while S-node is not a reordering-based method but an aggregation-based method of graph compression, it still gives an idea of partitioning a graph according to its characteristics. We could also use some known characteristics of graphs to select important partitions from other types of graphs and reorder the graph using the information.
- *Shortcomings*: S-node representation is limited to the representation of Web graphs, so the characteristic they found to identify important partitions in a graph may not apply to other types of graphs.

The third paper was the wavelet paper by Navlakha et al. [11]

- *Main idea*: Navlakha et al. represent the graph R based on the MDL principle with a high-level summary graph S, which highlights dominant trends to make visualization effective, and a set of edge corrections C for reconstructing the graph. The method also provides approximate representation by minimizing user-specified bounded error for fast graph query operations. It is the first work that applies the MDL principle for graph compression, and the method is advantageous in that it is parameterless and applicable to any graph.
- *Use for our project*: Navlakha et al. provide a powerful compression method by approximate representation, yet the original graph is reconstructible. It gives the idea that the original graph does not always have to be fully preserved to be reconstructible.
- *Shortcomings*: it is hard to find the representation with minimum cost because the algorithms used to minimize the cost are heuristic.

3 Proposed Method

We address the following challenges to reduce the number of nonzero blocks in an adjacency matrix:

- *How to remove the fundamental sparsity of the bipartite graph?*
- *How to get benefit from the property of bipartite graph?*
- *How to order vertices to form denser blocks in an adjacency matrix?*

The main ideas of IBSO are summarized as follows:

- *Separation.* IBSO separates indices of two partitions to gather fundamentally blank elements into limited areas.
- *Iterative Process.* IBSO iteratively sorts each partition to take advantage of independency between two partitions.
- *Bucket Shingle Ordering.* IBSO sorts vertices based on the bucket min hash function to group vertices in similar blocks.

3.1 Separation

To avoid the sparsity of nonzero blocks made from the inner partition edge, IBSO groups the nodes of each partition and separates those in the permutation as shown in Figure 10. Thus, fundamentally blank entries and possible nonzero entries are isolated, which allows nonzero blocks to exclude vacant elements and achieve the best density.

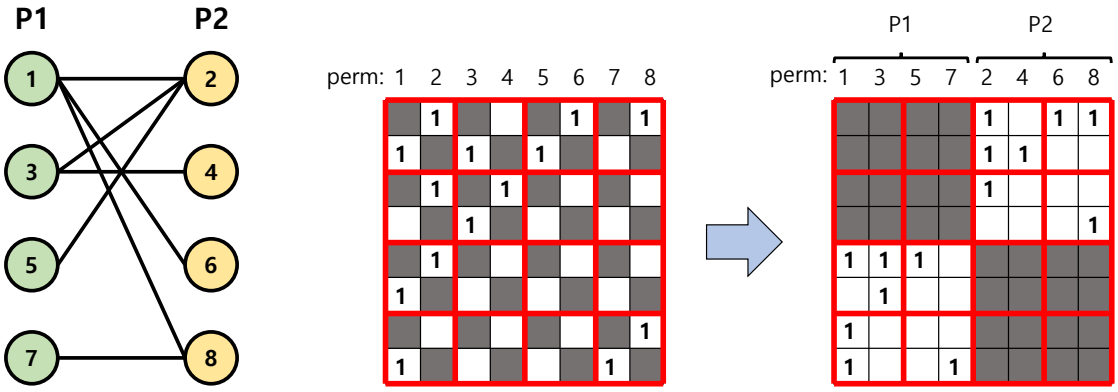


Figure 3: The separation between two partitions on the permutation groups meaningless edges of the bipartite graph's adjacency matrix in top-left and bottom-right corners. In this way, nonzero blocks do not contain any inner partition edge.

3.2 Iterative Process

We focus on the remaining area of the bipartite graph's adjacency matrix. They are composed of rows representing one partition and columns representing another partition. Thus, we need to find the permutation of each partition to form denser blocks in this area. We call the bottom-left half of this area a *CPS (cross partition submatrix)* in the rest of this paper. Unlike permutations of rows and columns in a normal graph's adjacency matrix, permutations of rows and columns in CPS are independent. In other words, we may exchange the order rows in CPS without interupt the order of columns.

It is easier to find the optimal permutation of rows under the fixed order of columns. Thus, IBSO alternately freezes the permutation of one partition and finds the optimal permutation of another partition. In this way, IBSO repeatedly solves "easier" problems to find the optimal permutation of vertices in a bipartite graph, which would not be possible in general graphs.

3.3 Bucket Shingle Ordering

Assume that the order of columns in CPS is given. How can we find the order of rows which makes the fewest nonzero blocks? To reduce the number of nonzero blocks, we need to make

each nonzero blocks denser. In other words, adjacent rows need to have nonzero elements in similar positions which are part of same blocks.

We propose a *BSO (Bucket Shingle Ordering)* heuristic to find similar rows. We define a bucket min-hash function as follows:

$$f_\sigma(r) \quad = \quad min_{r[i]=1}(\sigma(\left\lfloor \frac{i}{s} \right\rfloor)),$$

where r is a row, σ is a random permutation of blocks, and s is a size of the block. Note that this function indicates the leftmost nonzero chunk when we split the row into chunks of lenth s and reorder chunks in σ order. BSO performs bucket min-hash function on each row several times with different σs to obtain the fingerprint of each row as a tuple of hash values. Then, BSO sorts rows based on their fingerprints. In the same way with proving relationship between min-hash function and the Jaccard similarity [5], it can be shown that $P(f_\sigma(r_1) = f_\sigma(r_2))$ for the given r_1 and r_2 is a block-wise similarity between two rows. Thus, BSO locates rows that have nonzero entries in similar block closer, which leads to denser and fewer nonzero blocks in adjacency matrix.

3.4 Algorithm

Algorithm 2 shows the full process of IBSO, where BSO is a bucket shingle ordering function which takes a bipartite graph, two partitions, and a permutation of one partition as inputs and returns a permutation of another partition as an output. Since that time complexity of the bucket min hash function is linear to the number of nonzero elements in the row, each BSO function requires $O(|E|)$ time to get hash values and $O(|V| \log |V|)$ time to sort vertices. Thus, the total time complexity of IBSO is $O(|E| + |V| \log |V|)$.

Algorithm 1 IBSO (Iterative Bucket Shingle Ordering)

Require: Bipartite graph G, Partitions of vertices V_1, V_2
1: initialize the permutations of partitions P_1, P_2 //Separation
2: **for** i in range($iter$) **do** //Iterative Process
3: $P_1 \leftarrow \text{BSO}(G, V_1, V_2, P_1)$ //Bucket Shingle Ordering
4: $P_2 \leftarrow \text{BSO}(G, V_2, V_1, P_2)$
5: **end for**
6: $rtn \leftarrow \text{concat}(P_1, P_2)$
7: **return** rtn

4 Experiments

We perform experiments to answer the following questions:

Table 1: Summary of datasets.

Dataset	Nodes	Partition 1	Partition 2	Edges	Density(%)
MovieLens-1M[1]	9,746	6,040	3,706	1,000,209	4.4684
Marvel[2]	19,090	6,439	12,651	96,104	0.1179
FilmTrust[3]	3,579	1,508	2,071	35,492	1.1364
The Movies[4]	9,737	671	9,066	100,003	1.6438
NIPS[5]	1,723	1,320	403	1,320	2.4813
IMDB[6]	50,665	9,615	41,050	191,784	0.0485

Q1. **Performance**: How well does IBSO compress bipartite graphs compared to other methods?
Q2. **Speed**: Is IBSO scalable to large-scale real-world graphs?
Q3. **Visualization**: How does the adjacency matrix of IBSO look like?

4.1 Experimental Setup

We introduce our experimental setup including datasets and baseline approaches.

Datasets. We used five real-world bipartite graph data as summarized in Table 1. MovieLens-1M dataset contains movie rating constructed by the GroupLens research group which can be represented as a bipartite graph of users and movies. Marvel dataset is a bipartite graph of heroes of Marvel comics and the comic in which they appear. FilmTrust dataset and the Movies dataset are user-movie bipartite graph crawled from the FilmTrust website and the Movies website, respectively. NIPS dataset is a bipartite graph of authors and their papers in Neural Information Processing Systems 2015. IMDM dataset shows actors and films they acted on from IMDB feature films and documentaries as a bipartite graph.

Baselines. We compare IBSO with previous methods for compression-friendly reordering. We add a separation step before several unipartite graph oriented methods to make baselines for bipartite graph. We list our baselines as follows:

- Unipartite methods
  - **Random** orders nodes in random way.
  - **DegSort** orders nodes in decreasing order of degree of the nodes.
  - **BFS** orders nodes based on Breadth-First Search.
  - **SO** orders nodes based on their shingles made with min-hashes.

[1] `https://grouplens.org/datasets/movielens/1m/`
[2] `https://www.kaggle.com/code/ibauman/networkx-bipartite-unipartite-graphs/data`
[3] `http://konect.cc/networks/librec-filmtrust-trust/`
[4] `https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset`
[5] `https://www.kaggle.com/datasets/benhamner/nips-2015-papers`
[6] `https://www.kaggle.com/datasets/darinhawley/imdb-films-by-actor-for-10k-actors`

  - **SlashBurn** repeats gathering high-degree nodes and burning them.

- Bipartite methods
  - **Bi-Random** separates two partitions and then randomly orders nodes.
  - **Bi-DegSort** separates two partitions and then orders nodes in decreasing order of degree of the nodes.
  - **Bi-SO** separates two partitions and then orders nodes based on groups made with min-wise hashes.
  - **FastPI** separates two partitions and then repeats gathering high-degree nodes and burning them.

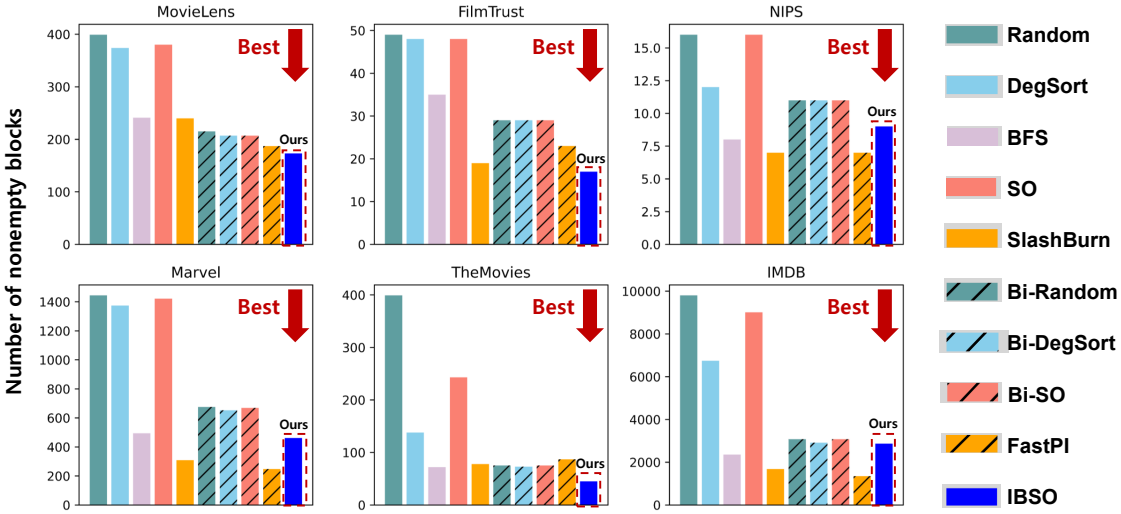


Figure 4: Number of nonempty blocks in each dataset after performing each method. IBSO has the least nonempty blocks in MovieLens-1M, FilmTrust and The Movies dataset and third least in Marvel, IMDB and NIPS dataset. IBSO shows the high performance overall.

4.2 Performance (Q1)

We compare IBSO and competitors in terms of the number of nonempty blocks in adjacency matrix on six real-world datasets as shown in Figure 4. Note that result with less number of nonempty blocks is better to compress. We set the size of each block as 512×512.

IBSO achieves the best performance in MovieLens-1M, FilmTrust, and The Movies dataset, having the least number of nonzero blocks. Also it has the third least number of nonempty blocks in Marvel, IMDB, and NIPS dataset. IBSO works the best on small

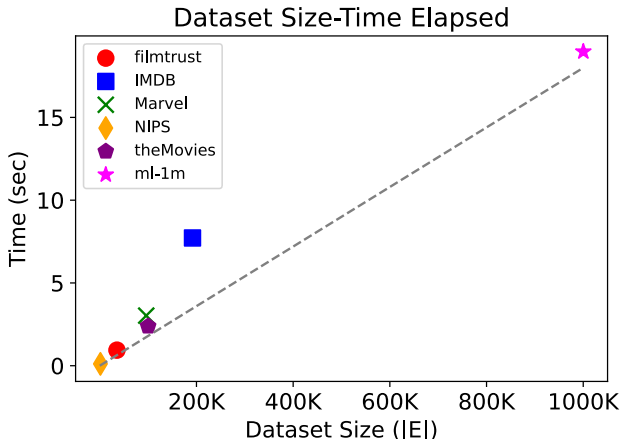


Figure 5: Performing time of IBSO. The performing time of IBSO increases linearly along with the size of dataset, showing that IBSO is efficient in performance time.

datasets like MovieLens-1M, FilmTrust, and The Movies dataset. However, it outperforms most of the previous methods also in experiments with large datasets. To conclude, IBSO performs better than most of the other methods and it has comparable performance to the most recent method Fast-PI.

4.3 Speed (Q2)

To verify the scalability of IBSO, we measure the performing time of IBSO in six real-world datasets and examine their change along with the size of the datasets. As shown in Figure 5, the performing time of IBSO increases linearly along with the size of datasets. The performing time of IBSO only increases about 3.5x when the size of the dataset increases about 5x, as shown in time difference of The Movies dataset and the Movielens dataset. Since performing time of many other methods increases exponentially when the size of datasets increases, IBSO is timely efficient and scalable.

4.4 Visualization (Q3)

We visualize adjacency matrices of IBSO and competitors using spyplot as shown in Figure 6. We find that a good method gathers zero entries and composes blank areas in the adjacency matrix. BFS makes repeating blank squares pattern because odd depth nodes and even depth nodes from the root node are from different partitions so those partitions are separated in the vertices permutation. SlashBurn makes large blank areas outside of the bow and arrow pattern which represents shattered graph fragments. Bipartite-oriented methods form a 2×2 checkerboard pattern because of the separation step as shown in Figure 10.

Meanwhile, adjacency matrices of Bi-SO and IBSO look similar since those methods are identical except for the min-hash and bucket min-hash functions. However, the adjacency matrices of the two methods are different in detail. Figure 7 shows the difference between the

submatrices of two matrices. BISO's adjacency matrix makes a clear mosaic pattern while Bi-SO's nonzero entries are randomly scattered over the whole area. This pattern occurs because of the property of the bucket min-hash function, and it helps reduce the number of nonzero blocks.

5 Conclusions

We propose IBSO (Iterative Bucket Shingle Ordering), a novel compression-friendly vertices reordering method for bipartite graphs. We summarize our contributions as follows:

- IBSO gives the best or comparable performance compared to other baselines in terms of the number of nonzero blocks in adjacency matrix.
- IBSO is scalable to large-scale real-world graphs.
- We observe the common aspect of good compression-friendly reordering methods.

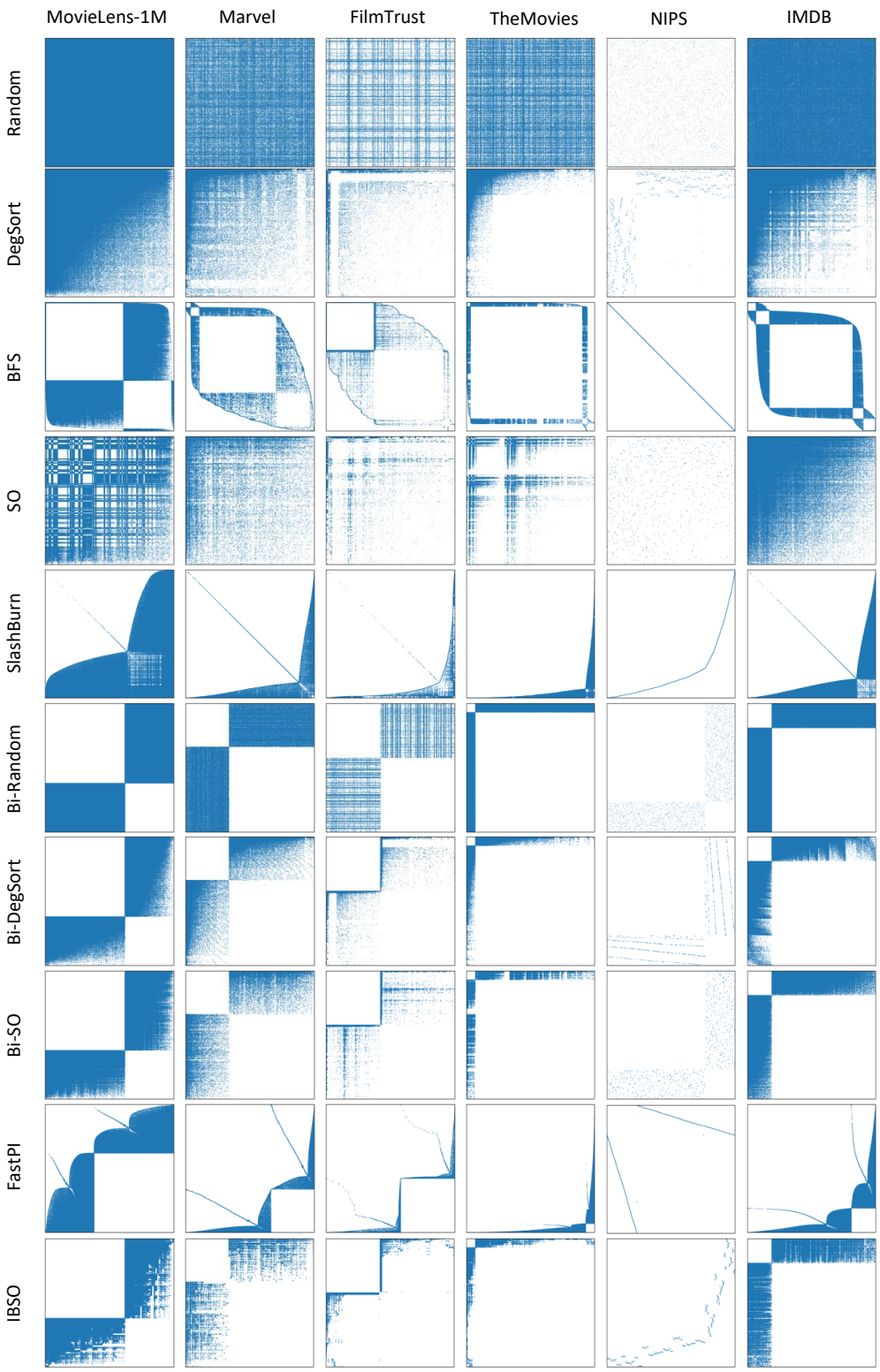


Figure 6: Adjacency matrices of IBSO and baselines on real-world datasets. Bipartite-oriented methods show 2×2 checkerboard pattern because of the separation step.

(a) Bi-SO (b) IBSO

Figure 7: Submatrix of Bi-SO's adjacency matrix (a) and submatrix of IBSO's adjacency matrix (b) on MovieLens-1M dataset. IBSO's adjacency matrix shows a clear mosaic pattern which leads to denser nonzero blocks compared to Bi-SO.

References

[1] Alberto Apostolico and Guido Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[2] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31. IEEE, 2016.

[3] Daniel K. Blandford and Guy E. Blelloch. Index compression through document reordering. In *2002 Data Compression Conference (DCC 2002), 2-4 April, 2002, Snowbird, UT, USA*, pages 342–351. IEEE Computer Society, 2002.

[4] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, page 595602, New York, NY, USA, 2004. Association for Computing Machinery.

[5] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336, 1998.

[6] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, 2009.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107113, jan 2008.

[8] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1535–1544, 2016.

[9] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *SIGMOD Conference*, pages 157–168. ACM, 2012.

[10] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *2009 Ninth IEEE International Conference on Data Mining*, pages 229–238, 2009.

[11] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *SIGMOD Conference*, pages 419–432. ACM, 2008.

[12] Sriram Raghavan and Hector Garcia-Molina. Representing web graphs. In *ICDE*, pages 405–416. IEEE Computer Society, 2003.

[13] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3), sep 2007.

A Appendix

A.1 Labor Division

The team performed the following tasks

[Final Labor Division]

- Problem formulation [Jongjin]
- Data collection and processing [Jaeri]
- Implementation of Random Ordering [Kahyun]
- Implementation of Bipartite Random Ordering [Jongjin]
- Implementation of BFS [Jongjin]
- Implementation of DegSort [Jongjin]
- Implementation of Bipartite DegSort [Jongjin]
- Implementation of BP [Jongjin]
- Implementation of shingle ordering [Jongjin]
- Implementation of Bipartite shingle ordering [Jongjin]
- Implementation of Slashburn [Kahyun]
- Implementation of FastPI [Jaemin]
- Designing IBSO [Jaemin, Jaeri, Jongjin, Kahyun]
- Implementation of IBSO [Jaemin, Jaeri, Jongjin, Kahyun]

[Revised Plan]

- Problem formulation [Jongjin]
- Data collection and processing [Jaeri]
- Implementation of Random Ordering [Kahyun]
- Implementation of BFS [Jongjin]
- Implementation of DegSort [Jongjin]
- Implementation of BP [Jaemin]
- Implementation of shingle ordering [Jongjin]
- Implementation of Rabbit Order [Jaeri]
- Implementation of Slashburn [Kahyun]
- Implementation of FastPI [Kahyun]
- Designing IBSO [Jaemin, Jaeri, Jongjin, Kahyun]
- Implementation of IBSO [Jaemin, Jaeri, Jongjin, Kahyun]

[Old Plan]

- Problem formulation [Jongjin]
- Data collection and processing [Jaeri]
- Implementation of BFS [Kahyun]
- Implementation of BP [Jaemin]
- Implementation of shingle ordering [Jongjin]
- Implementation of Rabbit Order [Jaeri]

- Implementation of Slashburn [Kahyun]
- Designing IBSO [Jaemin, Jaeri, Jongjin, Kahyun]
- Implementation of IBSO [Jaemin, Jaeri, Jongjin, Kahyun]

A.2 Full disclosure wrt dissertations/projects

Jongjin: He is not doing any project or dissertation related to this project: his thesis is on the recommendation system.

Kahyun: She is studying graph neural networks with heterophily and signed graphs. She has never studied graph compression before.

Jaemin: He is not doing any project or dissertation related to this project: his thesis is on stock price prediction.

Jaeri: She is not doing any project or dissertation related to this project: her thesis is on the recommendation system.

B Introduction

The problem we want to solve is the following:

- GIVEN: a bipartite graph $G = (V, E)$
- FIND: a permutation of vertices P
- to MINIMIZE: the number of nonzero blocks of size $k \times k$ in the adjacency matrix

Graph compression through compression-friendly vertices reordering is a common strategy to handle large-scale real-world graphs. Figure 8 shows the importance of the vertices reordering. Figures 8(b) and 8(d) describe the same graph with different permutations of vertices. However, the number of nonzero blocks in Figure 8(d) is 10 while that in Figure 8(b) is 12. Thus, we can store the graph by keeping only 10 submatrixes instead of 12 through vertices reordering, which requires 16.7% reduced storage capacity. Hence, it is important to find an optimal permutation of vertices to efficiently store the matrix.

A lot of real-world data such as user-item purchase data or music-artist data form bipartite graphs. However, the vertices reordering of bipartite graphs differs from the normal graph reordering. An adjacency matrix of a bipartite graph has a fundamental sparsity because of the definition of a bipartite graph. Figure 9 demonstrates the fundamental sparsity of a bipartite graph. Colored entries of Figure 9(b) are fixed to zero since vertices in the same partition cannot be connected. Thus, the density of the block cannot be maximized if the block covers the vertices of both partitions so the number of nonzero blocks increases to contain all edges of the matrix. However, previous vertices reordering methods do not consider this problem of bipartite graphs, which leads to unoptimized compression of the given bipartite graph.

We will propose IBSO, a novel bipartite graph compression method that carefully considers the property of the bipartite graph while reordering vertices.

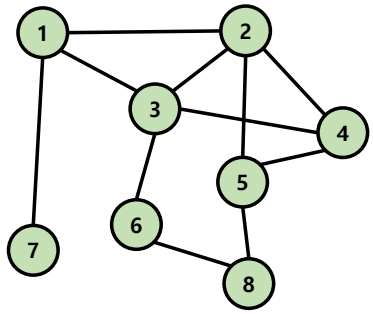
C Proposed Method

We address the following challenges to reduce the number of nonzero blocks in an adjacency matrix:

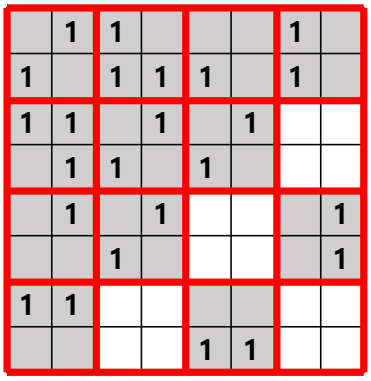
- *How to remove the fundamental sparsity of the bipartite graph?*
- *How to get benefit from the property of bipartite graph?*
- *How to order vertices to form denser blocks in an adjacency matrix?*

The main ideas of IBSO are summarized as follows:

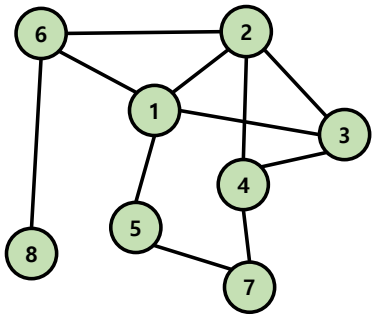
- *Separation.* IBSO separates indices of two partitions to gather fundamentally blank elements into limited areas.
- *Iterative Process.* IBSO iteratively sorts each partition to take advantage of independency between two partitions.
- *Bucket Shingle Ordering.* IBSO sorts vertices based on the bucket min hash function to group vertices in similar blocks.



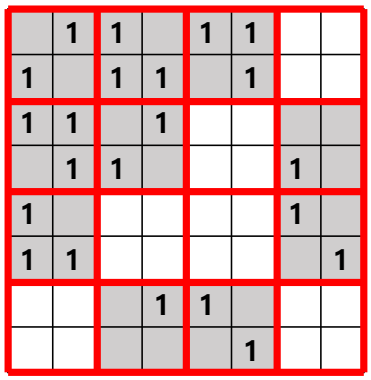
(a) Original graph



(b) Adjacency matrix



(c) Reordered graph



(d) Reordered adjacency matrix

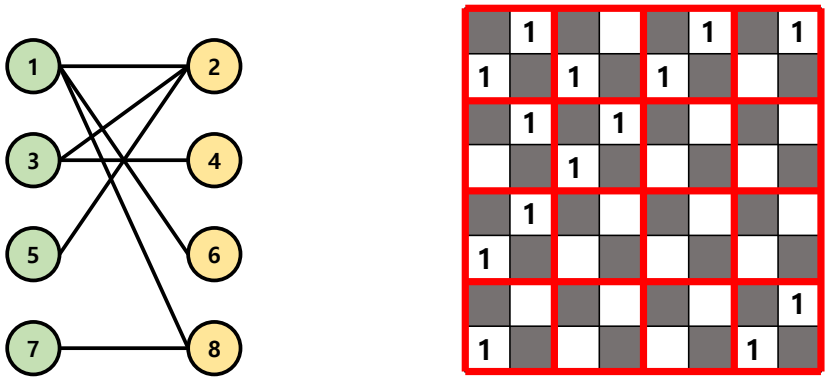
Figure 8: Example of the compression-friendly reordering of the given graph. (b) and (d) are adjacency matrixes of (a) and (c), respectively. They define the same graph structure with different vertices permutations. However, (d) has fewer nonzero blocks than (b), so it requires less storage capacity.

C.1 Separation

To avoid the sparsity of nonzero blocks made from the inner partition edge, IBSO groups the nodes of each partition and separates those in the permutation as shown in Figure 10. Thus, fundamentally blank entries and possible nonzero entries are isolated, which allows nonzero blocks to exclude vacant elements and achieve the best density.

C.2 Iterative Process

We focus on the remaining area of the bipartite graph's adjacency matrix. They are composed of rows representing one partition and columns representing another partition. Thus, we need to find the permutation of each partition to form denser blocks in this area. We call the bottom-left half of this area a *CPS (cross partition submatrix)* in the rest of this paper. Unlike



(a) Bipartite graph (b) Adjacency matrix

Figure 9: Example of (a) a bipartite graph and (b) its adjacency matrix. Colored elements in the adjacency matrix are fixed to zero, which leads to the fundamental sparsity of nonzero blocks.

permutations of rows and columns in a normal graph's adjacency matrix, permutations of rows and columns in CPS are independent. In other words, we may exchange the order rows in CPS without interupt the order of columns.

It is easier to find the optimal permutation of rows under the fixed order of columns. Thus, IBSO alternately freezes the permutation of one partition and finds the optimal permutation of another partition. In this way, IBSO repeatedly solves "easier" problems to find the optimal permutation of vertices in a bipartite graph, which would not be possible in general graphs.

C.3 Bucket Shingle Ordering

Assume that the order of columns in CPS is given. How can we find the order of rows which makes the fewest nonzero blocks? To reduce the number of nonzero blocks, we need to make each nonzero blocks denser. In other words, adjacent rows need to have nonzero elements in similar positions which are part of same blocks.

We propose a *BSO (Bucket Shingle Ordering)* heuristic to find similar rows. We define a bucket min-hash function as follows:

$$f_\sigma(r) \quad = \quad min_{r[i]=1}(\sigma(\left\lfloor \frac{i}{s} \right\rfloor)),$$

where r is a row, σ is a random permutation of blocks, and s is a size of the block. Note that this function indicates the leftmost nonzero chunk when we split the row into chunks of lenth s and reorder chunks in σ order. BSO performs bucket min-hash function on each row several times with different σs to obtain the fingerprint of each row as a tuple of hash values. Then, BSO sorts rows based on their fingerprints. In the same way with proving

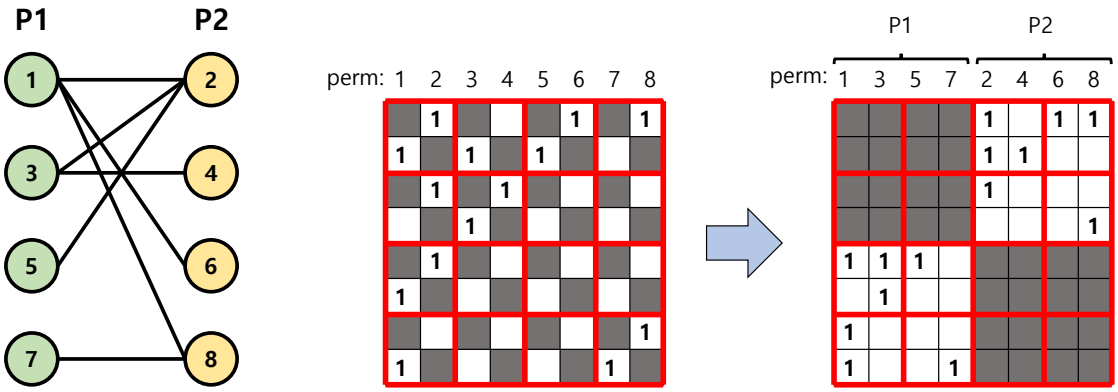


Figure 10: The separation between two partitions on the permutation groups meaningless edges of the bipartite graph's adjacency matrix in top-left and bottom-right corners. In this way, nonzero blocks do not contain any inner partition edge.

relationship between min-hash function and the Jaccard similarity [5], it can be shown that $P(f_\sigma(r_1) = f_\sigma(r_2))$ for the given r_1 and r_2 is a block-wise similarity between two rows. Thus, BSO locates rows that have nonzero entries in similar block closer, which leads to denser and fewer nonzero blocks in adjacency matrix.

C.4 Algorithm

Algorithm 2 shows the full process of IBSO, where BSO is a bucket shingle ordering function which takes a bipartite graph, two partitions, and a permutation of one partition as inputs and returns a permutation of another partition as an output. Since that time complexity of the bucket min hash function is linear to the number of nonzero elements in the row, each BSO function requires $O(|E|)$ time to get hash values and $O(|V|\log|V|)$ time to sort vertices. Thus, the total time complexity of IBSO is $O(|E| + |V|\log|V|)$.

Algorithm 2 IBSO (Iterative Bucket Shingle Ordering)

Require: Bipartite graph G, Partitions of vertices V_1, V_2

initialize the permutations of partitions P_1, P_2 //Separation
for i in range($iter$) **do** //Iterative Process
    $P_1 \leftarrow \text{BSO}(G, V_1, V_2, P_1)$ //Bucket Shingle Ordering
    $P_2 \leftarrow \text{BSO}(G, V_2, V_1, P_2)$
end for
$rtn \leftarrow \text{concat}(P_1, P_2)$
return rtn