Edge Labeling Schemes for Graph Data

Oshini Goonetilleke
RMIT University
Melbourne, Australia
oshini.goonetilleke@rmit.edu.au

Danai Koutra
University of Michigan
Ann Arbor, USA
dkoutra@umich.edu

Timos Sellis
Swinburne University of Technology
Melbourne, Australia
tsellis@swin.edu.au

Kewen Liao
Swinburne University of Technology
Melbourne, Australia
kliao@swin.edu.au

ABSTRACT

Given a directed graph, how should we label both its outgoing and incoming edges to achieve better disk locality and support neighborhood-related edge queries? In this paper, we answer this question with edge labeling schemes GrdRandom and FlipInOut, to label edges with integers based on the premise that edges should be assigned integer identifiers exploiting their consecutiveness to a maximum degree.

We provide extensive experimental analysis on real-world graphs, and compare our proposed schemes with other labeling methods based on assigning edge IDs in the order of insertion or even randomly, as traditionally done. We show that our methods are efficient and result in significantly improved query I/O performance, including with indexes built on directed attributed edges. This ultimately leads to faster execution of neighborhood-related edge queries.

CCS CONCEPTS

•**Information systems →Data management systems;** *Data structures; Information storage systems;* Database management system engines;

KEYWORDS

Edge Labeling, Graph databases, Graph storage management, Query Processing

ACM Reference format:
Oshini Goonetilleke, Danai Koutra, Timos Sellis, and Kewen Liao. 2017. Edge Labeling Schemes for Graph Data. In *Proceedings of SSDBM '17, Chicago, IL, USA, June 27-29, 2017,* 12 pages.
DOI: http://dx.doi.org/10.1145/3085504.3085516

1 INTRODUCTION

Graph analysis has attracted a lot of interest in the research community. Some of the reasons are (i) the availability and heterogeneity of data that naturally support graph data structures, such as social interactions and browsing activity data, and (ii) the emerged tools and techniques for the convenience of managing and analyzing such data. By leveraging more advanced tools, researchers and analysts have been able to gain more insights by asking new questions (queries) about their graph data. At the same time, they should expect to have their queries answered as efficiently as possible. Given that the throughput of many graph queries can be significantly affected by disk performance, graph analysis methods need to focus on effective graph storage and indexing for optimizing disk operations.

A graph management system typically assigns internal identifiers (IDs) to vertices and edges at insertion time to allow their fast reference and indexing. In many systems, IDs are simply assigned based on the order of insertion, which is dependent on the data source: a web graph could be labeled in the lexicographic order of web pages, and a social network in the order in which users are crawled. A graph is generally represented as an adjacency list or matrix. Other systems such as Sparksee [18] and SNAP [15] also maintain a graph as list of edges, especially useful for indexing edges with rich attributes and managing multigraphs. Representing edges of a graph is therefore an important aspect and devising optimal representations has an impact on the performance of such systems; moreover, as we show later in this paper, such representations may play an important role in other applications, such as graph streaming data.

Our motivation of this work stems from optimizing such systems, in particular for improving the efficiency of answering edge queries. For instance, given a node of a graph, a query could ask for its k-hop neighboring attributed edges (both incoming and outgoing) which possess a value of a particular edge attribute like the timestamp. In a friendship social network, such a query could be finding a person's friendship details (with both followers and followees) established at the date of 01/01/2017. Other examples of neighborhood queries are finding mutual ties in a co-authorship network and recommendations in a product network.

Figure 1 displays the retrieval of node-neighbourhood edge properties via edge indexes (left of the figure), while the underlying stored data file (right of the figure) is sorted by labeled edge IDs. The incident edges for a given node are indexed and contain pointers to the actual location of the edge records on disk. Each edge record consists of the assigned edge ID, along with a number of property value pairs that describes the edge. We argue that having consecutive edge IDs (over all node neighbourhoods) ensures edges are located in closer pages on disk, thus leading to better I/O performance for answering neighbourhood queries.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SSDBM '17, Chicago, IL, USA
© 2017 ACM. 978-1-4503-5282-6/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3085504.3085516

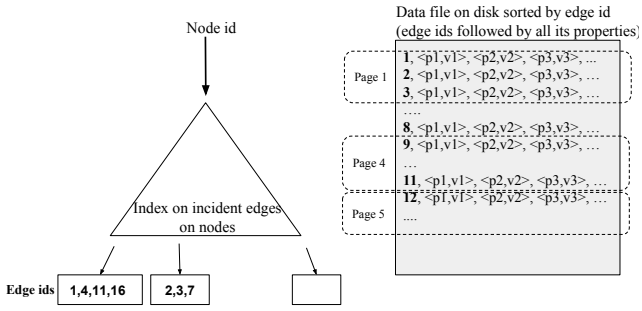


Figure 1: An example of indexing attributed edges.

In this work, our goal is to optimally assign edge labels coupled with edge indices to achieve improved disk locality for efficiently answering these typical graph queries, *without modification* to the storage internals of the graph system at hand. While node labeling has been widely studied, one should not overlook the related edge-labeling problems aiming at improving query performance of some current and future graph analysis systems that store edge lists for different reasons. We focus on edge-labeling schemes for *directed* graphs, and demonstrate that even simple labeling of edges alone can significantly improve the performance of some typical workloads of graph applications, by lowering the number of disk reads. Retrieving neighborhood of a node is at the heart of many operations conducted on graph data. We posit that better disk locality of outgoing and incoming edges incident to a particular vertex would result in significant speedup of the neighborhood query, and consequently, in better execution times for a vast majority of queries which are based on it.

We approach this problem as labeling (*encoding*, *ordering* or *numbering*) of edges, where each edge in a set of edges $\mathcal{E}$ is given a number (an edge identifier, ID or eid) between 1 and $|\mathcal{E}|$. The encoding is performed so that outgoing and incoming edges of nodes are given eids as consecutively as possible, thereby putting outgoing and incoming neighboring edges close together on disk based on these eids. While many graph database systems (Neo4j, Sparksee, etc.) take approaches to define memory hierarchies for efficient query processing, we investigate how graph data storage can be improved at physical level. These modern graph database systems are relatively new and require further investigation on topics such as physical data management to progress towards the level of maturity of relational database systems.

Figure 2 illustrates different edge-labeling strategies for a directed multigraph. Random ordering as shown in Figure 2a has not much consecutive guarantee on the eids assigned to edges. If edges are ordered by source nodes (a directed edge points from a source node to a target node), as in Figure 2b, all outgoing edges from a source node are guaranteed to be assigned consecutive IDs. However an obvious drawback of this approach is that incoming edges of target nodes are overlooked, resulting in these edges possibly scattered across a range of IDs, undesirable for the underlying physical data storage. On the other hand, Figure 2c shows a perfect edge numbering such that both incoming and outgoing edges are given consecutive numbers. While for real graphs it is often impossible to perform such a perfect labeling, we can attempt to maximize the overall consecutiveness.

In this work, we formulate edge-labeling as an optimization problem, and present two scalable approaches, GrdRandom and FlipInOut, to label edges in a way that maximizes the total edge consecutiveness of graph, i.e., maximize the number of sequentially labeled edges to enable sequential storage, thereby increasing the locality of disk accesses. GrdRandom is based on the idea that numbering should be alternated between incoming and outgoing neighbors to strike a balance between the edge directions. FlipInOut extends this idea by taking into account the neighborhood information, and prioritizing high-degree nodes. Our contributions in this work can be summarized as follows:

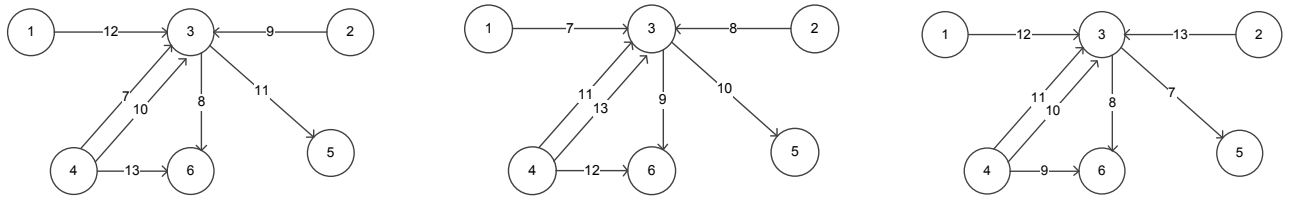
- **Formulation:** We propose an edge consecutiveness metric on directed graphs (that takes into account both outgoing and incoming edges) and formulate edge-labeling as a maximization problem of this metric.
- **Methods:** We introduce GrdRandom and FlipInOut as two edge labeling algorithms that focus on the balance between numbering outgoing and incoming edges.
- **Experiments:** We conduct extensive experiments on real graphs, and show significant benefits of our approaches over baselines in disk I/Os and query times.
- **Applications:** We demonstrate a case study of our methods to be applied in streaming graph partitioning.

We conduct experiments to evaluate disk I/O performance, and the subsequent speedup of various graph operations (e.g., friend-of-friend queries and shortest paths). Among the systems that index edges, we use Sparksee as a representative graph analysis system. Other systems, such as Unicorn [7] can also take advantage of edge-labeling. Unicorn has several types of edges (i.e., relationships among users, posts etc. in Facebook), and any index built on these edges based on edge properties can leverage a good labeling scheme to achieve disk locality and efficient edge indexing.

2 RELATED WORK

In this section we first discuss several categories of existing work that are closely related to the edge-labeling problem, and then introduce an implementation, Sparksee, which stores its edges as bitmaps on disk.

Node arrangement. The most relevant body of work is node reordering methods, which are introduced with different goals in mind. SlashBurn [16], for example, is a recent approach for renumbering the nodes so that the non-zero elements of the adjacency matrix are grouped together, thus enabling faster execution of matrix operations and better compression. SlashBurn investigates the 'no good cut' problem [14] for power law graphs and propose techniques for node reordering. Shingle Ordering [6] groups similar nodes to form dense communities by exploiting the link reciprocity of social networks. It focuses on solving MLOGA and MLOGGAPA minimization problems, whereas we focus on solving a maximization problem. Recently, Wei et al. [28] exploits node ordering for improving CPU cache performance. Although node reordering or relabeling can result in improved performance



(a) Random order: $C_{in}(G) = 0.4, C_{out}(G) = 0.4$. **(b) Source-based order:** $C_{in}(G) = 0.4, C_{out}(G) = 1.0$. **(c) Perfect order:** $C_{in}(G) = 1.0, C_{out}(G) = 1.0$.

Figure 2: Illustration of different ordering strategies.

in many node-oriented queries, it does not necessarily guarantee good performance for edge-oriented ones.

Graph compression. Existing work exploits the property of locality in graphs with graph compression as a primary objective. Early approaches have focused on compressing web graphs with similarity and locality features [24], lexicographic localities [4]—later extended to social networks [6, 8]—, or a BFS approach [2]. It must be noted that compression is not our main focus, although a benefit in compression may be a side-effect of our proposed encoding schemes.

Space filling curves. Hilbert and other curves [19] are also closely related if we view the edge encoding problem as a mapping of edges to IDs. Hilbert curves generate a mapping between a 1-dimensional and a 2-dimensional space known to achieve good locality of reference. Different types of space filling curves are widely used to index spatial objects based on proximity. Intuitively, Hilbert curves recursively partition an (x,y) coordinate in a 2-dimensional space such that it can be mapped to a single integer.

In our case, for every edge we can calculate a Hilbert index using the combination of the adjacent endpoints. We can then use this index to label the edges. The Hilbert index of an edge is sensitive to the neighboring node labels. In the original use of the space filling curve, the two endpoints refer to an actual (x,y) coordinate of a point in space. But in the graph space unless the x,y node IDs are 'close' (distance-wise in the graph space), we cannot guarantee that the edges will be assigned consecutive, or even close numbers.

Graph partitioning, Community detection and Clustering. The well-studied problem of graph partitioning is also pertinent to our work. The objective of partitioning algorithms is to reduce the number of edges crossing partitions, i.e., *edge cuts*, so that the nodes belonging to the same partition can be grouped together as a coherent unit of storage. A multitude of partitioning algorithms have been developed over the years in response to variations of the classic partitioning. METIS [12], one of the most widely used in practice, belongs to the category of multi-level partitioning strategies [9, 13]; many distributed algorithms [23, 27] work well with large graphs; DIDIC [10] and EvoCut [1] require only local computations eliminating expensive global operations on the graph. Community detection and clustering algorithms [3, 5, 20, 22] have a very similar objective of grouping densely connected regions of a graph (e.g. cliques and bipartite cores) which are loosely connected with the rest of the graph.

All of these algorithms achieve some degree of locality within the graph by considering homogeneous regions in the network. One may be tempted to leverage a partitioning or clustering approach to derive an edge numbering. For example, a method such as METIS (known to have good edge-cuts) can be used to partition the graph, and then guide the arrangement of the edges on disk. For instance, the inter-edges can be assigned to the partition of the source or destination node, decided at random, while the intra-edges can be numbered in some arbitrary sequence. However, once a node numbering is known it is not straightforward to define a method that labels the edges in a way that their *consecutiveness* is maximized. Moreover, techniques like SlashBurn and METIS operate on *undirected* graphs, so only the existence of the edge is sufficient to obtain the final numbering. Naturally edge directionality is ignored when placing a node in a partition, cluster or community, while for us directionality is of utmost importance.

Sparksee. Sparksee is one of the graph database management systems that store edges for query processing, this is also the reason we use it as our testbed in experiments. In Sparksee, vertices, edges, attributes are stored internally as a combination of compact bitmaps enabling efficient bit operations for query processing. The motivation behind using a bitmap representation in Sparksee is two-fold: First, bitmaps are able to hold large amounts of information in reduced amount of memory. Second, graph queries can be converted to a series of logic (bit) operations that can be performed very efficiently.

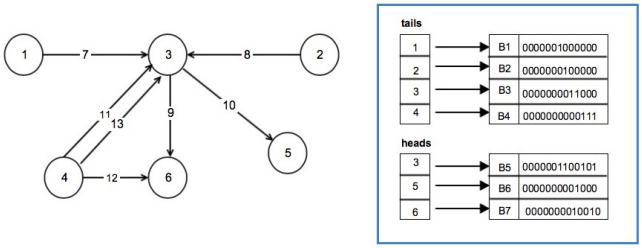


Figure 3: Bitmaps representing relationships for a graph with edges sorted by the source

The structure of the bitmaps depend on the IDs (labels) that are assigned to the edges. Figure 3 shows the corresponding bitmaps (LSB on left) for storing relationships on a simple graph with edges sorted by the source. The `tail`/`head` group shows the `IDs` of all edges of which each node is a tail/head. Let us consider node `ID`

3 as an example: Node ID 3 is the tail for edges with IDs 9 and 10. Therefore, in the bitmap for 3 in the tails group (B3), the 9th and the 10th bits are set to one. Similarly, node ID 3 is the head for edges with IDs 7,8,11 and 13. Hence, its bitmap, B5, has the bits 7,8,11 and 13 set to one. Notice that a different edge-labeling will result in a different bitmap in the relationships.

As illustrated in Figure 3, when the graph edges are sorted by source, 1s in the tail group will be grouped together however, the 1s in the head group are scattered across the bitmap requiring more pages to represent the bitmap on disk. Consequently, a query that involves retrieving the incoming neighbors of node ID 3 will incur more disk I/O operations. Maximizing consecutiveness of edge IDs is crucial to performance, since achieving consecutive 1s at a maximum will allow compression of bitmaps and therefore faster bit-level operations as well as less storage.

3 EDGE-LABELING SCHEMES

Answering a query on a directed graph may either involve traversing through a node's outgoing edges, incoming edges, or a combination of both. A neighborhood query is the basis of most graph queries, if not all, used in practice. Consider a graph-based recommendation query on a who-follows-whom network on Twitter. Recommending users to follow to a user u may involve finding the 2-step followees (2-hop outgoing neighbors) who u is not following (1-hop outgoing neighbors). Moreover, in label propagation, at any round, both the incoming and outgoing edges of a node need to be accessed as messages are exchanged between neighbors. As such, our focus is primarily on improving the locality of the neighborhood query. Although an optimal arrangement of the edges is dependent on the characteristics of the graph and the type of query, retrieving the 1-hop neighborhood is fundamental to (almost) all graph operations.

Accessing neighboring edges of a node on disk requires reading pages from disk. If the neighboring edges are placed closer together on disk, this will reduce the costly random reads required for the fundamental neighborhood query. As such our objective is to place neighboring edges with improved locality independent of the type of graph. In Sparksee and other systems, this translates to assigning numbers to both outgoing and incoming edges as consecutively as possible. Since there are many ways to number the edges in a graph an exhaustive search is not feasible; we propose a 'balanced labeling' technique that alternately numbers edges of opposite types (incoming and outgoing). As we confirm in the experiments, how well we can achieve consecutiveness depends on graph characteristics and the type of query workload. In the following, we first present the edge-labeling problem and then our proposed methods. We provide descriptions of some graph related notations in Table 1.

Table 1: Some graph related notations used in algorithms.

Symbol	Description
$\|\mathcal{V}\|$	number of vertices
$\|\mathcal{E}\|$	number of edges
$\mathcal{E}'$	reordered set of edges in a graph
T	edge type: 'in' or 'out'
T'	an inverse edge type
$\mathcal{L}$	set of unlabeled edges in a graph
$N_T(v)$	neighbors of vertex v of type T
$deg_T(v)$	degree of vertex v of edge type T
$\mathcal{E}_v$	set of unlabeled edges incident to v
(v, x)	an outgoing edge of v, or incoming edge of x
visited $[T]$	set of visited vertices of type T

3.1 Problem formulation

Let $G(\mathcal{V}, \mathcal{E})$ be a directed (multi-)graph with a set of vertices, $\mathcal{V}$, and a set of edges, $\mathcal{E}$. Internally, each vertex $v \in \mathcal{V}$ and edge $e \in \mathcal{E}$ is identified/labeled by a unique integer ID. We consider G stored as edge lists (for reasons outlined in Section 1). First, we define the edge consecutiveness metric for any vertex as below:

Definition 3.1 (Edge Consecutiveness). Given a directed graph $G = (\mathcal{V}, \mathcal{E})$, and a mapping $\pi : \mathcal{E} \rightarrow \mathbb{Z}$ of edges to integer eids, the incoming edge consecutiveness of a vertex v, $C_{in}(v)$, is defined as the number of pairs of its incoming edges with consecutive eids under the ordering π. Also, if $deg_{in}(v) = 1$, then $C_{in}(v) = 1$. The outgoing edge consecutiveness $C_{out}(v)$ can be defined similarly.

Note that the maximum value of $C_{in}(v)$ is $deg_{in}(v) - 1$ (similarly for $C_{out}(v)$) except for the cases $deg_{in}(v) = 1$ and 0. The edge-labeling problem we consider is then formulated as follows:

Problem 1. [edge-labeling] Given a directed graph $G = (\mathcal{V}, \mathcal{E})$, we want to find the best labeling π, such that the 'total' in- and out-consecutiveness of the graph, $C(G)$ as below, is maximized:

$$\mathrm{C(G)} = \underbrace{\frac{1}{|\mathcal{E}| - n_{in}} \sum_{v \in \mathcal{V}} C_{in}(v)}_{C_{in}(G)} + \underbrace{\frac{1}{|\mathcal{E}| - n_{out}} \sum_{v \in \mathcal{V}} C_{out}(v)}_{C_{out}(G)} \quad (1)$$

where n_{in} and n_{out} are the number of nodes with > 1 incident edges of their respective edge types.

The scaling factors in the above consecutiveness formulation is to ensure that $C(G) \in [0, 2]$, since the maximum value of the in-consecutiveness (out-consecutiveness) of G is $|\mathcal{E}| - n_{in}$ ($|\mathcal{E}| - n_{out}$). For example, the graph in Figure 2b has in-consecutiveness—$C_{in}(G)$=0.4, since $|\mathcal{E}| - n_{in} = 7 - 2 = 5$ due to node 3 and 6, and $C_{in}(1) = C_{in}(2) = C_{in}(4) = C_{in}(6) = 0$, and $C_{in}(3) = C_{in}(5) = 1$. Similarly, its out-consecutiveness is $C_{out}(G) = 1$, so the total consecutiveness is 1.4.

The theoretical maximum for perfect in-/out- consecutiveness is 1.0 which means that all the edges of the graph are consecutive. For a given graph G, the out- (or in-) consecutiveness can be easily made 1.0 by labeling all the outgoing (or incoming) edges of each vertex i consecutively. However, it is unlikely that even in the optimal case, *both* incoming and outgoing edges can be made perfectly consecutive, because in directed graph labeling one type of edges also labels the edges of the inverse type. An intuitive attempt to maximizing $C(G)$ is to locally maximize $C_{out}(v)$ and $C_{in}(v)$ for each node $v \in \mathcal{V}$ by 'taking turns' in labeling edges of opposite directions. This is the core idea of our proposed algorithms to be presented next.

3.2 Labeling schemes

In this section we first describe the baseline methods for labeling the edges of a graph, and then give the details of our proposed methods. Our solution seeks to maximize the consecutiveness at

each individual vertex $C(v)$ so that local decisions greedily make progress towards the global optimal.

3.2.1 **Baselines for labeling.** There are three natural ways that the edges of a graph can be ordered in the input file, independent of the type of graph. We use the following methods to form our baselines:

- **Random**. The edges are listed in an order given by a random permutation.
- **consecIN**. The incoming edges of each node are labeled consecutively, edge IDs are sorted over the edge target/destination nodes.
- **consecOUT**. The outgoing edges of each node are labeled consecutively, edge IDs are sorted over the edge source nodes.

As explained in Section 2, existing node reordering methods are not directly applicable in solving our graph consecutiveness maximization problem with edge-labeling. However, correlations may exist between node and edge ordering methods for serving different types of graph queries. We leave such correlation studies for future work and instead focus on solving the edge-labeling problem here.

3.2.2 **Proposed Method: GrdRandom.** The two baseline methods consecIN and consecOUT are biased towards labeling the respective edge type consecutively. We propose an intuitive algorithm, GrdRandom, in which the labeling does not favor a single edge type. In this greedy approach we first consider a random permutation of the nodes to inform the visit order. For each of the nodes we flip a coin to decide if the outgoing or the incoming edges of that node should be labeled consecutively. Once an edge is given a number, it is not changed. The idea is that we alternate between the type of edge we number so that we do not favor a single edge type.

Algorithm 1 shows the general idea of the GrdRandom algorithm. The algorithm randomly numbers the edges and can complete fairly quickly. In the case that not all edges are labeled (due to randomness), a restart procedure can always be performed on labeling unlabeled edges. The algorithm takes O($|\mathcal{V}|$) time to run the random permutation, and performs the number assignment in O($|\mathcal{E}|$). Therefore the complexity of the algorithm is $O(|\mathcal{V}| + |\mathcal{E}|)$.

Algorithm 1 GrdRandom

Input: Graph $G = (\mathcal{V}, \mathcal{E})$
Output: Re-labeled edge list $\mathcal{E}'$

```
1: E' ← {}                                          ▷ labeled edge list to return
2: L ← E                                            ▷ list of unlabeled edges
3: V^R ← random_permutation(V)
4: for v ∈ V^R do
5:     if rand() > 0.5 then T ← 'out' else 'in' end if
6:     /* E_v: unlabeled edges incident to v */
7:     if T == 'out' then                           ▷ unlabeled edges
8:         E_v ← {(v, x) ∈ E} ∩ L                   ▷ outgoing
9:     else
10:        E_v ← {(x, v) ∈ E} ∩ L                   ▷ incoming
11:    end if
12:    for e ∈ E_v do
13:        E' ← E' ∪ {e}                            ▷ edge added in order
14:        L ← L \ {e}
15:    end for
16:    exit if |L| == 0
17: end for
```

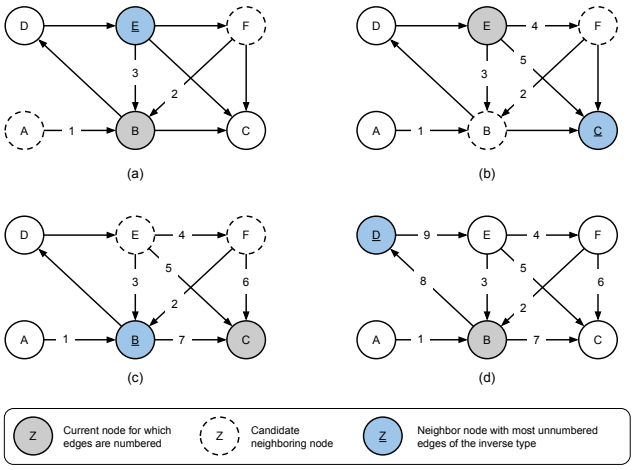


Figure 4: FlipInOut Algorithm: Example.

3.2.3 **Proposed Method: FlipInOut.** Our first approach, GrdRandom, is simple and easy-to-implement, and, as we show in our experiments, it outperforms the baselines. In our proposed method, FlipInOut, we further advance GrdRandom's main idea, and carefully incorporate more features to consciously improve the edge consecutiveness. We first give a simplified example of our proposed algorithm in Figure 4 and then discuss its main features in detail.

Illustrative Example. As shown in Figure 4 (a), the algorithm starts with the vertex of the highest total degree (node B) and numbers the edge type that has most unlabeled edges. From the in-neighbors of B (*candidate* vertices: A, E and F), it then picks the in-neighbor with the most unlabeled edges of the inverse (flipped) edge type (i.e., `edgeType = out`). For each node the algorithm keeps track of the number of unlabeled out- and in-neighbors. Node E is the selected vertex for the next iteration as neighbor A has no more out-edges, while E has 2 and F has 1. E's remaining outgoing edges are labeled in sequence (4(b)). Our method continues by considering the neighbors of E and selecting the vertex with the highest unlabeled edges of `edgeType = in` etc. (Figure 4 (c)-(d)). If all neighboring nodes have their edges labeled, the algorithm restarts with the node with the highest remaining total degree.

As more edges are labeled there are many node instances with all their neighbors labeled which results in the algorithm to restart often. Our proposed algorithm, FlipInOut (Algorithm 2), incorporates the following three main ideas with the goal of labeling edges in both directions as consecutively as possible:

I1. Alternate. Similar to GrdRandom, at each iteration, the algorithm flips between labeling outgoing and incoming edges (hence the name Flip-In-Out) to balance the consecutiveness. The node visit order of FlipInOut is based on the number of unlabeled edges of the flipped type, while GrdRandom is neighborhood agnostic. For example, if the incoming edges of a node was the last to be labeled, FlipInOut will examine the *outgoing* edges of the current

node's neighbors, and vice versa. A swap procedure explained later (after I3) ensures the continuity in labeling.

I2. Prioritize. High-degree nodes are given high priority, and are labeled earlier in the algorithm. As in the example, when presented with a choice, FlipInOut numbers the edges of the highest-degree neighboring node. The intuition is that locality is especially important for high-degree nodes; If the edges of high in- and out-degree nodes are assigned consecutive numbers, a larger proportion of edges will be consecutive. As a result, it is more important that edges incident to "popular" nodes are closer together on disk, compared to a node with only a couple of incident edges. Any query that involves accessing the neighborhood at a depth greater than one (e.g. shortest paths) is more likely to reach a high-degree node due to its large number of connections. If the neighbors of these high-degree nodes are not close on the disk, a query can quickly become very inefficient. We therefore seek to minimize the overall I/O cost for accessing the graph by minimizing the I/O activity of the high-degree nodes.

I3. Terminate Early. This idea is applicable and particularly important for large graphs, where it is common to have frequent vertex restarts after a significant portion of the edges are labeled. The idea behind early termination is to differently order the last $\delta\%$ of the edges. Specifically, we employ a neighborhood-agnostic approach, which decides the visit order of the vertices with unlabeled incident edges and terminates the 'flipping edge' idea. Each node v is represented as a set of at most two pairs: (i) $(v, deg_{in}(v))$, if it has unlabeled incoming edges; and (ii) $(v, deg_{out}(v))$, if it has unlabeled outgoing edges. The resulting pairs are ordered in decreasing order of degree (in or out) to inform the order in which the vertices will be visited. Their incident edges of the corresponding type are then labeled consecutively. In our experiments, $\delta = 12\%$ achieved good performance in the largest graphs that we used and eliminated the frequent restart problem. The percentage δ is set once for all the remaining edges. For brevity, we have excluded the early termination process from Algorithm 2 (the criterion on line 5 would change to $|\mathcal{L}| > \delta|\mathcal{E}|$).

The **alternate** and **prioritize** steps are employed to locally maximize the individual consecutiveness $C(v)$ (Problem 1) for each vertex v. When this greedy approach terminates, the algorithm is making progress towards an optimal solution. As we mentioned in the **alternate** step, we employ a **swap procedure** to ensure continuity. As described in the example, at any given step, out of the candidate neighboring nodes, the next vertex to visit is selected based on the number of unlabeled edges a vertex has of the flipped edge type. When the vertex is chosen, the selected vertex inherits at least one edge from the current node. Before labeling the edges of the selected vertex a condition is tested: If the edge connecting the current vertex and the selected vertex (i.e., the common edge) does not have the highest edge number seen so far, the edge number is swapped with the maximum sibling edge ID. This ensures the continuity in numbers assigned to the inherited edge and the edges of the selected vertex that are about to be labeled. This is another strategy to ensure that the local consecutiveness of a given node is at the maximum.

Algorithm 2 FlipInOut

Input: Graph $G = (\mathcal{V}, \mathcal{E})$
Output: Re-labeled edge list $\mathcal{E}'$

1: $\mathcal{E}' \leftarrow \{\}$ ▷ re-labeled edge list to return
2: $\mathcal{L} \leftarrow \mathcal{E}$ ▷ list of unlabeled edges
3: visited[out] $\leftarrow \{\}$ and visited[in] $\leftarrow \{\}$
4: $[v, T]$ = ChooseVertex($\mathcal{L}$, visited) ▷ starting vertex
5: **while** $|\mathcal{L}| \neq \emptyset$ **do** ▷ There are still unlabeled edges
6: /* $\mathcal{E}_v$: unlabeled edges incident to v */
7: **if** T == 'out' **then**
8: $\mathcal{E}_v \leftarrow \{(v, x) \in \mathcal{E}\} \cap \mathcal{L}$ ▷ outgoing
9: **else**
10: $\mathcal{E}_v \leftarrow \{(x, v) \in \mathcal{E}\} \cap \mathcal{L}$ ▷ incoming
11: **end if**
12: **for** (x, y) in $\mathcal{E}_v$ **do**
13: $\mathcal{L} \leftarrow \mathcal{L} \setminus \{(x, y)\}$
14: $\mathcal{E}' \leftarrow \mathcal{E}' \cup \{(x, y)\}$ ▷ edge added in order
15: visited[T] $\leftarrow$ visited[T] $\cup \{v\}$
16: $deg_T(x)$−=1; $deg_T(y)$−=1 ▷ current degree
17: **end for**
18: T $\leftarrow$ T′ ▷ flip the edge type ('in' or 'out')
19: /* $N_T(v)$: neighbors of node v of type T*/
20: **if** $|\{(x, y) \in \mathcal{E} | x \in N_T(v)\} \cap \mathcal{L}| \neq \emptyset$ **then**
21: /* Find the next vertex to visit, from v's neighbors */
22: $v = \text{argmax}_{x \in N_T(v) \setminus \text{visited[T]}} \{deg_T(x)\}$
23: **else**
24: $[v, T]$ = ChooseVertex($\mathcal{L}$, visited)
25: **end if**
26: **end while**
27: /* Choose the new starting vertex */
28: **function** ChooseVertex($\mathcal{L}$, visited)
29: $v = \text{argmax}_{v_i \in \mathcal{V} \setminus \{\text{visited[T]} \cap \text{visited[T']}\}} \{deg_T(v_i)\}$
30: **if** $deg_{out}(v) > deg_{in}(v)$ **then** T $\leftarrow$ 'out' **else** 'in'

Runtime Complexity. Line 4 in Algorithm 2 spends time $O(|V|)$. Per node v, we execute lines 12-17 in $O(deg(v))$, and either line 22 or 24, which are $O(deg(v))$ and $O(|V|)$, respectively. So, FlipInOut is $O(|V| + |E| + max\{|E|, |V|^2\})$. Its worst case complexity, $O(|V|^2)$, occurs only when it keeps restarting (line 24—i.e., the graph consists of disconnected stars). In practice, restarts happen only towards the end of the algorithm, and FlipInOut is very efficient needing only 2.8 and 4.9 minutes to label 33M and 69M edges, respectively.

4 EXPERIMENTAL EVALUATION

We conducted experiments to demonstrate the performance of our encoding methods on a variety of real graphs. In the following subsections, we answer the questions:

- Can we speed up popular graph queries using FlipInOut edge-labeling?
- Can we observe improved disk I/O as a result of better locality when storing the graph on disk?
- Do GrdRandom and FlipInOut show benefit in storage compared to a random ordering and other baseline schemes?

4.1 Experimental Setup

Environment. The experiments were conducted on a Linux machine with 4.00GHz Intel Core i7-4790K, 8GB of memory and 60GB SSD.

Datasets. We conducted experiments on six directed real-world graphs with edges ranging from 100,000 to 69 million, which we

obtained from SNAP[1] and KONECT[2]. In Table 2 we summarize basic statistics of each graph considered, such as their sizes and clustering coefficients.

Setup. Our experiments used Sparksee 5.2 (cf. Section 2) for the creation of databases. Given a dataset, node labeling is identical for all labeling methods, but the edge-labeling differs.

For every graph, a Sparksee database is created for each of the labeling methods including the baselines—Random, consecIN, consecOUT, GrdRandom and FlipInOut. The timing reported in experiments is only the query execution time (i.e., the database creation time is excluded). In order to run queries on edge properties, we augment each of the datasets by adding randomly generated integer attributes on all the edges, representing weight or timestamp property.

4.2 Speedup of Queries

We perform experiments to demonstrate the speedup of some popular graph queries: (i) friend-of-friend queries, which explore a node's 2-hop neighborhood (ii) shortest path queries, and (iii) queries that retrieve edge properties. The performance for all queries is shown in Figure 5. Each plot shows the average execution time (y-axis) of running the query for 100 instances (node IDs). For a dataset, the same instances (node IDs for FoF, property queries and ID pairs for shortest paths) are used for all queries and labeling methods.

4.2.1 **Friend-of-Friend (FoF) Queries.** We perform two types of friend-of-friend queries, which explore neighborhoods at depth 2, to inspect both directions: FoF-in and FoF-out. These two queries are chosen to show the behaviour of different schemes when the query is biased. For high-degree nodes these queries involve having to traverse through a large neighborhood. The FoF query performance for different labeling schemes is shown in Figure 5a-b.

We observe that for a FoF-out query (Figure 5a), consecOUT numbering has the lowest execution time reported across all graphs. This is expected as consecOUT is the optimal arrangement of edges for an FoF-out query with all outgoing edges having consecutive numbers ($C_{out}(G) = 1.0$). However, the same query performed on a database with consecIN edge encoding ($C_{in}(G) \approx 0$), has performance close to that of a random ordering.

Similarly, a consecIN numbering performs best for an FoF-in query (Figure 5b), but performs similar to a random ordering when running an FoF-out query. To put this in context, for the Flickr graph, while consecOUT is 8× faster compared to consecIN for FoF-out queries, it becomes 7× slower for FoF-in queries. Thus, consecIN and consecOUT are biased towards a single direction (in/out, resp.) and are only suitable for the query type on which the database is built.

If the query workloads are known apriori to be only of one type, certainly these approaches work very well. But in practice this is a strong assumption and rarely the case. Therefore, we need methods to strike a balance between achieving locality of both incoming and outgoing edges. Across a variety of query workloads, our approaches meet halfway between the biased labeling and stay consistent with the best- performing methods irrespective of query type.

Observation 1. FlipInOut is closer to the best performing consecOUT for FoF-out query (Figure 5a) and closer to the best performing consecIN for FoF-in queries (Figure 5b). For FlipInOut, the average relative performance improvement for FoF-out (FoF-in) queries ranges from 36% to 76% (38% to 66%, resp.) compared to random numbering.

Although GrdRandom does not perform as well as FlipInOut, its execution is still consistent across the query types. It outperforms the random and consecIN schemes for out-specific queries, and the random and consecOUT schemes for in-specific queries. Recall that GrdRandom was a fairly straightforward and easy-to-implement approach which gives acceptable results. In Section 4.3 we confirm that the main reason behind the better timing in our methods is improved disk I/O operations.

4.2.2 **Shortest Path Queries.** We chose shortest path queries to represent the category of queries that employ both outgoing and incoming edges. The shortest path from the source vertex s to the target vertex t involves a bidirectional breadth-first strategy, which leads to significant speedup in the algorithm by reducing the number of visited vertices. The idea is to perform a forward search from s via its outgoing edges, and a backward search from t via its incoming edges until a common node is processed. Thus, the query requires going through the outgoing and incoming edges of a graph simultaneously. The shortest path query performance of all labeling methods is shown in Figure 5c.

Observation 2. For shortest path queries, FlipInOut outperforms all the encoding methods, which are biased towards edges in one direction.

Thus, if a query needs to retrieve both outgoing and incoming neighbors (e.g. the optimized shortest path query), a balanced numbering clearly results in better query performance. Overall, the average performance improvement for FlipInOut ranges from 18% to 86% compared to random numbering, and shows up to 7× speedup (Flickr).

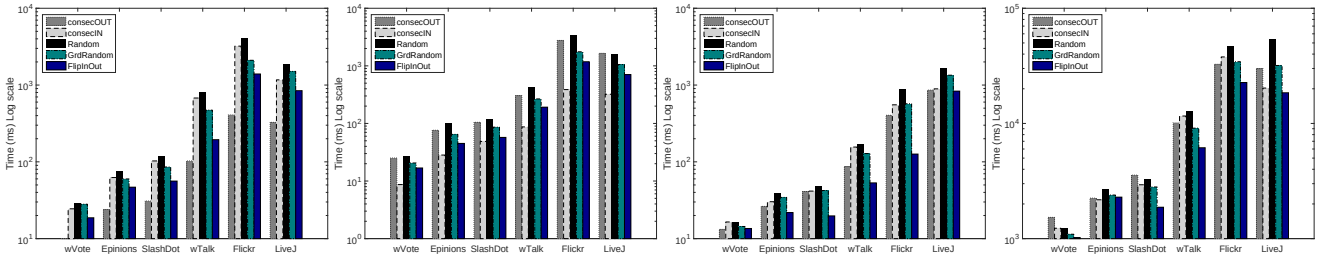
Upon closer inspection, we note that for WikiVote and LiveJ datasets, the timing difference between FlipInOut and the fastest baseline is marginal. For WikiVote, this can be attributed to the small path lengths between node pairs (most lengths are 1-2, and 19% of them are 0—non reachable node pairs). For LiveJ, we attribute the marginal difference to its high average clustering coefficient (0.27) compared to other graphs. The clustering coefficient of a vertex indicates how well-connected the neighborhood of that vertex is, and is defined as the ratio of actual edges between neighbors over the maximum number of potential edges. If its neighbors are well-connected, it is likely that the nodes required to perform an FoF-out, FoF-in, or a shortest path query are already available in memory, thus, leading to low execution time.

4.2.3 **Edge-Property Queries.** For the next set of experiments we select a pattern matching query on edges. Any query that filters the incident edges of a node based on a given property value is an example. We use a query that filters the incident neighborhood

[1] https://snap.stanford.edu/data/index.html
[2] http://konect.uni-koblenz.de

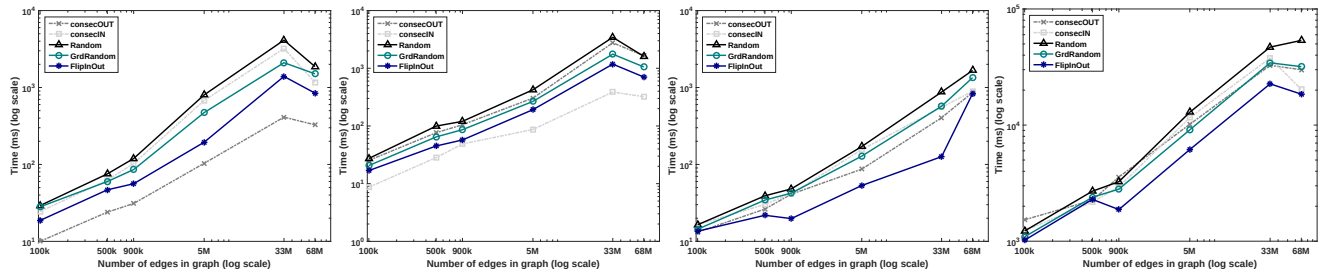
Table 2: For each dataset, we give the number of nodes and edges, the average ratio of outgoing and incoming edges over the total number of edges (OIR), the number of edges in the largest strong connected component (LCC), the average clustering coefficient (ACC), and a short description of the graph representation.

Dataset	Nodes	Edges	Avg. OIR	LCC	ACC	Description
WikiVote	8,297	103,690	0.73 / 0.27	0.38	0.14	who-votes-whom
Epinions	75,879	508,837	0.56 / 0.43	0.87	0.14	who-trusts-whom
Slashdot	82,168	948,464	0.43 / 0.56	0.96	0.06	social network
WikiTalk	2,394,385	5,021,410	0.03 / 0.96	0.29	0.05	Wiki talk network
Flickr	2,585,568	33,140,018	0.42 / 0.57	0.82	0.10	social network
LiveJ	4,847,571	68,993,773	0.49 / 0.50	0.95	0.27	social network



(a) FoF-out **(b) FoF-in** **(c) Shortest Path** **(d) Edge-property**

Figure 5: Query Performance (in ms) in real networks: FlipInOut and GrdRandom have the best combined performance for in- and out-specific queries. The runtime is measured as the average execution time over 100 runs.



(a) FoF-out **(b) FoF-in** **(c) Shortest Path** **(d) Edge-property**

Figure 6: Query Performance (in ms) vs. number of edges in each input graph (log-log scale): FlipInOut's relative improvement is robust to the graph size both for FoF-out / FoF-in (a-b), and shortest path queries (c).

(both edge directions) of a given node v based on a edge property value x. The value of x is selected such that the selectivity is around 5%-10%. The query involves retrieving both the outgoing and incoming incident edges of v. The edge-property query performance for the different encoding schemes is shown in Figure 4d.

We observe that FlipInOut numbering results in lowest average execution time across all graphs. Similar to shortest path queries, since FlipInOut attempts to balance the numbering it results in better performance when dealing with queries that involve incident neighborhood irrespective of direction. Overall, the average performance improvement for FlipInOut ranges from 10% to 78% compared to a random numbering.

Observation 3. For edge property queries, FlipInOut outperforms all the encoding methods with an average relative performance improvement ranging from 10% to 78% compared to a random numbering.

The behaviour of consecOUT and consecIN varies depending on the query mix–if the node IDs consistently have higher out-degree nodes than the in-degree of the same node, consecOUT performs better and the reverse otherwise.

4.2.4 **Scalability.** Figure 6 presents the average execution time of the encoding schemes as a function of the size of graph edges. The x-axis in the plot corresponds to the number of edges in each of the datasets (Table 2) in increasing order and the y-axis to the average time in log scale. For FoF-out and FoF-in, the dark blue star line representing FlipInOut remains consistent across the graphs regardless of the query direction.

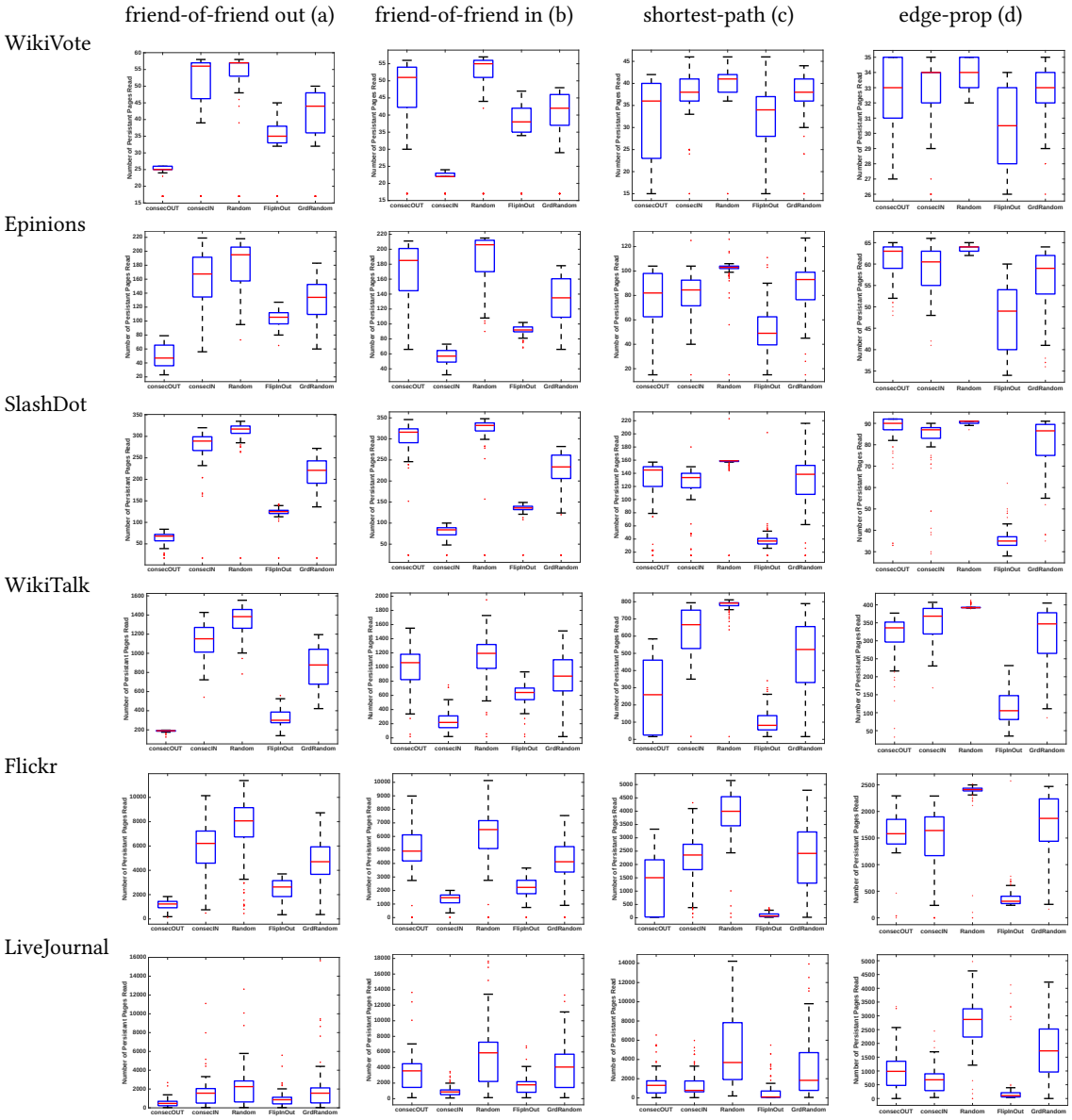


Figure 7: Disk I/O Performance of Queries

Observation 4. FlipInOut scales well with the size of the input graph, and its relative improvement is robust to the graph size regardless of the edge direction in the queries.

As explained before, the drop in timing for the LiveJ graph is likely related to the high connectivity between the neighbors that are already loaded into memory.

4.3 Disk I/O Performance

To measure locality preservation on disk and confirm our hypothesis that better edge-labeling improves the number of disk accesses, we monitored the disk I/O performance of each query. In Figure 7, for each type of query and encoding scheme, we show the total number of persistent page reads from disk (y-axis) over 100 instances of node IDs. The disk I/O for FoF-out, FoF-in, shortest path, edge property queries are shown in columns a, b, c, and d, respectively. The reported number of pages read refers to all the internal structures, including the indexes and actual data stored in bitmaps. All the statistics are recorded the first time a query is run, on a cold cache.

Figure 7 shows how the page accesses vary across different labeling schemes and their relative differences. For FoF-out and FoF-in queries, a consecOUT and consecIN layout clearly show a benefit consistently across all the graphs. FlipInOut is closer to the winner in each of the respective methods. Both the shortest path

and edge property queries seem to benefit from having a more non-biased edge layout on disk. Peculiar behaviour in the LiveJournal graph is also exhibited in these plots. Variations on these plots help explain the behaviour of query times in the previous section.

Observation 5. The number of persistent page reads correlates with the query time of the encoding schemes.

For WikiVote, there appears to be a big difference in the number of reads between the methods (which should affect its runtime), but the disk page reads are consistent across methods, ranging from 15 to just 60. The small number of reads is likely due to the network's small size—its 100K edges can be cached.

Furthermore, with the promising results shown with improved disk I/O, we believe that the proposed labeling schemes will also be useful in a distributed setting where the graph is partitioned across machines. Having locality in neighboring edges would mean less communication overhead across the network.

Disk Storage Benefit. We also compare the schemes with respect to the raw sizes of the databases they created with the different encoding methods. As shown in Table 3, FlipInOut and GrdRandom achieve comparable or better storage benefit than consecIN and consecOUT, ranging between 10%–27% reduction compared to a database with random ordering.

Observation 6. FlipInOut and GrdRandom achieve comparable or better storage benefit than consecIN and consecOUT, ranging between 10%–27% reduction compared to a database with random ordering.

The reduction in size compared to a graph with consecOUT or consecIN is marginal (at most 8%). The reason is that when the edges are sorted by source (in consecOUT), the bitmaps in the tails group (Figure 3) is optimal, but the disarray in the heads cancels out its storage benefit. The low compression benefit of consecIN for WikiTalk is due to its high in-ratio (Table 2), which means that many nodes have only incoming edges. In Sparksee (Section 2), this translates to sparse head group bitmaps, and numbering the edges of such a graph with consecIN is almost equivalent to a random numbering.

Table 3: Storage Benefit (%) compared to the Random encoding scheme. Higher is better.

Method	wVote	Epinions	SlashDot	wTalk	Flickr	LiveJ
FlipInOut	19.6	19.5	23.2	15.4	26.7	26.5
GrdRandom	18.5	18.7	21.0	10.4	25.2	24.4
consecIN	19.6	16.6	18.9	1.9	24.5	24.0
consecOUT	18.5	16.9	18.5	15.0	24.6	24.1

In conclusion, our methods generally have better and balanced runtime and disk I/O performance for a wide range of queries, and also have the side-benefit of better or comparable storage benefit to the baseline encoding schemes.

4.4 Balance of Labeling

The edge consecutiveness of graph G defined in Section 3.1 is a combination of individual metrics $C_{\text{in}}(G)$ and $C_{\text{out}}(G)$. As discussed in Section 3, our proposed methods attempt to maximize the consecutiveness and have a balance between the in- and out-consecutiveness. The method consecIN performs perfectly on the $C_{\text{in}}(G)$ metric but penalizes $C_{\text{out}}(G)$. Methods that are non-biased to a single edge direction possess the property that $C_{\text{in}}(G) \approx C_{\text{out}}(G)$, i.e., the *balance* $C_{\text{in}}(G)/C_{\text{out}}(G) \approx 1$. For query workloads that are uniform with respect to accesses of incoming and outgoing edges, it is desirable to increase total $C(G)$ while maintaining the balance of labeling.

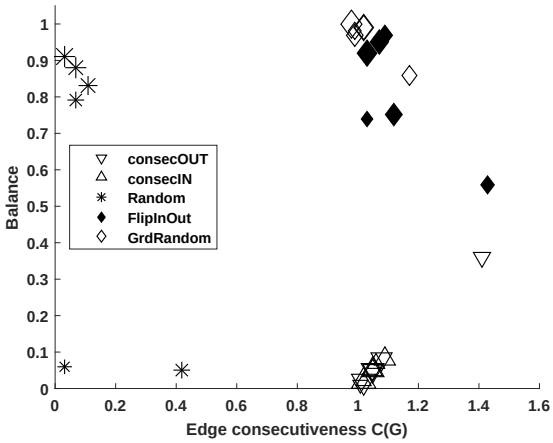


Figure 8: Trade-off between consecutiveness and balance. Markers are scaled according to graph size.

Figure 8 shows the trade-off between these two properties (balance on y-axis and total graph consecutiveness on x-axis) for all labeling methods on the datasets. The markers on the figure are scaled with the graph size. We aim to be at the upper right corner of the matrix maximizing on both consecutiveness and balance. In the bottom right, consecIN and consecOUT algorithms exhibit high $C(G)$, however fail to balance the consecutiveness. We observe that our proposed methods are consistently placed in the upper right corner demonstrating the desired balanced property of these algorithms.

5 APPLICATION: STREAMING GRAPH PARTITIONING

In this section we investigate an application area that can benefit from an improved edge labeling, namely streaming graph partitioning.

In the application of partitioning, our conjecture is that if the input stream is already pre-processed to ensure locality, better performance (e.g. lower edge cuts) can be achieved. For example, the FlipInOut edge labeling and its locality benefits can be achieved via small modifications in the crawling procedure by leveraging the true or estimated (via sampling) marginal degree distribution of the input graph.

We start by briefly summarizing the basics of the streaming partitioning model. For an *undirected* graph $G = (\mathcal{V}, \mathcal{E})$, the vertices arrive in a stream, each with the set of its adjacent neighbors. The goal is to divide the set of vertices into k disjoint partitions $(P_1, ..., P_k)$ such that inter-edges, i.e., the *edge cuts*, are minimized. Streaming algorithms [21, 25, 26], which focus on scalable partitioning solutions to large graphs with time and space constraints, assign each vertex to a partition using local graph information (e.g.

the existing partitions), and never move it. Existing work, such as LDG [25] and Fennel [26], shows that the produced edge cuts are comparable to the ones created by offline versions with access to the whole graph, such as METIS [12].

Given that the node assignment is based on increasing amount of information (i.e., the partitions at time t), the *streaming order* is an important consideration that affects the performance of the greedy node assignments to partitions. Existing work in this area usually considers three *node* streaming orders; Random, Breadth First Search (BFS), and Depth First Search (DFS). The Random ordering is practical, does not involve pre-processing, and is preferred for very large graphs.

Based on FlipInOut, we introduce a streaming graph partitioning method, FlipCut, which is (almost) agnostic to the neighborhood of each incoming vertex. We define as k the number of partitions with capacity C, and $P^{(t)}(i)$ the i^{th} partition at time t. Unlike other methods, which consider a streaming order of *vertices*, FlipCut considers one *edge*, (v_1, v_2), at a time in the FlipInOut order, and assigns its endpoints to partitions based on the following three rules:

(1) If v_1 and v_2 are already assigned to partitions, FlipCut ignores the incoming edge.
(2) If v_1 and v_2 have not been assigned to a partition yet, both of them are assigned to the partition with the minimum load at time t, i.e., $argmin_{i\in\{1,2,\ldots,k\}} \frac{|P^{(t)}(i)|}{C}$.
(3) If only v_1 is not assigned to a partition, it is assigned to the partition of its neighbor v_2, i.e., $arg_{i\in\{1,2,\ldots,k\}} P^{(t)}(i) \cup v_2$. If that partition is full, rule 2 is applied, and v_1 is assigned to the current smallest partition, i.e., $argmin_{i\in\{1,2,\ldots,k\}} \frac{|P^{(t)}(i)|}{C}$. If v_2 is the only unassigned endpoint, it is handled similarly.

In other words, the endpoints of any streaming edge dictate the order in which FlipCut will visit the vertices. We strive to keep a set of simple rules assuming that we are already working on an edge set that has improved locality. An advantage of FlipCut over other baselines is that it only inspects one edge at a time, and decides on the placement of the incoming edges' vertices *without* accessing the subgraph of already seen vertices. FlipCut does only one pass over the edges of a directed graph, and thus runs in $O(|\mathcal{E}|)$ time.

5.1 Baseline Methods and Methodology

To evaluate the performance of our streaming graph partitioning method, FlipCut, with respect to the percentage of edge cuts, we compare it to three state-of-the-art methods:

• BFS + LDG [25]: This is the best performing method among 3 node orderings and 7 partitioning heuristics in [25]. The nodes are being read in BFS order, and assigned to partitions according to the Linear Deterministic Greedy (LDG) heuristic. The idea of LDG is to assign an incoming node v to the partition with most of its neighbors, while penalizing larger partitions and imposing size of $\sim |V|/k$ vertices in each partition. In order to make the decision for a given node v, LDG looks at the partitions of all the neighbors of v.

• Random + LDG [25]: Nodes arriving in a random order is another streaming order tested due to its simplicity and scalability to large graphs. On average, it has been observed [25] to report comparable performance to BFS + LDG.

• Hashing [11, 17]: A node is hashed to a partition independent of the graph structure. The vertices can be distributed evenly across the partitions and the expected fraction of edge cuts for $k \geq 1$ partitions is $1 - \frac{1}{k}$. This technique is widely used in practice because it is simple and can efficiently determine the partition of a node without maintaining a mapping table. The performance of hashing also acts as a classic upper bound.

5.2 Results

To compare FlipCut with the baseline methods, graphs were made undirected for the LDG baselines. As the graph size grows, it is more sensible to test with higher value for the number of partitions, k. Thus, for small graphs, we test with partition sizes 4 and 8, and for the larger graphs (Flickr, LiveJ) we vary k from 12 to 100.

The percentage of edge cuts is shown in Table 4 for $k = 4$ and $k = 8$. We see that with the exception of Epinions, LDG with BFS and random ordering exhibit similar performance, which has been confirmed on other datasets as well [25].

Observation 7. For $k = 4$ partitions, FlipCut has 8% to 42% reduction in edge cuts compared to the LDG variants, and 26%-50% reduction compared to the Hashing method.

For 8 partitions, the benefit compared to BFS+LDG becomes smaller, and is very similar in the case of Epinions and WikiTalk. FlipCut outperforms the Hash partitioning by a large margin, and also has potential for practical use, given that it requires observing only a single edge at a time, without accessing the whole graph. As expected, for all the methods, the fraction of edge cuts increases with more partitions.

Table 4: Percentage of edge cuts for 4 and 8 partitions. Lower (in bold) is better. Italics indicate near-ties.

Data	BFS+LDG	Random+LDG	FlipCut	Hashing
k = 4				
WikiVote	63.63	69.93	**41.41**	75.0
Epinions	26.94	64.98	**24.87**	75.0
SlashDot	63.32	65.20	**36.48**	75.0
WikiTalk	56.45	54.47	**48.54**	75.0
k = 8				
WikiVote	78.35	82.09	**65.73**	87.5
Epinions	**41.33**	76.56	*42.5*	87.5
SlashDot	76.52	76.81	**66.85**	87.5
WikiTalk	**62.46**	64.17	*63.12*	87.5

Table 5 shows the fraction of edge cuts for Flickr and LiveJ. As done in [25], for the larger graphs we compare our method to the natural ordering provided in the original dataset. In addition, we also test with a random node permutation.

In the case of Flickr, our method shows a reduction in edge cuts compared to all the other methods. For LiveJ, although the edge cuts are improved compared to the Random and Hashing counterparts, this is not the case compared to the Natural Order +

Table 5: Percentage of edge cuts for the largest graphs with higher number of partitions, k. Lower (in bold) is better.

Data	k	Natural + LDG	Random + LDG	FlipCut	Hashing
Flickr	12	55.85	85.45	**27.04**	91.7
	24	61.65	89.74	**54.12**	95.8
LiveJ	24	**41.01**	87.88	63.67	95.8
	50	**46.99**	90.56	70.16	98.0
	100	**51.74**	92.04	75.00	99.0

LDG. We speculate that one reason for this discrepancy is the high clustering coefficient of LiveJ compared to the other graphs (Table 2). It may have had an adverse effect on FlipCut, since it does not use the graph structure information (other than the current edge) when deciding the vertex assignments. We note that independent work also reports that the LiveJ graph exhibits different behavior from other social networks; Chierichetti et al. [6] focus on network compression and claim that the natural crawl order outperforms their Shingle ordering method.

This experiment shows the potential applicability of FlipCut on generating partitions in a streaming setting. Our results are promising, as they display consistently better cuts compared to LDG with random ordering. We emphasize that FlipInOut, which was designed with a different goal in mind and was not optimized for reducing the edge cuts, has the side-benefit of supporting streaming graph partitioning. Lastly, existing algorithms generally work on undirected graphs inspecting $2|E|$ edges while FlipCut can produce comparable results observing only half the edges for a directed graph.

6 CONCLUSION AND FUTURE WORK

In this paper we proposed the problem of edge-labeling on directed graphs, mainly to support efficient neighborhood-related edge queries. We presented two novel and efficient edge-labeling schemes, GrdRandom and FlipInOut, which seek to balance and maximize the consecutiveness at every individual vertex so that local decisions greedily make progress towards the global optimal. Our purpose was to experimentally evaluate the potential overall benefit that a relabeled edge list will have on query processing. We observed that our edge-labeling schemes did in fact lead to significantly improved query times and disk I/O performance by achieving a better layout and locality of edges on disk. Based on FlipInOut, we also introduced FlipCut, an effective one-pass, neighborhood-agnostic strategy for streaming graph partitioning.

For future work, we plan to first validate our conjecture that the edge-labeling maximization problem is polynomial time solvable. On the other hand, proving its $\mathcal{NP}$-hardness and coming up with bounded approximation algorithms will also be interesting. There also exist several node reordering methods [6, 8, 28] that are not directly applicable to solve our consecutiveness maximization problem. However, correlations may exist between node and edge ordering methods for optimizing different types of graph queries. Performing such correlation studies is also necessary. Last but not least, online node and edge-labeling schemes should be considered together for optimizing real-time systems such as graph streaming.

REFERENCES

[1] R. Andersen and Y. Peres. Finding sparse cuts locally using evolving sets. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, 2009.

[2] A. Apostolico and G. Drovandi. Graph Compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[3] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast Unfolding of Communities in Large Networks. *J. Stat. Mech. Theor. Exp.*, 2008(10), 2008.

[4] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. 2004.

[5] J. J. Carrasco, D. C. Fain, K. J. Lang, and L. Zhukov. Clustering of bipartite advertiser-keyword graph. In *ICDM*, 2003.

[6] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[7] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *PVLDB*, 6(11):1150–1161, Aug. 2013.

[8] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. 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. ACM, 2016.

[9] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC*, 1982.

[10] J. Gehweiler and H. Meyerhenke. A distributed diffusive heuristic for clustering a virtual p2p supercomputer. In *IPDPSW*. IEEE, 2010.

[11] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, 2009.

[12] G. Karypis and V. Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[13] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49, 1970.

[14] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical Properties of Community Structure in Large Social and Information Networks. In *WWW*, 2008.

[15] J. Leskovec and R. Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.*, 8(1):1:1–1:20, July 2016.

[16] Y. Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Trans. Knowl. Data Eng.*, 2014.

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[18] N. Martínez-Bazan, M. A. Águila Lorente, V. Muntés-Mulero, D. Dominguez-Sal, S. Gómez-Villamor, and J.-L. Larriba-Pey. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering and Applications Sysmposium*, 2012.

[19] M. F. Mokbel and W. G. Aref. *Encyclopedia of Database Systems*, chapter Space-Filling Curves. 2009.

[20] M. E. J. Newman. Modularity and community structure in networks. *PNAS*, 103(23), 2006.

[21] J. Nishimura and J. Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, 2013.

[22] S. Papadimitriou, J. Sun, C. Faloutsos, and P. S. Yu. Hierarchical, Parameter-Free Community Discovery. In *ECML PKDD*, 2008.

[23] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *, 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 2013.

[24] K. H. Randall, R. Stata, J. L. Wiener, and R. G. Wickremesinghe. The link database: Fast access to graphs of the web. In *Proceedings of the Data Compression Conference*, DCC '02, 2002.

[25] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2012.

[26] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, 2014.

[27] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *Proceedings of ICDE*, 2014.

[28] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.