I/O Efficient Search of Large Social Networks

Paul T. Stanton
Johns Hopkins University

Randal Burns
Johns Hopkins University

Abstract

We introduce an I/O efficient algorithm and data structure to support fast decentralized search in large graphs modeling social networks. We structure network data in a homophily-based social hierarchy using an append-only, block-aligned skip list with an embedded tree micro-index, which reduces I/O and cache line faults. We further minimize I/O when building the skip list by combining an extended memory sorting algorithm with buffered insertion. The system supports ad hoc queries efficiently: the combined cost of skip list construction and search improves performance in large social network graphs (11 million vertices and 150 million edges) by a factor of four over the fastest known external memory search algorithm and by several orders of magnitude over a traditional in-memory search algorithm. Subsequent searches that use the same skip list as an index are an order of magnitude faster still. Finally, these search structures work efficiently over a much wider range of homophily values than theory predicts.

1 Introduction

Graphs of communication and interaction occurring in a social network are an effective tool for post-event analysis and forecasting. Logs of information flows can be transformed into a graph and searched to answer questions of the form, “is node B impacted by an event triggered by node A?” In such a graph, we assign a vertex for each communicating node and an edge to indicate that a communication event has occurred. A path from A to B implies that B may have been affected by an event triggered by A, whereas the absence of a path supports a definitive claim that an event generated by A has had no affect on B. Examples of scenarios where this type of analysis is valuable include measuring the effectiveness of online advertisements [9], network forensic analysis [8], and determining connections in terrorist networks [14].

However, current graph construction and search algorithms can be prohibitively slow when analyzing data sets that exceed memory capacity. Graphs derived from logs of e-mail traffic, cell phone conversations, text messages, tweets, and other forms of communication easily exceed memory bounds with millions of nodes and billions of edges; a recent study of Microsoft Messenger reports that an average of 90 million distinct users participate in 1 billion conversations with 7 billion exchanged messages in a day [10]. Traditional in-memory search algorithms prove too slow, suffering as many as \( O(E + V) \) I/Os in which \( E \) is the number of graph edges and \( V \) the number of vertices.

Prior work in external memory graph algorithms and theoretical work in social networks address portions of the problem in isolation. I/O costs dominate performance for large graphs [18], resulting in much work focused on performing efficient construction and search of out-of-core graphs [2, 12, 13]. However, prior efforts have been designed to work in the general case without considering specialized algorithms for graphs that exhibit characteristics common in social networks. Watts and Strogatz captured some of these characteristics in a “small-world” framework [17] in which nodes are highly clustered but also have a few “long” edges that create short paths between any two communicating nodes. Kleinberg later defined a hierarchical model that supports a simple greedy search algorithm for finding these short paths [7].

We develop a data structure and algorithm to efficiently build and search out-of-core graphs that model social networks. We intersect prior work in external memory algorithms with previous efforts to find short paths in small-world networks using our efficient skip list. First, we employ the I/O efficient STXXL library [5] to convert a list of communication events into an adjacency list graph representation. Then we develop a homophily-based social hierarchy using a block-aligned, append-only skip list. Finally, we execute a greedy search of the graph using the height of a node and its adjacent nodes’ least common ancestor in the hierarchy as a selection metric.

Creating and querying the skip list are efficient operations owing to a block-aligned and cache-conscious design that employs an embedded tree micro-index [11] and buffering. We adjust the fan-out of the skip list to coincide with the number of elements that will fit in a disk block to minimize I/O. We also embed a small tree of keys in each block to index the block’s data, thus offsetting the poor cache performance resulting from searching disk optimized blocks. Nodes of the embedded tree are aligned with the cache line size, suffering a small, constant number of cache misses in contrast to the \( \log_2 b \) cache misses associated with a binary search. We also amortize I/O during skip list creation by buffering all inserts and only fully inserting them when the node-sized buffer fills to capacity.

Our search algorithms and supporting data structure result in considerable speedup over existing approaches. The construction of our skip list and accompanying hierarchical search provide a factor of four improvement over general external memory searches and a speedup of several orders of magnitude over an in-memory search of large graphs. Our algorithm successfully builds and searches a graph with over 11 million vertices and 150 million edges in less
than 8 minutes, indicating that ad hoc search of large social networks is feasible. Subsequent searches over the same data set reuse the skip list index and avoid the one-time creation cost, improving the cost of additional searches by two orders of magnitude over performing an additional breadth first search. These combined results indicate that specialized algorithms can be very efficient extensions to existing approaches for particular subsets of graph problems.

2 Background and Related Work

Small-world Graphs: A small-world graph exhibits the special property that even though each node has a small constant number of neighbors, any two nodes are connected by a short path. Watts and Strogatz formalized a framework that employs social connections that form “long” edges to cross-cut a clustered graph instead of following edges for the entire diameter to traverse from one node to another [17]. The existence of the resultant short paths indicates that efficient search is possible.

However, algorithmically determining the edges that contribute to short paths is difficult. Large graphs that exceed memory capacity or social network graphs in which nodes are only aware of their immediate neighbors require decentralized search—finding a target relies on only local knowledge of the graph topology. Searching for short paths in this environment is especially difficult due to social network graph characteristics that include node sparsity, large diameter, poor spatial locality, and poor temporal locality during search using traditional algorithms [1].

Decentralized Search with Hierarchical Models: Kleinberg presented a hierarchical structure and associated decentralized search algorithm to find short paths [7]. He structured data into a tree based on social relationships in which leaf nodes represent the vertices from the original graph and parent nodes define the social commonality among children. The tree edges have no relation to the original graph edges, but rather connect nodes based on some mutual social relationship. For instance, positions within an organization can be represented in a hierarchical fashion: a payroll clerk reports to the head of the payroll division who reports to the head of finance who reports to the CEO (see Figure 1).

The hierarchy forms a tree $T$ such that a recursive algorithm can determine the social distance between a source node $s$ and a target $t$ by finding their least common ancestor. A resultant tree $T'$ is rooted at the common ancestor with $t$ further contained within a subtree $T''$ (Figure 1). The algorithm greedily chooses the next source node in a graph traversal by minimizing the height of $T'$ and attempting to select a useful node in $T''$ that is connected with edges from the original graph. If $s$ is not connected to a useful node in $T''$, the algorithm selects the next node from $s$’s adjacency list in lexicographical order.

This process finds short paths when certain graph properties hold regarding the node out-degree, the probability of edge distribution, and the strength of the social connection between nodes. For the latter condition, Kleinberg introduced a homophily parameter, $\alpha$, to define the social tendency for nodes to associate with others that share similar characteristics. Specific to the hierarchical model, homophily measures how likely nodes from one subtree in the hierarchy are to be connected with nodes in subtrees rooted at different tree levels. Formally, the homophily parameter contributes to the necessary condition that the probability of two nodes being connected is proportional to $b^{-\alpha h}$ in which $h$ is the height of the tree and $b$ is the tree fan-out. Another necessary property establishes a bound on node out-degree as $k = c \log_\alpha n$ in which $c$ is a constant and $n$ is the number of nodes in the original graph. When $\alpha = 1$ and $c$ is suitably large, a search results in paths of length $O(\log_\alpha n)$ [6].

Adjusting the value of $\alpha$ adjusts the probability of how a node’s edges are distributed across the various levels of the hierarchy. When $\alpha = 1$, each node’s edges are equally likely to connect to another node at each level; there is equal likelihood that there is an edge connecting to an immediate node with a nearest ancestor height of $1, 2, \ldots \log_\alpha n$. When $\alpha < 1$, there is an increasing degree of randomness in the likelihood of edges connecting to nodes at particular levels until $\alpha$ reaches 0, at which point the distribution of edges is completely random. In contrast, as $\alpha$ grows larger than 1, the distribution of edges becomes increasingly clustered.

The variance in edge distribution across the hierarchy translates to the likelihood of finding short paths in social networks. Kleinberg proved that short paths can be found efficiently and algorithmically when edges are evenly distributed ($\alpha = 1$), but that short paths cannot be found efficiently otherwise. As $\alpha$ approaches 0, there are a large number of “long” edges in the graph, but they are too random to find; a search jumps from node to node erratically. When $\alpha$ is too large, there are not enough “long” edges and a path must traverse the diameter of the graph.

Parallel Disk Model: External memory algorithms are often described using the parallel disk model to explain how they exploit data locality to minimize I/O. The parallel disk model...
model [16] uses the following parameters to describe I/O complexity:

\[ N = \text{problem size (in units of data items)} \]
\[ M = \text{internal memory size (in units of data items)} \]
\[ B = \text{block transfer size (in units of data items)} \]
\[ D = \text{number of independent disk drives} \]

As an example, scanning \( N \) items is \( \Theta(N/DB) \), implying that data items are tightly grouped into block-sized units and separated on independent disks such that during the time to perform one I/O, the system retrieves the number of items that fit within a block simultaneously from each disk. Similarly, the I/O complexity for sorting \( N \) items is \( \Theta(N/DB \log_{M/B} N/B) \). For brevity, these two common operations are often simply referred to as \( \text{scan}(N) \) and \( \text{sort}(N) \).

**External Memory Search:** From the considerable prior work in designing I/O efficient construction and search algorithms for large graphs, we select the Mehlhorn and Meyer breadth first search (MM-BFS) [12] algorithm for analytical comparison owing to Ajwani’s empirical study demonstrating its superior performance on large sparse graphs [2].

MM-BFS extends the Munagala and Ranade BFS (MR-BFS) algorithm [13] to reduce the I/O complexity during the search of large graphs. A traditional, in-memory implementation of a BFS algorithm can exhibit poor I/O performance due to two factors: (1) determining the next node to evaluate can result in evaluating \( O(|E|) \) vertices causing \( O(|E|) \) I/Os, and, (2) the adjacency list for each node is unstructured resulting in \( O(|V|) \) I/Os in the worst case. Therefore, an inefficient approach has I/O complexity of \( O(|V| + |E|) \). MR-BFS minimizes I/O by addressing the first deficiency. MR-BFS reduces the number of nodes that must be evaluated at each BFS level by removing the nodes evaluated during the development of the two previous BFS levels. Removing nodes requires scanning and sorting the nodes from previous evaluations. The resulting I/O is \( O(|V| + \text{sort}(E + V)) \). MM-BFS addresses the second deficiency by first preprocessing the graph into disjoint groups of vertices that fit within memory and then applying MR-BFS to the smaller subgraphs. Intuitively, the partitioning guarantees that the nodes in the adjacency list fit within memory and do not incur an I/O upon evaluation. The final I/O complexity of MM-BFS is on the order of \( O(\text{sort}(E + V)) \) I/Os.

3 Design

Data structures used to represent graphs and to aid in graph search must be created efficiently. Data sets that represent communication events are often structured as logs, raw data files, or database entries. To provide analytical benefit, this data must first be converted into a graph which is then searched. Problems associated with constructing graphs and searching social networks are exacerbated by their dynamic and ad hoc nature that quickly reduce a graph’s relevance and utility. The analytical significance of the graph data is determined by parameters such as a time period and a set of node characteristics; changing these parameters requires a revised graph and new search. Additionally, the continual and ad hoc interaction between nodes in a social network necessitates building the supporting data structures quickly.

Data structures and algorithms must support both one-time ad hoc searches as well as repeated search over the same data set. Whereas one-time searches support point queries regarding nodes and their relationships, repeated searches help to find connections between multiple communicating nodes [8, 14]. The cost of building data structures needs to be minimal to support the former, and using the data structures to aid search must be optimized to support the latter.

Toward meeting these requirements, we combine existing algorithms and data structures with a search-optimized skip list and homophily-based search algorithm to efficiently create and search graphs modeling social networks. We focus on a specific subset of graph problems—those within a social context that can be modeled hierarchically, those that contain nodes with a sufficiently large out-degree, and those with a searchable \( \alpha \) homophily value.

**STXXL:** We use the Standard Template Library for Extra Large Data Sets (STXXL) [5] to create and traverse a large graph. STXXL is specifically designed to overlap computation and I/O, avoiding disk access latency by directly pipelining results from one algorithm’s computation to the next algorithm without intermediate writes to disk. Users configure the block size, memory size, and disk space on independent drives to make use of the \( B, M \), and \( D \) parameters from the parallel disk model. We use the STXXL streaming layer to convert a graph represented as a random file of edges into a sorted and merged adjacency list representation without duplicates in \( O(\text{sort}(E + V)) \) I/Os.

**Search-Efficient Skip Lists:** We construct a block-aligned, cache-conscious skip list to store a hierarchical tree representing the social structure of the network. Block-alignment ensures that (1) a single I/O retrieves all elements within a node and (2) the fan-out is sufficiently large to minimize the data structure height, limiting all query operations to \( O(\log_B N) \) I/Os. We opt to use a skip list due to its efficient handling of append-only workloads that avoids costly tree rotations associated with B-tree variants. STXXL’s sorted adjacency list of vertices supports buffering append-only inserts in node-sized units to amortize updates. Once the buffer fills to capacity, an entire node is fully inserted into the leaf level of the skip list, resulting in only a single update to parent nodes.

Using large block sizes improves I/O performance, but it may result in poor cache performance due to cache
misses incurred while searching the many elements within a block [4]. We use an embedded tree micro-index [11] to offset the otherwise poor cache performance demonstrated during search operations with disk optimized block sizes. We embed a small tree of data keys within each block that index the block’s data. Our organization tightly packs the keys into embedded tree nodes that are aligned with the system’s cache line size. Searching for a data element results in traversing the embedded tree to find the element’s key, converting the key to a page offset, and retrieving the data element by offset. The search incurs only a single cache miss at each level of the embedded tree and an additional cache miss to load the actual element.

The micro-index is statically aligned according to system parameters to permit tree traversal based on address arithmetic instead of following pointers. The disk block size is divided by the size of a key/value element to determine the number of data elements that will fit within a block, \( B \). It then constructs the tree from the bottom up, ensuring that all cache lines containing leaf nodes are full and no more than one line on each higher level is less than full (Figure 2). The number of levels is \( \log_c B \) in which \( c \) is the number of fixed-length indexing keys that fit within a cache line. For realistic values of \( B \) and \( c \), there are no more than 3 levels in the internal tree, resulting in 4 cache faults to retrieve the key value. This small, constant number of cache misses is in contrast to the \( \log_2 B \) cache misses incurred during a binary search of a flat block (using 512 KB blocks, \( B = 1776 \) resulting in 11 cache faults).

**Hierarchy-based Search Algorithm:** Our homophily-based search is a simple greedy algorithm that selects the next node to traverse based on the shortest tree distance to the target. The inputs are a graph that has been stored in an adjacency list representation. Our hierarchical search then uses the resultant sorted adjacency list to build the skip list as a prerequisite to initiating the search. The algorithm selects the next node as the one with an ancestor at the least height.

We resolve ties (nodes that reside in a subtree with a nearest ancestor at the same height) greedily by selecting the first node encountered with the common ancestor. Depending on the context and the social parameter on which the social hierarchy is built, ties can be resolved more efficiently: if there is a semantically meaningful way to compare subtrees within the social context of the hierarchy which makes a particular node closer to the target, that node should be selected. However, the definition of “closer” is not always clear. In the example in Figure 1, the neighbor nodes of \( s \) are encountered in lexicographical order \( \{x, u, v, w\} \). The node \( x \) is originally selected to be the best option, but then is replaced by \( u \) after \( u \) is evaluated. When \( v \) is evaluated, it has the same nearest ancestor and resides in the same subtree as \( u \). The best option greedily remains \( u \) since it is not semantically clear whether “Current Operations” or “Planning Operations” is closer to the target in “Payroll”. We default to the greedy choice to keep the implementation universally applicable.

**4 Evaluation**

We evaluate our implementation of the hierarchy-based search to determine its performance characteristics at scale. All experiments are performed on a Dell Precision T7400 workstation with dual quad-core 2GHz processors, 8GB of RAM and a 6 MB L2 cache. We configure STXXL with one 16 GB disk, 256 MB of RAM, and 512 KB blocks.

**Graph Generation:** In order to evaluate our system’s performance on large social network graphs, we need data sets that exhibit the properties of Kleinberg’s model (Section 2). Specifically, nodes need to fit within a social hierarchy, they must have sufficient out-degree, and the edge distribution must align with tree properties. Further, we need to manipulate these parameters and the size of the graph to analyze the impact on the algorithms’ performance. As a result, we developed a graph generator able to parameterize the hierarchy’s fan-out \( b \), the height \( h \), the value of the homophily parameter \( \alpha \), and the out-degree of nodes \( k \).

All times reported for our evaluations include the time to convert raw data representing node events into search results. The output of the graph generator is a file consisting of an unordered list of edges of the form \( \{(u, v)\} \). These edges must first be converted into a graph; all of our evaluated systems use an adjacency list representation. Our hierarchical search then uses the resultant sorted adjacency list to build the skip list as a prerequisite to initiating the search, whereas other search algorithms immediately begin the search process. Thus, the reported times include the time to read the file of events, build the necessary approach-specific data structures, and perform the search.

**Ad Hoc Search:** We evaluate the data structure construction and search time to demonstrate our system’s ability to support ad hoc searches. We compare the Boost Graph Library’s implementation of breadth-first search [15], Ajwani’s STXXL-based implementation of the Mehlhorn and Meyer BFS (MM-BFS) algorithm [3], and our hierarchy-based search. For the hierarchy search we set \( c = 10, b = 15 \) and \( \alpha = 1 \). We create and search multiple graphs that range in size from 3000 nodes to 11 million nodes.

For large graphs, our hierarchy-based search outperforms MM-BFS by a factor of four (Figure 3); we highlight...
that it is our system’s combined time to construct the graph, build the skip list, and search that compares favorably to the MM-BFS algorithm. This is significant, because previous work showed that MM-BFS outperforms other search algorithms on large graphs [2]. The performance of the hierarchical search demonstrates that a specialized search algorithm can significantly lower cost in relation to a general implementation if graphs can be categorized with searchable homophily parameters and a suitable out-degree. The Boost implementation outperformed the external memory algorithms until the size of the graph exceeded the available memory; the overhead of creating an external-memory efficient graph representation is not worthwhile until graphs exceed the memory bounds. Once Boost ran out of memory, the search took over 10 hours.

**Indexed Search:** We compare our algorithm to MM-BFS in additional detail to demonstrate the benefit derived from subsequent searches over an existing skip list index. We separate the one-time skip list construction cost from the search cost in Figure 4. Both approaches use the same algorithm to construct the on-disk graph representation, explaining their near comparable time. MM-BFS requires no additional data structures, whereas our hierarchical search must construct the skip list. However, the resultant data structure can then be reused for subsequent searches over the same data set. The comparison of our system’s performance of additional searches of the same graph results in two orders of magnitude improvement over MM-BFS. Subsequent search of the 11 million node graph with different source and target nodes requires 17.7 seconds for hierarchy-based search and 1702.5 seconds for MM-BFS.

**Adjusting Homophily Parameter:** Next we adjust the homophily parameter to determine the impact of randomness on the searchability of a graph and to define a range in which hierarchy-based searches are effective. We create multiple graphs using a range of values for $\alpha$ while holding $c = 10$, $b = 5$, and $h = 8$ (there are 390k nodes and 3 million edges in the graph). We provide search times for MM-BFS as a baseline.

Adjusting the homophily parameter changes the probability distribution of how edges connect a node to other nodes at varying heights of the hierarchical tree. When $\alpha = 1$, a node’s edges are equally likely to be distributed across each height of the tree. When $\alpha = 0$ edges are distributed randomly across the varying heights of the tree. Values less than 1 have an increasing degree of randomness as they approach 0. Values greater than 1 result in nodes that are more likely to be clustered without long edges.

Our results in Figure 5 demonstrate that efficient search exceeds the theoretical prediction associated with short paths; short paths are not a requirement for efficient search. Kleinberg’s model shows that homophily values bounded near one result in short paths, but efficient search does not share the upper bound. For $\alpha$ values approaching 0, there is an increasing degree of randomness that makes the target node difficult to find and search inefficient; short paths exist, but they are difficult to find. As $\alpha$ approaches one, short paths exist and can be found efficiently resulting in fast search. Importantly, as $\alpha$ surpasses one, search remains

![Figure 3: Comparison of an internal memory algorithm (Boost BFS), an efficient external memory algorithm (MM-BFS), and our hierarchical search on large graphs.](image1)

![Figure 4: Detailed comparison of time to construct data structures and search using an efficient external memory algorithm (MM-BFS) and our hierarchical-based search.](image2)

![Figure 5: Varying the homophily parameter.](image3)
fast even though nodes are clustered and there are few short paths. Even though the path is long, the algorithm continually directs the search toward the target and the efficiency of the skip list comparisons makes long paths manageable.

5 Conclusion
We have demonstrated that specialized algorithms and data structures can dramatically improve search performance on large graphs modeling social networks. We have combined the two otherwise disparate fields of external memory graph algorithms and social networking algorithms by implementing an efficient out-of-core data structure for managing operations on large social network graphs.

References