Atlas: Approximating Shortest Paths in Social Graphs

Lili Cao, Xiaohan Zhao, Haitao Zheng, and Ben Y. Zhao

Computer Science Department, U. C. Santa Barbara

Abstract. The search for shortest paths is an essential primitive for a variety of graph-based applications, particularly those on online social networks. For example, LinkedIn users perform queries to find the shortest path "social links" connecting them to a particular user to facilitate introductions. This type of graph query is challenging for moderately sized graphs, but becomes computationally intractable for graphs underlying today's social networks, most of which contain millions of nodes and billions of edges. We propose *Atlas*, a novel approach to scalably approximate shortest paths between graph nodes using a collection of spanning trees. Spanning trees are easy to generate, compact relative to original graphs, and can be distributed across machines to parallelize queries. We demonstrate its scalability and effectiveness using 6 large social graphs from Facebook, Orkut and Renren, the largest of which includes 43 million nodes and 1 billion edges. We describe techniques to incrementally update Atlas as social graphs change over time. We capture graph dynamics using 35 daily snapshots of a Facebook network, and show that Atlas can amortize the cost of tree updates over time. Finally, we apply Atlas to several graph applications, and show that they produce results that closely approximate ideal results.

1 Introduction

The search for shortest paths is a critical component underlying numerous Internet services and applications. It is a particularly useful primitive for operators of online social networks (OSNs), whose operations involve analyzing and understanding the relationships of users in massive social graphs. OSNs like Facebook, Twitter and LinkedIn are some of the most popular destinations on the Internet. With more than 400 million users, OSNs like Facebook face the difficult challenge of processing shortest-path related queries on the massive social graph in near real time.

More specifically, OSN operators are often interested in resolving two types of graph queries. First, they would like to support high level graph analysis operations such as computing graph eccentricity in subgraphs, or detecting/ locating central nodes in subgraphs for targeted ad placement. These applications each require large numbers of shortest path distance computations between pairs of nodes. Second and more importantly, OSN operators need to determine and display the social relationships that connect two users. For example, a user in the LinkedIn business network generates queries to determine how she is connected through mutual friends to potential new contacts. Similarly, in the Overstock social auction site, a user is more likely to make a purchase if she knows the sequence of friendships that connect her to a seller [26]. In these scenarios, we not only need the number of hops separating two users, but also an *exact path of nodes* connecting them.

Two additional issues make the problem even more challenging. First, popular OSNs have social graphs with hundreds of millions of users and billions of edges. In many cases, even a barebone edge-list representation of a large graph will not fit in a server's main memory, making even basic graph operations impractical. Second, these graphs are constantly changing, and a graph analysis system must be amenable to frequent graph updates.

While there are a number of different techniques for computing shortest paths and node distances in graphs, none are appropriate for the scale and type of graphs present in online social networks. For a graph with $|V|$ nodes and $|E|$ edges, traditional techniques like Dijkstra, Breadth-first-search (BFS), or Floyd-Warshall can find shortest paths in $O(|V|log|V| + |E|)$ time, or shortest-path for all node pairs in $\Theta(|V|^3)$ time. These techniques do not scale to million-node graphs, and can take up to a minute per path computation even on today's hardware [22]. A second approach is to precompute all shortest paths offline. However, the set of all possible paths for large graphs would not fit in physical memory of today's servers. In addition, online social networks are highly dynamic, and the addition of new nodes and edges can quickly invalidate precomputed paths. Finally, techniques such as the A* algorithm assume planar properties missing from social graphs [16], and graph embedding approaches can only estimate distance between nodes, but not provide actual paths connecting them [29].

In this paper, we investigate the feasibility of an alternative, light-weight approach towards approximating shortest paths between node pairs on a graph, using large social graphs from real world measurements. We propose *Atlas*, a system that provides accurate estimates of shortest paths by using a constant number of spanning trees to capture significant structures of the graph. By consulting the path between a pair of nodes on multiple spanning trees, we can find real paths between them that closely approximate or match their actual shortest path on the full graph.

Our approach to shortest path computation has several key benefits. First, the spanning trees used by our approach are both compact and easy to compute, making it ideal for distributed computation across clusters. For a graph with $|V|$ nodes and $|E|$ edges, each spanning tree consists only of $|V|$ nodes and $|V| - 1$ edges, and can be built using a single breadth-first-search (BFS) across the graph (in $O(|E|)$ time). For highly connected social graphs with average node degree close to 100, this represents a two-orders of magnitude size reduction compared to the original graph. An OSN provider can easily distribute Atlas across a cluster by maintaining one spanning tree per server. Even if a single tree becomes too large for a server's main memory, it is easy to further decompose and distribute the tree into multiple branches on multiple servers using a node index server. Second, spanner forests approximate not only node distance, but also provide actual short paths connecting each node pair. Finally, even as the actual graph changes with the addition or removal of nodes and edges, we can maintain the accuracy of Atlas paths by incrementally updating spanning trees based on query results.

We make four key contributions. First, we describe the Atlas approach towards approximating shortest paths on large social graphs (Section 3). We also explore different spanning tree construction techniques, investigate their impact on shortest paths, and propose mechanisms to process graphs that do not fit into main memory (Section 4). Second, we evaluate the accuracy of Atlas paths on a variety of social graphs from

OSN measurements, including data from Facebook, Orkut, Flickr, LiveJournal, and finally Renren, the largest OSN in China (Section 5). Our approach scales to a large Renren graph with 43 million nodes and 1 billion edges. Results show that with a small number of trees ($\sim$20), Atlas can produce paths between nodes that closely approximate their shortest path, using only 0.3 milliseconds per query on commodity servers. Third, we also examine the efficacy of using Atlas in graph analysis applications, and show that its approximate path can be used to produce accurate application results (Section 5.5). Finally, we propose and evaluate different approaches to maintain Atlas under graph dynamics, and show that we can maintain high accuracy by performing simple, incremental updates to the spanning trees (Section 6).

2 Goals and Potential Solutions

Before we describe Atlas in more detail, we first define our problem, and specify our goals. We then group and describe existing approaches, and consider their appropriateness for our problem. Finally, we describe graph spanning trees and explain why they are a potential attractive solution.

2.1 Goals and Problem Definition

We are primarily interested in the problem of identifying short paths between node pairs in large social graphs. While we target dual goals of estimating node distance and finding a short path between nodes, we focus our attention on the latter, since an approximate shortest path would answer both questions. More specifically, given a social graph $G = \{V, E\}$, our goal is to build a system that answers shortest path queries in real-time. Given a query with two nodes a and $b \in V$, the answer is an approximate shortest path connecting a to b. These queries are more general than queries that only require shortest path length (node distance), since node distance can always be computed from approximate paths.

We assume graph G is large, *i.e.* has more than 100,000 nodes, and exhibits the general characteristics of a social graph, *i.e.* high connectivity with an approximate Power-law degree distribution. For instance, the social graph of Renren [4] has 43 million nodes and 1 billion edges as of Jan. 2010, which requires 22 GB to store using an efficient neighbor list representation. Graph processing libraries such as the Boost Graph Library or NetworkX generally require a 5-7x increase in memory footprint. Thus processing queries by loading the entire graph into the memory of a single machine is not feasible. Note that processing a query usually needs much more memory than merely storing the graph.

2.2 Existing and Related Work

Having defined the challenge before us, we begin by first reviewing existing and recent related work in the area. More specifically, we look at several different classes of algorithms for computing or estimating shortest paths in large graphs, and consider the feasibility of applying each to our problem.

Dijkstra's Algorithm. Dijkstra's Algorithm [8] is the most basic algorithm to compute point-to-point shortest paths for general graphs. On unweighted graphs of $|V|$ nodes and $|E|$ edges, it reduces to Breadth-First-Search (BFS), and the complexity is

$O(|V|+|E|)$. To find shortest path between a and b, it searches the nodes inside a circle of all equidistant nodes centered at a. The circle gradually grows until it touches b, at which point the shortest path is found. An improved version is Bidirectional Dijkstra, which grows two circles from both a and b, and the shortest path is found once the two circles intersect. On large-scale graphs, particularly those with high average node degree, a Dijkstra (or Bidirectional Dijkstra) search will visit a significant portion of the graph within $3-4$ hops, and thus does not provide the performance we desire for massive social graphs.

Planar-Driven Algorithms. Other techniques have been proposed to improve Dijkstra's efficiency on large geographic maps. We categorize them into two types. One type leverages the planar property of map graphs. The A* algorithm [16] is similar to Dijkstra, except that it grows the search area with a bias. Preference is given to the nodes that are estimated to be closer to the destination. On planar geographic graphs, Euclidian distance [16, 21] provides a good estimate of proximity. The ALT algorithm [12, 13] uses a set of carefully chosen landmarks to improve this estimate. Estimates on the distance between any pair of nodes are obtained by applying the triangular inequality to their respective distances to a set of landmarks. To obtain good estimates, the landmarks are usually chosen at the edge of the planar area [13]. Others [20] partition the graph into regions and guide the search using region-specific information.

Structure-Driven Algorithms. The second type of techniques leverages the inherent structural properties of geographic graphs. The idea is to focus the search on structural highways in the graph to shrink the search space. For example, reach-based algorithms [11, 14] identify structural highways using the notion of *reach*. Informally, the reach [14] of a node measures how important it is to the shortest paths between other pairs in the graph. A node with a high reach maps to a crucial point on a backbone, and a node with a low reach usually exists on a local road system. By focusing on high-reach nodes, reach-based algorithms greatly speed up the search. The authors in [24] propose to apply the highway extraction technique iteratively to obtain a hierarchy of increasingly simplified highways. A search starts from the lowest level of the hierarchy and moves to the higher levels as it goes.

Landmark-Based Algorithms. Researchers have extensively used landmark-based approaches to estimate distances in graph structures. These approaches select a subset of nodes as landmarks and pre-compute the distances from each landmark to all other nodes in the graph. To answer a query, they examine the shortest paths through any of the landmarks, and pick the shortest one among them. [17] shows that randomly picking landmarks can achieve good theoretical guarantees. [22] experimentally shows that selecting landmarks using basic metrics may generate more accurate approximations. The Orion system [29] uses a similar idea, but embeds the entire graph into a low-dimension Euclidean space, then uses Euclidean distance computation to estimate node distances. Finally, Song et al. recently proposed two solutions to the complementary problem of scalable proximity estimation for online social networks [25]. And these algorithms are designed to estimate *shortest path distance*, not to locate an actual path.

Spanner-Based Algorithms. In graph theory, people have used the notion of graph spanners to compute approximate shortest paths. A spanner is a subgraph with all the nodes and only a small subset of edges. Many algorithms with provable guarantees

have been designed to construct spanners that approximate shortest paths. Algorithms in [9, 10] produce a spanner with $O(|V|^{3/2})$ edges, where shortest paths between any pair of nodes on the spanner are at most 2 hops longer than those in the original graph. With the same upper bound on spanner size, [6] produces a spanner where shortest paths are at most 3 times longer than original.

2.3 Our Approach: Graph Spanners

We customize our solution based on the structural properties of large social graphs. More specifically, we consider two key characteristics, *dense connectivity* and *small-world property*, both of which have been observed in real social graphs [19, 28]. For example, typical average degree in social graphs is 60–100 [28], compared to 4–5 in geographic graphs [13]. Similarly, average distance in social graphs is 4–6 hops [28], compared to 100+ hops in geographic graphs.

These two characteristics make planar- and structure-driven algorithms inefficient, since they leverage the planar connectivity and backbone structure in geographic graphs. But given their dense connectivity and small average distance, social graphs lack a planar or backbone structure.

On the other hand, these properties of social graphs are ideal for spanner-based algorithms. Because of the dense connectivity, there are often many different shortest paths between two nodes. As a result, a sparse spanner is likely to preserve short paths between arbitrary node pairs, while greatly reducing the number of edges stored in the spanner. In comparison, landmark-based algorithms force all paths to go through one of the landmarks. For two close by nodes, this means local paths between them are ignored, thus producing significantly longer approximate paths.

3 Atlas Design

We describe Atlas, a system for approximating shortest paths on large social graphs. We start by outlining the main challenges and presenting an overview of the system, followed by details of the proposed algorithms.

3.1 Overview

While previous work on graph spanners has demonstrated the effectiveness of this approach on shortest path problems, they are impractical for our problem context. More specifically, prior work focused on general graphs, and algorithms produced graph spanners that approximate shortest paths either by a multiplicative factor of 3 [6] or by an additive factor of 2 [9, 10]. For social graphs with typically short path lengths of 4–6, these theoretical bounds are clearly unacceptable. To guarantee an additive factor of ≤ 2, the spanners must be $O(|V|^{3/2})$ in size, which is not a significant saving compared to the original graph. Thus, we seek to construct graph spanners especially customized for social graphs to produce accurate approximations of short and long paths.

Our insight in designing Atlas is to utilize a probabilistic approach to locating short paths. Instead of searching for a single graph spanner with particular properties, we instead propose to construct for every graph multiple spanners, each centered at a different root node. As shown in Figure 1, each spanner contains all nodes, but only a small fraction of edges of the original graph. The result is a group of compact spanners,

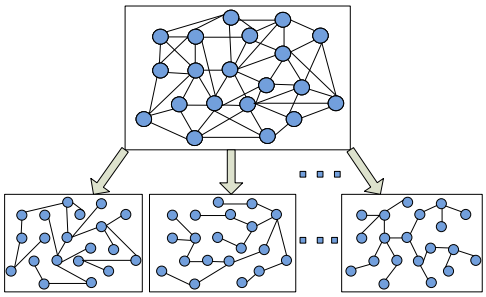


Fig. 1. Atlas constructs multiple spanners from the original graph. Each spanner contains all nodes in the original graph, but only a small fraction of the edges. To find a short path between two nodes, Atlas finds the shortest path between them on each spanner, then selects the shortest among the result paths.

each of which is likely to fit into the main memory of today's commodity servers. We resolve a shortest path query by searching on each graph spanner for a short path, then selecting the shortest path among the resulting paths.

We designed Atlas to scale extremely well across distributed computing resources, *e.g.* computing clusters. For example, running Atlas on a single machine means we load each of the (potentially large) graph spanners sequentially into memory, query it, then repeat. To minimize data transfer latency from secondary storage (*e.g.* disk storage) to main memory, a much more efficient approach would distribute each spanner onto a separate server, then parallelize queries across main memory queries at each server.

3.2 Choosing Spanners

Before describing Atlas's algorithm for constructing spanners, we first identify the types of spanners suitable for our problem context. We outline the desired properties of the spanners, and show that spanning trees based spanners fit perfectly to our requirements.

To support efficient shortest path query on large social graphs, we design Atlas spanners to satisfy the following three properties:

• *Small in size.* Each spanner should contain only a small fraction of edges of the original graph, for easy storage and processing.

• *Fast path computation.* Each spanner should display special structure thus computing shortest paths on it is much faster than running a generic algorithm (*e.g.* BFS).

• *Complementary coverage.* Ideally, spanners should cover complementary paths, so the shortest path between each node pair should be well approximated by at least one of the spanners.

A Case for Spanning Trees. The simplest structure satisfying the first two properties is a spanning tree graph. It contains the minimum number of edges to connect all the nodes in the graph. The shortest path between two nodes on a tree can be uniquely found by routing the nodes to their lowest common ancestor [5] using highly efficient algorithms [15]. To realize the third property, we need strategies for building multiple (complementary) trees that together approximate the original graph. In Atlas we experiment with several heuristics for constructing and growing trees.

We note that spanning trees are especially well suited for distributed computing. A single spanning tree too large to fit into memory can be further divided into separate subtrees and distributed across servers. For example, we can divide a tree into subtrees rooted at the children of the root. If both nodes in a query are on the same subtree, then it suffices to query only the subtree. Otherwise, the shortest path is constructed by routing both nodes to the root, *i.e.* by querying their individual subtrees separately.

3.3 Constructing Spanning Trees

To build spanning trees, we consider a simple construction algorithm as follows. To build a single tree, we first choose a root node from the graph. Starting from this root, we iteratively add edges, where each edge connects a new node (not on the current tree) to a node on the current tree. This process terminates when all nodes have been added.

The above algorithm requires two major design decisions: namely, how to choose the root for each tree, and how to grow a tree by choosing an edge in each iteration. Both decisions will affect the coverage of the resulting spanning trees. In the following, we propose several strategies in choosing the root and growing the tree.

Step I: Selecting Roots. Intuitively, for a single tree, selecting a root that is relatively central in the graph tends to minimize the tree depth, thus minimizing the average path length of the tree. Similar strategies have been shown to be effective in landmark selection problems [22]. There are several widely accepted metrics to measure a node's centrality, among which the degree centrality or node degree is the simplest. Alternative centrality metrics, such as closeness, betweenness and eigenvector centrality, are very difficult to compute [22] and thus not considered in our study.

To select roots for k spanning trees, we consider the following three strategies:

• *Top-k Centrality.* It chooses k nodes with the highest node degrees, seeking to minimize the depth of each tree. It does not explicitly make the trees complementary to each other.

• *Scattered Top-k Centrality.* It selects the node with the largest degree as the root for the first tree. For each subsequent tree, it selects the largest degree node among all nodes that are *at least* 2 *hops away* from previously chosen roots. Intuitively, two trees with very close roots will tend to overlap significantly in their covered paths. Thus this strategy tries to build more complementary trees by selecting roots with some minimal separation.

• *Random.* As a baseline strategy, this algorithm randomly selects k nodes as roots.

Step 2: Growing Trees. Given a root node, the remaining question is how to grow the trees, *i.e.* iteratively choosing edges to add them to our spanning tree. Our goals in this process are minimizing tree depth to reduce average node distance, and producing trees with complementary coverage.

When it comes to minimizing tree depth, it is easy to see that a BFS is an ideal candidate. At each step, BFS adds to the tree an edge that is attached to the node with the least depth. When there are multiple candidate edges corresponding to equal node depth, we consider two strategies to break the tie: *random tie-break* that chooses one candidate edge randomly, and *complementary tie-break* that chooses an edge that is least used by previous trees.

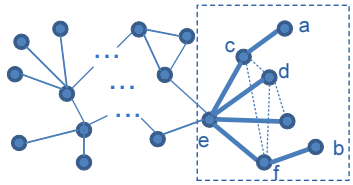


Fig. 2. An example showing the limitation of a BFS tree. The boxed area is a community-like subgraph connected to the main graph. Each BFS tree rooted from the main graph will include the bold edges but not the dotted edges. As a result, all trees will report a distance of 4 between a and b, despite the actual distance of 3.

On the other hand, while BFS trees tend to minimize average node distance, they can fail to cover some critical paths. We illustrate this artifact in Figure 2, where a community-like structure (the boxed area) is connected to the main graph via an "entrance" node e. In this case, any BFS tree rooted in the main graph will include the bold edges ($e \leftrightarrow c$, $e \leftrightarrow d$ and $e \leftrightarrow f$), but not the dotted edges ($c \leftrightarrow f$, $c \leftrightarrow d$ and $d \leftrightarrow f$). They will ignore the dotted edges, and produce suboptimal paths whenever the shortest path includes a dotted edge. It is easy to show that these artifacts apply to general graphs containing community-like structures with either single or multiple "entrances."

This artifact occurs because BFS trees tend to cover "global" edges that interconnect communities, but ignore "local" paths that connect nodes within a community. To overcome this limitation, we consider an edge selection strategy that optimizes for "local" paths. Specifically, in each step we prefer an edge that was chosen the least number of times by previous trees. We refer to this strategy as *Least-Covered-Edge-First*. In the above example, after adding $c \leftrightarrow e$ to build a global path, we add $c \leftrightarrow d$ and $c \leftrightarrow f$ to cover local paths.

In summary, we consider the following three strategies for growing spanning trees. We also illustrate and compare each strategy in Figure 3 using a simple example.

• *BFS with Random Tie-Break.* It always adds new edges that are attached to nodes with the smallest depth. When there are multiple qualified edges, it randomly chooses one. Figure 3 shows three trees. Because of the random tie-breaker, tree 2 and 3 are identical.

• *BFS with Complementary Tie-Break.* It follows the above strategy, but when breaking the tie, it chooses the least chosen edge when building existing trees. This aims to minimize the overlap among trees. Thus in Figure 3, trees 1–3 are quite different in the last level.

• *Least-Covered-Edge-First.* In each step, it chooses the edge that was least used by previous trees. This leads to the minimum overlap among trees.

3.4 Atlas's Complexity Analysis

Building Trees. The complexity of building each tree depends on the tree growing strategy. Traditional BFS's complexity is $O(|V| + |E|)$. For BFS with Random Tie-Break, upon visiting each node, we need to shuffle a subset of its children on the tree. Since shuffling can be done in linear time [18], the complexity is still $O(|V| + |E|)$.

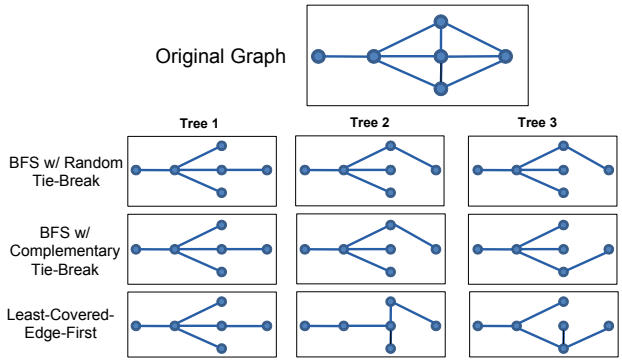


Fig. 3. An example showing 3 trees formed using 3 tree growing strategies. The BFS starts from the left-most node.

For BFS with Complementary Tie-Break, we need to sort this subset, resulting in a complexity of $O(|V|log|V|+|E|)$. For Least-Covered-Edge-First, we need to maintain the list of active edges to grow a tree. Implementing this list using a priority queue, we can build each tree within $O(|V|+|E|log|E|)$.

Finding Shortest Paths. Each tree can be stored using a dictionary, which maps each node to its parent. Given two query nodes, we first find the lowest common ancestor on the tree, which can be done in constant time using an auxiliary data structure [15]. We then route both nodes to the lowest common ancestor in $O(L)$ time, where L is the distance between the two nodes.

4 Handling Massive Graphs

For social graphs that cannot fit into a server's main memory, conventional approaches for building spanning trees are no longer feasible. This is because conventional BFS implementations require a large number of *random accesses* to the graph, each retrieving a given node's neighbors. This is extremely inefficient when the graph cannot fit in main memory, since random access queries will produce many disk seeks, each incurring latency several orders of magnitude greater than memory-bound reads.

Our proposal is to implement Atlas with support for *external memory* storage, by computing BFS trees while "streaming the graph" from disk to memory. We can produce a BFS tree in several rounds, where the graph is read once from disk in each round. Starting from the root node, in each round we sequentially scan through the whole graph's edges on the disk and construct one additional level of the tree. Specifically, in round $r \geq 0$, for each edge $a \leftrightarrow b$ that connects a on the $r-th$ level to an unvisited node b, we assign b to be a child of a. We store the growing tree in the main memory when streaming the graph on the disk. Note that this design only implements a fixed priority to break ties when growing the tree.

To implement the BFS random tie-break, we need to randomly assign a parent to a node on the next level when multiple nodes on the current level are connected to it on the graph. Specifically, if we encounter an edge $a \leftrightarrow b$ with a on the rth level and b already assigned to be a child of c on the rth level, we allow a probability that b's

parent is changed to a from c. The probability values can be chosen using the reservoir sampling [27]. To implement the BFS complementary tie-break, we tag each edge on the disk with the number of times that it is chosen in previous trees, and use this information to decide the parent assignment.

This implementation is highly memory-efficient. The only data stored in the main memory is the actual tree. We represent the tree using an array T, where $T[i]$ stores the parent node of i. Additionally, for each node i we record its level in the tree. Our largest social graph, Renren, only requires 0.3GBs of memory, compared to 21GB of memory for the standard implementation.

The time complexity of this implementation is $O(R \cdot |E|)$, where R is the number of rounds we need to stream the whole graph. It is clear that this number is equal to the depth of the tree, or the largest distance from the root to any node, which is around $8-12$ in today's online social networks. Each round costs $O|E|$ as we perform constant time operations for each scanned edge. For the Renren graph, it takes 5.8 hours to build a tree, slightly slower than the 1.6 hours for the in-memory implementation.

5 Experiments on Social Graphs

In this section, we verify Atlas using real social graphs collected from five of today's popular OSNs.

5.1 Social Graph Datasets

Our dataset includes data we have collected from Facebook [28] (largest OSN worldwide, 500+ million users) and RenRen [4] (largest OSN in China, 150+ million users). We also use data shared from external measurements on Orkut [3], LiveJournal [2] , and Flickr [1].

Facebook. Our dataset includes anonymized social graphs from 23 of Facebooks largest regional networks [28], measured between March and May 2008, totalling more than 10 million users and 940 million links. The availability of these graphs allows us to select a small set with enough diversity. For example, two representative graphs from the Norway and New York City regions (Table 4) display very different size and connectivity [28].

Renren. Launched in 2005, it is the largest OSN in China now with more than 150 million users [4]. It offers functionality very similar to Facebook. We collected in October 2009 a large social graph of roughly 43 million users and 1 billion edges. To the best of our knowledge, this is the largest dataset ever crawled from a single OSN.

Orkut. This social graph was collected by Mislove during October and November 2006 [7]. Since our work focuses on connected social graphs, we extract the largest connected subgraph from the raw data.

LiveJournal. It is a popular online community to publish blog, journal or diary [2]. The dataset used in this paper was crawled in December 2006 [7]. We convert all directed edges to undirected, and extract its largest connected component listed in Table 4, which includes about 98% users of the raw data.

Flickr. It is a popular website for photo sharing [1]. We use Flickr data collected in January 2007 [7], convert all directed edges to undirected, and extract its largest connected subgraph.

Social Graphs	Nodes	Edges
Renren	43,197,391	1,040,429,110
Orkut	3,072,440	117,185,083
LiveJournal	5,189,808	48,942,197
Flickr	1,715,255	15,555,042
Facebook: Norway	293,500	5,589,802
Facebook: New York City	377,619	3,616,816

Fig. 4. Six representative social graphs (with undirected edges) from five popular OSNs.

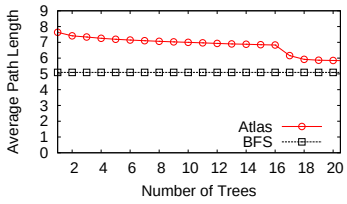


Fig. 5. Comparing the average path lengths produced by Atlas to their original distances. Atlas with $k = 20$ is able to approximate the average path length within 1-hop absolute error.

We focus on two performance metrics. First, we study the *accuracy in path length*, measured by the difference between the path length returned by Atlas to the exact length returned by BFS. Second, we study the *accuracy in path*, defined as the maximum overlap between the path returned by Atlas and any shortest path reported by BFS on the original social graph. During the process, we also compare different root selection and tree growing strategies. We observe consistent results for all the social graphs. We only show representative results from the Renren graph due to space limit.

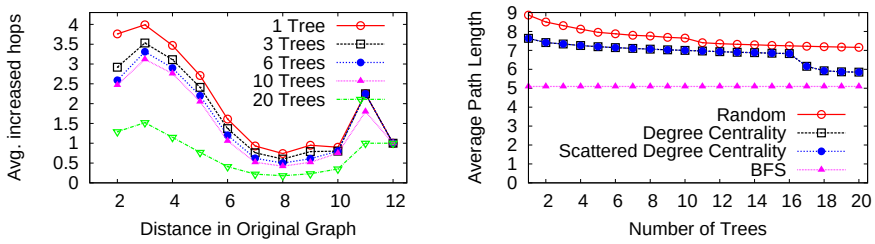
5.2 Accuracy in Path Length

We start from examining Atlas's accuracy in estimating shortest path length. We first examine Atlas with basic strategies and then examine the impact of root selection and tree growing strategies.

Atlas with Basic Strategies. We first evaluate Atlas by its basic configuration: using the Top-k Centrality to select roots and the BFS with Random Tie-Break to grow trees. For each social graph, we first build k spanning trees, randomly select 200,000 pairs of nodes on the graph, and calculate the shortest path length generated by Atlas and the original BFS based search. Atlas returns the path length result by querying each of the k trees separately and selecting the shortest path returned from these k trees.

Figure 5 compares the average path length answered by Atlas to that answered by BFS, as a function of the number of trees k, $k = 1...20$. We make two key observations. First, as k increases, the average path length produced by Atlas approaches the optimal value. k=10-20 is likely to be a "sweet point" since the improvement decreases quickly and becomes marginal when k exceeds beyond 10. Second, for all the graphs, Atlas with $k = 20$ is able to approximate the average path length within 1-hop absolute error.

To understand the accuracy of Atlas at individual path lengths, we classify the node pairs by their distance computed using BFS, and compute for each distance value the average difference in path length between Atlas and BFS. Figure 6(a) shows the results with $k = 1, 3, 6, 10, 20$. Interestingly, Atlas's accuracy in path length increases with the node path length, especially when the number of trees k is small. Even for $k = 20$, Atlas produces more accurate results for long paths. This coincides with our expectation in Section 3 and Figure 2 where Atlas with BFS Random Tie-Break focuses on covering "global" paths that interconnect communities but ignores "local" paths that connect



(a) Impact of original node distance (b) Comparing root selection strategies

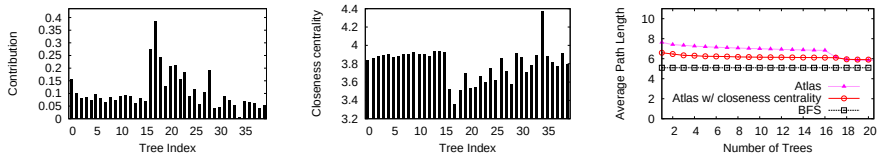
Fig. 6. a) Impact of original node distance to the accuracy of Atlas, in terms of the average amount of increases in distance measured by Atlas over the original value. Atlas's accuracy in path length increases with the node path length. Even for $k = 20$, Atlas produces more accurate results for long paths. b) Comparing different root selection strategies. *Random* performs poorly because it fails to minimize the tree depth. The two centrality-based strategies are consistently similar.

nodes within a community. This also motivates us to examine other tree growing strategies that seek to cover local paths.

Comparing Root Selection Strategies. Figure 6(b) compares the three root selection strategies: *Top-k Centrality*, *Scattered Top-k Centrality* and *Random*, while using the BFS with Random Tie-Break to grow trees. As expected, *Random* performs poorly compared to the centrality-based strategies because it fails to minimize the tree depth and thus suffers from larger average path length. On the other hand, the two centrality-based strategies perform consistently similar to each other. A similar observation was found in landmark selection [22]. From the graph statistics, we infer that this is because many nodes with largest degree centrality are separated by more than 2 hops. For Renren, the two strategies select exactly the same set of roots.

Comparing Tree Growing Strategies. We also examine Atlas with different tree growing strategies, *Random Tie-break*, *Complementary Tie-break* and *Least-Covered-Node-First*. We omit the detailed results due to space limit. Our results show that in terms of average path length, the default Random Tie-break strategy with $k = 20$ always outperforms the other two. Complementary Tie-break only improves the accuracy slightly by 0.1 hop when the number of trees is small ($k < 5$). Least-Covered-Node-First only improves the accuracy when the node distance is 1. This shows that covering global paths is crucial for providing accurate shortest path estimation. Atlas's multiple trees, together, help to cover local paths ignored by individual trees. Overall, Random Tie-break is still the best strategy among the tree.

Contribution of Individual Trees. Since Atlas uses multiple spanning trees, we are interested in understanding the contribution of individual trees to the overall shortest path computation. To do so, we measure the contribution of a tree by the probability that it outputs the minimum path length among all k trees. Note that for a given query, there might be multiple trees that achieve the same minimum length, and we count the query at all these trees. We plot in Figure 7(a) the result for Renren using the basic strategies. We see that tree 16–18 make much higher contribution than others. From the traces, we found that the contribution of each tree is highly correlated with the closeness centrality



(a) Contribution of individual trees (b) Closeness centrality of tree roots (c) Using closeness centrality in root selection

Fig. 7. Contribution of individual trees to overall shortest path computation, using the Renren social graph. (a) The contribution defines the probability a tree outputs the minimum path length among all k trees. It varies across trees. (b) A tree's contribution is heavily correlated with its root's closeness centrality value, motivating us to use closeness for root selection (although highly complex to compute). (c) Yet using closeness centrality does not help when k is large.

of its root (Figure 7(b)). A tree whose root has a lower closeness value produces a higher contribution. This is not surprising because closeness is an accurate metric on a node's connectivity with the rest of the graph.

Because computing closeness is highly difficult [22], Atlas uses node degree to select tree roots. To understand the impact of such practical decision, we show in Figure 7(c) the average path length result of Atlas, Atlas using closeness centrality and the optimal BFS approach. Interestingly, when $k \geq 16$, using closeness centrality does not lead to any gain. This shows that having multiple trees compensates the imperfect root selection strategy, and proves the efficiency and effectiveness of our Atlas design.

5.3 Accuracy in Shortest Path

A unique advantage of Atlas is that it can not only estimate the shortest path length but also record all nodes along the path, *i.e.* produce the actual path. Information on the actual path is useful to many applications. For example, users can monitor how their information is delivered by those media nodes that are on the paths. In this case, if there is less deviation for a shortest path (produced by Atlas) compared to the real one (produced by BFS), users can reduce/avoid such monitoring. In other words, the path produced by Atlas, if displays minimum deviation from the actual path, will be more reliable to users and thus more accurate.

We use the metric of maximum overlap of shortest path produced by Atlas with the original path to evaluate Atlas's path accuracy. Because using BFS, there could also be several shortest paths between any two nodes. We compute all possible shortest paths using BFS and also use Atlas to produce a shortest path. Then, we compare the Atlas shortest path to the multiple BFS shortest paths and count the number of overlapped nodes between Atlas path and each BFS shortest path. We define the maximum matching path as the BFS path with the maximum number of overlapped nodes . We then compute the deviation of shortest path as the number of nodes that are in the maximum matching path but not in Atlas path, and use this to quantify the path accuracy.

In our implementation, we randomly select 20 nodes as source nodes in each graph and compute 100 random paths for each node, producing 2000 samples. To avoid bias, we do not use any root node or nodes with large degree. We plot the distribution of the deviation of shortest path in Table 1. We see that at least 30% Atlas paths do not miss

# of nodes missed by Atlas path	Frequency of occurrence	
	New York	Norway
0	0.341	0.314
1	0.072	0.058
2	0.2395	0.3235
3	0.29	0.2895
4	0.0555	0.015
5	0.002	0

Table 1. Deviation of Atlas shortest path

Social Graph	Atlas Tree construction	Atlas query on 20 trees	BFS query
Renren	96min/tree	0.3ms/query	86.5s/query
Orkut	8min/tree	0.3ms/query	12.0s/query
LiveJournal	2.5min/tree	0.3ms/query	19.8s/query
Flickr	0.8min/tree	0.3ms/query	6.2s/query

Table 2. Atlas runtime performance in terms of constructing trees and answering query.

any node and very few paths miss more than 3 nodes. By averaging the overlap to the path length, Atlas can cover 60.6% nodes in Norway graph and 64.6% nodes in NY graph. This shows Atlas's confidence in approximating actual shortest paths.

5.4 Runtime

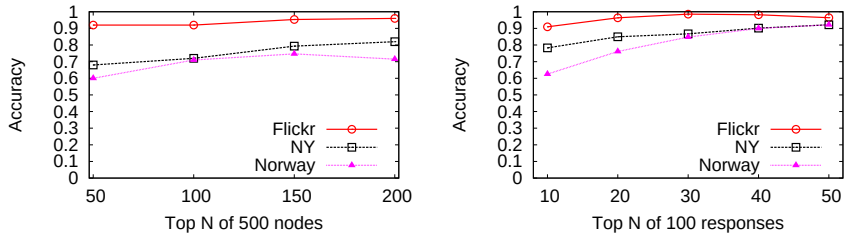
Having examined Atlas's complexity bound, we now examine its practical runtime performance. Table 2 lists Atlas's runtime metrics for constructing spanning trees and processing shortest path queries. This was measured on a commodity server of 2.4GHz, dual core with 32GB of RAM. We implement Atlas tree construction in C and shortest path query in Python. Our approach scales to the extremely large Renren graph with 43 million nodes and 1 billion edges, taking a reasonable 1.6 hours to build one tree. To compute a shortest path, Atlas searches through all $k = 20$ trees sequentially (no parallelism). It takes $<$60 seconds to process 200,000 queries, translating into 0.3ms per query even for the Renren graph. This is orders of magnitude smaller compared to the conventional shortest path query time.

5.5 Application Benchmarks

Having examined Atlas using shortest path analysis, we now examine the efficacy of using Atlas in graph analysis application. Our goal is to examine whether the approximate shortest path produced by Atlas can be used to produce accurate application results. We consider two graph analysis applications, *node centrality computation* and *ranked social search*. We run these applications on three social graphs: Facebook Norway, Facebook New York, and Flickr.

Node Centrality Computation. Closeness centrality is the average path length from one node to all the other nodes in the graph. It can be used as a critical metric to estimate how fast information traverses the whole graph. Intuitively, a node of smaller average path length can be considered as higher centrality. That means information can spread through the graph in a shorter time.

Since Atlas can estimate shortest path length between any two nodes accurately, we can estimate centrality precisely. To evaluate the accuracy of centrality computed by Atlas, we use the results computed by BFS as the ground truth. We randomly select 500 nodes from each graph and compute centrality for each node using Atlas and BFS. Specifically, we rank the nodes by their average path length to all other nodes and select the top N nodes in both methods. We then compute the accuracy as the amount of overlap between Atlas and BFS results, normalized by N.



(a) Node Centrality Computation (b) Ranked Social Search

Fig. 8. The accuracy of using Atlas in graph analysis applications.

Figure 8 (a) plots the results for different N values. The accuracy of Atlas increases gradually with N because nodes with larger centrality are being included and that Atlas produces less error in estimating longer paths (shown in Figure 6 (a)). Overall our system promises at least 60% accuracy for Norway, 70% for New York and 90% for Flickr. The difference between these graphs is because both Flickr and New York display larger average shortest path length.

Ranked Social Search. Users in OSNs often choose search results based on the relationship between themselves to responders. In other words, they prefer the results provided by closer nodes, and order results by their distance to the responders. This is called social-based search rank. To implement such shortest path length based ranking, we randomly select 500 nodes from each graph. Each node will receive 100 random responses from 100 distinct nodes. Each node ranks the responses based on its shortest path length to the responder. Again we choose top N responses and count the overlap between Atlas and BFS results.

Figure 8(b) shows that the accuracy of Atlas's selection increases with N. Even when selecting only top 10% responses ($N = 10$), Atlas provides 60% accuracy for Norway, 80% for New York and 90% for Flickr. Similar to the case of centrality computation, the accuracy for Norway is lower because its average path length is smaller.

6 Handling Dynamic Graphs

Online social networks display dynamic characteristics, including users adding/removing accounts, changing relationship status, and evolution of user interactions [28]. These dynamics make the corresponding social graphs time-varying. In this section, we discuss how Atlas updates its spanning trees to handle dynamic graphs efficiently.

To help motivate and validate our design, we have collected several dynamic social graphs. One particular example is that using our Facebook measurement, we obtained 35 daily snapshot social graphs of the San Francisco regional network during October 2008, where the number of users in the graph grew from 160K to 170K. This sequence of social graphs captures the dynamics of users joining (and leaving) Facebook and updating their friend connections. Table 3 lists the degree of variation in both nodes and edges across the 35 daily snapshots, indicating a slow but steady growth.

6.1 Atlas for Dynamic Graphs

Intuitively, the simplest solution to address graph dynamics is to update Atlas's trees whenever there is a change in the social graph. Given the scale of today's social graphs,

Statistics	Nodes		Edges	
	Added	Removed	Added	Removed
Mean	0.36%	0.22%	0.61%	0.39%
Standard deviation	0.0025	0.0012	0.0044	0.0025

Table 3. The mean and standard deviation of proportion of nodes and edges added/removed per-day across our 35 San Francisco snapshot social graphs.

updating all the trees in real-time is intractable. For example, it takes 1.6 hours on commodity servers to build a single tree for Renren, thus 32 hours for 20 trees. This means that to support large dynamic graphs, we can only update the spanning trees periodically and/or incrementally.

With this in mind, we introduce two new mechanisms on top of Atlas's original design for dynamic graphs:

• *Incremental Tree Update.* Atlas periodically updates the spanning trees to reflect changes in the social graph, *e.g.* addition and removal of graph nodes and edges. To reduce the overhead, Atlas only updates a small subset of trees rather than all the trees.

• *Path Validation.* With periodic and incremental update, some trees become obsolete and could return invalid paths that contain deleted nodes and edges. We add to Atlas a verification process to verify paths returned from these trees, and discard invalid paths. The process is simple and efficient since it only involves several hash table lookups. In rare cases that none of the trees return valid paths, Atlas answers the query by running a generic algorithm on the current social graph.

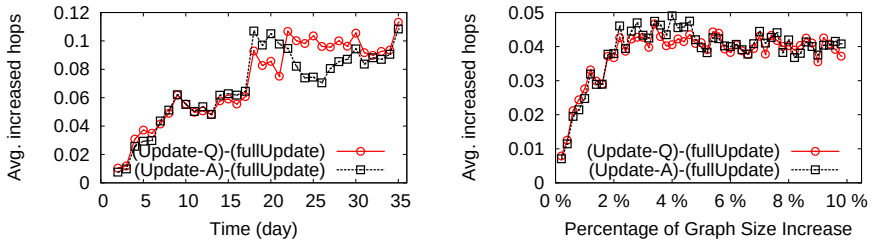
In the following, we focus on dynamic tree update because path validation is straightforward to design.

Incremental Tree Update. The motivation behind our design comes from the actual observed dynamic patterns, shown in Table 3. Between each graph update, the number of changed nodes/edges is relatively small compared to the overall graph scale ($\approx$0.5%). Thus it is highly likely that slightly obsolete trees are still good for the majority of shortest path queries and thus do not need to be replaced immediately.

Motivated by the above observation, we propose in Atlas to build only one new tree per update period or when detecting sufficient changes on the graph, and use it to replace an existing tree. The new tree covers newly added nodes (and edges) without significant computation overhead. The design question here is "which existing tree should we replace?" Intuitively, the tree to replace should be the most "obsolete" in terms of answering shortest path queries. For this we introduce two metrics on the level of "obsoleteness":

• *Query-Accuracy.* We evaluate each tree by its performance in answering queries. In this case, two types of events will affect the obsoleteness of a tree. First, the tree does not contain one of the query nodes because it is recently added to the social graph. Second, the tree returns an invalid path that contains nodes and/or edges that no longer exist. We score each tree by the total occurrence of both events, and the tree with the highest score is the most obsolete one and will be replaced.

• *Age or Tree Creation Time.* This metric measures the level of obsoleteness by the time it is created, where we always replace the oldest tree in the set.



(a) Facebook San Francisco Snapshots (b) Mode-generated Dynamic Social Graphs

Fig. 9. Comparing Atlas with incremental tree update (Update-Q, Update-A) to Atlas with full tree rebuild (fullUpdate), in terms of the difference in average path length. Incremental update performs closely to full update. There is no significant difference between query-accuracy and age based incremental update.

6.2 Experiment on Dynamic Social Graphs

We performed detailed experiments to evaluate Atlas's shortest path query with dynamic social graphs, and to compare the above two tree updating strategies. In particular, we examine three options in Atlas tree update:
i) *Update-Q*, the proposed incremental tree update mechanism using the Query-performance metric;
ii) *Update-A*, incremental tree update using the Age metric;
iii) *fullUpdate*, which completely rebuilds all $k = 20$ trees.

Graph Data Set. We use both measured and model-generated dynamic graphs. We use synthetic graphs because they allow us to examine the impact of dynamics in a longer period than the measured ones and to control the frequency of graph/tree update.

• *Measured Dynamic Social Graphs.* This is the set of 35 daily snapshots of the San Francisco regional network discussed in the above. In this case, the social graph is updated once per day, upon which Atlas also updates its tree(s).

• *Model-generated Dynamic Social Graphs.* We use a modified Nearest Neighbor(NN) model [23] to generate dynamic graphs. This model has been shown to statistically approximate real social graphs with high confidence [23]. We use an iterative process to build dynamic graphs: first construct a synthetic graph that closely approximates the Facebook New York City graph, then follow the model to gradually grow the graph by 0.5% of the original size per iteration. We produce a sequence of 100 graphs where the last graph is 50% larger than the first graph. Different from real dynamic graphs, these synthetic graphs do not capture node or edge removal. In this case the social graph is updated once per 0.5% growth, so are Atlas trees.

For both datasets, we build/update Atlas trees using the Top-k Centrality strategy to choose roots and the BFS with Random Tie-Break to grow trees. We choose this configuration based on the evaluation result in Section 5.

Results: Incremental Update vs. Full Update. We first compare Atlas with incremental and full update by the average path length returned for $200,000$ random queries performed on each graph snapshot. To highlight the difference we plot in Figure 9 the

difference in average path length between Update-Q and fullUpdate, and Update-A and fullUpdate, for the three dynamic graphs.

For both graphs, the mean difference between progress and full update is roughly 0.1 hop. We also found that full update is within 0.6 hop from the optimal solution that uses BFS on the actual social graph. This result demonstrates the effectiveness of Atlas's incremental tree update. For both scenarios, we see that the error initially increases with time and then flattens and even decreases. The initial increase is due to the edge effect since we build 20 fresh trees at time 0. The flattening (and decreasing) trend indicates that the proposed incremental tree update can quickly stabilize over time.

We also observe that there is no significant difference between Update-Q and Update-A. This result is as expected but also encouraging. Because the amount of change between snapshots is still relatively small given the overall graph scale, the majority path queries can be answered by the slightly obsolete trees. As the trees get updated one by one, the oldest tree will likely produce the worst query results, and should be updated.

Results: Impact of Individual Events. For a detailed look at the impact of graph dynamics, we categorize the dynamics into four events and analyze their impact.

• *Removing an edge.* Removing an edge ϕ affects a tree $\mathbb{T}$ only if $\mathbb{T}$ contains ϕ and thus could return invalid paths that contain ϕ. We measure the impact by ϕ's weight on the tree, defined as the size of the subtree rooted at ϕ, which relies on ϕ to connect with other nodes in the graph. Intuitively, removing an edge with larger weight leads to heavier impact.

• *Removing a node.* Removing a node μ also removes all of its edges. This event affects an existing tree $\mathbb{T}$ only if $\mathbb{T}$ contains μ. Similarly, we measure the impact by its weight, defined by the size of the subtree rooted at node μ.

• *Adding an edge.* Adding an edge $\phi = (u \leftrightarrow v)$ to a graph can produce large impact on a tree $\mathbb{T}$ if nodes u and v were widely separated on the tree. Thus we measure its direct impact on Atlas's accuracy by the distance between u and v observed on $\mathbb{T}$. Another related question is whether adding ϕ to the graph makes $\mathbb{T}$ no longer a BFS tree on the new graph. If so, this will introduce significant changes to the tree when it gets updated. To examine this we record the difference in u and v's depth on $\mathbb{T}$. When the difference is no more than 1, $\mathbb{T}$ is still a BFS tree on the new graph and adding ϕ will not introduce changes to $\mathbb{T}$.

• *Adding a node.* It produces no impact unless followed by adding an edge, which is already covered.

Figure 10 plots the impact measures using the San Francisco dynamic social graphs. We make several observations. First, of all the edges removed, 89% have no impact on any tree, *i.e.* they were not on any tree (results not shown for brevity). We plot in Figure 10(a) the weight of edges that were on at least one tree. More than 90% of them have a weight of 5 or less, meaning that their removal only affects a small subtree of 5 nodes. Similar observations were found on node removal (Figure 10(b)). This implies that the majority of node/edge removed were on the boundary of the graph. They do not affect the core graph structure, and have minimal impact on the spanning tree structure.

Figure 10(c)-(d) examine the impact of adding an edge $u \leftrightarrow v$ by the distance of u and v on a tree and the difference of their depths on the tree. Figure 10(c) shows that edges were uniformly added among nodes that were close-by and well-separated. This

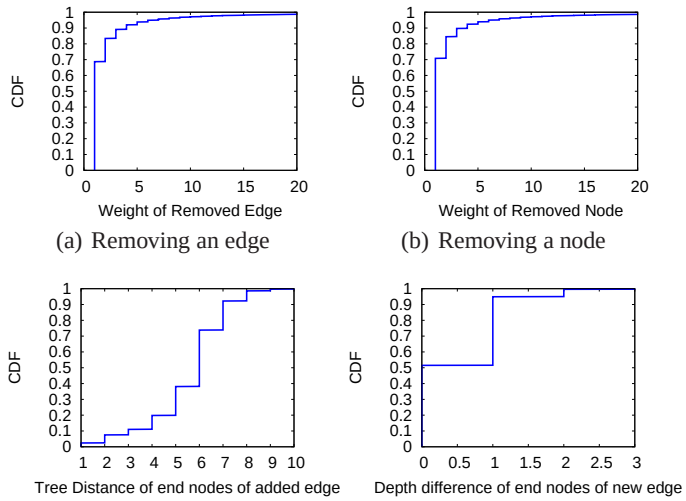


Fig. 10. The impact of each dynamic event on Atlas trees, using the San Francisco snapshots as the dynamic social graph. (a) Most of edges removed have a small weight and affect only a small number (5) of nodes. (b) Most nodes removed have a small weight and affect a minimum number of queries. (c) Edges were added between nodes with different separations. This triggers the majority of errors. (d) For 95% of added edges, the two end nodes' tree depths differs by 1 or 0. This means that the tree is still a BFS on the new graph.

is the key source for Atlas's path estimation error. Figure 10(d) shows that for 95% of new edges, the depths of u and v only differ by 1. This means that with high probability adding a new edge still makes a tree a BFS tree on the new graph and thus has no effect on the tree. Overall, our detailed analysis confirms that each individual event has minimum impact on the tree structure.

7 Conclusion

This paper describes *Atlas*, a novel approach to scalably approximate shortest paths between graph nodes using multiple spanning trees. Atlas easily distributes across distributed machine clusters to handle massive graphs, and produces short paths that closely match the ideal shortest path. We demonstrate its scalability and effectiveness using large graphs from multiple social networks, the largest of which includes 43 million nodes and 1 billion edges. Using snapshots of graph dynamics, we also show that Atlas can be incrementally updated over time to handle dynamics graphs. Experiments show that Atlas can help real graph applications scale to massive graphs. With the rapid growth of OSNs, we believe Atlas will be a critical component for graph analysis in the future. We are already in discussions to deploy Atlas at a large social network provider.

References

1. Flickr. http://www.flickr.com.
2. Livejournal. http://www.livejournal.com.

3. Orkut. http://www.orkut.com.
4. Renren.com. http://www.renren.com.
5. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. On finding lowest common ancestors in trees. In *Proc. of STOC* (1973).
6. ALTHÖFER, I., ET AL. On sparse spanners of weighted graphs. *Discrete Comput. Geom. 9*, 1 (1993), 81–100.
7. A.MISLOVE, ET AL. Measurement and analysis of online social networks. In *Proc. of IMC* (2007).
8. DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.
9. DOR, D., HALPERIN, S., AND ZWICK, U. All-pairs almost shortest paths. *SIAM J. Comput. 29*, 5 (2000), 1740–1759.
10. ELKIN, M., AND PELEG, D. $(1 + \epsilon, \beta)$-spanner constructions for general graphs. In *Proc. of STOC* (2001).
11. GOLDBERG, A., KAPLAN, H., AND WERNECK, R. Reach for A*: Efficient point-to-point shortest path algorithms. In *ALENEX* (2006).
12. GOLDBERG, A. V., AND HARRELSON, C. Computing the shortest path: A* search meets graph theory. In *Proc. of SODA* (Philadelphia, PA, USA, 2005), pp. 156–165.
13. GOLDBERG, A. V., AND WERNECK, R. F. F. Computing point-to-point shortest paths from external memory. In *ALENEX/ANALCO* (2005), pp. 26–40.
14. GUTMAN, R. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *ALENEX* (2004).
15. HAREL, D., AND TARJAN, R. E. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput. 13*, 2 (1984), 338–355.
16. HART, P. E., NILSSON, N. J., AND RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. *SIGART Bull.*, 37 (1972), 28–29.
17. KLEINBERG, J., SLIVKINS, A., AND WEXLER, T. Triangulation and embedding using small sets of beacons. In *Proc. of FOCS* (2004).
18. KNUTH, D. E. *The Art of Computer Programming, Vol. 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley, 1973.
19. LESKOVEC, J., KLEINBERG, J., AND FALOUTSOS, C. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proc. of ACM KDD* (2005).
20. MÖHRING, R. H., ET AL. Partitioning graphs to speed up Dijkstra's algorithm. In *WEA* (2005), pp. 189–202.
21. NICHOLSON, T. A. J. Finding the shortest route between two points in a network. *Computer J.*, 9 (1966), 275–280.
22. POTAMIAS, M., ET AL. Fast shortest path distance estimation in large networks. In *Proc. of CIKM* (2009).
23. SALA, A., ET AL. Measurement-calibrated graph models for social network experiments. In *Proc. of WWW* (2010).
24. SANDERS, P., AND SCHULTES, D. Highway hierarchies hasten exact shortest path queries. In *Proceedings 17th European Symposium on Algorithms (ESA)* (2005).
25. SONG, H. H., ET AL. Scalable proximity estimation and link prediction in online social networks. In *Proc. of IMC* (2009).
26. SWAMYNATHAN, G., ET AL. Do social networks improve e-commerce: a study on social marketplaces. In *Proc. of SIGCOMM WOSN* (August 2008).
27. VITTER, J. Random sampling with a reservoir. *ACM Trans. Math. Softw. 11*, 37–57.
28. WILSON, C., ET AL. User interactions in social networks and their implications. In *Proc. of EUROSYS* (2009).
29. ZHAO, X., ET AL. Orion: Shortest path estimation for large social graphs. In *Proc. of WOSN* (June 2010).