Discovering Critical Edge Sequences in E-Commerce Catalogs

Kaushik Dutta* Anindya Datta* Debra VanderMeer* Krithi Ramamritham†

April 22, 2001

Abstract

Web sites allow the collection of vast amounts of *navigational data* – clickstreams of user traversals through the site. These massive data stores offer the tantalizing possibility of uncovering interesting patterns within the dataset. For e-businesses, always looking for an edge in the hyper-competitive online marketplace, this possibility is of particular interest. Of significant particular interest to e-businesses is the discovery of *Critical Edge Sequences* (CES), which denote frequently traversed subpaths in the catalog. CESs can be used to improve site performance and site management, increase the effectiveness of advertising on the site, and gather additional knowledge of customer interest patterns on the site.

Using traditional graph-based and web mining strategies to find CESs could turn out to be expensive in both space and time. In this paper, we propose a number of approximate algorithms to compute the most popular paths bewteen node pairs in a catalog, which are then used to discover CESs. Our methods are both *space-efficient* and *accurate*, providing a vast reduction in the storage requirement with a minimum impact on accuracy. These algorithms, which can be executed off-line in batch mode, are also practical with respect to running time. As variants of single-source shortest-path, they run in log linear time.

*Georgia Institute of Technology
†University of Massachusetts and IIT, Bombay

1 Introduction

Web sites allow the collection of vast amounts of *navigational data* – clickstreams of user traversals through the site. These massive data stores offer the tantalizing possibility of uncovering interesting patterns within the dataset. For e-businesses, always looking for an edge in the hyper-competitive online marketplace, this possibility is of particular interest.

Consider, for example, *Books.com*, a fictional online bookseller. *Books.com* tracks visitors as they navigate through the site, and subsequently analyzes this information. Assume that such analysis reveals the following two pieces of information: (a) if a visitor, in a session, navigates through *Orchid Gardening* and *Military History*, he tends to eventually make a purchase of a *Music History* book; and (b) the shorter the time a user takes to traverse the site between *Orchid Gardening* and *Military History*, the higher the likelihood of an eventual purchase. The knowledge of such a relationship represents a potentially valuable opportunity to *Books.com*, if it can appropriately recognize the popular traversed paths between *Orchid Gardening* and *Music History*. For instance, if such popular paths could be identified, then they could be analyzed to determine the length of the traversal time from *Orchid Gardening* to *Music History*. If such analysis revealed that the most popular paths were time-consuming to traverse, this would indicate possible areas of improvement on the site. Finding popular paths can be useful in several other scenarios as well. For instance, such paths can be used in the prediction of traversal behavior for caching, or to provide preferential resource allocation for heavily-traversed sequences on the site. Analysis of these paths can also reveal valuable interest patterns that can aid marketing and advertising efforts on the site, e.g., for delivering advertisements, e-coupons, or running promotional campaigns.

To motivate the specific problem addressed in this paper, we return to the example of *Books.com*. Suppose that an analysis of the most frequently traversed paths between *Orchid Gardening* and *Military History* found two distinct paths:

1. *Orchid Gardening* → *Home Improvement* → **Anthropology** → **Medicine** → **Mythology** → *Buddhist Literature* → *Military History*

2. *Orchid Gardening* → *Computer Science* → **Anthropology** → **Medicine** → **Mythology** → *Woodworking* → *Military History*

While it is interesting, and indeed necessary, to find these most-traversed paths, what is of practical value to the site is the fact that the most popular paths between *Orchid Gardening* and *Military History* share a common subsequence, i.e., *Anthropology* → *Medicine* → *Mythology*. Clearly, this subsequence contains some information of interest, i.e., that users frequently traverse the subsequence *Anthropology* → *Medicine* → *Mythology* when navigating from *Orchid Gardening* to *Military History* (regardless of which exact path they may have followed). We call such edge sequences *Critical Edge Sequences* (CESs). We are currently working with a major clicks-and-mortar retailer where finding such CESs is a major concern. This concern prompted a collaborative project; our work with the site motivated the work reported in this paper. Experiences with this real-life case are reported throughout the paper.

The quest for "interesting" information from accumulated data has always been of interest to researchers. Such explorations have been carried out both from a systems perspective (e.g., the generation of frequency histograms for use in database query processing algorithms) and from an applications perspective (e.g., the mining of association rules from a large dataset). This paper is more in the latter category than the former; we design algorithms to extract useful information from large volumes of data. In particular, we are interested in finding interesting traversal patterns through web sites. This paper's main contribution is in this area.

The focus of this paper is the identification of critical traversal patterns at a web site. In particular, given a hierarchical navigation structure (i.e., a catalog), and a set of traversal records over this structure (i.e., user navigation sessions over the catalog), we are interested in discovering critical edge sequences (CES) between a pair of nodes. Such CES are discovered by a two step process: (a) the k most traversed paths are identified between the two nodes; then (b) the CESs, i.e., the commonly occurring subsequences are discovered within these k paths (recall the previous example). Part (a) is essentially a path finding problem in a graph context, whereas the part (b) is a sequential pattern mining problem with a few twists. It turns out that part (a) is a more difficult problem to solve that part (b) – as a result the major thrust in this paper is to explore part (a). In particular, we will demonstrate that it is impractical, indeed, in most cases impossible, to get exact solutions for part (a) for catalogs encountered in practice. Accordingly, we will propose efficient *approximate* methods to solve part (a). Since the output from part (a) is approximate, a concern is the quality of the solutions from part (b). We will demonstrate that, in spite of the potential for inexactness in the first part of this problem, the impact on accuracy on the CESs discovered by the second part is minimal. Moreover, we will provide guidelines regarding how to maximize obtainable accuracy given a set of input parameters.

The remainder of this paper is organized as follows. Section 2 describes our approach and the contributions of this paper. Section 3 discusses related work. Section 4 describes a method of finding CESs based on full session information. Section 5 presents approximate CES-finding methods. Section 6 presents the results of our performance tests. Section 7 concludes the paper.

2 Preliminaries and Contributions

In this section, we describe our overall approach to finding CESs on an e-commerce site. We first describe the underlying model assumed throughout the paper, then move on to discuss the issues underlying CES discovery and summarize the contributions of this paper.

2.1 Site Model

In this paper, we assume a graph-based representation of a site, much like the model underlying [10], and concepts of user-site interaction found in [8]. A site is a graph $G = (V, E)$ where each vertex $v \in V$ is a *node* in the e-commerce site, and represents a particular concept, perhaps a product or a category of products, and each edge $e \in E$ represents the links on the site, i.e., the possible navigation between nodes on the site.

In our model, users interact with the site by navigating across the nodes of the site using the links. A particular customer's traversal through the site can be described as a *clickstream*, i.e., a sequence of nodeids identifying which vertices the user visited, and the order in which he visited them.

Based on ideas in [8], our model separates the (abstract) site content representation from the (concrete) page delivered to a user. Essentially, we assume that content associated with multiple nodes in the site graph can be delivered to a user on a single page, and allows limitless combinations of site content to be generated.Practically speaking, this type of model maps well to a recent phenomenon in Web content delivery – dynamic page generation. Here, the pages served to a user are generated on the fly, based on the site's business logic and the user's state at run-time.

Often, a site's business logic utilizes a user's previous navigation patterns to build new pages for a user. For example, consider the personalization features available on amazon.com's site. Here, the site offers a link to the "Page You Made", which features products a user recently viewed, as well as recommended products (similar to those recently viewed). Clearly, amazon.com's site uses a model similar to that described in [8], where content for a page may be drawn from multiple nodes in the site graph, and the business logic that determines the content of a page is dependent on the user's previous traversal through the site.

2.2 The Problem

We present a set of algorithms which answer the question "What are the critical edge sequences (CESs) on the paths leading from N_a to N_b?". To answer this question, one may adopt a two-step approach: (a) find the most traversed paths leading from N_a to N_b; and (b) analyze the output of step (a) to determine which edge sequences are critical. The first step is crucial in the process, as (1) it is difficult to perform in a timely and accurate manner, and (2) it directly affects the accuracy of step (b). Thus, finding the most popular paths is a major thrust of this paper.

While an abundance of path-finding algorithms exist in graph theory, the traditional input to these algorithms is a graph, in which edges weights are *independent* of one another. However, we are dealing with a problem domain in which the probability of traversing an edge, i.e., the edge weight, is dependent on the prior traversal path up to the edge. As we elaborate later, we are unaware of algorithms that address this type of problem.

As mentioned above, our algorithms need as input (a) a site catalog and (b) a record of user traversals. Since the primary source for (b) is a Web server log, it can easily be seen that (b) is a significant space consumer, especially if log data must be archived over time. For example, consider the clicks-and-mortar site where our methods are being tested. On average, this site experiences 10 million unique user sessions in a day.[1] Each user requests, on average, 10 pages on the site. A log entry on this site consists of some 300 bytes of data, and each user click, on average, generates 10 log entries (since each embedded object, such as an image, requested on the site is logged separately). In this case, the size of the log file generated *per day* is $10,000,000 \times 10 \times 300 \times 10$ bytes or 0.3 TB. Over the course of a single year, the data volume grows to almost 109 TB. Clearly, storing and querying such large data volumes is prohibitively expensive. At larger sites, such as Yahoo, the numbers are much greater. Developing algorithms that can operate on such very large data sets is a challenging issue.

2.3 Our Approach

As mentioned earlier, the problem of finding CESs between two nodes on a site can be divided into two steps: (1) the task of finding the k most-traversed paths; and (2) the task of analyzing those k paths to find CESs.

The first task, that of finding the k most-traversed paths on a site, is much more complex and difficult than the second. Thus, we focus much of our effort in this work on efficiently and accurately finding these paths.

The second task, that of analyzing the output of a k-most-traversed-path-finding algorithm to find CESs is much more straightforward. CESs can be derived from subsequences of the k paths in many different ways. The simplest way to do this is to extract all common subsequences across the k paths. Typically, not all of the CESs are of interest; rather they are filtered based on some application-dependent criteria. For example, some applications may be interested in subsequences of a particular length, while others may impose a minimum probability threshold or frequency of occurrence in the k paths. Given a set of k most-traversed paths, the application of these criteria is not difficult.

2.4 Contributions

We propose three algorithms to handle the first task. They are described in Sections 4 and 5. The first of these is an exhaustive algorithm which is impractical for all but very small catalogs. This motivates the need for the development of approximate algorithms, the category to which our second and third algorithms belong. In the context of large data sizes encountered in practice, these algorithms possess a few noteworthy properties:

Property 1: Small size requirement. Our algorithms take as input a *smaller, aggregated* representation of user traversal information, which greatly reduces storage requirements. For example, in one of our live-data tests, aggregation reduced the storage requirement for a set of traversal data from approximately order of terabytes to 24 MB. While the commonly held notion that storage is cheap is indeed valid, reducing storage requirements offers a number of important benefits.

First, reducing overall storage requirements reduces storage maintenance costs. While purchase costs for storage run at about $1 per megabyte, the cost of maintaining each megabyte of data stored is approximately $7 *per year* [14]. As the volume of data stored grows, so does the annual cost of storing it.

Second, and more importantly, storage size reduction will minimize physical space requirements. This is an important consideration for sites, especially in the context of the current trend to outsource hosting (to infrastructure providers, such as Exodus). In this scenario, the cost of hosting a site is often tied directly to the physical space it occupies at the hosting facility, i.e., hosting a site that requires two racks to accommodate its data needs will cost significantly more than a site requiring only one rack. As more and more e-commerce sites outsource the hosting of their sites, optimizing rack space will become a serious concern – and one of the most significant consumers of rack space real estate is data storage.

Property 2: Low time complexity. Our algorithms are also practical, with low time complexity. Our methods are variants of Dijkstra's Single-Source Shortest-Path algorithm [7], without significant increases

[1] Many sites experience a much higher user load. Consider the example of Yahoo, which serves more than 600 million pages per day. (see http://docs.yahoo.com/docs/pr.1q00pr.html)

in time complexity; the average running-time is still *O(n log n)*. Since this type of analysis can be executed in batch mode off-line, log linear complexity is clearly practical. In real terms, the running time of our longest-running test required only 15 minutes on 3 GB of data. Interestingly, a similar test using an optimal (exact) method on 3 GB of unaggregated data (which represents a much smaller number of user traversals) consumed 20 hours.

Property 3: Accurate, even though approximate. It can easily be inferred, based on the discussion in (1), that our algorithms are, by necessity, *approximate* – generating exact results requires storing and analyzing all accumulated traversal information, which, based on the example above, is infeasible. Still, the algorithms provide high accuracy, quantified using a set of interesting metrics adapted from the well-known IR notions of *precision* and *recall*.

In summary, we propose a set space-efficient, time-efficient, and accurate methods for finding CESs on a site. They take as input a smaller, aggregated representation of user traversal data than standard Web logs. We have tested our methods on both synthetic as well as extensive real data.

3 Related Work

Having laid out the general structure of the problem, our solution approach, and the contributions of our work, we now proceed to place our work in the context of existing literature in each of the two subproblems identified above, i.e., finding the most popular paths between two nodes, and deriving critical edge sequences from those paths.

3.1 Finding the k Most Traversed Paths Between Two Nodes

Path-finding algorithms are a staple of graph-theoretic literature. These algorithms, such as Dijkstra's Single-Source Shortest-Path (SSSP) [7], appear similar in nature to the the problem of finding the most popular paths between a node pair on an e-commerce catalog – both problems involve the need to identify paths through a graph, based on some criterion. However, the criterion for ranking paths is different in the two problems – SSSP finds the shortest paths through the site, based on a set of edge weights, while we are interested in finding heavily-traversed sequences of edges in a graph.

The fundamental difference between these two problems is that of dependency. The edge weights in the SSSP scenario are assumed to be independent of one another, i.e., the weight of some edge is not dependent on the weight of any other edge in the graph. This is a general property of path-finding algorithms – consider algorithms for finding minimum spanning trees and maximum flow, for example, where the weight of one edge is independent of the weights of other edges. In the CES-finding problem, however the probability of traversing some edge $\langle N_i, N_j \rangle$ is dependent on the traversal path leading up to N_i, e.g., the probability of traversing $\langle N_i, N_j \rangle$ may be greater if the user arrives at N_i from some node N_a, than if the user arrives at N_i from some other node N_b. Accordingly, the methods proposed in this paper draw from existing path-finding literature, but include their own means of handling the dependencies among edge probabilities.

Aside from the dependency issue discussed above, another feature of the problem domain needs to be mentioned. On a web site with moderate to heavy traffic, the storage requirement for maintaining user traversal data can grow to consume terabytes, or even petabytes, of storage space over the course of few months (we will demonstrate this in short order). Clearly, the cost of maintaining and analyzing a data store this large is prohibitively high. While the type of analysis that we are proposing can be done in batch mode off line, it is clearly desirable for such analyses to consume minutes or hours rather than days or weeks. We will demonstrate shortly that it is impractical, indeed even impossible, to exactly identify the most popular paths on a site, purely because of the storage and time complexities associated with such a task. To address the issue of input data size, we *aggregate* the input data, rather than storing the full set of user traversal information. Thus, out of necessity, our algorithms are *approximate*. As we will show in this paper, *our methods provide a vast reduction in the storage size required and time consumed, but with minimum impact on the accuracy of the results.*

Now we move on to the second issue, i.e., that of actually identifying the CESs from the output of the first step.

3.2 Discovering Critical Edge Sequences from a set of Identified Popular Paths

Primary related work falls within the area of *web usage mining* (e.g., [15, 12, 6]). Much work on mining algorithms for web usage mining (e.g., [15, 17]) is based on fundamental work in data mining in general [1, 2, 11, 5] and sequential mining in particular [3, 16]. An important difference between the CES discovery problem and the traditional sequential pattern mining (SPM) problem discussed in the above references is the issue of *contiguity*. More specifically, SPM algorithms consider an identified pattern as valid provided the order of items in the discovered pattern is identical to the order in the original input to the SPM algorithm (of course, SPM requires other conditions for validity, such as confidence and support, as well). In our case however, in addition to simple ordering, a valid pattern *must also correspond to some exact contiguous subsequence in the original input.* Clearly, however, an SPM algorithm (such as [3]) can be modified to address this constraint.

The work reported in [13] is also relevant. Here, the authors mine longest repeating subsequences (LRSs), a subset of k^{th}-order Markov models, for the purpose of predicting user navigation on a Web site.

While there is a fair amount of existing work that can be adapted to discovering CESs from the paths output by the first part of our approach for practical values of k (≤ 10), the time burden of performing this task (i.e., common substring extraction from a small set of input strings) is negligible – our experiments consumed, for $k = 10$, time in the order of single digit milliseconds. Thus, it turns out that discovering the k most traversed paths between specified node pairs is the "hard" part of the CES discovery problem. Since our proposed methods are approximate, the second part assumes special significance as well, as the discovered CESs are not guranteed to be accurate. We therefore evaluate the quality of the CESs discovered from the paths output from the first step..

Before embarking on a discussion of our algorithms, we present a table of notation used in the remainder of the paper in Table 1.

Symbol	Meaning
$\langle N_1, ..., N_i \rangle$	Sequence of i nodeids, denoting $i-1$ edge traversals
$G = (V, E)$	graph of V vertices and E edges
k	number of number of popular paths to be found
l	maximum user session length (in clicks)
f	average fanout of a site graph
d	depth of bound on conditional probability
$\mathcal{P}^{\langle N_i, N_j \rangle}$	Probability that a user will traverse the edge $\langle N_i, N_j \rangle$, given that he has arrived at N_i
$\mathcal{P}^{\langle N_i, N_j \rangle}_{\langle N_1, N_2, ..., N_d, N_i \rangle}$	Probability that a user will traverse the edge $\langle N_i, N_j \rangle$, given that he has traversed the path $\langle N_1, N_2, ..., N_d, N_i \rangle$
$\rho^{i,j}$	single probability value that represents a vector of probabilities
P	the set of k most-traversed paths returned
m	the number of subsequences in P
ϵ	the proportion of CESs found in error
δ	the proportion of CESs missed

Table 1: Table of Notation

4 Exact Solution

In this section, we describe a method for exactly calculating the k most traversed paths from node N_a to N_b by storing full session information in a relational table and querying over it to find the most traversed paths. [2] We call this approach the *Exact Solution Method* (ESM).

We identify a user session as a sequence of nodes. For example, in Figure 1, user U_b visits N_5, N_6, N_2, N_3, N_4 and then exits. Thus, the session for U_b's visit can be represented as $\langle N_5, N_6, N_2, N_3, N_4 \rangle$.

[2] Clearly, there are many alternative ways of representing and analyzing the data. For example, we considered using a hash-based approach, where traversals would be stored based on a hash of the path (we discarded the idea because it is less amenable to analysis than the approach described here). What is important here is not the chosen structure and particular analysis method, but the fact that any analysis method would require (a) storage of the full data set from the logs and (b) the same basic underlying logic to analyze the data.

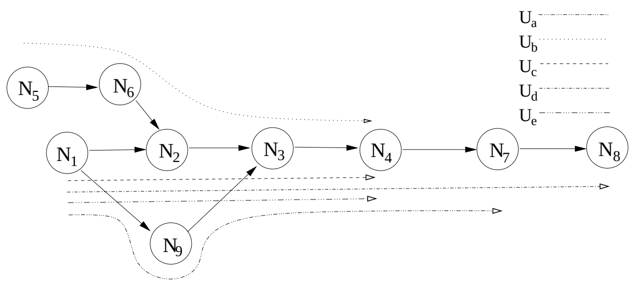

Figure 1: Session Graph

We can represent these user sessions in a *Session Table*, a simple unnormalized table structure, shown in Table 2. Here, each distinct l-length session is represented as $l-1$ tuples, where each tuple represents a hop in the session. For example, the session traversed by both U_c and U_e, i.e., $\langle N_1, N_2, N_3, N_4 \rangle$ appears in Table 2 identified as session_id 2.

The *session_id* field identifies a distinct session (i.e., sequence of traversed nodes), where multiple user-visits can result in the same session. The *count* field records the number of times a distinct session has been traversed. The *hop* field uniquely identifies each sequential edge traversal within a session. Together, the *session_id* and the *hop* form the primary key of the *Session Table*. The *from_node* and *to_node* fields record the source and destination nodes of an edge traversal. To illustrate the use of the *Session Table*, Table 2

session_id	*count*	*hop*	*from_node*	*to_node*
1	1	1	N_5	N_6
1	1	2	N_6	N_2
1	1	3	N_2	N_3
1	1	4	N_3	N_4
2	2	1	N_1	N_2
2	2	2	N_2	N_3
2	2	3	N_3	N_4
3	1	1	N_1	N_2
3	1	2	N_2	N_3
3	1	3	N_3	N_4
3	1	4	N_4	N_7
3	1	5	N_7	N_8
4	1	1	N_1	N_9
4	1	2	N_9	N_3
4	1	3	N_3	N_4
4	1	4	N_4	N_7

Table 2: An Example of Session Table Contents

shows the contents of the *Session Table* for the user traversals shown in Figure 1. Consider, for example, the sequence of nodes followed by users U_b and U_d, namely $\langle N_1, N_2, N_3, N_4 \rangle$, identified as *session_id* =2. Since two users have followed this distinct session, *count* =2. Since this session consists of four nodes, there are three *hops* representing it in the *Session Table*, one for each edge traversal: *from_node* $= N_1$, *to_node* $= N_2$; *from_node* $= N_2$, *to_node* $= N_3$; and *from_node* $= N_3$, *to_node* $= N_4$.

Clearly, the *Session Table* explicitly stores all session information relevant to the exact most-traversed path analysis. Finding the k most traversed paths from node N_a to node N_b from this table is a three-step process. This process is perhaps best explained using an example. Consider the session data in Table 2. Suppose we wished to analyze this dataset for the k most-traversed paths from N_1 to N_4, where $k = 1$, i.e., finding single most traversed path from N_1 to N_4. We would do this by following the four steps described below.

Step I: First, we find all sessions that include a path from node N_a to node N_b. Intuitively, this can be done by selecting sessions with *from_node* $= N_a$ at some *hop* $= x$ and *to_node* $= N_b$ at some *hop* $= y$, where $x < y$. This can be expressed in the following SQL statement:

```
select distinct X.session_id, X.count from Session X
where X.from_node = N_a
and X.hop ≤ ( select Y.hop from Session Y
              where Y.to_node = N_b
              and X.session_id = Y.session_id)
```

Here, the nested select statement finds the sessions containing an edge traversal with *to_node* $= N_b$. The outer select statement finds the sessions (from the set returned by the inner select statement) with *from_node* $= N_a$ and *hop* value less than the *hop* of the *to_node* corresponding to N_b. The output of the SQL statement is a list of *session_ids* that include a path from N_a to N_b, along with each session's corresponding *count*.

In our example, the nested select statement would return *session_ids* 2, 3 and 4. Since $k = 1$, we are interested in the single most traversed path. Clearly, based on the data in Table 2, the expected output is $\langle N_1, N_2, N_3, N_4 \rangle$.

Step II: Second, we find the actual session paths that contain the path of interest. In order to do this, we can use the following SQL statement, where *sid* is a *session_id* returned from the SQL statement in the first step:

```
select R.from_node, R.to_node from Session R
where R.session_id = sid
and R.hop ≥ ( select S.hop from Session S
              where S.session_id = sid and S.from_node = N_a)
and R.hop ≤ ( select T.hop from Session T
              where T.session_id = sid and T.to_node=N_b)
order by hop
```

Here, the first nested select statement finds the *hop* x of the *from_node*$= N_a$. The second nested select statement finds the *hop* y of the *to_node*$= N_b$. The outer select statement returns the *from_node* and *to_node* for all *hops* of *session_id*=*sid* where $x \leq y$. Returning to our example based on Table 2, running this SQL query with *sid*=2 would return a set of tuples representing the following sequence of edges: $\langle N_2, N_3, N_4 \rangle$. Running the SQL query for *sid*=3 would return the same sequence of edges. For *sid*=4, the SQL query would return $\langle N_1, N_9, N_4 \rangle$.

Step III: Third, we sum the counts of matching subpaths to determine the number of traversals over a distinct traversed path leading from N_a to N_b. Using the data in Table 2, we would sum the counts for *session_ids* 2 and 3, since they both contain the same subpath from N_1 to N_4, i.e., $\langle N_1, N_3, N_4 \rangle$, and associate that summed count with the subpath. The count for $\langle N_1, N_9, N_3, N_4 \rangle$, i.e., *session_id* 4 remains 1, since no subpath matches it.

Step IV: Finally, we order the subpaths by count, and take the subpaths with the k largest counts as our final answer. In the example, since $k = 1$, we are looking for the single most traversed subpath, which is $\langle N_1, N_3, N_4 \rangle$.

Note that the *Session Table* is unnormalized for the purposes of illustrating how to find the k most traversed paths from it. Normalizing the *Session Table* and developing the corresponding SQL statements is quite straightforward; rather than going into the details of this normalization, we analyze the main problem with the above approach; namely, the storage required.

In the worst case, each user session is unique among all sessions, necessitating the storage of all user sessions. Here, if we assume a maximum session length l and fanout f, the storage complexity is $O(|V| \times f^l)$, i.e., for every possible starting node, there are f^l possible sessions to be stored.

4.1 Case Study

To illustrate the storage problem, we consider the 150,000-product e-commerce site that served as the basis of our tests. The log contains user traces, each of which has an average session length $l = 8$. The site graph has fanout $f = 10$.[3] Each of the five values in the *Session Table* (shown in Table 2) is a 4-byte integer. In this case, then, the storage size required for full information (in the worst case) is $150,000 \times 10^8 \times 5 \times 4 = 3 \times 10^{15}$ bytes, i.e., on the order of petabytes. This represents the worst case storage scenario, in which each user traverses a unique path through the site.

Performing ESM analysis on such a large table is a difficult and time-consuming task, virtually impossible to perform on a daily or even weekly basis. To illustrate this, we ran the nested queries used in ESM on very small sets of session data (up 3 GB), and report the running times in Figure 2. We ran these tests using an Oracle8*i* database on a Sun EnterpriseR Server with four 450MHz CPUs, 1 GB RAM, and 120 GB of hard disk space.

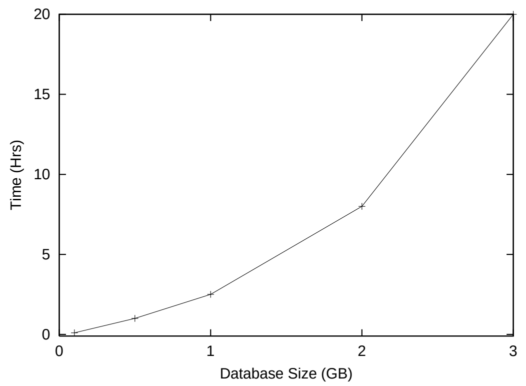

Figure 2: Cost of ESM

Clearly, the time required for this type of analysis grows exponentially with the size of the underlying database. Since ESM required 20 hours to complete its analysis of 3 GB of data, one can only guess at the number of years required to run this type of analysis on a database in the low-terabyte range.

Given the difficulty in dealing with such large data sizes, we conclude that ESM is not well suited for finding the k most traversed paths for large online shopping sites. Therefore, we consider approximate methods, which require fewer resources.

5 Approximate Methods

In this section, we present a class of algorithms which provide approximate solutions. Rather than storing entire session information (as in ESM), here we store *edge-specific* information. In particular, we store sequences of edges of a configurable depth d (where each d-length sequence represents a sequence of edges actually traversed by at least one user on the site), and a count of the number of times the sequence was traversed. It is then possible to associate a probability value with each edge $\langle N_i, N_j \rangle \in E$ that indicates the probability of a user U traversing $\langle N_i, N_j \rangle$, given the d-length path traversed by U immediately prior to navigating the edge $\langle N_i, N_j \rangle$.

As the reader will recall from our discussion in Section 1, the fundamental difference between the problem of finding CESs in a graph and the SSSP problem is the issue of dependence – SSSP assumes independence of edge weights. In our case, the independence assumption is not valid. In fact, each edge weight in the CES-finding problem is dependent on the path traversed to reach the edge. *The notion of depth introduced here essentially places a bound on the dependence of an edge weight on previous path traversal.* Clearly, if traversal data prior to the most recent d-depth clicks is unavailable, the system cannot consider dependence on prior traversals, essentially placing a bound on the dependence on prior traversals.

The value ascribed to d, then, denotes the degree to which an edge weight, i.e., the probability of traversing the edge, depends on previous path traversal. In the base case, where $d = 0$, we are assuming that edge weights are independent of previous path traversal. For some $d > 0$, say for example $d = 3$,

[3] Note that we consider the edges of the site graph to consist of *navigable* edges, i.e., hyperlinks. Thus, even at the leaf level of the catalog hierarchy, the vertices would have outgoing edges.

we place the bound of dependence at 3 edges, i.e., the probability of traversing an edge is assumed to be dependent on only the three most recently traversed edges; all previous edge traversals are assumed to be independent. At the margin, $d = l$, where l is the maximum user session length. In this case, we are assuming complete dependence of an edge weight on previous path traversal – this is the case considered by ESM.

Clearly, the cost and accuracy of the result is a function of d. We are interested in exploring the cost/accuracy tradeoff of varying d. In particular, we address two questions: (a) *to what degree does this independence assumption affect accuracy?* (b) *how does accuracy scale, i.e., does increasing d provide marginal returns?*

In order to explore these questions further, we present two algorithms for finding highly traversed paths, given varying dependence assumptions. First, we present the *Zero-Depth Algorithm*, which considers the case where $d = 0$, i.e., the complete-independence case. Second, we present the *Configurable-Depth Algorithm*, where $d > 0$, and the d represents the bound of dependence of an edge's probability on previous path traversals.

5.1 Zero-Depth Algorithm (ZDA)

ZDA is a variant of Dijkstra's Single-Source Shortest-Path algorithm. The similarities between the two underlying problems are clear: both consider paths through a graph, and both seek to find the "best" paths – the shortest-path problem considers "best" to be the least sum of edge-weights, while "best" for ZDA is actually the most frequently traversed. However, the problem domains of SSSP and the most traversed paths differ in several major ways. Consequently, the algorithms also differ. In particular, they differ in (a) the meaning of the edge weights; (b) the function used to compare paths to one another; (c) the function used to determine the path weight; and (d) the number of solutions generated. We now describe the ZDA algorithm in more detail.

5.1.1 Details of the ZDA Algorithm

As we know of no algorithm for single-pair shortest-path problem that runs asymptotically faster than the best single-source algorithms in the worst case, we modify Dijkstra's Single-Source Shortest-Path algorithm [9]. Algorithm 1 shows our modification of SSSP to find the k most traversed paths between a pair of nodes. The modifications map directly to the differences in the underlying problems noted above. We first describe the three functions of the algorithm itself, then consider each of these four differences in turn.

The ZDA Algorithm The main function of Algorithm 1 is BEST-PATHS, which takes as input a graph, i.e., a set of vertices and edges, a start vertex s, k (the number of most-traversed paths to return), and a set of edge probabilities $\mathcal{P}^{u,v} \forall (u, v) \in G$. Throughout execution, Algorithm 1 maintains k path probabilities $P[v][k]$ for each edge (u, v). These path probability values are maintained in descending order, i.e., $\P[v][0] > P[v][1] > P[v][2]$, and so on. Corresponding to these path probabilities, k predecessor vertices $\pi[v][k]$ are also maintained for each vertex v. At the end of execution, the P and π matrices hold the k result paths.

The first step in the execution of BEST-PATHS is a call to INITIALIZE-SINGLE-SOURCE (line 1). Here, all $k \times |V|$ path probabilities are set to 0, with the exception of the first path probability of the start vertex ($P[s][0]$), which is set to 1. All $k \times |V|$ predecessor vertices are set to NIL.

After initializing all path probabilities and predecessor vertices, all $k \times |V|$ elements of P are inserted into a queue Q, which is ordered by descending value of $P[v][k]$ (line 2). Next, each item in Q is returned in turn, using the EXTRACT-MAX function, which simply returns the item in the Q with the largest value (lines 3-5).

For each $P[v][k]$ element returned by EXTRACT-MAX, we *relax* each outgoing edge of v to determine whether or not the element's path probability should be updated, based on the path from the start node to v at that point in the execution of the algorithm (lines 6-7 of BEST-PATHS). For this, we use the RELAX function.

In the first step RELAX (line 1), a temporary variable *temp* is assigned the product of the path probability up to the input vertex u and the probability of the edge (u, v). Then (in lines 3-6), the value

Algorithm 1 Zero-Depth Algorithm

Input:

$G \rightarrow$ traversal graph.
$V \rightarrow$ vertex set of traversal graph G.
$s \rightarrow$ source in the single source k most-traversed path problem.
$Adj[v] \rightarrow$ Adjacency list of vertices $v \in V$.
$k \rightarrow$ number of most traversed paths to find for each pair of nodes.
$\mathcal{P}^{u,v} \rightarrow$ edge probabilities for edge $(u,v) \in E$ where $u \in V$ and $v \in V$. $\mathcal{P}^{u,v} \geq 0$ for each edge $(u,v) \in E$ otherwise $\mathcal{P}^{u,v} = 0$.
$P[v][k] \rightarrow$ for each vertex $v \in V$ we maintain an ordered k length array (start index 1) which are the probabilities of k most traversed paths from s to v. This ordered array is implemented as an ordered list.
$\pi[v][k] \rightarrow$ a k-length list for each vertex, indicating the k most traversed paths after the algorithm has been run
$Q \rightarrow$ a queue of all $\P[v][k]$ elements, ordered by descending probability.

BEST-PATHS(G, $\mathcal{P}^{u,v}$, s, k, $\mathcal{P}^{u,v} \forall (u,v) \in G$)
  INITIALIZE-SINGLE-SOURCE(G, s)
  $Q \leftarrow P[v][i] : \forall v \in V$ and $1 \leq i \geq k$
  while $Q \neq \Phi$ **do**
    $P[u][j] \leftarrow$ EXTRACT-MAX(Q)
    $Q \leftarrow Q - P[u][j]$
    for each $v \in Adj[u]$ **do**
      RELAX(u, v, j, $\mathcal{P}^{u,v}$)

INITIALIZE-SINGLE-SOURCE(G, s)
  for each $v \in V$ **do**
    for each $i, 1 \geq i \leq k$ **do**
      $P[v][i] \leftarrow 0$
      $\pi[v][i] \leftarrow NIL$
  $P[s][0] \leftarrow 1$

RELAX(u, v, j, $\mathcal{P}^{u,v}$)
  $temp \leftarrow P[u][j] \times \mathcal{P}^{u,v}$
  for each $i, 1 \geq i \leq k$ **do**
    if $max(P[v][i], temp) == temp$ **then**
      insert $temp$ before $P[v][i]$ (i.e., i^{th} position) in ordered list $P[v]$
      set $\pi[v][i] \leftarrow \pi[u][j] + u$
      delete last element from ordered list $P[v]$

in *temp* is compared to the k path probability values for the edge (u,v), and, if *temp* is found to be larger than any of the current path probability values, then *temp* is inserted into the set of k path probabilities (in the appropriate position), the corresponding predecessor vertex is set, and the lowest of the original k path probabilities is removed.

Having described the BEST-PATHS algorithm, we now move on to discuss the major differences between BEST-PATHS and SSSP.

Meaning of edge weights in ZDA. In the case of SSSP, edge weights denote some measure of distance between nodes. We are not interested in a distance between nodes in finding the most-traversed paths; rather, we wish to associate a probability of traversal with each edge in the graph. Consequently, we now define the probability of traversing an edge for ZDA.

In ZDA, (i.e., the case where $d = 0$), the probability of traversing an edge $\langle N_i, N_j \rangle \in E$ is conditioned only on a user's arrival at N_i. Accordingly, we define the probability of traversing an edge $\langle N_i, N_j \rangle \in E$ (given that the user is at node N_i) as follows:

$$\mathcal{P}^{N_i,N_j} = \mathcal{P}\{choose\ link\ to\ N_j | user\ is\ at\ N_i\} = \frac{\#\ of\ traversals\ of\ \langle N_i, N_j \rangle}{\#\ of\ traversals\ through\ N_i}$$

These probabilities can be easily computed and updated periodically by tracking traversals over the site. The logged session data represents a *traversal subgraph* of the actual web site graph, where the subgraph contains only the vertices and edges actually traversed by users. In this way, we associate a probability value (as described above) with each edge in the traversal subgraph. The ZDA algorithm is then applied on this graph, using the probabilities associated with each edge as edge weights.

Path Comparison Function. In SSSP, the goal is to find minimum-distance paths between nodes; thus, a minimization function is used for path comparison. In CES analysis, however, we wish to maximize over edge weights to find the highest possible probability of traversing a path. Thus, in Algorithm 1, line 3 of Function RELAX, we replace SSSP's minimization function with a maximization function.

Path Weight Function. In the SSSP case, the length of a path is simply a summation of the edge weights on the path. Summation clearly will not work in CES analysis, where edge weights are probability values. Thus, we have developed a simple function for determining the weight of a path for CES. In particular, given that ZDA assumes independence of edge probabilities, we multiply the edge probabilities of a path to approximate the traversal probability of a path.

Given a traversal sequence $\langle N_i, N_j, N_k \rangle$, we can consider the edge probability for $\langle N_j, N_k \rangle$ to be independent of the edge probability for $\langle N_i, N_k \rangle$. Thus, the path probability for $\langle N_i, N_j, N_k \rangle$ can be approximated as follows:

$$\mathcal{P}\langle N_i, N_j, N_k \rangle \approx \mathcal{P}(\langle N_i, N_j \rangle \cap \langle N_j, N_k \rangle) = \mathcal{P}^{\langle N_i, N_j \rangle} \times \mathcal{P}^{\langle N_j, N_k \rangle}$$

We can generalize this to express the path probability $\mathcal{P}^{\langle N_1, N_2, .., N_k \rangle}$ for path $\langle N_1, N_2, ..., N_k \rangle$ in terms of edge probabilities $\mathcal{P}^{\langle N_i, N_j \rangle}$, $\langle N_i, N_j \rangle \in E$, where E is the set of edges, as follows:

$$\mathcal{P}^{\langle N_1, N_2, ..., N_k \rangle} = \prod_{i=1}^{k-1} \mathcal{P}^{\langle N_i, N_{i+1} \rangle}$$

This modification can be seen in Algorithm 1, in line 1 of Function RELAX, where we replace the summation in SSSP with a multiplication operation.

Number of Solutions Generated. SSSP generates a single result path for any pair of vertices in the graph. In CES analysis, however, we are interested in finding k solutions, where $k >= 1$. Thus, we associate k probability values (and k corresponding predecessor nodes) with each vertex in the graph. This can be seen in the definition of P, the data structure for path weights the input section of Algorithm 1. In SSSP, $P[v]$ stores a single path weight and $\pi[v]$ stores a single "best" predecessor node for each vertex in the graph. In CES, we store k possible probability values and predecessor nodes for each vertex, resulting in much larger data structures, i.e., $P[v][k]$ (for probability values) and $\pi[v][k]$ (for the corresponding predecessor nodes). The probability values for a particular edge leading to vertex v are maintained in *decreasing order*, i.e., the highest probability value is stored in $P[v][0]$, the next-highest in $P[v][1]$, and so on.

The changes in the data structures for storing path weights and predecessor nodes result in several other modifications to SSSP. We first discuss the changes to other data structures, and then the changes to each function in the algorithm.

In terms of data structures, the size of the priority queue Q is expanded from $P[v]$ elements in SSSP to $P[v][k]$ elements in CES in order to allow consideration of each vertex in the graph k times (thus generating the k best path weights from the start node to vertex v for k different paths).

The changes to the data structures require that changes be made to all three functions of Algorithm 1 as well. We discuss the changes to each function in turn.

In line 2 of BEST-PATHS, Q is initialized to hold all of the $k \times |V|$ elements corresponding to the needed $k \times |V|$ probability values in $P[v][k]$. Lines 3-4 extract each of these $k \times |V|$ elements from Q in order of decreasing probability.

In lines 1-4 of INITIALIZE-SINGLE-SOURCE, the values in the matrices $P[v][k]$ and $\pi[v][k]$ are initialized to 0 and *NIL*, respectively. Line 5 of INITIALIZE-SINGLE-SOURCE sets the first probability value for the start node, i.e., $P[s][0]$ to 1 (this will be the first node extracted from Q).

In RELAX, each of the k probability values for the input edge must be relaxed, where the corresponding SSSP version of the RELAX function requires relaxing the single distance value associated with the edge. The **for** loop corresponding to this requirement can be seen in line 2 of RELAX. Lines 4-6 of RELAX maintain the ordering of path probability values for a particular edge.

5.1.2 ZDA Correctness

Due to space considerations, we present an informal proof of correctness, rather than a formal one. Dijkstra's SSSP algorithm is known to be correct as an instance of a semiring (R, MIN, $+$, $+\infty$, 0), where R is the set of nonnegative reals including $+\infty$ [4]. Similarly, ZDA maps to a semiring ([0,1], MAX, $\times$, 0, 1) for $k = 1$ (i.e., the case where ZDA searches for a single path).

For $k > 1$, the algorithm's correctness is based on the process for finding and storing the k most traversed paths. In BEST-PATHS, the storage structure for the k most traversed paths is a sorted array; when a path is found with a higher traversal probability than one of the paths in the k-length array, the path with the lowest traversal probability is removed, and the new path is added to the array in sorted order. In this way, at each step of BEST-PATHS, the k-length array contains the k-most traversed paths thus far. With the termination of execution of the algorithm, the array contains the k-most traversed paths.

5.1.3 ZDA Time Complexity

In terms of time complexity, we first describe the complexity of the modules within ZDA, and then the complexity of the overall algorithm.

INITIALIZE-SINGLE-SOURCE visits each node once and executes a **while** loop k times for each node, resulting in a $O(k \times |V|)$ time complexity for the module.

Line 2 of RELAX contains a loop that executes k times. Each operation within this loop executes at most once, indicating an overall time complexity of $O(k)$ for RELAX.

The main function, BEST-PATHS, first calls INITIALIZE-SINGLE-SOURCE. After initialization, the priority queue Q (implemented as a Fibonacci heap) is loaded. In particular, each vertex is inserted k times. Inserting a single value into a Fibonacci heap requires time logarithmic in the number of values to be inserted into the heap. Thus, loading each vertex k times into the the priority queue requires $O((k \times |V|)\ log\ (k \times |V|))$ time.

Once the queue has been loaded, the while loop in line 3 of BEST-PATHS runs. Here, vertices are removed one at a time via the EXTRACT-MAX function, which removes the vertex with the greatest probability from the queue. Each call to EXTRACT-MAX for a Fibonacci heap requires $O(log\ (k \times |V|))$ time. Each item in the heap is removed once, resulting in an overall cost for line 4 of BEST-PATHS of $O((k \times |V|)\ log\ (k \times |V|))$.

For each vertex u returned by EXTRACT-MAX, the outgoing edges of u are considered by the RELAX function. Since each call to RELAX requires $O(k)$ time, and RELAX is called $k \times |E|$ times, the overall complexity of line 7 is $O(k^2 \times |E|)$.

Thus, the total time complexity of BEST-PATHS will be the sum of the cost of initializing the data structures (via INITIALIZE-SINGLE-SOURCE) and the cost of loading each vertex into the priority queue k times and the cost of the removing the each vertex from the priority queue k times and the cost of relaxing each edge k times. Thus, the overall time complexity for the ZDA algorithm is: $O(k \times |V|) + O((k \times |V|)\ log\ (k \times |V|)) + O((k \times |V|)\ log\ (k \times |V|)) + O(k^2 \times |E|)$. Clearly, the log linear terms dominate. Thus, the overall complexity is $O(k|V|\ log\ (|V|))$.

5.1.4 ZDA Space Complexity

The ZDA method requires much less storage than the ESM – for each edge in the web site graph, we need only store the number of users who have traversed it. Clearly, this requires $O(|V| \times f)$ storage, as compared to $O(|V| \times f^l)$ for ESM (for maximum fanout f and session length l). For example, consider the e-commerce site that served as the basis of our tests (discussed at the end of Section 4). For an average fanout of 10, we need only store, in the worst case, $150,000 \times 10 = 1,500,000$ edge probabilities for ZDA. For each probability, we need only store two nodeids (i.e., from and to) and a probability of traversing the edge. If each nodeid is a 4-byte integer, and the probability value is a float (8 bytes), then the storage required is $1,500,000 \times 16$ bytes, or 24 MB – far less storage than the petabyte storage requirement of ESM. Note that ZDA *aggregates* traversal results; only this aggregated information must be maintained. Once log file traversal data has been incorporated in these structures, the logs are no longer needed.

5.1.5 Applying ZDA to Real Data

To give a sense of ZDA's running time on real data, we ran ZDA on user traversal from the e-commerce site that served as the basis for our tests. The site provided two highly-correlated products from their product catalog for use as *from* and *to* nodes, and provided log data representing approximately 20 million user visits to the site. We ran the ZDA algorithm, using these inputs, for $k = 3$. The algorithm completed running in 9 seconds on the same hardware and software described in the Real-life Case Study in Section 4. Recall that running ESM on the same set of log data required more than 20 hours.

5.2 Accuracy

The storage savings provided by the ZDA algorithm comes at a cost, namely accuracy. If a node N_i has only one outgoing edge, then users arriving at N_i can be expected, with high probability, to choose the sole outgoing link. Clearly, in this case, the probability of error is low.

An extreme error case is shown in Figure 3. Here, User U_1 has followed the path $\langle N_5, N_1, N_2, N_7\rangle$, U_2 has followed the path $\langle N_6, N_2, N_3, N_8\rangle$, and U_3 has followed the path $\langle N_7, N_3, N_4, N_9\rangle$. In this scenario, ZDA finds the following edge probability values: for edge $\langle N_1, N_2\rangle$, $\mathcal{P}^{\langle N_1,N_2\rangle} = 1$; for edge $\langle N_2, N_3\rangle$, $\mathcal{P}^{\langle N_2,N_3\rangle} = 0.5$; and for edge $\langle N_3, N_4\rangle$, $\mathcal{P}^{\langle N_3,N_4\rangle} = 0.5$. Based on this, ZDA finds a probability for the path $P^{\langle N_1,N_2,N_3,N_4\rangle} = \mathcal{P}^{\langle N_1,N_2\rangle} \times \mathcal{P}^{\langle N_2,N_3\rangle} \times \mathcal{P}^{\langle N_3,N_4\rangle} = 0.25$. However, in this case, *no user has actually followed this path.* We can expect the error to increase as the average fanout in the graph increases. As demonstrated in Section 6.3.3 (where we introduce two metrics to quantify accuracy), our intuition here is correct.

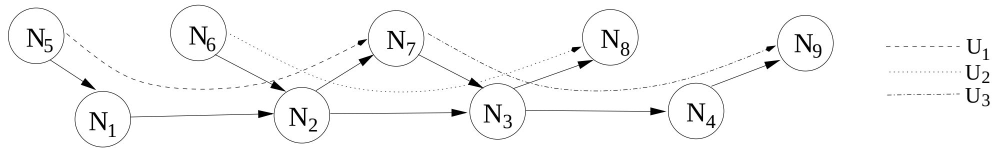

Figure 3: Extreme Error Condition

5.3 Configurable-Depth Algorithm (CDA)

In ZDA, the major source of error is the assumption that the probability of traversing an edge is independent of the traversal path leading to the edge. In essence, the error on an edge $\langle N_u, N_v\rangle$ stems from the lack of information about users' traversal paths up to arrival at N_u. In this section, we propose a *Configurable-Depth Algorithm* (CDA) based on ZDA, but maintain additional path information for each edge, up to a depth of the d most recently traversed edges.

In CDA, rather than assuming independence, we assume a *bounded dependence* on previous path traversals of depth d. For example, in Figure 3, user U_1 has traversed two edge sequences of depth $d = 2$, $\langle N_5, N_1, N_2\rangle$ and $\langle N_1, N_2, N_7\rangle$. Considering edge sequences provides more traversal information than single edges, thereby reducing the error in the probabilities, yet still requires far less storage than ESM. (We elaborate on this claim later in this section, after we have described the algorithm itself.)

In order to accommodate sequences of more than one edge, we need to redefine the notion of edge probability introduced in Section 5.1. Rather than associating a single probability value with an edge $\langle N_u, N_v\rangle$, we will introduce a vector of probability values, where the i^{th} element of the vector represents the probability of arriving at u from u's i^{th} incoming path.

5.3.1 Details of the CDA Algorithm

Associating a vector of probability values with each edge allows us to maintain previous path probabilities up to depth of d. However, it also presents two major problems in utilizing the algorithm shown in Algorithm 1 for CDA. In particular, the multiplication function (line 1 of RELAX) and the *max* function (line 3 of RELAX, and within the EXTRACT-MAX function of line 4 of BEST-PATHS) both assume scalar operands, which is no longer the case. Therefore, we define new functions, *CDA-MULT* and *CDA-MAX* to resolve this issue. In this section, we first describe how we associate a vector of probabilities with an edge, and then *CDA-MULT* and *CDA-MAX*.

Associating a Vector of Probabilities with an Edge. We define the probability of following edge $\langle N_i, N_j\rangle$ on the condition that user has followed a path $\langle N_1, N_2, ..., N_d\rangle$ before arriving at node N_i as follows, where d is the depth of previous path stored.

$$\mathcal{P}^{\langle N_i,N_j\rangle}_{\langle N_1,N_2,...,N_d,N_i\rangle} = \frac{\textit{Number of traversals through path } \langle N_1, N_2, ..., N_d\rangle \textit{and then edge } \langle N_i, N_j\rangle}{\textit{Number of traversals through path } \langle N_1, N_2, ..., N_d\rangle \textit{to arrive at } N_i}$$

Now we can define edge probability $\mathcal{P}^{\langle N_i,N_j\rangle}$ associated with an edge $\langle N_i, N_j\rangle$ in terms of $\mathcal{P}^{\langle N_i,N_j\rangle}_{\langle N_1,N_2,...,N_d,N_i\rangle}$ as follows: $\mathcal{P}^{\langle N_i,N_j\rangle} = \langle \mathcal{P}^{\langle N_i,N_j\rangle}_{\langle N_1,N_2,...,N_d,N_i\rangle} | (\langle N_d, N_i\rangle \in E) \, \&_{x=1}^{d-1} (\langle N_x, N_{x+1}\rangle \in E)\rangle$

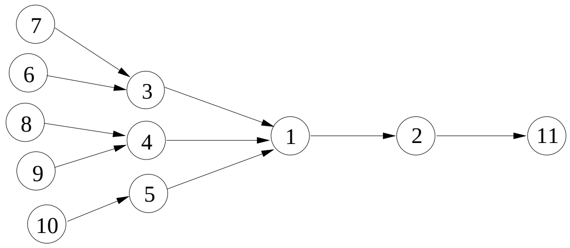

Figure 4: Example Traversal Graph

In order to clarify the above equation, we consider the graph in Figure 4. Suppose we would like to find the probability for edge $\langle N_1, N_2\rangle$ with $d = 0$. Here, $\mathcal{P}^{\langle N_1,N_2\rangle}$ is a single-element vector, as defined in Section 5.1. Since $\langle N_1, N_2\rangle$ has only one outgoing edge, $\langle N_2, N_{11}\rangle$, the probability of traversing $\langle N_1, N_2\rangle$, given that a user has reached N_2 is 1.

More complicated cases arise for $\langle N_1, N_2\rangle$ with $d = 1$ and $d = 2$. For $d = 1$, $\mathcal{P}^{\langle N_1,N_2\rangle}$ will have three elements, one corresponding to each of $\langle N_1, N_2\rangle$'s incoming edges, $\langle N_3, N_1\rangle$, $\langle N_4, N_1\rangle$ and $\langle N_5, N_1\rangle$. Here the edge probability vector is $\mathcal{P}^{\langle N_1,N_2\rangle} = \langle \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_3,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_4,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_5,N_1\rangle}\rangle$. For conditional depth $d = 2$, the probability vector for edge $\langle N_1, N_2\rangle$ will be a 5-element vector, since there are 5 different length-2 paths leading to node N_1. Thus the probability vector for $\langle N_1, N_2\rangle$ is $\mathcal{P}^{\langle N_1,N_2\rangle} = \langle \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_7,N_3,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_6,N_3,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_8,N_4,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_9,N_4,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_{10},N_5,N_1\rangle}\rangle$.

CDA-MULT. Given the above definition of an edge probability vector, we can now define a path probability $\mathcal{P}^{\langle N_1,N_2,...,N_d\rangle}$ for path $\langle N_1, N_2, ..., N_d\rangle$ in terms of the edge probabilities $\mathcal{P}^{\langle N_i,N_j\rangle}$, for edges $\langle N_i, N_j\rangle \in E$ similar to the path weight defined in section 5.1, i.e., as the multiplication of probabilities of edges on a path. However, since we now have a vector of values, rather than a singe value, for each edge probability, we must multiply these probability vectors. Thus, we define the multiplication of edge probability vectors $\mathcal{P}^{\langle N_i,N_j\rangle} = \mathcal{P}^{\langle N_i,N_l\rangle} \times \mathcal{P}^{\langle N_l,N_j\rangle}$ where l is an intermediate node for path $N_i \rightsquigarrow N_j$, as follows: if $\mathcal{P}^{\langle N_i,N_j\rangle}_{\langle N_1,N_2,...,N_d,N_i\rangle} \in \mathcal{P}^{\langle N_i,N_j\rangle}$ and $\mathcal{P}^{\langle N_i,N_l\rangle}_{\langle N_1,N_2,...,N_d,N_i\rangle} \in \mathcal{P}^{\langle N_i,N_l\rangle}$ and $\mathcal{P}^{\langle N_l,N_j\rangle}_{\langle N'_1,N'_2,...,N'_d,N_l\rangle} \in \mathcal{P}^{\langle N_l,N_j\rangle}$, then $\mathcal{P}^{\langle N_i,N_j\rangle} = \langle \mathcal{P}^{\langle N_i,N_j\rangle}_{\langle N_1,N_2,...,N_d,N_i\rangle} = \mathcal{P}^{\langle N_i,N_l\rangle}_{\langle N_1,N_2,...,N_d,N_i\rangle} \times \mathcal{P}^{\langle N_l,N_j\rangle}_{\langle N'_1,N'_2,...,N'_d,N_l\rangle}$ given that $\forall Path_{\langle N'_1,N'_2,...,N'_d\rangle} \in Path_{\langle N_1,N_2,...,N_d,N_i \rightsquigarrow N_l\rangle}$ *and* N'_k *is the node just before node* N_l *on the path* $N_i \rightsquigarrow N_l\rangle$.

To clarify these notions, we consider the example of finding the probabilities of paths ending at $\langle N_2, N_{11}\rangle$ in Figure 4 with depth $d = 2$. Before calculating path probabilities, we need to first find the edge probabilities for $\mathcal{P}^{\langle N_2,N_{11}\rangle}$, $\mathcal{P}^{\langle N_1,N_2\rangle}$, and $\mathcal{P}^{\langle N_1,N_{11}\rangle}$: $\mathcal{P}^{\langle N_2,N_{11}\rangle} = \langle \mathcal{P}^{\langle N_2,N_{11}\rangle}_{\langle N_3,N_1,N_2\rangle}, \mathcal{P}^{\langle N_2,N_{11}\rangle}_{\langle N_4,N_1,N_2\rangle}, \mathcal{P}^{\langle N_2,N_{11}\rangle}_{\langle N_5,N_1,N_2\rangle}\rangle$; $\mathcal{P}^{\langle N_1,N_2\rangle} = \langle \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_7,N_3,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_6,N_3,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_8,N_4,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_9,N_4,N_1\rangle}, \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_{10},N_5,N_1\rangle}\rangle$; and $\mathcal{P}^{\langle N_1,N_{11}\rangle} = \mathcal{P}^{\langle N_1,N_2\rangle} \times \mathcal{P}^{\langle N_2,N_{11}\rangle} =$
$\langle \mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_7,N_3,N_1\rangle}, \mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_6,N_3,N_1\rangle}, \mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_8,N_4,N_1\rangle}, \mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_9,N_4,N_1\rangle}, \mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_{10},N_5,N_1\rangle}\rangle$.

Using these edge probabilities and vector multiplication, we can find path probabilities for the following paths: $\mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_7,N_3,N_1\rangle} = \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_7,N_3,N_1\rangle} \times \mathcal{P}^{\langle N_2,N_{11}\rangle}_{\langle N_3,N_1,N_2\rangle}$ and $\mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_6,N_3,N_1\rangle} = \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_6,N_3,N_1\rangle} \times \mathcal{P}^{\langle N_2,N_{11}\rangle}_{\langle N_3,N_1,N_2\rangle}$ and $\mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_8,N_4,N_1\rangle} =$

$\mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_8,N_4,N_1\rangle} \times \mathcal{P}^{\langle N_2,N_{11}\rangle}_{\langle N_4,N_1,N_2\rangle}$ and $\mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_9,N_4,N_1\rangle} = \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_9,N_4,N_1\rangle} \times \mathcal{P}^{\langle N_2,N_{11}\rangle}_{\langle N_4,N_1,N_2\rangle}$ and $\mathcal{P}^{\langle N_1,N_{11}\rangle}_{\langle N_{10},N_5,N_1\rangle} = \mathcal{P}^{\langle N_1,N_2\rangle}_{\langle N_{10},N_5,N_1\rangle} \times \mathcal{P}^{\langle N_2,N_{11}\rangle}_{\langle N_5,N_1,N_2\rangle}$.

CDA-MAX. In order to find the k most-traversed paths, we must have a means of comparing paths to one another. In order to do this, we define a function for path comparison, *CDA-MAX*. If we have two probability vectors $\mathcal{P}'$ and $\mathcal{P}''$, then we can define the *CDA-MAX*($\mathcal{P}'$,$\mathcal{P}''$) as follows:

$$CDA\text{-}MAX(\mathcal{P}',\mathcal{P}'') = \begin{cases} \mathcal{P}' & if \ \sum_{\forall \mathcal{P}'_{N_i} \in \mathcal{P}'} \mathcal{P}'_{N_i} \geq \sum_{\forall \mathcal{P}''_{N_i} \in \mathcal{P}''} \mathcal{P}''_{N_i} \\ \mathcal{P}'' & otherwise \end{cases}$$

That is, between two paths for $N_i \rightsquigarrow N_j$ with their corresponding probability vector $\mathcal{P}'$ and $\mathcal{P}''$ we choose the path which has highest sum of probability values in the probability vector.

In order to use *CDA-MAX*, we must be able to represent a path probability as a single value, rather than as a vector of probability values. To achieve this, we define $\rho^{\langle N_i,N_j\rangle}$, a single probability value for the path $N_i \rightsquigarrow N_j$, as the sum over the d-depth probabilities in the path $N_i \rightsquigarrow N_j$: $\rho^{\langle N_i,N_j\rangle} = \sum_{\forall \mathcal{P}_d \in \mathcal{P}^{\langle N_i,N_j\rangle}} \mathcal{P}_d$. We define two important properties of the above probability value ρ.

- For path length l and conditional probability depth d, we will get an exact probability value ρ for the path when $l \leq d$.
- The error in path probability value ρ will increase as $l - d$ increases, for $d < l$.

Due to the second property, we can tune our algorithm based on available resources and desired quality of results.

Having defined *CDA-MULT* and *CDA-MAX* operations for our probability vectors, we can modify Algorithm 1 to handle probability vectors rather than scalar probability values. Only two changes must be made. First, the multiplication operation in line 1 of RELAX must use *CDA-MULT*. Second, the EXTRACT-MAX function in line 4 of BEST-PATHS and the *max* function in line 3 of RELAX use the *CDA-MAX* function defined in this section. The remainder of Algorithm 1 remains unmodified for CDA.

5.3.2 CDA Correctness

The correctness of CDA is almost exactly similar to the correctness for ZDA. The only difference is in the mapping to semiring. Rather than the MAX and $\times$ functions in the semiring for ZDA, CDA uses the $CDA - MAX$ and $CDA - MULT$ functions: ([0,1], $CDA - MAX$, $CDA - MULT$, 0, 1)

5.3.3 CDA Time Complexity

Having described the CDA algorithm, we now estimate its time complexity. We first consider the time complexity of individual modules in the CDA algorithm, and then the overall time complexity. The INITIALIZE-SINGLE-SOURCE for CDA is exactly similar to its counterpart in ZDA. Thus, INITIALIZE-SINGLE-SOURCE for CDA also has a time complexity of $O(k \times |V|)$. There are two main differences between ZDA and CDA, in the time required for the RELAX function and in the time required for queue operations

The main differences between RELAX for ZDA and RELAX for CDA can be seen in lines 1 and 3. In line 1, the scalar multiplication of ZDA is replaced by vector multiplication (as defined in Section 5.3) in CDA. Let f be the average fanout of nodes in the traversal graph G. The average number of elements in an edge probability vector is f^d, where d is the conditional depth under consideration. The multiplication of two probability vectors involves time complexity of $O(f^d \times f^d) = O(f^{2d})$. In line 3, the scalar *max* function of ZDA is replaced by a vector *max* function, *CDA_MAX*, (as defined in Section 5.3) in CDA. The vector *CDA_MAX* function requires time $O(f^d)$, because comparison of two probability vectors requires two sums, each of which requires $O(f^d)$. The overall time complexity of a call to RELAX is the k times the cost of the vector *CDA_MAX* function plus the cost of the vector multiplication: $O((k \times f^d) + f^{2d}) = O(f^{2d})$.

In ZDA, the time required for a single queue insert or remove operation is $O(k|V| \ log \ (|V|))$. Each such operation requires $log \ (k|V|)$ comparisons. ZDA uses scalar comparison, which requires constant time for each comparison operation. However, as mentioned above, this comparison in CDA requires vector comparison, which is of $O(f^d)$ time. Thus, each queue operation in CDA requires $O(log\ (k \times |V| \times f^d))$. Since $k \times |V|$ vertices are added to and removed from the queue, the overall time required to for queue insertion and removal operations in CDA is $O(k \times |V|\ log\ (k \times |V| \times f^d))$.

Thus, the total time complexity of BEST-PATHS for CDA is the sum of the cost of initializing the data structures (via INITIALIZE-SINGLE-SOURCE) and the cost of loading each vertex into the priority queue k times and the cost of the removing the each vertex from the priority queue k times and the cost of relaxing each edge k times. Thus, the overall time complexity for the CDA algorithm is: $O(k\times|V|)+O(k\times|V|\ log\ (k\times|V|\times f^d))+O(k\times|V|\ log\ (k\times|V|\times f^d))+O(k\times|E|\times f^{2d})$. Clearly, as in ZDA, the log linear terms dominate. Thus, the overall complexity is $O(k \times |V|\ log\ (k \times |V| \times f^d)) = O(k|V|\ log\ (|V|) \times f^d)$.

5.3.4 CDA Space Complexity

Having considered the time complexity of CDA, we move on to the storage complexity of the algorithm. Storing the vectors for each edge is dependent on the depth d and the fanout f. In particular, we must store a value for each of f possible predecessors for each edge. Taken to depth d, this is $O(f^d)$ per edge. If the average fanout f is estimated at $f = \frac{|E|}{|V|}$, then for all edges the storage requirement is $O(|V| \times f^{d+1})$.

To illustrate this, we return to the case of the e-commerce site that served as the basis of our tests (considered previously in Section 5.1 and Section 4). The graph has an average fanout of 10. Suppose we wish to perform our CDA analysis for a depth of $d = 2$. Thus, we need to store $150,000$ edge probability vectors. Each vector for an edge $\langle N_i, N_j \rangle$ contains f^{d+1} probability-path pairs. A probability-path pair consists of (a) a sequence of $d+1$ nodeids leading to node N_j and the nodeid for N_j itself, and (b) the probability of traversing that sequence of nodes. If we assume that each nodeid is a 4-byte integer and each probability value is a 8-byte float, then each probability-path pair in an edge probability vector for $d = 2$ will require 20 bytes of storage (4 integers for nodeids, and one float for the probability value). Thus, in this case, the CDA storage requirement (in the worst case) is $150,000 \times 10^3 \times 24$ bytes, or 3.6 GB. This is larger than the megabyte storage requirement for ZDA, but still much smaller than the petabyte storage requirement for ESM. We show, in our performance results in Section 6, that this vast reduction in storage cost results in very little loss in accuracy.

Interestingly, the $O(|V| \times f^{d+1})$ complexity can describe all possible storage scenarios, from minimum information (ZDA, described in Section 5.1) to full information (ESM, described in Section 4). In the case of ZDA, where $d = 0$, and the storage complexity is $O(|V| \times f^{0+1})$, or simply $O(|V| \times f)$. In the worst case for ESM, $d = l$ (where l is the maximum session length), giving a storage complexity of $(O(|V| \times f^l))$.

5.3.5 Applying CDA to Real Data

In order to give a sense of the running time required for CDA on real data, we ran CDA on user traces from the e-commerce site that served as the basis for our tests. We ran the CDA algorithm for $k = 3$ and $d = 2$. The algorithm completed running in 15 minutes on the same hardware and software described in the case studies in Sections 4 and 5.1. For an offline process, executed in batch mode, this running time is clearly practical. The overall storage requirement for the data in these tests was 3 GB – again, a workable data size, particularly when compared to the requirements of ESM.

5.4 Summary of Methods

Having described three methods of finding the k most traversed paths on an e-commerce site, i.e., ESM, ZDA, and CDA, we summarize the properties of our methods here. The running time and storage requirement test results cited here refer to tests carried out in cooperation with our partner e-commerce site. Here, the size of the raw input user traversal data is the same for all experiments; the amount of storage varies among the different methods depending on the degree of aggregation of the traversal data for differing depth d values in ZDA and CDA. These tests were run for $k = 3$ for both CDA and ZDA; in terms of depth, $d = 0$ for ZDA, and $d = 2$ for CDA.

ESM, which stores full information, provides the correct k most-traversed paths as output. However, for a site of moderate size and traffic load, both the amount of storage required to maintain full session data (on the order of petabytes) and the time required to query this data (see the discussion of Figure 2) make practical use of ESM infeasible.

At the other end of the spectrum, ZDA stores minimum session information, based on the assumption that the probability of traversing a particular edge is independent of previous path traversals on the site. This allows ZDA to run extremely quickly over its small data size (9 seconds over 24 MB of data for our example e-commerce site). However, the fast running time and low storage requirement come at the cost of accuracy.

CDA provides a good tradeoff between ESM and CDA. By bounding the dependence of a particular edge probability to d previous edges, CDA limits the storage requirements and running time, the major problems of ESM. By maintaining previous path information up to a depth of d previous edges, CDA provides more accurate results than ZDA. While the running time and storage requirements of CDA (15 minutes on 3 GB of data) are larger than those of ZDA, they are still quite practical.

6 Performance Evaluation

In this section, we evaluate the performance of our algorithms. Since storage space and running time are not really issues (as demonstrated previously), our focus here is on determining the accuracy of our algorithms. Specifically, we would like to determine to what extent varying the independence assumption (as encoded by d) affects accuracy. Measuring the accuracy of our algorithms requires comparing the quality of the output of ZDA and CDA to that of ESM (where ESM produces correct results). However, it is not feasible to run ESM on any kind of real data (recall Figure 2). Thus, we use a synthetic site of 10,000 nodes, based on the characteristics of the 150,000 node site that served as the basis for our tests. This synthetic site is large enough to show realistic accuracy trends, yet small enough to make ESM results tractable. We have, in fact, found similar results on real-world data based on a portion of a site (a sub-tree of the site catalog). Rather than reporting partial results, however, we decided to simulate a smaller version of the real site and report those results.

Readers may, at this point, wonder how quality of solution would scale with the size of the graph. It turns out that the size of the site graph has no impact on the quality of the results (as will be demonstrated in Section 6.3.5). Thus, the choice of a smaller site graph, while enabling accuracy computation, does not contaminate the quality of our results.

Our discussion of our performance results is structured as follows. We first describe the underlying model of the experiments, then our performance metrics. Finally, we present the results of our experiments.

6.1 Experimental Model

As described in Section 2, we model a web site as a graph $G = (V, E)$. For the purposes of these experiments, G consists of $|V|$ nodes and $\mathit{MeanFanout} \times |V|$ edges, where *MeanFanout* is the average number of outgoing edges from a node in the graph.

NumUsers users traverse G, each starting at a node chosen from a uniform distribution across all nodes in G. Note that this confers a significant disadvantage to our algorithms – in the real world, users will begin traversing the site at a limited number of start nodes (e.g., the home page, and perhaps the first-level category pages). Had we used a limited set of starting nodes in these experiements, our accuracy results would be significantly better than those reported here.

Users traverse the site G for *SessionLength* clicks, and choose links according to a *TraversalLocality*, following, for example, the familiar 80/20 rule. Here, we represent *TraversalLocality* by the percentage of nodes that are frequently traversed (i.e., 0.20 for the 80/20 rule).

Table 3 shows the range of parameter values for these variables in our experiments.

Parameter	**Minimum**	**Maximum**	**Baseline**
$\lvert V\rvert$	5000	20000	10000
MeanFanout	2	10	5
NumUsers	4000	4000	4000
SessionLength	20	20	20
TraversalLocality	0.1	0.3	0.2
d	0	3	1

Table 3: Nominal Parameter Values

6.2 Performance Metrics

We wish to know how much error is introduced by our approximate methods (ZDA and CDA) as compared to ESM, which produces exact results. We base our metrics, then, on how well ZDA and CDA are able to identify the CESs derived from ESM results based on the same underlying session data.

In order to discuss error metrics, we must first describe how one can compare the set of k paths returned by ESM to the set returned by ZDA or CDA. Let P_{exact} be the set of paths returned by ESM and P_{approx} be the set of paths returned by either ZDA or CDA. By definition, P_{exact} is the exact set of CESs, and represents the no-error case. Intuitively, the more the subpaths (i.e., CESs) of P_{approx} overlap with the subpaths of P_{exact}, the more accurate the solution provided by the approximating method P_{approx}. More formally, we define an m-edge sequence set $\pi_m(P)$ of P as follows.

$$\pi_m(P) = \{p : |p| = m \textit{ and } p \in P\}$$

That is, $\pi_m(P)$ is the set of all subpaths p of length m in P.

As the reader will recall from the discussion in Section 2, CESs can be derived from a set of k popular paths by filtering subsequences of those k paths, based on application-dependent criteria, such as a minimum probability threshold or frequency of occurrence in the k paths. For the purposes of error checking, however, we compare the *unfiltered* results in P_{approx} to those in P_{exact}. Note that this places CDA and ZDA at a distinct disadvantage: low-probability subsequences in ESM are likely sources of error in our approximate methods, yet these subsequences have the priority in our accuracy metrics as high-probability subsequences. In this way, we are considering the worst-case scenario in terms of the accuracy of our methods.

We propose two error metrics, ϵ and δ, based roughly on *precision* and *recall* measures in IR literature. Here, ϵ measures the proportion of CESs found in error, in much the same way that *precision* measures error in a set of returned items. Intuitively, ϵ tells us what proportion of the results *should not have been returned.*

On the other hand, δ measures correct CESs missed by the approximating algorithm. This is similar to *recall.* Intuitively, δ tells us what proportion of the correct result set *should have been returned, but wasn't.*

We define ϵ as a similarity function:

$$\epsilon = \frac{\sum_{l=1}^{m} l \times |\pi_l(P_{exact} - P_{approx})|}{\sum_{l=1}^{m} l \times |\pi_l(P_{exact})|}$$

Here, the greater the overlap of edge sequences between P_{exact} and P_{approx}, the closer the result is to the ideal solution, and, thus, the lower the error. The multiplication by l gives more weight to longer matching subpaths than to shorter ones. Intuitively, this makes sense – longer matching subpaths are better than shorter ones, since they indicate a stronger overall correlation between P_{exact} and P_{approx}. If all edge sequences of P_{exact} match those of P_{approx}, then all CESs have been found. This is the no-error case, where $\epsilon = 0$. Similarly, if none of the edge sequences match, we have full error, and $\epsilon = 1$.

For example, consider the traversals shown in Figure 5. Here, the path $P_{exact} = \langle N_1, N_2, N_3, N_4, N_5\rangle$ was returned from ESM, while the path $P_{approx} = \langle N_1, N_2, N_3, N_6, N_7, N_8, N_4, N_5\rangle$ was returned by CDA with depth $d = 1$. Here, $m = 7$, and the denominator, $\sum_{l=1}^{m} l \times |\pi_l(P_{exact})| = 20$. To find the numerator, we wish to count the number of CESs of each length from P_{exact} that do not appear in P_{approx}. Here, one CES of length 1, two CESs of length 2, three CESs of length 3, and four CESs of length 4 from P_{approx} fail to match CESs in P_{exact}, making $\sum_{l=1}^{m} l \times |\pi_l(P_{exact} - P_{approx})| = 13$. By our expression, then, $\epsilon = 0.65$. Here, error is quite high. This occurs primarily for two reasons. First, our example (for simplicity) considers only two path sequences leading from N_1 to N_5; presumably, in a realistic scenario considering a site graph with large average fanout, there would be more than two paths leading from the start node to the end node. Second, the longest matching subsequence consists of only two edges (i.e., $\langle N_1, N_2, N_3\rangle$) – a longer matching subsequence would significantly reduce error, since ϵ gives preference to longer matching sequences than shorter ones.

While ϵ measures the similarity between P_{exact} and P_{approx}, we would also like to measure the difference in the results. There is clearly quite a bit of difference between the two paths shown in Figure 5: the

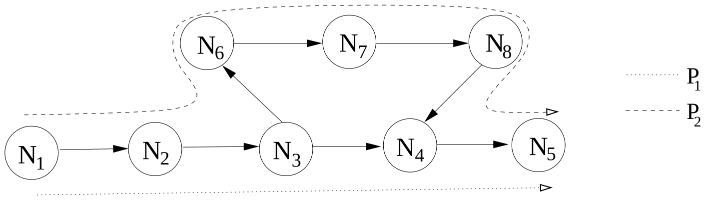

Figure 5: Similarity and Difference of Paths

segment $N_3 \rightsquigarrow N_4$, P_{approx} contains only the sequence $\langle N_3, N_4\rangle$, while P_{exact} contains a much longer sequence, $\langle N_3, N_6, N_7, N_8, N_4\rangle$. This difference is not necessarily dependent on ϵ, since ϵ considers only matching subsequences. We measure this difference with δ, defined as follows:

$$\delta = \frac{\sum_{l=1}^{m} l \times |\pi_l(P_{approx} - P_{exact})|}{\sum_{l=1}^{m} l \times |\pi_l(P_{approx})|}$$

where $\pi_m(P_{approx} - P_{exact}) = \pi_m(P_{approx}) - \pi_m(P_{exact})$. Ideally, $\delta = 0$, i.e., the paths in P_{exact} do not differ at all from the paths in P_{approx}. Similarly, if the paths in P_{exact} and P_{approx} are disjoint, then $\delta = 1$. Also like ϵ, multiplication by l gives more weight to longer non-matching subpaths than shorter ones. Here, the longer the non-matching subpath, the greater the error, and the higher the δ value.

To illustrate the use of δ, we return to the example in Figure 5. Here, $m = 7$, and $\sum_{l=1}^{m} l \times |\pi_l(P_{approx} - P_{exact})| = 74$, since there are 4 CESs of length one in P_{approx} but not P_{exact}, as well as 5 CESs of length two, 5 CESs of length three, 4 CESs of length four, 3 CESs of length five, 2 CESs of length six, and 1 CES of length seven. The denominator, $\sum_{l=1}^{m} l \times |\pi_l(P_{approx})| = 84$, since there are 84 possible CES matches. Thus, $\delta = 0.88$ in this example. Like ϵ, δ here is quite high. This occurs for the same reasons – the small number of paths compared, and the length of the longest non-matching subpath (i.e., the 4-edge path $\langle N_3, N_6, N_7, N_8, N_4\rangle$ in P_{approx} that is not in P_{exact}).

6.3 Experimental Results

In this section, we report the results of our performance experiments. In particular, we are interested in how well the algorithms finds the CESs on paths leading from N_a to N_b given varying assumptions of dependence, as well as the algorithms' sensitivity to fanout in the graph, traversal locality, and graph size.

All performance graphs exhibit mean values that have relative half-widths about the mean of less than 10% at the 90% confidence level. We only discuss statistically significant differences in the ensuing reporting section. Each data point represents the average of five experimental values.

6.3.1 Baseline Experimental Results

We first discuss how the two error values, i.e., ϵ and δ interact with k (i.e., the number of popular paths considered) and d (i.e., the depth to which traversal probabilities are dependent).

Figures 6A and 6B show the ϵ and δ results respectively for ESM, ZDA and CDA for a set of baseline parameters, *MeanFanout*= 5 and *TraversalLocality*= 0.2. Clearly, the ZDA curve corresponds to $d = 0$, while CDA curves are plotted for three different depths, $d = 1$, $d = 2$, and $d = 3$. On the X-axis, k is varied between 1 and 5. We include ESM for completeness; since this method uses full information and always produces exact results, i.e., ϵ and $delta$ for ESM are always 0.

In Figure 6A, the *epsilon* results portray an increasing trend with progressively diminishing slope. At low k, ϵ values are low, signifying that the algorithms do relatively well, but do progressively worse as k increases. The increasing trend is easy to explain – essentially, as fewer traversals are recorded over less popular paths (compared to more popular paths), our algorithms have a harder time recognizing frequently traversed sequences. More accurately, it is easier to distinguish between the most popular and the 3^{rd} most popular paths, than between the 3^{rd} and 5^{th} most popular paths.

The explanation of the diminishing slope is a little more subtle. This occurs because the error values of any k^{th} most popular path computation already includes the error effects of the $k-1$ most popular

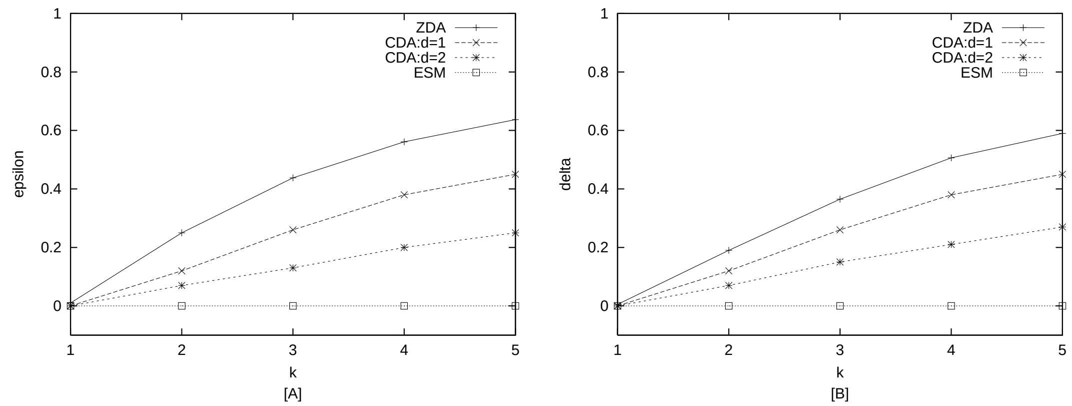

Figure 6: Effect of depth d on Accuracy

paths. For instance, the ϵ value of 5 most popular paths will automatically factor in the error effect of the 4 most popular paths, as these 4 paths will be included in the set of 5 most popular paths. Thus, the only additional error that is introduced is due to the k^{th} path. Clearly, as k increases, the error effect of the first $(k-1)$ paths increasingly dominate the overall error effect of all k paths.

Having described the overall shape of the curves, we now turn our attention to comparing the curves. As expected, ZDA is the worst, as it is completely "memoryless", i.e., it assumes that the probability of traversing an edge is independent of previous path traversal. Our conclusion is that ZDA is not preferable in discovering CESs, especially when comparatively large values of k are used.

Now we consider the CDA curves – as is apparent from Figure 6A, the quality of solutions increase with increasing depth. Larger d values correspond to increased relaxation of the independence assumption of the ZDA algorithm, leading to lower errors for all levels of k. In particular, for a moderate depth of 2, at k of 5, approximately 80% of CESs were discovered accurately, and approximately 90% were discovered correctly for $k = 3$. Our recommendation to our click and mortar retailer was to use CDA with depth of 2 or 3.

It is interesting to note that in all cases, all the CDA experiments yielded exact results for $k = 1$. In other words, when considering only the single most traversed path leading from the start node to the end node, CDA found the same most traversed path as ESM.

Having considered the curve for ϵ, we move on to describe the curve for δ in Figure 6B. It turns out that the trends of δ curves closely resemble those of the trends for ϵ curves, for very similar reasons. Thus, we report only ϵ results for the remainder of this paper.

6.3.2 Looking for an Optimal Depth

An interesting question to consider is whether or not there exists an optimal depth d, such that the cost/accuracy tradeoff is optimized. If such an optimal depth exists, such a value would be of significant practical value for an analyst in discovering CESs.

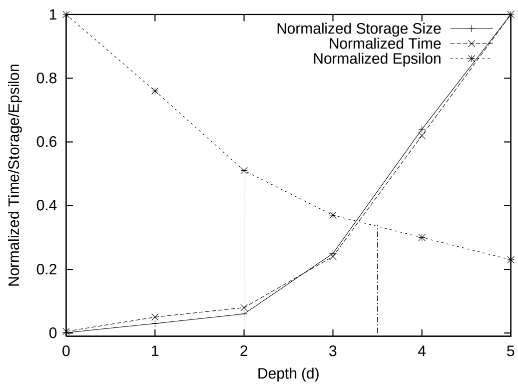

Figure 7: Cost of accuracy

In order to explore this issue, we have graphically shown how cost and accuracy vary with depth in Figure 7, based on our baseline experiments above. Here, the x-axis shows varying values of d, while the y-axis represents normalized values for storage size, time, and ϵ. Here, the metrics are normalized on a scale from 0 to 1 in order to aid in graphing. To normalize the values, we took the maximum experimental values for each metric (based on the baseline experiments for $k = 3$), and divided all experimental values for that metric by that maximum value. For instance, consider the normalized ϵ values. We obtained the maximum value of ϵ, i.e., 0.43, the $d = 0$ (ZDA) case in Figure 6A, and used it as the normalized maximum value in Figure 7. The normalized ϵ values for larger d's were obtained by dividing the corresponding ϵ values by the maximum values.

The first notable feature in Figure 7, though somewhat obvious in hindsight, is the similarity of the curves for time and storage size. These curves are exponential, i.e., as depth increases, the rate of increase of costs for storage and time increases. Intuitively, this trend makes sense – increasing d requires storing larger vectors of probability values for each edge (increasing storage costs), which in turn increases the time required to run CDA on the data.

The curve for ϵ, on the other hand, trends downward. Since ϵ is a measure of error, this trend makes sense – as d increases, the amount of previous path traversal information considered in calculating the probability of traversing an edge increases, which leads to decreasing error. The curve is asymptotic, i.e., ϵ decreases rapidly at low depth values, but the rate of decrease slows as d increases. This also makes sense. At low d values, much accuracy can be gained by increasing d. However, as d approaches l, the maximum user session length, the rate of accuracy gain decreases, since each incremental increase of d provides only marginal gains in accuracy.

Clearly, then, there does exist an optimal range of d values in the experiments reported. In Figure 7, this range is from $d = 2$ to d slightly greater than 3. Since d must be an integer value, the effective range is from $d = 2$ to $d = 3$. Ideally, it would have been interesting to derive this range analytically; however, we were unable to do this for this paper.

6.3.3 Sensitivity to Fanout

Here, we discuss the effect of fanout on the accuracy of our methods. Intuitively, one would expect higher fanouts to result in lower accuracy due to the larger number of possible paths through the site, while lower fanouts should lead to greater accuracy, since this would decrease the number of possible paths through the site. Figure 8A shows the results of experiments varying fanout f. Here, the curve for Fanout=5 is exactly similar to the baseline curve, shown in Figure 6A, with *TraversalLocality* of 0.2. In order to show the effect of varying the fanout in the graph, we show ϵ for Fanout=2 and Fanout=10.

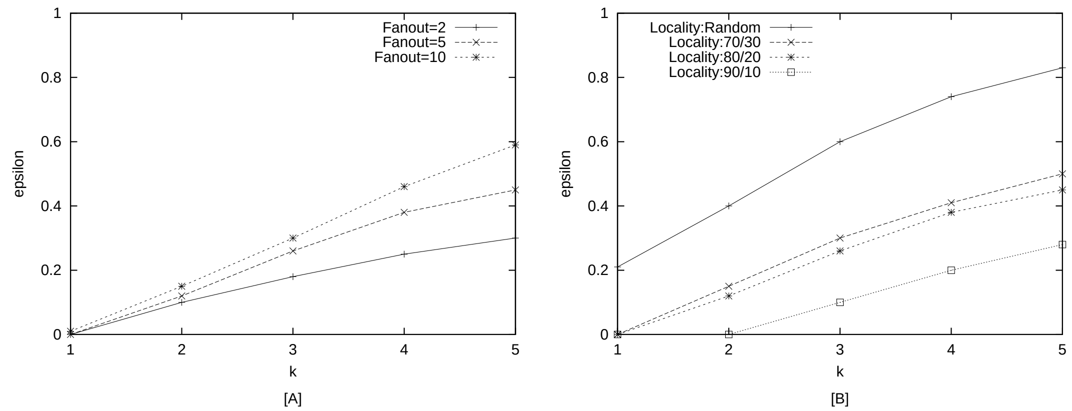

Figure 8: Sensitivity to Fanout and Locality

The curve for Fanout=2 is similar in shape to the baseline curve, but the overall rate of increase of error and the error values themselves are much lower. This occurs because a lower fanout in the graph affords fewer possibilities for error. Similarly, the curve for Fanout=10 has a shape similar to that of the baseline curve, but with a much higher rate of increase of error and higher overall error values. An increased fanout provides the opportunity for a larger number of errors and, as expected, shows increased error over the baseline case.

6.3.4 Sensitivity to Locality of Reference

Locality of reference refers to the magnitude of popularity of items in a dataset, in this case paths through an e-commerce site. Intuitively, one would expect increased locality of reference (i.e., greater popularity of popular paths) within a site to lead to greater accuracy, while decreased locality of reference would result in lower accuracy. Figure 8B shows the results of experiments varying *TraversalLocality*. Here, the curve for Locality:80/20 is exactly similar to the baseline curve, shown in Figure 6A, with *MeanFanout* of 5. In order to show the effect of varying the locality of reference in the graph, we show ϵ for Locality:Random (i.e., uniformly distributed), Locality:70/30, and Locality:90/10.

The curve for Locality:70/30 shows the same general shape as the baseline curve, with higher overall error rates. In fact, the curve is quite similar to the baseline curve. This occurs because, for a fanout of 5, the difference between 70% of users choosing 30% of the links at a given decision point is not all that different from 80% of users choosing 20% at the same decision point – in this situation, 30% of the links at a fanout of 5 is very similar to 20% of those same links.

One would expect, given the similarity between the Locality:70/30 and the baseline curves (Locality:80/20), that the curve for Locality:90/10 would also be similar. While the general shape follows the same trend, the overall ϵ values are much lower. This occurs because the Locality:90/10 experiments all found the 2^{nd} most popular path, thus reducing the cumulative error for $k > 2$ (see the discussion on cumulative error in Section 6.3.1).

The curve for Locality:Random shows the same general shape as the baseline curve as well, but with higher error rates than Locality:70/30 and the baseline case. The increased error rates occur because lower locality of reference leads to lower frequencies of traversal for the most traversed paths, which in turn leads to increased error rates in finding them.

6.3.5 Sensitivity to Graph Size

Throughout this paper, we have discussed several factors that can affect the accuracy of our methods (e.g., d, fanout, and traversal locality). However, the size of the graph is not among these factors in our discussions. Intuitively, the actual size of the graph has little effect on user traversal decisions, since any given user can only traverse a small portion of the graph at any given time. This set of experiments, shown in Figure 9, bears out this intuition.

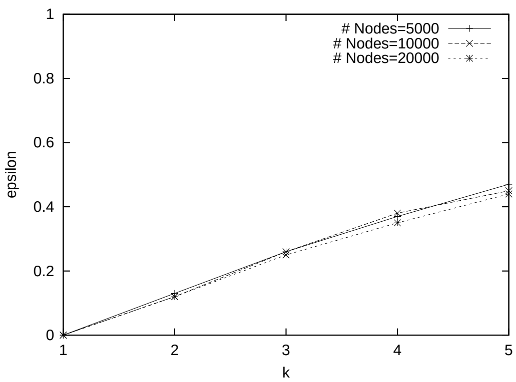

Figure 9: Sensitivity to Graph Size

The curves here show the results of experiments varying the size of the graph. Here, the curve for #Nodes=10,000 is exactly similar to the baseline curve, shown in Figure 6A, with *TraversalLocality* of 0.2. In order to show the effect of varying the size of the graph, we show ϵ for #Nodes=5000 and

#Nodes=20000. The curves for #Nodes=5000 and #Nodes=20000, as expected intuitively, are quite similar to the baseline curve. Based on this experimental evidence, we can draw the conclusion that the size of the underlying graph has little effect on the accuracy of our methods.

7 Conclusion

In this paper, we are interested in finding the Critical Edge Sequences (CESs), i.e., frequently traversed sequences of edges on the paths leading from N_a to N_b on a web site. We propose approximate methods for finding these frequently traversed paths. Our methods are both space-efficient and practical, providing a tunable parameter d (depth), which provides an option to trade increased time and space costs for greater accuracy of results. We present three methods for finding frequently traversed paths: (a) ESM, which produces exact results, but requires extremely large time and storage resources; (b) ZDA, which requires minimal time and storage, at the cost of accuracy, and assumes that the probability of traversing an edge is independent of previous path traversal; and (c) CDA, which provides greater accuracy than ZDA by relaxing the independence assumption.

We explore the performance of our algorithms using synthetic datasets, using performance metrics based on the familiar IR notions of precision and recall. In particular, we are interested in how varying the independence assumption affects how well the algorithms find the CESs on the paths leading from N_a to N_b, as well as the algorithms' sensitivity to fanout in the graph, traversal locality, and depth of conditional probability.

Based on our results, we reach five conclusions: (a) as the value of k increases, the error in the i^{th} ($i <= k$) path in the result increases as well, i.e., the error for $k = 1$ is less than the error for $k = 2$, and so on; (b) as the depth d of conditional probability increases, the error in the result decreases; (c) there exists an optimal depth d, with the best possible tradeoff between cost and accuracy; (d) as the fanout f in the graph increases, the error in the result increases as well; and (e) as the locality of reference increases, i.e., as the size of the frequently-referenced portion of the site decreases for a constant overall site size, the error in the result decreases.

Our plans for future work include modifying our algorithms to support user-defined goals of frequency/rarity, or path properties.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB 1994)*, pages 487–499, 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, March 1995.

[4] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[5] R.J. Bayardo and R. Agrawal. Mining the most interesting rules. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 145–154, 1999.

[6] M-S Chen, J.S. Park, and P.S. Yu. Data mining for path traversal patterns in a web environment. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 385–392, 1996.

[7] T.H. Cormen, C.E. Lieserson, and R.L. Rivest. *Introduction to Algorithms.* McGraw Hill, 1998.

[8] A. Datta, D. VanderMeer, K. Ramamritham, and S. Navathe. Toward a comprehensive model of the content and structure of, and user interaction over, a web site. In *Proceedings of the VLDB Workshop on Technologies for E-Services*, Cairo, Egypt, September 2000.

[9] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web sites. In *Proceedings of the 25th VLDB Conference*, pages 627–638, September 1999.

[11] F. Korn, A. Labrinidis, Y. Kotidis, and C. Faloutsos. Quantifiable data mining using ratio rules. *VLDB Journal*, pages 254–266, 2000.

[12] B. Mosbasher, N. Jain, E. Han, and J. Srivastava. Web mining: Pattern discovery from world wide web transactions. Technical Report 96-050, University of Minnesota, Dept. of Computer Science, Minneapolis, 1996.

[13] J. Pitkow and P. Priolli. Mining longest repeating subsequences to predict world wide web surfing. In *Proceedings of USITS'99: The 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, October 1999.

[14] D. Simpson. Corral your storage management costs. *Datamation*, pages 88–93, April 1997.

[15] M. Spiliopoulou, L.S. Faulstich, and K. Winkler. A data miner analyzing the navigaitional behavior of web users. In *International Conference of ACAI'99: Workshop on Machine Learning in User Modelling*, 1999.

[16] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology*, pages 3–17, March 1996.

[17] M.J. Zaki, N. Lesh, and M. Ogihara. Planmine: Sequence mining for plan failures. In *Proceedings of the 4th Intl. Conference on Knowledge Discovery and Data Mining*, pages 369–373, 1998.