Seeking Stable Clusters in the Blogosphere

Nilesh Bansal
University of Toronto
nilesh@cs.toronto.edu

Fei Chiang
University of Toronto
fchiang@cs.toronto.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Frank Wm. Tompa
University of Waterloo
fwtompa@uwaterloo.ca

ABSTRACT

The popularity of blogs has been increasing dramatically over the last couple of years. As topics evolve in the blogosphere, keywords align together and form the heart of various stories. Intuitively we expect that in certain contexts, when there is a lot of discussion on a specific topic or event, a set of keywords will be correlated: the keywords in the set will frequently appear together (pair-wise or in conjunction) forming a cluster. Note that such keyword clusters are temporal (associated with specific time periods) and transient. As topics recede, associated keyword clusters dissolve, because their keywords no longer appear frequently together.

In this paper, we formalize this intuition and present efficient algorithms to identify keyword clusters in large collections of blog posts for specific temporal intervals. We then formalize problems related to the temporal properties of such clusters. In particular, we present efficient algorithms to identify clusters that persist over time. Given the vast amounts of data involved, we present algorithms that are fast (can efficiently process millions of blogs with multiple millions of posts) and take special care to make them efficiently realizable in secondary storage. Although we instantiate our techniques in the context of blogs, our methodology is generic enough to apply equally well to any temporally ordered text source.

We present the results of an experimental study using both real and synthetic data sets, demonstrating the efficiency of our algorithms, both in terms of performance and in terms of the quality of the keyword clusters and associated temporal properties we identify.

1. INTRODUCTION

The popularity of blogs has been increasing dramatically over the last couple of years. It is estimated [15] that the size of the blogosphere in August 2006 was two orders of magnitude larger than three years ago. According to the same sources, the total number of blogs is doubling every two hundred days. Technorati, a weblog tracking company, has been tracking fifty million blogs. Blogging is gaining popularity across several age groups. Young people in the age group of 13-29 are generating the bulk (91%) of blogging activity [13].

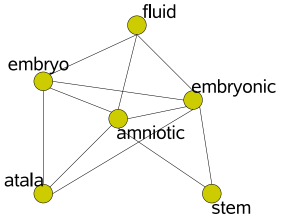

Figure 1: An example cluster of keywords appearing in the blogosphere on January 8 2007 corresponding to the following event: On January 7 2007, scientists at Wake Forest University led by Dr. Anthony Atala report discovery of a new type of stem cell in amniotic fluid. This may potentially provide an alternative to embryonic stem cells for use in research and therapy.

Bloggers are producing vastly diverse kinds of information. General topics include personal diaries, experiences (traveling, concerts), opinions (about products, events, people, music groups, businesses), information technology, and politics to name a few. The importance of this information is highly significant. The blogosphere is an unregulated collective, and it evolves by the contributions of individuals; collecting, monitoring and analyzing information on blogs can provide key insights on 'public opinion' on a variety of topics, for example products, political views, entertainment, etc. At the University of Toronto we have been building BlogScope, a feature-rich search and analysis engine for blogs (`www.blogscope.net`). The search engine incorporates algorithms to aid navigating the blogosphere, points to events of interest via information bursts, plots relevant blogs on a geographical map, and presents keywords related to a search. At regular time intervals BlogScope collects, parses and indexes new blog posts and updates several structures in its keyword index. At the time of this writing, BlogScope was indexing around 75 million posts containing over 13 million unique keywords. A complete description of the system and its architecture is available elsewhere [3, 2].

As topics evolve in the blogosphere, keywords align together and form the heart of various stories. Intuitively we expect that in certain contexts, when there is a lot of discussion on a specific topic or event, a set of keywords will be correlated: the keywords in the set will frequently appear together (pair-wise or in conjunction) forming a cluster. In other words, keywords are correlated if a large number of bloggers use them together in their respective blog posts. Note that such keyword clusters are temporal (associated with specific time periods) and transient. As topics recede, associated keyword clusters dissolve, because their keywords do not

VLDB '07, September 23-28, 2007, Vienna, Austria.

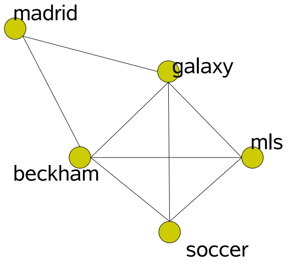

Figure 2: An example cluster of keywords appearing in the blogosphere on January 12 2007 corresponding to the following event: Soccer star David Beckham announces on Jan 11 he is to leave Real Madrid and join Major League Soccer (mls) team LA Galaxy at the end of the season.

appear frequently together anymore. For example, we would expect that the keywords ‘saddam’, ‘hussein’, ‘trial’ formed a cluster when the trial of the former Iraqi president took place (on November 5 2006) as many people blogged about the trial of Saddam Hussein. However the keywords ‘saddam’, ‘hussein’ and ‘dead’ would form a cluster after his execution on December 30 2006. For more examples, consider Figures 1 and 2. Identifying such clusters for specific time intervals is a challenging problem. The associations between keywords reveals *chatter* in the blogosphere that may be of significant actionable value for many domains (e.g., marketing, law enforcement). Moreover it can be of value for improving and refining the quality of search results for specific keywords. If a search query for a specific interval falls in a cluster, the rest of the keywords in that cluster are good candidates for query refinement.

In this paper we formalize and provide solutions for problems related to the temporal association of sets of keywords in the blogosphere. Although we focus on the domain of blogs (since we have a large collection of data via BlogScope), our discussion and techniques are generic enough to apply to any other temporally ordered text source. In particular, we make the following contributions in this paper:

- We present fast algorithms to identify sets of correlated keywords (keyword clusters) in the blogosphere at any specified temporal interval. BlogScope currently contains more than 13M keywords in its index. Any algorithm aiming to identify keyword associations at this scale needs to be efficient.

- We formalize and present algorithms for the notion of *stable* keyword clusters. Since associations of sets of keywords is dynamic, stable clusters aim to identify sets of keywords that exhibit associations over several temporal intervals. Such keyword sets would probably point to events of interest, as it is evident that there is significant use of the keywords in the set, in conjunction, for extended periods of time.

- Since temporal information sources evolve continuously we present streaming (online) versions of our algorithms. This enables us to update the result set efficiently as new information arrives without re-computing everything. Such a requirement on algorithms is essential in order to cope with the temporal nature of our problem domain.

- We present an evaluation of our algorithms demonstrating their practical significance using real data sets and evaluate their scalability for very large data collections and problem settings.

Our core technology extends beyond blogs to social networking sites making heavy use of tagging, such as `flickr.com` and `del.icio.us`. Related processing to the one we conduct for keywords in blogs can be conducted on tags as well. This paper is organized as follows: in Section 2 we briefly review related work. Section 3 presents our methodology for cluster generation. In Section 4 we formally define stable clusters and present our algorithms for identifying them. Section 5 presents the results of a quantitative comparison of our algorithms for various parameters of interest. Qualitative results for clusters discovered from real data are also presented in the same section. Finally Section 6 concludes this paper.

2. RELATED WORK

Graph partitioning has been a topic of active research (see [10] and references therein). A k-way graph partitioning is defined as a partitioning of a graph G into k mutually exclusive subsets of vertices of approximately the same size such that the number of edges of G that belong to different subsets is minimized. The problem is hard, and several heuristic approaches have been proposed. In particular, multilevel graph bisection [10] has attracted research attention. Although such heuristic techniques have been tested on fairly large graph sizes (on the order of half a million vertices and few million edges) [10], they have the constraint that the number of partitions has to be specified in advance (as is common with clustering algorithms).

Correlation clustering [1] drops this constraint, and it produces graph cuts by specifying global constraints for the clusters to be produced. More specifically given a graph in which each edge is marked with a ‘+’ or a ‘-’, correlation clustering produces a partitioning of the graph such that the number of ‘+’ edges within each cluster and the number of ‘-’ edges across clusters is maximized. Although approximation algorithms are provided for this problem, the algorithms presented in [1] (as well in subsequent work [9] for a more restricted version of the problem) are very interesting theoretically, but far from practical. Moreover the existing algorithms require the edges to have binary labels, which is not the case in the applications we have in mind.

Flake et al. [8] present an alternative formulation of graph clustering in which they solve the problem using network flows. The drawback of this approach is that it requires the specification of a sensitivity parameter α before executing the algorithm, and the choice of α affects the solutions produced significantly. Moreover the running time of such an algorithm is prohibitively large for the graphs we have in mind, as they require solutions of *multiple* max-flow problems. Even the fastest algorithms known for max-flow are $O(VE)$, for V vertices and E edges, both of which are in the order of millions in our problem. (In our implementation, the algorithm of Flake et al. required six hours to conduct a graph cut on a graph with a few thousand edges and vertices.) Moreover it is not clear how to set parameters of this algorithm, and no guidelines are proposed in [8].

Various measures have been utilized in the past to assess associations between keywords in a corpus [12]. We employ some of these techniques to infer the strength of association between keywords during our cluster generation process.

3. CLUSTER GENERATION

Let $\mathcal{D}$ denote the set of text documents for the temporal interval of interest. Let $D \in \mathcal{D}$ be a document, represented as a bag of words, in this document collection. For each pair of keywords u, v, $A_D(u, v)$ is assigned one if both u and v are present in D

Date	File Size	# keywords	# edges
Jan 6	3027MB	2889449	138340942
Jan 7	2968MB	2872363	135869146

Table 1: Sizes of resulting keyword graphs (each for a single day) for January 6 and 7 2007 after stemming and removal of stop words.

and zero otherwise. Addition of $A_D(u,v)$ over all documents, $A(u,v) = \sum_{D\in\mathcal{D}} A_D(u,v)$, represents the count of documents in $\mathcal{D}$ that contain both u and v. This way, triplets of the form $(u, v, A(u,v))$ can be computed. Let V be the union of all keywords in these triplets. Each triplet represents an edge E with weight $A(u,v)$ in graph G over vertices V. Further, let $A(u)$ denote the number of documents in $\mathcal{D}$ containing the keyword u. This additional information is required for computing $A(u,\overline{v})$, which represents the number of documents containing u but not v.

For our specific case, the BlogScope crawler fetches all newly created blog posts at regular time intervals. The document collection $\mathcal{D}$ in this case is the set of all blog posts created in a temporal interval (say every hour or every day). The number $A(u,v)$ represents the number of blog posts created in the selected temporal interval containing both u and v. BlogScope is currently indexing around 75 million blog posts, and fetches over 200,000 new posts everyday. The computation of the triplets $(u, v, A(u,v))$ therefore needs to be done efficiently. We used the following methodology: A single pass is performed over all documents in $\mathcal{D}$. For each document D, output all pairs of keywords that appear in D after stemming and removal of stop words. Since $A(u)$ also needs to be computed, for each keyword $u \in D$, (u,u) is also included as a keyword pair appearing in D. At the end of the pass over $\mathcal{D}$ a file with all keyword pairs is generated. The number of times a keyword pair (u,v) appears in this file is exactly the same as $A(u,v)$. This file is sorted lexicography (using external memory merge sort) such that all identical keyword pairs appear together in the output. All the triplets are generated by performing a single pass over the output sorted file. Table 1 presents sizes of two of the keyword graphs (each for a single day) after stemming all keywords and removing stop words.

Given graph G we first infer statistically significant associations between pairs of keywords in this graph. Intuitively if one keyword appears in n_1 fraction of the posts and another keyword in a fraction n_2 we would expect them both to occur together in $n_1 n_2$ fraction of posts. If the actual co-occurrence percent deviates significantly from this expected value, the assumption that the two keywords are independent is questionable. This effect can be easily captured by the χ^2 test:

$$\chi^2 = \frac{(E(uv) - A(uv))^2}{E(uv)} + \frac{(E(\bar{u}v) - A(\bar{u}v))^2}{E(\bar{u}v)} + \frac{(E(u\bar{v}) - A(u\bar{v}))^2}{E(u\bar{v})} + \frac{(E(\bar{u}\bar{v}) - A(\bar{u}\bar{v}))^2}{E(\bar{u}\bar{v})} \qquad (1)$$

In this formula, $A(uv)$ is the number of times keywords u, v appear in the same post (document). $E(uv)$ is the expected number of posts in which u and v co-occur under the independence assumption. Thus, $E(uv) = \frac{A(u)A(v)}{n}$ where $A(u)$ ($A(v)$) is the total number of times keyword u appears in posts and n is the total number of posts. Similarly, $A(\bar{u})$ is the number of posts not containing keyword u. The value χ^2 has a chi-squared distribution. From standard tables, we identify that only 5% of the time does χ^2 exceed 3.84 if the variables are independent. Therefore, when $\chi^2 > 3.84$ we say that u and v are correlated at the 95% confidence level. This test can act as a filter omitting edges from G not correlated according to the test at the desired level of significance. Note that this test can be computed with a single pass of the edges of G.

While this test is sufficient to detect the presence of a correlation, it cannot judge its strength. For example, when u and v are indeed correlated their χ^2 values will increase as the number of data points (number of posts in our case, $n = |\mathcal{D}|$) grows. The correlation coefficient ρ, is a measure of the strength of correlation. It is defined as follows:

$$\rho(u,v) = \frac{(\sum_i (A_i - \mu_u)(B_i - \mu_v))}{n\sqrt{\sigma_u^2 \sigma_v^2}} \qquad (2)$$

where μ_u is the mean of the number of times keyword u appears in the document collection (n documents in total), that is $\frac{A(u)}{n}$, σ_u^2 is the variance of the appearance of u in the posts and A_i is 1 if and only if post i contains u. It is evident that ρ is between -1 and 1, and it is zero if u and v are independent. The correlation coefficient is important because it is often the case that we have enough data to find weak but significant correlations. For example once an hour posts might contain two terms together. With enough data over a day, the χ^2 test will (correctly) assess non-independence. The correlation coefficient however will report a weak correlation. For all edges that survive the χ^2 test, we compute the correlation coefficient between the incident vertices. This computation can again be conducted efficiently by re-writing Formula 2 as

$$\rho(u,v) = \frac{nA(u,v) - A(u)A(v)}{\sqrt{(n - A(u))\,A(u)}\sqrt{(n - A(v))\,A(v)}} \qquad (3)$$

using the fact that $\sum A_i^2 = \sum A_i$.

Given graph G (excluding the edges eliminated by the χ^2 test), assume we have annotated every remaining edge with the value of ρ indicating the strength of the correlation. This graph can be further reduced by eliminating all edges with values of ρ less than a specific threshold. Since our problem is binary (a keyword either appears in the post or not) focusing on edges with $\rho > 0.2$ will further eliminate any non truly correlated vertex pair, making the probability of a false (non correlated pair) being included very small [6]. These correlations are important since the strong ones offer good indicators for query refinement (e.g., for a query keyword we may suggest the strongest correlation as a refinement) and also track the nature of ‘chatter’ around specific keywords.

Let G' be the graph induced by G after pruning edges based on χ^2 and ρ. Observe that graph G' contains only edges connecting strongly correlated keyword pairs. We aim to extract keyword clusters of G'. Although we can formally cast our problem as an optimization problem for graph clustering [1, 8], adopting any of the known approximation algorithms is impossible as such algorithms are of high polynomial complexity. Running any such algorithm on the problems of interest in this study is prohibitive. Moreover, the access patterns of such approximation algorithms require the entire graphs to be in memory and do not have efficient secondary storage realizations. For this reason, we propose a simple and (as we will demonstrate) effective heuristic algorithm to identify such clusters. Our algorithm is fast, suitable for graphs of the scale encountered in our setting and efficient for graphs that do not fit in memory. We empirically evaluate the quality of the clusters we identify in Section 5.

Our algorithm identifies all articulation points in G' and reports all vertices (with their associated edges) in each biconnected component as a cluster. An articulation point in a graph is a vertex such that its removal makes the graph disconnected. A graph with at

Algorithm 1 Algorithm to Identify Biconnected Components

```
Initialize $time = 0$ and $un[u] = 0$ for all $u$
Algorithm Art($u$)
$time \leftarrow time + 1$
$un[u] \leftarrow time$
$low[u] \leftarrow time$
**for** each vertex $w \neq u$ such that $(u, w) \in E$ **do**
    **if** $un[w] < un[u]$ **then**
        add $(u, w)$ to Stack
    **end if**
    **if** $un[w] = 0$ **then**
        call Art($w$)
        $low[u] \leftarrow min\{low[u], low[w]\}$
    **end if**
    **if** $low[w] \geq un[u]$ **then**
        Pop all edges on top of Stack until (inclusively) edge
        $(u, w)$, and report as a biconnected component
    **else**
        $low[u] \leftarrow min\{low[u], un[w]\}$
    **end if**
**end for**
```

least two edges is biconnected if it contains no articulation points. A biconnected component of a graph is a maximal biconnected graph. Thus, the set of clusters we report for G' is the set of all biconnected components of G' plus all trees connecting those components. The underlying intuition is that nodes in a biconnected component survived pruning, due to very strong pair-wise correlations. This problem is a well studied one [7]. We adopt algorithms for its solution and demonstrate via experiments that they are memory efficient. Let G_π be a depth first tree of G'. An edge in G' is a back edge iff it is not in G_π. The root of G_π (the vertex from which we initiated the depth first traversal) is an articulation point of G' if it has at least two children. A non-root vertex $u \in G_\pi$ is an articulation point of G' if and only if u has a child w in G_π such that no vertex in the subtree rooted at w (in G_π), denoted $subtree(w)$, is connected to a proper ancestor of u by a back edge. Let $un[w], w \in G_\pi$ be the order in which w is visited in the dfs of G'. Such a dfs traversal can be performed efficiently, even if the graph is too large to fit in memory [5, 4]. We define:

$$low[w] = \min \begin{cases} un[w] & \\ un[x]: & \begin{array}{l} x \; is \; joined \; to \; subtree(w) \; via \; a \\ back \; edge \; where \; x \; is \; a \; proper \\ ancestor \; of \; w \end{array} \end{cases}$$

A non root vertex $u \in G_\pi$ is an articulation point of G' if and only if u has a child w such that $low[w] \geq un[u]$. Algorithm 1 presents pseudocode for the algorithm to identify all biconnected components of G'. A similar technique can be used to report all articulation points of G'. The algorithm as presented in the pseudo code requires as many accesses to disk as the number of edges in G'. Since in our graphs we expect $|E| >> |V|$, using the techniques of [5] we can run it in $O((1 + |V|/M)scan(E) + |V|)$ I/Os, where M is the size of available memory. Since the data structure in memory is a stack with well defined access patterns, it can be efficiently paged to secondary storage if its size exceeds available resources. In our experiments, presented in Section 5, this was never the case.

EXAMPLE 1. *Figure 3 shows an example of applying the Algorithm 1 to G' in (a). The DFS tree, G_π is shown in (b) with the final $un(u)$ and $low(u)$ values. Back edges (c, a) and (f, d) (shown as dashed edges in G_π) lead to $low(u)$ being updated during the backtracking for all parent nodes. Internal nodes b and d are articulation points. The biconnected components of G' are shown in (c).*

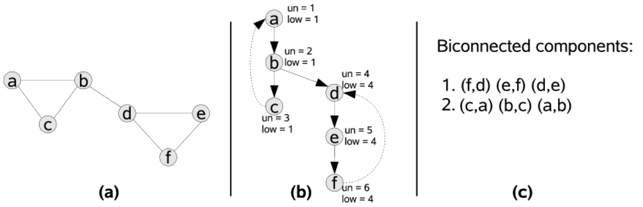

Figure 3: (a) Example graph G' (each vertex represents a keyword), (b) DFS tree G_π, (c) Biconnected components of G'.

4. STABLE CLUSTERS

Let $t_1, \ldots t_m$ be (without loss of generality) m successive temporal intervals. Let $T_1 \ldots T_m$ be the number of clusters identified for each of the intervals $t_1 \ldots t_m$ using the algorithm in Section 3. Let c_{ij} be the clusters identified $1 \leq i \leq m, 1 \leq j \leq T_i$. Analysis of the affinity (e.g., overlap) of the keywords in these clusters across the temporal intervals can provide very valuable information. For example, a cluster of keywords that always appear together across the m temporal intervals probably points to an event that triggered increased use of the keywords in the consecutive temporal intervals by enough people (bloggers) to force a persistent (stable) cluster across the intervals. Similarly, clusters that appear in some of the temporal intervals, or clusters that appear for a few intervals then vanish and appear again, might also be of interest as they point to events that triggered increased use of the keywords for a few intervals.

Let $G_1, \ldots G_m$ be the sets of clusters produced for each temporal interval $1 \leq i \leq m$. Given two clusters $c_{kj}, c_{k'j'}, k \neq k'$, we can quantify the affinity of the clusters by functions measuring their overlap. For example, $|c_{kj} \cap c_{k'j'}|$ or $Jaccard(c_{kj}, c_{k'j'})$ are candidate choices. Other choices are possible taking into account the strength of the correlation between the common pairs of keywords. Our framework can easily incorporate any of these choices for quantifying cluster affinity. We consider clusters with affinity values greater than a specific threshold θ ($\theta = 0.1$) to ensure a minimum level of keyword persistence. Given the clusters G_i, we form a graph $\mathcal{G}$ by evaluating the affinity between select pairs of $G_i, G_j, i \neq j, i \leq j + g + 1, 1 \leq i, j \leq m$. The choice of pairs to compute, dictates the structure and connectivity of $\mathcal{G}$. We refer to the value of g as a *gap* for a specific construction of $\mathcal{G}$. Gaps are useful to account for clusters (chatter) that are persistent for a few intervals, then vanish and appear again (see Figure 4 for example). $\mathcal{G}$ is a weighted graph with edge weights equal to the affinity of the clusters incident to the edge. For any path in $\mathcal{G}$ we define the *weight* of the path by aggregating the weights of the edges comprising the path. Notice that the types of paths existing in $\mathcal{G}$ is a construction choice. Graph $\mathcal{G}$ may range from an m-partite graph to a fully connected graph, depending on the choice of g. $\mathcal{G}$ is an undirected graph.

PROBLEM 1 (kl-STABLE CLUSTERS). *Given a graph $\mathcal{G}$ constructed using a specific affinity function, we define the problem of stable clusters as the problem of identifying the k paths of length l of highest weight.*

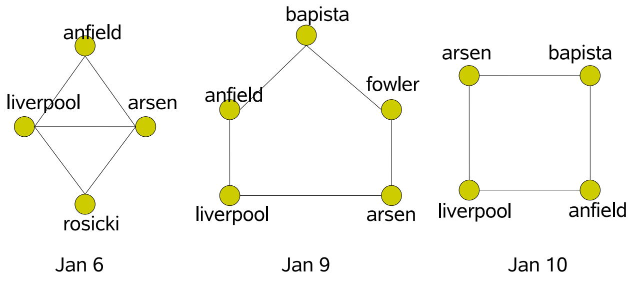

Figure 4: Example of stable clusters with gaps. Three clusters are shown for Jan 6, 9 and 10 2007 and no clusters were discovered for Jan 7 and 8 (related to this topic). These clusters correspond to the following event: English FA cup soccer game between Liverpool and Arsenal with double goal by Rosicky at Anfield on Jan 6. The same two teams played again on Jan 9, with goals by Baptista and Fowler. Note that the keywords are stemmed.

A variant of the above problem is the following:

PROBLEM 2 (NORMALIZED STABLE CLUSTERS). *Given a graph $\mathcal{G}$ constructed using a specific affinity function we define the problem of normalized stable clusters as the problem of identifying the k paths of length at least l_{min} of the highest weight normalized by their lengths.*

In order to construct the graph $\mathcal{G}$ for a set of clusters $G_1, \ldots G_m$, each computed for an interval $t_1 \ldots t_m$ fixing a gap value $g \geq 0$, we have to compute the affinity between clusters in $G_i, G_j, i \leq j + g + 1, 1 \leq i, j \leq m, 0 \leq g \leq m - 1$. Assuming T_i clusters for each interval t_i the suitable affinity predicate (e.g., intersection size) can be computed between each pair of clusters of the corresponding intervals, assuming the clusters for the pair of intervals fit in memory. If T_i (and the associated cluster descriptions in terms of their keywords) is too large to fit in memory, we can easily adapt technology to quickly compute all pairs of clusters for which the affinity predicate is above some threshold. Notice that each cluster description is a set of keywords. Thus, the problem is easily reduced to that of computing similarity (affinity) between all pairs of strings (clusters) for which the similarity (affinity) is above a threshold. Efficient solutions for conducting such computations for very large data sets are available and can easily be adapted [11].

Given graph $\mathcal{G}$, we now present our solutions to the kl stable clusters problem. Note that the top-k paths produced may share common subpaths which, depending on the context, may not be very informative from an information discovery perspective. Variants of the kl-stable cluster problem with additional constraints are possible to discard paths with the same prefix or suffix. For simplicity, we focus on the original problem and present three solutions that can later be adapted for more refined variants of the problem. The three associated algorithms are: (a) an algorithm based on breadth first search on $\mathcal{G}$, (b) an algorithm based on depth first search, and (c) an adaptation of the well known threshold algorithm [14]. Our focus is on cases for which the number of clusters and their associated descriptions for all temporal intervals are too large to fit in memory and we propose efficient solutions for secondary storage.

4.1 The Cluster Graph

Let $\mathcal{G}$ denote the cluster graph. Figure 5 shows an example cluster graph over 3 temporal intervals. Each interval has 3 nodes (keyword clusters). Edges between two nodes indicate that they have a non-zero affinity. While conceptually the model has undirected edges, we add a source node at the beginning and a sink at the end, and make edges directed. Each edge has a weight in the range $(0, 1]$[1]. Thus the length of an edge over a single gap of length g is considered to be $g + 1$. Edge length is defined as the length of the temporal interval between two participating nodes. For example, in Figure 5, the length of edge $c_{11}c_{21}$ and $c_{13}c_{22}$ is one, while that of $c_{11}c_{32}$ is two. The gap size is selected as $g = 1$ in this example, and therefore all edges have length less than or equal to $g + 1 = 2$. The length and weight of edges connecting source or sink with other nodes is assumed to be zero.

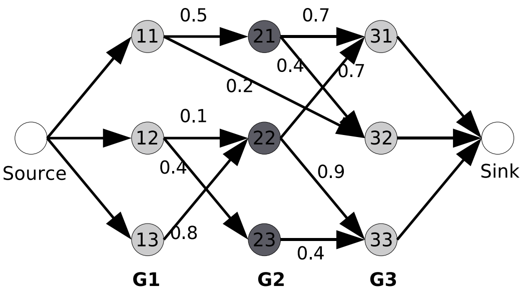

Figure 5: An example of cluster graph for three temporal intervals, each with three clusters. The maximum gap size is chosen to be $g = 2$. Edges are labeled with associated weights (affinity between clusters).

Let $G_i = \{c_{i1} \ldots c_{i|T_i|}\}$ be the clusters at interval t_i. We refer to a node c_{ij} as a child of another node $c_{i'j'}$ if there is an edge between the two nodes and $i > i'$. In this case, $c_{i'j'}$ is a parent of c_{ij}. Let $interval(c)$ be the index of the temporal interval to the cluster to which c belongs. For example if $c \in G_i$, $interval(c) = i$.

4.2 Breadth First Search

We first present a breadth first search based algorithm for detecting stable clusters. At the end of the algorithm we seek to find the top-k paths with highest weights of length l. As the algorithm progresses, we annotate each node in the graph with up to l heaps, each of size less than or equal to k. For a node c_{ij}, we denote this data structure as h^x_{ij}, for $1 \leq x \leq l$, each of which represents top-k (or fewer) highest weighting subpaths of length x ending at c_{ij}. Observe that annotating each node of an arbitrary graph with such information is a non-trivial task requiring many random disk I/Os. We take advantage of the special structure of the graph in our case, which is very similar to an n-partite graph (except for the gaps). Such graphs have a nice property that a node from G_i cannot have a parent from a temporal interval before $i - g - 1$, where g is the size of the maximum gap allowed. This means that if all nodes from temporal intervals $\{i - g - 1, \ldots, i - 1\}$ can be kept in memory, subpaths ending at all nodes from G_i can be computed without performing any I/O.

For all the nodes belonging to G_1, all the associated heaps are initialized to be empty. To compute heaps for a node $c_{ij} \in G_i$, all nodes from the previous $g + 1$ intervals are read in memory along with their l heaps. After reading all the nodes from the previous $g + 1$ intervals, nodes from G_i are read one after the other. For

[1] Some affinity functions such as intersection do not guarantee weights to be in the range $(0, 1]$. In such cases, the maximum score seen so far can be maintained to normalize all weights to the range $(0, 1]$.

Algorithm 2 BFS based algorithm for kl-clusters

```
INPUT $\mathcal{G} = \{G_1, \ldots, G_m\}$, $l$, $k$, $g$
Initialize $H = \phi$, heap of size $k$
for $i$ = 2 to $m$ do
    Read $G_{i'}$ in memory, $i - g - 1 \le i' \le i - 1$
    for $c_{ij} \in G_i$ do
        Initialize $h^x_{ij} = \phi$, heap of size $k$, $1 \le x \le l$.
        for $c_{i'j'} \in parents(c_{ij})$ do
            $len = i - i'$ {comment: since $c_{i'j'} \in G_{i'}$, this is the
              length of the edge $c_{i'j'}c_{ij}$}
            for $x$ = 1 to $l - len$ do
                for $\pi \in h^x_{i'j'}$ do
                    $\pi' = append(\pi, c_{i'j'}c_{ij})$
                    check $\pi'$ against $h^{x+len}_{ij}$
                    check $\pi'$ against $H$ {"check" operation on $\pi'$
                      against a fix-sized heap checks for the inclusion
                      of $\pi'$ in the heap}
                end for
            end for
        end for
    end for
    save $c_{ij}$ along with $h^x_{ij}$ to disk
end for
return $H$
```

each node $c_{ij} \in G_i$, all its parents are probed (which are already in memory) to update its associated heaps h^x_{ij}. Consider the example cluster graph presented in Figure 5 with $l = 2$ and $k = 2$. Computing heaps for nodes from the second temporal interval, all nodes in G_1 are read in memory. Each node from the second interval will have only a single heap associated with it, since there are no paths of length two ending there. The heaps for nodes in G_2 are:

$$h^1_{21} = \{c_{11}c_{21}\}; h^1_{22} = \{c_{12}c_{22}, c_{13}c_{22}\}; h^1_{23} = \{c_{12}c_{23}\}$$

Computing heaps for nodes from G_3, all nodes from G_1 and G_2 are kept in memory. Since there are three paths of length 2 reaching c_{31}, only the best two are retained. Since the weight of $c_{12}c_{22}c_{31}$ (which is 0.8) is less than that of $c_{13}c_{22}c_{31}$ (1.5) and of $c_{11}c_{21}c_{31}$ (1.2), it is discarded. Although c_{11} is a parent of c_{32} (with direct edge between the two), due to the gap, $c_{11}c_{32}$ is an edge of length two. Thus,

$$\begin{gathered} h^1_{31} = \{c_{21}c_{31}, c_{22}c_{31}\}; h^1_{32} = \{c_{21}c_{32}\}; \\ h^1_{33} = \{c_{22}c_{33}, c_{23}c_{33}\}; h^2_{31} = \{c_{11}c_{21}c_{31}, c_{13}c_{22}c_{31}\}; \\ h^2_{32} = \{c_{11}c_{21}c_{32}, c_{11}c_{32}\}; h^2_{33} = \{c_{13}c_{22}c_{33}, c_{12}c_{22}c_{33}\} \end{gathered}$$

$\mathcal{G}$ is stored in the form of an adjacency matrix so that for any cluster c_{ij} we can easily retrieve $parents(c_{ij})$ the set of all clusters at intervals $t_{i'} < t_i, i' \in [i-g-1, i-1]$ with edges incident to c_{ij}. As an invariant assume that $h^x_{i'j}, i' \in [i-g-1, i-1]$ have been computed while building heaps for nodes from G_i. We compute $allpaths(c_{ij}, x)$ as the set of all paths of length x ending at c_{ij} that can be derived from information maintained in $parents(c_{ij})$. More formally:

$$allpaths(c_{ij}, x) = \{append(\pi, c_{i'j'}c_{ij}) \mid c_{i'j'} \in parents(c_{ij}) \text{ and } \pi \in h^{x-i+i'}_{i'j'}\}$$

where $append(\pi, c_{i'j'}c_{ij})$ represents the path obtained by appending the edge $c_{i'j'}c_{ij}$ at the end of subpath π. Thus,

$$h^x_{ij} = \text{top-}k\ paths\ among\ allpaths(c_{ij}, x)$$

Computing h^x_{ij} using $parents(c_{ij})$ can be conducted in a straightforward way by considering each possible element of $allpaths(c_{ij}, x)$. In practice, h^x_{ij} can be computed directly using a heap without maintaining the intermediate result $allpaths(c_{ij}, x)$. For each c_{ij} in memory we need to maintain at most kl subpaths of highest weight ending at c_{ij}. These are the best (highest weight) paths of lengths $1, 2, \ldots, l$, maintained using heaps, one for each length. This means that $allpaths(c_{ij})$ is updated for each value of $x \in [1, l]$. Algorithm 2 presents pseudocode.

Maintaining the solution to the kl-stable clusters problem is conducted by maintaining a heap H during the execution of the algorithm in which the k highest weight paths of length exactly l are identified. When a new interval t_{i+1} is encountered, G_{i+1} is brought into memory and G_{i-g-1} is discarded. This computation is performed for $1 \le i \le m$. After G_m is encountered, the solution to the kl-stable clusters problems is located in H. In the running example of Figure 5, whenever a new path of length 2 is discovered, it is checked against the global heap H. Six paths, from h^2_{31}, h^2_{32}, and h^2_{33}, are checked for candidacy in H in this example. The size of H is bounded by $k = 2$. In the end, the best two paths are identified as $c_{13}c_{22}c_{31}$ and $c_{13}c_{22}c_{33}$.

If $l = m - 1$, i.e., when finding full paths from t_1 to t_m, we need not maintain heaps h^x_{ij} for each $1 \le x \le l$. Instead, maintaining one heap per node suffices. For a node $c_{ij} \in G_i$, only h^i_{ij} needs to be computed. This reduces the computation by a factor of l for this special case.

The algorithm as described requires enough memory to maintain all clusters for $g + 1$ intervals in memory. Under this assumption the algorithm requires a single pass over all G_i, $1 \le i \le m$. The total number of I/Os required is linear in the number of intervals considered and linear in the total number of clusters. Assume that accommodating clusters for $g + 1$ intervals in memory requires an amount equal to M_{req} but only M memory units are available ($M < M_{req}$). In order to compute heaps for all clusters in the i-th interval M_{req}/M passes will be required. This situation is very similar to block-nested loops.

Claim 1. The BFS based algorithm described outputs the correct set of highest weighting top-k paths in the cluster graph $\mathcal{G}$.

4.3 Depth First Search

We present a solution to the kl-stable clusters problem based on a depth first search (DFS) traversal of the cluster graph $\mathcal{G}$ suitably adapted for secondary storage. DFS can be performed using a stack. We show that the size of the stack will be bounded by m, the number of temporal intervals. The complexity of this algorithm in the worst case is linear to the number of edges in the graph, but practically can be much less due to pruning. Unlike the BFS algorithm presented in the previous subsection, this algorithm requires significantly less memory to operate, but performs much more I/O.

For each node (cluster) c_{ij}, we maintain a list of its children (nodes in $G_{i'}$, $i' \in [i+1, i+g+1]$ incident to c_{ij}) as $children(c_{ij})$ which is precomputed during the generation of $\mathcal{G}$. Also we maintain a global top-k list as a heap (containing the current k paths of length l of highest weight) and a stack, both initialized to be empty in the beginning. As the algorithm progresses, we will maintain the following information with each node c_{ij} (on disk):

- One flag denoting whether the node has already been visited. If the flag is set, we are confident that all descendants of the node have been considered. If not, its descendants may or may not have been traversed.
- If the objective is to find full paths (of length $m - 1$), one number denoting the aggregate weight of the highest weight

path from the source to that node. If the objective is to find subpaths of length l, one number for each x, $\max(1, l+i-m) \le x \le \min(l, i-1)$, denoting the aggregate weight of the highest weighting path of length x ending at that node. We represent this data structure by $maxweight(c_{ij}, x)$ for paths of length x, and use this data structure for pruning.

- If the objective is to find full paths of length $m-1$, a single heap of top-k best (highest weighting) paths starting at that node is maintained. If the objective is to find subpaths of length l, a heap for each $\max(1, l+i-m) \le x \le \min(l, i-1)$, containing top-$k$ best paths of length x starting at that node is maintained. We denote this data structure by $bestpaths(c_{ij}, x)$ for paths of length x. Contrasting this case with the case of the same data structure in the BFS algorithm, we note that paths contained in this case start at c_{ij} (instead of ending at c_{ij}). The size of $bestpaths$ for any node is bounded by k when seeking full paths of length $m-1$, and $k \cdot l$ in the case of subpaths of length l.

The algorithm performs a depth first search on the input cluster graph. Pseudocode for this algorithm is presented in Algorithm 3.

We provide an operational description of the algorithm. Start by pushing the $source$ node along with $children(source)$ onto the stack. Now iteratively do the following: Take the top element c from the stack, remove an element c' from the list $children(c)$. Check if c' is already visited. If yes, update $bestpaths(c)$ using $bestpaths(c')$ as described later, and discard c'. If not, mark c' as visited and push it on the stack. Update $maxweight(c', x)$ using $maxweight(c, x)$ for each x.

$$maxweight(c', x) = \max(maxweight(c', x),$$
$$maxweight(c, x - length(cc')) + weight(cc'))$$

where $length(cc')$ is the length of the edge between c and c'. The following pruning operation can be conducted (when searching for subpaths of length l): If for all $\max(1, l + interval(c') - m) \le x \le \min(l-1, interval(c') - 1)$,

$$maxweight(c', x) + l - x < \text{min-}k,$$

where min-k is the minimum weight among all paths in H (the current top-k), remove c' from the stack. Also unmark the visited flag for all the nodes in the stack (including c'). This is based on the observation that, given the current information about the weight of the path from the source to c', it is unlikely that any of the paths containing cc' can be in the top-k . Therefore we postpone considering descendants of c' until we find (if it exists) a higher weighting path from the source to c'. We unmark the visited flag of all nodes on the stack since the guarantee that all descendants have been traversed no longer holds true for them. Therefore, pruning assumes that all edge weights are between $(0, 1]$ (which is true for some affinity measures like Jaccard; normalization is required for others e.g., intersect).

If c is at the top of the stack such that $children(c) = \phi$, i.e., all children of c have been considered (either traversed or discarded by pruning), remove c from the stack. Let c' be the next element on the stack. Update $bestpaths(c')$ using $bestpaths(c)$ (this is actually back tracking an edge in DFS).

To update $bestpaths(c)$ using information about one of its children c'; first find all possible paths starting at c by augmenting the edge cc' with all paths in $bestpaths(c', x)$, and add them to $bestpaths(c, x + length(cc'))$, for all $x + length(cc') \le l$. Now prune $bestpaths(c, x + length(cc'))$ so that it does not contain more than k paths. When a new path π of length l is added to $bestpaths(c, l)$ for some node c, π is also checked against the global top-k heap for inclusion.

Algorithm 3 DFS based algorithm for kl-clusters

```
INPUT $\mathcal{G} = \{G_1, \ldots, G_m\}$, $l$, $k$, $g$
initialize $H = \phi$, heap of size $k$
initialize $stack = \phi$
push $(source, children(source))$ to $stack$
while $stack$ is not empty do
    $(c, children(c))$ = peek from $stack$ {peek operation returns the top element from the stack without removing it}
    if $children(c)$ is not empty then
        $c'$ = remove top element from $children(c)$
        read from disk information associated with $c'$
        if $c'$ is visited then
            update $bestpaths(c, x)$ using info from $c'$, $x \le l$
            for each newly added path $\pi$ in $bestpaths(c, l)$ do
                check $\pi$ against $H$ {“check” operation on $\pi$ against a fix-sized heap checks for the inclusion of $\pi$ in the heap}
            end for
        else
            mark $c'$ visited, and push $(c', children(c'))$ on stack
            update $maxweight(c', x)$ using $maxweight(c, x)$
            if $CanPrune(c')$ then
                unmark visited flag for all nodes in $stack$
                pop $c'$ from $stack$
                save $c'$ and associated information to disk
            end if
        end if
    else
        pop $c$ from $stack$ and save on disk
        $(c', children(c'))$ = peek from $stack$
        update $bestpaths(c', x)$ using info from $c$, $x \le l$
        for each newly added path $\pi$ in $bestpaths(c', l)$ do
            check $\pi$ against $H$
        end for
    end if
end while
output $H$
DEFINE $CanPrune(c')$
min-$k$ = minimum score in $H$
for $x = \max(1, l + interval(c') - m)$ to $\min(l - 1, interval(c') - 1)$ do
    if $maxweights(c', x) + l - x \ge$ min-$k$ then
        return false
    end if
end for
return true
```

The size of the stack is at most m entries during the execution of this algorithm. When the algorithm terminates, the global top-k heap H will contain the required result. Furthermore, each node will be annotated with a list of top-k $bestpaths$ starting at that node.

Addition of each node to the stack requires one random I/O to read the associated data structures from disk. Updating these data structures and marking/unmarking of nodes takes place in main memory. Removal of a node requires an additional random I/O for writing back associated data structures. In the absence of the pruning condition, the number of read operations is bounded by the number of edges, and the number of write operations is bounded by the number of nodes in the graph. With every pruning operation, in the worst case, both these numbers can increase by an amount equal to the size of the stack at that time. But pruning is also expected to discard many nodes and edges without actually calculating path lengths for them, which can reduce I/Os significantly.

EXAMPLE 2. *We show the execution of the algorithm over the cluster graph presented in Figure 5 for $k = 1$ and $l = 2$. In this example, since we are required to find full paths ($l = 2$), only one heap and one $maxweight$ structure is associated with each node. Table 2 shows the order in which the nodes are considered and actions taken at those steps. Observe that pruning takes place when c_{22} is first explored. However c_{22} is explored further when it is reached again via c_{13}. The final result is printed as $\{c_{13}c_{22}c_{33}\}$. Note that other execution orders are also possible, depending on how the children lists for each node are sorted.*

Claim 2. The DFS based algorithm described outputs the correct set of highest weighting top-k paths in the cluster graph $\mathcal{G}$.

For effective pruning, it is important that paths of high weights are considered early. For this reason, as a heuristic, while precomputing the list of children for all nodes, we sort them in the descending order of edge weights. Formally if $c_1, c_2 \in children(c)$ and $weight(c_1, c) > weight(c_2, c)$, then c_1 precedes c_2 in the list $children(c)$. This will ensure that the children connected with edges of high weight are considered first. It must be noted that this heuristic is for efficient execution, and correctness of the algorithm is unaffected by it.

4.4 Adapting the Threshold Algorithm

The Threshold Algorithm (TA) [14] can also be adapted to find full paths of length $m - 1$ in $\mathcal{G}$. For each pair of temporal intervals t_i and $t_{i'}$, $|i - i'| \leq g + 1$, one list of edges is maintained. These lists are sorted in descending order of edge weights.

We read edges from sorted lists in a round robin fashion and maintain a global heap H for intermediate top-k results. When an edge $c_{ij}c_{i'j'}$ ($i < i'$) is encountered, we perform random seeks to lookup all paths containing the edge. Let d be the maximum out-degree in the graph $\mathcal{G}$. Unlike the vanilla-TA, where each attribute belongs to exactly one tuple, in this case there may be multiple paths that contain $c_{ij}c_{i'j'}$. Perform random seeks in edge lists to find all the paths that start with $c_{i'j'}$, and all the paths that end at c_{ij} to construct all paths containing $c_{ij}c_{i'j'}$. Check all these paths for inclusion in H, and discard them if they fail to qualify. Terminate when the score of the lowest scoring path in the buffer H falls below that of the virtual tuple. The virtual tuple is the imaginary path consisting of the highest scoring unseen edge from each list.

A path may be discovered more than once in the above algorithm. As an optimization to reduce I/O, two additional hash tables, $startwts$ and $endwts$, can be maintained. For a node c, $startwts(c_{ij})$ ($endwts(c_{ij})$) records the aggregate weight of the highest weighting path starting (ending) at c_{ij}. These hash tables are initialized to be empty at start, and are updated as the algorithm progresses. When all paths starting (ending) at a node c_{ij} are computed by performing random probes, $startwts(c_{ij})$ ($endwts(c_{ij})$) is updated. When an edge $c_{ij}c_{i'j'}$ is read from the edge list, and if $startwts(c_{i'j'})$ and $endwts(c_{ij})$ are available in the hash tables, an upper bound on weight of all paths containing $c_{ij}c_{i'j'}$ can be computed without any I/O. This upper bound can be compared with the score of lowest scoring path in H, and $c_{ij}c_{i'j'}$ can be discarded without performing any further computation if the former is smaller. This pruning can result in large savings in I/O.

Node	Action taken and Updates
Explored	to $maxweights$ and $bestpaths$
c_{11}	none
c_{21}	$maxweight(c_{21}, 1) = 0.5$
c_{31}	$maxweight(c_{31}, 2) = 1.2$
c_{21}	$bestpaths(c_{21}, 1) = \{c_{21}c_{31}\}$
c_{32}	$maxweight(c_{32}, 2) = 0.9$
c_{21}	none
	$\{c_{32}c_{21}\}$ failed to qualify for $bestpaths(c_{21}, 1)$
c_{11}	$bestpaths(c_{11}, 2) = \{c_{11}c_{21}c_{31}\}$
	and $H = \{c_{11}c_{21}c_{31}\}$
c_{32}	none
	$maxweight(c_{32}, 2)$ remains unchanged at 0.9
c_{11}	none
	$\{c_{32}c_{11}\}$ failed to qualify for $bestpaths(c_{11}, 1)$
c_{source}	none
c_{12}	none
c_{22}	$maxweight(c_{22}, 1) = 0.1$ and
	c_{22} is pruned since min-k=1.2
c_{12}	none
c_{23}	$maxweight(c_{23}, 1) = 0.4$
c_{33}	$maxweight(c_{33}, 2) = 0.8$
c_{23}	$bestpaths(c_{23}, 1) = \{c_{23}c_{33}\}$
c_{12}	$bestpaths(c_{12}, 2) = \{c_{12}c_{23}c_{33}\}$
c_{source}	none
c_{13}	none
c_{22}	$maxweight(c_{22}, 1) = 0.8$
	c_{22} is not pruned this time
c_{31}	$maxweight(c_{31}, 2) = 1.5$
c_{22}	$bestpaths(c_{22}, 1) = \{c_{22}c_{31}\}$
c_{33}	$maxweight(c_{33}, 2) = 1.7$
c_{22}	$bestpaths(c_{22}, 1) = \{c_{22}c_{33}\}$
c_{13}	$bestpaths(c_{13}, 2) = \{c_{13}c_{22}c_{33}\}$
	and $H = \{c_{13}c_{22}c_{33}\}$

Table 2: Example execution of DFS.

If the maximum out-degree in the graph $\mathcal{G}$ is d, this might lead to as many as m^{d-1} random seeks in the absence of gaps ($g = 0$). In the presence of gaps this number can be much higher. Hence this algorithm is not suitable when either of m or d is high. We validate this observation in the experimental results section. Further, this algorithm is restricted to discovery of full paths only and thus requires $l = m - 1$.

4.5 Normalized Stable Clusters

In the previous sections we have presented algorithms for identifying kl-Stable Clusters in $\mathcal{G}$. In this section, we present algorithms for identifying normalized stable clusters. Let $length(\pi)$ define the length of path π. Let $weight(\pi)$ define the aggregate weight (sum of edge weights) for path π. We wish to find top-k paths in $\mathcal{G}$ with the highest normalized weights, $stability(\pi) = \frac{weight(\pi)}{length(\pi)}$. To avoid trivial results, we constrain the paths to be of length at least l_{min}.

In this case we are not required to provide the lengths of paths as input. Pruning paths becomes tricky in the absence of this information. We make the following observation: if a path π can be divided in two parts π_{pre} and π_{curr}, such that $\pi = \pi_{pre}\pi_{curr}$, and stability of π_{pre} is less than that of π_{curr}, irrespective of the suffix (unseen part) π_{suff} to follow, one may drop π_{pre} from the path. Formally, THEOREM 1. *If $\pi_{pre}\pi_{curr}$ is a valid path such that,*

$$stability(\pi_{pre}) \leq stability(\pi_{curr}),$$

then for any possible suffix π_{suff},

$$stability(\pi_{pre}\pi_{curr}) \leq stability(\pi_{pre}\pi_{curr}\pi_{suff})$$
$$\Rightarrow stability(\pi_{pre}\pi_{curr}\pi_{suff}) \leq stability(\pi_{curr}\pi_{suff}).$$

Proof (Sketch) We use the fact that if $a, b, c, d \in \mathcal{R}^+$

$$\frac{a}{b} < \frac{c}{d} \Leftrightarrow \frac{a}{b} < \frac{a+c}{b+d} < \frac{c}{d}$$

Let weights of π_{pre}, π_{curr}, and π_{suff} be w_p, w_c, and w_s; and lengths be n_p, n_c, and n_s. Given $\frac{w_p}{n_p} < \frac{w_c}{n_c}$, it follows that

$$\frac{w_p + w_c}{n_p + n_c} \leq \frac{w_p + w_c + w_s}{n_p + n_c + n_s} \Rightarrow \frac{w_p + w_c + w_s}{n_p + n_c + n_s} \leq \frac{w_c + w_s}{n_c + n_s} \quad \Box$$

For brevity we omit the complete algorithm and describe only modifications to the algorithm presented in Section 4.2. With each node c_{ij}, we need to maintain:

- All paths of length less than l_{min} ending at that node. Let $smallpaths(c_{ij}, x)$ denote this for all paths of length x ending at c_{ij}.
- A list $bestpaths(c_{ij})$ of top scoring paths of length l_{min} or greater ending at that node. This list can be pruned at each node using Theorem 1. A path $\pi_{pre}\pi_{curr} \in bestpaths(c_{ij})$ can be pruned to just π_{curr} if $length(\pi_{curr}) \geq l_{min}$ and $stability(\pi_{pre}) \leq stability(\pi_{curr})$. In words, the prefix can be discarded if its contribution to the stability is less than that of the last l_{min} edges in the path.

The algorithm in this case proceeds in the same way as in Section 4.2. The data structures are updated as follows: to update $smallpaths(c)$ for node c after discovery of a new edge $c'c$ from $c' \in parents(c)$,

$$smallpaths(c, length(c'c)) = smallpaths(c, length(c'c)) \cup \{c'c\}$$

and for $length(c'c) < x < l_{min}$,

$$\begin{aligned} smallpaths(c, x) \;\; = \;\; & smallpaths(c, x) \bigcup \{append(\pi, c'c) \\ & \mid \pi \in smallpaths(c', x - length(c'c))\}. \end{aligned}$$

To update $bestpaths(c)$, first all possible candidates are computed as described below

$$\begin{aligned} bestpaths(c) \;\; = \;\; & bestpaths(c) \\ & \bigcup \{append(\pi, c'c) \mid \pi \in smallpaths(c', l_{min} - length(c'c))\} \\ & \bigcup \{append(\pi, c'c) \mid \pi \in bestpaths(c')\} \end{aligned}$$

After computing all the possible candidates, perform pruning. If $\pi_1, \pi_2 \in bestpaths(c)$ and π_2 is a subpath of π_1, then π_2 can be deleted from $bestpaths(c)$. Also if $\pi_{pre}\pi_{curr} \in bestpaths(c)$, $length(\pi_{curr}) \geq l_{min}$ and $stability(\pi_{pre}) \leq stability(\pi_{curr})$, then delete $\pi_{pre}\pi_{curr}$ and add π_{curr} to $bestpaths(c)$. After updating $bestpaths$, check each newly generated path against the global top-k list of paths for inclusion.

The above algorithm can be used with the DFS framework (presented in Section 4.3) as well. The basic idea is the same, and pruning uses the result of Theorem 1. Details are omitted for brevity.

4.6 Online Version

New data arrive at every time interval. Hence it is important for the algorithms presented to be amenable to incremental adjustment of the data structures. Notice that the BFS based algorithm of Section 4.2 is amenable to such adjustment. Since heaps for each temporal interval are computed separately, when nodes for the next temporal interval G_{m+1} arrive, heaps for them can be computed without redoing any past computation. If the heaps for all the nodes in $\mathcal{G}$ are maintained on disk, a single pass over them is sufficient to compute the global top-k .

The DFS based algorithm in its original form is not an online streaming algorithm since only the *source* is known and the *sink* changes constantly as new data arrives. DFS requires the knowledge of a $sink$ to operate. Observe that the input graph is symmetric. Therefore, $\mathcal{G}$ can be modified by adding the source at the last temporal interval and the sink at the first interval to perform DFS. As the data for new intervals arrive, only the source needs to be shifted (while keeping everything else the same). Therefore, since $bestpaths$ for each node in $\mathcal{G}$ is maintained, DFS can be used in an incremental fashion as new data arrives.

Note that when streaming, both BFS and DFS actually perform the same operations at each iteration. The only difference is the bootstrap process.

5. EXPERIMENTS

In this section we discuss the results of a detailed experimental evaluation comparing our algorithms in terms of performance, and we present qualitative results. We first present results for our cluster generation procedure and then discuss our stable cluster identification algorithms.

5.1 Cluster Generation

In our first experiment we assess the performance of our cluster generation procedure introduced in Section 3. We implemented this algorithm, and we report its performance as the pruning threshold (correlation coefficient) increases. Figure 6 presents the running time of our entire approach for the data set Jan 6 of Table 1. We measure the time required by the entire procedure, namely reading the raw data files, conducting the χ^2 test, pruning based on the correlation coefficient and then running the Art algorithm (Algorithm 1) to find biconnected components. The execution of the Art algorithm is secondary storage based; we only maintain in memory the biconnected component edges in the stack. As ρ increases, time decreases drastically since the number of edges and vertices remaining in the graph decreases due to pruning.

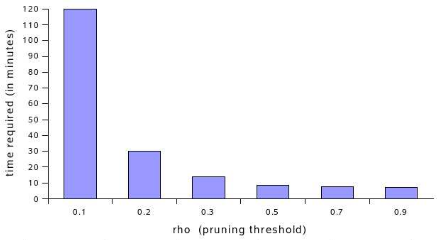

Figure 6: Running time of the Art algorithm.

5.2 Stable Clusters

Our algorithms for stable cluster identification were implemented in Java and executed on a Linux machine with a single 2.0 GHz processor. To capture the effect of I/O on performance accurately, the page cache was disabled during the experiments. Enough memory to keep nodes from the last $g + 1$ intervals was available during experimentation with the BFS based algorithms. In order to be able to vary the parameters of interest in our performance study in a consistent fashion (e.g., number of nodes, average node out degree, etc.) we generated synthetic graphs and we report performance on them. We chose the range of our parameters so as to keep response times manageable, while being able to observe performance trends. The data was generated by first creating a set of nodes of size n for each of the m temporal intervals. For pairs of temporal intervals i and i', $i - i' \leq g + 1$ (where g is the gap size), edges were added as follows: for each node c_{ij} from the first temporal interval, its out degree d_{ij} was selected randomly and uniformly between 1 and $2 \cdot d$, and then d_{ij} nodes were randomly selected from the second temporal interval to construct edges for c_{ij}. Edge weights were selected from $(0, 1]$ uniformly.

Table 3 presents running times in seconds for identifying top-5 full paths (of length $l = m - 1$) comparing the three algorithms. Each temporal interval had $n = 400$ nodes, gap size was selected as $g = 0$, and average out degree of nodes was $d = 5$. Since the TA based algorithm is exponential in m, its running times were significantly higher for $m > 9$ and hence not reported. It can be observed that the BFS based algorithm outperforms DFS by a large margin in terms of running time. But it must be noted that BFS requires significantly larger amounts of memory as compared to the DFS based algorithm. For example, for finding top-3 paths of length 6 on a dataset with $n = 2000$, $m = 9$ and $g = 0$, DFS required less than 2MB RAM as compared to 35MB for BFS.

m =	3	6	9	12	15
BFS	0.65	2.09	4.49	7.95	12.49
DFS	60.3	368.8	754.8	805.94	792.05
TA	0.35	11.11	133.89	$>$ 10 hours	

Table 3: Comparing BFS, DFS and TA based algorithms for different values of m.

Since the TA based algorithm is not applicable to identify subpaths (it requires $l = m - 1$), and due to its high running times on large data sizes when identifying full paths, we focus on the BFS and DFS based algorithms in the sequel. We first explore the sensitivity of the BFS based algorithm for different values of the gap size g in Figure 7. We next show the sensitivity of the same algorithm for different values of average out degree d in Figure 8. In both cases, when either of g or d is increased, the number of edges grows, leading to an increase in the amount of computational effort required. Running times therefore are positively correlated with both g and d, as expected.

Figure 9 demonstrates the scalability of the algorithm; we show the performance of the BFS algorithm as the number of nodes (clusters) for each temporal interval is increased. Observe that the running times are linear in the number of nodes, establishing scalability. The figure shows running times for $m = 25$ and $m = 50$.

Figure 10 presents performance results for the BFS algorithm when seeking top-5 subpaths of length l. The graphs demonstrate that running times increase as l increases due to the larger number of heaps maintained with each node. As expected, running times are linear in the number of nodes per temporal interval.

Figure 11 displays running times of the DFS based algorithm for different values of m and n. Figure 12 shows the sensitivity of the same algorithm for different values of the gap size and average node out degree. As the average out degree or gap size increases, the number of edges increases, directly affecting the running time of the DFS based algorithm. Contrast these results with those of Figure 7 and observe that the DFS based algorithm is more sensitive towards g than the BFS based algorithm. The running times of the DFS based algorithm increases by a factor of more than two as g is increased from 0 to 2, unlike Figure 7, where the effect of an increase in g is milder. Figure 13 shows the performance of the DFS algorithm while seeking subpaths for different values of l. As expected, running times increase with increasing l and n.

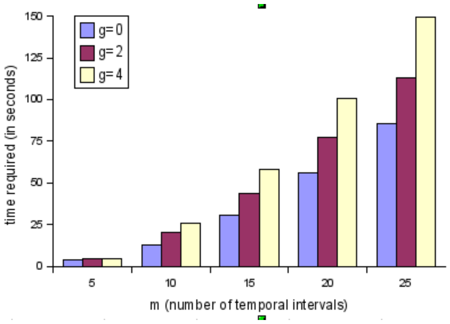

Figure 7: Running times for BFS based algorithm seeking top-5 full paths for different values of g as the number of temporal intervals is increased from 5 to 25. Number of nodes per temporal interval was fixed at $n = 1000$ and average out degree was set to $d = 5$.

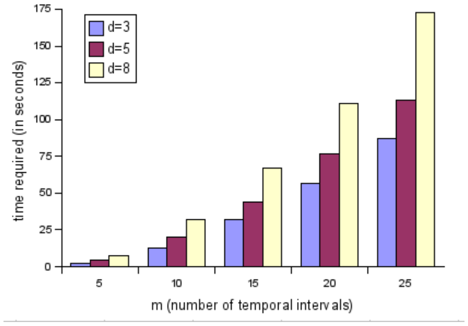

Figure 8: Running times for BFS based algorithm seeking top-5 full paths for different values of d as the number of temporal intervals is increased from 5 to 25. Number of nodes per temporal interval was fixed at $n = 1000$ and gap size was set to $g = 2$.

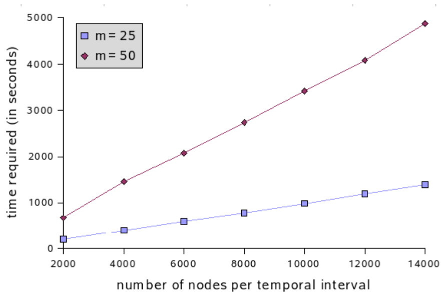

Figure 9: Running times for BFS based algorithm seeking top-5 full paths for two different values of m as the number of nodes in each temporal intervals is increased from 2000 to 14000. Average out degree was set to $d = 5$ and gap size was selected as $g = 1$.

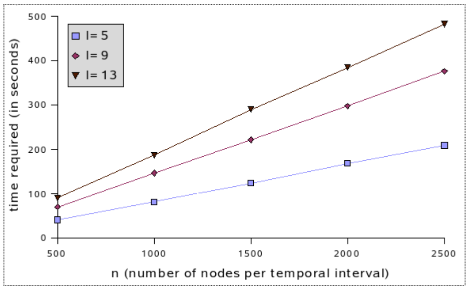

Figure 10: Running times for BFS based algorithms for different values of l over $m = 15$ temporal intervals, as the number of nodes in each temporal intervals is increased from 500 to 2500. Average out degree was set to $d = 5$ and gap size was selected as $g = 2$.

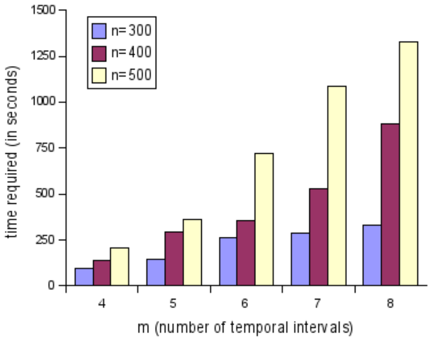

Figure 11: Running times for DFS based algorithms seeking top-5 full paths for different values of m and n. $g = 1$ and $d = 5$ were selected.

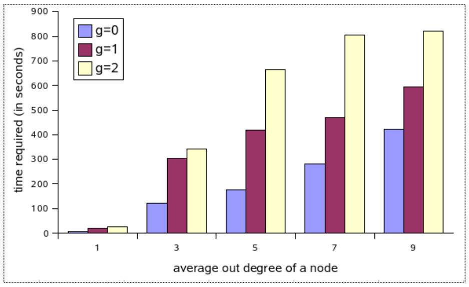

Figure 12: Running times for DFS based algorithms seeking top-5 full paths for different values of g, as the average out degree of nodes is increased. $m = 6$ and $n = 400$ were selected.

Figure 14 displays performance trends for the BFS algorithm seeking normalized stable clusters. Unlike the previous case, where only paths up to length l had to be maintained, the algorithm seeking normalized stable clusters needs to maintain paths of all lengths (those which survive pruning). This leads to an increase in running times as m increases. Experimental results validate this intuition. Running times are positively correlated with l_{min} as larger values of l_{min} results in more paths being maintained with each node. We omit graphs where we vary n, g and d due to space limitations. Trends are as expected, running times increase gracefully with increase in n, g and d.

The impact of k, the number of top results required, on the performance of all the algorithms is minimal, and as k increases running times increase slowly. Experimental results obtained validate that the BFS based algorithm performs better than the DFS based algorithm. The running time of the BFS algorithm increases linearly with an increase in n, while that of DFS increases much more rapidly. This is because the number of edges is proportional to $n \cdot d$. For our problem setting, the running times of the adaptation of TA based algorithms is exponential in m and hence not practical for any realistic problem size. The main advantage of DFS is its low memory requirement. DFS should be used as an alternative to BFS in memory constrained environments.

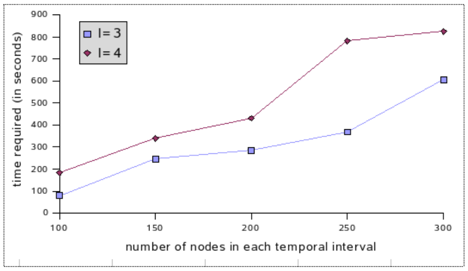

Figure 13: Running times for DFS based algorithms seeking top-5 sub paths of length l for different values of l. $m = 6$, $d = 5$, and $g = 1$ were selected.

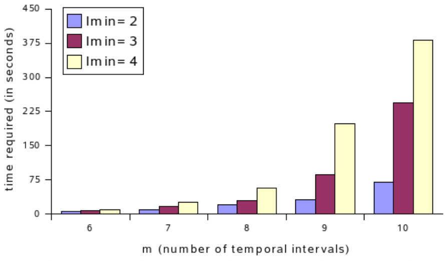

Figure 14: Running times for BFS based algorithms seeking top-5 normalized stable clusters of length greater or equal to l_{min} for different values of m. $n = 400$, $d = 3$, and $g = 0$ were selected.

5.3 Qualitative Results

We have tested our algorithms on large collections of real data obtained from BlogScope. For purposes of exposition, we focus on data obtained for a single week (week on Jan 6 2007) and present results commenting on the output of our algorithms. We set the temporal interval for our construction of graph $\mathcal{G}$ to a day, analyzing seven days. Clusters for a single day were computed using our methodology in Section 3 using $\rho = 0.2$. Around 1100-1500 connected components (clusters) were produced for each day. Affinity between clusters was computed using the Jaccard coefficient and 42 full paths spanning the complete week were discovered.

Table 1 provides data about keyword graph sizes for two days; sizes for the rest of the days were comparable. Figure 1 and Figure 2 show example clusters we were able to identify after the procedure described in Section 3. It is evident that our methodology can indeed capture clusters of keywords with strong pairwise correlations. Taking into account how such correlations are generated (lots of bloggers talking about an event) it is evident that our methodology can identify events that spawn a lot of chatter in the blogosphere. A stable cluster with a path of length 3 and $g = 2$ is shown in Figure 4. Figure 15 presents a stable cluster that persisted

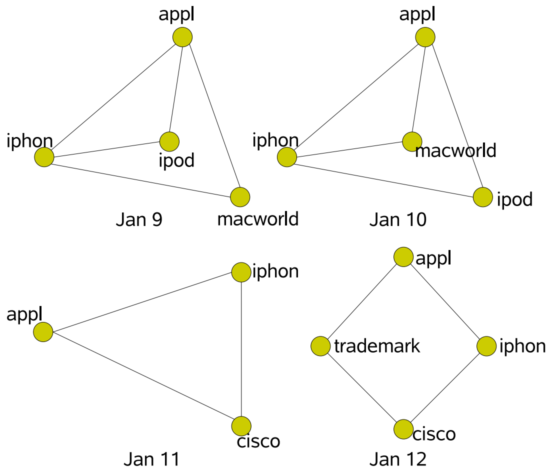

Figure 15: Stable clusters with path of length 3 without gaps. Apple's iPhone was launched on Jan 9 2007. Discussion started with initial talk of iphone's features. Clusters shift on Jan 11 to the Cisco-Apple trademark infringement lawsuit announced on Jan 10. Note that the keywords are stemmed.

for four days without any gaps. An example full length cluster (i.e., that persisted for all seven days) is shown in Figure 16.

Our definition of stable clusters computes cluster similarity between clusters from consecutive time periods only, instead of considering similarity between all pairs in a path. This allows us to capture the dynamic nature of stories in the blogosphere, and their evolution with time. For example notice that in Figure 15 we are able to identify the shift of discussion from iPhone features to the Apple vs Cisco lawsuit related to iPhone. The nature of stable clusters demonstrated in the figures attests that our methodology can indeed handle topic drifts.

6. CONCLUSIONS

In this paper, we formally define and provide solutions for problems related to temporal association of sets of keywords in the blogosphere (or any other streaming text source for that matter). Our technique consists of two steps, (1) generating the keyword clusters, and (2) identifying stable clusters. For both steps we propose efficient solutions. For the problem of kl-stable clusters, we propose three solutions, based on breadth first search, depth first search, and one based on an adaptation of the well-known threshold algorithm. Detailed experimental results are provided, demonstrating the efficiency of the proposed algorithms. Qualitative results obtained using real data from BlogScope are reported that attest to the effectiveness of our techniques.

7. REFERENCES

[1] N. Bansal, A. Blum, and S. Chawla. Correlation Clustering. *Machine Learning*, pages 89–113, 2004.

[2] N. Bansal and N. Koudas. BlogScope: A System for Online Analysis of High Volume Text Streams. In *VLDB*, 2007.

[3] N. Bansal and N. Koudas. Searching the Blogosphere. In *WebDB*, 2007.

[4] A. L. Buchsbaum, M. H. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *SODA*, 2000.

[5] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *SODA*, 1995.

[6] W. Conover. *Practical Non Parametric Statistics*. Wiley, 1980.

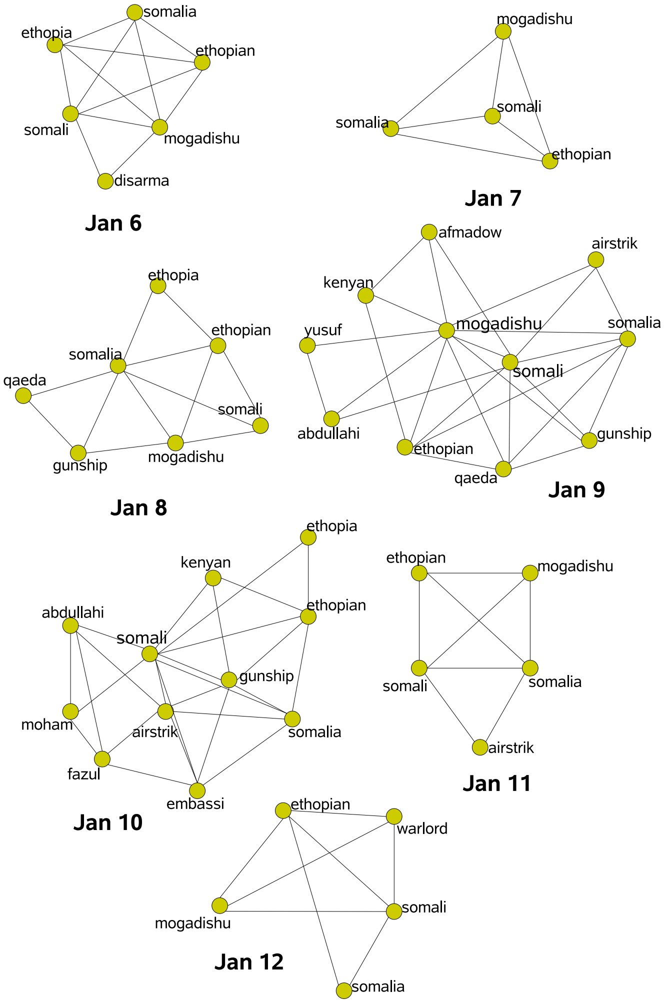

Figure 16: Stable clusters with path of full length. The event corresponds to the battle of Ras Kamboni fought by Islamist militia against the Somali forces and Ethiopian troops. Observe the increase in cluster sizes on Jan 9 after Abdullahi Yusuf arrives in Mogadishu, Somalia on Jan 8 for the first time after being elected as the president. On Jan 8, gunships belonging to the US military had attacked suspected Al-Qaeda operatives in Southern Somalia. Note that the keywords are stemmed.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw Hill and MIT Press, 1990.

[8] G. Flake, R. Tarjan, and K. Tsioutsiouliklis. Graph Clustering and Minimum Cut Trees. *Internet Mathematics*, 2005.

[9] I. Giotis and V. Guruswami. Correlation Clustering with a Fixed Number of Clusters. *ACM SODA*, 2006.

[10] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distributed Computing*, 1998.

[11] N. Koudas, A. Marathe, and D. Srivastava. Approximate String Processing Against Large Databases in Practice. *VLDB*, 2004.

[12] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[13] Perseus - blog survey weblog. *http://www.perseus.com/blogsurvey/blogsurvey.html*.

[14] R. Fagin, A. Lotem and M. Naor. Optimal Aggregation Algorithms For Middleware. *PODS*, June 2001.

[15] State of the Blogosphere - aug 2006. *http://www.sifry.com/alerts/archives/000436.html*.