Data Mining for Knowledge in Large Distributed Stores of Self-referential Data

S Terry Brugger <zow@llnl.gov>
Andrew Hurst <hurst9@llnl.gov>
Keith Mehl <khmehl@ucdavis.edu>

June 7, 2003

Abstract

There is a significant rise in the amount of self-referential data available in distributed data stores. Like most large data stores, we would like to use data mining techniques to gain knowledge based on this data. This paper examines algorithms and techniques for data mining such large, distributes stores of self-referential data, from the use of basic graph algorithms, to mining for new knowledge, to general characterization of the graph. Techniques are illustrated using examples from network intrusion detection, genome processing, and web searching.

1 Introduction

Data mining is the process of searching data for patterns or for relationships between the data that are not obvious from the data itself. In general, data mining is done to find 1) associations, where one data point is connected to another data point, 2) sequences (or paths), where one data point leads to another data point or a causality relationship or association can be drawn between data points, 3) classification, or finding patterns within data points, 4) clustering, finding related groups of data points, or 5) forecasting, or finding associations that will let us make predictions about future events from current data sets.

Many commercial companies build applications for mining data in business enterprise systems, such as these found in a brief Google search: SPSS Inc., Salford Systems, Oracle Inc., and Thinking Machines, Inc. However, general business data mining systems are beyond the scope of our paper, so we will generally ignore these from here on.

We are interested in what knowledge can be ascertained from a large store of distributed data given the constraint that the data is self referential. Such data can be found in applications such as mining network connection data for intrusion detection purposes, or finding important relationships between genome sequences. This data can be conceptually modeled as a graph. For example, GrIDS modeled network connection data as a graph, and used subgraph matching to identify patterns of intrusive activity [33]. Such a model allows us to transcend the traditional data mining model examining similarity between data items or the order in which data items occur. Instead, it allows us to derive knowledge on the similarities between the relationships of data items.

A basic example of self-referential data can be seen in the following table (named PEOPLE):

NAME1	NAME2	RELATIONSHIP
Alice	Bob	Husband
Bob	Alice	Wife
Alice	Charlie	Daughter
Charlie	Alice	Mother
Bob	Charlie	Daughter
Charlie	Bob	Father
Alice	David	Brother
David	Alice	Sister

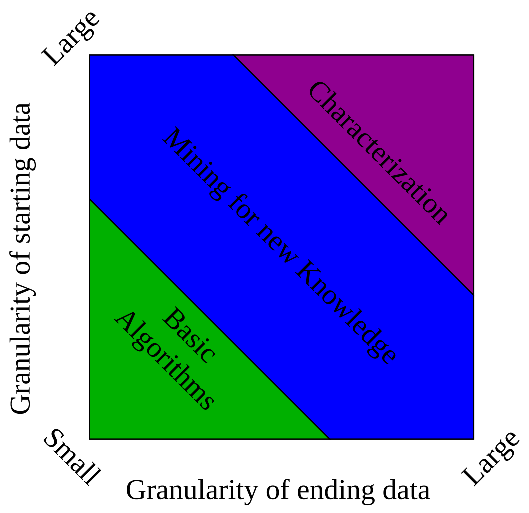
One can't see the relationship between Bob and David given any single row in the table, however if we join the table to itself with the SQL statement `SELECT a.name1, a.relationship, b.relationship, b.name2 FROM people a, people b WHERE a.name2=b.name1`, we get the following table:

a.NAME1	a.RELATIONSHIP	b.RELATIONSHIP	b.NAME2
Alice	Husband	Wife	Alice
Alice	Husband	Daughter	Charlie
Bob	Wife	Husband	Alice
Bob	Wife	Daughter	Charlie
Bob	Wife	Brother	David
	⋮		

Here we can see that David is the brother of Bob's wife (making them Brothers-in-law). The important point here is that when we joined the table with itself (the self-referential property), we were able to deduce information that wasn't in the base table. The problem arises however that, in this simple example, the resultant table would have contained 18 rows, a number that grows at the rate of $O(n^m)$ where n is the number of rows and m is the number of times that it is joined to itself.

For this paper we will examine what algorithms and techniques exist, have been proposed, or need to be developed, that exploit such self-referential data in a distributed data store. We will concentrate on the distributed nature of such tasks. We will briefly review interesting research results pertaining to distributed graph algorithms, particularly because they provide useful insights as to the algorithmic complexity of operations on distributed self-referential data. For the most part however, such algorithms only serve to answer simple queries, much as SQL statements do in traditional relational schemas. Our main interest lies in data mining operations, that is, algorithms and techniques that exploit the self-referential nature of the data to provide new knowledge from, or insight to, the data. It is worth noting that all the techniques we describe are also applicable on subsets of the data, hence we will not consider variants of techniques that run on subgraphs of our main graph. In this paper, when we refer to the graph nature of this self-referential data, we will consistently use the terminology vertices and edges. The term node will refer to a node in the distributed database system, is used synonymously with processor, and may imply a smaller granularity than the term site.

Different data mining techniques on self-referential data fall somewhere on a two dimensional continuum, where one axis represents the granularity of data that we begin with, and the other axis represents the granularity of data that we end with. It looks something like this:



The lower left side represents those operations for which we already have some of the data values identified, and we want to find related data values, for instance the values that provide a path from one value to another. These basic algorithmic operations will be covered in section 3. The upper right side represents those operations for which we do not begin with any particular data values, and which provide general characterizations about the referential nature of our data (meta-data). These operations will be covered in section 5. The heart of our paper are the operations in the middle, that exploit the self-referential nature of the data to either point to data items of interest given the entire dataset, or generalize information about specific data items. These operations will be covered in section 4. For each of these, we will provide a few examples from sample applications. We'll conclude by suggesting future directions for research.

Before we concentrate on our specific area of interest, we will briefly review several data mining systems from the literature. These serve as a good overall introduction to the problem in which we are interested. First, we examined some data mining clustering algorithms.

Clustering is the analysis of data points, each having several attributes, in order to group the data points so that points within a group are similar to each other and points in separate groups are different from each other. Two representative clustering algorithms are CLIQUE [?] and MAFIA [?]. CLIQUE and MAFIA are algorithms designed to detect and analyze clusters of data points in repositories of business data (data warehouses) and in large-scale scientific data such as satellite images. These data sets have the property that data points have many attributes, thus the number of dimensions in which "distance" between data points are measured may be quite large. The problem of finding clusters increases in complexity as the number of attributes, or dimensions, increases, and decreases in complexity as the granularity of the data attributes decreases. However, as granularity of the attributes increases, the accuracy of clustering algorithm suffers.

CLIQUE uses an n-dimensional grid, where n is the number of attributes per data point, and fixes the grid's granularity in each dimension by scanning the data points for clusters in each dimension. The grid is built dimension-by-dimension, or one attribute and related dimension at a time, using a pass over the data for each dimension. Density along a dimension k is determined at each pass by using the previous $k-1$ dimensions and a lexicographic ordering of the k'th attribute of the data points. The k'th grid interval is then selected so that the grid lines would be placed in such a manner as to break up the fewest clusters from the previous $k-1$ passes. The running time of the algorithm is exponential on the highest dimensionality of dense clusters in the data set, so the authors apply various pruning methods to speed processing. No provision is given in [?] for distributed data processing.

MAFIA is based on CLIQUE, however the authors note that some of the pruning methods used in CLIQUE may destroy cluster information in clusters that are not apparent until later passes through the data points. Instead, they use an adaptive grid size instead of a fixed grid size, so that a cluster is not split arbitrarily by grid lines. This is done by looking at adjacent "bins", or groups separated by a grid line, and merging groups if they contain a minimum threshold of similar data points. If this process results in a single bin along that dimension, the dimension is considered as not contributing to a cluster, and can be ignored. Reducing the dimensionality of the cluster greatly reduces the processing time. The authors of [?] consider parallelism in the context of Single Program Multiple Data (SPMD) environments, and show that their algorithm scales well in distributed environments.

However, both the CLIQUE and MAFIA algorithms deal mainly with data that is multi-dimensional but not self-referencing. Other algorithms we examined [?, ?] have this same feature. They make no attempt to use closely correlated data elements to discover patterns between distant data points in their data sets.

Another area we examined is mining textual data, such as in libraries or collections of survey or questionnaire data. Information Extraction is the discovery of knowledge in unformatted texts, such as from books in libraries. Text Mining is the analysis of organized natural language data for knowledge discovery, such as data from surveys. Two systems we examined are DiscoTEX [?] for information extraction and MedLEE [?] for text mining. These two problems are similar and interesting because of the almost unlimited way text can be organized and ideas expressed.

DiscoTEX (for DISCOvery from Text EXtraction) uses information extraction and traditional knowledge discovery techniques to discover a set of rules applicable to a particular corpus of text to then allow generating a more structured database from the original data. This structured database can then be mined using text mining or other data mining techniques. MedLEE (for Medical Language and Encoding System) is used at Columbia-Presbyterian Medical Center to process clinical narrative reports to generate a database which can then be mined. The reports are free form, and contain a limited set of very specific terms that can be associated with other terms with a high degree of accuracy; for example, "mild pulmonary vascular congestion" has an exact medical meaning. These exact terms may be interspersed with other terms that might not carry specific meaning to the general medical community, such as examination times or room numbers. Mining these resulting databases involves creating rules for determining associations among the data, for example, interestingness rules [?] like $A \rightarrow B$ where A and B are facts and the association indicates the probability that both A and B are true in the same data field. A high association would indicate an interesting relationship.

Text data are usually either unrelated or related in an externally known, fixed manner. Little internal relationship data exists within the data themselves, so this work, while interesting, contributes little towards our goal.

2 Applications

2.1 Network Intrusion Detection

When mining network connection logs, we semantically have a table that is horizontally fragmented across multiple sites. This table stores information about each network connection (such as a TCP/IP connection) to or from that site. This table includes information such as the source and destination addresses of the traffic, the time the connection started, the duration of the connection, number of bytes transferred, and application protocol used. While much useful knowledge can be gained using traditional, data item-centric data mining techniques, we can exploit the source and destination addresses in the connection data to gain additional insight by virtue of examining the traffic flow. Such an approach has already been done by GrIDS, which identified malicious activity by matching the traffic flow to a signature which was represented as a subgraph [33]. A trivial example of this is that an outgoing FTP connection from a local client to a suspicious IP address [1] may not be worrisome in and of itself, however if that connection occurs shortly after a suspicious incoming SSH connection to the same client, there may be more cause for alarm. Of course, a simple example such as that can likely be found through the derivation of frequency episode rules. More complex sequences involving an intruder first compromising a machine in the DMZ, using that machine to access a protected client, and finally to a trusted internal server, would be easier to derive when using the self-referential aspects of the data rather than just treating it as episodic. The distributed nature of this problem can be seen if the logs for the DMZ firewall and the internal firewall are stored in separate databases, possibly at different sites across the enterprise.

2.2 Genomics processing

Another booming application area for distributed database systems is in the storage of genomic data. An interesting aspect to genomic data is that it contains a great deal of redundancy, and only a small portion of it is actually used for coding protein sequences. The rest (approximately 97% of the human genome) is called non-coding sequences, and seems to contain meta-data concerning what protein sequences should be active for a given cell, and provide integrity information (the equivalent of computers' error checking and recovery bits). For the most part however, the exact structure and purpose of the non-coding sequences is not well understood [**?**]. It may be useful to store the genome data in a self-referential way such that known structures (such as protein sequences) can be connected. This would allow researchers to find the similarities and differences in the

[1]For example, an IP address that does not reverse resolve to a DNS name.

non-coding sequences or known coding sequences between pairs of protein sequences, either stored redundantly from the same organism or, more likely, between different samples of the same organism or different species of organisms. In particular, this would more easily allow researchers to track a genome through the evolution of a species, and in the process provide insight to the purpose of genetic sequences that were not previously understood.

2.3 Knowledge Discovery on the World Wide Web

In searching for knowledge on the World Wide Web, we can often leverage data on the structure of the pages (what pages they refer to and what pages refer to them). This is the basis of the Google PageRank algorithm [5]. The PageRank of a web page roughly raises the more web sites link to it. This can be problematic with some web sites that are highly self-referential. Examples of this include the web sites `http://www.kuro5hin.org` and `http://www.advogato.org/`. Both of these sites contain many links back to themselves on almost every page on their site. Users of these sites post stories and comments to the sites, which other users may read and comment on. Each comment and story include many links: to the author's information, to the comment itself, to reply to that comment or story, to score the comment, and a standard navigation set of links to other parts of the sites. In the case of large stories with many comments, some pages may get upwards of 1000 links to other pages on the same site.

Another source of highly referential data on the web is web logs. Web logs form a very large and self-referential subgraph within the web that can also complicate the ranking algorithms. In general the web log community is a community of sites that are updated roughly daily with new links or comments about different subjects. It is common for each web log to link to other web logs that the author identifies with. It is not uncommon to see web logs with upwards of 30 links to other web logs, all of which reference other web logs creating a deep link structure, until the originating log is linked back to.

These highly referential sources of data on the web can cause problems with ranking algorithms such as PageRank which can get biased by a type of network effect or group think. When one idea or topic is raised on a site it can automatically show up as a very highly reputable source for information by counting links, though it may not be a good source in actuality. In this case, we can use outliers to find all of the information that matches a specific search, but doesn't fall within the standard cluster of self-referential data. This idea in terms of network intrusion detection is discussed more in section 4.2.

Information about communities on the Internet can be found by examining these graphs. By examining the section of the graph with large amounts of web logs, you can find groups of web logs that all roughly keep together. By examining the nature of the cluster, you might find the most "important" web log in the bunch, thus showing the user that is the most highly regarded by the rest of the cluster. How that cluster interacts with other clusters of web logs allows you to derive still more information, such as political views, or the number of web logs shared between clusters.

3 Basic Algorithms

When working with large distributed graphs, it would be naive to throw out all of the current work in graph traversal. In this section we will cover the standard graph searching algorithms, and their limitations in a distributed setting. It should be noted that with the size and type of the graphs that we are discussing in this paper, it would be foolish to create the whole graph outside of the already relational or other representation of the data. The amount of data to store would be roughly doubled and the gain in ease of searching and implementing would be minimal. Most of the following algorithms can be executed even without an actual "graph" to work on, as parts of the graph can be generated as needed.

One of the most difficult parts of creating and using distributed graph algorithms is that the centralized counterparts are often inherently sequential. This is to be expected though, as these algorithms need to keep track of all of the vertices that have been seen and where to go next, which is hard to do reliably and quickly without a central repository of knowledge. Also when these algorithms start being distributed, duplication of effort needs to be considered. In the case of distributed BFS and DFS, the distributed versions might produce different graphs than the sequential versions because of the vertices being visited in a different order than with the centralized version.

Another aspect to think about when designing distributed graph algorithms is whether or not the graph is weighted or unweighted, and if it is directed or undirected. Based on our example applications, the following algorithms will focus on the directed and unweighted case.

3.1 Basic Graph Searching

There are four basic graph searching algorithms we will cover in this paper: breadth-first search, depth-first search, shortest path, and all-paths. Which to use depends on what you want to find out. BFS and DFS are the most basic, but they can be used to create the shortest path and the all-paths algorithms, which provide more information.

BFS The Breadth-First Search (BFS) algorithm in its centralized form is an algorithm for crawling and mapping the entire graph from a single starting point. First a vertex is chosen as the root vertex (or starting vertex). Vertices are added to the graph (in the undirected case) by first finding all vertices of distance 1 from the root vertex and adding them to the to-be-processed queue. Then each vertex in that queue is processed and each of its children are added to the queue, and so on until all vertices have been found. This algorithm finds all vertices at distance n away before it finds vertices at distance $n+1$, until all vertices have been reached. This will prove to be one of the more useful algorithms in distributed graph searching as described in this paper, and thus it will have the most space accorded to it.

Some work that has covered this before includes [35, 36]. Tel mentions that because of the queue data structure used in the sequential BFS algorithm, it is very difficult to distribute [36]. He proceeds to caution against using BFS in a distributed setting, if possible. Regardless of his warning, there are some distributed algorithms for it that he describes. These algorithms are almost exactly the same as the sequential BFS, with the difference in how information is passed around about each vertex. In the centralized case the program running the algorithm knows all about which vertices have been visited and when, and where to search next. In the distributed case (and specifically in Tel's examples) each vertex is at a different site, and all information gets distributed through message passing. This changes the main cost of the algorithm from the number of vertices, to the number of messages passed between nodes. This will be a common theme in these distributed graph algorithms as you will see in later sections.

Another method is to parallelize the searching of each vertex's subtrees. The change to the original algorithm would be to assign each vertex found by the first iteration to another processor[2]. Each processor would be given essentially the command of "start a BFS from vertex k and report the results" where k would be one of the vertices found in the first pass of the vertices off of the root vertex. This procedure could be duplicated as each list of children is found, until all of the processors are used.

A problem can arise with this approach: duplication of effort. How do you ensure that each processor in the algorithm doesn't search parts of the graph that have already been searched by another processor? There are two approaches to this, depending on how much information can be stored with each vertex. The first is to mark each vertex as visited when it is first happened upon in the algorithm. This makes the implementation of this algorithm very easy, but it takes up much more space. Also it introduces issues as in what happens if multiple BFSs are occurring at once, how do you tell if a vertex has been marked from one or the other? To finish the algorithm, each processor would send the results of its search to its parent, when it has exhausted its search space. The root processor would contain the whole BFS tree when the algorithm was finished.

A second approach is to have each processor report back to its parent processor the vertices that it has found in each iteration of the algorithm. The processor would have to report back the vertices it finds anyway at the end of the algorithm, so this incremental notification would not be too large of an addition. After each iteration, the parent processor would receive the list of vertices that its children have found, and process the lists in the order that it dispatched the processors (that is, the order that it found the processor's starting vertices). It would report back to its children the list of vertices that earlier searches already found, and that they should discard. The child processors would take this into account and remove those vertices from their search lists. After a processor has searched all of the vertices in its part of the graph, it would send its results back to the parent processor which would merge them in with the results from its other child processors. This would continue until the entire resultant search tree resides at the root vertex. This approach, though it would take less memory storage at each individual node, would require much more messaging overhead between the different processors. As

[2]The term processor is used here to represent individual agents of a search program, or individual processors in a machine. The exact meaning would be implementation specific.

with most problems in Computer Science, the solution is a trade-off between space and running time. Variations of the two algorithms can be developed that would return only the paths to the vertices that found the particular search term can be developed as well. This would greatly benefit the space usage, as the whole graph would no longer need to be returned.

A problem that can arise in distributing this algorithm is that the resultant graph in most cases will not end up looking the same as the graph from a purely sequential algorithm. In the parallel version, vertices will be visited in a different order, and thus whole subtrees could end up on different parts of the resultant tree. The work described above describing duplication of effort can fix this problem, however. In the latter approach utilizing message passing, each processor runs symmetrically (or can be made to) with the other processors for each level of the iteration. The root vertex of each processor gives favoritism to vertices discovered by the earlier child vertices, so that the vertices will fall into the tree in the same place they would with the sequential version. If the algorithm is implemented by marking vertices, the solution is a little more complex. Because each vertex is essentially first-come first-serve, the graph could be changed dramatically by a very fast processor searching the whole graph quickly. A solution to this is to store the depth that each vertex was found along with the marker. If this vertex was found by a processor at a depth deeper than the one the currently running processor is on, it claims the vertex as its own and notifies the vertex's previous owner.

There is one limitation of these algorithms as described that the astute reader may have picked up on. Strictly as described above, a new processor is used for every vertex found, and the original processor just waits idly until it gets responses from its children. The solution to this is described in [11] for distributed DFS, but it can be applied just as easily to distributed BFS as well. This is described more in-depth in the paragraphs on Load Balancing on page 13.

Applications of a distributed BFS is readily apparent given the example in the introduction of mining log data for intrusion detection. A situation can be imagined where an intrusion was made by a certain IP address to a certain host on an operator's network. To see how many other hosts this attacker might have gained access to, it would be useful to use the access logs of hosts on the network as the basis for a BFS on the entire network to see how many hosts on the network this attacker probed, starting from the compromised host. Direct application of a distributed BFS would be uncommon though. More often it would be used as a basis for another algorithm like shortest path or all-paths to convey more useful information.

DFS The Depth-First Search (DFS) algorithm is essentially the opposite of the BFS. It is an algorithm for searching a graph structure by traversing the graph to the farthest reaches possible from a starting vertex's edge, before trying another edge off of that same vertex. More than a few papers have described work in distributing this algorithm: [11, 25, 35, 36].

[36] describes a few distributed algorithms for DFS that make heavy use of message passing, and some optimizations for them. The optimizations focus more on lessening the number of messages passed and less on parallel computation, and accordingly the algorithms he describes are mainly sequential and similar to the centralized case, but with message passing added in.

[11] describes and references quite a few algorithms for parallelizing DFS on Multiple-Instruction Multiple Data stream and Single Instruction Multiple Data stream computers. Though his paper doesn't describe distributing these algorithms across multiple computers in a shared nothing configuration, we don't believe it would take much effort to make that change utilizing some sort of message passing algorithm.

With these two approaches, the two main difficulties with distributing DFS arise from the message passing overhead between all of the different sites and load balancing. Trees generated by DFS can be very imbalanced, and this can prove detrimental for naive parallel algorithms. An example of this is a tree where the root has two children, one makes up a subtree of one element, the other has thousands of elements. If the algorithm only assigned the processing based on those two children, one processor would finish almost immediately, while the other would take much longer.

Thus there needs to be some load balancing between the different parts of the search to minimize communication and maximize processor utilization. The main method described in [11] for fixing this load balancing problem, is to divide the algorithm into two parts (outside of the basic sequential algorithm): task partitioning and subtask distribution. Task partitioning is concerned with the optimal partitioning of the subgraphs to keep all of the separate processing nodes busy as much as possible. The two main methods for this are stack splitting and vertex splitting. Vertex splitting works by giving vertices to other processors that the current processor has come across but hasn't gotten around to searching yet, while sticking to the search ordering of DFS (that is, the processor gives the vertex that was most recently happened upon but not searched). Stack splitting gives other processors a part of the set of all vertices that the current processor has seen but not had the time to search yet. As one can see these are similar approaches that mainly differ in the size of the work partitioned. Subtask distribution is concerned with when to partition, and once the work has been partitioned where should it go to be processed. When to partition can be decided on-demand (processors request to get more work from other, busier, processors) or assigned (the processor doing all the work decides when it has too much, splits it up and sends it off to some idle processors). But before work can be distributed, it must know where its going. Two similar solutions present themselves here: processors that ask for work can get it, or the processor that does the partitioning can assign the work.

In the case of a distributed graph based data structure, the solution of having the main processor (the one doing the current round of the DFS) decide when to split and who to send it to seems the most feasible. In many cases large parts of the graph would reside on different sites. A search would start on a root site, who would start the DFS at the desired vertex. When the next vertex to be searched is on a different site, then that processor would send a message to the other site of the form "start a DFS at vertex t on your site and give me the results." After sending the message, it could continue the DFS from the last vertex it searched on its own site. As you can see this is quickly running into the problems discussed (and solved) in the BFS section, page 10. We refer you there for more information on duplication of effort.

This has a similar application as BFS to graph based intrusion detection. In fact the DFS and BFS will both give the same results, in a strict searching sense. It is for this reason that the difference of when to use BFS over DFS is mainly an implementation issue. Its hard to imagine a case where the operator of this intrusion detection system would really mind what search was used, unless the graph was extremely swayed such as to be optimized for a particular algorithm.

Shortest Path Given two vertices s and t, find the shortest path between them. Because we're focusing on unweighted directed graphs, the shortest path problem is much easier to solve. In the unweighted problem, the shortest path from one vertex to another can be determined by a simple BFS [32]. Starting from vertex s, do a distributed BFS until vertex t is found. That path is the shortest path between the two vertices. Thus the work presented earlier in section 3.1 on page 10 can be leveraged to provide an optimal solution to this problem. This, of course, relies on the graph resulting from the distributed BFS to be the same as the graph resulting from a sequential BFS. Methods are described in the earlier sections to ensure that.

In application to distributed intrusion detection, this algorithm could be used to find a path of logins between two hosts in a network, where one host would represent a server that has been broken into, and the other host would represent a server with important data such as a source repository or customer data. When used in this setting it is quite possible that the shortest path between two hosts in this case would be through the system administrator's computer. Assuming the system administrator trusts himself, he would need to find results that don't include his computer. This is where constraints come into play.

Constraints can be used to limit the search results to data that is more useful. As the previous example showed, one method might be to find the shortest path between two vertices while ignoring the system administrator's workstation. Another might be to constrain the search to include a router or host that is suspected of being insecure in the first place, to narrow the search space and get faster results. As far as the algorithm design goes, a constraint requiring that a certain computer not be in the shortest path would result in the BFS algorithm treating that vertex as if it didn't exist. For ensuring that a specific vertex q was in the results, a BFS can be run from s to q and then from q to t. The concatenation of these paths would give the shortest path from s to t through q.

There is a method to speed up the shortest path computation as well: run the distributed BFS in parallel. Start two BFSs at the same time, one from s searching for t, and one from t searching for s. There would need to be some extra logic to make sure that both BFSs were aware of each other, probably by message passing to make sure that they didn't duplicate effort. In the best case it appears that both searches would meet in the middle, roughly halfway between s and t. This would give a rough speedup of about 2 times depending on the number of messages passed between sites, which could probably be minimized to one per round of BFS.

All-paths This algorithm returns all paths in a graph between two vertices. It is useful in the case that the shortest path doesn't give you enough or useful information. I.e. the shortest path between two workstations is google.com, which isn't useful from a network security standpoint (one could assume that there was no attack originating from google.com[3]). Thus you find all paths between the two vertices, and look through them to find suspicious ones.

Generally this algorithm would give much too many results for one person to handle, so you would use constraints to make this return more relevant results. For example, return all paths between these two computers, ignoring connections to google.com and portal.examplesubnet.com, and only paths of length less than 5 computers. The latter constraint is one of the most useful, and reduces the all-paths problem to the k shortest paths problem — Find the k shortest paths between two vertices in a graph.

[10] describes a relatively easy method to find the k shortest paths by using BFS (again!). Though his description is for a digraph, it can be expanded to arbitrary graphs as well. By executing a simple BFS on the graph and keeping track of the paths found to each vertex in turn as the graph is explored, you can find all paths to a particular vertex in the same time it takes to do an exhaustive BFS. Since you would most likely constrain the algorithm to the k shortest paths, then you could stop after k paths were found. This algorithm would probably benefit the most by the vertex-marking BFS algorithm described earlier, because the first processor to find a vertex with $k - 1$ marks would be the kth mark, and thus could call for the end of the algorithm and all processors would send their found paths back to the root vertex.

3.2 Min-cut

The minimum cut algorithm (also referred to as the maximum flow problem) is concerned with finding the set of vertices or edges that, when removed from the graph, cause the graph to be partitioned into two disconnected sets. This has many applications. In genomics, this could be used to find the gene that would cause the most change in an

[3]It is up to the reader to decide if thats a good network security policy or not.

organism when removed or mutated. In network intrusion detection, this information can be used as an assurance measure to make sure that there is only one way out of your internal network. While that might seem like a contrived case, think about the possibility of an employee having an ISDN line or modem into their work computer, so that they may easily work from home. If they're not careful, its very easy to bridge the networks with their workstation. The minimum cut algorithm can be used to find cases such as these.

The general case for the graph minimum cut is an NP-hard problem [8]. The subject has spawned a number of approaches dealing with heuristics to estimate the minimal cut of a graph, some of which are described in [13]. Neither of these papers reference the distributed problem. Luckily we can get almost all of the information we need out of the much easier to solve (runs in polynomial time) case where we need to find the minimal cut between two vertices s and t. For example, to make sure that there is only one path out of your network, the operator could choose a standard workstation on the interior of the network and one on the exterior, and find the minimum cut of them. If the minimum cut is greater than one, then there is more than one way into the network and the cut vertices (or edges) need to be examined.

Two of the simplest solutions to this problem leverage work in earlier sections on distributed graph algorithms. The first method is to perform the All-paths[4] algorithm between the vertices s and t. Once all of the paths have been found perform a min-cut analysis on the resulting subgraph. Since the list of all paths between the vertices will likely be much smaller, it would be possible to run a brute force analysis on the results, to find the minimum cut.

The other method is to find all the shortest paths between s and t and perform a min-cut across those paths. This can be done in time n for a path of length n by examining the number of vertices or edges at each shell level away from s or t and picking the shell level with the smallest number of vertices or edges to remove from the path. Then repeat for the next shortest path between s and t, until there are no more paths from s to t. The set of vertices or edges that were removed make the minimal cut set. If the maximum non-cyclic path length from s to t is m, the run time of this approach is $\mathcal{O}(mn)$.

Finding the min-cut based on vertices or edges is application spe-

[4]Or the k shortest paths

cific. For instance, you might want the vertex minimal cut into your network to be one (your company firewall) whereas you might want the edge minimum cut to be no less than three (redundant Internet connections all feeding into your firewall).

4 Mining for New Knowledge

The algorithms that we are most interested in are those that will provide us with new, detailed knowledge given very little initial input, beyond the set of self-referential data. Essentially, we are looking for techniques that will provide us with new insight into our data, with a minimum amount of work on our part. Of course, as with much data mining, these algorithms can be expected to produce a lot of information that is taken as common knowledge to the users, and much more that is completely spurious. Much work can be done on providing constraints to the algorithms to limit such results, however we do not address such techniques as research in that area is premature given the infancy of research in the core area, and such techniques are almost always domain (application) specific.

Much of the work in this section is in its infancy, not only in distributed systems, but in general. Given the proliferation of distributed, self-referential data sets and the potential for high-payoff from these techniques for knowledge discovery, it is our assertion that these techniques should receive more attention. This section will cover what work has been done thus far on each technique, our thoughts on approaches to applying the technique in a distributed setting, and examples of how the techniques could be useful in our example application areas.

We begin by looking for patterns in the graph formed by the self-referential nature of the data, then we consider the detection of data points that fall outside expected patterns (outliers). Next we consider the identification of important data points. Finally, we look at two techniques related to clustering: finding sets of generally related data points, and finding groups of strongly connected data points.

4.1 Frequent Subgraph Identification

Given that our data is self-referential, some of the most interesting knowledge that we can discover are frequently reoccurring patterns in that data. This is generally known as frequent subgraph identification. Work in this area developed out of general graph algorithm research and progressed slowly for a number of years, advancing to the point of Chen and Yun creating an algorithm for finding the maximum common subgraph based on maximum cliques and graph coloring [6]. This area has started to receive more attention when researchers found the need to perform data mining on large sets of graph-based data, in particular the composition of chemical compounds. In these datasets they have multiple data items, each represented by a graph, and they want to find the similarities between them. Inokuchi et al built an algorithm to do this based on Agrawal's apriori-based method for mining association rules [14]. Kuramochi and Karypis then built a similar algorithm which appears to be more efficient and scalable [22]. The important thing to note about these algorithms is that they assume that each data item is an entire graph, independent from all the other graphs, whereas in our dataset, each data item is a separate vertex, and will likely be connected to others.

The first task that needs to be considered is how these algorithms can be extended to work on a single graph of arbitrary size and conductivity. It seems to us that the apriori-based approach has merit. The key here is that, instead of starting with n arbitrary graph-based data items, we should start with subgraphs anchored at each of the n data items. Knowing that a single edge is not an interesting subgraph, we can begin with a double-edged structure, where the connecting vertex can serve as the anchor. There are then four possible edges that can be added to build a three edged subgraph (one from each of the vertices, or one connecting the two end vertices). One can easily see when adding a fourth edge, for which there are 16 possible combinations of vertices and edges, that the enumeration of subgraphs grows exponentially with the addition of new edges. Intuitively, the only way to identify which subgraphs of m edges exist in our graph is to enumerate over them, implying that this operation is NP-complete. We can probably optimize the operations somewhat by eliminating generally useless structures, however such approaches are almost certainly application specific. For instance, in some applications, a subgraph where every vertex has only one or two edges (a line essentially), may not be of any interest, whereas in others it might be. At the same time, one must be careful how this operation is done because if the expansion of possible subgraphs is represented as a tree, pruning the tree of a "useless" structure to far up may preclude the generation of a more useful structure. For example, a loop of p edges is only a line of $p-1$ edges until the last edge is connected.

Fortunately, some of the computational intensity of the problem may be alleviated by virtue of operating within a distributed database system. That said, we're left with the problem of how to actually implement the algorithm in such a setting. We will assume that the subgraphs may extend further into the data present on other nodes than we can reasonably store as ghost vertices on this node. Examining the two usual approaches, migration of data or migration of computation, it seems that migration of computation would be more efficient. Consider the case where we need to determine if a vertex on another node is part of a subgraph structure that forms a triangle, yet in actuality that vertex is connected to 1000 leaf vertices. If we do the calculation by migrating the data, we've just copied the information on 1000 edges and vertices for no good reason. If instead, we send a message to the other site asking if vertex v was part of a structure $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_1$ and the remote site responded with either a one (yes) or zero (no), we've gotten the same answer, done the same amount of work (just at different sites), and transferred a lot less over the network.

One problem with finding frequently occurring subgraphs in a large interconnected graph such as that formed by our self-referential data is defining when two subgraphs are equivalent. This is a problem because the structure of our graph is based on using a single vertex to represent all occurrences of that value in the data. For example, all instances of the IP address "1.2.3.4" will be represented by the same vertex. As a result, we can not base our subgraphs on vertex value matching. This leaves us with two possibilities: we can either match using only topology information without using any values, or we can match based on vertex type. The latter assumes that each row in our tables contains both source and destination values, as well as source and destination types. For instance, in web searching, we may have HTML page x that is linked to PDF document y, hence our database will need to capture that x is an HTML page and y is a PDF document, then we could use this type information to do our subgraph matching.

This technique will not likely provide any instant answers for any application; in fact it will likely generate more questions. Fortunately, the answers to those questions will likely provide a great deal of insight into the dataset. For instance, one might find a pattern in a network connection dataset consisting of a line of hosts, each one forming connections with the next, and a single host that connects to all of them. Is that single host an attacker, systematically compromising each machine in the chain, or is that host a system administrator connecting to all the hosts to administer them?

4.2 Outliers

At the opposite end of the spectrum from finding frequent patterns in the data, is finding vertices that do not follow such patterns. These are known as outliers. Much research has been done on detecting outliers, which are typically thought of as data points that are inconsistent with the other observed data, based on some measure of similarity from each data point to the others. Shekhar et al decided to focus on outliers where the similarity between vertices is determined based on their graph conductivity rather than a Euclidean distance [31]. They tested their algorithm on a dataset of automobile traffic data and were able to readily identify a number of anomalies in the dataset. They provide a rather detailed analysis of the I/O performance of their technique. Central to their technique is the application of a graph clustering method, which we will discuss in more detail in section 4.4. A natural approach to applying this outlier detection technique in the distributed realm would be through the application of a clustering technique designed to operate on distributed data, and then to allow each site to identify those data points assigned to them which appear to be outliers. Alternatively, since outlier detection is based heavily on calculations involving close neighbors, each site should be able to perform such detection using only one or two layers of ghost vertices. In that case, the only communication that would be required between sites would be to maintain the ghost vertices and combine the results of the outlier detection.

Outliers are typically of interest in most applications. When examining network connection data, most attacks appear as outliers because the attacker must do something abnormal to gain access to a system. In genome analysis, an outlier may represent a mutation, or a unique protein sequence that serves a special purpose. In web searching, outliers may represent pages that contain opposing viewpoints and hence, are typically not linked to by other pages in a given community.

Outlier detection could also be performed on the graph based on the directionality of edges. Specifically, disparities between the number of in-edges versus out-edges are typically of interest. For example, in network connection graphs, vertices which only serve as the source of connections are clients, and vertices that only serve as the destination of connections are servers. Hence, if a machine seems to be a client (has a large number of out-edges), yet it has maybe one in-edge, that inbound connection is worth investigating.

4.3 Important Vertices

Most graphs have certain vertices that are particularly important. In many cases, this importance is due to that vertex having a high degree of conductivity with other vertices throughout the graph. Graphs that are formed based on a power-law distribution (see section 5.3 naturally form these so-called hubs. In many cases, these vertices are essential to the conductivity of the graph, and their elimination will cause a discontinuity in the graph. For this reason, such hubs are typically selected as part of a min-cut of the graph (see section 3.2).

The exact semantics associated with importance of a vertex is application dependent. For this reason, the algorithms to calculate such importance measures are application dependent as well. At the core of most of these algorithms though is the notion of counting the number of in-edges and out-edges for each vertex in the graph. Actually doing this counting on each site in the distributed system is trivial: given a modern RDBMS, a basic query with grouping and aggregation will give us what we need. The trick becomes in combining all the counts for each vertex together. Such a problem has long been present in the parallel processing community, and is known as the reduction problem [27, 18]. The Message Passing Interface (MPI) specification includes specific calls in its API for taking an array of values on each node and summing the individual elements together, saving the result in an array on either one node (the "root" node), or all nodes [27]. The exact manner in which this is done is left to the implementor, who will typically leverage any knowledge of the topology of the system, and may even utilize specialized hardware to accomplish the task.

Even without the specialty hardware, we can likely leverage such reduction techniques by using something like MPICH[5] directly [12]. This, unfortunately, has two drawbacks: 1. The performance of MPICH

[5]A software-only portable MPI library

for reduction operations in a widely distributed setting is lacking and 2. We need two arrays representing the counts for all vertices in the entire system on every node. Kielmann et al address the first problem with MagPIe, an extension to MPICH for optimizing performance in a distributed setting by performing operations in a hierarchical manner based on the proximity of nodes [18]. Since the source to MPICH is available, and the source to MagPIe is apparently available, we can address the second problem by modifying the reduction call to take a hash instead of an array, with the understanding that if a node doesn't have a value, that value is zero. As the results are being tabulated between nodes, a given node only needs to store the sum for a given vertex if it already has that vertex. The exception is nodes higher in the hierarchy, representing a cluster of computational nodes. While they need to store the counts for all the vertices on all the nodes they represent, a hash representation will probably still be more efficient.

When doing network intrusion detection, we typically separate hosts into inside and outside sets. The inside hosts are those on our network that we are monitoring and attempting to protect. All others are considered outside hosts. We expect to see a number of hosts, both inside and outside, with a large number of inbound connections; as previously noted, these hosts are servers. We also expect to see a number of inside machines with a large number of outgoing connections. These are our local clients. Now, when we see a see an outside machine generating connections to a large number of inside machines, they are likely scanning us, attempting to find vulnerable hosts that they can attack.

Web searching represents the most popular application of utilizing important vertices today. The popular search engine Google utilizes an algorithm called PageRank which produces a score for every page based in part to its conductivity to other pages, particularly the number of other pages that link to it [5]. This is based on the premise that if a page has a high number of links pointing to it, then a large number of other authors must have found it useful, so it must be good.

4.4 Clustering

We're interested in operations that cluster vertices together based on the self-referential nature of the data. The data can be clustered using other methods such as Latent Semantic Indexing [24, 38], however that is beyond the scope of this paper. Instead, we will focus on clustering vertices based on proximity, and on strongly connected subgraphs.

4.4.1 Partitioning

Partitioning of graphs is a widely studied problem as it is a vital step in modern scientific simulations where an unstructured mesh, represented as a graph, is partitioned such that different pieces can be assigned to different processors. An optimal graph partition is one that minimizes the number of edge cuts while maintaining a distribution of vertices between partitions that is as close to even as possible. It is well known[6] that finding such an optimal graph partition is NP-complete. Fortunately, we can find a close approximation, which is sufficient both for load-balancing scientific simulations and determining what vertices are related in data-mining.

Dobbelaere gives a nice survey of techniques for partitioning graphs using parallel methods in [9]. He provides an excellent overview of the field and explains some of the terminology and techniques, for instance the Kernighan-Lin / Fiduccia-Mattheyses (KL/FM) algorithm. He concludes that multilevel approaches to graph partitioning seem to be the most promising [9]. One of the packages he looked at was ParMETIS, which was specifically designed for doing parallel graph partitioning, and was designed by Karypis, Kumar, and Schloegel at the University of Minnesota. The approach ParMETIS takes to graph partitioning is to iteratively coarsen the graph by collapsing related vertices together until the graph is of a reasonable size[7], such that an optimal or near-optimal partition can be found. Then the graph is iteratively uncoarsened, and refined using the KL/FM algorithm at each iteration. This appears to result in an excellent partition in a very short time [30, 29, 17, 16, 15].

For most applications, a basic partition such as described here won't provide much useful information. For network intrusion detection, we expect to see vertices partitioned roughly by the sites whose logs they appeared in, and for genome processing, we expect to see sequences from the same chromosome together. As noted in section 4.2, the real interesting knowledge lies in which data points are not consistent with such a partitioning. Another area where partitions can be useful is the refinement of results. For example, when searching for

[6]Which is to say that every paper on the subject notes this without citing the original source

[7]Typically in the hundreds of vertices

a concept on the web, it would be useful to eliminate all the results from a partition that may represent a different concept with the same name.

4.4.2 Strongly Connected Subgraphs

Graph partitioning, while useful, is inexact. It was designed primarily for load-balancing graphs between computational nodes in scientific simulations. While finding an optimal partition is NP-complete, a graph may have multiple such optimal partitions. The difference between these optimal partitions may or may not have any semantic meaning associated with it. Strongly connected subgraphs on the other hand, are formally defined, so what we get is an exact answer. Recall that a strongly connected subgraph is one in which there is a path from every vertex to every other vertex in the subgraph. Finding all such strongly connected subgraphs is fairly easy in a distributed setting. Specifically, we can utilize the parallel DFS technique discussed in section 3.1. When a cycle is detected, we union all the vertices on that cycle, and any strongly connected subgraphs those vertices belong to, together to form a new strongly connected subgraph.

Strongly connected subgraphs, unlike general graph partitions, are typically of interest in applications. In network intrusion detection, they typically indicate a trust relationship between the hosts in the subgraph. A system administrator should be able to look at such a subgraph and immediately identify any machines that should not be present in such a relationship (and if the information on trust relationships is stored in the database, the machine should be able to deduce this automatically). In genome processing, a strongly connected subgraph may be indicative of a group of proteins with a similar function. Likewise, in web searching, a user may identify a page of interest and use the strongly connected subgraph for that page to find other closely related material.

5 Characterization of the Overall Structure

Mining algorithms applied to self-referential graphs may yield information about the nature of the information held in the graphs themselves. The results of mining deeper relationships, and analysis of the resulting graphs, will provide information useful beyond the discovered data itself. For example, analyzing navigability from Web sites will show related sites as being nearer, while those sites at greater distances will be less closely related.

.

5.1 Navigability / Conductivity

Navigability of a graph refers to the likelihood that a path exists from a vertex to another vertex. Conductivity is a measure of the number of such paths that pass through any particular vertex. Analyzing the navigability and conductivity of a graph resulting from application of our algorithms to a distributed self-referential database may yield knowledge about the importance or activities associated with a vertex in the graph.

For example, in the genome example, a vertex with very high conductivity may indicate that that a non-coding gene, represented in that vertex, is associated with checking for correct production of a protein that was created earlier in the evolution of that organism. Less-connected vertices may show sequences that have arisen more recently.

In the World Wide Web, the type of a vertex, and the types of vertices it is connected to, may be a measure of the importance and accuracy of the information on the site. Sites that are more important will be referred to by many other sites, and these sites will be referenced again by other sites. In addition, the lack of a path from one site to another site may show that the sites are unrelated. Users of the WWW may then use such path-related knowledge to tell that a site is useful or not when trying to find information on a particular topic.

5.2 Ontology induction

Ontology is the specification of rules for the knowledge of and relationships between entities. The application of our algorithms to self-referential data may yield clues to the relationships between the vertices.

For example, in the PEOPLE table, creation of a fully connected sub-graph indicates a nuclear family, each related to all of the others. For example, a vertex has a "father" relationship to two other vertices. We can induce that the two vertices will have either "daughter" or "son" relationships to the original vertex, and that they will have either "brother" or "sister" relationships to each other. From these basic rules, we can create data-mining knowledge-discovery rules to extract further information about the relationships between the individuals in the PEOPLE table.

For the network topology example, the type of the node indicates what other types of nodes to which it can connect. For example, a client node may connect directly to a router, or may only connect to a switch, then a router. By analyzing the relationships between client nodes and their neighbors, we can induce a set of rules that state what types of nodes can be connected to what other types nodes. Again, with these rules we can create other rules to mine knowledge from the network data.

5.3 Distribution Identification

The variability of the number of edges going through each vertex in a graph gives a good picture of the distribution of the graph. In homogeneous networks, where the number of edges per vertex is relatively constant, failure of a single node will likely cause a small portion of the graph to become disconnected. However, in heterogeneous networks, such as the WWW, a small number of nodes are critical (see the discussion in section 4.3, Important Vertices). Loss of these nodes will cause large portions of the graph to become disconnected. By analyzing the distribution of the graph, we can learn about the vulnerability of the network to attacks or to failures of a number of nodes.

Barabási, et al, in [1] discuss the resilience of networks to failure or attack, modeling the World-Wide Web and a randomly generated (exponential) network, showing that the Web is quite resilient to failure of a small set of nodes chosen randomly, but very vulnerable to a deliberate attack on a small set of carefully chosen nodes. A randomly generated homogeneous network is; however, much more likely to break into disconnected subsections upon failure of a small set of nodes. It is also equally vulnerable to a deliberate attack upon a small set of carefully chosen nodes.

The distribution knowledge could be used to mine additional knowledge from the graph. For example, in the genome project, the presence of an important vertex could possibly indicate a critical gene, the mutation of which would cause a serious problem.

6 Open Problems and Conclusion

In this paper, we have examined the nature of self-referential data in a distributed database system. We have looked at the breath of ways in which the self-referential nature of the data can be exploited to gain new knowledge, ranging from basic graph operations to general graph characterization, with a focus on finding new knowledge with minimal knowledge of the data. As much as possible, we have focused on the distributed nature of such techniques. To illustrate the usefulness of the techniques we've covered, we've included examples from the application areas of network intrusion detection, genomics, and web searching.

There is a great deal of work that needs to be done in this area. Some of the techniques we've covered have barely received any attention in published literature and would be served by more research, even in the basic centralized case. These areas are

1. Frequent subgraph discovery on large, connected graphs
2. Outlier detection in graphs

In other cases, research on techniques in the centralized case is fairly well established, however there is a great potential for further research on the technique in the distributed setting. These areas are

1. Breadth-first search
2. Finding the minimum cut between two points
3. Discovery of important vertices

7 Auspices and Acknowledgements

This paper is a personal work of the authors. It was not developed under the auspices of the United States Department of Energy, the University of California or Lawrence Livermore National Lab. Any opinions expressed herein are solely those of the authors, and may or may not reflect the views of their employers or the University of California, Davis.

References

[1] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Attack and error tolerance of complex networks. *Nature*, 406:378–382, 27 July 2000.

[2] William A. Andersen, James A. Hendler, Matthew P. Evett, and Brian P. Kettler. Massively parallel matching of knowledge structures. *Massively Parallel Artificial Intelligence*, pages 52–73, 1994.

[3] Jerzy Bala, Sung Baik, Ali Hadjarian, B. K. Gogia, and Chris Manthorne. Application of a distributed data mining approach to network intrustion detection. In *AAMAS*, pages 1419–1420, 2002.

[4] Albert-László Barabási, Réka Albert, and Hawoong Jeong. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A*, 281:69–77, 2000.

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[6] C.-W. K. Chen and D. Y. Y. Yun. Unifying graph-matching problem with a practical solution. In *Proc. International Conference on Systems, Signals, Control, Computers*, September 1998.

[7] Edmond Chow, Tina Eliassi-Rad, and Van Emden Hensen. Parallel graph algorithms for complex networks. Internal proposal, 2003.

[8] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, P. D. Seymour, and Mihalis Yannakakis. The complexity of multiterminal cuts. *SIAM J. Comput.*, 23(4):864–894, 1994.

[9] Jeffrey A. Dobbelaere. Parallel k-way partitioning schemes for irregular graphs.

[10] David Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.

[11] A. Y. Grama and V. Kumar. A survey of parallel search algorithms for discrete optimization problems. Personnal communication, 1993.

[12] William Gropp and Ewing Lusk. User's guide for MPICH, a portable implementation of MPI.

[13] Karin Högstedt, Douglas N. Kimelman, Vadakkedathu T. Rajan, Tova Roth, and Mark N. Wegman. Graph cutting algorithms for distributed aplications partitioning. *Performance Evaluation and Review*, 24(4), March 2001.

[14] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery*, pages 13–23, 2000.

[15] George Karypis and Vipin Kumar. Parallel Multilevel k-way Partitioning Schemes for Irregular Graphs. Technical Report 036, University of Minnesota, Minneapolis, MN 55454, May 1996.

[16] George Karypis and Vipin Kumar. A coarse-grain parallel formulation of multilevel k-way graph-partitioning algorithm. In *Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.

[17] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[18] T. Kielmann, R. F. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MPI's reduction operations in clustered wide area systems. In *Proc. Message Passing Interface Developer's and User's Conference*, Atlanta, GA, March 1999.

[19] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[20] Kumar, Raghavan, Rajagopalan, Sivakumar, Tomkins, and Upfal. Stochastic models for the web graph. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000.

[21] S. Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Extracting large-scale knowledge bases from the web. In *The VLDB Journal*, pages 639–650, 1999.

[22] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.

[23] Chain-Wu Lee. TERRESA: A task-based message-driven parallel semantic network system. Technical Report 99-01, Department of Computer Science and Engineering, University at Buffalo, 30 January 1999.

[24] Todd A. Letsche and Michael W. Berry. Large-scale information retrieval with latent semantic indexing. *Information Sciences - Applications*, 1996.

[25] S. A. M. Makki and George Havas. Optimal Distributed Algorithms for Constructing a Depth-First-Search Tree. In *Proc. ICPP'94 – International Conference on Parallel Processing*, 1994.

[26] D. R. Mani. *The Design and Implementation of Massively Parallel Knowledge Representation and Reasoning Systems: A Connectionist Approach*. PhD thesis, University of Pennsylvania, 1995.

[27] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Department of Computer Science, The University of Tennessee at Knoxville, 1994.

[28] Sriram Raghavan and Hector Garcia-Molina. Representing web graphs. Technical report, Stanford University, June 2002.

[29] Kirk Schloegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. Supercomputing 2000*, pages 75–75, 2000.

[30] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. *Lecture Notes in Computer Science*, 1900:296–??, 2001.

[31] Shashi Shekhar, Chang-Tien Lu, and Pusheng Zhang. Detecting graph-based spatial outliers: algorithms and applications (a summary of results). In *Proc. 7th Int'l Conf. on Knowledge Discovery and Data Mining (KDD 2001)*, pages 371–376, 2001.

[32] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.

[33] Stuart Staniford-Chen, S. Cheung, Rick Crawford, Mark Dilger, J. Frank, Jim Hoagland, Karl N. Levitt, Christopher Wee, R. Yip, and D. Zerkle. GrIDS – A graph-based intrusion detection system for large networks. In *Proc. of the 19th National Information Systems Security Conference*, Baltimore, MD, 1996. National Institute of Standards and Technology (NIST).

[34] K. Stoffel, J. Hendler, and J. Saltz. Parka on MIMD-supercomputers. In *3rd International Workshop on Parallel Processing in AI*, Montral, August 1995.

[35] Gerard Tel. Distributed graph exploration, 1997.

[36] Gerard Tel. Distributed control for AI, 1998.

[37] Soon-Hyung Yook, Hawoong Jeong, and Albert-László Barabási. Modeling the internet's large scale topology. *Proc. of the Nat'l Academy of Sciences*, 99:13382–13386, 2002.

[38] Clara Yu, John Cuadrado, Maciej Ceglowski, and J. Scott Payne. Patterns in unstructured data: Discovery, aggregation, and visualization. Presentation to the Andrew W. Mellon Foundation, 2002.

[39] Mohammed J. Zaki. Parallel and distributed data mining: An introduction. In *Large-Scale Parallel Data Mining*, volume 1759 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, Berlin, 2000.