Relational Graph Analysis with Real-World Constraints: An Application in IRS Tax Fraud Detection

Eric Bloedorn, Neal J. Rothleder, David DeBarr, Lowell Rosen

The MITRE Corporation
7515 Colshire Dr.
McLean, VA 22102-7508
703.983.5274 (Bloedorn)

{bloedorn, neal, debarr, lrosen}@mitre.org

ABSTRACT

In this paper, we describe research and application of relational graph mining in IRS investigations. One key scenario in this domain is the iterative construction of models for identifying tax fraud. For example, an investigator may be interested in understanding variations in schemes involving individuals sending money off-shore. This domain lends itself naturally to a graph representation with entities and their relationships represented as node and edges, respectively. There are two critical constraints in this application which make it unsuitable for existing work on relational graph mining. First, our data set is large (20 million nodes, 20 million edges, in 500GB) and includes multiple types of entities and relationships. Second, due to both the size, and the active nature of this data, it is necessary to do the mining directly against the database. Extracting and maintaining a separate data store would be impractical and costly to maintain. We focus on describing our approach to one of the core tasks in this process: allowing the investigator to mine potentially illegal activity by iteratively suggesting and refining loosely defined scenarios. Our current methodology combines three components: (1) a graph representation language which allows flexibility for inexact matches, (2) custom data structures, combined with dynamically generated sequences of SQL queries, to perform efficient mining directly against the database, and (3) exploiting cost-based optimization information to help improve our results search. A prototype solution has been deployed and used by the IRS and has resulted in both identification of criminal activity and accolades for ease of use and efficacy.

1. Introduction

Among its missions, the US Internal Revenue Service (IRS) is charged with enforcing the nation's tax laws. In this capacity, it is necessary to sift through a great deal of individual and organization tax filings in an effort to identify fraudulent tax claims. According to one former Treasury official (Burman, 2003), "The IRS could net almost $28 billion from tax fraud and errors that are identified and ripe for collection." Of course, the general nature of business financial transactions, coupled with the complex tax laws, makes identifying fraud difficult. Corporation and individuals can be involved in extremely complex financial partnerships and revenue exchanges, most of which are perfectly legal. Trusts, partnerships, and subchapter S corporations are often of interest, because they don't pay taxes. Instead, the tax liability simply flows through to the beneficiaries, partners, or shareholders. Identifying subsets of these entities that are engaged in abusive tax shelters is the focus of this work. Towards that end, here are some example questions a fraud investigator might ask:

- Show me all individuals in a partnership with <companyX> (some previously identified suspect company)
- Show me all individuals in more than 10 partnerships (unusual for an individual to do this)
- Show me common filing patterns and exceptions to those (a bit more exploratory – what happens a lot, and what are the variations in this)
- Show me situations where there are other taxpayers that have a closely held partnership used to funnel money to a trust that sends the money offshore
- What role do offshore trusts play in known tax fraud?

The purpose is to find Taxpayer Identification Numbers (TINs) for individuals and entities that are involved in suspicious relationships. The results can be used to direct the investigation to relevant collections of returns. After review by domain experts, the results can be passed on as workload to revenue agents, or the results can be further analyzed and filtered using other data mining techniques for refining the target patterns.

The IRS analysts often conceptualize tax return data as large networks or graphs. Various individual and business entities form the nodes, while, for example, the K-1 reports of gains and loss allocations are edges. The approach adopted to address this situation involves relational link analysis. By representing the domain as a graph structure, it is possible to both better capture the complex relations formed in order to avoid taxes and also present this information back to the IRS investigators in a more natural way. Figure 1 illustrates one tax avoidance scheme - sending gains off-shore using tiers of flow-through entities, which include trusts, partnerships and subchapter S corporations. Note some terms that will used throughout our discussion: AGI = Adjusted Gross Income, K1 is a schedule on various forms (e.g. 1041) used to record the allocation of gains and losses for "flow-through" entities: trusts, partnerships, and S corporations.. A tax haven is a country that offers favorable tax laws for American taxpayers, with secrecy provisions preventing scrutiny by American tax administration authorities.

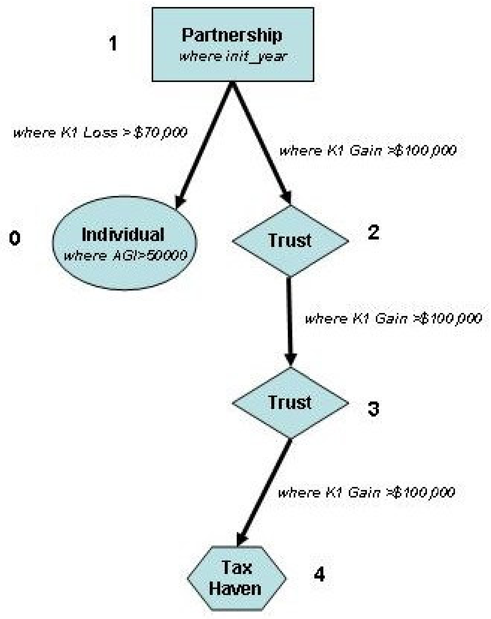


Figure 1: Example Tax Entity Graph

The investigators here are subject matter experts in tax law and investigations. They understand the power of advanced analytical tools, but have no particular expertise in databases, SQL, or data mining. They need tools which provide them an intuitive way to identify suspect activity and pursue investigations in real time.

1.1 Requirements and Constraints

The application possesses a few requirements and constraints which we found difficult to satisfy using existing graph or link analysis approaches. First, mining and analysis should take place directly against the database. Due to budget, logistics, expertise, and size of the data, it is impractical to expect to generate and work from data extracts or maintain a separate, transformed data store. Our initial prototype looked at one year of K-1 data (approximately 20 million records), capturing over 20 million nodes and 20 million edges, resulting in 500GB of storage across numerous database tables. The IRS stores multiple years of these records and hopes to use them in future analysis. Since the data grows constantly over time, static snapshots are insufficient. An additional requirement is the ability to handle multiple types of entities and relationships, as well as their attributes in the analysis. Table 1 shows some examples of the types of nodes, edges, and properties of these which need to not only be represented, but easily integrated into the analysis.

Entities (nodes)	Relationships (edges)	Attributes (properties)
Individual	Gains (in dollars)	Initial filing year
Partnership	Loss (in dollars)	Type of loss/gain
Trust	Subsidiary	Address

Table 1. Types of Nodes, Edges, and Attributes

Finally, whatever approach is adopted must have a useful interface. It must be natural to the user (not requiring specialized coding skills, for example), respond with results in a few minutes, and provide context around solutions (rather than a simple list of suspect names, for example).

It is important to note that there are many other real-world applications that share some or all of these requirements. For example, other law enforcement and intelligence services have similar needs and constraints in their investigations. The tools and methodology described in this paper are easily transferable.

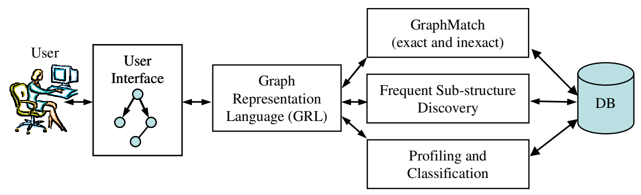
1.2 The General Framework

With the requirements of this application area in mind, we are developing the REGAL framework (RElational Graph AnaLysis) as a research and application platform. Figure 2 illustrates the current high-level plans for this framework. It currently calls for five main components sitting between the user and the database.

REGAL is being investigated and prototyped in such a way as to allow staged deployment and usage. In the remainder of this paper, we focus on two specific components:

Graph representation language – simple design which also allows flexibility for inexact matches

GraphMatch – Custom data structures, implemented in Perl, combined with dynamically generated sequences of SQL queries, to perform efficient mining directly against the database. This approach also makes use of cost-based optimization information, provided from the relational Database Management System (DBMS), to help improve the results search.


User
User Interface
Graph Representation Language (GRL)
GraphMatch (exact and inexact)
Frequent Sub-structure Discovery
Profiling and Classification
DB

User Interface	A graph-based graphical user interface which facilitates investigations about tax fraud.
Graph Representation Language (GRL)	Intermediate representation of general graph structures. Permits incomplete graph specifications (for inexact matching) as well as general graph queries.
GraphMatch	Supports the search and identification of both exact and inexact matching to a given sub-graph structure.
Frequent sub-structure discovery	Identifies frequently occurring graph sub-structures. Supports the identification of loosely defined structures.
Profiling and Classification	Given a group of graph structures sharing some known label (e.g., fraud), identifies distinguishing features, substructures, and attributes.

Figure 2: REGAL Framework and Components

2. Related Work

Graphs representations are extremely general and widely studied data structures from mathematics (Tutte, 1984), (McKay, 2005) to social network analysis theory (Wasserman, 1994) to software engineering (Hall and Kennedy, 1992). Despite this work, some seemingly basic operations on graphs are non-trivial. For example, a 'simple' operation like matching a small graph against a larger graph (subgraph isomorphism) belongs to the class of NP-Complete problems (Garey and Johnson, 1979).

A number of researchers and vendors provide tools that address some aspects of this problem. The SRI LAW project (Berry et al., 2004) demonstrates a platform with similarities to REGAL. Their graph matching representation is somewhat different, enabling a hierarchy of classes in the nodes, and permitting a more limited vocabulary of constraints. One key constraint of LAW is its limited ability to scale, however. The authors estimate query times of approximately 1.6 hours for a graph containing seven million edges. This is an unacceptable response time for realistic usage (the authors note this). (Blau, Immerman, and Jensen, 2002) also describe a similar graph mining system which includes a very rich graph description language (including both query and update). One key way in which they gain greater scalability is to retrieve elements of the database only as they are needed. To do this, they employ Proximity, a system that uses an open source vertical DBMS called MonetDB to store and retrieve graph data via the QGraph language. Unfortunately, in the types of domains that REGAL is designed for, it is important for deployment to use the RDBMS already in place. (Kuramochi and Karypis, 2004) have looked at the problem of finding frequent subgraphs in large graphs, but do so for graphs that do not include directed edges, and they search for non-overlapping subgraphs. IRS investigators want to find all matches of possible fraud regardless of edge overlap I2's Analyst Notebook (http://www.i2.co.uk/Products/Analysts_Notebook/) and Visual Analytic's Visual Links (http://www.visualanalytics.com/Products/Visualinks.cfm) are popular tools for storing and visualizing information about entities and relationships. They provide connectivity to a database for storage, but do not support the querying for subgraphs within a larger graph that we are describing here. There are also some commercial tools which support graph analysis. A number of commercial and research efforts are underway to support storage and traversal of tree structured data (e.g. Oracle supports for querying tree structures using the 'connect-by' clause) and (Comai et al., 1999). However, this feature is not useful for full graphs with cycles, or bi-directional paths which are common in tax data.

We believe the ability to exploit available domain knowledge in relational representations will enable us to achieve greater scalability, improve predictive accuracy and find more interesting results. Others have noted that available domain knowledge is useful for constraining problems and making them more tractable: (Kuramochi and Karypis, 2004) note that their FSG (Frequent SubGraph discovery) algorithm is more efficient for translating graph structures to simpler representations when edge and vertex labels are present. ILP methods, such as (Dzeroski, 2003), are well suited for exploiting available domain knowledge, but this was not pursued because of the database connectivity requirement and the size of our domain.

3. Graph Representation and Graph Matching

Taken together, the graph representation language (GRL) and graph matching (GraphMatch) components make a natural first phase of implementation within the REGAL framework. They can be implemented and tested in a stand-alone way and still provide a generous value-add beyond the capabilities of the IRS investigators. The goal of GRL and GraphMatch is a general scenario description language and accompanying execution engine, that mirror the expert's thinking. The IRS experts regularly employ a link/node (graph) conceptualization, and then mine the database by "querying" a wide variety of general scenarios. The goal is to identify both specific Taxpayer Identification

Numbers (TINs) involved in likely fraud, as well as related, but characteristically different, scenarios used to commit fraud. With GRL and GraphMatch, this is a relatively simple and fast process. The remainder of the section describes GRL, the GraphMatch algorithm, and the associated performance considerations.

3.1 Graph Representation Language

The representation language used in REGAL is called Graph Representation Language (GRL). It is used as the canonical storage of all graph and graph queries used by the REGAL components. It serves as an abstraction layer for the database storage, and as a common communication protocol between REGAL components. All user input is translated into GRL, and graph output from the mining components are returned in GRL format. It is quite straightforward, easy to read, and easy to dynamically construct. In addition to these obvious needs, GRL was designed for this application with two additional requirements in mind: (1) It must allow representation of underspecified graphs, (think of these as *descriptions* of graph classes). In this way, graph queries can be constructed to represent, for example, "A partnership node linked within 3 edges to a trust node with at least $3000 in earnings". (2) It must require minimum translation into SQL in order to facilitate interaction of queries with the database. All the components in REGAL use dynamically constructed SQL to retrieve data directly from the database. It is critical, therefore, that the representation language we use translate relatively seamlessly. As discussed in Section 2, none of the existing graph representations are designed with this functionality in mind.

The GRL language is fairly straightforward: one line is used for each node and one line for each edge. The syntax of the various GRL statements is shown below in Table 2. For example, this statement

```
n 0 individual where AGI > $500,000
```

refers to a NodeType of individual with a constraint on the node attribute AGI. The various NodeTypes are defined in an external configuration file and they are associated with the actual database tables containing individual tax return data. The NodeCondition specified in Table 2 follows the syntax of the where clause of an SQL select statement.

To illustrate GRL further, consider the scenario shown previously in Figure 1. It can be represented with the following GRL code:

```
n 0 individual where AGI > 500000
n 1 partnership where INIT_YEAR
n 2 trust
n 3 trust
n 4 haven
d 1 0 k1 where LOSS > 70000
d 1 2  k1  where GAIN > 100000
d 2 3 k1 where GAIN > 100000
d 3 4 k1  where GAIN > 100000
```

The lines beginning with "n" define the nodes in the graph structure, and each node is assigned a unique reference number.

GRL statement type	Statement Syntax
Node	n *nodeID nodetype* where *nodeCondition*
Directed Edge	d *fromNodeID toNodeID edgetype[:n:m]* where *edgeCondition*
Undirected Edge	u *nodeID1 NodeID2 edgetype[n:m]* where *edgeCondition*
"No" edge	!d *fromNodeID toNodeID edgetype* where *edgeCondition*
Node-to-Node join	j *nodeID*1 *nodeID2* where *joinCondition*;
Node-to-Edge join	j *nodeID edgeID* where j*oinCondition*
Edge-to-Edge join	j *edgeID1 edgeID2* where *joinCondition*

node ID is a unique integer.

nodeType and *edgeType* are domain specific. They are defined by metadata in a configuration file, mapping GRL type names to database table names.

fromNodeID and *toNodeID* are integers, corresponding to source and target nodes connected by an edge.

nodeCondition and *edgeCondition* follow the syntax of a SQL "where" clause. These conditions will be copied into the dynamically generated SQL statements at runtime.

joinConditons follows SQL syntax for a table join, except that #nodeID or #edgeID is given in place of database table name.

Indirect paths are indicated by [min:max] following the *edgeType* declaration. The interval specifies the minimum and maximum number of edges (of *edgeType)* which may be traversed.

Table 2: GRL Syntax

The lines beginning with “d” define directed links with the from/to node numbers indicated. Attribute constraints for both links and nodes can be indicated by a clause identical in syntax with the SQL where clause. The “d 1 0” edge specifies there is a payment between a partnership and an individual, and that the total loss reported is greater than $70,000. The “d 1 2” link describes a link between the partnership and a trust, and likewise the "d 3 4" link describes the link between a trust and an offshore tax haven

In addition to directed links designated by a “d”, undirected links can be specified in the pattern be using a “u”. The remaining specification is identical. Undirected links allow a match regardless of the direction implied in the underlying data.

3.1.1 Wildcards

The previous example involved a relatively simple scenario in which most of the necessary graph structure was specified. Due to the nature of the IRS investigations, it is necessary to support looser graph structure definitions.

A wildcard, “*”, can be used in place of the NodeID to refer to neighboring nodes in the specified or returned graph structures. It permits REGAL to represent statements about edge characteristics around a node. For example, the GRL statement

```
d * 2 k1 where GAIN>1000 and count(*)>3
```

describes a scenario in which node 2 has at least 4 incoming edges, of type k1, each with a gain greater than $1000. This is illustrated in a graph representation in Figure 3.

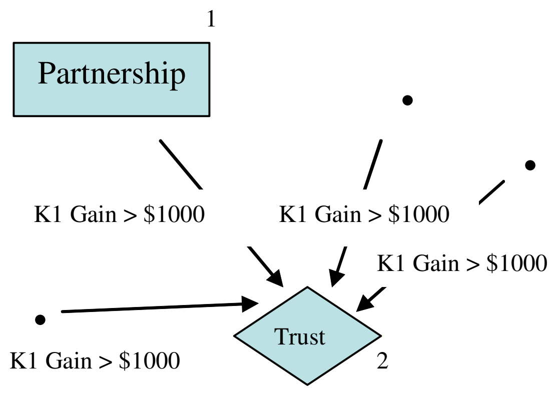


Figure 3: Example match using a wildcard

These wildcards can also provide a means of referring to the aggregate data describing the neighborhood about a node. For example, the GRL statement

```
d * 2 k1 having sum(GAIN) > 30000
```

captures sub-structures in which a node (2) has incoming k1 type links whose aggregate gain is greater than $30000. Notice the use of SQL’s “group by.. having” syntax in GRL (illustrated in Figure 4).

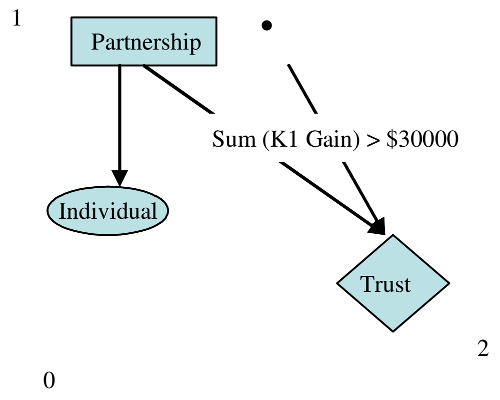


Figure 4: Example match using wildcards and

3.1.2 Indirect Paths

In the examples thus far, all links between nodes involve a single edge. A more interesting example might include *indirect* paths: those involving links between nodes consisting of an arbitrary path length. For example, the payment from a partnership might pass through any number of flow-through entitles before reaching an off-shore tax haven. Thus there is an “indirect” linkage from the partnership to the tax haven.

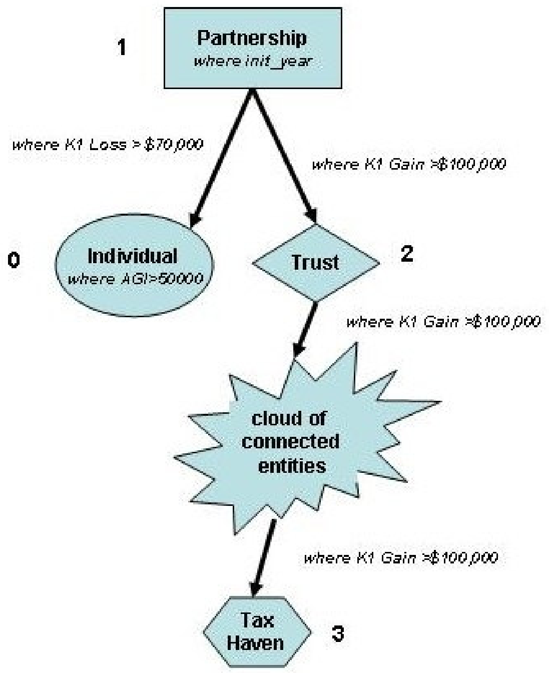


Figure 5: Example of indirect matching on edges

This variation is difficult to express with pure SQL. However, consider the following GRL example:

```
n 0 individual where TPI > 500000
n1 partnership where INIT_YEAR
     and FINAL_YEAR
n2 trust
n 3 haven
d 1 0 k1 where LOSS > 70000
d 1 2 k1 where Gain > 100000
d 2 3 k1:1:5  where GAIN > 100000
     and CLOSE=1
```

Of particular note is the last line specifying a directed edge from node 2 to node 3. Notice the colon notation after the NodeType specification "*k1:1:*5". This describes any sequence of length 1-5 of type k1 edges.

3.1.3 Attribute Constraints

A GRL description can also constrain some relationship in the *attributes* between a pair of nodes. For example, it is fairly common to describe a situation in which the address and zip code of two entities are identical. With GRL this is accomplished with a join statement, denoted by "j". Here is an example:

```
n 0 partnership
n 1 scorp
n 2 individual
n 3 individual
d 0 2 k1 where loss < -10000
d 1 3 k1 where loss < -100000
j 0 1 where #0.ZIP = #1.ZIP
     and  #0.ADDRS = #1.ADDRS
```

This is a situation where an individual is linked to a partnership with K-1 losses, another individual is linked to an S-corporation with K-1 losses. The final GRL statement, the join, specifies that the partnership and S-corporation have the same address and zip-code. Notice that the format of the join condition closely resembles a join in SQL. This capability in GRL is useful, for example, because it helps describe an initial "scenario of interest" for an inflated basis transaction. These are known as the Son of Bond and Option Sales Strategies (BOSS), where a break-even straddle is used to generate large losses offsetting gains from another source. While this scenario can be generalized, it represents a starting point provided earlier by the domain experts (assuming use of absolute value for partnership allocations).

3.2 GraphMatch

GraphMatch is the component within REGAL that is responsible for matching graph structures. Note that these are any graph structures that can be represented in GRL. Thus, they need not be fully-specified nodes and edges. They can be classes of graphs, or queries for types of graphs. In graph terminology, GraphMatch finds subgraphs of the database that are isomorphic to the target graph scenario. Of course, we're dealing with a looser definition of isomorphism than is generally used.

Since this application, like many others, has graph structures too large to fit in main memory, and it is not practical to transform and maintain the entire database in some alternative, optimized form, GraphMatch makes a series of direct calls to the database through dynamically generated SQL queries. GraphMatch is currently written in Perl, and works with the Oracle RDMS through the DBI interface. These specific choices of Perl and Oracle are somewhat arbitrary, however, and the techniques and algorithms described here can be implemented in a variety of programming languages coupled with other SQL-based databases.

The basic algorithm is a search of the "main" graph (all the tax information in the database) via repeated expansion of currently matched nodes. Of course, implementing a brute-force exhaustive search of the full graph is extremely inefficient in practice, especially for large databases (Washio and Motoda, 2003). In order to improve performance, GraphMatch uses a number of heuristics to prune the search space and generally speed performance.

1. Best-first search, with back-tracking, is used for a more organized, efficient search.
2. SQL calls to the database include added constraints to prune the number of nodes added to the search queue.
3. Cost-based optimization information is exploited to help rank the nodes in the search queue.
4. Domain-specific rules are added to the search to help prune the search-space.

Summarizing intuitively, GraphMatch implements the overall search strategy (ordering nodes, managing partial solutions, generating output, etc.) while exploiting the database engine (Oracle) to evaluate the goodness of potential next nodes and retrieve candidate matches to those nodes which satisfy the criteria imposed by the user scenario.

3.2.1 Best-First Search with Back-Tracking

Best-first search with back-tracking is a general technique for organizing the exhaustive search for a solution to a combinatorial problem (Valiente, 1988). The technique consists of keeping a queue of nodes, ordered according to some "goodness" criteria. In each step of the search, the best node is selected to expand in the search-tree. This repeatedly extends a partial solution to the problem, satisfying certain constraints, and shrinking the representation of the partial solution whenever a partial solution cannot be further extended. The extension of the partial solution is done by calling a recursive procedure to fill each new candidate node.

In this manner GraphMatch grows matches to the target graph, one node at a time. At the same time it grows the connecting links to all previously solved nodes. Failure to grow a new node (i.e., failure to find any matching structures in the existing graph matches thus far) results in going back to the previous level in the solution tree and going down the next branch. When all the nodes are filled a complete solution is found. This is recorded and the algorithm backtracks to the previous level and resumes.

3.2.2 Adding Constraints to Individual SQL Queries

At each step of the match process, GraphMatch attempts to grow the existing partial solutions by one node. This involves finding candidates in the database that meet all the constraints imposed by both the next, best node selected (see below for how this is done), and those imposed by the partial solution grown to that point.

GraphMatch finds all the candidates for the individual nodes by executing a program-generated SQL *select* statement on the table hosting the instance data for the type of node being filled. It uses the conditions in the where clause for the GRL definition of the node. In addition to the node constraint, the existence of edges or paths (e.g. K-1 payments) to previously matched nodes must also be satisfied. Candidates for subsequent nodes are found with a SQL select statement with instance values of previous node/link attributes substituted in for SQL bind variables. Scenario patterns specified with *indirect* paths (see Section 3.1.2) require additional branching between the two levels solution tree representing the beginning and endpoint of the path.

3.2.3 Exploiting Cost-Based Information

Of course, in order to implement best-first search, it is necessary to have a goodness function for ordering nodes in the search queue. In ad-hoc experimentation of arbitrary goodness functions, runtimes of GraphMatch (called with identical arguments in the IRS dataset) varied between a few minutes to a day or more. In informal observation, the most efficient node ordering begins with the node with the least number of candidates (adjacent nodes with matching constraints) in the database. In GraphMatch, this critical information was derived using Oracle's native cost-based optimization function, Explain Plan.

GraphMatch implements a heuristic node goodness function via this Explain Plan information. Calls are made to the Oracle optimizer to determine the lowest cardinality node (fewest rows returned). This is selected as the first node. All possible order permutations with this as the first node are generated and each possible ordering is evaluated as a weighted sum of the total CPU costs for all of the node queries that would be required to complete the solutions, if that ordering was used. The ordering (permutation) that has the lowest *total* cost is then selected and used to execute the match. The weighted sum assumes that fewer candidates remain near the bottom of the tree. This heuristic approach seems to work well in the IRS data set.

3.2.4 Domain-Specific Constraints

The final heuristic exploited by GraphMatch is the utilization of domain specific knowledge. For example, in the world of tax fraud, a node (entity) which is connected to a very large number of other nodes (entities) is typically not of interest. Consider, for example, large home mortgage lending institutions. These will obviously have connections to an enormous number of individuals. With a standard SQL join approach, expanding these institutional nodes in the graph, and exploring potential isomorphic matches among all its neighbors would be prohibitively expensive. Logic bypassing these sections of the graph is coded into GraphMatch to speed up execution.

3.3 Achieving Good Performance through Cost-Based Execution Ordering

The order in which the nodes are filled has a big effect on the overall execution time. For example, within the IRS dataset, the same query can vary between a few minutes to a day or more, just by the order in which the nodes are processed by the backtracking code. Usually the most efficient sequence is to start with the node that has the least number of candidates. The next node filled (solution tree level 2) must be adjacent. The expected runtime (CPU cost) or expected rows returned (cardinality) computed by Oracle's Explain Plan is important information for choosing the order.

GraphMatch implements a heuristic node order algorithm using this information. Calls are made to the Oracle optimizer (Explain Plan) to determine the lowest cardinality node (fewest rows returned). This is selected as the first node. All possible order permutations with this as the first node are generated and each possible ordering is evaluated as a weighted sum of the total CPU costs for all of the node queries that would be required to complete the solutions, if that ordering was used. The ordering (permutation) that has the lowest total cost is then selected and used to execute the query. The weighted sum assumes that fewer candidates remain near the bottom of the tree. This heuristic approach seems to work well in the IRS data set. However, the optimality of this approach is not proven mathematically and so other approaches for ordering might also work as well or better.

4. Prototype and Field Testing

The two components described in Section **Error! Reference source not found.** (GRL and GraphMatch) were deployed to the IRS recently. They are being used by a small number of analysts, but already they credit this tool with helping to identify entities suspected of abusively sheltering millions of dollars from taxes. For obvious reasons of security, details of the preliminary trial of REGAL cannot be discussed.

5. Summary and Future Work

The REGAL framework brings new capabilities to practical IRS fraud detection and other domains which share a relational graph representation. Beyond the critical capabilities of intuitive interface, and real-time results processing, REGAL is designed to overcome some common limitations of existing technology when faced with real-world constraints. In particular, REGAL handles extremely large data sets, integrates directly with a relational database, represents multiple types of nodes and edge, and allows very flexible, loosely defined graph structures to be defined, queried, and mined.

The Graphical Representation Language (GRL) forms the basis for storing and communicating graph information among all the REGAL components and the user interface. It is based in a very simple syntax to define typical node and edge properties. It permits SQL syntax in many places, however, in order to permit more complex descriptions – including constraints on node attributes, edge attributes, and aggregations among neighborhoods of nodes. The GRL canonical form is compact, and is easily transformed at run-time into SQL queries that can be passed directed to the database.

REGAL's GraphMatch component is responsible for finding structures in the database which match given target graph scenarios. It uses a depth-first search, with backtracking, in which dynamically generated SQL queries are made to the database to retrieve node information. In order to improve performance, various domain-specific constraints are built in, and the cost-based information from the database is used to selected the nodes likely to prune the search space by the greatest amount.

Future work on REGAL will include research and development on the remaining core components. A prototype graphical user interface (GUI) is already in progress. This will permit much more user independence. Work has also begun on algorithm design for the Frequent Sub-structure Discovery. Of course, this component will work directly with the database and make use of the GraphMatch component. While our practical results and feedback have been remarkable, and the informal performance evaluation and testing that has been completed thus far has been thorough enough to build confidence for our current approach, it is worthwhile to explore a more rigorous performance evaluation and testing, particularly as compared with alternate algorithms and existing tools. Limiting this testing thus far has been acceptable as the focus of this work is on advancing the capabilities of the field in real-world, practical applications.

A prototype system of GRL and GraphMatch has been deployed and used by IRS in their fraud investigations. It has resulted in both identification of criminal activity and accolades for ease of use, value-add, and efficacy. It is easy to envision how this work could be applied in other domains such as law enforcement and intelligence analysis.

6. Acknowledgements

Dave DeBarr was partially supported in this work by a contract from the IRS under contract TIRNO-99-D-00005 task 103. The remainder of the work was supported through MITRE internal research funding. The views and conclusions contained in this paper are solely those of the authors.

7. References

Berry, P., Harrison, I., Lowrance, J., Rodrigues, A., Ruspini, E., Thomere, J., and Wolverton, M., "Link Analysis Workbench, SRI International". Technical Report: AFRL-IF-RS-TR-2004-247, Air Force Research Laboratory, September, 2004.

Blau, H., Immerman, N. and Jensen, D. *A visual language for querying and updating graphs.* University of Massachusetts Amherst Computer Science Technical Report 2002-037. 2002.

Burman, L., "Statement of Leonard E. Burman before the United States House of Representatives Committee on the Budget; On Waste, Fraud, and Abuse In Federal Mandatory Programs", July 9, 2003. http://www.taxpolicycenter.org/publications/template.cfm?PubID=900641

Comai S., Damiani, E., Fraternali, P., Paraboschi, , S., Tanca, L., "XML-GL: a Graphical Language for Querying and Restructuring XML Documents", Computer Networks, Vol. 31 (1999).

Dzeroski, S., "Multi-Relational Data Mining: An Introduction", SIGKDD Explorations Newsletter of the ACM Special Interest Group on Knowledge Discovery and Data Mining, Vol 5, Issue 1, July 2003

Garey, M. and Johnson, D. *Computers and Intractability: A guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

Hall, M. W. and Kennedy, K. *Efficient call graph analysis.* Letters on Programming Languages and Systems 1, 3. http://citeseer.ist.psu.edu/hall92efficient.html, 1992.

Kuramochi, M., and Karypis, G., "Finding Frequent Patterns in a Large Sparse Graph", Proceedings of the Fourth SIAM Data Mining Conference, 2004

McKay, B. *Nauty User's Guide (version 2.2)*, http://cs.anu.edu.au/people/bdm/nauty/nug.pdf. Computer Science Department, Australian National University, ACT 0200, Australia, 2005

Tutte, W.T. *Graph theory.* Encyclopedia of Mathematics and its Applications v 21, Addison-Wesley Publishing Co., Reading, Mass., 1984.

Valiente, G. *Algorithms on Trees and Graphs*, Spring-Verlag, 1988.

Washio, T and Motoda, H. *State of the Art in Graph-based Data Mining*. SIGKDD Explorations Newsletter of the ACM Special Interest Group on Knowledge Discovery and Data Mining, Vol 5, Issue 1, July 2003

Wasserman, Stanley and Katherine Faust. *Social Network Analysis: Methods and Applications.* Cambridge: Cambridge University Press. 1994