

Rank Semantic Model of Relational Data Source: Metrics and Evaluation

**ACRC, University of South Australia**

**Version: 0.1**

**Data to Decisions CRC**

Level 1, Base 64

64 North Tce

Kent Town SA 5067

Ph: +61 8302 8978

info@d2dcrc.com.au

www.d2dcrc.com.au

**Table of Contents**

1. Introduction 4

2. Motivation Example 6

3. Problem Definition 11

4. Our Approach 13

4.1 Extract schema mapping patterns **Error! Bookmark not defined.**

4.2 Generate and Rank Models 14

5. Prototype Error! Bookmark not defined.

6. Evaluation 18

7. Related Work 33

8. Conclusion 34

9. Reference 35

# Introduction

The problem of integrating heterogeneous data sources into an ontology is highly relevant in the database field. Our solution extends Karma – a data modeling and integration framework[[1]](#footnote-1). User can import data from a variety of sources including relational databases, spreadsheet, XML files and JSON files into Karma. They can also import the domain ontologies they want to use for modeling the data. The system then automatically suggests a semantic model for the loaded source. Karma provides an easy-to-use graphical user interface to let users interactively refine the learned semantic models if needed. Once a semantic model is created for the new source, users can publish the data as RDF by clicking a single button. Karma is used to model the data from Smithsonian American Art Museum[[2]](#footnote-2) and then publish it into the Linked Open Data cloud.

Karma exploits the knowledge from a domain ontology and *known semantic models* of sources in the same domain to automatically learn a rich semantic model for a new source. Given sample data from the new source, they use labeling technique to annotate each source attribute with a set of candidate semantic types from the ontology. Next, they build a weighed directed graph from the known semantic models, learned semantic types and domain ontology. This graph models the space of plausible semantic models. Then, we find the most promising mapping from the source attributes to the nodes of the graph, and for each mapping, they generate a candidate model by computing the minimal tree that connects the mapped nodes. Finally, they score the candidate models to prefer the ones formed with more coherent and frequent patterns.

The relationship recovery method of Karma does a good work when CDM (domain ontology) is much larger than data source (normally a data source means a single table in Karma). In this scenario, the frequency of edges in CDM is sufficient to determine a correct path or edge between two entities of a new data source. However, in the ILE project, the size of CDM is almost similar with a specified data source as there are a lot of multiple join table and entities in it. Simply depending on frequency of edges will not yield a correct semantic model for a new data source.

With regard to the above consideration, in this work, we develop a new method to recover the relationships between recognized entities. Instead of using frequency of edges, we extract all of schema mapping patterns that are frequently happening, and then put them into the match between a new data source and CDM. We try to compare the structure of join tables of new data source and the known data source for the maximum reuse of the extracted schema mapping patterns. Finally, the relationships are recovered and semantic model is created for the new data source.

# Motivation Example

Here is part of our common data model.



**Fig. 1.** Part of our common data model

Considering there are three data sources, namely S1, S2 and S3, that are mapped to the CDM.



1. Data Source *S1* and its mapping to CDM



1. Data source *S2* and its mapping to CDM



1. Data source *S3* and its mapping to CDM

**Fig. 2.** CDM and three data sources (S1, S2, S3) that mapped to CDM. Class nodes (ovals) and links correspond to classes and properties in the CDM.

# Preliminary

The general framework for relational-to-ontology schema mapping is decomposed into two parts [18]. First, *semantic labeling* finds correspondences between attributes from data sources and data nodes of the target ontology. Second, in the *schema mapping* part we want to generate the semantic models of data sources by identifying the connecting paths for the matched data nodes.

## Semantic Labelling

Training component aims to provide a basic knowledge base for semantics extraction from target data sources to CDM data lake and semantic typing. All known data sources, i.e. are annotated and mapped to the CDM manually, generating mapping files.

The training process consists of four main steps. The first step, Assign Semantic Types, involves mapping each column of source to a node in the ontology. This is a user-guided process where the system assigns types automatically based on the data values in each column and a set of learned probabilistic models constructed from assignments done in prior sessions. If the semantic type assigned by the system is incorrect, the user can select from a menu the correct node in the graph. The system learns from this assignment and records the learned assignment in its database. Conditional random field (CRF) [7] is exploited to learn a labeling function. 21 discriminative features are defined to test whether the name, value or the resulting tokens have a particular feature. The most commonly instantiated ones are:

* nameContainsToken(X),
* nameStartsWith(X),
* valueContainsToken(X),
* valueStartsWith(X),
* valueHasCapitalizedToken(),
* valueHasAllUppercaseToken(),
* valueHasAlphabeticalTokenOfLength(X),
* valueHasNumericTokenWithOrderOfMagnitude(X),
* valueHasNumericTokenWithPrecision(X),
* valueHasNegativeNumericToken().

The output of the labeling step for is , where in , is the semantic type learned for the attribute and is the associated confidence value which is a decimal value between 0 and 1. After the candidate labels are provided, end-users will determine the semantic type for each column manually.

Before we recover the whole semantic model for the data source, we first construct a graph G defining the space of all possible mappings between the source and the CDM, and the most popular patterns connecting the candidate types. To achieve that, we reuse some methods in Karma [8]. The steps for constructing the graph G is depicted in the following. (1) all of the known semantic models are merged into a consolidated one. (2) Once the known semantic models are added to G, we add the semantic types learned for the attributes of the target source. (3) We use the CDM to find all the paths that relate the current class nodes in G. It is significant to assign weight to edges in the graph G. The factor we consider to weighting the edges coming from the known semantic models is the popularity of the edges, i.e., the number of known semantic models supporting the link. We assign to each edge where is the number of known semantic models and is the number of identifiers the edge is tagged with. We assign to the edges that are introduced directly from the CDM in the step (2). is where is the number of edges in G.

Now, the problem is that we need to figure out the semantic data model for a new data source . Assume that we have already semantic annotated all the columns of by using existing schema matching approaches, for an instance, conditional random field (CRF) [7]. Then, by applying schema mapping patterns, we recover all the relationships between entity types and finally return the semantic data model of .

## Schema Mapping

# Problem Definition

**Definition 1:** Relational Table

A relational data table is defined as a collection of ordered pairs where denotes an attributes name (e.g. , etc.) and denotes the set of data values corresponding to the attribute (e.g., if is , the set will have values like “07-12-1980”, etc.)

**Definition 2:** Semantic Data Model

Let semantic data model be a four-tuple .

is the set of entity nodes,

is the set of data nodes,

is the set of directed data property links, of which the starting nodes are entity nodes and the end nodes are data nodes,

is the set of directed object property links, of which the starting nodes are entity nodes and the end nodes are other entity nodes.

**Definition 3:** Distinguished Semantic Data Model

Let be a semantic data model. For any two entity nodes , if there exists and only exists one directed object property between and , is a distinguished semantic data model.

**Definition 4:** Data Source

A data source is a five-tuple . is a collection of relational data tables, i.e. , is a set of row data value around an entity, such as Person, in all of relational tables. represents a N-tuple {. represents a row of data value in the table . is the semantic model, is an attribute mapping function that connects the source to the model, and can be written as a conjunctive query over the predicates of the federated semantic data model , is the binary relations between relational data tables, i.e. .

Let be a federated semantic data model corresponding to the data sources . Given a new data source , in which are the relational data tables, are different rows of data distributed in different tables, and are the relations between different row data and .

# Our Approach

## Framework



**Fig. 3.** Overall approach of relationship recovery for multiple join tables

The overall relationship recovery method for multiple join tables is shown in Figure 3. The framework consists of two components, i.e. (1) *subgraph matching* module, and (2) *Attribute value statistics* module.

### Property knowledge graph

### Value-based Pattern graph

A *Value-based Pattern Graph* is defined as a tuple , where

* and are the set vertices and the set of directed edges, respectively;

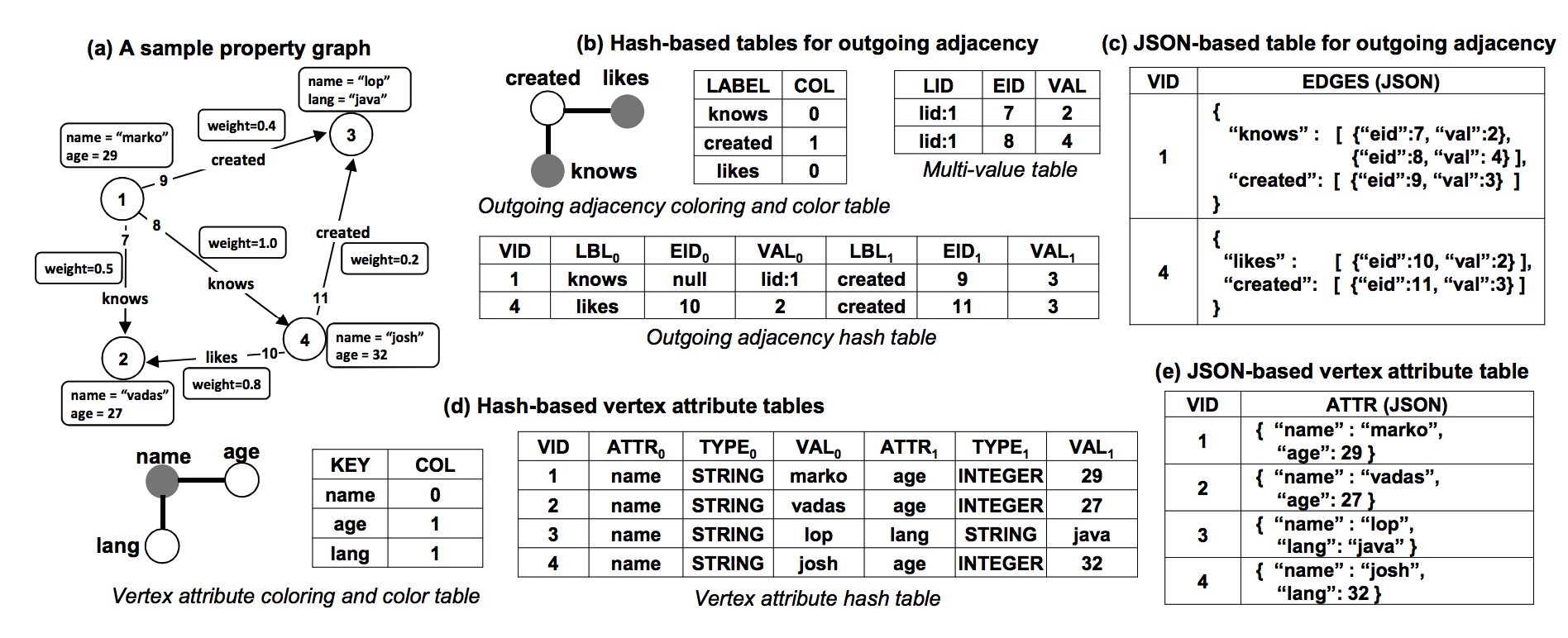


Fig. 36. Value-based Pattern Graph

## Scoring Mechanism

If we use Steiner tree algorithm to obtain a set of semantic models for each single table or a number of join tables, it is necessary to sort these candidates. In our work, we sort them according to the *frequency* that each candidate semantic model – if we regard it as a sort of *graph pattern* – appears in the *entity linking graph*.



**Fig. 4.** Scoring mechanism of candidate semantic models

## Matching Algorithm

We can model an entity linking graph as a large graph, where vertices represent entity types (Person, Location, Organization and so on) and edges represent the relationships between them.



Fig. xxx. Pattern match query in entity linking graph

**Definition 1** Let be a subgraph of a graph . A subgraph isomorphism of to is an injective function satisfying (a) for all nodes , and (b) and for all edges .

How to match a pattern into big graph could be reduced to a constraint satisfaction problem (CSP). CSP is represented as tuple where (a) is an ordered set of variables, (b) is a set of domains corresponding to variables , and (c) is a set of constraints between the variables in X. A *solution* for the CSP is an assignment to the variables in , such that all constraints in are satisfied. The subgraph isomorphism problem can be mapped to a CSP as follows.

**Definition 2** Let be a subgraph of a graph . The subgraph to graph CSP, is a CSP where:

1. contains a variable for every node .
2. is the set of domains for each variable . Each domain is a subset of .
3. Set contains the following constraints:
   1. , for all distinct variables .
   2. , for every variable .
   3. , for all such that .

In the following section, we consider the optimization of our relationship recovery algorithm above, especially that computes subgraph isomorphism.

* Push-down pruning

The subgraph generation tree is constructed by extending a parent subgraph with *another substructure[[3]](#footnote-3)* at a time. Since the parent is a substructure of its children, those assignments that were pruned from the domains of the parent, cannot be valid assignment for any of its children.

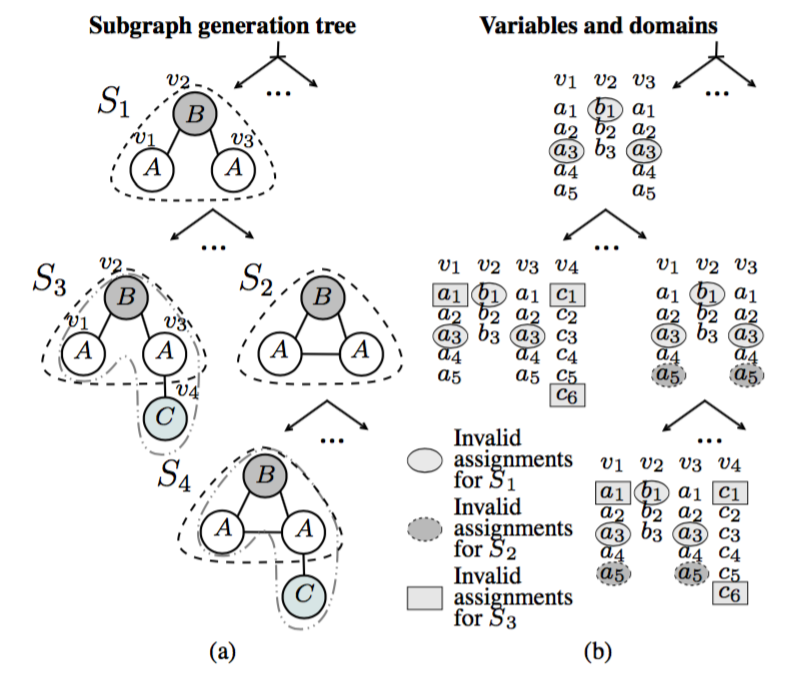


Fig. XXX. Just for demo. Will be REPLEASED

# Evaluation

# Related Work

# Conclusion

­­­

# Reference

1. Anuj Jaiswal, and David J. Miller, Uninterpreted Schema Matching with Embedded Value Mapping under Opaque Column Names and Data Values, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 22, No. 2, 2010.
2. C. E. Miller, A. W. Tucker, and R. A. Zemlin, Integer Programming Formulation of Traveling Salesman Problems, *Journal of the ACM (JACM)*, vol. 7, no. 4, pp.326-329, 1960.
3. M. Taheriyan, et al., Learning the semantics of structured data sources, *Journal of Web Semantics*, January, 2016.
4. R2RML: RDB to RDF Mapping Language, W3C Recommendation 27 September 2012. Available at: <https://www.w3.org/TR/r2rml/>
5. P. Winter, Steiner Problem in Networks – A Survey, Networks 17 (1987) 129-167.
6. G. Bhalotia, A. Hulgeri, C. Nakhe, et al., Keyword Searching and Browsing in Databases Using BANKS, in: Proceedings of the 18th International Conference on Data Engineering, 2002, pp. 431-440.
7. Laerty, J., McCallum, A., Pereira, F.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: Proceedings of the Eighteenth International Conference on Machine Learning. pp. 282–289 (2001).
8. Mohsen Taheriyan, Craig A. Knoblock, Pedro Szekely, et al., Learning the Semantics of Structured Data Sources, In: Journal of Web Semantics, Vol. 37-38, pp. 152-169, 2016.
9. Rob Baker, Alena Frankel, and Peter Rodgers, Dover: Scalable Algorithms for graph mining in java, Lot 5 of Data Analytics FY 16/17, Final Technical Report: TR UoK/Dstl 2017/1, available at: <https://www.cs.kent.ac.uk/projects/dover/>
10. Shijie Zhang, Jiong Yang and Wei Jin, SAPPER: Subgraph indexing and approximate matching in large graphs, In: the 36th International Conference on Very Large Data Bases (VLDB 2010), 2010.
11. Remco Dijkman, Marlon Dumas and Luciano Garcia-Banuelos, Graph Matching Algorithms for Business Process Model Similarity Search, In: BPM’09.
12. Haibin Liu, Lawrence Hunter, Vlado Keselj, and Karin Verspoor. Approximate Subgraph Matching-based Literature Mining for Biomedical Events and Relations. PLOS ONE, 8:4 e60954, 2013.
13. Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, et al., GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph, In: VLDB 2014.
14. Lei Zhou, Lei Chen, and M. Tamer Ozsu, Distance-Join: Pattern Match Query in a Large Graph Database, In: VLDB 2009.
15. Wajdi Dhifli, Sabeur Aridhi, and Engelbert Mephu Nguifo, MR-SimLab: Scalable subgraph selection with label similarity for big data, *Information Systems* 69 (2017) 155-163.
16. Guanfeng Liu, Yi Liu, Kai Zheng, et al., MCS-GPM: Multi-Constrained Simulation Based Graph Pattern Matching in Contexual Social Graphs, In: *TKDE*, Vol. 6, No. 1, 2017.
17. Diego De Una, et al., Machine Learning and Constraint Programming for Relational-To-Ontology Schema Mapping, In: Proceedings of 27th International Joint Conference on Artificial Intelligence (IJCAI-18), pp.1277-1283.
18. Doan et al., Principles of data integration, 2012.

# Appendix

## Generation of knowledge graph based the CDM

In order to evaluate how our method can improve the accuracy rate comparing to just Steiner tree algorithm. We try to create a synthetic knowledge graph based on a given common data model.

**Step 1: Prepare for the CDM**



**Fig. 104.** A small CDM

A small CDM is described in Figure xx where a random knowledge graph is generated based on. There are 5 entity types define in this CDM, i.e. Person, Location, Bank Account, Financial Department and Vehicle. A person can open one or more bank account. A bank account is only opened by a person. A financial department pays for one or more person. A person is only paid by a financial department. A financial department transfers salary to one or more bank accounts. A bank account is linked with one and only one financial department. A person lives in one and one location. A location accommodates one or more persons. A person owns one and one vehicle. A vehicle is owned by one and one person. A person can rent one or more vehicles. A vehicle is rented by one or more persons.

There might be some constraints predefined in the CDM. For example,

RULE:

IF a *bank account* is opened by a person, and this bank account is transferred by a financial department,

THEN the person should be paid by and only by that financial department.

**Step 2: Sample the degree for each entity**

The random knowledge graph generation algorithm is described below. Here, represents the existing vertices in the current knowledge graph. Each entity (e.g. *J. Smith*) in the knowledge graph will maintain a hash table named *neighbouring links*. The keys of the hash table are those relationship types of all the outgoing and incoming links at the start (or target) of this node. The values of this hash table are the number of links with a specific relationship type (e.g. owns). Below is an example of *neighbouring links* hash table with regard to a *person* entity in the knowledge graph based on the CDM shown in Figure 104.

**Table YY**. An example of neighbouring links hash table about ***person*** entity

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Lives\_in | Opens | Pays\_for | Rents | Owns |
| 1 | 3 | 1 | 6 | 7 |

As shown in the Table YY, there are 5 different types of links linking with a person entity (e.g. *J. Smith*). The number of links named *lives\_in* is 1. The number of links named *opens* is 3. The number of links named *pays\_for* is 1. The number of links named *rents* is 6. The number of links named *owns* is 7.

**Step 3: Generate Knowledge Graph**

First, let us give some definitions.

* Single edge between nodes

In our current implementation, we do not allow there exist two or more relationships from a *start* entity to a *target* entity in the knowledge graph.



**Fig. 105.** Multiple edges in the same direction are NOT ALLOWED in KG

For instance, there are two relationships named *rents* in the same direction between the two entities J.Smith and Honda Civic SA4342 in a knowledge graph which is shown in Fig. 105. Although this situation might happen with very low possibility in reality, it is not allowed to appear during the generation of our artificial knowledge graph.

* Weight Option of Relationship

Here, we import the concept of *weighting option* for a specified relationship type, e.g. *lives in*, during the process of KG generation. This is one of the input parameters when we generate a knowledge graph.

We consider this input parameter as long as the cardinality of the *source* of the relationship is 1 while the cardinality of the *target* of the relationship is , or the cardinality of the *source* is while the cardinality of the *target* is 1. We do not consider weighting option is the cardinality of a relationship does not follow the above assumption.

There are two kinds of values as the weighting option. The one is equal, and the other one is incremental.



Fig. 106 Weighting Option of a Relationship

If the weight option is equal, then all of the weight between the source entity and the target entity are equal. For example, all of the edges between University and Person are assigned the weight 1 when a University and a Person are paired randomly, which is shown in the left-hand side of Fig. 106.

Another situation for the weight option is incremental. In this case, the assigned weight on the relation between the source and the target entity are incremental. For instance, as shown in the right-hand side of Fig. 106, the weights are assigned as 1,2,3,4 incrementally between Person and Bank Account as the links are paired in the sequence of t1, t2, t3 and t4.

Here, we will give our main algorithm to create KG.

Algorithm XX. Main Algorithm to generate KG

|  |
| --- |
| Input: 1. Vertices Pool, 2. Expected # of entities *total* in KG, 3. Expected proportion *p* of different entities in KG, 4. List of Relationships  Output: knowledge graph |

There are 4 inputs for the main algorithm. The first input is *vertices pool* that provides sufficient entities of the knowledge graph. The second input is the number of entities in the expected knowledge graph, such as 1K, or 10K. The third input the proportion of different entities in the expected knowledge. It is *n* tuple like , which means the percentage for entities of Person, Vehicle and Location will be 40%, 30% and 30%. The fourth input is a sequence of relationship types, which will define the sequence of filling relationships into the knowledge graph. The output is a knowledge.

The procedure of the main algorithm is described in the following. First, we compute the number of entities of different types according to the expected total number and the predefined proportion of entity types. All of the required entities will be picked up from vertices pool. The out-degree/in-degree of these entities have already sampled according to the distribution function of each entity type. These nodes will be populated into a hash map , where the key is a certain entity type, e.g. Person, and the value is a set of entities of the type. All the nodes in the map are added into the KG.

So far, we have already created a number of entities with different types (e.g. Person, Location). An intuitive approach to pair vertices is to just pick up a pair of random vertices in the *uniform distribution* and then match. However, a lot of meaningful graph structures (i.e. patterns) will not be guaranteed in this way.

To this end, we turn to another way to pair entities (Line 2-6). Different relationships are filled into according to a specified *order*, which is predefined in the list . When a specified relationship is to be filled into , we pick up the specified entity sets and , corresponding to the type of the source and the target of respectively, and then the function is used to link the entities in with the entities in then is updated. For example, Person entities are paired with University entities since Person and University are the source and target of a specified relationship employs. The iteration will not terminate until all of the relationship types are added into the knowledge graph.

Algorithm XX. Pair the entities s and t in KG with a relationship r

|  |
| --- |
| Input: 1. Relationship type , 2., 3. , 4. Continuous failing times tolerance threshold .  Output: updated knowledge graph   1. Set all of entities in and *unvisited.* 3. t |

Before we give the algorithm that pairs two types of entities in KG, we first give a definition with regard to the distance between two arbitrary entities in a KG.

Let be a knowledge graph. There is a path between two nodes and , denoted , if and only if there exists a sequence of nodes with and such that for all holds .

Now, let us assume that we would like to add a specified relationship between a source entity and a target entity . There is a shortest path between and . Assume that there is a relationship type on each edge on the path . Let be the weight on the edge , and be the coefficient for a specified relationship type , the distance between and is:

A distance function is used to the probability link with with a specified relationship type . Give an entity , the probability that exists depends on:

Here, we give the algorithm to pair two types of entities in . There are three input for the *pairing* algorithm. The first input is a specified relationship. The second input is , the entity set of which the type is equal to the type of the *source* of . The third input is , the entity set of which the type if equal to the type of the *target* of . The fourth input is , a threshold representing the maximum times that the algorithm tolerates the continuous pairing failures for. The output of the algorithm is an updated knowledge graph .

The procedure for two types of entities and with a relationship is described as follows. Get all of the entities of which the type is the *source* of . A set with *m* entities is generated, i.e. . Get all of the entities of which the type is *target* of . A set with *n* entities is generated, i.e. . Set all of the entities in and *unvisited*, which means all of these entities contain available stubs that could be paired each other.

Pick up an entity from and an entity randomly from . If both and contain available stubs (i.e. unvisited), and and has never been paired yet, then pair with using an edge followed by updating # of stubs of and . Next, assign a specified weight w to . Repeat Step 3 until all of the entities in either S or T exhaust all the stubs. Alternatively, if pairing fails happen over continuously, Step 3 also breaks.

Clear all of the orphan entities.

Ideally, the pairing action stops when all the related *stubs*, either in the set of source or target, run out completely. However, we find that it is hard to guarantee that all of the stubs are exhausted in a reasonable timeframe so as to finish the whole process of pairing. Accordingly, a *lazy search* strategy is used that if we pick up pairs of entities and then fail to pair them, we will stop the process of pairing. We find that the performance of pairing is much more increased by using the *lazy search* strategy.

**Step 4:** Revise the KG

There is another requirement that we need to create *patterns* in the knowledge graph to satisfy some constraints defined in the CDM, making the knowledge graph more meaningful. In current stage, we only consider a graph structure (without properties) as a *graph pattern* in the knowledge graph.

**Example 1:**

RULE:

IF a *bank account* is opened by a person, and this bank account is transferred by a financial department,

THEN the person should be paid by and only by that financial department.



**Fig. 108.** Checking rule 1

In this checking rules, bank account is set as anchor point.

**Example 2:**

For instance, a graph pattern is shown in Fig. 109. This pattern actually exists in the knowledge graph frequently, although it is not obvious in the common data model in Fig. 104.

PATTERN:

IF two *persons* lives in a same position,

THEN there is 50% possibility that these two persons are friends.



**Fig. 109.** Generating patterns in the knowledge graph

So far, we only consider to generate graph pattern like *triangles* (e.g. the pattern in Fig. 109). In order to generate this graph pattern in the knowledge graph, before we link Person A (the upper left corner of Fig.109) with Person B (the upper right corner of Fig. 109), we will inspect if Person A and Person B share a common entity named Location as *successor*. If it is, we will link Person A and Person B using with the probability 50%.

## Selection of pattern matching tool

We have investigated quite a few open-source *approximate subgraph matching* methods and tools in order to build our prototype. Below we list some tools that we investigated.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | Complexity | Validated on | Graph similarity measure | Implementation Language | Source code existed |
| Dover [9] | NA | Stanford Large Network[[4]](#footnote-4) | Graph sampling & Exact matching | java | Available |
| SAPPER [10] | O(mnd3) | Protein Network | GED | C++ | NA |
| APROMORE [11] | O(n3) | Process model | GED+ | java | Available |
| ASM [12] | O(mn) | Bio-informatics | GED | java | Available |
| GraMi [13] |  | Network graph |  | java | Available |

* Dover

The approximate subgraph matching algorithm takes a sample of subgraphs from the target set and compares them with a given pattern graph. The number of enumerated subgraphs is specified by the user, along with the desired target graph. For each of these enumerated subgraphs, the *exact subgraph isomorphism* algorithm is run, with a label comparator. Once all subgraphs have been compared, the result are output. The approximate subgraph isomorphism algorithm can run quicker than the exact version where a node will have potential matches with a larger number of target nodes [9]. The source code of Dover is available on the Github[[5]](#footnote-5).

There are numerous methods for *sampling* subgraphs from a larger graph and many have bias towards particular subgraphs. The method used is a neighborhood sampling method which randomly builds a subgraph in a combined depth and breadth approach.

The user must first define how many subgraphs per node are to be generated and the number of attempts the system may take to build a subgraph.

The method takes each node in turn. For the node, a random neighbor is chosen and added to the subgraph along with the starting node. From then, one of the nodes in the subgraph is chosen at random and one of the neighbors of this node is chosen at random. If there are no neighbors, or the chosen neighbor is already in the subgraph, then another node and neighbor is picked at random instead. If the system continues to pick invalid nodes, up to the limit defined by the user, then the subgraph is abandoned.

Once the required number of nodes have been added to the subgraph, then any edges connecting these nodes are induced. This subgraph is now complete and stored for further use.

The *exact subgraph isomorphism* is the process of finding instances of a *pattern graph* in *target graph*. In Dover, it is an NP-Complete problem, and so worst case solutions have exponential time complexity.

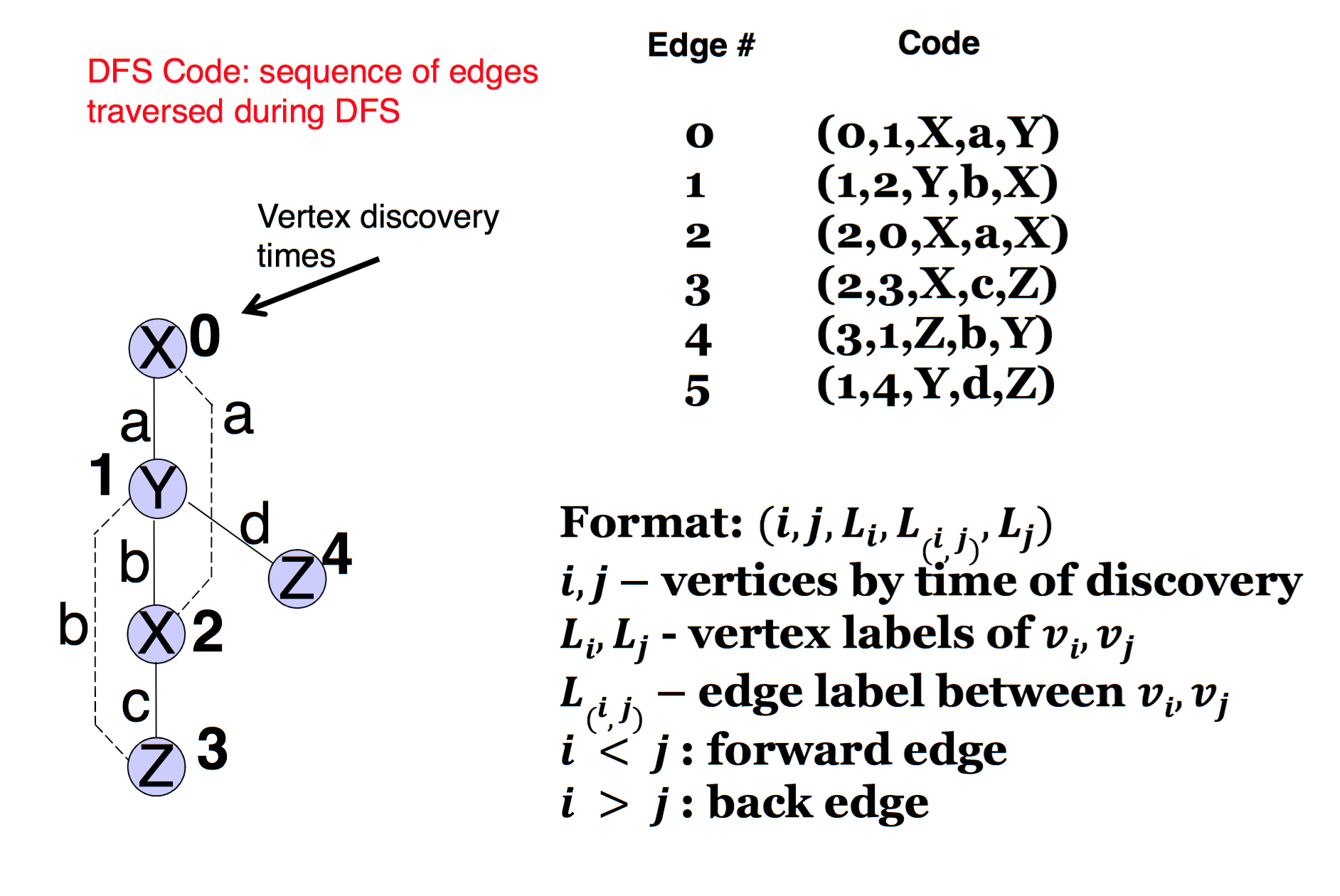
The method Dover uses applies a brute force backtracking search to find all possible matches of the pattern in the target. Initially, potential target node matches in the pattern graph are identified. A node in the pattern graph is a potential match with a node in the target graph if the pattern node has the same or less degree than the target node. A backtracking search is then performed, taking a node from the pattern and attempting to find a match from its matching set of target nodes. For each node in the target, , the neighbors of and must correspond. That is, if a node is a neighbor of and it has an assigned match in the other graph, the matching node must be a neighbour of . It is similar for the neighbours of . If a match is found, the next node in the first graph is checked for possible matches. If no matches can be found for , backtracking occurs, and the last previously matched node in the first graph is checked for other matches. If all nodes in the target are matched, the matching nodes are stored and the process continues. The process ends when all nodes have been tested.

* ASM

ASM (Approximate Subgraph Matching) is a subgraph matching tool for dependency graphs, which is directed graphs satisfying our requirements. It is the Java implementation of the approximate subgraph matching algorithm designed for matching dependency graphs. The source code of ASM is available on the Sourceforge[[6]](#footnote-6). ASM is based on a previous exact subgraph matching (ESM) algorithm[[7]](#footnote-7), which is designed for dependency graphs using a backtracking approach. ASM is capable of detecting approximate subgraph matching based on a subgraph distance. Assume that the *target* graph and the *pattern* graph have and vertices, and and edges respectively, the total worst-case algorithm complexity is .

* GraMi

During the process of extension, every graph has a unique graph index, i.e. DFS code, which is a sequence of edges traversed during Depth First Search.



Solving the CSP can still take exponential time in the worst case. In order to support large graphs in real-life applications, GraMi employs a heuristic search and a seies of optimizations the significant improve performance.

* + Prune large portions of the search space,
  + Prioritize fast and postpone slow searches,
  + Take advantage of special graph types and structures.

By avoiding the exhaustive of special graph types and structures and using the proposed optimizations, GraMi supports larger graphs and smaller frequency thresholds than existing approaches. For example, to compute the frequent patterns of the 100K nodes/1M edges graph that the state-of-art grow-and-store method crashed after a day, GraMi needs only 16 minutes.

The key of gSpan (and maybe GraMi):

* + Since the design combines the subgraph isomorphism test and frequent subgraph growth into one procedure, gSpan dramatically accelerates the mining process.
  + A hierarchical search space should be built to facilitate the search.
  + Enumerate subgraphs in increasing lexicographic order, which is consistent with the depth-first traversal of the search space.

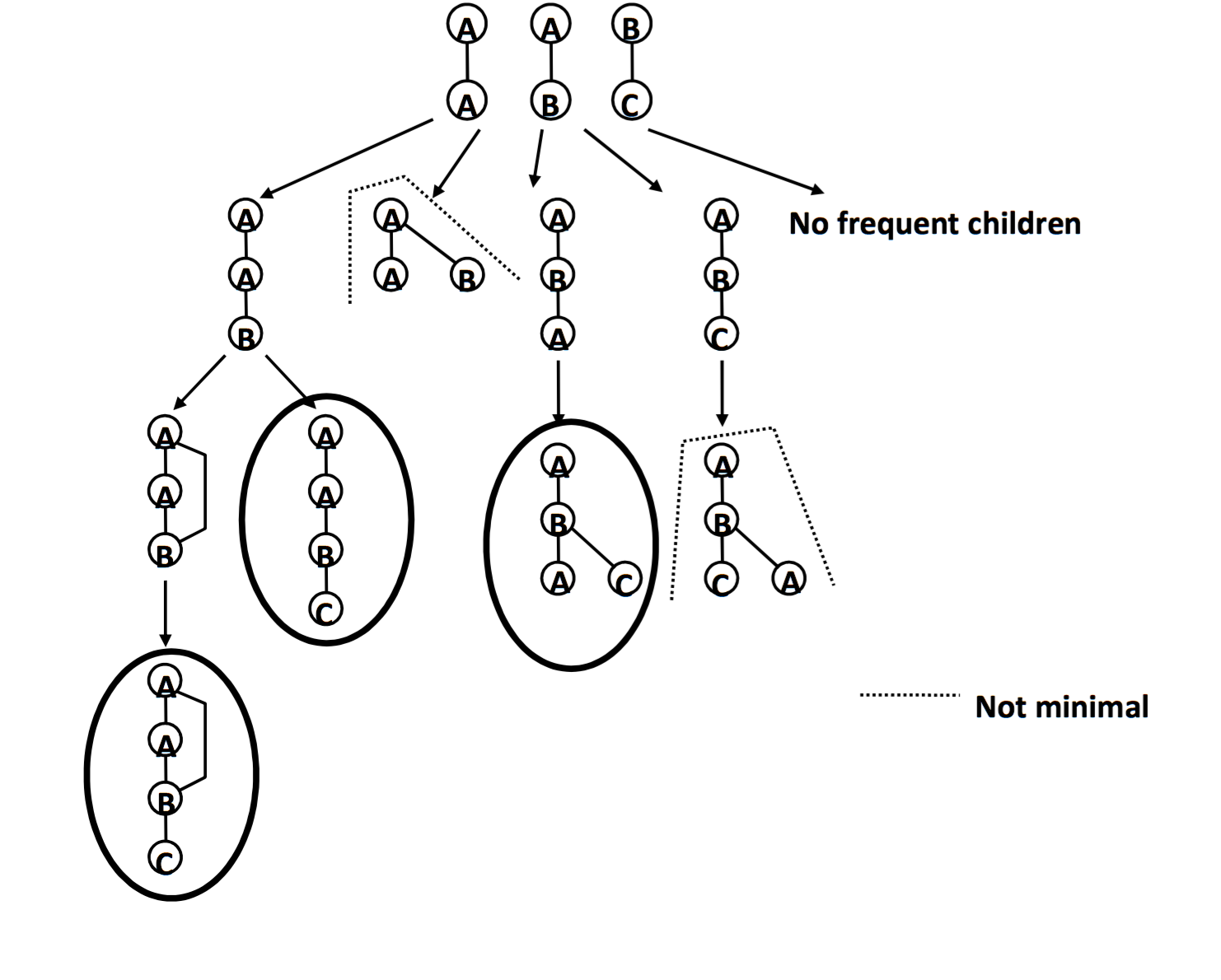


Fig. xxx. A Search Space: DFS Code Tree

We experimentally re-evaluated GraMi based on CiteSeer[[8]](#footnote-8) which is a directed graph with 3312 vertices and 4590 edges. The experiment is conducted using Java JRE v1.8.0 on a OS X 10.11.6 machine with 7 cores running at 2.5GHz with 16GB RAM and 251GB disk. CiteSeer represents a directed graph where each node has a single label representing a Computer Science area. Each edge has a label (0 to 100) that measures the similarity between the corresponding pair of publications, a smaller label denotes a stronger similarity.

Fig. xxx. Performance of GraMi

Since we goal is just to use the function of subgraph matching in GraMi, we test the capability of approximate subgraph matching in a big graph given a high threshold of pattern occurrence. In order to evaluate it, we adopt the following strategy.

* + Step1: We create a pattern graph with around 30 vertices.
  + Step2: Copy the pattern graph for 1000 times, i.e., .
  + Step3: Link these 1000 graphs together into a big graph after adding more edges.
  + Step4: Call subgraph matching function of GraMi to count the occurrence of .

1. http://karma.isi.edu [↑](#footnote-ref-1)
2. http://americanart.si.edu [↑](#footnote-ref-2)
3. For the common subgraph matching problem, it is just one edge. [↑](#footnote-ref-3)
4. https://snap.stanford.edu/data/ [↑](#footnote-ref-4)
5. https://github.com/peterrodgers/dover [↑](#footnote-ref-5)
6. https://sourceforge.net/projects/asmalgorithm/?source=navbar [↑](#footnote-ref-6)
7. http://esmalgorithm.sourceforge.net [↑](#footnote-ref-7)
8. https://linqs.soe.ucsc.edu/data [↑](#footnote-ref-8)