ICS

TECHNICAL REPORT

Exploiting Relationships for Data Cleaning

Dmitri V. Kalashnikov, Sharad Mehrotra, Zhaoqi Chen

UCI-ICS Technical Report No. 04-11
Department of Computer Science
University of California, Irvine

May 2004

Information and Computer Science
University of California, Irvine

Exploiting Relationships for Data Cleaning

DMITRI V. KALASHNIKOV, SHARAD MEHROTRA, and STELLA CHEN

University of California, Irvine
Information and Computer Science
Tech Report 04-11
May 19, 2004

In this paper we address the problem of data cleaning when multiple data sources are merged to create a single database. Specifically, we focus on the problem of determining if two representations in two different sources refer to the same entity. Current research has focused on linking records from different sources by computing the similarity among them based on their attribute values. Our approach explores a new research direction by exploiting *relationships* among records for the purpose of cleaning. Our approach is based on the hypothesis that if two representations refer to the same entity, there is a high likelihood that they are strongly connected to each other through multiple relationships implicit in the database. We view the database as a graph in which nodes correspond to entities and edges to relationships among the entities. Any one of the existing conventional approaches is first used to determine possible matches among entities. Graph analysis techniques are then used to disambiguate among the various choices. While out approach is domain independent, it can be tuned to specific domains by incorporating domain specific rules. We demonstrate the applicability of our method to a large real dataset.

Categories and Subject Descriptors: H.m [**MISCELLANEOUS**]: Data Cleaning

General Terms:
Additional Key Words and Phrases: relationship-based data cleaning, record linkage, similarity retrieval

Milestones

(1) Jul-Aug, 2003, The RelDC framework is developed. The weight-based approach is implemented.

(2) Sep-Oct, 2003, The optimizations are developed to scale the RelDC to large datasets (Citeseer and HPSearch).

(3) Nov-Mar, 2004, The solver-based solution is developed.

(4) Apr-May, 2004, The probabilistic approach is implemented.

This material is based upon work supported by the National Science Foundation under Grant No. IIS-0331707 and IIS-0083489 and by the Knowledge Discovery and Dissemination (KD-D) Program.

1. INTRODUCTION

Recent surveys [Discovery 2003] show that more than 80% of researchers working on datamining projects spend more than 40% of their project time on cleaning and preparation of data. The data cleaning problem often arises in data warehouse environments when information from heterogeneous sources is collected and merged. Consider an input table I containing records that are to be merged with the reference database R. It is assumed that the database R is clean[1] in which every entity is represented in its canonical form. Representation of the entity in I may differ from that of in R. For example, in a database containing business information of companies, a corporation named ACME may be represented as "ACME Corporation". The same corporation may be referred to as "ACME Corp." in the input table.

Such differences may arise due to differences in data representation, inconsistencies, and/or incompleteness (missing attribute values) of data sources. Often, it is desirable that such linkages among data be identified while merging in order to ensure high accuracy/quality of the resulting merged database. Quality of the data has direct implications to the quality of the data analysis/ data mining which, in turn, may have an significant impact the quality of business decisions and hence revenues. As a result, data cleaning tools that empower analysts overseeing the merging process to perform successful and efficient matching among records are important. Typically, such tools work by computing the similarity between two records based on their attribute values. For example, the two strings representing ACME corporation in the example above, though not identical, are sufficiently similar to suggest that they potentially refer to the same entity.

While an approach to matching entities/tuples based solely on matching corresponding attribute values may work well in many instances, difficulties arise (as is often the case) if the attribute values for an entity/record of an input table matches description of more than one object in the reference tables. What basis should then be used to disambiguate among the multiple matches? One possible approach is for the cleaning tool to output all potential matches and for the analyst to decide among the various choices. Even so, the challenge still remains – what basis should the analyst use for disambiguation?

org_id	org_name	country	area
boeing_usa	Boeing	USA	aerospace
boeing_aus	Boeing	AUS	farming
beoing_ger	B*eo*ing	GER	finance
a	A	–	farming
g	G	GER	–
u_1	U_1	–	aerospace
u_2	U_2	USA	–
u_3	U_3	USA	–

Table I. Reference table: company information

[1]It is commonly assumed that reference tables do not contain duplicates and R is complete, i.e. whenever a tuple in the input table is referring to an object, this object is present in R.

trans_id	org_id1	org_id2
T_1	boeing_usa	u_1
T_2	boeing_usa	u_1
T_3	u_1	u_2
T_4	u_2	u_3
T_5	boeing_aus	a
T_6	boeing_aus	a
T_7	beoing_ger	g
T_8	beoing_ger	g
T_9	beoing_ger	g

Table II. Organization transactions

input_id	org_name1	org_name2
I_1	Boeing	A
I_2	Beoing	G
I_3	Beoing	U_1
I_4	Beoing	U_2
I_5	Beoing	U_3

Table III. Input table: org_name1 to be cleaned, input_id and org_name2 are known to be clean

In this paper, we hypothesize (and we will justify via examples shortly) that the knowledge about entities and their relationships latent in the database can be used to achieve such a disambiguation. Based on this hypothesis, we propose an approach to data cleaning that can be applied for choosing among various matching alternatives when merging relational data.

Before we describe the proposed approach in detail, let us motivate it through an example. Consider again the data warehouse scenario where data from multiple sources about companies and their business transactions is being merged to create a common database. Let the warehouse schema contain two tables: the *company* and *transaction* tables explained below. Assume the *input* table contains new data that needs to be merged.

Table I shows a sample content of the company table for eight companies. It serves as a reference table. That is, the company name in the column org_name is considered as a canonical representation and this representation will be used to determine the matches during data cleaning. The other attributes are organization id, its specialization area, the country in which the organization is located. For some of the companies the country or area may be unknown.

Table II shows information about companies' business transactions. Each tuple contains transaction id and two id's of companies between which the transaction has occurred. The intent of that table is to show that Boeing of USA has many transactions with company named U_1, company U_1 normally does business with company U_2, company U_2 with company U_3, also all current transactions of Boeing of AUS are with company A, and company B*eo*ing of GER has many transactions with company G.

Table III contains data to be cleaned in the form of new transactions specified as *names* (not id's) of companies between which those transactions have occurred. We assume in this example that only values of the org_name1 attribute are to be

cleaned and the values of the other attributes are known to be clean.

To clean the data in Table III, current algorithms such as [Chaudhuri et al. 2003] essentially would try to do string-based similarity matches between names of companies in the input table and company table, which serves as a reference table. Correspondingly such methods would unavoidably conclude that "Beoing" companies mentioned in Table III most likely correspond to the B*eo*ing of GER because this company has the best string match with "Beoing" among all possible candidates.

However, for this specific domain it could be the case that two companies from the same country, having the same specialization areas, that have previously worked together (have multiple business transactions with each other) are more likely to have transactions together rather than two companies from different countries and having different specializations and which have not had transactions with each other previously. Let us assume that the rules mentioned above do apply and see what effect this has on data cleaning.

Further refinement [I_1]: Consider cleaning of tuple I_1. Any string-based approach would output that Boeing in I_1 is, with equal probability, either Boeing of USA or Boeing of AUS. Another (lesser) possibility is of Boeing being misspelled and it is in reality Beoing of GER. However, given our assumptions and the fact that company A has previously had all its transactions with Boeing of AUS, and Boeing of AUS and A have the same specialization, one can further assert that the highest probability to be the right match should be given to Boeing of AUS, second highest to Boeing of USA, and the lowest to Beoing of GER.

Consistent [I_2]: For tuple I_2 any string-based algorithm would identify Beoing of GER as being the most likely candidate. This is also consistent with the current content of the database.

Contradiction [I_3, I_4, I_5]: For I_3, I_4, and I_5 any string-based method would output that Beoing of GER is the most likely correct match. These cases are interesting because if one relies solely on our assumptions and ignores what is returned by any string-based method then one will conclude that Boeing of USA, not Beoing of GER, is the likely correct match. This is because company U_1 has previously had transactions only with Boeing of USA and further, they both have the same specialization. Company U_2 shares the same country with Boeing of USA, and it has worked with U_1 which has worked with Boeing of USA – these are weak connections, but nevertheless the other possible candidates do not have such connections at all. The situation with company U_3 is similar, except for the second connection chain to Boeing of USA is now longer and weaker: U_3 has transactions with U_2, which has transactions with U_1, which has transactions with Boeing of USA.

Above we have shown cases where knowledge of the application domain and relationships among entities can impact our decisions about matches between entities beyond string-based match approaches.

While at first glance such an analysis might seem like an ad hoc (domain dependent) solution, a generic solution emerges if we view the data in the form of a graph of entities that are interconnected via relationships between them. In the graph, we will use the term *node* for vertex and *link* for edge. The term *path*, unless specified otherwise, refers to a simple path with no duplicate nodes. Figure 1

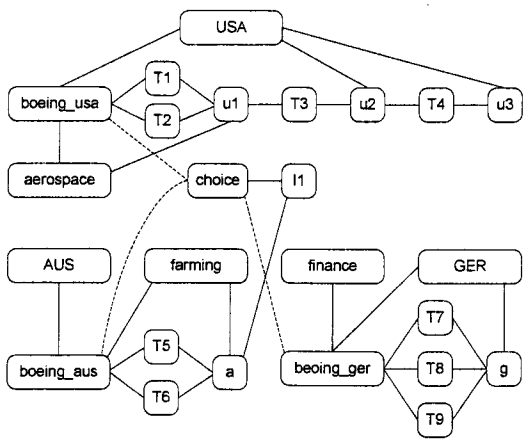

Fig. 1. Graph for the warehouse example

shows a possible graph representation of the reference warehouse database. We also show a single entity in the input table I_1 which needs to be resolved. I_1 is shown connected to organization A and to the three possible "boeing" choices via a choice node. How to construct such a graph will be made clear in the following sections. The analysis leading to cleaning in the example above can then be viewed as determining the connectivity between the entity being cleaned and the various choices that exist in the database. The measure of connectivity between nodes in a graph can be estimated by analyzing various paths that exist between the two nodes. The above example summarizes the essence of our approach. What we require is a mechanism for estimating the strength of connections among entities in a graph based on relationships among them and an algorithm to do it efficiently.

Empowered with these observations, we develop a generic approach for off-line domain independent data cleaning which we refer to as Relationship-based Data Cleaning (RelDC). RelDC views the underlying database as a graph and exploits relationships among entities to make accurate match predictions. We explore several optimizations of the algorithm to scale RelDC to very large data sets, we show a few of those in Section 5, more optimizations are available in [Kalashnikov and Mehrotra].

While the approach presented in the paper is generic, to validate our approach, we had to find one large *public-domain* real dataset to demonstrate its applicability. For this purpose we have chosen the author matching problem explained in Section 2. The rest of this paper is organized as follows. Section 3 presents RelDC approach. Experimental results are presented in Section 6. Finally, related work is covered in Section 7 and Section 8 concludes the paper.

2. AUTHOR MATCHING PROBLEM

In the author matching problem we consider merging of two sources: CiteSeer[CiteSeer] and HPSearch[HomePageSearch]. From CiteSeer we extract an (input) *paper* table with the schema (`paper_id`, `FI`, `last_name`, `name1`, `name2`, `name3`). These attributes correspond to the CiteSeer's paper id, author first initial, last name, first name, second name and third name. The last three attributes are not always avail-

able. From HPSearch we extract a (reference) *author* table with the schema (`name1`, `name2`, `last_name`, `department`, `university/organization`). The attributes are author name and affiliation, the latter is not always specified in HPSearch.

In the author matching problem main entities are authors and papers. An author name in the *paper table* from CiteSeer can match several authors in HPSearch. The goal is to identify the true author. For example, J. Smith who authored paper P1 may correspond to one of the many entries in HPSearch *author* table, e.g., John Smith, Jane Smith, etc. The cleaning task is to link the record in the *paper* table to the record of the real author of the paper in the *author* table.

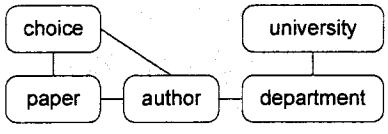

Fig. 2. Connectivity graphs for author matching

Our approach to data cleaning exploits relationships to determine linkage among entities. The types of relationships among entities in the author matching problem is depicted in Figure 2. In the figure nodes correspond to entity types and edges represent the relationships between the entities. The figure shows, for example, that a paper entity is related to the author entity. Similarly author is related to department, but there is no direct relationship between paper and department. While Figure 2 shows the type of entities and relationships among them, using the author and paper tables, a large instance level graph can be be constructed similar in nature to Figure 1. In such a graph, a node will be created for each distinct author, university, department and paper and the entities will be related to each other based on relationships identified in Figure 2. A set of choice nodes will be added to the graph the purpose of which will become evident shortly.

Note that in the author matching problem we have extracted only a few simple attributes that introduce relationships useful for our cleaning task and suffice to illustrate the concepts and validate the approach. In reality CiteSeer and HPSearch pages are much richer in content and one could have extracted many other attributes, e.g. for some of the papers in CiteSeer it is possible to extract author affiliations, paper conferences, etc. This will further improve accuracy of our method.

3. RELDC APPROACH

RelDC analyzes various relationships between entities and their strength for data cleaning. It is based on the following assumption:

Relationships between entities are important for data cleaning.

Correspondingly, if the assumption is not true for a particular domain, then the solution presented in the paper is not applicable. But as will be shown, the assumption is true for many of the domains in practice. The basics steps of the RelDC approach are as follow.

- *Step 1.* Use an existing similarity approach to cleanse data. Wherever the existing approaches, such as [Chaudhuri et al. 2003], fail in disambiguating records (i.e., it finds multiple matching candidates) identify those records.
- *Step 2.* Determine the strength of connection between the to-be-cleaned records and their matching candidates based on relationships.

- *Step 3.* Resolve the choices based on the connection strengths.

The above three steps are an abstract representation of our proposed approach. Step 1 is straightforward, we simply take existing approaches to determine the possible choices. The rest of this section is dedicated to discussing steps 2 and 3 which together represent the novel aspect of our approach. Before we proceed, we note that steps 2 and 3 can be realized in a variety of different ways. What we discuss below is one particular implementation. We do not claim the specific implementation/realization of steps 2 and 3 is the best (either in terms of effectiveness of the cleaning or the performance). It is an intuitively appealing approach that is able to achieve high accuracy as will be shown in Section 6. Moreover it suffices our purpose of establishing the importance of relationships in data cleaning which we believe is our fundamental contribution in this paper.

In explaining our approach we view the database as a graph. We could have explained our approach directly in terms of the relational view of the data, but we find the graphical view more intuitive and natural for our purpose. Chains of relationships between entities correspond to paths (or connections) in the graph. Thus in terms of the graph terminology the assumption can be restated as follows.

Connections between entities in the graph are important for data cleaning.

We note that graphical view of structured data is commonly used in many data analysis and designing situations. For example, ER diagrams are commonly used as a starting step for relational data design. The ER model is used since it is more conceptual compared to the relational model and hence easier/more intuitive for the designer to deal with. Once the database designer has established the initial design, the resulting ER design is mapped (often automatically) to the corresponding relational design via an ER to relational mapping.

In our situation, what we need is the reverse mapping – to view data as a graph, we need to map the relational data into the corresponding graph of entities interconnected via relationships. Such a reverse mapping has also been studied in the database design literature in the past [Mannila and Raiha 1992]. Such a mapping, while in certain situations can be automated by exploiting knowledge of database constraints (e.g., foreign key constraints), in general, it requires human intervention. We do not dwell on this issue any further except to observe that the analyst must guide such a mapping.

Edges as well as vertices in the graph can be labeled with weights. Edge labels are needed to interpret the degree of confidence the relationship exists between the two entities. For example, in Figure 1 specialization are of company Beoing of GER might not be known precisely, but the system might be able to associate confidence of 0.6 that its specialization is finance. In that case the corresponding edge will be labeled 0.6. The construction of the graph as it is used for our data cleaning algorithm is detailed in Section 3.1.

An implementation of RelDC The basic steps of our implementation of RelDC are outlined in Figure 3 and explained in detail is the subsequent sections.

Data Cleaning Process Figure 4 shows the data cleaning process for relational DBMSs in the context of our algorithm. First a graph of interest is created from a relational DBMS. Then RelDC is applied to that graph producing a cleaned version of it. Finally the results are interpreted and stored back in DBMS. In certain

Input:
(1) a reference database R
(2) a 'dirty' table I to be merged with R
(3) to-be-cleaned attributes in I and the corresponding attributes in R
Algorithm:
1. create representation of R as a graph G
2. merge input table I with G
   (a) use fms to determine initial matches
   (b) create choice nodes
3. resolve choices
   (a) discover relationships
   (b) create equations for weights
   (c) resolve weights
4. interpret weights

Fig. 3. RelDC Approach

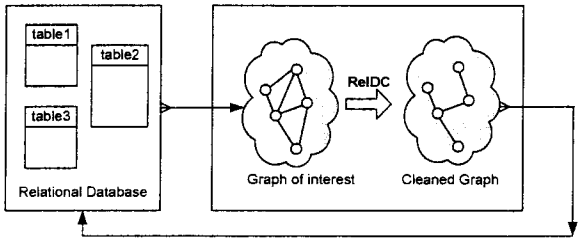

Fig. 4. RelDC: Data cleaning cycle

situations this process can be made to continue in cycle in order for the analyst to apply the domain specific rules as discussed in [Kalashnikov and Mehrotra].

3.1 Graph construction

This section explains step 1 in Figure 3. The graph used in the system reflects various relationships, represented as edges, between entities represented as nodes. Each edge has a weight associated with it. This weight reflects the level of confidence in the fact that the two nodes connected by the edge are actually related. Nodes, in general, have weights too, which reflect the relevance of the nodes to a particular matching task.[2]

When processing a tuple from the input table a node is created for that tuple such that attributes of that tuple becomes attributes of the node. When processing a node with to-be-cleaned attribute(s), the goal is to link this node correctly to the existing graph based on the values of these attributes for that node. As an example consider an author matching problem where a paper $paper_1$ is processed and needs to be correctly linked to the graph, see Figure 5. Assume in addition to what is mentioned in Figure 2, the algorithm can handle author and paper specialization area. Assume $paper_1$ is coauthored by "J. Smith" and "C. Unique", and the paper's specialization area is "DB". Our algorithm first utilizes one of the

[2]For the author matching problem weight of all nodes is 1.

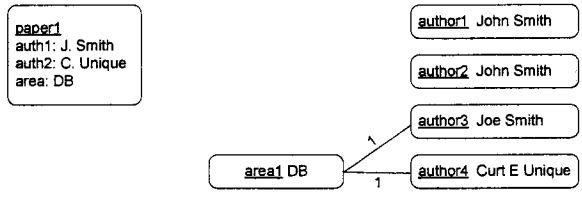

Fig. 5. Processing a new paper

on-line string-based methods, such as [Chaudhuri et al. 2003], to construct initial matches. Assume according to a string-based similarity function author "J. Smith" can match only three people shown in Figure 5, and author "C. Unique" can match only one person. Furthermore, assume it is known that authors "Joe Smith" and "Curt E. Unique" work in DB area, whereas the other two authors do not.

3.2 Choice creation

In this section we show how the input table is merged with the graph, which corresponds to step 2 in Figure 3. Since name "C. Unique" can correspond to only author "Curt E. Unique", $paper_1$ is linked directly to $author_4$ with weight of 1 as shown in Figure 6. "J. Smith" cannot be resolved with certainty. The goal is to determine who he really is based on the knowledge already stored in the database. Our solution to the problem is to create an instance of a special type of node: a

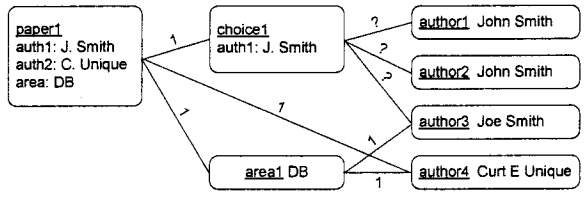

Fig. 6. Choice creation

choice node, see Figure 6. Each choice is connected with weight 1 to the entity with the to-be-cleaned attribute for which it has been created. In Figure 6 $choice_1$ is connected to $paper_1$. It is also connected to all the entities, the value of the to-be-cleaned attribute can refer to. These entities are initial matches determined by a string-based similarity function. In Figure 6 $choice_1$ is connected to authors 1, 2, and 3. Such entities are called *hypotheses* or *options* of the choice. The goal is to assign the appropriate weights to the edges linking the choice and its options based on the current content of the database. A higher weight of an edge should correspond to a higher chance for the option of being the correct match.

For notational convenience we will use "weight of an option" for "weight of the edge linking the choice and option nodes". We will use "a choice" for "a choice node". When we talk about "resolving a choice" we mean finding the weights of its options.

Notation	Meaning
$choice_i$	choice number i
$option_{ij}$	option number j of $choice_i$
weight of $option_{ij}$	weight of edge $choice_i \leftrightarrow option_{ij}$
w_{ij}	raw weight of $option_{ij}$
$\overline{w}_{ij}$	normalized weight of $option_{ij}$

Table IV. Terminology

3.3 Defining option weights

Now let us see how steps 3(a) and 3(b) in Figure 3 are implemented in our approach. Determining the weight of j^{th} option of $choice_i$, $option_{ij}$, in our context means one needs to find and measure some evidence of a hypothesis that the value for which $choice_i$ has been created indeed corresponds to $option_{ij}$, see Figure 7. The string-

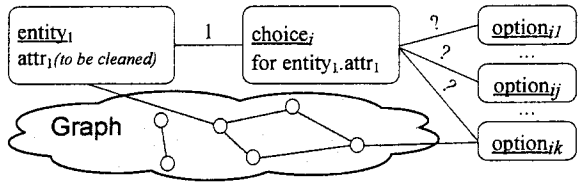

Fig. 7. Choice resolution

based method returns one such measure w_{ij}^{fms} which is our notation for weight of $option_{ij}$ returned by a string-based similarity function. Weight w_{ij}^{fms} can be based for example on edit distance or any other reasonable measure. Acronym *fms*, introduced in [Chaudhuri et al. 2003], stands for fuzzy match similarity and we use it as a more generic term for any string-based similarity (or number-based similarity).

• **Discovering relationships** Unlike any fms, RelDC analyzes relationships between entities. In a graph, relationships correspond to paths. To analyze $option_{ij}$ of $choice_i$ our algorithm first *discovers* all relevant relationships. It does so by invoking *AllSimplePaths* algorithm which finds all simple paths in the graph linking $option_{ij}$ and entity $entity_1$ for which $choice_i$ has been created.

Not all paths returned by AllSimplePaths algorithm are considered, some of them must be ignored. Clearly the path via the $choice_i$ node (i.e., $option_{ij} \rightarrow choice_i \rightarrow entity_1$) must be ignored since our purpose in analyzing $option_{ij}$ of $choice_i$ is to determine the strength of the connection between $entity_1$ and $option_{ij}$ based on *other* relationships. Also if a path goes through some other choice $choice_k$, it cannot contain two edges $choice_k \rightarrow option_{km}$ and $choice_k \rightarrow option_{kl}$, $l \neq m$, where $option_{kl}$ and $option_{km}$ are any two options of $choice_k$. Such a path is illegal since our underlying assumption is that the attribute of the entity a choice is trying to resolve in reality matches exactly one option. Thus these edges are mutually exclusive in a single path.

As an example, consider Figure 6. There is no legal simple path between $paper_1$ and $author_1$ and between $paper_1$ and $author_2$: simple paths $author_1 \rightarrow paper_1$ and

$author_2 \rightarrow paper_1$ are discarded as paths going via $choice_1$ (as described above). There are two simple paths between $paper_1$ and $author_3$: $author_3 \rightarrow area_1 \rightarrow paper_1$ and $author_3 \rightarrow area_1 \rightarrow author_4 \rightarrow paper_1$.

• **Analyzing and weighing relationships** For each path returned by AllSimplePaths our algorithm computes *connection strength* of that path as explained in Section 3.3.1. Note that we show the reason behind our formula for connection

```
1. for i ← 1 to num_choices do
2.   for j ← 1 to choice_i.num_options do
3.     Paths_ij ← AllSimplePaths(option_ij, choice_i.entity)
4.     w_ij = ConStrength(Paths_ij)

5. for i ← 1 to num_choices do
6.   m ← choice_i.num_options
7.   for j ← 1 to m do
8.     option_ij.w = w̄_ij = Normalize(j, w_i1, ..., w_im)
```

Fig. 8. RelDC: defining w_{ij} and $\overline{w}_{ij}$

strength but one potentially can compute it differently. The algorithm combines individual connection strength of each path to compute the raw weight w_{ij} of $option_{ij}$. Then, for each $option_{ij}$ its normalized (to 1) weight, $\overline{w}_{ij}$, is computed as explained in Section 3.3.2, $\overline{w}_{ij}$ is used for successive processing as weight of $option_{ij}$, i.e. it replaces the corresponding "?" in Figure 7. The procedure for defining the raw and normalized weights is outlined in Figure 8.

• **Interpreting weights** This addresses step 4 in Figure 3. Once the final normalized weight are computed they can be interpreted in the similar fashion as weights of any string-based approach. One of the approaches used in the literature is for analyst to specify a threshhold $\mathcal{T}$, e.g. 0.80. If the weight of one of the matching candidates is greater than the threshold, then this candidate is chosen as the answer of the algorithm. If multiple candidates qualify, the one with the highest weight is choosen, ties are broken randomly. If no such candidate cannot be chosen, then such cases are flagged for the analyst to handle manually. If the analyst specifyies the threshhod of zero, then the algorithm will work in fully automatic mode and will not require interaction with the analyst. In our experimental result section $\mathcal{T}$ is always zero.

3.3.1 *Computing connection strength.* Once the set of all simple paths is determined, the raw weight is computed as a function of individual connections. But which factors should be taken into account when computing the connection strength of each individual path?

Figure 9 illustrates two different connections between authors $author_1$ and $author_2$. Let us understand which connection is better, and why. Both connections have an equal length of two. One connection is going via node *Researchers*, the other one via $paper_1$. It is no wonder it is possible to connect the authors via *Researchers* and such a connection should definitely have a lesser strength than a connection via $paper_1$. Notice, this is reflected in the fact that *Researchers* is connected to every

author, but $paper_1$ connects only the two authors. Based on this intuition let us define a connection strength of a general simple path between two nodes.

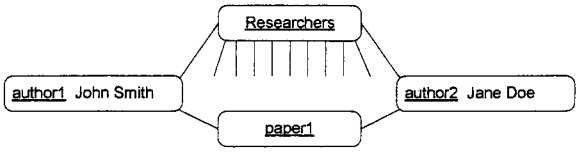

Fig. 9. Strong and weak connections

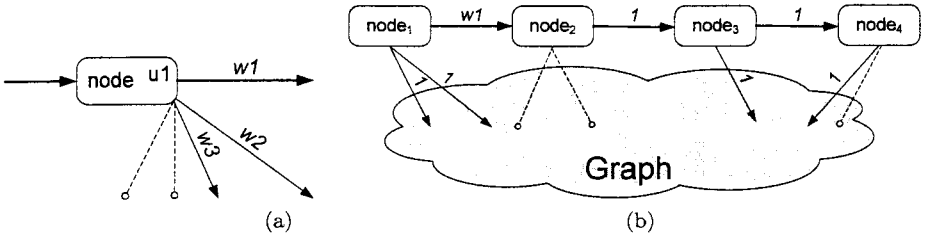

Fig. 10. (a) Link coefficient (b) Connection strength of path $node_1 \rightarrow node_4$ is $\frac{w_1}{2}$

Assume path p consist of k links: $l_1, \ldots, l_k$. Let us define link coefficient C_i for each link l_i in the given path. The formula for C_1 is different from the formula for C_i when i is greater than 1. For l_1, C_1 is chosen to be the weight of the link w_{l_1}. Figure 10(a) shows a case where a path goes from left to right through an intermediate node. In our system certain paths are not valid, as explained above, and also paths must be simple so if a node is visited already the path cannot go back to that node. To reflect this fact Figure 10(a) shows that because the node has been entered via a certain link, the path can continue only via three links depicted in solid lines, and cannot continue via links depicted in dashed lines. If the path goes through the link with weight w_1 the link coefficient is calculated as $\frac{w_1}{w_1+w_2+w_3}$. This corresponds to the probability of following the link if doing a random walk in the graph, following only simple paths and following links with probability proportional to their weights. The connection strength w_p of the whole path p is computed as a product of link coefficients: $w_p = \prod_{l \in p} C_l$, where C_l is the link coefficient of link l in path p. The intuition behind the formula is that it is closely related to the probability of reaching the destination from the source via this path if doing a random walk in graph. To compute the connection strength w_{n_1,n_2} between two nodes n_1 and n_2 one would need to find the set of all simple paths between the two nodes P_{n_1,n_2} and sum up the weight of all individual paths:

$$w_{n_1,n_2} \stackrel{\text{def}}{=} \sum_{p \in P_{n_1,n_2}} w_p = \sum_{p \in P_{n_1,n_2}} \left(\prod_{l \in p} C_l \right) \quad (1)$$

It can be seen that the total connection strength between n_1 and n_2 defined in Equation 1 is related to the probability of reaching n_2 if doing a random walk in the graph starting from n_1.

Section 4 in the appendix discuss how to adjust the link coefficient formula to capture the fact that longer relationships are weaker and accommodate the node relevance.

Example Connection strength of path $node_1 \rightarrow node_4$ shown in Figure 10(b) is $\frac{w_1}{2}$. This is so because the link coefficient C_1 of the first link by definition is simply its weight. The link coefficient of the second link C_2 is 1 because even though $node_2$ has 4 edges, due to the fact that it was entered from a specific link, it can be left only via one specific link. The link coefficient of the last link in the path, C_3, is $\frac{1}{2}$ because the node can be left via two links of equal weight. The connection strength of the path is computed as $C_1 \cdot C_2 \cdot C_3$ which gives us $\frac{w_1}{2}$.

3.3.2 *Option weight normalization.* Once the raw weights of the options of a choice are computed they are further normalized to 1 to reflect the fraction of confidence the algorithm places for each option to be the right match.

o *Normalization method 1* One way of doing such a normalization is to use simple formula for normalized weight $\overline{w}_{ij}$:

$$\overline{w}_{ij} = \begin{cases} 0 & \text{if } w_{ij} = 0; \\ \frac{w_{ij}}{\sum_j w_{ij}} & \text{if } w_{ij} > 0. \end{cases}$$

It is interesting to note that such normalized weights is related to to the probability of reaching the options following links while adhering to certain rules. Method 1 treats connection strength as the only evidence available for determining those weights.

o *Normalization method 2* One could argue that the formula above does not address the situation shown in Figure 11. In Figure 11(a) a choice has three options,

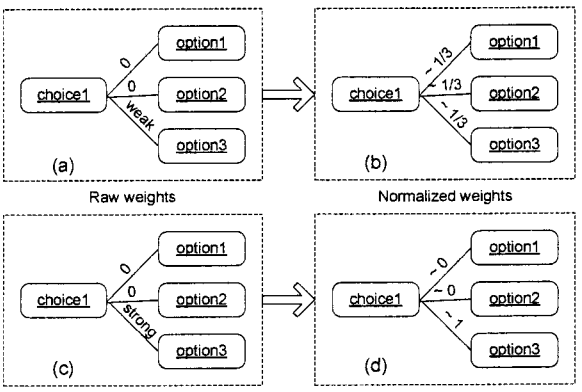

Fig. 11. Motivation for *Normalization method 2*

raw weights of two of which is zero and of the third one is small with respect to the other weights in the system. This means that RelDC has not been able to find any evidence that options 1 and 2 can be the right match and found insubstantial

evidence for option 3. Normalization method 1 will produce weights of 0 for the first two options and weight of 1 for the third option. One interpretation of this might be that the algorithm is 100% confident $option_3$ is the correct answer. One can argue that in such a situation, since the evidence for $option_3$ is very weak, more appropriate normalized weights should be roughly equal, i.e. the weight of each option should be close to $\frac{1}{3}$ in this case, as shown in Figure 11(b), and $option_3$ should have a slightly greater weight than that of $option_1$ and $option_2$.

Figure 11(c) is similar to Figure 11(a), but this time the weight of the third option is strong. Following the logic presented above weights of options 1 and 2 should be close to zero, the weight of $option_3$ should be close to 1, as in Figure 11(d).

To achieve such a normalization we assume that each option, just because it has been picked by fms in the first place, has a default small positive weight $\alpha \in \mathbb{R}^+$. The normalized weight $\overline{w}_{ij}$ is then computed as:

$$\overline{w}_{ij} = \frac{w_{ij} + \alpha}{\sum_j (w_{ij} + \alpha)}.$$

3.4 Resolving weights

This section addresses step 3(c) of RelDC approach shown in Figure 3. Recall, see Figure 8, that first a string-based method is utilized to create initial matches, then choice nodes are created for values that can match more than one entity. If there is only one choice after that, then our algorithm will simply compute the weights of its options as described above and stop. However, if there are multiple choices, when resolving a particular choice it might turn out that the weight of its options will depend on the weight of options of other choices. In that case further analysis is needed.

System of nonlinear equations Once all simple paths between all attributes to be resolved are determined, the raw weights can be expressed as functions of normalized weight of other options (see Figure 8) and the new normalized weights can be expressed as functions of the raw weights, as described in Section 3.3.2. These dependencies can be expressed in the form of a system of nonlinear equations. Figure 12 shows a sample of such a system of equations produced as an output of

$$\begin{cases} //raw\ weight\ equations \\ w_{1,2} = 0.010204 \cdot \overline{w}_{4,3}/(\overline{w}_{1,3} + \overline{w}_{4,3}) + 0.5 \cdot \overline{w}_{4,2} \\ w_{5,6} = 0.104167 \\ w_{5,7} = 0.0 \\ \vdots \\ //normalized\ weight\ equations \\ \overline{w}_{8,9} = w_{8,9}/(w_{8,9} + w_{8,10}) \\ \vdots \end{cases}$$

Fig. 12. System of nonlinear equations

the algorithm for an experiment with real data.[3] In Figure 12, $w_{i,j}$ denotes the raw weight of the edge between nodes with id's i and j, $\overline{w}_{i,j}$ denotes normalized weight.

NLP problem A solution of such a system should adhere to additional constraint: all raw weights must be nonnegative values. Thus the problem can be converted to a nonlinear programming (NLP) problem where the constraint functions are specified as above and the objective function will be specified later in this section. There is a large body of work [Hillier and Lieberman 2001] that establishes conditions under which solutions for NLP problems exist or does not exist. We can always reformulate our problem to guarantee that a solution always exists as follows. Let n_i in the rest of this section denote the number of options of $choice_i$ and m denote the total number of choices. Each raw weight equation $w_{ij} = f_{ij}(\overline{w}_{11}, \ldots, \overline{w}_{mn_m})$ is transformed into $f_{ij}(\overline{w}_{11}, \ldots, \overline{w}_{mn_m}) - \delta_{ij} \leq w_{ij} \leq f_{ij}(\overline{w}_{11}, \ldots, \overline{w}_{mn_m}) + \delta_{ij}$, where δ_{ij} is the tolerance for w_{ij}. The objective function is set to minimize $y = \sum_{i,j} \delta_{ij}$. Such a system always has a solution because maximum possible connection strength is limited for each graph: for any graph $\exists M \in \mathbb{R}^+ : f_{ij}() < M, \forall i, j$. Thus solution (assuming normalization method 2) $w_{ij} = 0$, $\overline{w}_{ij} = \frac{1}{n_i}$, $\delta_{ij} = M$, $\forall i, j$ is one of the valid solutions, the goal of course is to find a better solution by minimizing all δ_{ij}'s.

The second issue that arises is what if there is more than one solution. Which solution should be the final answer then? This issue is normally resolved by associating an objective function in addition to the set of constraints. A feasible solution that minimizes/maximizes the objective function is presented as the solution. So our task is to specify an objective function in our context. We specify the objective function as minimization of $z = \sum_i \sum_{j=1}^{n_i} |\overline{w}_{ij} - \frac{1}{n_i}|$.

The motivation behind this formula is as follows. If multiple solutions exist one (conservative) approach that seems very intuitive is to try to avoid making disambiguation decisions if there is no direct evidence for those. For example, if weight $\overline{w}_{ij}$ of $option_{ij}$ can vary, try to make it as much neutral as possible by minimizing $|\overline{w}_{ij} - \frac{1}{n_i}|$. Notice, n_i is the number of options of $choice_i$ and if there is no evidence supporting any of the options the algorithm will output $\frac{1}{n_i}$ weight for each option. By specifying such an objective function the algorithm tries to abdicate from making decisions in the situations where there is not enough evidence: those cases instead are left for the analyst to handle manually later, if the threshold is set accordingly, see Section 3.3.

Another issue is to find an appropriate method that would solve our NLP problem. It is known that the general NLP problem is unresolved. Unfortunately we could not map our problem to any existing NLP problems, methods of solving of which are known. To solve our system we have tried several solvers. Given the scale of the problem (50K-100K equations) the solvers we used would often not terminate in a reasonable amount of time SNOPT[GAMS/SNOPT solver] solver which we have used in most of the experiments works well for smaller problems. To overcome this we also have implemented a simple iterative method described in Section A.1 in the appendix.

[3]The indexes in the subscript are changed for clarity.The method described in Section 4 is to be applied to that system.

4. OTHER ISSUES

Node relevance We have developed the above algorithm under the assumption that every node in a connection is equally important in disambiguating a choice. However in general some of the paths found by AllSimplePaths algorithm might go through certain types of nodes which are irrelevant or only partially relevant to the particular cleaning task. For example authors' similar music taste, given such information is included in the graph, is likely to have little impact on them writing papers together. In such cases an analyst can specify node weights given the particular cleaning task. Figure 10(a) depicts the case where a node is assigned relevance of u_1. In such a case a new formula for computing a link coefficient is $\frac{w_1 \cdot u_1}{w_1+w_2+w_3}$. In our setup for the author matching problem all nodes are relevant, i.e. their weight are 1, consequently node weights are not used. Normally there will be many nodes present in the graph, so assigning a node weight could present a challenge. Fortunately, these nodes will be of a limited number of node *types*, such as types "author" and "paper". It is expected that most of the nodes of the same type will get assigned the same weight reflecting their relevance.

Decay factor In current formula the fact that longer relationships chains are weaker is only partially captured. For example, assume in the input table shown in Table III attribute `org_name2`, instead of `org_name1`, needs to be cleaned, while the other attributes are known to be clean. Assume there is an extra tuple (I_6, boeing, U_x), consequently U_x can correspond to U_1, U_2, or U_3. By referring to Figure 1 one can see, especially if nodes "USA" and "aerospace" are removed, that U_x is likely to correspond to U_1. However, assuming all the nodes depicted in that figure have relevance of 1, neither any string-based similarity functions, nor the formula we have presented up until this point will be able to give preference to U_1. Thus in certain situations it can be desirable to shift towards the graph random walk model where the next step itself is done with certain probability. One way of achieving this is to specify a decay factor per node type. The modified formula for the link coefficient of a link going from node with decay factor of d_1 is: $\frac{w_1 \cdot u_1 \cdot d_1}{w_1+w_2+w_3}$.

5. DEALING WITH SCALE

Our data cleaning approach is based on complex graph analysis which becomes computationally expensive as the size of the problem increases. If we are to make the approach scale to large problem sizes, clever techniques to speed up the analysis must be designed.

As far as optimization techniques are concerned, the RelDC procedure can be logically divided into two phases: the *discovering relationship* (a.k.a. AllSimplePaths) phase and *weight computation* phase. Recall, for the weight computation phase we either use an off-the-shelf solver or our own iterative method which we call *Iterative Graph-based Data Cleaning (IGDC)*. We cannot apply any optimizations of the weight computation phase if a solver is used. So all the weight computation phase optimizations mentioned in this section, as well as most of the discussion in general, imply our iterative implementation of RelDC. Also for IGDC a clear bottleneck is the AllSimplePaths phase, whereas if a solver is used this is not necessarily the

case.[4]

The most computationally expensive part of our approach is *determining all simple paths* among a set of nodes. Since our problem and the well-studied problem of the maximum network flow (MNF) bear some similarity, we have considered utilizing it for our algorithm but realized later that MNF treats "weights" in principally different way and cannot be applied. Thus we had to devise our own optimizations. Note that in general the number of simple paths between two nodes can be very large[5]. In practice, the number of path is not that large, however a simple algorithm can make many redundant traversal while discovering those paths. We have developed several optimizations that reduce the number of redundant traversals improving performance by orders of magnitude. In the rest of the section we discuss several most important optimizations. We classify optimizations in three large categories summarized below:

I. Simplifying the problem by specifying rules, Section 5.1
II. Algorithmic optimizations of IGDC
  (1) Preprocessing optimizations
    (a) Use of ShortestPath algorithm, Section 5.2.1
  (2) Optimization for all (or only first) iterations
    (a) Use neighborhood information, Section 5.2.2
    (b) Use reachability information, Section 5.2.2
  (3) Optimization for speeding up the subsequent iterations
    (a) Store paths explicitly, Section 5.2.3
    (b) Use graph coloring, Section 5.2.3
III. Exploiting parallelism

The first category improves the efficiency of algorithm by adding additional restrictions to simplify the problem. The second category optimizes AllSimplePaths algorithm. The third category exploits parallelism: IGDC while processing choices (i.e. while computing the normalized weights) during a particular iteration i uses normalized weight of options only from iteration $i-1$. This provides an opportunity for executing the algorithm *in parallel* on multiple CPU's. That is, all choices can be divided into several groups and each CPU will be responsible for resolving all choices of one of the groups.

5.1 Constraining the problem

The problem can be simplified by adding additional constraints. In order to speed up the data cleaning process as well as to guide choice resolution, an analyst using such a system can specify *rules* to help avoid certain computations. Rules can be classified along two dimensions: domain dependence and purpose. Rules can be general (domain independent) or ad-hoc (domain dependent). In the purpose dimension we distinguish resolution and speedup rules, though rules that serve both

[4]The performance of the solve is greatly dependent on the max path limit parameter of AllSimplePaths, explained later in this section.

[5]Assume the worst case of fully interconnected graph of n nodes. Let us analyze the number of possible simple paths between any two nodes in the graph. There can be 1 path of length 1, $(n-2)$ paths of length 2 (pick one out of $(n-2)$ intermediate nodes), ..., $(n-2)!$ paths of length $(n-1)$. Thus the number of possible paths is exponential.

resolution and speed-up purposes exist. The resolution rules guide the choice resolution process, while speed-up rules are needed to speed-up the resolution process. Such rules can include:
(1) Limiting the maximum length of paths for AllSimplePaths algorithm. That is AllSimplePaths algorithm can be specified to look only for path of length less of equal than some parameter L, and to ignore path of length greater than L. This optimization is based on the premise that longer path tend to have smaller connection strength while RelDC will need to spend more time to discover those.
(2) Specifying path types of interest (or for exclusion) explicitly. For example, the analyst can specify that only paths of type "from any author node to any university node then to any author node" are of interest. Some of such rules are easy to specify, however it is clear that for a generic framework there should be some method (e.g., a language) for an analyst to specify more complex rules. Our ongoing work addresses the problem of such a language.
(3) Specifying a weight cut-off threshold. Assume $choice_i$ with n_i options. Such a threshold is specified per each $choice_i$, as some coefficient $\alpha \in (0, 1)$ multiplied by the average expected raw weight of each options: $\mathcal{T}_i = \alpha \cdot [(\sum_{j=1}^{n_i} w_{ij})/n_i]$. All options $options_{ij}$ with raw weights of less than $\mathcal{T}_i$ are removed from $choice_i$. The intuition is to remove options with raw weights that are too small relative to that of their peer options.

5.2 Optimizing AllSimplePaths algorithm

This section presents optimizations for AllSimplePaths algorithm. All optimizations are divided intro three categories. Section 5.2.1 describes one *preprocessing* optimization which is applied before the first invocation of AllSimplePaths. Section 5.2.2 presents optimizations applicable to *all* iterations. Finally, optimizations for speeding up the *subsequent* iterations (i.e., iterations number 2, 3, ...) are presented in Section 5.2.3.

5.2.1 *Preprocessing optimization.* Before executing costly AllSimplePaths algorithm for two nodes in the graph, one can first try to determine whether there is at least one path in a more efficient manner as follows. During the preprocessing phase "iteration zero" runs as any other iteration except for it uses a faster ShortestPath algorithm instead of AllSimplePaths. If ShortestPath algorithm is not able to find any legitimate simple path between given two nodes, then AllSimplePaths will not find any paths either. Thus the option will have the weight of zero and it is removed from further consideration.

5.2.2 *Optimizations for all iterations.*
(1) Utilizing neighborhoods for path pruning (NBH optimization) We use the term *neighborhood* of radius R_0 of a node v_0 for the set of all the nodes that are reachable from v_0 via at most R_0 links. Each member of the set is tagged with "the minimum distance to v_0" information.

The intuitive definition presented above can be rephrased formally: for graph $G = (V, E)$, the neighborhood of node v_0 of radius R_0, $\mathcal{N}(v_0, R_0)$, is a set of pairs:

$$\mathcal{N}(v_0, R_0) = \{(v, d) : v \in V,\ d = min\_dist(v, v_0),\ d \leq R_0\}.$$

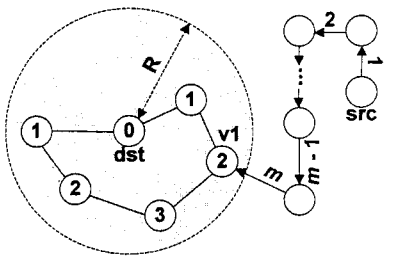

Fig. 13. Neighborhood information

A depth-first algorithm looking for all simple paths of length no greater than L can exploit neighborhood information to prune certain paths, see Figure 17. Let us assume the algorithm processes $choice_i$ created for attribute a of $entity_1$ and has k options: $option_{ij}$. The algorithm first computes the neighborhood $\mathcal{N}(entity_1, R)$ of radius $R : R \leq L$ around $entity_1$ node. Then for each option $option_{ij}$ it finds all simple paths from $option_{ij}$(source) to $entity_1$(destination) using a modified depth-first method. Assume the algorithm is at some intermediate stage where it currently observes a node $v_1 : v_1 \neq entity_1$ and the length of the intermediate path, i.e. the simple path from $option_{ij}$ to v_1, is m links. If m is such that $m + R < L$, the algorithm proceeds as the regular algorithm. If $m + R \geq L$, then v_1 must be inside $\mathcal{N}(entity_1, R)$: otherwise there is no need to continue with this path because even if it is possible to continue from v_1 and find a path to $entity_1$ the length of the resulting simple path will be no less than $L + 1$ exceeding our path length constraint of L. Further, if v_1 is inside $\mathcal{N}(entity_1, R)$ then it is possible to retrieve its min_dist information d to $entity_1$ from $\mathcal{N}(entity_1, R)$. Then m should be such that $m + d \leq L$: otherwise there is no need to continue with this path since its length is guaranteed to exceed L.

Let us introduce a new term *actual radius* of neighborhood $\mathcal{N}(v_0, R_0)$:

$$R_{act} = \max_{v \in \mathcal{N}(v_0, R_0)} (min\_dist(v_0, v)).$$

In practice, sometimes it happens[6] that $R_{act} < R_0$, i.e. the neighborhood of v_0 is a cluster which is not connected to the rest of the graph, then it might be possible to speed the algorithm further. In this situation $\mathcal{N}(v_0, R_{act})$ is equal to $\mathcal{N}(v_0, R), \forall R \in [R_{act}, \infty)$. In other words, we know the neighborhood of v_0 of radius ∞. Regarding searching all simple paths as described above, this means that all intermediates nodes must always be inside the according neighborhood. The AllSimplePaths algorithm given above should be modified in order to achieve the speedup – after computing $\mathcal{N}(entity_1, R)$ the following test should be done: if $R_{act} < R$ then $R \leftarrow \infty$.

(2) Using structural reachability for path pruning (SR optimization) Structural reachability (SR) optimization is similar to the optimization that utilizes neighborhoods (NBH): it speeds up the algorithm by pruning certain paths. SR optimization is based on the fact that often the graph on which the algorithm works

[6]Naturally, the greater the R_0 the more frequently this is likely to occur.

is not random, it is either structured or can be made structured. This is especially true for graphs constructed from relational tables of a relational DBMS.

Figure 2 shows the connectivity graph for the author matching problem. Observing this information it is clear that it is impossible to reach any node of *type* "paper" from any node of *type* "university" going via less than three links. One can create a table storing min_dist information for node types: the information about the minimum number of links one should traverse from any node of one type to reach any node of the second type. This table can be either created by the analyst or learned automatically. Similar to NBH optimization, using such information it is possible to prune certain paths. For example, assume the algorithm is using depth-first method for finding all paths (of length no greater than L). Assume it tries to reach a node of type *paper* and observes a node of type *university* and the length of the intermediate path is m links. Then if $m+3 > L$ then there is no need to continue with this path since even if it reaches the destination, the path length restriction will be violated.

Comparing SR and NBH As has been shown above, SR and NBH are quite similar, here we shall take up the differences between them. The advantage of SR optimization over NBH optimization is that SR utilizes the inherent structure of the graph while NBH needs first to invest time in building auxiliary data structures (i.e., neighborhoods). The advantage of NBH over SR, is that neighborhoods in general contain more information for pruning than the knowledge of reachability. SR provides information about the lower bound of the minimum number of links the algorithm will need to traverse from current node before it will reach a node of *type* equal to the type of the destination node. For example, it might be possible to reach some node of *type* paper in k links but this node is not guaranteed to be the destination node (a particular paper). NBH, if available, provides information about the exact minimum distance (in number of links) to the destination node. Thus SR algorithm is likely to make more redundant traversals than NBH. In our tests with a real dataset NBH optimization has shown better results than SR optimization. However SR optimization improves performance by an order of a magnitude in comparison to the the case where neither SR nor NBH are used.

5.2.3 *Optimizations for the subsequent iterations.*

(1) Storing discovered paths explicitly Once paths are discovered on the first iteration, they can be exploited for speeding up the subsequent iterations. One solution would be to store such paths explicitly in memory, if there is enough such, or on disk. Once paths are stored the subsequent iterations do not rediscover them, but rather work with the stored data.

Storing paths explicitly might not be always an acceptable solution in terms of memory and storage overhead, e.g. our rough experimental estimates show that storing paths in a compressed form for CiteSeer data for path length constraint of eight will require up to 4GB of storage space, whereas path length constraint of nine will require around 40GB of storage space. A space efficient solution using graph coloring is explored later in this Section.

Path compression We store paths because we need to recompute connection strength of the path which can change as weights of choice options change. An astute reader might notice that one way of compressing path information is to

find "fixed" path, i.e. path the connection strength of which will not change, and store the fixed connection strength (which actually can be aggregated with others) rather than the path itself. Given our definition of path connection strength, a path's connection strength is fixed if and only if none of the intermediate path nodes are incident to an edge weight of which might change.

(2) Graph coloring Storing the discovered paths explicitly in order to speed up the subsequent iterations, as discussed above is not very space efficient. This section proposes a solution which requires $O(|V|)$ space for graph $G = (V, E)$.

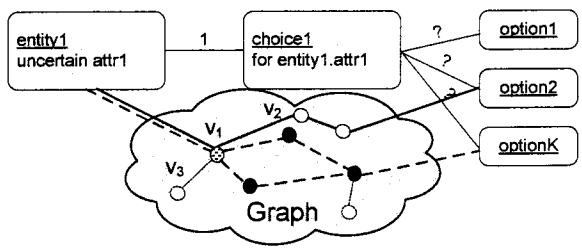

Fig. 14. Graph coloring

The main idea is the following. During the first iteration, while resolving say $option_k$ of $choice_1$, see Figure 14, all paths that are found are "colored" using the same color, say blue. Coloring of a path with a certain color means that all nodes in the path are assigned this color. Each node can have multiple colors, e.g. in general a node can have colors of blue and red at the same time, such as node v_1 in Figure 14. While resolving a different option say $option_2$ we color all discovered path with another color, say red. Now, during the second iteration coloring information can be utilized to prune certain paths. E.g. assume a modified depth-first algorithm needs to find all paths from $option_2$ to $entity_1$ after the first iteration and it knows that the path has been colored red during the first iteration, see Figure 14. Starting from $option_2$ the algorithm inevitably reaches node v_1 at which point it has several choices where to go next: it cannot go back to node v_2 because the path will not be simple one anymore, it cannot continue via v_3 because it is not colored, two other nodes have the wrong color (only blue color), the only choice where it can go is the destination node $entity_1$. Notice that the algorithm has been able to prune many dead-end directions following which it could have spent a lot of its time but which would have never led it to the destination.

Implementation of graph coloring In order to implement graph coloring each node needs to have an $n$bit "color" field. By default a node has no color, which corresponds to $n$bit vector with all bits set to 1: $\overline{1}$. A node has color i iff the i^{th} bit of its color has value of 0. Consequently a node cannot have more than n colors. To test if a node has color i one can simply do a bitwise `AND` operation with the node's color field and the i^{th} color's color-mask, $color_i$, defined as an $n$bit vector with all but i^{th} bits set to 0: $color_i = (1 << i)$. The reason why "no color" is $\overline{1}$ and not $\overline{0}$ is because that way it is easy to make algorithm ignore node color information by defining color-mask for "any color" as $\overline{0}$.

Picking colors A color is picked for choice-option pair once during the first iteration. Colors should be picked such that each color is present roughly equal number of times, i.e. the extreme case is when every node has just one color which is equivalent to not using the coloring optimization at all. The subsequent iteration should be able to determine which color was used during the first iteration. One way to implement that is to make the first iteration store this color explicitly, e.g. in a table, for each choice-option pair. Our implementation instead utilizes a hash function which maps a choice and option IDs to a number between 0 and $n-1$ to create one of the n colors. That way we do not need to keep a table for storing color assignments and achieve uniformity, i.e., each color is used roughly equally.

Limited number of colors Because the number of colors is limited to n, the same color, in general, will be assigned to multiple choice-option pairs. Consequently,the algorithm with the coloring optimization, might still explore paths which will not lead to the destinations. Predictably (and verified experimentally), the greater the n the more efficient the optimization and that the faster IGDC works, if the total number of nodes $|V|$ is fixed. Naturally this improvement is constrained since having more than $|V|$ colors is of little value. The side effect of having more colors is the space requirements: $O(n|V|)$ of bits of storage (i.e., n bits per each of $|V|$ nodes) is required for storing color information in every node. Also, if the number of colors is fixed, the greater the number of nodes the less efficient is the coloring optimization because more choice-option pairs will have the same color and many nodes will have (almost) all colors.

Removing not colored nodes If a node does not have any color it means it is not a part of any path and thus it can be removed from the graph. This will speed up the algorithm further because now it does not even have to retrieve the node in order to realize that it does not have a color. However one should be careful: removing the nodes means decreasing the degree of its neighbor nodes, which can disbalance the system which measure the connection strength. Thus it is imperative to store at each node aggregated information on links (such as link-weight) that were removed as the result of node removals.

6. EXPERIMENTAL RESULTS

In this section we experimentally study RelDC using real and synthetic datasets. We have used a Pentium 2GHz machine for our experiments. Our knowledge base contains information from two public-domain sources: CiteSeer[CiteSeer] and HPSearch[HomePageSearch]. Typically data cleaning approaches are tested for accuracy and efficiency. In our context two more types of tests are interesting: the effect of the number/amount of relationships and of longer relationships on the accuracy of RelDC. We have used our iterative implementation of RelDC (which we call IGDC) extensively because it does not suffer from the drawbacks mentioned in Section 3.4 and applicable to large number of equations (e.g. 100K). Using solver [GAMS/SNOPT solver] for smaller samples we have established the approximate solution of iterative implementation rarely fully coincide with the exact solution. Nevertheless, this section shows that the achieved accuracy of the disambiguation

based on such an approximate solution is very high.[7]

Some of the optimizations we have described (e.g., the graph coloring with less than 64 colors), without their further improvement (e.g., changing the graph coloring implementation to handle significantly more colors), while have showed excellent performance improvement for smaller dataset, they also showed not very significant improvement for very large datasets. In this section we will study the implementation of RelDC mostly with only those optimizations that are known to be very efficient not only for small datasets, but also for large datasets such as the whole citeseer. Two principal such optimizations are NBH and storing paths on disk. The optimization we have developed make the RelDC approach 1–2 orders of magnitude more efficient than a naïve implementation. In this section we will show that for IGDC (the iterative implementation of RelDC) *the bottleneck is AllSimplePaths algorithm*, not the weight computation part that follows it. Therefore most of the proposed optimization have been developed for optimizing AllSimplePaths. AllSimplePaths is specified to look for paths of length less or equal to L, where L is 8 by default.

In the context of the author matching problem, an entity is a paper and an uncertain attribute is one of the paper's unresolved authors for which a choice node is created. Frequently a single paper would have more than one author for which disambiguation is needed, therefore multiple choices are created for that one paper. Options of a choice are the corresponding authors. For author matching problem we use a modified depth-first AllSimplePaths algorithm which utilizes NBH (neighborhood) optimization, described in Section 5.

Data preparation We have constructed the author and paper tables as described in Section 2. The paper (input) table contains 573,123 tuples of type (`paper_id`, `FI`, `last_name`, `name1`, `name2`, `name3`), corresponding to 255,228 papers. The author (reference) table contains 176,626 tuples of type (`name1`, `name2`, `last_name`, `department`, `university`).

Accuracy on a real dataset In this context, the accuracy is the fraction of authors that are correctly resolved by the algorithm in fully automatic mode. We have applied RelDC to CiteSeer, for 8% of the authors it has computed only fms weights: these are 1-author papers or RelDC has not able to find any path, for the rest 92% it has computed non-trivial raw and normalized weights as well. However issues regarding the accuracy of the result are not trivial. We do not dwell any further on this issue except to observe that since the information about the real authors of the papers is not available, it is infeasible to determine the actual accuracy of the result.[8]

Instead in this section we propose a second experiment, which works with real data as well. It will demonstrate the accuracy of RelDC but not the accuracy of

[7]The accuracy of the approximate method is high for the following reason. Assume the exact solution for the three options of a particular choice is weights 0.8, 0.1, and 0.1. Assume the approximate solution is 0.7, 0.16, 0.14. Recall, those weights are *interpreted* to select the most likely correct match. In both cases the algorithm's output would be the same: that option 1 is the answer since it has the largest weight in both cases.

[8]We have resolved manually a random sample of authors (not in the above 8%) by going to their homepages containing their publications. The whole sample was resolved correctly but we could not test a large sample.

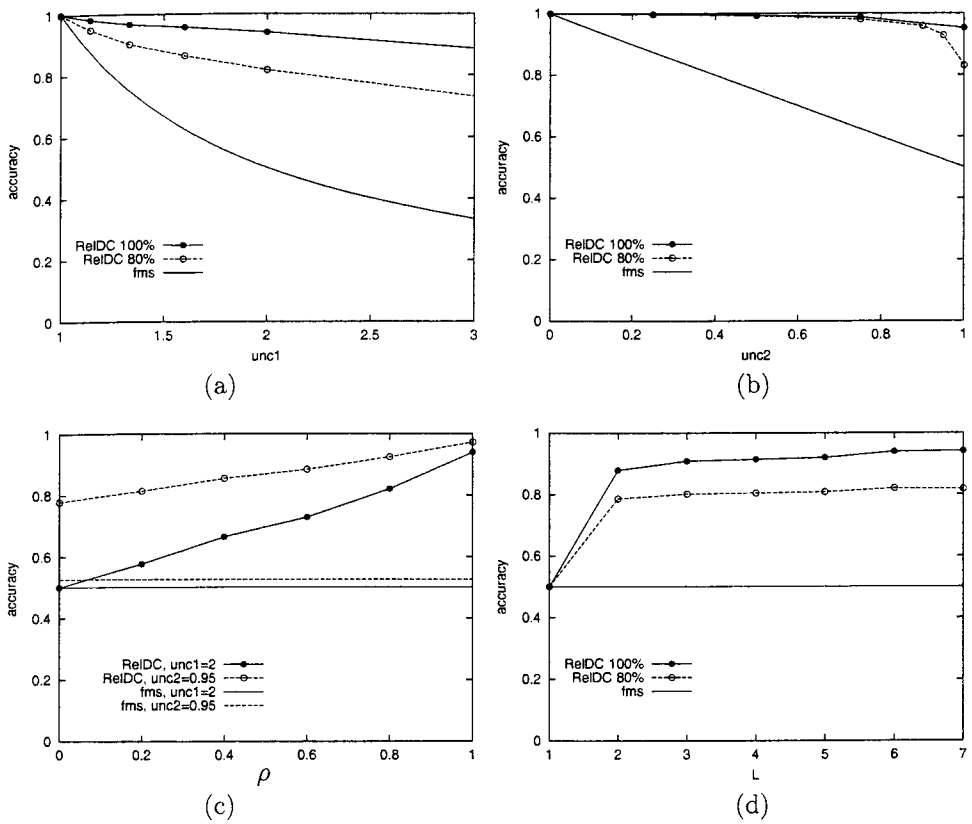

Fig. 15. Experimental Results

the above experiment.

Guessing author first names The idea of the experiment is to clean authors pretending only their first initials are known in the paper table and then see if RelDC guesses the right full first names of the authors correctly. For example, if we know from the paper table 'John Smith' is an author of paper P_1, then we pretend we only know 'J. Smith' is an author of P_1. 'J. Smith' can correspond, for example, to two authors in the *author* table: 'Jane Smith' and 'John Smith'. Note, in that case any fms algorithm must guess one candidate, whereas RelDC will do relationship analysis. If fms (RelDC) identifies record for 'John Smith' in the author table as its outcome, then it is a hit for fms (RelDC) since the first name 'John' is guessed correctly. In general, the fact that the author's first name is identified correctly does not imply the author is identified correctly[9], but it gives an idea about the quality of the algorithms.

To conduct such an experiment we need to prepare the data accordingly. In the *paper* table we now consider only those papers for which author full first names are

[9]For example, given "J. Doe" the algorithm can correctly identify the first name "John" but, if there are multiple John Doe's in the database, the algorithm can still pick an incorrect one.

available and that have at least one corresponding author in the *author* table. This procedures leaves 146,412 in the *paper* table, corresponding to 25.5% of CiteSeer. Next we pretend that for each author in the paper table only the first initial is known. The statistic about for how many authors (specified as only their FI, and last name) in the *paper* table there is exactly one (2, 3, ...) matching author(s) in in the *author* table characterizes the uncertainty in the system. For our experiment there is a direct one-to-one ($1 \rightarrow 1$) match for 82% of the cases, $1 \rightarrow 2$ for 9%, $1 \rightarrow 3$ for 3%, $1 \rightarrow 4$ for 2%, $1 \rightarrow 5$ for 1%, and the rest are less than 1% each.

Obviously, both fms and RelDC will have 100% accuracy for 82% of the authors with $1 \rightarrow 1$ match. First, we study the accuracy of those methods on the rest of 18% – the situation is interesting because the identities of all those authors are uncertain. For that case, the accuracy is computed as the fraction of choices resolved correctly. For these 18% the results are: fms's accuracy is 35.9%, RelDC has much higher accuracy of **63.2%**. Using more traditional definition of accuracy (i.e. in our context, the accuracy is the fraction of authors resolved correctly – including those with $1 \rightarrow 1$ match), the results are: fms's accuracy is $82\% + 18\% \cdot 0.359 = 88.5\%$, RelDC has higher accuracy of $82\% + 18\% \cdot 0.632 = 93.376\%$. In the rest of this section we assume the traditional definition of accuracy.

Accuracy on synthetic datasets A standard method of testing accuracy is to create an input dataset from a known dataset by introducing uncertainty in it. Thus we have created a dataset of 5000 papers and 1000 authors, which we have tried to make as reasonable as possible. Each author is affiliated with one of 25 universities each of which has 5 departments. Authors are divided into students and faculty. Most of the authors, including faculty, have (current or former) advisors. Advisors of authors who are currently students are likely to be from the same department of the same university; advisors of authors who are currently faculty are from random universities but likely from the same department type (e.g. "CS"). Correspondingly each advisor can have several generations of his students. A typical paper is written by a faculty member with other coauthors. Each coauthor can be (with different probabilities) a student of the faculty from one generation, another faculty member(student) from the same (different) department of the same (different) university. Notice, if there are no rules (e.g. such as above) of who writes papers, i.e., each coauthor in a paper is completely random, then RelDC is not applicable. Also recall that if a paper has *only one* author and after using fms a choice is created for that author, then RelDC will not be able to find any legal connections between options of the choice and the paper given the graph structure we have, see Figure 2. Thus the accuracy of RelDC for such cases is identical to that of fms. Therefore we will not consider such cases further and in our tests papers have 2, 3, or 4 authors most of the time.

Figures 15(a) and 15(b) demonstrate the effect of uncertainty on accuracy of RelDC and any fms for two different kinds of uncertainty. Each figure has three curves: one for fms and two for RelDC when for 100%(80%) of authors their universities/departments are known.

Uncertainty of type 1. In the first case, shown in Figure 15(a), there are N_{auth} (here always $N_{auth} = 1000$) unique authors which use N_{name} different names. Parameter $uncertainty_1$ in Figure 15(a) is $uncertainty_1 = \frac{N_{auth}}{N_{name}}$ ratio.

Example Let us see what this means if N_{name} is 750. For ease of explanation, assume author full names are integers. The name of author with ID k, where $k \in [0, 999]$, is computed as (k mod N_{name}) which is (k mod 750), since $N_{name} = 750$. Thus authors with IDs $0, \ldots, 749$ have names "0",...,"749". Authors with IDs $750, \ldots, 999$ have names "0",...,"249". Thus 500 authors (those with IDs $250, \ldots, 749$) have unique names (their names are "250",...,"749"). For each of the rest of 500 authors (those with IDs $0, \ldots, 249$ and $750, \ldots, 999$) there is another author with the identical name.

The authors in papers are always specified by their full names. Notice, for each author whose name is not unique, one can never identify with 100% confidence any paper this author has written, thus uncertainty for such authors is very high. When $uncertainty_1$ is 1, then there is no uncertainty and all methods show accuracy of 1. As expected, accuracy monotonically decreases as uncertainty increases. When $uncertainty_1$ is 2 uncertainty is very large: for any given author there is exactly one another author with the identical name. For this case, any fms have no choice but to guess one of the two authors, thus the accuracy of any fms, as shown in Figures 15(a), is 0.5. The accuracy of RelDC 100% (RelDC 80%), given the uncertainty, is quite high : 94%(82%). The gap between RelDC 100% and RelDC 80% curves shows that for these settings RelDC relies substantially on author affiliations for the disambiguations.

Uncertainty of type 2. Figure 15(b) shows the effect of different kind of uncertainty on the accuracy of RelDC. For simplicity of explanation of $uncertainty_2$, we will use integers for names again. The first name of author with ID of k, where $k \in [0, 999]$, is given as "F<k>" and last name as "(k mod 500)". For example, author with ID of 1 has name (F1, 1), author with ID of 501 has name (F501, 1). Thus if a full name of an author is given in a paper, then this author can be uniquely identified, but if only the first initial is given for the first name, then that author name can correspond to exactly two authors, e.g. author "F. 1" can correspond to both (F1, 1) and (F501, 1). Parameter $uncertainty_2$ in Figure 15(b) corresponds to fraction of author names in the *paper* table specified by only the first initial and last name. If this fraction is 0, then there is no uncertainty and the accuracy of all methods is 1. Also notice that the case when $uncertainty_1$ is 2 is equivalent to the case when $uncertainty_2$ is 1. Notice, in this test there is much less uncertainty than in the previous one: each author name is unique and for each author there is a chance that for some of his/her papers he/she is known as an author with 100% confidence. The accuracy decreases as uncertainty increases, but this time the accuracy of RelDC is much higher: it stays for both curves well above 95% when uncertainty is less than 0.9. The fact that curves for RelDC 100% and RelDC 80% are almost indiscernible until $uncertainty_2$ reaches 0.9, shows that RelDC relies less heavily on weak author affiliation information but rather on stronger connections via papers. Thus we have empirically shown the superiority of RelDC to any fms methods for achieving high disambiguation accuracy when information about the relationships among the entities is available.

Importance of relationships Figure 15(c) studies what effect the amount of relationships has on the accuracy of RelDC. In our experiments one can identify two major types of relationships: one is via papers introduced by the *paper* table,

the other is via affiliations, introduced by the *author* table. The x-axis shows the fraction of authors ρ for which their affiliation is known. If ρ is zero than the affiliation relationship is eliminated completely and RelDC has to rely solely on the knowledge stored in the *paper* table. If ρ is one that complete knowledge of author affiliations is available. Figure 15(c) shows four curves for accuracy as function of ρ: fms and RelDC for the case as in Figure 15(a) when $uncertainty_1$ is 2 and for the case as in Figure 15(b) when $uncertainty_2$ is 0.95.

The accuracy increases as ρ increases showing that the current formula factors in new relationships well. The accuracy of "RelDC $unc_1 = 2$" when ρ is 0 equals that of fms: 0.5. This is because when unc_1 is 2, for each author there is exactly one author with an identical name, but there is no affiliation information that can help disambiguate between authors.

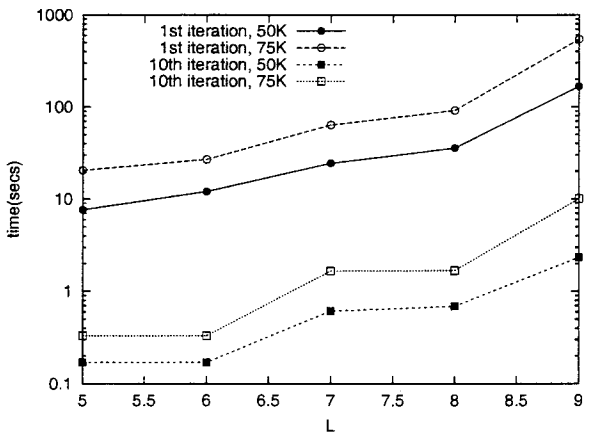

Fig. 16. The effect of L on the efficiency

Longer relationships Figure 15(d) examines the effect of path limit parameter L on the accuracy. The accuracy increases as L increases. It increases rapidly as L reaches 2 since *author*→*paper*→*author* relationship can be discovered and increases slowly after that. Tests with very long relationships have not been conducted due to slow performance of RelDC for such cases. In general, larger L leads to higher accuracy. The practical usefulness of longer paths depends on the combination of other parameters. For example, the difference between accuracy when L is 2 and when L is 7 is (a) 3% for RelDC 80%, (b) it is more substantial 6.2% for RelDC 100%, and (c) it is substantial 9.2% for RelDC 100% when the number of universities in the model is decreased to ten[10].

Efficiency of RelDC To prove the applicability of our approach to a large dataset we have used it to clean CiteSeer datasets described above. It take the algorithm approximately 6 hours to complete.The accuracy issues regarding this dataset are explained at the beginning of this section. The rest of the section will consider (faster) experiments with subsets of CiteSeer.

[10]The case is not shown in the figure.

Figure 16 studies the execution time of the 1^{st} and 10^{th} iterations of iterative implementation of RelDC as function of L. The tests are on 50K and 75K tuples from the *paper* table. This corresponds to 8.7% and 13% of CiteSeer, 10K and 15K choice nodes are created. Predictably, the execution time increases as the path limit L increases since more paths are found. The execution time of the 10^{th} iteration is much smaller than that of 1^{st} iteration because we apply optimization techniques to be able to find all simple paths more efficiently after those paths have been discovered once by AllSimplePaths on the first iteration.

NBH optimization

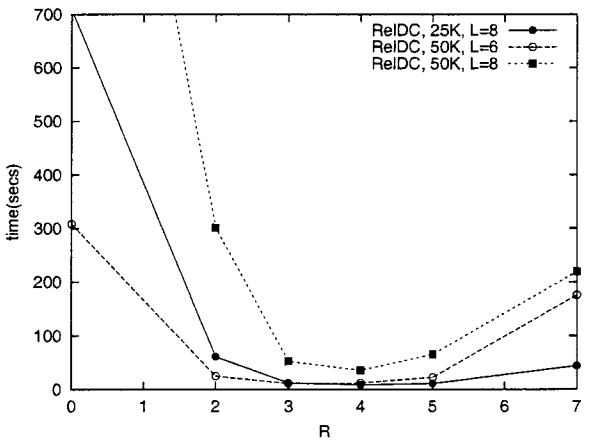

Fig. 17. NBH optimization

Figure 17 shows the execution time of RelDC as a function of neighborhood radius R. Radius of zero means the NBH optimization is not used. When R is set to $\lceil \frac{L}{2} \rceil$, the performance improves 28 times for 25K tuples and L of 6, it improves 82 times for L of 8. Performance improvement of 1–2 orders of magnitudes is typical for this optimization.

Solver

In our experiment, we use the General Algebraic Modeling System (GAMS) to solve our system. GAMS[GAMS] is a high-level modeling system for mathematical programming problems. It consists of a language compiler and a bunch of integrated solvers. GAMS transforms the mathematical representation to representations required by specific solver engines, such as MINOS[B. Murtagh and Kalvelagen], CPLEX[GAMS/CPLEX 9.0 user manual], OSL[Solutions and Library], SNOPT[GAMS/SNOPT solver], etc. By using GAMS, we can focus on modeling itself.

A GAMS model is a collection of statement in the GAMS Language. The GAMS language is similar to commonly used programming languages. Data are entered only once in list and table form. Models are described in concise algebraic statements which are easy to read for both humans and machines.

GAMS supports a broad range of model types, including linear programming, mixed integer programming and different forms of nonlinear programming. There

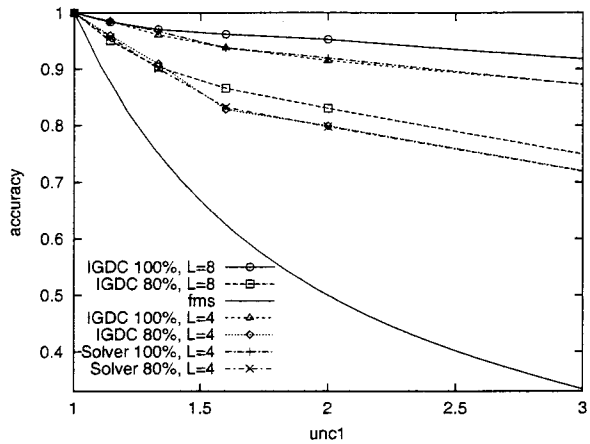

Fig. 18. Solver

are many solvers incorporated in GAMS. Different solver is capable of different model type(s). We only focus on solvers capable of nonlinear programming models. We have tried out several solvers, including MINOS[B. Murtagh and Kalvelagen], CONOPT[solver manual], SNOPT[Gill et al. 1997] and find out SNOPT (from Stanford) has the best output. This is probably because SNOPT is designed for large scale programs. It minimizes a linear or nonlinear function subject to bounds on the variables and sparse linear or nonlinear constraints, which is exactly our case.

The solver-based solutions proceeds as follows. First RelDC discovers all relevant simple paths, creates the corresponding system of equations, such as in Figure 12, and output those into a file. Then a special module takes the system of nonlinear equations from that that file as input and builds the according GAMS model. We use the tolerance based model as described in previous sections. For experiments on real citeseer data, the size of the models is usually more than 100K equations. For experiments on synthetic data, the size of the models varies depending on different path length and uncertainty we choose. Then one of the solvers, which understands GAMS, can be used to solve the problem. After the weights are found by the solver, they are interpreted to determine the most likely correct match in the identical fashion to that of iterative solution.

For example, for synthetic data (Solver 100%), if uncertainty is 2 (i.e., there are $1,000$ authors and the number of different author names is 500), the number of equations in the system is $30,464$. The size of the equation file is 1.66MB. We transformed it to a GAMS model file and the size of it is 2.5MB. It took 24 seconds for GAMS to run the solver and get the results on a P4 2.4G machine. The GAMS outputs a file with the size of 12.8MB. We use the tolerance based model and all the deltas are 0 at the end. We extract the values of weights variables and get the accuracy of 92.002%. If the uncertainty is 1.333 (i.e., the number of different author names is 750), there are 25186 equations in the system and the size of the equation file is 1.58MB. The transformed GAMS model file has the size of 2.3MB. The solver needs 17 seconds to get the results. The size of the GAMS output file

is 10.8MB. There are 18 non-zero deltas in the final results. The accuracy we get from using solver at this point is 96.73%.

7. RELATED WORK

The problem of data cleaning is a very important problem that has been studied extensively in the literature. It is also referred to as record linkage[Newcombe et al. 1959; Newcombe and Kennedy 1962; Fellegi and Sunter 1969], merge/purge[Hernandez and Stolfo 1995; 1998], object identification[Tejada et al. 2001], duplicate elimination[Bitton and DeWitt 1983; Ananthakrishna et al. 2002], or reference matching[?] etc. by different communities.

Probabilistic linkage technique has been used since the pioneer work of Fellegi and Sunter[Fellegi and Sunter 1969] who provide the theoretical foundation for the subsequent work. The basic idea is to view the problem of identifying matching records as classification task and use probabilistic model to decide matching and non-matching record-pairs. For the concern of the efficiency, they provide a blocking mechanism so that only records in the same block are compared.

Several methods follow the spirit of blocking mechanism of the Fellegi-Sunter theory and address the computational complexity and scalability issue. The *sorted neighborhood* method[Hernandez and Stolfo 1995] use multiple keys to sort the database and compare only those records within a sliding window. The *canopy* method[McCallum et al. 2000] uses an extremely inexpensive string distance metric, such as TF-IDF distance metric, to output overlapping clusters that contains possible matching records. More recently, Cohen et al. propose a scalable and adaptive methods for clustering and matching identifier names, in the sense that they can be trained to obtain better performance[Cohen and Richman 2002].

To improve the matching accuracy, many methods are proposed, varying by the extent to which human expertise are involved or the machine learning techniques are used. Rule-based methods require the most human involvement and are knowledge intensive. Human experts need to specify the conditions in which records are equivalent[Hernandez and Stolfo 1995; Galhardas et al. 2001; Lee et al. 2000; Raman and Hellerstein 2001]. A declarative rule language is presented in [Galhardas et al. 2001] to enable users to express the specifications.

Probabilistic methods are developed after the Fellegi-Sunter framework and use unsupervised machine learning methods. The powerful *expectation maximization (EM)* algorithm is employed to classify record pairs into the three classes: matched, non-matched, and possible matched based on statistical properties without any training data[Winkler 1994]. A domain-independent unsupervised approach is to treat the matching task as an information retrieval problem[Cohen 1998]. The approach uses the TF-IDF weighting scheme and employs cosine similarity in the vector space. *Database hardening* approach[Cohen et al. 2000] considers the problem of inferring the most likely "hard" (precise) database from a "soft" (noisy) database constructed from heterogeneous sources and which contains inconsistencies and duplication. It uses a graph of similarity values between records to obtain the best global record matching.

Many efforts have been made to solve the linkage problem by exploiting textual similarity. There are a number of distance metrics proposed by different commu-

nities, including edit-distance metrics[Monge and Elkan 1996; 1997], Jaro metric and its variants[Jaro 1989; 1995; Winkler], TF-IDF distance metrics based on token[Cohen 2000; Gravano et al.],and hybrid methods. A comparison of various string distance metrics are presented by [Cohen et al. 2003].

The most recent work on on-line domain independent data cleaning is by Chaudhuri et. al [Chaudhuri et al. 2003]. In this publication the authors propose a new string similarity function which is based not only on edit distance but also uses IR's tf-idf concept for creating a better similarity function. In order to speedup retrieval of top-K most likely candidates for cleaning a particular tuple, [Chaudhuri et al. 2003] propose to use the error tolerant index relation. Recall, in this paper we are using the term fms from [Chaudhuri et al. 2003], but we refer to *any* string-based similarity match functions.

Recently, researchers in AI community employ supervised machine learning techniques, such as Bayesian decision model[Verykios et al. 2003], Hidden Markov model[Christen et al. 2002], and Markov chain Monte Carlo[Pasula et al. 2002] to make the matching decision. The methods of learning string similarity to classify matched and unmatched records are highly interested[Ristad and Yianilos 1998; Sarawagi and Bhamidipaty 2002; Tejada et al. 2002; Bilenko and Mooney 2003].

In [Lee et al. 2004], the authors developed a method to determine the context similarity of records and identify the spurious links. A concept hierarchy is employed and association-rules mining is applied to the database to discover all associations among the attribute values, in order to identify the context attributes. The similarity of records is determined by how similar their context attributes are. This method is similar to our work in the sense of using context (relationship in our jargon), while our approach is more general and domain independent.

8. CONCLUSION AND FUTURE WORK

In this paper we have presented a domain-independent data cleaning approach called RelDC. The algorithm utilizes a completely different paradigm from conventional methods: it analyzes relationships between entities. The algorithm is largely self-tuning, i.e., it tunes itself based on the interconnections between entities in each particular situation, however some additional tuning capabilities, such as node relevance, are provided. We have empirically demonstrated that RelDC shows high accuracy as well as its applicability to a large dataset. Most importantly we have shown that relationships when available are important for data cleaning.

Many other issues open up that will be addressed in our future work. They can be divided into two categories: solving the same problem differently and solving the problem in the wider context. First of all, different from our's implementations of RelDC are possible. The most interesting include: computing all simple paths completely inside a relational DBMS using SQL, measuring connection strength by analyzing frequency of paths types between directly connected entities, using machine learning techniques for learning path weights. Issues of the second category are as follows. First, in practice reference tables can have missing entries and duplicates. Further there can be no reference table as such but rather two (not cleaned) sources of information will need to be merged. Another important issue is that of designing a general purpose cleaning toolkit for relational databases bases

on our approach. Such a toolkit would take a relational database and help analyst to clean data seamlessly. Designing such a tool has many challenges: there is need for a language for specifying rules (for path specification), there should be an easy way for specifying parameters of the algorithm and guiding graph construction.

REFERENCES

ANANTHAKRISHNA, R., CHAUDHURI, S., AND GANTI, V. 2002. Eli- minating fuzzy duplicates in data warehouses. In *Proc. of VLDB Conf.*

B. MURTAGH, M. SAUNDERS, P. G. R. R. AND KALVELAGEN, E. MINOS: A solver for large-scale nonlinear optimization problems. http://www.gams.com/solvers/minos.pdf.

BILENKO, M. AND MOONEY, R. 2003. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge Discovery and Data Mining (KDD-2003).*

BITTON, D. AND DEWITT, D. J. 1983. Duplicate record elimination in large data files. *ACM Transactions on Database Systems (TODS) 8,* 2 (June), 255–265.

CHAUDHURI, S., GANJAM, K., GANTI, V., AND MOTWANI, R. 2003. Robust and efficient fuzzy match for online data cleaning. In *Proc. of ACM SIGMOD Conf.*

CHRISTEN, P., CHURCHES, T., AND ZHU, J. X. 2002. Probabilistic name and address cleaning and standardisation. The Australasian Data Mining Workshop.

CITESEER. http://citeseer.nj.nec.com/cs.

COHEN, W. 2000. Data integration using similarity joins and a word-based information representation language. *Transactions on Information Systems 18,* 3 (Jul).

COHEN, W., KAUTZ, H., AND MCALLESTER, D. 2000. Hardening soft information sources. In *Proc. of KDD Conf.*

COHEN, W. W. 1998. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 201–212.

COHEN, W. W., RAVIKUMAR, P., AND FIENBERG, S. E. 2003. A comparison of string distance metrics for name-matching tasks. IIWeb Workshop 2003.

COHEN, W. W. AND RICHMAN, J. 2002. Learning to match and cluster large high-dimensional data sets for data integration. In *Proceedings of KDD-2002.*

DISCOVERY, K. 2003. www.kdnuggets.com/polls/2003/data_preparation.htm.

FELLEGI, I. AND SUNTER, A. 1969. A theory for record linkage. *Journal of the American Statistical Association 64,* 328, 1183–1210.

GALHARDAS, H., FLORESCU, D., SHASHA, D., SIMON, E., AND SAITA, C.-A. 2001. Declarative data cleaning: Language, model, and algorithms. In *Proc. of VLDB Conf.*

GAMS. http://www.gams.com/.

GAMS/CPLEX 9.0 USER MANUAL. http://www.gams.com/solvers/cplex.pdf.

GAMS/SNOPT SOLVER. www.gams.com/solvers/.

GILL, P., MURRAY, W., AND SAUNDERS, M. 1997. SNOPT: An sqp algorithm for large-scale constrained optimization. Report NA , 97-2.

GRAVANO, L., IPEIROTIS, P., JAGADISH, H., KOUDAS, N., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. Approximate string joins in a database (almost) for free. In *VLDB01.*

HERNANDEZ, M. AND STOLFO, S. 1995. The merge/purge prob- lem for large databases. In *Proc. of SIGMOD.*

HERNANDEZ, M. A. AND STOLFO, S. J. 1998. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery 2,* 1 (January), 9–37.

HILLIER, F. AND LIEBERMAN, G. 2001. *Introduction to operations research.* McGraw-Hill.

HOMEPAGESEARCH. http://hpsearch.uni-trier.de.

JARO, M. 1989. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association 84,* 406, 414C420.

JARO, M. 1995. Probabilistic linkage of large public health data files. *Statistics in Medicine 14,* 5C7 (Mar.CApr.), 491C498.

KALASHNIKOV, D. AND MEHROTRA, S. RelDC: a novel framework for data cleaning. RESCUE-TR03-04.

LEE, M., HSU, W., AND KOTHARI, V. 2004. Cleaning the spurious links in data. *IEEE Intelligent Systems*, 28–33.

LEE, M., T.W.LING, AND W.L.LOW. 2000. Intelliclean: A knowledge-based intelligent data cleaner. In *Proc. 6th Int'l Conf. Knowledge Discovery and Data Mining (KDD2000).*

MANNILA, H. AND RAIHA, K.-J. 1992. *The design of relational databases.* Addison-Wesley.

MCCALLUM, A. K., NIGAM, K., AND UNGAR, L. 2000. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining (KDD-2000).* 169–178.

MONGE, A. E. AND ELKAN, C. 1996. The field matching problem: Algorithms and applications. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96),Portland, OR.* 267–270.

MONGE, A. E. AND ELKAN, C. P. 1997. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proceedings of the SIGMOD 1997 Workshop on Research Issues on Data Mining and Knowledge Discovery,Tuscon, AZ.* 23–29.

NEWCOMBE, H. B. AND KENNEDY, J. M. 1962. Record linkage making maximum use of the discriminating power of identifying information. *Communications of the ACM 5,* 563–566.

NEWCOMBE, H. B., KENNEDY, J. M., AXFORD, S. J., AND JAMES, A. P. 1959. Automatic linkage of vital records. *Science 130,* 954–959.

PASULA, H., MARTHI, B., MILCH, B., RUSSELL, S., AND SHPITSER, I. 2002. Identity uncertainty and citation matching. In Advances in Neural Processing Systems 15. Vancouver, British Columbia:MIT Press.

RAMAN, V. AND HELLERSTEIN, J. 2001. Potter's wheel: An inte- ractive data cleaning system. In *VLDB Journal.*

RISTAD, E. AND YIANILOS, P. 1998. Learning string edit distance. *IEEE Trans. Pattern Analysis and Machine Intelligence 20,* 5 (May), 522–532.

SARAWAGI, S. AND BHAMIDIPATY, A. 2002. Interactive deduplication using active learning. In *Proceedings of the Eighth ACM SIGKDD international conference on Knowledge Discovery and Data Mining (KDD-2002).*

SOLUTIONS, I. O. AND LIBRARY. http://www-306.ibm.com/software/data/bi/osl/.

SOLVER MANUAL, C. http://www.gams.com/solvers/conopt.pdf.

TEJADA, S., KNOBLOCK, C. A., AND MINTON, S. 2001. Learning object identification rules for information integration. *Information Systems 26,* 8 (December), 607–633.

TEJADA, S., KNOBLOCK, C. A., AND MINTON, S. 2002. Learning domain-independent string tranformation weights for high accuracy object identification. In *Proceedings of the Eighth ACM SIGKDD international conference on Knowledge Discovery and Data Mining (KDD-2002),Hong Kong, China.*

VERYKIOS, V., G.V.MOUSTAKIDES, AND ELFEKY, M. 2003. A bayesian decision model for cost optimal record matching. *The VLDB Journal 12,* 28–40.

WINKLER, W. The state of record linkage and current research problems. In *U.S. Bureau of Census, TR99.*

WINKLER, W. E. 1994. Advanced methords for record linkage. In *U.S. Bureau of Census.*

A. RESOLVING WEIGHTS INTERNALLY

A.1 An iterative solution

One can base the final disambiguation decision not on the exact solution for the system of equations, but on an *approximate* solution obtained after a certain number of iterations of an iterative method for solving the system. In the experimental

section we show that using even a few iterations (e.g., 10–20) RelDC can achieve very high accuracy. Currently the system is written in the form

$$\begin{cases} (w_{11}, \ldots, w_{mn}) &= \vec{F}_1(\overline{w}_{11}, \ldots, \overline{w}_{mn}) \\ (\overline{w}_{11}, \ldots, \overline{w}_{mn}) &= \vec{F}_2(w_{11}, \ldots, w_{mn}) \end{cases}$$

The iterative semantics is introduced naturally if one treats computation of the current raw weights as a function of normalized weights from previous iteration. Let "(i)", e.g. as in $w_{11}^{(i)}$, denote i^{th} iteration. Then an iterative equivalent for the system above is

$$\begin{cases} (w_{11}^{(i)}, \ldots, w_{mn}^{(i)}) &= \vec{F}_1(\overline{w}_{11}^{(i-1)}, \ldots, \overline{w}_{mn}^{(i-1)}) \\ (\overline{w}_{11}^{(i)}, \ldots, \overline{w}_{mn}^{(i)}) &= \vec{F}_2(w_{11}^{(i)}, \ldots, w_{mn}^{(i)}) \end{cases}$$

To obtain an approximate solution first equal weights of $\frac{1}{n_i}$ are assigned to each option $option_{ij}$ of $choice_i$, where $choice_i$ has n_i options total. Then several iterations are carried out. The final matching decision is based on the approximate solution to the original system, obtained after several iterations of the algorithm. To obtain a better approximation of the solution one can speed up the convergence by first applying the substitution method for constant weights as described in Section A.2 and only then carrying out the iterative computations for "dynamic" weights only.

A.2 Solid evidence based weight computation

The raw weight of an option is composed of weights of each individual path as shown in Equation 1 in Section 3.3.1. The weight of a path is *constant* if it does not depend on weights of other options. It might turn out for a particular option the weight of each such path is constant, thus we refer to such a raw weight as *constant* and to other raw weights as *dynamic*. In Figure 12, weights $w_{5,6}$ and $w_{5,7}$ are constant, weight $w_{1,2}$ is dynamic. Based on the equations we can also classify normalized weighs into the same two categories: if they cannot change we call them *constant*, if they might – *dynamic*. Weight $\overline{w}_{8,9}$ in Figure 12, without further knowledge of $w_{8,9}$ and $w_{8,10}$, is dynamic.

A typical example of a constant weight equation is an equation of type $w_{i,j} = const$. For many of those *const* is simply zero, such as in the equation for $w_{5,7}$ in Figure 12, created when AllSimplePaths algorithm is not able to find any paths.

• **Substitution phase** Any occurance of constant weight variables can be substituted with their values in all equations. This will eliminate the dependence of those equations on those variables, possibly changing some of the dynamic weights into constant ones. This process is repeated until such a substitution is no longer possible. Because the total number of variables is limited the process is guaranteed to terminate. After the substitution process terminates, in general, some of the weights will be constant some of the weights will still remain dynamic.

Example For the system in Figure 12 every occurance of $w_{5,7}$ in the right side of all equations can be substituted with 0. If normalization method 1 is used, the equation for $\overline{w}_{5,7}$ will become $\overline{w}_{5,7} = 0$. Now the procedure can be repeated for $\overline{w}_{5,7}$.

• **Weight computation phase** The final weights are computed as follows. No

further processing is needed for the constant weights. The dynamic weights are processed in the following manner. Recall, the connection strength between two nodes is computed as the sum of the connections strengths of each individual path. The *solid evidence* way of computing weights is to sum the connections strength of only those paths which have constant weights, ignoring the paths with dynamic weights.

B. PROBABILISTIC MODEL

Notation	Meaning
$\exists x$	event "x exists" for some entity x
$\nexists x$	event "x does not exist" for some entity x
$\mathrm{P}_{\exists}(x)$	probability that x exists, i.e. $\mathrm{P}_{\exists}(x) = \mathrm{P}(\exists x)$
$\mathrm{P}_{\rightarrow}(E)$	probability to follow (edge) E, i.e. $\mathrm{P}_{\rightarrow}(E) = \mathrm{P}$(to follow E)
$\mathcal{P}$	the path being considered
v_i	a node on the path
E_i	edge (v_i, v_{i+1}) on the path
$E_{i,j}$	edge labeled with probability $p_{i,j}$, i.e. $\mathrm{P}_{\exists}(E_{i,j}) = p_{i,j}$
$a_{i,j}$	if $\exists E_{i,j}$ then $a_{i,j} = 1$ else $a_{i,j} = 0$, i.e. $\mathrm{P}(a_{i,j} = 1) = p_{i,j}$
$\mathbf{a}$, as a vector	$\mathbf{a} = (a_{1,1}, \ldots, a_{k,n_k})$
$\mathbf{a}$, as a set	$\mathbf{a} = \{a_{i,j} : i = 1, \ldots, k, j = 1, \ldots, n_i\}$
$\mathbf{a}$, as a variable	at each moment variable $\mathbf{a}$ is one instantiation of $\mathbf{a}$ as a vector
WM	the weight-based model
PM	the probabilistic model
WF	the WM's formulae for computing connection strength
PF	the PM's formulae for computing connection strength

Table V. Probabilistic model: Terminology

B.1 Introduction

One can call the model for computing connection strength presented so far as a *weight-based model (WM)*. To compute such a connection strength each edge is assigned some weight which reflects the degree of confidence the relationship between the two entities exists and the connection strength is computed based on the graph structure and those weights. Another interesting model that one can consider is a *probabilistic model (PM)*, where weights of edges are treated not as weights but as probabilities. In this section we discuss challenges of creating such a model and utilizing it in practice.

We have used the following philosophy in this section. PM is more complex that WM and PM needs more time to compute but nevertheless, as we will show later, the results of these models in terms of accuracy are very similar. Thus we do not advocate the use of PM since it is slower than WM while achieving similar accuracy. PM is interesting because, as explained later, it treats choice nodes more properly than WM given specific semantics of a choice node. We show that, because the existence of an edge can depend on the existence of other edges and other factors, the formal probabilistic model can quickly become computationally infeasible if one does not make any assumptions to simplify the model. One could abandon this

model when such assumptions need to be made, instead we present the reader with alternatives on how to simplify the model.

Before we proceed let us introduce the notation used in this section, see Table V. The meaning of this notation will become clear as we go along: it can be useful to refer back to Table V if terminology is not clear in one of the subsequent sections. We use $\exists x$ to denote event "x exists" for some entity x. Similarly, we use $\nexists x$ to denote event "x does not exist". Notation $\mathrm{P}_{\exists}(x)$ denotes the probability that x exists. Let $\mathrm{P}_{\rightarrow}(E)$ denote the probability to follow (edge) E, usually in the context of a specific path. Notation $\mathcal{P}$ denotes the path being considered. We will use v_i denote i^{th} node on the path as in Figure 21. Let E_i denote edge $(v_i, v_i + 1)$ on the path. Let $E_{i,j}$ denote edge labeled with probability $p_{i,j}$, notice $\mathrm{P}_{\exists}(E_{i,j}) = p_{i,j}$. Notation $a_{i,j}$ is defined as follows: `if` $\exists E_{i,j}$ `then` $a_{i,j} = 1$ `else` $a_{i,j} = 0$, notice $\mathrm{P}(a_{i,j} = 1) = p_{i,j}$. We use WM and PM to denote the weight-based and probabilistic models.

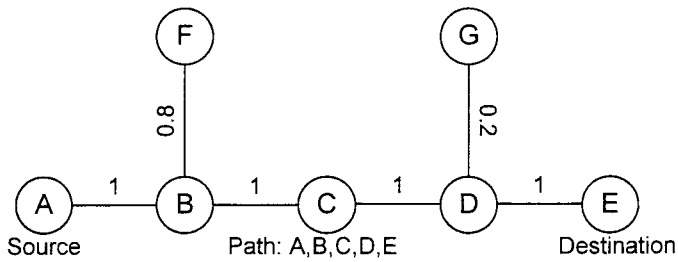

Fig. 19. Probabilistic model

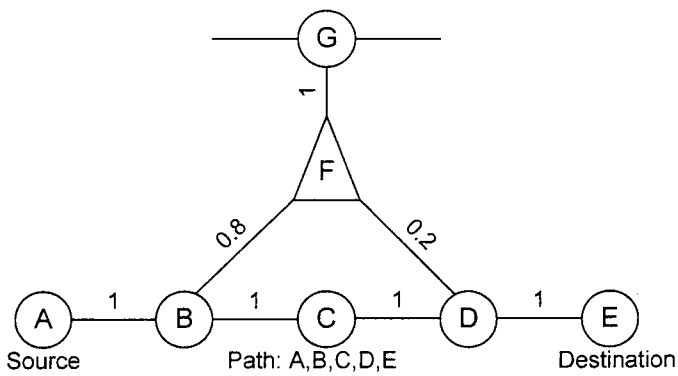

Fig. 20. Probabilistic model

Let us introduce PM by analyzing two examples shown in Figures 19 and 20. Let us consider how computation of connection strength will be different if one treats labels of edges as probabilities and not as weights. Both figures show a part of the graph with path $\mathcal{P}$=$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ which will be of interest to us. In Figure 19 we assume probability of an event "edge BF is present" is 0.8 and this

event is *independent* from an event "edge DF is present", the probability of which is equal to 0.2. In Figure 20 node F represents a choice created to resolve some record of entity represented by node G and where nodes B and D are options of this choice. That is, the record of the node G can correspond to only one: either B with probability 0.8 and D with probability of 0.2. Notice events "edge BF is present" and "edge DF is present" are strongly *dependent*: if one is present the other one must be absent due to the semantics of the choice node.

Weight-based formulae (WF). The weight based formulae used so far for computing the connection strength of a path would produce identical results for path $\mathcal{P}$ in both Figures 19 and 20. Because of that one can argue these examples demonstrate that WF does not consider the semantics of a choice node properly in all cases: WF consider only one type of dependence – when a choice node belongs to the path being considered. The connection strength of path $\mathcal{P}$, according to WF, is the product of the path's link coefficients, see Section 3.3.1: $w_{A,E} = C_{AB} \cdot C_{BC} \cdot C_{CD} \cdot C_{DE}$. Thus $w_{A,E} = 1 \cdot \frac{1}{1+0.8} \cdot 1 \cdot \frac{1}{1+0.2} = \frac{25}{54} \approx 0.463$.

Probabilistic formulae (PF). Let us first reintroduce part of the notation useful in the context of the PM, see Table V. We will use $\mathrm{P}_{\exists}(x)$ to denote the probability that some entity x exists, and for an edge E we will use $\mathrm{P}_{\rightarrow}(E)$ to denote the probability to follow E (in the context of a specific path). PM follows the same principle for computing the connection strength as WF: the connection strength of a path is closely related to the probability to reach the destination from the source by following this path. In PM computing connection strength becomes a two step process. First of all, path $\mathcal{P}$ should exist in the first place, which means each of its edges should exist. Thus the first step is to compute the probability $\mathrm{P}_{\exists}(\mathcal{P})$ that path $\mathcal{P}$ exists. Probability $\mathrm{P}_{\exists}(\mathcal{P})$ is equal to $\mathrm{P}(\exists AB \cap \exists BC \cap \exists CD \cap \exists DE)$. If existence of each edge is independent from existence of the other edges, e.g. like for the case shown in Figure 20, then $\mathrm{P}(\exists AB \cap \exists BC \cap \exists CD \cap \exists DE) = \mathrm{P}_{\exists}(AB) \cdot \mathrm{P}_{\exists}(BC) \cdot \mathrm{P}_{\exists}(CD) \cdot \mathrm{P}_{\exists}(DE)$. Since all of the edges of path $\mathcal{P}$ are labeled with 1's in both figures, the probability that $\mathcal{P}$ exists is 1. Now the second step is to consider probability $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$ to follow path $\mathcal{P}$, given that $\mathcal{P}$ exists. Once this probability is computed, it is easy to compute our goal – the probability to follow path $\mathcal{P}$, which we use as our measure of the connection strength of path $\mathcal{P}$: $\mathrm{P}_{\rightarrow}(\mathcal{P}) = \mathrm{P}_{\exists}(\mathcal{P}) \cdot \mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$. Probability $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$ is different for the cases of Figures 19 and 20. Thus, for those figures, the connection strength of path $\mathcal{P}$ computed using PF is different, whereas it is the same if WF is used.

Case 1: Independent edge existence. In Figure 19 two events "BF exists" and "DG exists" are independent. The probability to follow path $\mathcal{P}$ is the product of probabilities to follow each of its edges (under the assumption of independence of the corresponding events): $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P}) = \mathrm{P}_{\rightarrow}(AB|\exists\mathcal{P}) \cdot \mathrm{P}_{\rightarrow}(BC|\exists\mathcal{P}) \cdot \mathrm{P}_{\rightarrow}(CD|\exists\mathcal{P}) \cdot \mathrm{P}_{\rightarrow}(DE|\exists\mathcal{P})$. Given path $\mathcal{P}$ exists, the probability to follow edge AB in path $\mathcal{P}$ is one. The probability to follow edge BC is computed as follows. With probability 0.2 edge BF is absent and then the probability to follow BC is 1, with probability 0.8 edge BF is present and the probability to follow BC (given there are two links, BF and BC, that can be followed) is $\frac{1}{2}$. Thus the total probability is $0.2 \cdot 1 + 0.8 \cdot \frac{1}{2} = 0.6$. Similarly, the probability to follow CD is 1 and the probability to follow DE is $0.8 \cdot 1 + 0.2 \cdot \frac{1}{2} = 0.9$. The probability to follow path $\mathcal{P}$, given it exists, is the product

of probabilities to follow each edge of the path which is equal to $0.6 \cdot 0.9 = 0.54$. Since $\mathcal{P}$ in our example exists with probability 1, the final probability to follow this path is 0.54.

Case 2: Dependent edge existence. For the case shown in Figure 20 both BF and DF edges cannot be present at the same time. To compute $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$ we will consider two cases separately: $\exists BF$ and $\nexists BF$ and compute $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$ as $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P}) = \mathrm{P}_{\exists}(BF|\exists\mathcal{P}) \cdot \mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P} \cap \exists BF) + \mathrm{P}_{\nexists}(BF|\exists\mathcal{P}) \cdot \mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P} \cap \nexists BF)$.

Let us first assume $\exists BF$ and then compute $\mathrm{P}_{\exists}(BF|\exists\mathcal{P}) \cdot \mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P} \cap \exists BF)$. For the case of Figure 20, if no assumptions about the presence or absence of DF have been made yet, $\mathrm{P}_{\exists}(BF|\exists\mathcal{P})$ is simply equal to $\mathrm{P}_{\exists}(BF)$ which is equal 0.8. If BF is present then DF is absent and the probability to follow $\mathcal{P}$ is $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P} \cap \exists BF) = 1 \cdot \frac{1}{2} \cdot 1 \cdot 1 = \frac{1}{2}$. Now let us consider the second case $\nexists BF$ (and thus $\exists DF$). Probability $\mathrm{P}_{\nexists}(BF|\exists\mathcal{P})$ is 0.2. For that case $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P} \cap \nexists BF)$ is equal to $1 \cdot 1 \cdot 1 \cdot \frac{1}{2} = \frac{1}{2}$. Thus $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P}) = 0.8 \cdot \frac{1}{2} + 0.2 \cdot \frac{1}{2} = 0.5$.

B.2 Independent edge existence: general case.

Let us now see how to compute path connection strength in the context of PM in the general case assuming existence of each edge under consideration is independent from existence of the other edges under consideration.

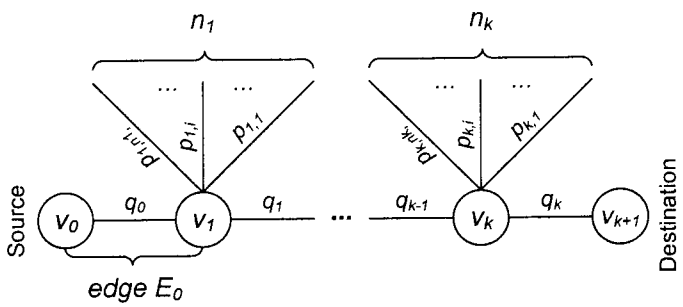

Fig. 21. Independent edge existence: general case.

Any non-trivial[11] path in general can be represented as shown in Figure 21. Path $\mathcal{P}$ can be viewed as a sequence of nodes $v_0, \ldots, v_{k+1}$ or as a sequence of edges $E_0, \ldots, E_k$, where $E_i = (v_i, v_{i+1})$, $\mathrm{P}_{\exists}(E_i) = q_i$.

Recall from Section B.1, our goal is to compute the probability to follow path $\mathcal{P}$, which we use as our measure of the connection strength of path $\mathcal{P}$:

$$\mathrm{P}_{\rightarrow}(\mathcal{P}) = \mathrm{P}_{\exists}(\mathcal{P}) \cdot \mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P}) \tag{2}$$

[11] A trivial path is a path of length one, i.e. the source and destination are connected via a single edge labeled with probability p. The connection strength of such a path, computed as the probability to reach the destination node from the source node via the path, is p. A path containing at least one intermediate node (i.e. of length at least 2) is called non-trivial.

The probability that $\mathcal{P}$ exists is equivalent to the probability each of its edges exists:

$$\mathrm{P}_{\exists}(\mathcal{P}) = \mathrm{P}(\bigcap_{i=0}^{k} \exists E_i) \tag{3}$$

Given our assumption of the independence, $\mathrm{P}_{\exists}(\mathcal{P})$ can be computed as:

$$\mathrm{P}_{\exists}(\mathcal{P}) = \prod_{i=0}^{k} \mathrm{P}_{\exists}(E_i) = \prod_{i=0}^{k} q_i \tag{4}$$

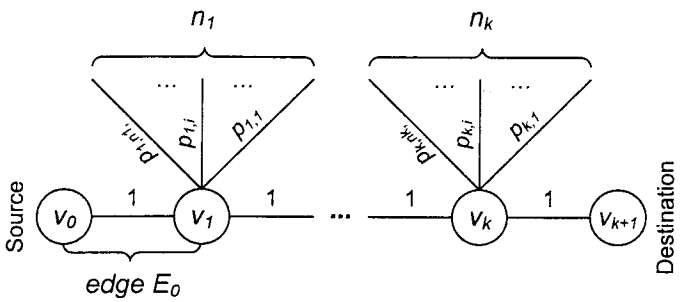

Fig. 22. Independent edge existence: general case. Assuming $\exists\mathcal{P}$.

Now we need to compute $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$. Since we assume $\exists\mathcal{P}$, we assume each edge of $\mathcal{P}$ exists and thus we are now interested in the case shown in Figure 22. To compute $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$ let us refer to Table V again. Let us define for each edge $E_{i,j}$ labeled $p_{i,j}$ a variable $a_{i,j} \in \{0,1\}$ where $a_{i,j} = 1$ if $\exists E_{i,j}$ and $a_{i,j} = 0$ if $\nexists E_{i,j}$. Let $\mathbf{a}$ denote the vector of all $a_{i,j}$ under consideration for path $\mathcal{P}$: $\mathbf{a} = (a_{1,1}, \ldots, a_{k,n_k})$. Let $\mathcal{A}$ denote all possible instantiations of $\mathbf{a}$, i.e. $|\mathcal{A}| = 2^{n_1+\cdots+n_k}$.

Probability $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$ can be computed as

$$\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P}) = \sum_{\mathbf{a}\in\mathcal{A}} \mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P} \cap \mathbf{a}) \cdot \mathrm{P}(\exists\mathcal{P} \cap \mathbf{a}) \tag{5}$$

where $\mathrm{P}(\exists\mathcal{P} \cap \mathbf{a})$ is the probability of instantiation $\mathbf{a}$ to occur while assuming $\exists\mathcal{P}$. Given our assumption of independence of probabilities $\mathrm{P}(\exists\mathcal{P} \cap \mathbf{a}) = \mathrm{P}(\mathbf{a})$. Probability $\mathrm{P}(\mathbf{a})$ can be computed as

$$\mathrm{P}(\mathbf{a}) = \prod_{i,j} p_{i,j}^{a_{i,j}} \cdot (1 - p_{i,j})^{1-a_{i,j}}. \tag{6}$$

Probability $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P} \cap \mathbf{a})$, which is the probability to go via $\mathcal{P}$ given a particular instantiation of $\mathbf{a}$ and $\mathcal{P}$ exists, can be computed as

$$\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P} \cap \mathbf{a}) = \prod_{i=1}^{k} \frac{1}{1 + \sum_{j=1}^{n_i} a_{i,j}}. \tag{7}$$

So the formula for computing $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$ is

$$\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P}) = \sum_{\mathbf{a}\in\mathcal{A}}\prod_{i=1}^{k}\frac{1}{1+\sum_{j=1}^{n_i}a_{i,j}}\cdot\prod_{i,j}p_{i,j}^{a_{i,j}}\cdot(1-p_{i,j})^{1-a_{i,j}} \tag{8}$$

and

$$\mathrm{P}_{\rightarrow}(\mathcal{P}) = \left(\prod_{i=0}^{k}q_i\right)\cdot\left(\sum_{\mathbf{a}\in\mathcal{A}}\prod_{i=1}^{k}\frac{1}{1+\sum_{j=1}^{n_i}a_{i,j}}\cdot\prod_{i,j}p_{i,j}^{a_{i,j}}\cdot(1-p_{i,j})^{1-a_{i,j}}\right) \tag{9}$$

Computing path connection strength in practice. Notice, Equation 9 iterates through all possible instantiations of $\mathbf{a}$ which is impossible to compute in practice given $|\mathcal{A}| = 2^{n_1+\cdots+n_k}$. However Equation 9 can be simplified to make computations feasible.

To achieve the simplification, we will use our assumption of independence of probabilities which allows us to compute $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P})$ as $\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P}) = \prod_{i=0}^{k}\mathrm{P}_{\rightarrow}(E_i|\exists\mathcal{P})$. In our model $\mathrm{P}_{\rightarrow}(E_0|\exists\mathcal{P})$ is always one, thus we need to specify how to compute $\mathrm{P}_{\rightarrow}(E_i|\exists\mathcal{P})$ for i greater than zero.

Let $\mathbf{a_i}$ denote vector $(a_{i,1},\ldots,a_{i,n_i})$, i.e. $\mathbf{a} = (\mathbf{a_1},\ldots,\mathbf{a_k})$. Let $\mathcal{A}_i$ denote all possible instantiations of $\mathbf{a_i}$, i.e. $\mathcal{A} = \mathcal{A}_1\times\ldots\times\mathcal{A}_k$ and $|\mathcal{A}_i| = 2^{n_i}$.

$$\mathrm{P}_{\rightarrow}(E_i|\exists\mathcal{P}) = \sum_{\mathbf{a_i}\in\mathcal{A}_i}\frac{1}{1+\sum_{j=1}^{n_i}a_{i,j}}\cdot\prod_{j=1}^{n_i}p_{i,j}^{a_{i,j}}\cdot(1-p_{i,j})^{1-a_{i,j}} \tag{10}$$

and

$$\mathrm{P}_{\rightarrow}(\mathcal{P}|\exists\mathcal{P}) = \prod_{i=1}^{k}\left(\sum_{\mathbf{a_i}\in\mathcal{A}_i}\frac{1}{1+\sum_{j=1}^{n_i}a_{i,j}}\cdot\prod_{j=1}^{n_i}p_{i,j}^{a_{i,j}}\cdot(1-p_{i,j})^{1-a_{i,j}}\right) \tag{11}$$

and

$$\mathrm{P}_{\rightarrow}(\mathcal{P}) = \left(\prod_{i=0}^{k}q_i\right)\cdot\prod_{i=1}^{k}\left(\sum_{\mathbf{a_i}\in\mathcal{A}_i}\frac{1}{1+\sum_{j=1}^{n_i}a_{i,j}}\cdot\prod_{j=1}^{n_i}p_{i,j}^{a_{i,j}}\cdot(1-p_{i,j})^{1-a_{i,j}}\right) \tag{12}$$

The effect of transformation. Notice, using Equation 9 the algorithm will need to perform $|\mathcal{A}| = 2^{n_1+\cdots+n_k}$ iterations (one per each instantiation of $\mathbf{a}$), whereas using Equation 10 the algorithm will need to perform $|\mathcal{A}_1|+\cdots+|\mathcal{A}_k| = 2^{n_1}+\cdots+2^{n_k}$ simpler iterations, which is a significant improvement. It is interesting to note here, that in WF $\mathrm{P}_{\rightarrow}(E_i)$ *corresponds to the link coefficient* of link E_i which is computed using formula which is computationally even less expensive: $\frac{q_{i+1}}{(\sum_{j=1}^{n_i}p_{i,j}+q_i+q_{i+1})-q_i}$.

Handling edges which have probability 1. The formula in Equation 10 assumes 2^{n_i} iterations will be needed to compute $\mathrm{P}_{\rightarrow}(E_i|\exists\mathcal{P})$, thus the cost can still be quite high. This formula can be simplified further for much more efficient computation given that in practice often some of the $p_{i,j}$'s or even all of them are 1's. Figure 23 shows the case where m, where $m\in[0,n_i]$, edges incident to node v_i are labeled with 1. Let $\mathbf{a}'_\mathbf{i}$ denote vector $(a_{i,1},\ldots,a_{i,n_i-m})$ and let $\mathcal{A}'_i$ be all possible

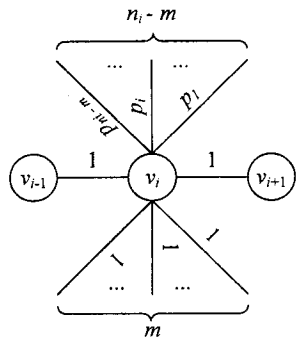

Fig. 23. Link coefficient of link $E_i = (v_i, v_{i+1})$ in PM.

instantiations of this vector. Then Equation 10 simplifies to:

$$\mathrm{P}_{\rightarrow}(E_i|\exists\mathcal{P}) = \sum_{\mathbf{a}'_i \in \mathcal{A}'_i} \frac{1}{1+m+\sum_{j=1}^{n_i-m} a_{i,j}} \cdot \prod_{j=1}^{n_i-m} p_{i,j}^{a_{i,j}} \cdot (1-p_{i,j})^{1-a_{i,j}} \tag{13}$$

The number of iteration is reduced from 2^{n_i} to 2^{n_i-m}.

Poisson trials. Performing 2^{n_i-m} iterations can still be expensive for the cases when $n_i - m$ is large. In this subsection we will present several solutions to deal with this issue, none of those solutions is a very good one. The following discussion shows several directions where one should investigate further if you are interested in using PM for your problem. We will conclude our discussion with the method we have actually used to compute $\mathrm{P}_{\rightarrow}(E_i|\exists\mathcal{P})$ when 2^{n_i-m} is large in our implementation of PM.

Method 1: Do not simplify further. Even though 2^{n_i-m} cost can be expensive, it is also true that for a particular data cleaning problem either (a)2^{n_i-m} is never expensive or (b) 2^{n_i-m} can be large but bearable and cases when it is large are infrequent. In those cases further simplification might not be required.[12]

Method 2: Estimate answer using facts from Poisson trials theory. Let us denote the following sum as s_i: $s_i = \sum_{j=1}^{n_i} a_{i,j}$. The binomial distribution gives the number of successes in n independent trials where each trial is successful with the same probability p. The binomial distribution can be viewed as a sum of several *i.i.d.* Bernoulli trials. The *Poisson trails* process is similar to the binomial distribution process where trials are still independent but not necessarily identically distributed, i.e. the probability of success in i^{th} trial is p_i.

We can modify Equation 12 to compute $\mathrm{P}_{\rightarrow}(E_i|\exists\mathcal{P})$ as follows:

$$\mathrm{P}_{\rightarrow}(E_i|\exists\mathcal{P}) = \sum_{j=0}^{k} \frac{1}{1+j} \cdot \mathrm{P}(s_i = j) \tag{14}$$

Notice, for given i we can treat $a_{i,1}, a_{i,2}, \ldots, a_{i,n_i}$ as a sequence of n_i Poisson trials with probabilities of success $\mathrm{P}(a_{i,j} = 1) = p_{i,j}$. Thus we can apply methods

[12] In our experiments cases when 2^{n_i-m} was large were infrequent, but unfortunately in those case the cost of performing 2^{n_i-m} iterations was not acceptable.

from Poisson trials theory to *estimate* $\mathrm{P}(s_i = j)$ *quickly*, rather than compute it exactly via iterations over the corresponding cases, i.e. one straightforward (and exact) iterative method is:

$$\mathrm{P}(s_i = l) = \sum_{\substack{\mathbf{a}_i \in \mathcal{A}_i \\ s_i = l}} \prod_{j=1}^{n_i} p_{i,j}^{a_{i,j}} \cdot (1 - p_{i,j})^{1 - a_{i,j}}$$

For example, if all $p_{i,j} = p$ for all $j \in [1, n_1]$ then we have a binomial distribution case and can compute $\mathrm{P}(s_i = l)$ quickly, and in this case exactly, as $C_{n_i}^{l} p^l (1-p)^{n_i - l}$.

Some work on Poisson trials mentions that Uspensky has studied estimation of $\mathrm{P}(s_i = l)$ for general case when all p_i's can be different, but we could not locate any relevant publication authored by this author. We have been able to locate a few html sources which sketch those estimations. Those sources had similar content with several variables in formulae left unexplained and no measures of goodness of the estimation has been provided.

Potentially, more famous formulae from Poisson trials theory, known as the Chernoff Bounds, can be used to estimate $\mathrm{P}(s_i = l)$. The Chernoff Bounds are defined for random variable $X = \sum_{i=1}^{n} X_i$ where X_i's are independently distributed random variables such that $\mathrm{P}(X_i = 1) = p_i$ and $\mathrm{P}(X_i = 0) = 1 - p_i$. The expected value μ for X is $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} p_i$. The bounds are as follows:

$$\mathrm{P}(X \geq (1+\delta)\mu) < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}} \right)^{\mu}, \text{ for } \forall \delta > 0 \tag{15}$$

$$\mathrm{P}(X \leq (1-\delta)\mu) \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}} \right)^{\mu}, \text{ for } \delta \in (0,1) \tag{16}$$

Those bounds can be used, for instance, if it turns out that using those bounds one can obtain a good estimations of $\mathrm{P}(s_i = l)$ by, for example, presenting

$$\begin{aligned} \mathrm{P}(s_i = l) &= \mathrm{P}(l - 0.5 < s_i < l + 0.5) \\ &= \mathrm{P}(s_i < l + 0.5) - \mathrm{P}(s_i \leq l - 0.5) \end{aligned} \tag{17}$$

Let $\mu_i = \mathbf{E}[s_i] = \sum_{j=1}^{n_i} p_{i,j}$. Assume without loss of generality that $l + 0.5 < \mu_i$. Let δ_1 and δ_2 be such that $(1 - \delta_1)\mu_i = l - 0.5$ and $(1 - \delta_2)\mu_i = l + 0.5$. According to Equation 16,

$$\mathrm{P}(s_i \leq (1-\delta_1)\mu_i) = \mathrm{P}(s_i \leq l - 0.5) = \left(\frac{e^{-\delta_1}}{(1-\delta_1)^{1-\delta_1}} \right)^{\mu_i} - \varepsilon_1$$

$$\mathrm{P}(s_i \leq (1-\delta_2)\mu_i) = \mathrm{P}(s_i \leq l + 0.5) = \left(\frac{e^{-\delta_2}}{(1-\delta_2)^{1-\delta_2}} \right)^{\mu_i} - \varepsilon_2$$

where $\varepsilon_1, \varepsilon_2 > 0$ represent the exact errors in the upper bounds (the values of ε_1 and ε_2 are not known to us). Thus

$$\begin{aligned} \mathrm{P}(s_i = l) &= \mathrm{P}(l - 0.5 < s_i < l + 0.5) \\ &= [\left(\frac{e^{-\delta_2}}{(1-\delta_2)^{1-\delta_2}}\right)^{\mu_i} - \varepsilon_2] - [\left(\frac{e^{-\delta_1}}{(1-\delta_1)^{1-\delta_1}}\right)^{\mu_i} - \varepsilon_1] \\ &= \underbrace{\left[\left(\frac{e^{-\delta_2}}{(1-\delta_2)^{1-\delta_2}}\right)^{\mu_i} - \left(\frac{e^{-\delta_1}}{(1-\delta_1)^{1-\delta_1}}\right)^{\mu_i}\right]}_{\text{result}} + \underbrace{[\varepsilon_1 - \varepsilon_2]}_{\text{error}} \end{aligned} \tag{18}$$

Thus Chernoff bounds might be useful if (a) one can show that the error is much smaller relatively to the result or (b) if one can estimate $\varepsilon_1 - \varepsilon_2$ or (c) if one can show that if the error is ignored the final result does not deviate significantly from the exact result.

Method 3: Use linear cost formula. In our implementation of PM we have used the following approach for computing the probabilistic link coefficient for edge E_i. We have a cut-off threshold to decide if 2^{n_i-m} iterations is acceptable. If we decide the number of iterations is acceptable we compute the coefficient precisely, using iterations. If the number of iterations exceeds the threshold, we use the following linear cost formula, applying which is *not correct* in general but which has some degree of support behind it. We compute the expected number of edges μ_i among n_i edges $E_{i,1}, E_{i,2}, \ldots, E_{i,n_i}$, where $\mathrm{P}(\exists E_{i,j}) = p_{i,j}$, as follows: $\mu_i = m + \sum_{j=1}^{n_i-m} p_{i,j}$. Then we say since there are $1+\mu_i$ possible links to follow on average, the probability to follow E_i can be estimated as:

$$\mathrm{P}_{\rightarrow}(E_i|\exists\mathcal{P}) \approx \frac{1}{1+\mu_i} = \frac{1}{1+m+\sum_{j=1}^{n_i-m} p_{i,j}} \tag{19}$$

Thus in most of the cases when 2^{n_i-m} is small we use the exact formula, and in rare cases when the number of iterations 2^{n_i-m} is too large we utilize the approximate formula of Equation 19.

B.3 Dependence in edge existence: general case.

In Section B.2 we have assumed that each event, corresponding to the existence of each edge under consideration, is independent from the other events, corresponding to the existence of the other edges under consideration. This assumption is reasonable for the cases when events are truly independent or when they are very weakly correlated so that it is reasonable to make a simplifying assumption that they are independent.

However the existence of one edge can strongly depend on the existence of another edge, as shown in Section B.1 in Figure 20. In this section we discuss how to address certain types of dependence. In particular we concentrate on how to deal with the cases like the one shown in Figure 20 since that is the case, that we are aware of, when the events are correlated the most.

As in Section B.2, we will need to compute $\mathrm{P}_{\rightarrow}(\mathcal{P})$ which we will use as our measure of the connection strength of $\mathcal{P}$. But now we will also consider cases with dependence, such as in Figure 20.

Equations 2 and 3 still hold, but Equation 4 might not be true in general. Since

we concentrate only on cases like in Figure 20, we will assume Equation 4 holds, but we will now show one simple method how to deal with the situation when Equation 4 does not hold. To compute $\mathrm{P}_{\exists}(\mathcal{P})$ one needs to use Equation 3 which will require computation of $\mathrm{P}(\bigcap_{i=0}^{k} \exists E_i)$. A simple observation can help to compute this probability: sometimes one easy way to compute $\mathrm{P}(\bigcap_{i=0}^{k} \exists E_i)$ would be to represent it as a product of the following probabilities:

$$\begin{aligned} \mathrm{P}_{\exists}(\mathcal{P}) &= \mathrm{P}(\bigcap_{i=0}^{k} \exists E_i) \\ &= \mathrm{P}_{\exists}(E_0) \cdot \mathrm{P}_{\exists}(E_1 | \exists E_0) \cdot \mathrm{P}_{\exists}(E_2 | \exists E_0 \cap \exists E_1) \times \cdots \times \mathrm{P}_{\exists}(E_k | \bigcap_{i=0}^{k-1} \exists E_i) \end{aligned} \tag{20}$$

Our goal is to come up with a formula like Equation 12 but when existence of certain edges is dependent. Let $\mathbf{f}$ ("f" for "free") be a set of all $a_{i,j}$'s such that events $\exists E_{i,j}$ are independent from each other. If we treat $\mathbf{a}$ as a set, we can compute set $\mathbf{d}$ ("d" for "dependent") of $a_{i,j}$'s for which events $\exists E_{i,j}$ are dependent as follows $\mathbf{d} = \mathbf{a} - \mathbf{f}$, i.e. $\mathbf{a} = \mathbf{f} \cup \mathbf{d}$, $\mathbf{f} \cap \mathbf{d} = \emptyset$. Similarly to $\mathbf{a_i}$ we can define $\mathbf{d_i}$ as $\mathbf{d_i} = \{a_{i,j} : a_{i,j} \in \mathbf{d}, j = 1, \ldots, n_i\}$ and $\mathbf{f_i}$ as $\mathbf{f_i} = \{a_{i,j} : a_{i,j} \notin \mathbf{d}, j = 1, \ldots, n_i\}$, i.e., $\mathbf{a_i} = \mathbf{f_i} \cup \mathbf{d_i}$ and $\mathbf{f_i} \cap \mathbf{d_i} = \emptyset$.

$$\begin{aligned} \mathrm{P}_{\rightarrow}(\mathcal{P}) = &\left(\prod_{i=0}^{k} q_i \right) \times \\ &\sum_{\mathbf{d} \in \mathcal{D}} \left(\left[\prod_{i=1}^{k} \left(\sum_{\mathbf{a_i} \in \mathcal{A}_i} \frac{1}{1 + \sum_{j=1}^{n_i} a_{i,j}} \cdot \prod_{j: a_{i,j} \in \mathbf{f_i}} p_{i,j}^{a_{i,j}} \cdot (1 - p_{i,j})^{1 - a_{i,j}} \right) \right] \mathrm{P}(\mathbf{d}) \right) \end{aligned} \tag{21}$$

Equation 21 iterates over all feasible instantiations of $\mathbf{d}$ and $\mathrm{P}(\mathbf{d})$ is the probability of a specific instance.

B.3.1 *Intra choice nodes dependence.* Equation 21 is generic in the sense that it is applicable to any kinds of dependence among edges for which $a_{i,j} \in \mathbf{d}$: one just need to be able to compute $\mathrm{P}(\mathbf{d})$ for each feasible instance of $\mathbf{d}$. Now we will consider one specific type of dependence, and for that type we will specify how to compute $\mathrm{P}(\mathbf{d})$.

Specifically, we will consider only those cases where $a_{i,j}$ is in $\mathbf{d}$ only because there is (at least one) another $a_{i',j'} \in \mathbf{d}$ such that events $\exists E_{i,j}$ and $\exists E_{i',j'}$ are dependent and both edges $E_{i,j}$ and $E_{i',j'}$ are options of the same choice node. Figure 20 is an example of such a case.

Let us formalize this case. Notice, for any $a_{i,j}$ such that $a_{i,j} \in \mathbf{d}$ there is at least one $a_{i',j'}$ such that (i) $a_{i',j'} \in \mathbf{d}$ and (ii) $\exists E_{i,j}$ and $\exists E_{i',j'}$ are dependent. Let $S(a_{i,j})$ denote a set which includes $a_{i,j}$ and *all* $a_{i',j'}$'s in $\mathbf{d}$ such that $\exists E_{i,j}$ and $\exists E_{i',j'}$ are dependent. Notice, for the cases being considered the following holds (a) $\forall a \in S(a_{i,j}) \Rightarrow S(a) \equiv S(a_{i,j})$ and (b) all edges $\{E_{i',j'} : a_{i',j'} \in S(a_{i,j})\}$ are options of the same single choice node.

In such a model we can represent set $\mathbf{d}$ as $\mathbf{d} = C_1 \cup \cdots \cup C_m$ such that
- $\forall a \in C_i \Rightarrow C_i \equiv S(a)$
- $\forall a \in C_i \Rightarrow a$ corresponds to an option of the same choice node c_i

- $C_i \neq \emptyset, i = 1, \ldots, m$
- $C_i \cap C_j = \emptyset, i \neq j; i, j = 1, \ldots, m.$

Since we consider only intra choice dependence and assume there is no inter choice dependence, then for each instantiation of $\mathbf{d}$ in our model $\mathrm{P}(\mathbf{d}) = \mathrm{P}(C_1) \times \cdots \times \mathrm{P}(C_m)$. Now let us specify how to compute $P(C_l)$ in this formula.

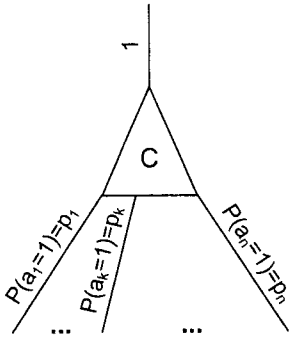

Fig. 24. Intra choice dependence.

Figure 24 shows a choice node C with n options labeled with probabilities $p_1, \ldots, p_n$. Assume C is one of those c_i's mentioned above. As before, to specify which edge is present and which is absent, each option has variable a_i associated with it: a_i is 1 if the corresponding edge is present and a_i is 0 of it is absent, i.e. $\mathrm{P}(a_i = 1) = p_i$, $\sum_{i=1}^{n} p_i = 1$. Let us assume, without loss of generality, that k first a_i's belong to $\mathbf{d}$, i.e. $a_i \in \mathbf{d}$ for $i \in [1, k]$ and $a_i \notin \mathbf{d}$ for $i \in [k+1, n]$.

Thus our goal is to be able to compute the probability of a particular instantiation of $\vec{a} = (a_1, \ldots, a_k)$. Notice, only one of $a_1, \ldots, a_n$ can be 1. First let us compute the probability of instantiation $\vec{a} = \vec{0} = (0, \ldots, 0)$. Assume $[0, 1]$ interval is divided into two intervals: $[0, \sum_{i=1}^{k} p_i)$ and $[\sum_{i=1}^{k} p_i, 1]$. Probability $\mathrm{P}(\vec{a} = \vec{0})$ is equal to the probability that a random number, generated according to $\mathrm{U}[0, 1]$ distribution, would fall into the second interval, thus $\mathrm{P}(\vec{a} = \vec{0}) = \sum_{i=k+1}^{n} p_n$. The other case that is left for consideration is when one $a_l = 1$, where $l \in [1, k]$, in which case $\mathrm{P}(a_l = 1) = p_l$. To summarize:

$$\mathrm{P}(\vec{a}) = \begin{cases} \sum_{i=k+1}^{n} p_n, \text{ if } a_i = 0, \text{ where } i = 1, \ldots, k \\ p_l, \text{ if } a_l = 1, a_i = 0, \text{ where } l \in [1, k], i = 1, \ldots, l-1, l+1, \ldots, k. \end{cases}$$

B.4 Computing the total connection strength.

B.4.1 *Summation operation in computation of connection strength.* Similarly to WF, in PF the connection strength between nodes n_1 and n_2 is computed as a sum of connection strengths of all simple paths between n_1 and n_2. For PM, instead of computing the (weight-based model's) normalized weight of option $option_{ij}$ (i.e., $\overline{w}_{ij}$), we compute the probability p_{ij} that $option_{ij}$ exists.

Let us motivate why the *summation* of individual simple paths is performed. We associate the connection strength between two nodes n_1 and n_2 with probability of

reaching n_2 from n_1 where you are allowed to follow only simple paths of length no greater than a parameter L. All those simple paths, lets name those $\mathcal{P}_1,\ldots,\mathcal{P}_k$, will be returned by AllSimplePaths algorithm. Lets call $\mathscr{P}$ the subgraph graph of union of those paths: $\mathscr{P} = \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_k$. Graph $\mathscr{P}$ is a subgraph of the complete graph $G = (V, E)$, where V is the set of vertices $V = \{v_i : i = 1,\ldots,|V|\}$ and E is the set of edges $E = \{E_i : i = 1,\ldots,|E|\}$. Let us define a_i as follows: if $\exists E_i$ then $a_i = 1$ else $a_i = 0$, and let $\mathbf{a}$ denote vector $(a_1,\ldots,a_{|E|})$ and $\mathcal{A}$ is all possible instantiations of $\mathbf{a}$.

We need to compute $\mathrm{P}_{\rightarrow}(\mathscr{P})$ which we treat as the measure of the connection strength. We can represent $\mathrm{P}_{\rightarrow}(\mathscr{P})$ as:

$$\mathrm{P}_{\rightarrow}(\mathscr{P}) = \sum_{\mathbf{a}\in\mathcal{A}} \mathrm{P}_{\rightarrow}(\mathscr{P}|\mathbf{a}) \cdot \mathrm{P}(\mathbf{a}) \tag{22}$$

Notice, when computing $\mathrm{P}_{\rightarrow}(\mathscr{P}|\mathbf{a})$ the complete knowledge of which nodes exist and which do not is available, as if all the edges are "fixed". That is, assuming one particular instantiation of $\mathbf{a}$, there is no dependence for node existences and each edge is either present with 100% probability or absent with 100% probability. Thus

$$\mathrm{P}_{\rightarrow}(\mathscr{P}|\mathbf{a}) = \mathrm{P}_{\rightarrow}(\mathcal{P}_1 \cup \cdots \cup \mathcal{P}_k|\mathbf{a}) = \sum_{i=1}^{k} \mathrm{P}_{\rightarrow}(\mathcal{P}_k|\mathbf{a}) \tag{23}$$

and

$$\begin{aligned}
\mathrm{P}_{\rightarrow}(\mathscr{P}) &= \sum_{\mathbf{a}\in\mathcal{A}} \mathrm{P}_{\rightarrow}(\mathscr{P}|\mathbf{a}) \cdot \mathrm{P}(\mathbf{a}) \\
&= \sum_{\mathbf{a}\in\mathcal{A}} \left[\left(\sum_{i=1}^{k} \mathrm{P}_{\rightarrow}(\mathcal{P}_k|\mathbf{a}) \right) \cdot \mathrm{P}(\mathbf{a}) \right] \\
&= \sum_{i=1}^{k} \left[\sum_{\mathbf{a}\in\mathcal{A}} \Big(\mathrm{P}_{\rightarrow}(\mathcal{P}_k|\mathbf{a}) \cdot \mathrm{P}(\mathbf{a}) \Big) \right] \\
&= \sum_{i=1}^{k} \mathrm{P}_{\rightarrow}(\mathcal{P}_k)
\end{aligned} \tag{24}$$

Thus Equation 24 explains why the connection strength is the sum of the connection strength of all simple paths.

It is important to note a subtle difference between the semantics of "following a path" in Equation 24 and that in other equations that we have considered before, e.g. such as Equation 21. The issue arise because in formulae we have considered before the term that corresponds to probabilistic link coefficient for the first link in the path is computed differently from the rest of the link coefficients, where as Equation 24 it is assumed to be computed the same way as for the rest of the links. In other words, for a particular path in previous equations we have assumed that from the source node in the path the algorithm, when computing the probability to reach the destination, cannot travel anywhere but the second node in the path; i.e. it cannot deviate via other links. However it is not so for Equation 24 where formula implies that a deviation from the very first link is possible.

A question might arise of why it has been decided to compute the first link coefficient differently from the others. The answer to that is the following. Assume we are trying to decide whether record R in some object P corresponds to object A_1 or A_2. Assume there 20 simple path from A_1 to P and there are 10 simple path from A_2 to P, where all paths have the same connection strength. Our current formulae are such that the probability that A_1 is the correct match will be greater than that of A_2. Now assume node A_1 has very large number of adjacent edges whereas A_2 has very few. If one decides to consider the probability of reaching P from A_1 and, unlike us, will consider possibility to deviate from the path for the very first link, then the probability of reaching P from A_2 might turn out to be (much) greater than the probability of reaching A_1. So if say one author, A_1, has written many papers whereas another one, A_2, has a few then A_1 will be unfairly punished for large number of papers and not be correctly identified as the correct match even though the evidence might indicate otherwise.

B.4.2 *Models for computing option probability.* We can use two models for computing p_{ij}'s. In both models one of the restrictions is that $\sum_j p_{ij} = 1$. Let c_{ij} denote the connection strength corresponding to $option_{ij}$. In the first *loose* model we can specify that if $c_{ij_1} > c_{ij_2}$ then $p_{ij_1} > p_{ij_2}$ must also be true and vice versa. In the second model we can specify more precisely that probabilities and the corresponding connection strengths are proportional: $p_{ij_1} \cdot c_{ij_2} = p_{ij_2} \cdot c_{ij_1}$.[13]

B.5 Experimental results

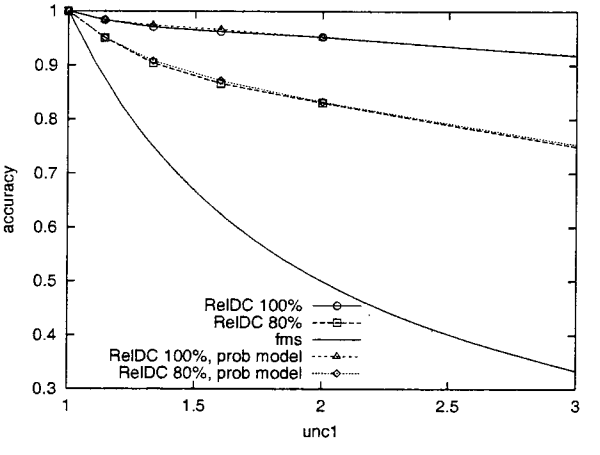

Fig. 25. Probabilistic model

Figure 25 is similar to Figure 15(a) except for it shows the result for both WF and PF. The results are similar: PF's results are marginally better that WF's results for the experiment shown in Figure 25. Such an outcome is not surprising as explained below.

[13]The second model corresponds to normalization method 1 of WM.

To understand why the results are similar it is useful to divide the algorithm into three logical phases: the AllSimplePaths, weight computation, and weight interpretation phases. The only difference PM and WM have is the way they compute weights/probabilities, i.e., only the weight computation phase is different. After the weight/probabilities are finally computed, they are *interpreted* to identify the most likely match. For example, if the three options of a choice node have weights $\overline{w}_1 = .80$, $\overline{w}_2 = .10$, and $\overline{w}_1 = .10$ the algorithm will output that option one (i.e. the one with the highest weight of 0.80) is the most likely correct match. It will output the same if the weights are (0.78, 0.11, 0.11) or (0.6, 0.22, 0.18). Examples in Section B.1 show very similar outcome for the cases of (a) WF (i.e., 0.463); (b) PF: independent event case (i.e., 0.54), and (c) PF: dependent event case (i.e., 0.5).[14] So it is not surprising that both PM and WM, *after the interpretation*, select the same options as the correct matches.

Time efficiency. Recall that both PM and WM share the same AllSimplePaths phase after which they differ in how the weights (or probabilities) are computed (the weight computation phase). The time needed for the weight computation phase for PM is roughly twice as much as that of WF.

...

[14]Notice, if all edges under consideration in those examples are labeled with probabilities 1, the the outcome of all three formulae will be absolutely identical.