A 12-page version of this document is accepted to ***SIAM International Conference on Data Mining (SDM)* 2005**

Exploiting relationships for domain-independent data cleaning*

Dmitri V. Kalashnikov Sharad Mehrotra

Computer Science Department
University of California, Irvine

Contents

*This material is based upon work supported by the National Science Foundation under Award Numbers 0331707 and 0331690. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

Abstract

In this paper we address the problem of *reference disambiguation*. Specifically, we consider a situation where entities in the database are referred to using descriptions (e.g., a set of instantiated attributes). The objective of reference disambiguation is to identify the unique entity to which each description corresponds. The key difference between the approach we propose (called RelDC) and the traditional techniques is that RelDC analyzes not only object features but also inter-object relationships to improve the disambiguation quality. Our extensive experiments over two real data sets and also over synthetic datasets show that analysis of relationships significantly improves quality of the result.

1 Introduction

Recent surveys [4] show that more than 80% of researchers working on data mining projects spend more than 40% of their project time on cleaning and preparation of data. The data cleaning problem often arises when information from heterogeneous sources is merged to create a single database. Many distinct data cleaning challenges have been identified in the literature: dealing with missing data, handling erroneous

data, elimination of duplicate entities, and so on. In this paper we address one such challenge which we refer to as *reference disambiguation*[1].

The reference disambiguation problem arises when entities in a database contain references to other entities. If entities were referred to using unique identifiers then disambiguating those references would be straightforward. Instead, frequently, entities are represented using properties/descriptions that may not uniquely identify them leading to ambiguity. For instance, assume a database stores information about two distinct individuals 'Donald L. White' and 'Donald E. White', then they both can be referred to as 'D. White' in the database. References may also be ambiguous due to differences in the representations of the same entity and errors in data entries (e.g., 'Don White' misspelled as 'Don Whitex'). The **goal** of reference disambiguation is for each reference to correctly identify the unique entity it refers to.

The reference disambiguation problem is related to the problem of *record deduplication* or *record linkage* [23, 7, 6] that often arise when multiple tables (from different data sources) are merged to create a single table. The causes of record linkage and reference disambiguation problems are similar; viz., differences in representations of objects across different data sets, data entry errors, etc. The differences between the two can be intuitively viewed using the relational terminology as follows: while the record linkage problem consists of determining when two records are the same, reference disambiguation corresponds to ensuring that references (i.e., "foreign keys"[2]) in a database point to the correct entities.

Given the tight relationship between the two data cleaning tasks and the similarity of their causes, existing approaches to record linkage can be adapted for reference disambiguation. In particular, *feature-based similarity (FBS)* methods that analyze similarity of record attribute values (to determine whether or not two records are the same) can be used to determine if a particular reference corresponds to a given entity or not. In the reference disambiguation problem, references occur within a context and define relationships/connections between entities. For instance, 'D. White' might be used to refer to an author in the context of a particular publication. This publication might also refer to different authors, which can be linked to their affiliated organizations etc, forming chains of relationships among entities. Such knowledge can be exploited alongside attribute-based similarity resulting in improved accuracy of disambiguation.

In this paper, we propose a domain-independent data cleaning approach for reference disambiguation, referred to as Relationship-based Data Cleaning (RelDC), that systematically exploits not only features but also relationships among entities for the purpose of disambiguation. RelDC views the database as a graph of entities that are linked to each other via relationships. It first utilizes a feature based method to identify a set of candidate entities (choices) for a reference to be disambiguated. Graph theoretic techniques are then used to discover and analyze relationships that exist between the entity containing the reference and the set of candidates.

The primary contributions of this paper are: (1) developing a systematic approach to exploiting both attributes as well as relationships among entities for reference disambiguation (2) establishing that exploiting relationships can significantly improve the quality of reference disambiguation by testing the developed approach over 2 real-world data sets as well as synthetic data sets.

This paper is a truncated version of a longer technical report [24] in which we discuss optimizations, computational complexity, sample content and sample graphs for real datasets, and other issues not covered by this paper. The rest of this paper is organized as follows. Section 2 presents a motivational example. In Section 3, we precisely formulate the problem of reference disambiguation and introduce notation that will help explain the RelDC approach. Section 4 describes the RelDC approach. The empirical results of RelDC are presented in Section 6. Section 7 contains the related work, and Section 8 concludes the paper.

[1] Also referred to as *cleaning spurious links* in [26]

[2] We are using the term foreign key loosely. Usually, foreign key refers to a unique identifier of an entity in another table. Instead, foreign key above means the set of properties that serve as a reference to an entity.

2 Motivation for analyzing relationships

In this section we will use an instance of the "author matching" problem to illustrate that exploiting relationships among entities can improve the quality of reference disambiguation. We will also schematically describe one approach that analyzes relationships in a systematic domain-independent fashion. Consider a database about *authors* and *publications*. Authors are represented in the database using the

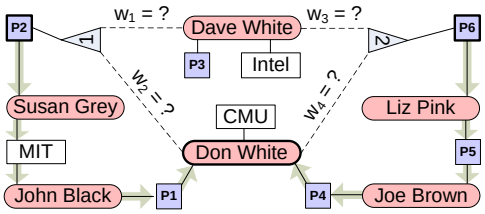

Figure 1: Graph for the publications example

attributes ⟨`id`, `authorName`, `affiliation`⟩ and information about papers is stored in the form ⟨`id`, `title`, `authorRef1`,`authorRef2`,..., `authorRefN`⟩. Consider a toy database consisting of the following *authors* and *publications* records.

1. ⟨A_1, 'Dave White', 'Intel'⟩,
2. ⟨A_2, 'Don White', 'CMU'⟩,
3. ⟨A_3, 'Susan Grey', 'MIT'⟩,
4. ⟨A_4, 'John Black', 'MIT'⟩,
5. ⟨A_5, 'Joe Brown', unknown⟩,
6. ⟨A_6, 'Liz Pink', unknown⟩.

1. ⟨P_1, 'Databases ...', 'John Black', 'Don White'⟩,
2. ⟨P_2, 'Multimedia ...', 'Sue Grey', **'D. White'**⟩,
3. ⟨P_3, 'Title3 ...', 'Dave White'⟩,
4. ⟨P_4, 'Title5 ...', 'Don White', 'Joe Brown'⟩,
5. ⟨P_5, 'Title6 ...', 'Joe Brown', 'Liz Pink'⟩,
6. ⟨P_6, 'Title7 ...', 'Liz Pink', **'D. White'**⟩.

The goal of the author matching problem is to identify for each `authorRef` in each paper the correct author it refers to.

We can use existing feature-based similarity (FBS) techniques to compare the description contained in each `authorRef` in papers with values in `authorName` attribute in authors. This would allow us to resolve almost every `authorRef` references in the above example. For instance, such methods would identify that 'Sue Grey' reference in P_2 refers to A_3 ('Susan Grey'). The only exception will be 'D. White' references in P_2 and P_6: 'D. White' could match either A_1 ('Dave White') or A_2 ('Don White').

Perhaps, we could disambiguate the reference 'D. White' in P_2 and P_6 by exploiting additional attributes. For instance, the titles of papers P_1 and P_2 might be similar while titles of P_2 and P_3 might not, suggesting that 'D. White' of P_2 is indeed 'Don White' of paper P_1. We next show that it may still be possible to disambiguate the references 'D. White' in P_2 and P_6 by analyzing relationships among entities even if we are unable to disambiguate the references using title (or other attributes).

First, we observe that author 'Don White' has co-authored a paper (P_1) with 'John Black' who is at MIT, while the author 'Dave White' does not have any co-authored papers with authors at MIT. We can use this observation to disambiguate between the two authors. In particular, since the co-author of 'D. White' in P_2 is 'Susan Grey' of MIT, there is a higher likelihood that the author 'D. White' in P_2 is 'Don White'. The reason is that the data suggests a connection between author 'Don White' with MIT and an absence of it between 'Dave White' and MIT.

Second, we observe that author 'Don White' has co-authored a paper ($P4$) with 'Joe Brown' who in turn has co-authored a paper with 'Liz Pink'. In contrast, author 'Dave White' has not co-authored any papers with either 'Liz Pink' or 'Joe Brown'. Since 'Liz Pink' is a co-author of P_6, there is a higher likelihood that 'D. White' in P_6 refers to author 'Don White' compared to author 'Dave White'. The reason is that often co-author networks form groups/clusters of authors that do related research and may publish with each other. The data suggests that 'Don White', 'Joe Brown' and 'Liz Pink' are part of the cluster, while 'Dave White' is not.

At first glance, the analysis above (used to disambiguate references that could not be resolved using conventional feature-based techniques) may seem ad-hoc and domain dependent. A general principle emerges if we view the database as a graph of inter-connected entities (modeled as nodes) linked to each other via relationships (modeled as edges). Figure 1 illustrates the entity-relationship graph corresponding to the toy database consisting of *authors* and *papers* records. In the graph, entities containing references are linked to the entities they refer to. For instance, since the reference 'Sue Grey' in P_2 is unambiguously resolved to author 'Susan Grey', paper P_2 is connected by an edge to author A_3. Similarly, paper P_5 is connected to authors A_5 ('Joe Brown') and A_6 ('Liz Pink'). The ambiguity of the references 'D. White' in P_2 and P_6 is captured by linking papers P_2 and P_6 to both 'Dave White' and 'Don White' via two "choice nodes" (labeled '1' and '2' in the figure). These "choice nodes" serve as `OR`-nodes in the graph and represent the fact that the reference 'D. White' refers to either one of the entities linked to the choice nodes.

Given the graph view of the toy database, the analysis we used to disambiguate 'D. White' in P_2 and P_6 can be viewed as an application of the following general principle:

Context Attraction Principle (CAP): *If reference r made in the context of entity x refers to an entity y_j whereas the description provided by r matches multiple entities $y_1, y_2, \ldots, y_j, \ldots, y_N$, then x and y_j are likely to be more strongly connected to each other via chains of relationships than x and y_l ($l = 1, 2, \ldots, N$, $l \neq j$).* □

Let us now get back to the toy database. The first observation we made, regarding disambiguation of 'D. White' in P_2, corresponds to the presence of the following path (i.e., *relationship chain* or *connection*) between the nodes 'Don White' and P_2 in the graph: $P_2 \rightarrow$ 'Susan Grey' $\rightarrow$ 'MIT' $\rightarrow$ 'John Black' $\rightarrow P_1 \rightarrow$ 'Don White'. Similarly, the second observation, regarding disambiguation of 'D. White' in P_6 as 'Don White', was based on the presence of the following path: $P_6 \rightarrow$ 'Liz Pink' $\rightarrow P_5 \rightarrow$ 'Joe Brown' $\rightarrow P_4 \rightarrow$ 'Don White'. There were no paths between P_2 and 'Dave White' or between P_6 and 'Dave White' (if we ignore '1' and '2' nodes). So, after applying the CAP principle, we concluded that the reference 'D. White' in both cases probably corresponded to the author 'Don White'. In general, there could have been paths not only between P_2 and 'Don White' but also between P_2 and 'Dave White'. In that case, to determine if 'D. White' is 'Don White or 'Dave White' we should have been able to measure whether 'Don White' or 'Dave White' is more strongly connected to P_2.

The generic approach therefore first *discovers connections* between the entity, in the context of which the reference appears, and the matching candidates for that reference. It then *measures the connection strength* of the discovered connections in order to give preference to one of the matching candidates. The above discussion naturally leads to two questions:

1. Does the context attraction principle hold over real data sets. That is, if we disambiguate references based on the principle, will the references be correctly disambiguated?
2. Can we design a generic solution to exploiting relationships for disambiguation?

Of course, the second question is only important if the answer to the first is yes. However, we cannot really answer the first unless we develop a general strategy to exploiting relationships for disambiguation and testing it over real data. We will develop one such general, domain-independent strategy for exploiting

relationships for disambiguation which we refer to as RelDC in Section 4. We perform extensive testing of RelDC over both real data from two different domains as well as synthetic data to establish that exploiting relationships (as is done by RelDC) significantly improves the data quality. Before we develop RelDC, we first develop notation and concepts needed to explain our approach in Section 3.

3 Problem definition

In this section we first develop notation and then formally define the problem of reference disambiguation. The notation is summarized in Table 1.

Notation	Meaning
$\mathcal{D}$	The database
$X = \{x_i\}$	The set of all entities in $\mathcal{D}$
$x_i.r_k$	the kth reference of entity x_i
$d[x_i.r_k]$	the (to-be-found) entity that $x_i.r_k$ refers to
$CS[x_i.r_k]$	the choice set for $x_i.r_k$
$y_1, y_2, \ldots, y_N$	the N elements of choice set $CS[x_i.r_k]$
$G(V, E)$	the entity-relationship graph for $\mathcal{D}$
$v[x_i]$	the (regular) node in G that corresponds to entity x_i
$cho[x_i.r_k]$	the choice node for reference $x_i.r_k$
e_0	edge $e_0 = (v[x_i], cho[x_i.r_k])$
e_j	edge $e_j = (cho[x_i.r_k], v[y_j])$ $(j = 1, 2, \ldots, N)$
w_j	the weight of edge e_j $(j = 0, 1, \ldots, N)$
L	the path length limit parameter
$\mathcal{P}_L(u, v)$	the set of all L-short simple paths between nodes u and v in G
$c(u, v)$	the connection strength between nodes u and v in G
$\mathcal{N}_r(v)$	the neighborhood of node v of radius r in graph G

Table 1: Notation

3.1 References

Let $\mathcal{D}$ be the database which contains references that are to be resolved. Let $X = \{x_i : i = 1, 2, \ldots, |X|\}$ be the set of all entities[3] in $\mathcal{D}$. Each entity x_i consists of a set of m_{x_i} properties $x_i.a_1, x_i.a_2, \ldots, x_i.a_{m_{x_i}}$ and contains a set of n_{x_i} references $x_i.r_1, x_i.r_2, \ldots, x_i.r_{n_{x_i}}$. Each reference $x_i.r_k$ is essentially a description and may itself consist of one or more attributes $x_i.r_k.b_1, x_i.r_k.b_2, \ldots$. For instance, in the example from Section 2, *paper* entities contain one-attribute `authorRef` references in the form ⟨*author name*⟩. If, besides author names, author affiliation were also stored in the *paper records*, then `authorRef` references would have consisted of two attributes – ⟨*author name*, *author affiliation*⟩.

Choice set. Each reference $x_i.r_k$ semantically refers to a single specific entity in X which we denote by $d[x_i.r_k]$. The description provided by $x_i.r_k$ may, however, match a set of one or more entities in X. We refer to this set as the *choice set* of reference $x_i.r_k$ and denote it by $CS[x_i.r_k]$. The choice set consists of all the entities that $x_i.r_k$ could potentially refer to. We assume $CS[x_i.r_k]$ is given for each $x_i.r_k$. If it is not given, we assume a feature-based similarity approach is used to construct $CS[x_i.r_k]$ by choosing all

[3]Entities here have essentially the same meaning as in the standard E/R model.

of the candidates such that FBS similarity between them and $x_i.r_k$ exceed a given threshold. To simplify notation, we will always assume $CS[x_i.r_k]$ has N (i.e., $N = |CS[x_i.r_k]|$) elements $y_1, y_2, \ldots, y_N$.

3.2 The entity-relationship graph

RelDC views the resulting database $\mathcal{D}$ as an undirected entity-relationship graph[4] $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each node corresponds to an entity and each edge to a relationship.[5] Notation $v[x_i]$ denotes the vertex in G that corresponds to entity $x_i \in X$. Note that if entity u contains a reference to entity v, then the nodes in the graph corresponding to u and v are linked since a reference establishes a relationship between the two entities. For instance, `authorRef` reference from paper P to author A corresponds to "A writes P" relationship.

In the graph G, edges have weights, nodes do not have weights. Each edge weight is a real number in $[0, 1]$, which reflects the degree of confidence the relationship, corresponding to the edge, exists. For instance, in the context of our author matching example, if we are 100% confident 'John Black' is affiliated with MIT, then we assign weight of 1 to the corresponding edge. But if we are only 80% confident, we assign the weight of 0.80 to that edge. By default all weights are equal to 1. Notation "edge label" means the same as "edge weight".

References and linking. If $CS[x_i.r_k]$ has only one element, then $x_i.r_k$ is resolved to y_1, and graph G contains an edge between $v[x_i]$ and $v[y_1]$. This edge is assigned a weight of 1 to denote that the algorithm is 100% confident that $d[x_i.r_k]$ is y_1. If $CS[x_i.r_k]$ has more than 1 elements, then graph G contains a **choice**

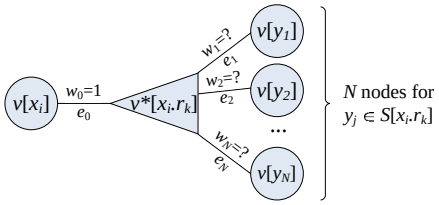

Figure 2: A choice node

node $cho[x_i.r_k]$, as shown in Figure 2, to reflect the fact that $d[x_i.r_k]$ can be one of $y_1, y_2, \ldots, y_N$. Node $cho[x_i.r_k]$ is linked with node $v[x_i]$ via edge $e_0 = (v[x_i], cho[x_i.r_k])$. Node $cho[x_i.r_k]$ is also linked with N nodes $v[y_1], v[y_2], \ldots, v[y_N]$, for each y_j in $CS[x_i.r_k]$, via edges $e_j = (cho[x_i.r_k], v[y_j])$ $(j = 1, 2, \ldots, N)$. Nodes $v[y_1], v[y_2], \ldots, v[y_N]$ are called the *options* of choice $cho[x_i.r_k]$. Edges $e_1, e_2, \ldots, e_N$ are called the *option-edges* of choice $cho[x_i.r_k]$. The weights of option-edges are called *option-edge weights* or simply *option weights*. The weight of edge e_0 is 1. Each weight w_j of edges e_j $(j = 1, 2, \ldots, N)$ is undefined initially. Since these option-edges $e_1, e_2, \ldots, e_N$ represent mutually exclusive alternatives, the sum of their weights should be 1: $w_1 + w_2 + \cdots + w_N = 1$.

3.3 The objective of reference disambiguation

To *resolve* reference $x_i.r_k$ means to choose one entity y_j from $CS[x_i.r_k]$ in order to determine $d[x_i.r_k]$. If entity y_j is chosen as the outcome of such a disambiguation, then $x_i.r_k$ is said to be *resolved to* y_j or simply

[4] A standard entity-relationship graph can be visualized as an E/R schema of the database that has been *instantiated* with the actual data.

[5] We will concentrate primarily on binary relationships. Multiway relationships are rare and most of them can be converted to binary relationships [16]. Most of the design models/tools only deal with binary relationships, for instance ODL (Object Definition Language) supports only binary relationships.

resolved. Reference $x_i.r_k$ is said to be *resolved correctly* if this y_j is $d[x_i.r_k]$. Notice, if $CS[x_i.r_k]$ has just one element y_1 (i.e., $N = 1$), then reference $x_i.r_k$ is automatically resolved to y_1. Thus reference $x_i.r_k$ is said to be *unresolved* or *uncertain* if it is not resolved yet to any y_j and also $N > 1$.

From the graph theoretic perspective, to resolve $x_i.r_k$ means to assign weights of 1 to one edge e_j, $1 \le j \le N$ and assign weights of 0 to the other $N-1$ edges $e_1, e_2, \ldots, e_{j-1}, e_{j+1}, \ldots, e_N$. This will indicate that the algorithm chooses y_j as $d[x_i.r_k]$.

The **goal** of reference disambiguation is to resolve all references as correctly as possible, that is, for each reference $x_i.r_k$ to correctly identify $d[x_i.r_k]$. We will use notation $Resolve(x_i.r_k)$ to refer to the procedure which resolves $x_i.r_k$. The *goal* is thus to construct such $Resolve(\cdot)$ which should be as accurate as possible. The *accuracy* of reference disambiguation is the fraction of references being resolved that are resolved correctly.

The **alternative goal** is for each $y_j \in CS[x_i.r_k]$ to associate weight w_j that reflects the degree of confidence that y_j is $d[x_i.r_k]$. For that alternative goal, $Resolve(x_i.r_k)$ should label each edge e_j with such a weight. Those weights can be **interpreted** later to achieve the main goal: for each $x_i.r_k$ try to identify only one y_j as $d[x_i.r_k]$ correctly. We emphasize this alternative goal since most of the discussion of RelDC approach is devoted to one approach for computing those weights. An interpretation of those weights (in order to try to identify $d[x_i.r_k]$) is a small final step of RelDC. Namely, we achieve this by picking y_j such that w_j is the largest among $w_1, w_2, \ldots, w_N$. That is, the outcome of $Resolve(x_i.r_k)$ is $y_j : w_j = \max_{l=1,2,\ldots,N} w_l$.

3.4 Connection Strength and Context Attraction Principle

As mentioned before, RelDC resolves references based on context attraction principle that was discussed in Section 2. We now state the principle more formally. Crucial to the principle is the notion of *connection strength* between two entities x_i and y_j (denoted $c(x_i, y_j)$ which captures how strongly x_i and y_j are connected to each other through relationships. Many different approaches can be used to measure $c(x_i, y_j)$ and will be discussed in Section 4. Given the concept of $c(x_i, y_j)$, we can restate the context attraction principle as follows:

Context Attraction Principle: Let $x_i.r_k$ be a reference and $y_1, y_2, \ldots, y_N$ be elements of its choice set $CS[x_i.r_k]$ with corresponding option weights $w_1, w_2, \ldots, w_N$ (recall that $w_1 + w_2 + \cdots + w_N = 1$). The context attraction principle states that for all $l, j \in [1, N]$, if $c_l \ge c_j$ then $w_l \ge w_j$, where $c_l = c(x_i, y_l)$ and $c_j = c(x_i, y_j)$.

4 The RelDC approach

We now have developed all the concepts and notation needed to explain RelDC approach for reference disambiguation. Input to RelDC is the entity-relationship graph G discussed in Section 3 in which nodes correspond to entities and edges to relationships. We assume that feature-based similarity approaches have been used in constructing the graph G. The choice nodes are created only for those references that could not be disambiguated using only attribute similarity. RelDC will exploit relationships for further disambiguation and will output a resolved graph G in which each entity is fully resolved.

RelDC disambiguates references using the following four steps:

1. **Compute connection strengths**. For each reference $x_i.r_k$ compute the connection strength $c(x_i, y_j)$ for each $y_j \in CS[x_i.r_k]$. The result is a set of equations that relate $c(x_i, y_j)$ with the option weights: $c(x_i, y_j) = g_{ij}(\overline{w})$. Here, $\overline{w}$ denote the set of all option weights in the graph G.
2. **Determine equations for option weights**. Using the equations from Step 1 and the CAP determine a set of equations that relate option weights to each other.

3. **Compute weights.** Solve the set of equations from Step 2.
4. **Resolve References.** Utilize/interpret the weights computed in Step 3 as well as attribute-based similarity to resolve references.

We now discuss the above steps in more detail in the following subsections.

4.1 Computing connection strength

Computation of $c(x_i, y_j)$ consists of two phases. The first phase discovers connections between x_i and y_j. The second phase computes/measures the strength in connections discovered by the first phase.

4.1.1 The connection discovery phase

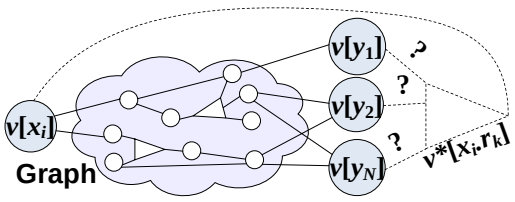

Figure 3: Graph

In general there can be many connections between $v[x_i]$ and $v[y_j]$ in G. Intuitively, many of those (e.g., very long ones) are not very important. To capture most important connections while still being efficient, the algorithm computes the set of all L-short simple paths $\mathcal{P}_L(x_i, y_j)$ between nodes $v[x_i]$ and $v[y_j]$ in graph G. A path is *L-short* if its length is no greater than parameter L. A path is *simple* if it does not contain duplicate nodes.

Illegal paths. Not all of the discovered paths are considered when computing $c(x_i, y_j)$ (to resolve reference $x_i.r_k$). Let $e_1, e_2, \ldots, e_N$ be the option-edges associated with the reference $x_i.r_k$. When resolving $x_i.r_k$, RelDC tries do determine weights of these edges via connections that exist in the remainder of the graph not including those edges. To achieve this, RelDC actually discovers paths not in graph G, but in $\widetilde{G} = G - cho[x_i.r_k]$, see Figure 3. That is, $\widetilde{G}$ is graph G with node $cho[x_i.r_k]$ removed. Also, in general, paths considered when computing $c(x_i, y_j)$ may contain option-edges of some choice nodes. If a path contains an option-edge of a choice node, it should not contain another option-edge of the same choice node. For instance, if a path used to compute connection strength between two nodes in the graph contained an option edge e_j of the choice node shown in Figure 2, it must not contain any of the rest of the option-edges $e_1, e_2, \ldots, e_{j-1}, e_{j+1}, \ldots, e_N$.

4.1.2 Measuring connection strength phase

A natural way to compute the connection strength $c(u, v)$ between node u and v is to compute it as the probability to reach node v from node u via random walks in graph G where each step is done with certain probability. Such problems have been studied for graphs in the previous work under Markovian assumptions. For example, White et al. in [37] proposes to use Markov chains to compute relative importance in graphs. The graph in our case, however, is not Markovian due to presence of illegal paths (introduced by choice nodes). For example, consider some path $p_1 = v \rightsquigarrow v[x_i] \rightarrow cho[x_i.r_k]$, where $v[x_i]$ and $cho[x_i.r_k]$ are from Figure 2. We can continue this path by following $cho[x_i.r_k] \rightarrow v[y_1]$ link with some probability, i.e. the new path is $v \rightsquigarrow v[x_i] \rightarrow cho[x_i.r_k] \rightarrow v[y_1]$. However if we consider a different path

$p_2 = v \rightsquigarrow v[y_2] \rightarrow cho[x_i.r_k]$, we cannot follow $cho[x_i.r_k] \rightarrow v[y_1]$ link because edges e_1 and e_2 are mutually exclusive. So, in general, for any path $p = v \rightsquigarrow cho[x_i.r_k]$, the probability to follow $cho[x_i.r_k] \rightarrow v[y_1]$ link depends not only on $cho[x_i.r_k]$ but also on other nodes on path p. This violates the Markovian assumptions. Thus those existing approaches cannot be applied directly.

In Appendix A we have developed a model called the *probabilistic model (PM)* which treats edge weights as probabilities that those edges exist and which is capable of handling illegal paths. In this section we present the *weight-based model (WM)* which is a simplification of PM. Other models can be derived from [14, 37].

WM is a very intuitive model which is suited well for illustrating issues related to computing $c(u, v)$. WM computes $c(u, v)$ as the sum $\sum_{p \in \mathcal{P}_L(u,v)} c(p)$ of the connection strength $c(p)$ of each path p in $\mathcal{P}_L(u, v)$. The connection strength $c(p)$ of path p from u to v is the probability to follow path p in graph G. Next we explain how WM computes $c(p)$. After explaining WM we shall briefly take up the differences between WM and PM.

Motivating $c(p)$ formula. Which factors should be taken into account when computing the connection strength $c(p)$ of each individual path p?

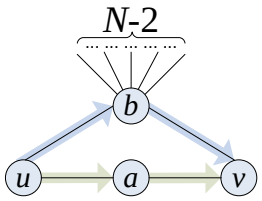

Figure 4: Motivating $c(p)$ formula

Figure 4 illustrates two different paths (or connections) between nodes u and v: p_a=u→a→v and p_b=u →b→v. Assume that all edges in this figure have weight of 1. Let us understand which connection is better.

Both connections have an equal length of two. One connection is going via node a, the other one via b. The intent of Figure 4 is to show that b "connects" many things, not just u and v, whereas a "connects" only u and v. We argue the connection between u and v via b is much weaker than the connection between u and v via a: since b connects many things it is not surprising we can connect u and v via b. For example, for the author matching problem, u and v can be two authors, a can be a publication and b a university.

To capture the fact that $c(p_a) > c(p_b)$, we measure $c(p_a)$ and $c(p_b)$ as the probabilities to follow paths p_a and p_b respectively. Notice, measures such as path length, network flow do not capture this fact. We compute those probabilities as follows. For path p_b we start from u. Next we have a choice to go to a or b with probabilities of $\frac{1}{2}$, and we choose to follow (u, b) edge. From node b we can go to any of the $N - 1$ nodes (cannot go back to u) but we go specifically to v. So the probability to reach v via path p_b is $\frac{1}{2(N-1)}$. For path p_a we start from u, we go to a with probability $\frac{1}{2}$ at which point we have no choice but to go to v, so the probability to follow p_a is $\frac{1}{2}$.

General WM formula. In general, each L-short simple path p can be viewed as a sequence of m nodes $\langle v_1, v_2, \ldots, v_m \rangle$, where $m \leq L + 1$, as shown in Figure 5. Figure 5 shows that from node v_i it is *possible to follow*[6] $n_i + 1$ edges labeled $w_{i0}, w_{i1}, \ldots, w_{in_i}$. WM computes the connection strength of path p as the probability Pr to follow path p: $c(p) = Pr$. Probability Pr is computed as the product of two probabilities: $Pr = Pr_1 \cdot Pr_2$, where Pr_1 is the probability that path p exists and Pr_2 is the probability "to follow path p given that p exists".

[6]It is not possible to follow edges following which would make path not simple.

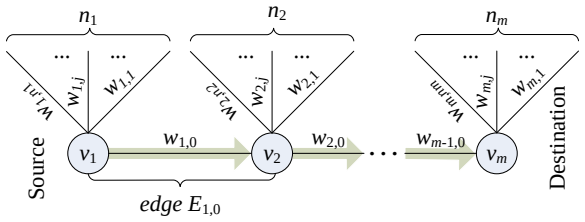

Figure 5: Computing $c(p)$ of path $p = v_1 \to v_2 \to \cdots \to v_m$. Only "possible-to-follow" edges are shown.

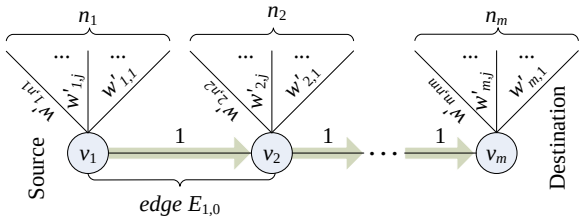

Figure 6: Computing $c(p)$: new labels under assumption that p exists.

First of all, path p should exist and thus each edge on this path should exist. WM computes the probability Pr_1 that p exist as the product of probabilities that each edge on path p exists: $Pr_1 = w_{10}w_{20} \times \cdots \times w_{m-1,0}$. That is, WM assumes that each edge E_{i0} $(i = 1, 2, \ldots, m-1)$ exists independently from other edges $E_{l,0}$ $(l = 1, 2, \ldots, m-1,\ l \neq i)$. Recall that WM is a simplification of PM presented in Appendix A. In Appendix A we show that such an assumption of independence is reasonable.

Next WM computes probability Pr_2 to follow path p given that p exists. If we assume that p exists, then situation will look like illustrated in Figure 6. In that figure all edges are labeled with weights $w'_{i,j}$ which reflect how weights $w_{i,j}$ change if we add the assumption that path p exists. For example, $w'_{i0} = 1$ $(i = 1, 2, \ldots, m-1)$ because each edge E_{i0} should exist. For each $w'_{i,j}$, where $j \neq 0$, either $w'_{i,j} = w_{i,j}$, or $w'_{i,j} = 0$. To understand why $w'_{i,j}$ can be zero, consider path p_1 = 'Don White' $\to P_4 \to$ Joe $\to P_5 \to$ Liz $\to P_6 \to$ '2' $\to$ 'Dave White' in Figure 1 as an example. If we assume p_1 exists, then edge ('2', 'Dave White') must exist and consequently edge ('2', 'Don White') does not exist. So, if path p_1 exists, the weight of edge ('2', 'Don White') is zero. That is why in general either $w'_{i,j} = w_{i,j}$, or, if the corresponding edge $E_{i,j}$ cannot exist under assumption that path p exists, then $w'_{i,j} = 0$.

WM computes probability Pr_2 "to follow path p given that p exists" as as the product of probabilities to follow each edge on p. In WM, the probability to follow an edge is proportional to the weight of the edge. For example, the probability to follow edge E_{10} in Figure 6 is: $\frac{1}{1+w'_{11}+w'_{1,2}+\cdots+w'_{1,n_1}}$. The connection strength of path p is computed as $c(p) = Pr_1 \cdot Pr_2$. The final formula for $c(p)$ is:

$$c(p) = \prod_{i=1,2,\ldots,m-1} \frac{w'_{i0}}{1 + \sum_{j=1,2,\ldots,n_i} w'_{i,j}}. \tag{1}$$

The total connection strength between nodes u and v is computed as the sum of connection strengths of paths in $\mathcal{P}_L(u, v)$:

$$c(u, v) = \sum_{p \in \mathcal{P}_L(u,v)} c(p). \tag{2}$$

Measure $c(u, v)$ is the probability to reach v from u by following only L-short simple paths, such that the probability to follow an edge is proportional to the weight of the edge.

Connection strengths in toy database. Let us compute connection strengths c_1, c_2, c_3, and c_4 for the toy database illustrated in Figure 1. Those connection strength are defined as follows: $c_1 = c(P_2, \text{'Dave White'})$, $c_2 = c(P_2, \text{'Don White'})$, $c_3 = c(P_6, \text{'Dave White'})$, and $c_4 = c(P_6, \text{'Don White'})$. Later, those connection strengths will be used to compute option weights w_1, w_2, w_3, and w_4.

Consider first computing $c_1 = c(P_2, \text{'Dave White'})$ in the context of disambiguating 'D. White' reference in P_2. Recall, for that reference choice node '1' has been created. The first step is to remove choice '1' from consideration. The resulting graph $\widetilde{G} = G - \text{'1'}$ is shown in Figure 7. The next step is to discover all L-short

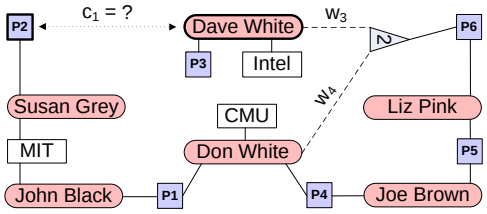

Figure 7: Computing $c_1 = c(P_2, \text{'Dave White'})$: $\widetilde{G} = G - \text{'1'}$.

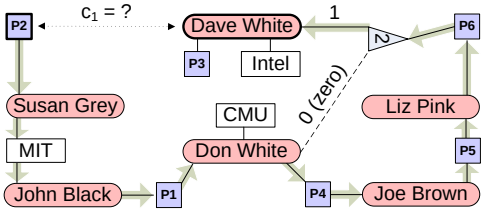

Figure 8: Computing $c_1 = c(P_2, \text{'Dave White'})$: under assumption that path ($P_2 \rightsquigarrow$ 'Dave White') exists. Edge '2' $\rightarrow$ 'Dave' exists, therefore edge '2' $\rightarrow$ 'Don' does not exist.

simple paths in graph $\widetilde{G}$ between P_2 and 'Dave White'. Let us set $L = \infty$, then there is only one such path: $p_1 = P_2 \rightarrow$ Susan $\rightarrow$ MIT $\rightarrow$ John $\rightarrow P_1 \rightarrow$ Don $\rightarrow P_4 \rightarrow$ Joe $\rightarrow P_5 \rightarrow$ Liz $\rightarrow P_6 \rightarrow$ '2' $\rightarrow$ Dave White. The discovered connection is too long to be meaningful in practice, but we will consider it for pedagogical reasons. To compute the connection strength of path p_1 we first compute the probability that path p_1 exists. Path p_1 exists if and only if edge between '2' and 'Dave White' exists, so the probability that p_1 exists is w_3. Now we assume that p_1 exists and will compute the probability to follow p_1 given that p_1 exists on the graph shown in Figure 8. That probability is $\frac{1}{2}$. So $c(p_1) = \frac{w_3}{2}$. The same result can be obtained by directly applying Equation (1). After computing c_2, c_3, and c_4 in a similar fashion we have:

1. $c_1 = c(P_2, \text{'Dave White'}) = c(p_1) = \frac{w_3}{2}$.
2. $c_2 = c(P_2, \text{'Don White'}) = c(P_2 \rightarrow \text{Susan} \rightarrow \text{MIT} \rightarrow \text{John} \rightarrow P_1 \rightarrow \text{'Don White'}) = 1$.
3. $c_3 = c(P_6, \text{'Dave White'}) = \frac{w_1}{2}$
4. $c_4 = c(P_6, \text{'Don White'}) = 1$

Notice, the toy database is small and 'MIT' connects only two authors. In more realistic examples, 'MIT' will connect many authors, so connections via 'MIT' will be weak.

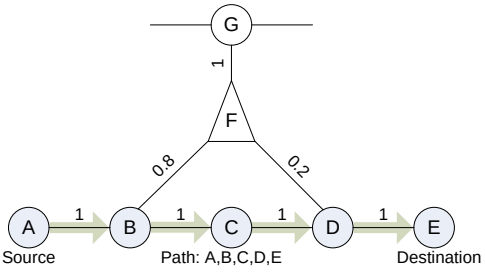

Figure 9: Comparing WM and PM.

Comparing WM and PM. To illustrate the differences between WM and PM let us consider an example shown in Figure 9. The objective is to compute the connection strength of path $p = A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$. Edges FB and FD are option-edges of choice node F.

For WM we use Equation (1) to compute $c(p)$ as $c(p) = 1 \cdot \frac{1}{1+0.8} \cdot 1 \cdot \frac{1}{1+0.2} = \frac{25}{54}$. Unlike WM, the probabilistic model (PM) treats edge weights not as *weights* but as *probabilities* that those edges exist. For a particular path p being considered PM identifies the group of edges E_p that can affect $c(p)$. PM iterates over possible combinations of which edge in E_p exists and which does not, and for each combination computes the probability to follow the path being considered. Appendix A discusses PM in detail and proposes several optimizations to speedup the computations.

For the case in Figure 9, PM will consider two mutually exclusive cases: edge BF exists and edge FD exists. First assume that edge BF exists. The probability of that is 0.8. If BF exists, the probability to follow p is $\frac{1}{2}$. Then PM will assume that edge FD exists. The probability of that is 0.2. The probability to follow p if FD exists is $\frac{1}{2}$. So the total probability to follow path p is $c(p) = 0.8/2 + 0.2/2 = 0.5$.

4.2 Determining equations for option-edge weights

Given the connection strength measures $c(x_i, y_j)$ for each unresolved reference $x_i.r_k$ and its options y_j, we can use the context attraction principle to determine the relationships between the weights associated with the option-edges in the graph G. Note that the context attraction principle does not contain any specific strategy on how to relate weights to connection strengths. Any strategy that assigns weight such that if $c_l \geq c_j$ then $w_l \geq w_j$ is appropriate, where $c_l = c(x_i, y_l)$ and $c_j = c(x_i, y_j)$. In particular, we use the strategy where weights $w_1, w_2, \ldots, w_N$ are proportional to the corresponding connection strengths: $w_j \cdot c_l = w_l \cdot c_j$. Using this strategy weight w_j $(j = 1, 2, \ldots, N)$ is computed as:

$$w_j = \begin{cases} \frac{c_j}{c_1 + c_2 + \cdots + c_N} & \text{if } (c_1 + c_2 + \cdots + c_N) > 0; \\ \frac{1}{N} & \text{if } (c_1 + c_2 + \cdots + c_N) = 0. \end{cases} \quad (3)$$

For instance, for the toy database we have:

1. $w_1 = c_1/(c_1 + c_2) = \frac{w_3}{2}/(1 + \frac{w_3}{2})$
2. $w_2 = c_2/(c_1 + c_2) = 1/(1 + \frac{w_3}{2})$
3. $w_3 = c_3/(c_3 + c_4) = \frac{w_1}{2}/(1 + \frac{w_1}{2})$
4. $w_4 = c_4/(c_3 + c_4) = 1/(1 + \frac{w_1}{2})$

4.3 Determining all weights by solving equations.

Given a system of equations relating option-edge weights as derived in Section 4.2, our goal next is to determine values for the option-edge weights that satisfy the equations. Before we discuss how such equations can be solved in general, let us first solve the option-edge weight equations in the toy example. These equations, given an additional constraint that all weights should be in $[0, 1]$, have a unique solution $w_1 = 0$, $w_2 = 1$, $w_3 = 0$, and $w_4 = 1$. Once we have computed the weights, RelDC will interpret these weights to resolve the references. In the toy example, weights $w_1 = 0$, $w_2 = 1$, $w_3 = 0$, and $w_4 = 1$ will lead RelDC to resolve 'D. White' in both P_2 and P_6 to 'Don White'.

In general case, Equations (3), (1), and (2) define each option weight as a function of other option weights: $w_i = f_i(\overline{w})$. The exact function for w_j is determined by Equations (3), (1), and (2) and by the paths that exists between $v[x_i]$ and $v[y_j]$ in G. Often, in practice, $f_i(\overline{w})$ is constant leading to the equation of the form $w_i = const$.

The goal is to find such a combination of weights that "satisfies" the system of $w_i = f_i(\overline{w})$ equations along with the constraints on the weights. Since a system of equations, each of the type $w_i = f_i(\overline{w})$, might not have an exact solution, we transform the equations into the form $f_i(\overline{w}) - \delta_i \leq w_i \leq f_i(\overline{w}) + \delta_i$. Here variable δ_i, called *tolerance*, can take on any real nonnegative value. The problem transforms into solving the NLP problem where the constraints are specified by the inequalities above and the objective is to minimize the sum of all δ_i's.[7] Such a system of equations always has a solution. This is because functions f_i's by the nature of their construction are such that $0 \leq f_i(\overline{w}) \leq 1$, for all $\overline{w}$ and i. Hence, if all deltas are one: $\delta_i = 1$, for all i, then the assignment where $w_i = 0$ for all i is a solution that satisfies the constraints of the NLP problem. The goal, of course, is to find a better solution by requiring that $\sum_i \delta_i$ is minimized.

The straightforward approach to solving the resulting NLP problem is to use one of the off-the-shelf math solver such as SNOPT. Such solvers, however, do not scale to large problem sizes that we encounter

[7] Additional constraints are: $0 \leq w_i \leq 1$, $\delta_i \geq 0$, for all w_i, δ_i.

in data cleaning as will be discussed in Section 5. We therefore exploit a simple iterative approach, which is outlined below. The iterative method first iterates over each reference $x_i.r_k$ and assigns weight of $\frac{1}{|CS[x_i.r_k]|}$ to each w_j. It then starts its major iterations in which it first computes $c(x_i, y_j)$ for all i and j, using Equation (2). After all $c(x_i, y_j)$ values are computed, they are used to compute all w_j's using Equation (3). Note that the values of w_j's will change from $\frac{1}{|CS[x_i.r_k]|}$ to new values. The algorithm performs several major iterations until the weights converge (the resulting changes across iterations are negligible) or the algorithm is explicitly stopped.

Let us perform one iteration of the iterative method for the example above. First $w_1 = w_2 = \frac{1}{2}$ and $w_3 = w_4 = \frac{1}{2}$. Next $c_1 = \frac{1}{4}$, $c_2 = 1$, $c_3 = \frac{1}{4}$, and $c_4 = 1$. Finally, $w_1 = \frac{1}{5}$, $w_2 = \frac{4}{5}$, $w_3 = \frac{1}{5}$, and $w_4 = \frac{4}{5}$. If we stop the algorithm at this point and interpret w_j's, then the RelDC's answer is identical to that of the exact solution: 'D. White' is 'Don White'.

Note that the above described iterative procedure computes only an *approximate* solution for the system whereas the solver finds the exact solution. Let us refer to iterative implementation of RelDC as *Iter-RelDC* and denote the implementation that uses a solver as *Solv-RelDC*. For both Iter-RelDC and Solv-RelDC, after the weights are computed, those weights are **interpreted** to produce the final result, as discussed in Section 4. It turned out that the accuracy of Iter-RelDC (with a small number of iterations, such as 10–20) and of Solv-RelDC is practically identical. This is because even though the iterative method does not find the exact weights, those weights are close enough to those computed using a solver. Thus, when the weights are *interpreted*, both methods obtain similar results.

4.4 Resolving references by interpreting weights.

When resolving references $x_i.r_k$ and deciding which entity among $y_1, y_2, \ldots, y_N$ from $CS[x_i.r_k]$ is $d[x_i.r_k]$, RelDC chooses such y_j that w_j is the largest among $w_1, w_2, \ldots, w_N$. Notice, to resolve $x_i.r_k$ we could have also combined w_j weights with feature-based similarities $FBS(x_i, y_j)$ (e.g., as a weighted sum), but we do not study that approach in this paper.

5 Key optimizations of RelDC

In this section we present only key optimizations of RelDC. A more complete list of optimizations, their taxonomy and analysis are presented in Appendix B. As far as optimization techniques are concerned, the RelDC procedure can be logically divided into two phases: the *relationship discovery* (a.k.a. AllPaths) phase and the *weight computation* phase. For RelDC, AllPaths is the major bottleneck, so most of the optimizations are for the AllPaths part of RelDC. We will conclude this section with the computational analysis of RelDC.

5.1 Constraining the problem

This section lists several optimizations that improve the efficiency of RelDC by constraining/simplifying the problem.

Limiting paths length. AllPaths algorithm can be specified to look only for paths of length no greater than a parameter L. This optimization is based on the premise that longer paths tend to have smaller connection strengths while RelDC will need to spend more time to discover those.

Weight cut-off threshold. This optimization can be applied after a few iterations of Iter-RelDC. When resolving reference $x_i.r_k$, see Figure 2, Iter-RelDC can use a threshold to prune several y_j's from $CS[x_i.r_k]$.

If the current weight w_j is too small with respect to the rest of weights $w_1, w_2, \ldots, w_{j-1}, w_{j+1}, \ldots, w_N$, then RelDC will assume y_j cannot be $d[x_i.r_k]$ and will remove y_j from further consideration.

The threshold is computed per choice basis. For $cho[x_i.r_k]$ it is computed as $T = \alpha \cdot \frac{1}{N}$, where α is a real number (a fixed parameter) from interval $[0, 1)$.[8] Consequently, RelDC permanently assigns weight of zero to those edges e_j's which currently have weight $w_j : w_j < T$. This means that RelDC decides that the y_j's corresponding to those w_j's cannot be $d[x_i.r_k]$. This optimization improves efficiency since RelDC will not recompute the connection strengths between x_i and those zero-weight y_j's any longer.

Restricting path types. The analyst can specify path types of interest (or for exclusion) explicitly.[9] For example, the analyst can specify that only paths of type *node_type*1 $\rightarrow$ *node_type*2 $\rightarrow$ *node_type*4 $\rightarrow$ *node_type*1 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.

5.2 Depth-first and greedy implementation of AllPaths.

We have considered two approaches for implementing AllPaths algorithm: the depth-first and greedy.[10] While the depth-first implementation is straightforward, the greedy implementation deserve more in-depth discussion. The depth-first implementation of AllPaths is a good choice of an algorithm for discovering all L-short simple paths if skipping of paths is not allowed. This is because it requires small amount of space (just to keep one intermediate path) and its decision of which node to expand next is performed in $O(1)$ time.

However a *greedy* implementation might be a better option if one is interested in fine-tuning the *accuracy vs. performance* trade-off and decides to restrict the running time of the AllPaths algorithm. If one decides to stop a depth-first version of AllPaths algorithm abruptly at one point, certain important paths can still be not discovered. To address this drawback, a greedy version of the algorithm discovers the most important paths first and least important last.

5.2.1 Data structures for the greedy implementation

When looking for all L-short simple paths of type $u \rightsquigarrow v$, the greedy version maintains not just one intermediate path (like the depth-first version), but several. It uses two main data structures: a *paths storage* and a *priority queue*.

Each path is stored as a list, in reverse order. The *paths storage* is organized as a set of *overlapping lists* as follows. Since all of the paths start from u, many of the paths share common prefix. Thus to store paths say $p_1 = u \rightarrow 1$, $p_2 = u \rightarrow 1 \rightarrow 2$, $p_3 = u \rightarrow 1 \rightarrow 2 \rightarrow 3$, and $p_4 = u \rightarrow 1 \rightarrow 2 \rightarrow 4$ it is not necessary to keep four separate lists of lengths 1, 2, 3, and 3 respectively. A more efficient solution is to create a list $1 \rightarrow u$ and make p_1 point to it. Then create a list $2 \rightarrow p_1$ and make p_2 point to it. And so on: for p_3 the list is $3 \rightarrow p_2$ and for p_4 it is $4 \rightarrow p_2$. The combined length of the data structure is just 4 nodes (versus 9 nodes if keeping separate lists). Also notice when the greedy algorithm expands path $u \overset{p_1}{\rightsquigarrow} v$ to path $u \overset{p_1}{\rightsquigarrow} v \rightarrow x$, at that point it knows the exact location of (i.e., *a pointer to*) path $u \overset{p_1}{\rightsquigarrow} v$ in *paths storage*. Path $u \overset{p_1}{\rightsquigarrow} v$ is a prefix of path $u \overset{p_1}{\rightsquigarrow} v \rightarrow x$, but the algorithm does not need to spend time searching for its location in the *paths storage*, since it knows the pointer to p_1.

[8] The typical choices for α in our experiments were 0.0 (i.e., the optimization is not used), 0.2 and 0.3.

[9] This optimization has not been used in our experiments.

[10] All of the optimizations mentioned in this paper can be applied to both of these approaches.

5.2.2 Greedy algorithm

The algorithm maintains pointers to intermediate paths that can be expanded next in the priority queue. The key in that queue is the path connection strength. The algorithm retrieves from the priority queue the best intermediate path $u \overset{p_1}{\rightsquigarrow} x$ to expand next. By examining all direct neighbors of x, the algorithm then identifies nodes $z_1, z_2, \ldots, z_n$ such that paths $u \overset{p_1}{\rightsquigarrow} x \rightarrow z_i$ are legal paths in the context of RelDC approach. Those paths are added to the paths storage as lists $z_i \rightarrow p_1$ $(i = 1, 2, \ldots, N)$. Then the algorithm inserts pointers to paths $u \overset{p_1}{\rightsquigarrow} x \rightarrow z_i$ $(i = 1, 2, \ldots, N)$ into the priority queue with key $c(u \overset{p_1}{\rightsquigarrow} x \rightarrow z_i)$, where $c(u \overset{p_1}{\rightsquigarrow} x \rightarrow z_i)$ is the connection strength of path $u \overset{p_1}{\rightsquigarrow} x \rightarrow z_i$.[11] Thus the greedy implementation discovers most important paths first and least important paths last.

5.2.3 Comparing complexity of greedy and depth-first implementations.

Assume both version uses the same path length limit L. The difference in computational complexity between depth-first and greedy implementations of AllPaths arises at two points: (1) when the algorithm decides which path to expand next and (2) during the process of expanding paths (analyzing direct neighbors).

For the depth-first implementation, it takes $O(1)$ time to decide which path to expand next and it takes $O(degree(v))$ time to expand path $u \rightsquigarrow v$ further. The greedy implementation uses priority queue [13] in order to pick the best path to expand next. If n is the size of the priority queue, `get` procedure takes $O(\lg n)$ time and `insert` procedure takes $O(\lg n)$ time as well [13]. That is why for the greedy implementation, it takes $O(\lg n)$ time to decide which path to expand next since it uses `get` operation to retrieve the "best" path from the priority queue. It takes $O(degree(v) \cdot \lg (n + degree(v)))$ to expand path $u \rightsquigarrow v$ further, since it not only examines neighbors (like depth-first method) but also puts the new feasible paths $u \rightsquigarrow v \rightarrow x$ into the priority queue.

Thus, if the goal is to discover all L-short simple paths *without skipping* any paths, the depth-first version is expected to show better results than the greedy version. However, since the greedy version discovers the most important path first, it should be a better choice in terms of the *accuracy vs. performance* trade-off than its depth-first counterpart. That is, the greedy version is expected to be better if the execution time of the algorithm needs to be restricted. The time and space required by the greedy implementation can be restricted by using several thresholds to limit the number of nodes that can be expanded, the number of edges that can be analyzed, paths length, and the total number of paths observed.

5.3 NBH optimization: utilizing neighborhoods for path pruning.

he NBH optimization is probably the most important performance optimization presented in this paper. It consistently achieves 1–2 orders of magnitude performance improvement under variety of conditions.[12]

The *neighborhood* of a node v_0 of radius r_0 is the set of all the nodes that are reachable from v_0 via at most r_0 edges. 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, $\mathcal{N}_{r_0}(v_0)$, is a set of pairs:

$$\mathcal{N}_{r_0}(v_0) = \{(v, d) : v \in V,\ d = min\_dist(v, v_0),\ d \leq r_0\}.$$

[11] Notice, the connection strength $c(u \overset{p_1}{\rightsquigarrow} x \rightarrow z_i)$ can be computed incrementally: as $c(u \overset{p_1}{\rightsquigarrow} x)$ – which is already known, multiplied by the probability to follow edge (x, z_i).

[12] When the NBH optimization is used, the 1-*to-N implementation of AllPaths* optimization (from Section B.2.2) is not applicable and the GraphReach optimization (from Section B.2.2 as well) has little additional effect (for the tested datasets).

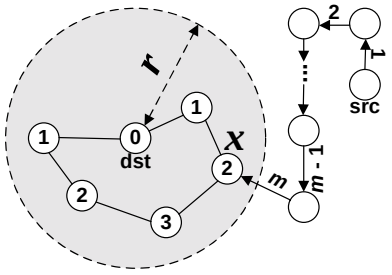

Figure 10: Neighborhood information

Recall, when resolving reference $x_i.r_k$, the algorithm will need to invoke AllPaths for N pairs – to compute $\mathcal{P}_L(x_i, y_j)$ $(j = 1, 2, \ldots, N)$ see Figure 2. This computation can be optimized by exploiting the *neighborhood information* of node x_i to prune certain paths as follows, see Figure 10.

When resolving a particular reference $x_i.r_k$, the algorithm first computes the neighborhood $\mathcal{N}_r(v[x_i])$ of $v[x_i]$ of radius r, where $r \leq L$. Then for each option y_j $(j = 1, 2, \ldots, N)$ it finds all simple paths from $v[y_j]$(source) to $v[x_i]$(destination). Assume the algorithm currently processes one of the N options, say $v[y_j]$, i.e. it is in the process of computing $\mathcal{P}_L(x_i, y_j)$. Assume the algorithm is at some intermediate stage where it currently observes intermediate path $p_1 : v[y_j] \overset{p_1}{\rightsquigarrow} x$ of length m (i.e., m edges), where node $x \neq v[x_i]$. If m is such that $(m + r) < L$, then the algorithm proceeds as usual. However, if $(m + r) \geq L$, then x must be inside $\mathcal{N}_r(v[x_i])$. If it is not inside, then path p_1 is pruned, since there cannot be an L-short path $v[y_j] \overset{p_1}{\rightsquigarrow} x \overset{p_2}{\rightsquigarrow} v[x_i]$ for any path $p_2 : x \overset{p_2}{\rightsquigarrow} v[x_i]$. Further, if x is inside $\mathcal{N}_r(v[x_i])$, then it is possible to retrieve from $\mathcal{N}_r(v[x_i])$ the minimum distance d between x and $v[x_i]$. This distance d should be such that $(m + d) \leq L$: otherwise path p_1 is pruned since there cannot be an L-short path $v[y_j] \overset{p_1}{\rightsquigarrow} x \overset{p_2}{\rightsquigarrow} v[x_i]$ for any path $p_2 : x \overset{p_2}{\rightsquigarrow} v[x_i]$.

After processing reference $x_i.r_k$ in such a fashion, neighborhood $\mathcal{N}_r(v[x_i])$ is discarded and memory it occupied is freed.

The NBH optimization can be improved further. Let us introduce a new term – the *actual radius* of neighborhood $\mathcal{N}_{r_0}(v_0)$:

$$r_{act} = \max_{v:(v,d)\in\mathcal{N}_{r_0}(v_0)} (min\_dist(v_0, v)).$$

While usually $r_{act} = r_0$, sometimes[13] $r_{act} < r_0$. This happens when nodes from the neighborhood of v_0 and their incident edges form a cluster which is not connected to the rest of the graph (or this cluster is the whole graph). In this situation $\mathcal{N}_{r_{act}}(v_0)$ is equal to $\mathcal{N}_r(v_0), \forall r \in [r_{act}, \infty)$. In other words, we know the neighborhood of v_0 of radius $r = \infty$. Regarding searching all simple paths as described above, this means that all intermediate nodes must always be inside the according neighborhood. The above methods for exploiting neighborhoods to prune certain paths should be modified to achieve the speedup as follows. After computing $\mathcal{N}_r(v[x_i])$ the following operation needs to be inserted: if $r_{act} < r$ then $r \leftarrow \infty$.

5.4 Storing discovered paths explicitly.

Once the paths are discovered on the first iteration, they can be exploited for speeding up the subsequent iterations when those paths need to be rediscovered again. One solution would be to store such paths explicitly in memory, if there is enough such, or on disk. After paths are stored, the subsequent iterations do not rediscover them, but rather work with the stored paths. Next we present several techniques that reduce the storage overhead of storing paths explicitly.

[13]Naturally, the greater the r_0 the more frequently this is likely to occur.

Path compression. We store paths because we need to recompute the connection strengths of those paths (on subsequent iterations), which can change as weights of option-edges change. One way of compressing path information is to find fixed-weight paths. Fixed-weight paths are paths the connection strength of which will not change because it does not depend on any other system variables that can change. Rather than storing a path itself, it is more efficient to store the (fixed) connection strength of that path, which, in turn, can be aggregated with other fixed connection strengths. For WM model, a path connection strength is guaranteed to be fixed if none of the intermediate nodes on the path are incident to an option-edge (the weight of which might change).

Storing graph instead of paths. Instead of storing paths one by one, it is more space efficient to store the connection subgraphs. The collection of all L-short simple paths $\mathcal{P}_L(u,v)$ between nodes u and v defines the connection subgraph $\mathcal{G}(u,v)$ between u and v. Storing $\mathcal{G}(u,v)$ is more efficient because in $\mathcal{P}_L(u,v)$ some of the nodes can be repeated several times, whereas in $\mathcal{G}(u,v)$ each node occurs only once. Notice, in all cases, we store only nodes: edges need not be stored since they can be restored from the original graph G. There is a price to pay for storing only $\mathcal{G}(u,v)$: the paths need to be rediscovered. However this rediscovering happens in a small subgraph $\mathcal{G}(u,v)$ instead of the whole graph G.

5.5 Computational complexity of RelDC.

The procedure that computes $c(u,v)$ greedily, provided in Section 5.2, discovers L-short simple $u \rightsquigarrow v$ paths such that it finds paths with the highest connection strength first and with the lowest last. It achieves that by maintaining the current connection strength for intermediate paths and by using a priority queue to retrieve the best (in terms of connection strength) intermediate path to expand next. This procedure has several thresholds that limit the number of nodes it can expand, the total number of edges it can examine, the length of each path, the total number of u-v paths it can discover, and the total number of all paths it can examine. Those thresholds can be specified as constants, or as functions of $|V|$, $|E|$, and L. If they are constants, then the time and space complexity of $c(u,v)$ is limited by constants C_{time} and C_{space}. Assume there are N_{ref} references that need to be disambiguated, typically $N_{ref} = O(|V|)$. The average cardinality of their choice sets is typically a constant, or $O(|V|)$. Thus, Iter-RelDC will call $c(x_i, y_j)$ procedure for at most $O(|V|^2)$ pairs of (x_i, y_j) per iteration. Therefore the time complexity of an iteration of Iter-RelDC is $O(|V|^2)$ multiplied by the complexity of the $c(u,v)$ procedure, plus the cost to construct all choice sets using an FBS approach, which is at most $O(|V|^2)$. Notice, once $c(u,v)$ is computed for one pair, its space can be reused for the next pair. Hence the space complexity is $O(|V| + |E|)$ to store the graph plus the space complexity of one $c(u,v)$ procedure.

6 Experimental Results

In this section we experimentally study RelDC using two real (*publications* and *movies*) and synthetic datasets. RelDC was implemented using C++ and SNOPT solver [2]. The system runs on a 1.7GHz Pentium machine. We test and compare the following implementations of RelDC:

1. **Iter-RelDC** vs. **Solv-RelDC**. The prefixes indicate whether the corresponding NLP problem discussed in Section 4.3 is solved *iteratively* or using a *solver*. If none of those prefixes is specified, Iter-RelDC is assumed by default. Solv-RelDC is applicable only to more restricted problems (e.g., smaller graphs and smaller values of L) than Iter-RelDC. Solv-RelDC is also slower than Iter-RelDC.
2. **WM-RelDC** vs. **PM-RelDC**. The prefixes indicate whether the weight-based model (WM), described in Section 4.1.2, or probabilistic model (PM), described in Appendix A in appendix, has been used for computing connection strengths. By default WM-RelDC is assumed.

3. **DF-RelDC** vs. **GRD-RelDC**. The prefixes specify whether the depth-first (DF) or greedy (GRD) implementation of AllPaths is used. By default DF-RelDC is assumed.

4. Various optimizations of RelDC can be turned on or off. By default, optimizations from Section 5 are on.

In each of the RelDC implementations, the value of L used in computing the L-short simple paths is set to 7 by default. In this section we will demonstrate that WM-DF-Iter-RelDC is on of the best implementations of RelDC in terms of both accuracy and efficiency. That is why the bulk of our experiments use that implementation.

6.1 Case Study 1: the publications dataset

6.1.1 Datasets

In this section we will introduce RealPub and SynPub datasets. Our experiments will solve *author matching (AM)* problem, defined in Section 2, on these datasets.

RealPub dataset. RealPub is a real data set constructed from *two* public-domain sources: CiteSeer[1] and HPSearch[3]. CiteSeer can be viewed as a collection of research publication, HPSearch as a collection of information about authors. HPSearch can be viewed as a set of ⟨`id`, `authorName`, `department`, `organization`⟩ tuples. That is, the affiliation consists of not just organization like in Section 2, but also of department. Information stored in CiteSeer is in the same form as specified in Section 2, that is ⟨`id`, `title`, `authorRef1`,`authorRef2`,..., `authorRefN`⟩ per each paper.

CiteSeer Paper ID	Author Name
51470	Hector Garcia-Molina
51470	Anthony Tomasic
351993	Hector Garcia-Molina
351993	Anthony Tomasic
351993	Luis Gravano
641294	Luis Gravano
641294	Surajit Chaudhuri
641294	Venkatesh Ganti
273193	Venkatesh Ganti
273193	Johannes Gehrke
273193	Raghu Ramakrishnan
273193	Wei-Yin Loh

Table 2: Sample content of the *publication* table derived from CiteSeer.

Author ID	Author Name	Organization	Department
1001	Hector Garcia-Molina	Stanford	cs.stanford
1002	Anthony Tomasic	Stanford	cs.stanford
1003	Luis Gravano	Columbia Univ.	cs.columbia
1004	Surajit Chaudhuri	Microsoft	research.ms
1005	Venkatesh Ganti	Microsoft	research.ms
1006	Johannes Gehrke	Cornell	cs.cornell
1007	Raghu Ramakrishnan	Univ. of Wisconsin	cs.wisc
1008	Wei-Yin Loh	Univ. of Wisconsin	stat.wisc

Table 3: Sample content of the *author* table derived from HPSearch. Author from CiteSeer not found in HPSearch are also added.

Tables 2 and 3 show sample content of two tables derived from CiteSeer and HPSearch based on which the corresponding entity-relationship graph is constructed for RelDC. Figure 11 shows a sample entity-relationship graph that corresponds to the information in those two tables.

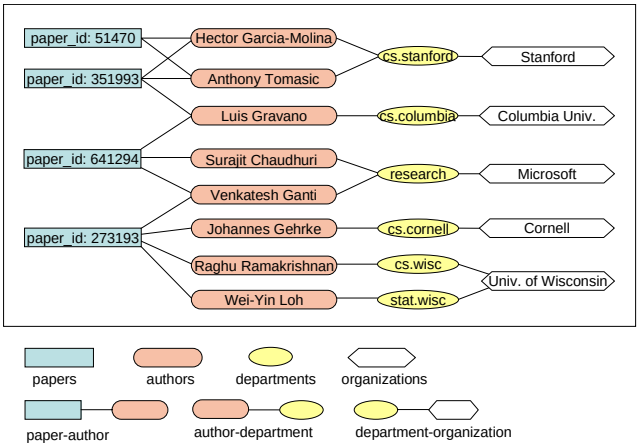

Figure 11: Sample entity-relationship graph for *Publications* dataset. A paper with `paper_id` of, say 51470 can be retrieved from CiteSeer via URL: `http://citeseer.ist.psu.edu/51470.html`

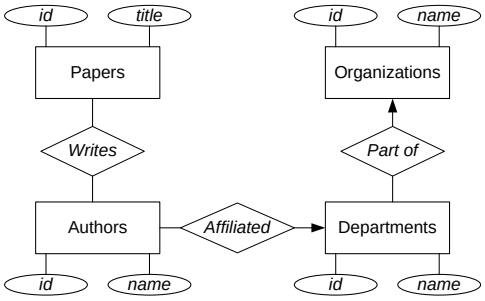

Figure 12: E/R diagram for RealPub

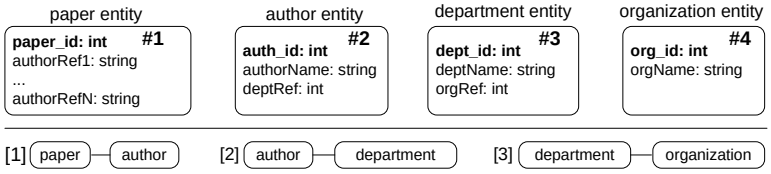

Figure 13: No affiliation in *paper* entities, thus FBS cannot use affiliations.

The various types of entities and relationships present in RealPub are shown in Figure 13. RealPub data consists of 4 *types* of entities: papers (255K), authors (176K), organizations (13K), and departments (25K). To avoid confusion we use "`authorRef`" for author names in *paper* entities and "`authorName`" for author names in *author* entities. There are 573K `authorRef`'s in total. Our experiments on RealPub will explore the efficiency of RelDC in resolving these references.

To test RelDC, we first constructed an entity-relationship graph G for the RealPub database. Each node in the graph corresponds to an entity of one of these types. If author A is affiliated with department D, then there is $(v[A], v[D])$ edge in the graph. If department D is a part of organization U, then there is $(v[D], v[U])$ edge. If paper P is written by author A, then there is $(v[A], v[P])$ edge. For each of the 573K `authorRef` references, feature-based similarity (FBS) was used to construct its choice set.

In the RealPub data set, the paper entities refer to authors using **only their names (and not affiliations)**. This is because the paper entities are derived from the data available from CiteSeer which did not directly contain information about the author's affiliation. As a result, only similarity of author names was used to initially construct the graph G.

This similarity has been used to construct choice sets for all `authorRef` references. As the result, 86.9% (498K) of all `authorRef` references had choice set of size one and the corresponding papers and authors were linked directly. For the remaining 13.1% (75K) references, 75K choice nodes were created in the graph G. RelDC was used to resolve these remaining references. The specific experiments conducted and results will be discussed later in the section. Notice that the RealPub data set allowed us to test RelDC only under the condition that a majority of the references are already correctly resolved. To test robustness

of the technique we tested RelDC over synthetic data sets where we could vary the uncertainty in the references from 0 to 100%.

SynPub dataset. We have created two synthetic datasets SynPub1 and SynPub2, that emulate RealPub. The synthetic data sets were created since, for the RealPub dataset, we do not have the true mapping between papers and the authors of those papers. Without such a mapping, as will become clear when we describe experiments, testing for accuracy of reference disambiguation algorithm requires a manual effort (and hence experiments can only validate the accuracy over small samples). In contrast, since in the synthetic data sets, the *paper-author* mapping is known in advance, accuracy of the approach can be tested over the entire data set. Another advantage of the SynPub dataset is that by varying certain parameters we can manually control the nature of this dataset allowing for the evaluation of all aspects of RelDC under various conditions (e.g., varying level of ambiguity/uncertainty in the data set).

Both the SynPub1 and SynPub2 datasets contain 5000 papers, 1000 authors, 25 organizations and 125 departments. The average number of choice nodes that will be created to disambiguate the `authorRef`'s is 15K (notice, the whole RealPub dataset has 75K choice nodes). The difference between SynPub1 and SynPub2 is that author names are constructed differently: SynPub1 uses unc_1 and SynPub2 uses unc_2 as will be explained shortly.

6.1.2 Accuracy experiments

In our context, *accuracy* is the fraction of all `authorRef` references that are resolved correctly. This definition includes references that have choice sets of cardinality 1.

Experiment 1 (RealPub: manually checking samples for accuracy). Since the correct *paper-author* mapping is not available for RealPub, it is infeasible to test the accuracy on this dataset. However it is possible to find a portion of this *paper-author* mapping *manually* for a sample of RealPub by going to authors web pages and examining their publications.

We have applied RelDC to RealPub in order to test the effectiveness of analyzing relationships. To analyze the accuracy of the result, we concentrated only on the 13.1% of uncertain `authorRef` references. Recall, the cardinality of the choice set of each such reference is at least two. For 8% of those references there were no $x_i \rightsquigarrow y_j$ paths for all j's, thus RelDC used only FBS and not relationships. Since we want to test the effectiveness of analyzing relationships, we remove those 8% of references from further consideration as well. We then chose a random samples of 50 papers that were still left under consideration. For this sample we compared the reference disambiguation result produced by RelDC with the true matches. The true matches for `authorRef` references in those papers were computed manually. In this experiment, RelDC was able to resolve **all** of the 50 sample references correctly! This outcome is in reality not very surprising since in the RealPub data sets, the number of references that were ambiguous was only 13.1%. Our experiments over the synthetic data sets will show that RelDC reaches very high disambiguation accuracy when the number of uncertain references is not very high.

Ideally, we would have liked to have performed further accuracy tests over RealPub by either testing on larger samples (more than 50) and/or repeating the test multiple times (in order to establish confidence levels). However, due to the time-consuming manual nature of this experiments, this was infeasible at the time of writing of the paper. □

Experiment 2 (RealPub: accuracy of identifying author first names). We conducted another experiment over the RealPub data set to test the efficiency of RelDC in disambiguating references which we describe below. We first remove from RealPub all the paper entities which have an `authorRef` in format "*first initial + last name*". This leaves only papers with `authorRef`'s in format "*full first name + last name*". Then we pretend we only know "*first initial + last name*" for those `authorRef`'s. Next we run

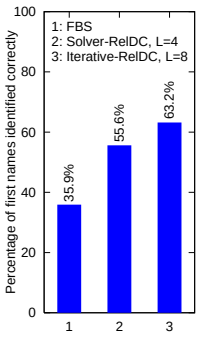

Figure 14: RealPub: Identifying first names

FBS and RelDC and see if they would identify correctly an author with the right full first name. In this experiment, for 82% of the `authorRef`'s the cardinality of their choice sets is 1 and there is nothing to resolve. For the rest 18% the problem is more interesting: the cardinality of their choice sets is at least 2. Figure 22(a) shows the outcome for those 18%.

Notice that the reference disambiguation problem tested in the above experiment is of a limited nature – the tasks of identifying the correct first name of the author and the correct author are not the same in general.[14] Nevertheless, the experiment allows us to test the accuracy of RelDC over the entire database and does show the strength of the approach. □

Accuracy on SynPub. The next set of experiments tests accuracy of RelDC and FBS approaches on SynPub dataset. "RelDC 100%" ("RelDC 80%") means for 100% (80%) of *author* entities the affiliation information is available. Once again, *paper* entities do not have author affiliation attributes, so FBS cannot use affiliation, see Figure 13. Thus those 100% and 80% have no effect on the outcome of FBS. Notation "L=4" means RelDC explores paths of length no greater than 4.

Experiment 3 (Accuracy on SynPub1). SynPub1 uses *uncertainty of type 1* defined as follows.

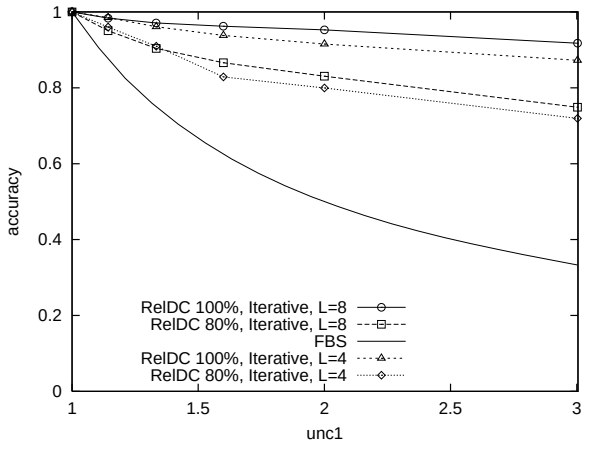

Figure 15: SynPub1: Accuracy vs. unc_1

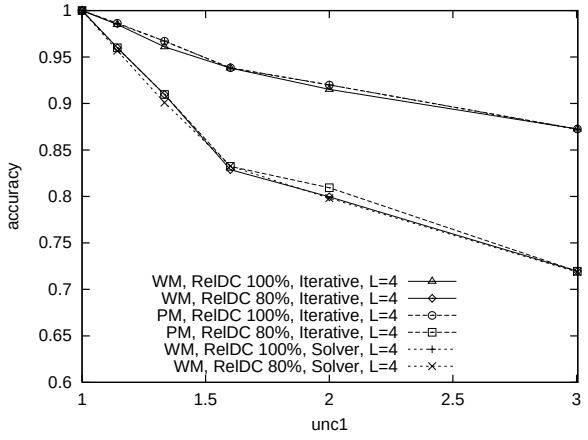

Figure 16: SynPub1: The accuracy results for Solv-RelDC, Iter-RelDC, and Iter-RelDC with PM model are comparable.

[14] That is, it is not enough to correctly identify that 'J.' in 'J. Smith' corresponds to 'John' if there are multiple 'John Smith"s in the dataset.

There are $N_{auth} = 1000$ unique authors in SynPub1. But there are only $N_{name} \in [1, N_{auth}]$ unique `authorName`'s. We construct the `authorName` of the author with ID of k ($k = 0, 1, \ldots, 999$) as "name" concatenated with (k mod N_{name}). Each `authorRef` specifies one of those `authorName`'s. Parameter unc_1 is $unc_1 = \frac{N_{auth}}{N_{name}}$ ratio. For instance, if N_{name} is 750, then the authors with IDs of 1 and 751 have the same `authorName`: "name1", and $unc_1 = \frac{1000}{750} = 1\frac{1}{3}$. In SynPub1 for each author whose name is not unique, one can never identify with 100% confidence any paper this author has written. Thus the uncertainty for such authors is very high.

Figure 15 studies the effect of unc_1 on accuracy of RelDC and FBS. If $unc_1 = 1.0$, then there is no uncertainty and all methods have accuracy of 1.0. As expected, the accuracy of all methods monotonically decreases as uncertainty increases. If $unc_1 = 2.0$, the uncertainty is very large: for any given author there is exactly one another author with the identical `authorName`. For this case, any FBS have no choice but to guess one of the two authors. Therefore, the accuracy of any FBS, as shown in Figures 15, is 0.5. However, the accuracy of RelDC 100% (RelDC 80%) when $unc_1 = 2.0$ is 94%(82%). The gap between RelDC 100% and RelDC 80% curves shows that in SynPub1 RelDC relies substantially on author affiliations for the disambiguation.

Comparing the RelDC implementations. Figure 16 shows that the accuracy results of WM-Iter-RelDC, PM-Iter-RelDC, WM-Solv-RelDC implementations are comparable. Figure 17 shows that Iter-RelDC is the fastest implementation among them. The same trend has been observed for all other tested cases. □

Experiment 4 (Accuracy on SynPub2). SynPub2 uses *uncertainty of type 2*. In SynPub2, `authorName`'s

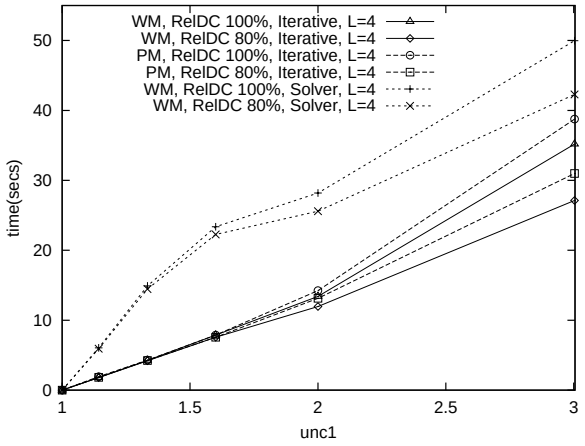

Figure 17: SynPub1: Iter-RelDC is more efficient than (i)Solv-RelDC and (ii)Iter-RelDC with PM model.

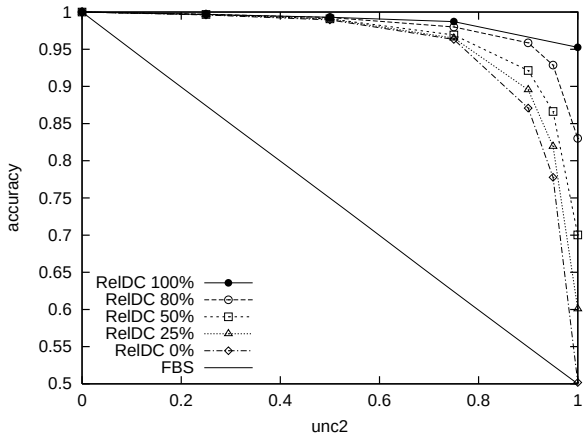

Figure 18: SynPub2: Accuracy vs. unc_2

(in *author* entities) are constructed such that the following holds, see Figure 13. If an `authorRef` reference (in a *paper* entity) is in the format "*first name + last name*" then it matches only one (correct) author. But if it is in the format "*first initial + last name*" it matches exactly two authors. Parameter unc_2 is the fraction of `authorRef`'s specified as "*first initial + last name*". If $unc_2 = 0$, then there is no uncertainty and the accuracy of all methods is 1. Also notice that the case when $unc_2 = 1.0$ is equivalent to $unc_1 = 2.0$.

There is less uncertainty in Experiment 4 then in Experiment 3. This is because for each author there is a chance that he is referenced to by his full name in some of his papers, so for these cases the *paper-author* associations are known with 100% confidence.

Figure 18 shows the effect of unc_2 on the accuracy of RelDC. As in Figure 15, in Figure 18 the accuracy decreases as uncertainty increases. However this time the accuracy of RelDC is much higher. The fact that curves for RelDC 100% and RelDC 80% are almost indiscernible until unc_2 reaches 0.5, shows that

RelDC relies less heavily on weak author affiliation relationships but rather on stronger connections via papers. □

6.1.3 Other experiments

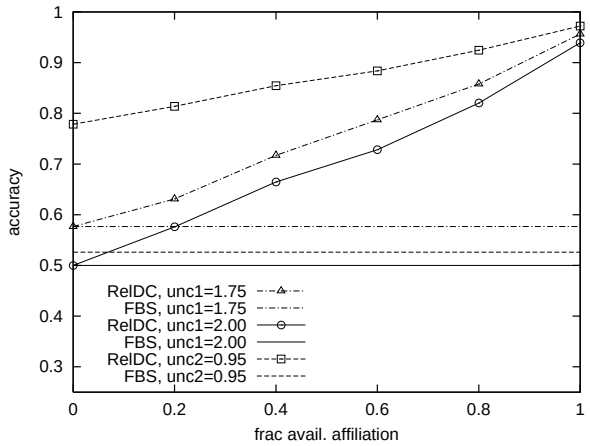

Figure 19: SynPub: Accuracy vs. frac avail. affiliation

Experiment 5 (Importance of relationships). Figure 19 studies what effect the number of relationships and the number of relationship *types* have on the accuracy of RelDC. When resolving `authorRef`'s, RelDC uses three types of relationships: (1) *paper-author*, (2) *author-department*, (3) *department-organization*.[15] The affiliation relationships (i.e., (2) and (3)) are derived from the affiliation information in *author* entities.

The affiliation information is not always available for each *author* entity in RealPub. In our synthetic datasets we can manually vary the amount of available affiliation information. The x-axis shows the fraction ρ of *author* entities for which their affiliation is known. If $\rho = 0$, then the affiliation relationships are eliminated completely and RelDC has to rely solely on connections via *paper-author* relationships. If $\rho = 1$, then the complete knowledge of author affiliations is available. Figure 19 studies the effect of ρ on accuracy. The curves in this figure are for both SynPub1 and SynPub2: $unc_1 = 1.75$, $unc_1 = 2.00$, and $unc_2 = 0.95$. The accuracy increases as ρ increases showing that RelDC deals with newly available relationships well. □

Experiment 6 (Longer paths). Figure 20 examines the effect of path limit parameter L on the accuracy. For all the curves in the figure, the accuracy monotonically increases as L increases with the only one exception for "RelDC 100%, unc1=2" and $L = 8$. The usefulness of longer paths depends on the combination of other parameters. For SynPub, L of 7 is a reasonable compromise between accuracy and efficiency. □

Experiment 7 (The neighborhood optimization). We have developed several optimizations which make RelDC 1–2 orders of magnitude more efficient. Figure 22 shows the effect of one of those optimizations, called NBH (see Section 5.3), for subsets of 11K and 22K papers of CiteSeer. In this figure, the radius of neighborhood is varied from 0 to 8. The radius of zero corresponds to the case where NBH is not used. Figure 23 shows the speedup achieved by NBH optimization with respect to the case when NBH is off. The figure shows another positive aspect of NBH optimization: the speed up grows as the size of the dataset and L increase. □

[15] Note, there is a difference between a *type of relationship* and a *chain of relationships*: e.g. RelDC can discover paths like: paper1-author1-dept1-org1-dept2-author2.

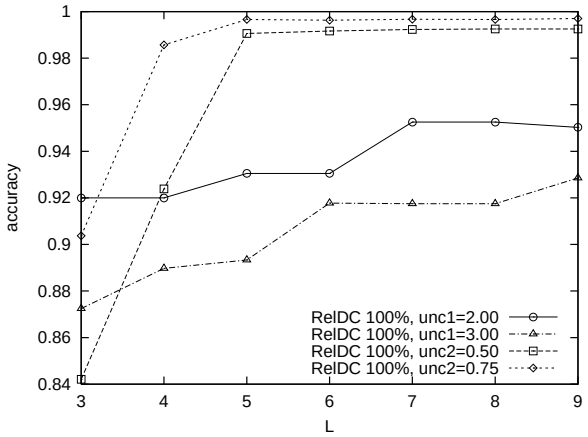

Figure 20: SynPub: Accuracy vs. path length limit

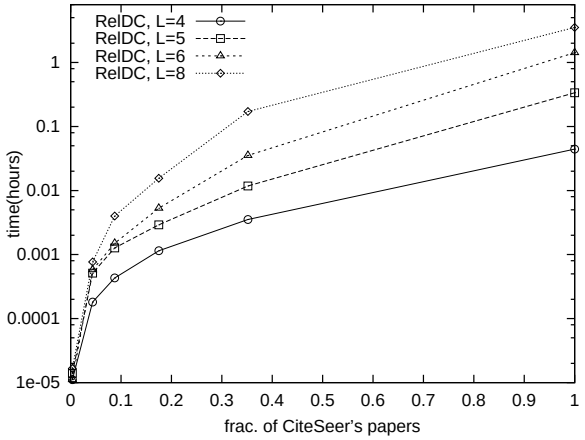

Figure 21: RealPub: Time vs. frac of RealPub's papers

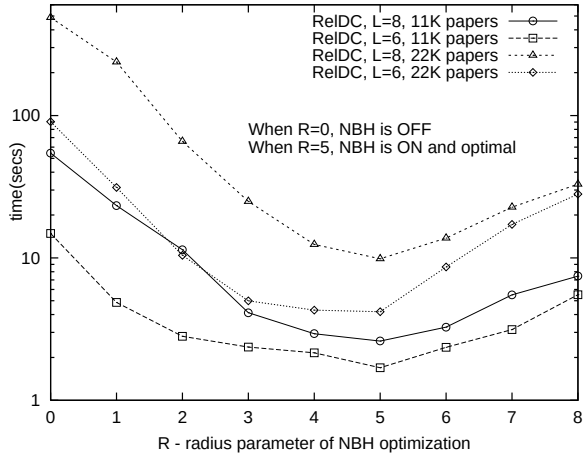

Figure 22: RealPub: Optimizations are crucial.

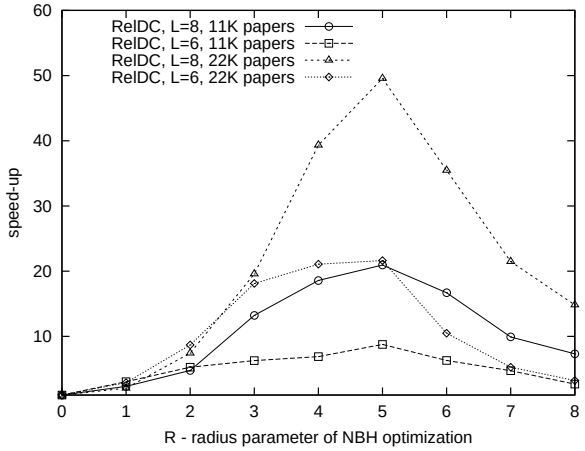

Figure 23: RealPub: Speed-up achieved by NBH optimization.

Experiment 12 that studies the path coloring optimizations is presented in Section B.2.3.

Experiment 8 (Efficiency of RelDC). To show the applicability of RelDC to a large dataset we have successfully applied it to clean RealPub with L ranging from 2 up to 8. Figure 21 shows the execution time of RelDC as a function of the fraction of papers from RealPub dataset, e.g. 1.0 corresponds to all papers in RealPub (the whole CiteSeer) dataset. □

Experiment 9 (Greedy vs. Depth-first AllPaths implementations). This experiment compares accuracy and performance of greedy and depth-first versions of RelDC. As the name suggests, the depth-first version discovers exhaustively *all* paths in a depth-first fashion. RelDC has been heavily optimized and this discovery process is very efficient. The greedy implementation of AllPaths discovers paths with the best connection strength first and with the worst last. This gives an opportunity to fine-tune in a meaningful way when to stop the algorithm by using various thresholds. Those thresholds can limit, for example, not only path length but also the memory that all intermediate paths can occupy, the total number of paths that can be analyzed and so on.

Figures 24 and 25 study the effect of N_{exp} parameter on the accuracy and efficiency of GRD-RelDC

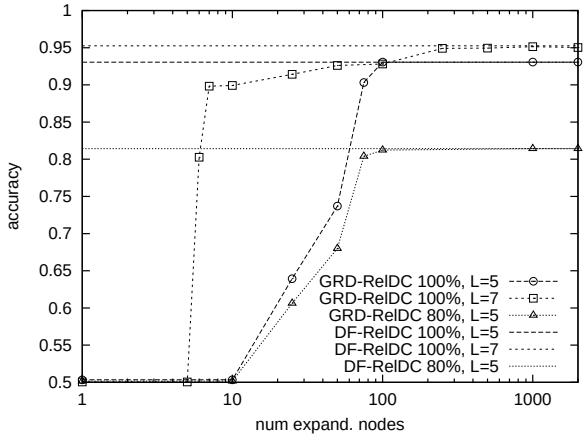

Figure 24: SynPub1: $unc_1 = 2$. Accuracy of GRD-RelDC vs. DF-RelDC.

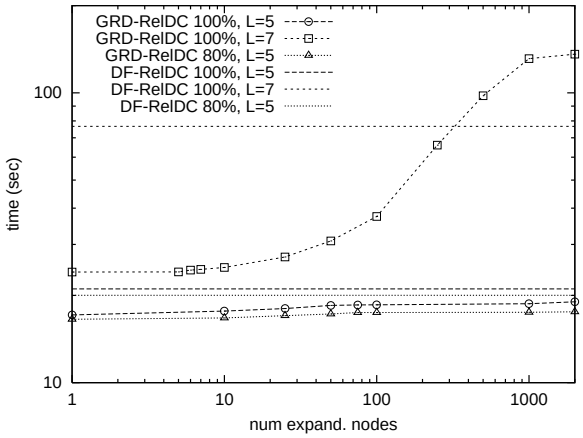

Figure 25: SynPub1: $unc_1 = 2$. Time of GRD-RelDC vs. DF-RelDC.

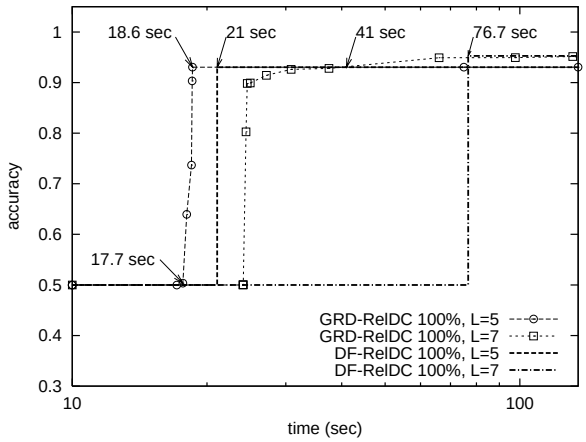

Figure 26: SynPub1: $unc_1 = 2$. Choosing the best among GRD-RelDC $L = 5$, GRD-RelDC $L = 7$, DF-RelDC $L = 5$, DF-RelDC $L = 7$ at each moment in time.

and DF-RelDC. Parameter N_{exp} is the *upper bound* on the number of paths that can be extracted from the priority queue for GRD-RelDC. The AllPaths part of GRD-RelDC stops if either N_{exp} is exceeded or the priority queue is empty.

The series in the experiment are obtained by varying: (1) DF vs. GRD, (2) path length limit $L = 5$ and $L = 7$ and (3) the amount of affiliation information 100% and 80%. Since DF-RelDC does not use N_{exp} parameter, all DF-RelDC curves are flat. Let us analyze what behavior is expected from GRD-RelDC and then see if the figures corroborate it. We will always assume that **path length is limited** for both DF-RelDC and GRD-RelDC.

If N_{exp} is small then GRD-RelDC should discover only a few paths and its accuracy should be close to that of FBS. If N_{exp} is sufficiently large, then GRD-RelDC should discover the same paths as DF-RelDC. That is, we can compute $m_{ij} = |\mathcal{P}_L(x_i, y_j)|$, where $|\mathcal{P}_L(x_i, y_j)|$ is the number of paths in $\mathcal{P}_L(x_i, y_j)$. Then if we choose N_{exp} such that $N_{exp} \geq m$, where $m = \max_{i,j}(m_{ij})$, then the set of all paths that GRD-RelDC will discover will be identical to that of DF-RelDC. So, the accuracy of GRD-RelDC should stabilize and be equal to that of DF-RelDC. The execution time of GRD-RelDC should stabilize as well and, at that point, be larger than the execution time of DF-RelDC. Thus the accuracy of GRD-RelDC is expected to

increase monotonically and then stabilize as N_{exp} increases. The execution time of GRD-RelDC is expected to behave the same way.

The curves in Figures 24 and 25 behave as expected except for one surprise: when $L = 5$, GRD-RelDC is actually faster than DF-RelDC. It is explained by the fact that when $L = 5$, NBH optimization prunes very effectively many paths. That keeps the priority queue small. Thus the performance of DF-RelDC and GRD-RelDC becomes comparable. Notice, in all of the experiments NBH optimization was turned on, because the efficiency of any implementation of RelDC with NBH off is substantially worse than the efficiency of any implementation with NBH on.

Figure 26 combines Figures 24 and 25. It plots the achieved accuracy by DF- and GRD-RelDC 100% when $L = 5$ and $L = 7$ as a function of time. Using this figure it is possible to perform a retrospective analysis of which implementation has shown the best accuracy when allowed to spend only at most certain amount of time t on the cleaning task. For example, in time interval $[0, 17.7)$ RelDC cannot achieve better accuracy than FBS, so it would be more efficient to just use FBS. In time interval $[17.7, 18.6)$ it is better to use GRD-RelDC with L set to 5. If one is allowed to spend only $[18.6, 41)$ seconds, it is better to use GRD-RelDC with L set to 5 for only 18.6 seconds. If you intend to spend between 41 and 76.7 seconds it is better to use GRD-RelDC with L set to 7. If you can spend 76.7 seconds or more, it is better to run DF-RelDC with L set to 7 which will terminate in 76.7 seconds. □

6.2 Case Study 2: the movies dataset

6.2.1 Dataset

RealMov is a real public-domain movies dataset described in [38] which has been made popular by the textbook [16]. Unlike RealPub dataset, in RealMov all the needed correct mappings are known, so it is possible to test the disambiguation accuracy of various approaches more extensively. However, RealMov dataset is much smaller compared to the RealPub data set. RealMov contains entities of three types: *movies* $(11,453$ entities), *studios* (992 entities), and *people* $(22,121$ entities). There are five types of relationships in the RealMov dataset: *actors*, *directors*, *producers*, *producingStudios*, and *distributingStudios*. Relationships *actors*, *directors*, and *producers* map entities of type *movies* to entities of type *people*. Relationships *producingStudios* and *distributingStudios* map *movies* to *studios*.

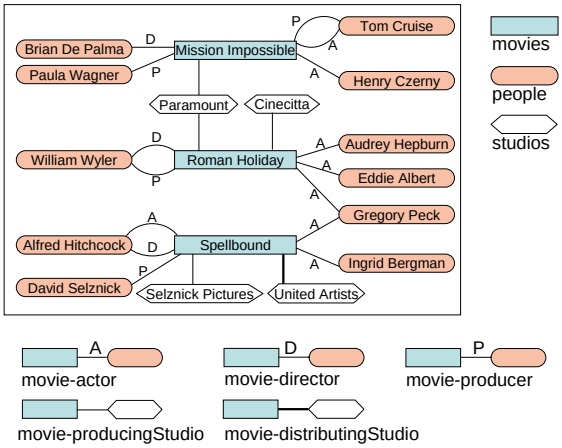

Figure 27: Sample entity-relationship graph for *movies* dataset.

Figure 27 presents a sample graph for RealMov dataset. Tables 4, 5, 6, and 7 demonstrate sample content of the *people*, *movies*, *studios* and *cast* tables derived from the movies dataset. The sample graph in Figure 27 is constructed from those tables.

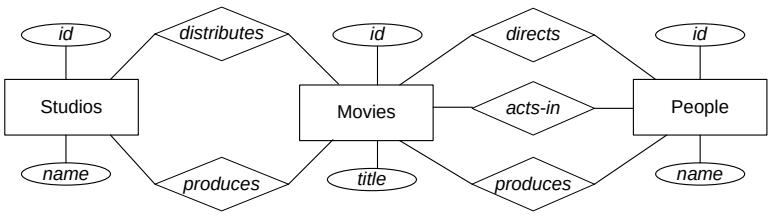

Figure 28: E/R diagram for RealMov.

Stage name	DOW	Name at birth	Gen	DOB	Role	Orig	Notes
Tom Cruise	1981-1989	Thomas Cruise Mapother IV	M	1962	Hero	Am	
B.dePalma	1968-1987	Brian De Palma	M				
Paula Wagner			F			Am	
Henry Czerny			M	1959		Ca	
Wyler	1925-1959	William Wyler		1902		Am	Or(Ge)
Audrey Hepburn	1951-1981	Audrey Hepburn-Ruston	F	1929	pert	Be	
Eddie Albert	1938-1982	Eddie Albert Heimberger	M	1908	honest Joe	Am	
Gregory Peck	1943-1982	Gregory Peck	M	1916	likeable	Am	
Ingrid Bergman	1934-1978	Ingrid Bergman	F	1915	strong beauty	Sw	
Hitchcock	1925-1976	Alfred Hitchcock		1899		Br	Ty(Susp , Nior)
Selznick		David O. Selznick		1902		Ru	Ww(Hitchcock)

Table 4: The *people* table. Some of the notation used: DOW (dates of work), Gen (gender), DOB (date of birth), type (kinds of roles actor played), orig (origin), Am (America).

ID	Title	Year	Director	Producer	Studio	Color	Genre
BdP30	Mission Impossible	1996	B.dePalma	Tom Cruise, Paula Wagner	Paramount	col	Action
WW67	Roman Holiday	1953	Wyler	Wyler	Cinecitta, Paramount	bnw	Romantic
H42	Spellbound	1945	Hitchcock	Selznick	Selznick Pictures	bnw	Suspect

Table 5: The *movies* table.

Name	Full name	City	Country	First	Last	Founder	Successor
Paramount	Paramount Corp.	Los Angeles	USA	1916	1993	W. Hodkinson	Paramount, Viacom
Cinecitta		Rome	Italy	1937			
Selznick	Selznick Pictures	Hollywood	USA	1936	1944		
U.A.	United Artists	Hollywood	USA	1919	1983	Chaplin, Pickford, etc.	MGM-UA

Table 6: The *studios* table.

Movie ID	Actor name
BdP30	Tom Cruise
BdP30	B.dePalma
BdP30	Paula Wagner
BdP30	Henry Czerny
WW67	Wyler
WW67	Audrey Hepburn
WW67	Eddie Albert
WW67	Gregory Peck
H42	Gregory Peck
H42	Ingrid Bergman
H42	Hitchcock
H42	Selznick

Table 7: The *cast* table.

6.2.2 Accuracy experiments

Experiment 10 (RealMov: Accuracy of disambiguating *director* references). In this experiment we study the accuracy of disambiguating references from movies to directors of those movies.

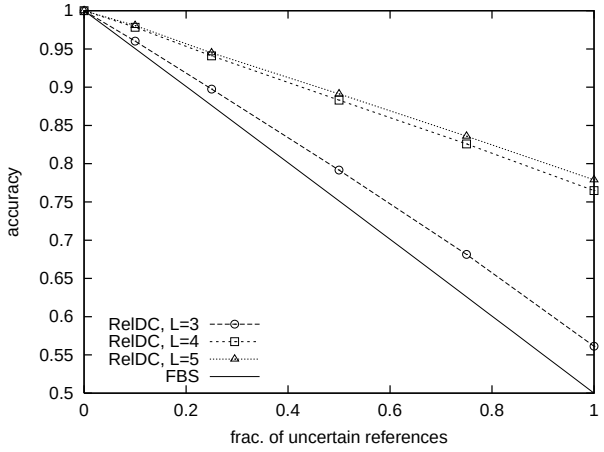

Figure 29: RealMov: disambiguating *director* references. The size of the choice set of each *uncertain* reference is 2.

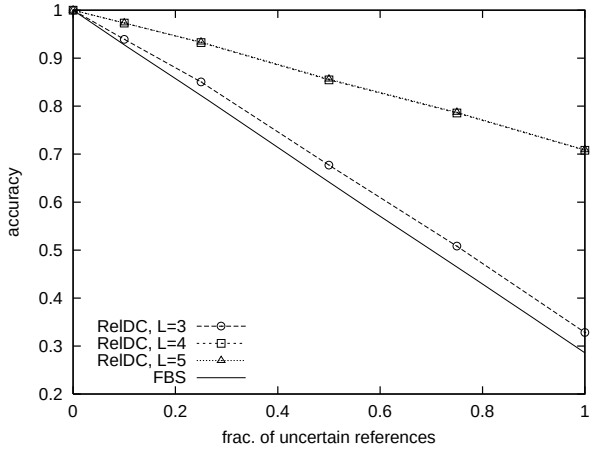

Figure 30: RealMov: disambiguating *director* references. The pmf of sizes of choice sets of *uncertain* references is given in Figure 31.

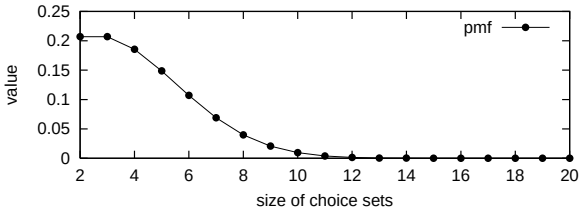

Figure 31: PMF of sizes of choice sets.

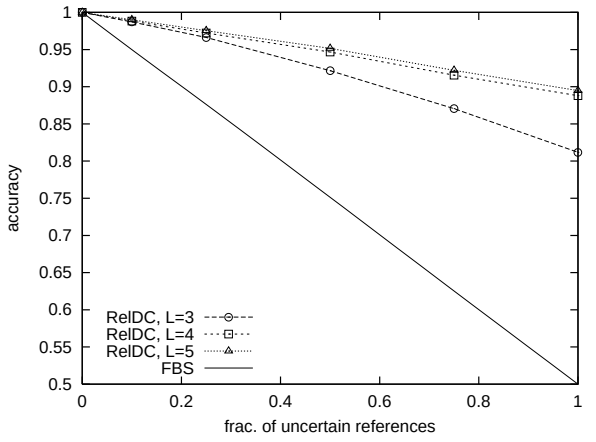

Figure 32: RealMov: disambiguating *studio* references. The size of the choice set of each *uncertain* reference is 2.

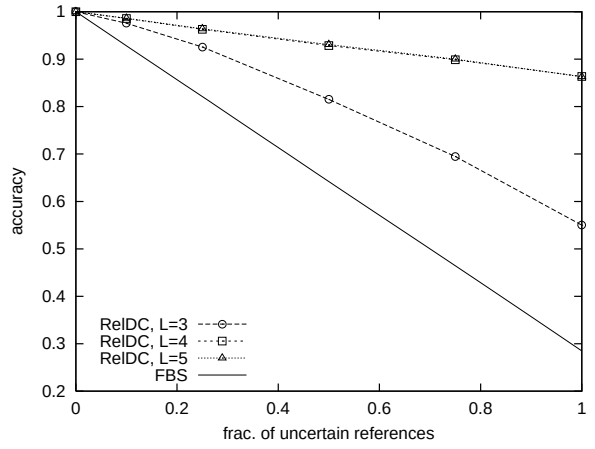

Figure 33: RealMov: disambiguating *studio* references. The pmf of sizes of choice sets of *uncertain* references is given in Figure 31.

Since in RealMov each reference, including each *director* reference, already points directly to the right match, we artificially introduce ambiguity in the references manually. Similar approach to testing data cleaning algorithms have also been used by other researchers, e.g. [7]. Given the specifics of our problem, to study the accuracy of RelDC we will simulate that we used FBS to determine the choice set of each reference but FBS was uncertain in some of the cases.

To achieve that, we first choose a fraction ρ of *director* references (that will be uncertain). For each reference in this fraction we will simulate that FBS part of RelDC has done its best but still was uncertain as follows. Each *director* reference from this fraction is assigned a choice set of N people. One of those people is the true director, the rest $(N-1)$ are chosen randomly from the set of *people* entities.

Figure 30 studies the accuracy as ρ is varied from 0 to 1 and where N is distributed according to the probability mass function (pmf) shown in Figure 31, see [24] for detail. Figure 29 is similar to Figure 30 but N is always 2. The figures show that RelDC achieves better accuracy than FBS. The accuracy is 1.0 when $\rho = 0$, since all references are linked directly. The accuracy decreases almost linearly as ρ increases to 1. When $\rho = 1$, the cardinality of the choice set of each reference is at least 2. The larger the value of L, the better the results. The accuracy of RelDC improves significantly as L increases from 3 to 4. However, the improvement is less significant as L increases from 4 to 5. Thus the analyst must decide whether to spend more time to obtain higher accuracy with $L = 5$, or whether $L = 4$ is sufficient. □

Experiment 11 (RealMov: Accuracy of disambiguating *studio* references). This experiment is similar to the previous Experiment 10, but now we disambiguate *producingStudio* references, instead of *director* references. Figure 32 corresponds to Figure 29 and Figure 33 to Figure 30. The RelDC's accuracy of disambiguating *studio* references is even higher. □

7 Related Work

Many research challenges have been explored in the context of data cleaning in the literature: dealing with missing data, handling erroneous data, record linkage, and so on. The closest to the problem of reference disambiguation addressed in this paper is the problem of record linkage. The importance of record linkage is underscored by the large number of companies, such as Trillium, Vality, FirstLogic, DataFlux, which have developed (domain-dependent) record linkage solutions.

Researchers have also explored domain-independent techniques [31, 15, 20, 35, 5, 28]. Their work can be viewed as addressing two challenges: (1) improving similarity function, as in [6, 23]; and (2) improving efficiency of linkage, as in [7]. Typically two-level similarity functions are employed to compare two records. First, such a function computes attribute-level similarities by comparing values in the same attributes of two records. Next the function combines the attribute-level similarity measures to compute the overall similarity of two records. A recent trend has been to employ machine learning techniques, e.g. SVM, to learn the best similarity function for a given domain [6]. Many techniques have been proposed to address the efficiency challenge as well: techniques that use specialized data structures (e.g. specialized indexes [7]), merge/purge, sorting and window based techniques.

Those domain-independent techniques deal only with attributes. The only work we are aware of that can be viewed as using relationships for data cleaning is [5, 26]. In [5] Ananthakrishna et al. employs co-occurrence similarity function to compare contextual attributes for a case with hierarchical relationships. In [26] Lee et al. develop a association-rules mining based method where similarity of records is determined by similarity of the context attributes. Also, in DKDM04, Getoor et al. proposes a method that solves the author matching problem using the notion of context and clustering techniques. The method requires "seeding" of clusters which makes it of limited applicability in practice. To summarize, previous approaches to data cleaning that explored relationships have either addressed other data cleaning problems (not reference disambiguation) or have only considered specific types of relationships (e.g. hierarchy). None of those approaches *discovers* relationship chains and their analysis is limited to directly linked entities. RelDC is the first data cleaning framework that employs systematic relationship analysis for the purpose of cleaning.

8 Conclusion

In this paper we have shown that analysis of inter-object relationships is important for data cleaning and demonstrated one approach that utilizes relationships. As future work we plan to apply similar techniques to the problem of record linkage and to develop an approach which, given a sample resolved graph, would automatically determine which relationships are irrelevant for a particular disambiguation task.

References

[1] CiteSeer. `http://citeseer.nj.nec.com/cs`.

[2] GAMS/SNOPT solver. `http://www.gams.com/solvers/`.

[3] HomePageSearch. `http://hpsearch.uni-trier.de`.

[4] Knowledge Discovery. `http://www.kdnuggets.com/polls/2003/data_preparation.htm`.

[5] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proc. VLDB*, 2002.

[6] M. Bilenko and R. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, 2003.

[7] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. of ACM SIGMOD Conf.*, 2003.

[8] P. Christen, T. Churches, and J. X. Zhu. Probabilistic name and address cleaning and standardization. The Australasian Data Mining Workshop, 2002.

[9] W. Cohen, H. Kautz, and D. McAllester. Hardening soft information sources. In *Proc. of ACM SIGKDD Conf.*, 2000.

[10] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. of ACM SIGMOD Conf.*, 1998.

[11] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. IIWeb Workshop, 2003.

[12] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proc. of ACM SIGKDD Conf.*, 2002.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2001.

[14] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *Proc. of SIGKDD*, 2004.

[15] I. Fellegi and A. Sunter. A theory for record linkage. *Journal of Amer. Statistical Association*, 64(328):1183–1210, 1969.

[16] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems: the complete book*. Prentice Hall, 2002.

[17] L. Getoor. Multi-relational data mining using probabilistic relational models: research summary. In *Proceedings of the First Workshop in Multi-relational Data Mining*, 2001.

[18] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proc. of VLDB Conf.*, 2001.

[19] G. Grimmett and D. Stirzaker. *Probability and random processes.* OXFORD University Press, 2002.

[20] M. Hernandez and S. Stolfo. The merge/purge prob- lem for large databases. In *Proc. of SIGMOD*, 1995.

[21] M. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406), 1989.

[22] M. Jaro. Probabilistic linkage of large public health data files. *Statistics in Medicine*, 14(5-7), Mar-Apr 1995.

[23] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *Proc. of DASFAA Conf.*, 2003.

[24] D. Kalashnikov and S. Mehrotra. An algorithm for entity disambiguation. Univ. of Calif., Irvine, TR-RESCUE-04-10.

[25] e. a. L. De Raedt. Three companions for data mining in first order logic. In *Dzeroski, S. and Lavrac, N., ed. Relational Data Mining.* Springer-Verlag, 2001.

[26] M. Lee, W. Hsu, and V. Kothari. Cleaning the spurious links in data. *IEEE Intelligent Systems*, Mar-Apr 2004.

[27] M. Lee, H. Lu, T. Ling, and Y. Ko. Cleansing data for mining and warehouse. In *Proc. of DEXA Conf.*, 1999.

[28] A. K. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. of ACM SIGKDD Conf.*, 2000.

[29] A. E. Monge and C. Elkan. The field matching problem: Algorithms and applications. In *Proc. of SIGKDD Conf.*, 1996.

[30] A. E. Monge and C. P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proc. of SIGMOD Wshp. on Research Issues on Data Mining and Knowledge Discovery*, 1997.

[31] H. Newcombe, J. Kennedy, S. Axford, and A. James. Automatic linkage of vital records. *Science*, 130:954–959, 1959.

[32] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. Identity uncertainty and citation matching. In Advances in Neural Processing Systems 15, 2002.

[33] E. Ristad and P. Yianilos. Learning string edit distance. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20(5):522–532, May 1998.

[34] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of ACM SIGKDD Conf.*, 2002.

[35] S. Tejada, C. A. Knoblock, and S. Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *Proc. of ACM SIGKDD Conf.*, 2002.

[36] V. Verykios, G.V.Moustakides, and M. Elfeky. A bayesian decision model for cost optimal record matching. *The VLDB Journal*, 12:28–40, 2003.

[37] S. White and P. Smyth. Algorithms for estimating relative importance in networks. In *Proc. of ACM SIGKDD Conf.*, 2003.

[38] G. Wiederhold. The movies dataset. `http://www-db.stanford.edu/pub/movies/doc.html`.

[39] W. Winkler. The state of record linkage and current research problems. In *U.S. Bureau of Census, TR99*.

[40] W. E. Winkler. Advanced methods for record linkage. In *U.S. Bureau of Census*, 1994.

Appendix

A Probabilistic model for computing connection strength

In Section 4.1.2 we have presented the weight based model (WM) for computing connection strength. In this section we study a different connection strength model, called the *probabilistic model (PM)*. In the probabilistic model an edge weight is treated not as "weights" but as "probabilities" that the edge exists.

Notation	Meaning
$x^{\exists}$	event "x exists" for (edge,path) x
$x^{\nexists}$	event "x does not exist" for (edge,path) x
$x^{\rightarrow}$	event corresponding to following (edge,path) x
$dep(e_1, e_2)$	if events e_1 and e_2 are independent, then $dep(e_1, e_2) =$ `true`, else $dep(e_1, e_2) =$ `false`
$\mathrm{P}(x^{\exists})$	probability that (edge,path) x exists
$\mathrm{P}(x^{\rightarrow})$	probability to follow (edge,path) x
$\mathscr{P}$	the path being considered
v_i	ith node on path $\mathscr{P}$
E_i	(v_i, v_{i+1}) edge on path $\mathscr{P}$
E_{ij}	edge labeled with probability p_{ij}
a_{ij}	$a_{ij} = 1$ iff. $E_{ij}^{\exists}$; and $a_{ij} = 0$ iff. $\not{E}_{ij}^{\exists}$
$a_{i0} = 1$	dummy variables: $a_{i0} = 1$ for any i
$p_{i0} = 1$	dummy variables: $p_{i0} = 1$ for any i
$opt(E)$	if edge E is an option-edge, then $opt(E) =$ `true`, else $opt(E) =$ `false`
$cho[E]$	if edge E is an option-edge, then $cho[E]$ denotes the choice node associated with E
a, as a vector	$\mathbf{a} = (a_{10}, a_{11}, \ldots, a_{kn_k})$
a, as a set	$\mathbf{a} = \{a_{ij} : i = 1, 2, \ldots, k; j = 1, 2, \ldots, n_i\}$
a, as a variable	at each moment variable **a** is one instantiation of **a** as a vector

Table 8: Probabilistic model: Terminology

A.1 Preliminaries

Notation. Let us introduce notation that we will use in this section, see Table 8. We will operate with probabilities of certain events. Notation $\mathrm{P}(A)$ refers to the probability of event A to occur. We use $E^{\exists}$ to denote event "E exists" for edge E. Similarly, we use $E^{\exists}$ for event "E does not exist". So, $\mathrm{P}(E^{\exists})$

refers to the probability that E exists. We will consider situations where the algorithm decides what is the probability to follow a specific edge E, usually in the context of a specific path. This probability is denoted as $\mathrm{P}(E^{\rightarrow})$, where $E^{\rightarrow}$ denote the event of going via edge E. Notation $\mathscr{P}$ denotes the path being considered. To discuss dependence of events e_1 and e_2 we will use $dep(e_1, e_2)$ functions. Function $dep(e_1, e_2)$ returns true if two events are dependent and false if they are independent.

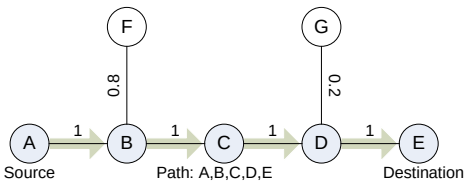

Figure 34: Toy example: independent case

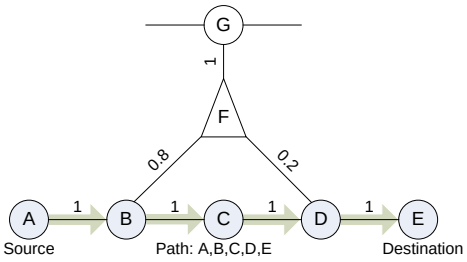

Figure 35: Toy example: dependent case

Introductory examples. We will introduce PM by analyzing two examples shown in Figures 34 and 35. Let us consider how to compute the connection strength when edge weights are treated as probabilities that those edges exist. Each figure show a part of a small sample graph with path $\mathscr{P}$=A→B→C→D→E which will be of interest to us.

In Figure 34 we assume the events "edge BF is present" and "edge DG is present" are *independent*. The probability of the event "edge BF is present" is 0.8. The probability of the event "edge DG is present" is 0.2. In Figure 35 node F represents a choice created to resolve a reference $x_i.r_k$ of entity x_i represented by node G. Nodes B and D are options of choice F. That is, semantically, $x_i.r_k$ can correspond to only one: either B with probability 0.8 or D with probability of 0.2. Events "edge BF exists" and "edge DF exists" are mutually exclusive (and hence strongly *dependent*): if one edge is present the other edge must be absent due to the semantics of the choice node.

PM computes the connection strength of a path as 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 $\mathscr{P}$ should exist in the first place, which means each of its edges should exist. Thus the first step is to compute the probability $\mathrm{P}(\mathscr{P}^{\exists})$ that path $\mathscr{P}$ exists. For the path $\mathscr{P}$ in Figures 34 and 35, probability $\mathrm{P}(\mathscr{P}^{\exists})$ is equal to $\mathrm{P}(AB^{\exists} \cap BC^{\exists} \cap CD^{\exists} \cap DE^{\exists})$. If the existence of each edge in the path is independent from the existence of other edges, e.g. like for the cases shown in Figures 34 and 35, then $\mathrm{P}(AB^{\exists} \cap BC^{\exists} \cap CD^{\exists} \cap DE^{\exists}) = \mathrm{P}(AB^{\exists}) \cdot \mathrm{P}(BC^{\exists}) \cdot \mathrm{P}(CD^{\exists}) \cdot \mathrm{P}(DE^{\exists})$. Since all of the edges of path $\mathscr{P}$ are labeled with 1's in both figures, the probability that $\mathscr{P}$ exists is 1. Now the second step is to consider the probability $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists})$ to follow path $\mathscr{P}$, given that $\mathscr{P}$ exists. Once this probability is computed, it is easy to compute our goal – the probability to follow path $\mathscr{P}$, which we use as our measure of the connection strength of path $\mathscr{P}$: $c(\mathscr{P}) = \mathrm{P}(\mathscr{P}^{\rightarrow}) = \mathrm{P}(\mathscr{P}^{\exists}) \cdot \mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists})$. The probability $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists})$ is computed differently for the cases in Figures 34 and 35. This will lead to different values for the connection strength of $\mathscr{P}$.

Example A.1.1 (Independent edge existence). Let us first consider the case where the existence of each edge is independent from the existence of the other edges. In Figure 34 two events "BF exists" and "DG exists" are independent. The probability to follow path $\mathscr{P}$ is the product of probabilities to follow each of the edges on the path: $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists}) = \mathrm{P}(AB^{\rightarrow}|\mathscr{P}^{\exists}) \cdot \mathrm{P}(BC^{\rightarrow}|\mathscr{P}^{\exists}) \cdot \mathrm{P}(CD^{\rightarrow}|\mathscr{P}^{\exists}) \cdot \mathrm{P}(DE^{\rightarrow}|\mathscr{P}^{\exists})$. Given path $\mathscr{P}$ exists, the probability to follow edge AB in path $\mathscr{P}$ is one. The probability to follow edge BC is computed as follows. With probability 0.2 edge BF is absent, in which case the probability to

follow BC is 1. With probability 0.8 edge BF is present, in which case the probability to follow BC is $\frac{1}{2}$ – because there are two links, BF and BC, that can be followed. Thus the total probability to follow BC 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 $\mathscr{P}$, given it exists, is the product of probabilities to follow each edge of the path which is equal to $1 \cdot 0.6 \cdot 1 \cdot 0.9 = 0.54$. Since for the case shown in Figure 34 path $\mathscr{P}$ exists with probability 1, the final probability to follow $\mathscr{P}$ is $c(\mathscr{P}) = \mathrm{P}(\mathscr{P}^{\rightarrow}) = 0.54$. □

Example A.1.2 (Dependent edge existence). Let us now consider the case where the existence of an edge can dependent on the existence of the other edges. For the case shown in Figure 35 edges BF and DF cannot exist both at the same time. To compute $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists})$ we will consider two cases separately: $BF^{\exists}$ and $BF^{\nexists}$. That way we will be able to compute $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists})$ as $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists}) = \mathrm{P}(BF^{\exists}|\mathscr{P}^{\exists}) \cdot \mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists} \cap BF^{\exists}) + \mathrm{P}(BF^{\nexists}|\mathscr{P}^{\exists}) \cdot \mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists} \cap BF^{\nexists})$.

Let us first assume $BF^{\exists}$ (i.e., edge BF is present) and then compute $\mathrm{P}(BF^{\exists}|\mathscr{P}^{\exists}) \cdot \mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists} \cap BF^{\nexists})$. For the case of Figure 35, if no assumptions about the presence or absence of DF have been made yet, $\mathrm{P}(BF^{\exists}|\mathscr{P}^{\exists})$ is simply equal to $\mathrm{P}(BF^{\exists})$ which is equal to 0.8. If BF is present then DF is absent and the probability to follow $\mathscr{P}$ is $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists} \cap BF^{\nexists}) = 1 \cdot \frac{1}{2} \cdot 1 \cdot 1 = \frac{1}{2}$. Now let us consider the second case $BF^{\nexists}$ (and thus $DF^{\exists}$). The probability $\mathrm{P}(BF^{\nexists}|\mathscr{P}^{\exists})$ is 0.2. For that case $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists} \cap BF^{\nexists})$ is equal to $1 \cdot 1 \cdot 1 \cdot \frac{1}{2} = \frac{1}{2}$. Thus $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists}) = 0.8 \cdot \frac{1}{2} + 0.2 \cdot \frac{1}{2} = 0.5$. So $c(\mathscr{P}) = \mathrm{P}(\mathscr{P}^{\rightarrow}) = 0.50$, which is different from that of the previous experiment. □

A.2 Independent edge existence

Let us consider how to compute path connection strength in general case, assuming the existence of each edge is independent from existence of the other edges.

A.2.1 General formulae

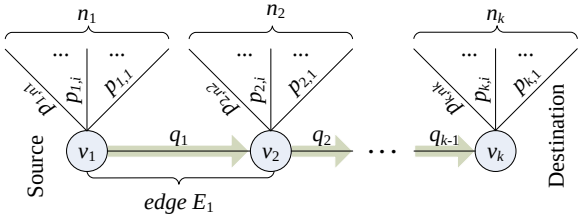

Figure 36: Independent edge existence. Computing $c(v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k)$. All edges shown in the figure are "possible to follow" edges in the context of the path. Edges that are not possible to follow are not shown.

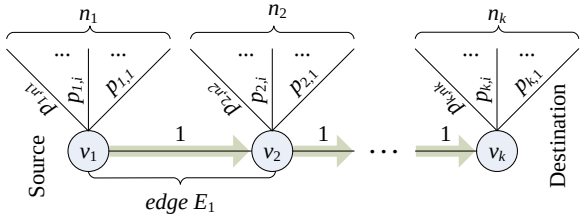

Figure 37: The case in this figure is similar to that of Figure 36 with an additional assumption that path $\mathscr{P}$ exists.

In general, any path $\mathscr{P}$ can be represented as a sequence of k nodes $\mathscr{P} = \langle v_1, v_2, \ldots, v_k \rangle$ or as a sequence of $k-1$ edges $\mathscr{P} = \langle E_{10}, E_{20}, \ldots, E_{(k-1)0} \rangle$, see Figure 36. Here, $E_{i0} = (v_i, v_{i+1})$ and $\mathrm{P}(E_{i0}^{\exists}) = p_{i0}$ $(i = 1, 2, \ldots, k-1)$. The goal is to compute the probability to follow path $\mathscr{P}$, which is the measure of the connection strength of path $\mathscr{P}$:

$$c(\mathscr{P}) = \mathrm{P}(\mathscr{P}^{\rightarrow}) = \mathrm{P}(\mathscr{P}^{\exists}) \cdot \mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists}) \tag{4}$$

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

$$\mathrm{P}(\mathscr{P}^{\exists}) = \mathrm{P}(\bigcap_{i=1}^{k-1} E_i^{\exists}) \tag{5}$$

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

$$\mathrm{P}(\mathscr{P}^{\exists}) = \prod_{i=1}^{k-1} \mathrm{P}(E_i^{\exists}) = \prod_{i=1}^{k-1} q_i \tag{6}$$

So, to compute $\mathrm{P}(\mathscr{P}^{\rightarrow})$ we now need to compute $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists})$. Let us see what changes in Figure 36 if we assume $\mathscr{P}^{\exists}$. Given that q_i is defined as $q_i = \mathrm{P}(E_i^{\exists})$, the new label $\widetilde{q}_i$ is computed as $\widetilde{q}_i = \mathrm{P}(E_i^{\exists}|\mathscr{P}^{\exists}) = 1$. Similarly, each p_{ij} is defined as $p_{ij} = \mathrm{P}(E_{ij}^{\exists})$. Thus each new label $\widetilde{p}_{ij}$ can be computed as $\widetilde{p}_{ij} = \mathrm{P}(E_{ij}^{\exists}|\mathscr{P}^{\exists})$. Therefore, given our assumption of independence $\widetilde{p}_{ij} = p_{ij}$. The new labeling is shown in Figure 37.

Let us define a variable a_{ij} for each edge E_{ij} (labeled p_{ij}) as follows: $(a_{ij} = 1)$ if and only if $(E_{ij}^{\exists})$; also $(a_{ij} = 0)$ if and only if $(E_{ij}^{\nexists})$. Also, for notational convenience, let us define a set of dummy variables a_{i0} and p_{i0}: $a_{i0} = 1$ and $p_{i0} = 1$ $(i = 1, 2, \ldots, k-1)$.[16] Let $\mathbf{a}$ denote a vector consisting of all a_{ij}'s: $\mathbf{a} = (a_{10}, a_{11}, \ldots, a_{k-1,n_{k-1}})$. Let $\mathcal{A}$ denote the set of all possible instantiations of $\mathbf{a}$, i.e. $|\mathcal{A}| = 2^{n_1+n_2+\cdots+n_{k-1}}$. Then probability $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists})$ can be computed as:

$$\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists}) = \sum_{\mathbf{a}\in\mathcal{A}} \Big\{ \mathrm{P}(\mathscr{P}^{\rightarrow}|\mathbf{a} \cap \mathscr{P}^{\exists}) \cdot \mathrm{P}(\mathbf{a}|\mathscr{P}^{\exists}) \Big\} \tag{7}$$

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

$$\mathrm{P}(\mathbf{a}|\mathscr{P}^{\exists}) = \mathrm{P}(\mathbf{a}) = \prod_{\substack{i=1,2,\ldots,k \\ j=0,1,\ldots,n_i}} p_{ij}^{a_{ij}} \cdot (1-p_{ij})^{1-a_{ij}}. \tag{8}$$

Probability $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathbf{a} \cap \mathscr{P}^{\exists})$, which is the probability to go via $\mathscr{P}$ given (1) a particular instantiation of $\mathbf{a}$; and (2) the fact that $\mathscr{P}$ exists, can be computed as:

$$\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathbf{a} \cap \mathscr{P}^{\exists}) = \prod_{i=1}^{k-1} \frac{1}{1 + \sum_{j=1}^{n_i} a_{ij}} \equiv \prod_{i=1}^{k-1} \frac{1}{\sum_{j=0}^{n_i} a_{ij}}. \tag{9}$$

Thus

$$\mathrm{P}(\mathscr{P}^{\rightarrow}) = \left(\prod_{i=1}^{k-1} q_i \right) \cdot \left(\sum_{\mathbf{a}\in\mathcal{A}} \left\{ \left[\prod_{i=1}^{k-1} \frac{1}{\sum_{j=0}^{n_i} a_{ij}} \right] \cdot \left[\prod_{ij} p_{ij}^{a_{ij}} \cdot (1-p_{ij})^{1-a_{ij}} \right] \right\} \right) \tag{10}$$

A.2.2 Computing path connection strength in practice

Notice, Equation (10) iterates through all possible instantiations of $\mathbf{a}$ which is impossible to compute in practice given $|\mathcal{A}| = 2^{n_1+n_2+\cdots+n_{k-1}}$. This equation must be simplified to make the computation feasible.

Computing $\mathbf{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists})$ as $\prod_{i=1}^{k-1} \mathbf{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists})$. To achieve the simplification, we will use our assumption of independence of probabilities which allows us to compute $\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists})$ as the product of the probabilities to follow each individual edge in the path:

$$\mathrm{P}(\mathscr{P}^{\rightarrow}|\mathscr{P}^{\exists}) = \prod_{i=1}^{k-1} \mathrm{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists}). \tag{11}$$

[16] Intuitively (1) $a_{i0} = 1$ corresponds to the fact that if $\mathscr{P}^{\exists}$ then $E_i^{\exists}$; and (2) $p_{i0} = 1$ corresponds to $p_{i0} = \mathrm{P}(E_i^{\exists}|\mathscr{P}^{\exists}) = 1$.

Let $\mathbf{a}_i$ denote vector $(a_{i0}, a_{i1}, \ldots, a_{in_i})$, that is $\mathbf{a} = (\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_{k-1}})$. Let $\mathcal{A}_i$ denote all possible instantiations of $\mathbf{a}_i$. That is, $\mathcal{A} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_{k-1}$ and $|\mathcal{A}_i| = 2^{n_i}$. Then

$$\mathrm{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists}) = \sum_{\mathbf{a}_i \in \mathcal{A}_i} \left\{ \left[\frac{1}{\sum_{j=0}^{n_i} a_{ij}} \right] \cdot \left[\prod_{j=0}^{n_i} p_{ij}^{a_{ij}} \cdot (1-p_{ij})^{1-a_{ij}} \right] \right\} \tag{12}$$

Combining Equations (4), (11) and (12) we have:

$$\mathrm{P}(\mathscr{P}^{\rightarrow}) = \left(\prod_{i=0}^{k} q_i \right) \cdot \prod_{i=1}^{k} \left(\sum_{\mathbf{a}_i \in \mathcal{A}_i} \left\{ \left[\frac{1}{\sum_{j=0}^{n_i} a_{ij}} \right] \cdot \left[\prod_{j=0}^{n_i} p_{ij}^{a_{ij}} \cdot (1-p_{ij})^{1-a_{ij}} \right] \right\} \right) \tag{13}$$

The effect of transformation. Notice, using Equation (10) the algorithm will need to perform $|\mathcal{A}| = 2^{n_1+n_2+\cdots+n_{k-1}}$ iterations – one per each instantiation of $\mathbf{a}$. Using Equation (13) the algorithm will need to perform $|\mathcal{A}_1| + |\mathcal{A}_2| + \cdots + |\mathcal{A}_{k-1}| = 2^{n_1} + 2^{n_2} + \cdots + 2^{n_{k-1}}$ iterations. Furthermore, each iteration requires less computation. These factors lead to a significant improvement.

Handling weight-1 edges. The formula in Equation (12) assumes 2^{n_i} iterations will be needed to

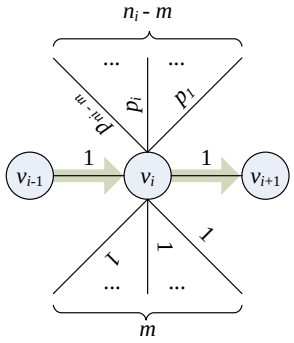

Figure 38: Probability to follow edge $E_i = (v_i, v_{i+1})$

compute $\mathrm{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists})$.This formula can be modified further to achieve more efficient computation as follows. In practice, some of the p_{ij}'s, or even all of them, are often equal to 1. Figure 38 shows the case where m, $0 \leq m \leq n_i$, edges incident to node v_i are labeled with 1. Let $\widetilde{\mathbf{a}}_i$ denote vector $(a_{i0}, a_{i1}, \ldots, a_{i(n_i-m)})$ and let $\widetilde{\mathcal{A}}_i$ be the set of all possible instantiations of this vector. Then Equation (12) can be simplified to:

$$\mathrm{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists}) = \sum_{\widetilde{\mathbf{a}}_i \in \widetilde{\mathcal{A}}_i} \left\{ \left[\frac{1}{m + \sum_{j=0}^{n_i-m} a_{ij}} \right] \cdot \left[\prod_{j=0}^{n_i-m} p_{ij}^{a_{ij}} \cdot (1-p_{ij})^{1-a_{ij}} \right] \right\} \tag{14}$$

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

Computing $\mathbf{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists})$ as $\sum_{j=0}^{k} \frac{1}{1+j} \cdot \mathbf{P}(s_i = j)$. Performing 2^{n_i-m} iterations can still be expensive for the cases when $(n_i - m)$ is large. Next we discuss several methods to deal with this issue.

Method 1: Do not simplify further. In general, the value of 2^{n_i-m} can be large. But for a particular instance of a cleaning problem it can be that (a)2^{n_i-m} is never large or (b) 2^{n_i-m} can be large but bearable and the cases when it is large are infrequent. In those cases further simplification might not be required.

Method 2: Estimate answer using results from Poisson trials theory. Let us denote the following sum as s_i: $s_i = \sum_{j=1}^{n_i} a_{ij}$. From a basic probability course we know that the binomial distribution gives the number of successes in n independent trials where each trial is successful with the same probability $\mathscr{P}$ [19]. The binomial distribution can be viewed as a sum of several *i.i.d.* Bernoulli trials. The *Poisson trials* 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 (13) to compute $\mathrm{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists})$ as follows:

$$\mathrm{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists}) = \sum_{l=0}^{k} \frac{1}{1+l} \cdot \mathrm{P}(s_i = l) \tag{15}$$

Notice, for a given i we can treat $a_{i1}, a_{i,2}, \ldots, a_{in_i}$ as a sequence of n_i Bernoulli trials with probabilities of success $\mathrm{P}(a_{ij} = 1) = p_{ij}$. One would want to *estimate* $\mathrm{P}(s_i = l)$ *quickly*, rather than compute it exactly via iterating over all cases when $(s_i = l)$. That is, one straightforward (and exact) method to compute $\mathrm{P}(s_i = l)$ is:

$$\mathrm{P}(s_i = l) = \sum_{\substack{\mathbf{a}_i \in \mathcal{A}_i \\ s_i = l}} \prod_{j=0}^{n_i} p_{ij}^{a_{ij}} \cdot (1 - p_{ij})^{1 - a_{ij}}$$

For example, if $p_{ij} = p$ for each $j = 1, 2, \ldots, n_i$, then we have a binomial distribution case and can compute $\mathrm{P}(s_i = l)$ quickly, and in this case exactly, as $C_{n_i}^{l} p^l (1-p)^{n_i - l}$.

In certain cases it can be possible to utilize the Poisson trials theory to estimate $\mathrm{P}(s_i = l)$. For example, if each p_{ij} is small then from the probability theory we know that

$$\mathrm{P}(s_i = l) = \frac{\lambda^l e^{-\lambda}}{l!} \left\{ 1 + O\left(\lambda \max_{j=1,2,\ldots,n_i} p_{ij} + \frac{l^2}{\lambda} \max_{j=1,2,\ldots,n_i} p_{ij} \right) \right\}, \text{ where } \lambda = \sum_{j=1}^{n_i} p_{ij}. \tag{16}$$

One can also utilize the following "Monte-Carlo like" method to compute $\mathrm{P}(s_i = l)$. The idea is to have several runs. During run number m, the method decides by generating a random number ("tossing a coin") if edge E_{ij} is present (variable a_j will be assigned 1) or absent ($a_j = 0$) for this run based on the probability p_{ij}. Then the sum $S_m = \sum_{j=1}^{n_i} p_{ij}$ is computed for that run. After n runs the desired probability $\mathrm{P}(s_i = l)$ is estimated as the number of S_i's which are equal to l, divided by n.

Method 3: Use linear cost formula. The third approach is to use a cut-off threshold to decide if the cost of performing $2^{n_i - m}$ iterations is acceptable. If it is acceptable then compute $\mathrm{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists})$ precisely, using iterations. If it is not acceptable (typically, rare case), then try to use Equation (16). If that fails, use the following (linear cost) approximation formula. First compute the expected number of edges μ_i among n_i edges $E_{i1}, E_{i,2}, \ldots, E_{in_i}$, where $\mathrm{P}(E_{ij}^{\exists}) = p_{ij}$, as follows: $\mu_i = m + \sum_{j=1}^{n_i - m} p_{ij}$. Then since there are $1 + \mu_i$ possible links to follow on average, the probability to follow E_i can be coarsely estimated as:

$$\mathrm{P}(E_i^{\rightarrow}|\mathscr{P}^{\exists}) \approx \frac{1}{1+\mu_i} = \frac{1}{m + \sum_{j=0}^{n_i - m} p_{ij}} \tag{17}$$

A.3 Dependent edge existence

In this section we discuss how to compute connection strength if occurrence of edges is not independent. In our model, dependence between two edges arises only when those two edges are option-edges of the same choice node. We next show how to compute $\mathrm{P}(\mathscr{P}^{\rightarrow})$ for those cases.

There are two principal situations we need to address. The first is to handle all choice nodes on the path. The second step is to handle all choice nodes such that a choice node itself is not on the path but at least two of its option nodes are on the path. Next we address those two cases.

A.3.1 Choice nodes on the path

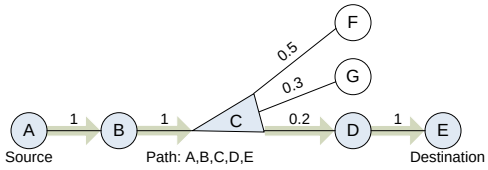

Figure 39: Choice node on the path

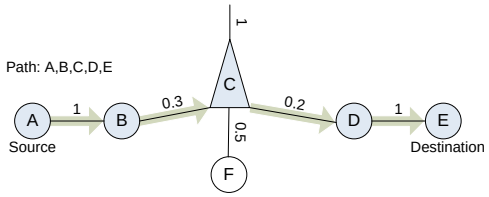

Figure 40: Choice node on the path: illegal path

The first case of how to deal with choice nodes on the path is a simple one. There are two sub-cases in this case illustrated in Figures 39 and 40.

Figure 39 shows a choice node C on the path which resolves some reference of the entity represented by node B and which have options D, G, and F. Recall, we compute $\mathrm{P}(\mathscr{P}^{\rightarrow}) = \mathrm{P}(\mathscr{P}^{\exists}) \cdot \mathrm{P}(\mathscr{P}^{\rightarrow} | \mathscr{P}^{\exists})$. When we compute $\mathrm{P}(\mathscr{P}^{\exists})$ each edge of path $\mathscr{P}$ should exist. Thus edge CD must exist, which means edges CG and CF do not exist. Notice, this case is equivalent to the case where (a) edges CG and CF are not there (permanently eliminated from consideration); and (b) node C is just a regular (not a choice) node connected to D via an edge (in this case the edge is labeled 0.2). If we now consider this equivalent case, then we can simply apply Equation (13) to compute the connection strength.

In general, all choice nodes on the path, can be "eliminated" from the path one by one (or, rather, "replaced with regular nodes") using the procedure above.

Figure 40 shows a choice node C on the path which have options B, F, and D, such that $B \rightarrow C \rightarrow D$ is a part of the path $\mathscr{P}$. Semantically, edges CB, CF, and CD are mutually exclusive: any two of those edges cannot exist at the same time. Since BC and CD are mutually exclusive, path $\mathscr{P}$ can never exist. Such paths are said to be *illegal* and they are ignored by RelDC.

A.3.2 Options of the same choice node on the path

Assume now we have applied the procedure from Section A.3.1 and all choice nodes are "eliminated" from path $\mathscr{P}$. At this point the probability $\mathrm{P}(\mathscr{P}^{\exists})$ can be computed as $\mathrm{P}(\mathscr{P}^{\exists}) = \prod_{i=1}^{k-1} q_i$. The only case that

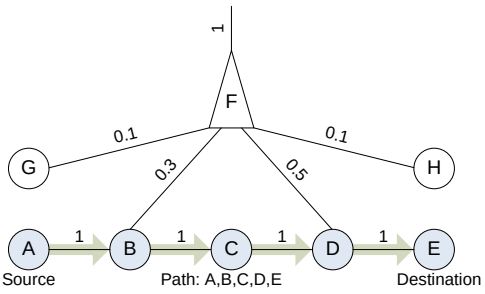

Figure 41: Options of the same choice node on the path

is left to be considered is where a choice node itself is not on the path but at least two of its options are on the path. An example of this case is illustrated in Figures 35 and 41. In Figure 41 choice node F has four options: G, B, D, and H, two of which B and D belong to the path being considered. After choice nodes are eliminated from the path, the goal becomes to create a formula similar to Equation (13), but for the general "dependent" case.

Let as define two sets $\mathbf{f}$ and $\mathbf{d}$ – of 'free' and 'dependent' a_{ij}'s as follows:

$$\begin{aligned}\mathbf{f} &= \{a_{ij} : \forall l,k \text{ where } l \neq i \text{ or } k \neq j \Rightarrow dep(a_{ij}, a_{l,k}) = \texttt{false}\} \\ \mathbf{d} &= \{a_{ij} : \exists l,k \text{ where } l \neq i \text{ or } k \neq j \Rightarrow dep(a_{ij}, a_{l,k}) = \texttt{true}\}\end{aligned} \tag{18}$$

Notice, $\mathbf{a} = \mathbf{f} \cup \mathbf{d}$ and $\mathbf{f} \cap \mathbf{d} = \emptyset$. If $\mathbf{d} = \emptyset$, then there is no dependence and the solution is given by Equation (13), otherwise we proceed as follows. Similarly to $\mathbf{a}_i$ we can define $\mathbf{f}_i$ and $\mathbf{d}_i$ as follows:

$$\begin{aligned}\mathbf{f}_i &= \{a_{ij} : a_{ij} \in \mathbf{f}, j = 0, 1, \ldots, n_i\} \\ \mathbf{d}_i &= \{a_{ij} : a_{ij} \in \mathbf{d}, j = 1, 2, \ldots, n_i\}\end{aligned} \tag{19}$$

Notice, $\mathbf{a}_i = \mathbf{f}_i \cup \mathbf{d}_i$ and $\mathbf{f}_i \cap \mathbf{d}_i = \emptyset$. We define $\mathcal{D}$ as the set of all possible instantiations of $\mathbf{d}$, and $\mathcal{F}_i$ as the set of all possible instantiations of $\mathbf{f}_i$. Then

$$\mathrm{P}(\mathscr{P}^{\rightarrow}) = \underbrace{\left(\prod_{i=1}^{k-1} q_i\right)}_{\mathrm{P}(\mathscr{P}^{\exists})} \times \sum_{\mathbf{d} \in \mathcal{D}} \underbrace{\left\{ \left[\prod_{i=1}^{k-1} \left(\sum_{\mathbf{f}_i \in \mathcal{F}_i} \left\{ \left[\frac{1}{\sum_{j=0}^{n_i} a_{ij}} \right] \cdot \left[\prod_{j: a_{ij} \in \mathbf{f}_i} p_{ij}^{a_{ij}} \cdot (1 - p_{ij})^{1 - a_{ij}} \right] \right\} \right) \right]}_{\Psi(\mathbf{d})} \cdot \mathrm{P}(\mathbf{d}) \right\} \tag{20}$$

Equation (20) iterates over all feasible instantiations of $\mathbf{d}$ and $\mathrm{P}(\mathbf{d})$ is the probability of a specific instance. Equation (20) contains term $\sum_{\mathbf{d} \in \mathcal{D}} \{\Psi(\mathbf{d}) \cdot \mathrm{P}(\mathbf{d})\}$. What this achieves is that a particular instantiation of $\mathbf{d}$ "fixates" a particular combination of all "dependent" edges, and $\mathrm{P}(\mathbf{d})$ corresponds to the probability of that combination. Notice, $\Psi(\mathbf{d})$ directly corresponds to $\mathrm{P}(\mathscr{P}^{\rightarrow} | \mathscr{P}^{\exists})$ part of Equation (13). To compute $\mathrm{P}(\mathscr{P}^{\rightarrow})$ in Equation (20), we only need to specify how to compute $\mathrm{P}(\mathbf{d})$.

Computing P(d). Recall, we now consider cases where a_{ij} is in $\mathbf{d}$ only because there is (at least one) another $a_{rs} \in \mathbf{d}$ such that $dep(E^{\exists}_{ij}, E^{\exists}_{rs}) = \texttt{true}$ and $cho[E_{ij}] = cho[E_{rs}]$. Figure 35 is an example of such a case. So, for each $a_{ij} \in \mathbf{d}$ we can identify choice node $v^*_l = cho[E_{ij}]$ and compute set $C_l = \{a_{rs} \in \mathbf{d} : cho[E_{rs}] = v^*_l\}$. Then, for any two distinct elements $a_{ij} \in C_l$ and a_{rs} the following holds: $dep(E^{\exists}_{ij}, E^{\exists}_{rs}) = \texttt{true}$ if and only if $a_{rs} \in C_l$.

In other words, we can split set $\mathbf{d}$ into non intersecting subsets $\mathbf{d} = C_1 \cup C_2 \cup \cdots \cup C_m$. The existence of each edge E_{ij} such that a_{ij} is in one of those sets C_l depends only on the existence of those edges E_{rs}'s whose a_{rs} is in C_l as well. Therefore $\mathrm{P}(\mathbf{d})$ can be computed as $\mathrm{P}(\mathbf{d}) = \mathrm{P}(C_1) \times \mathrm{P}(C_2) \times \cdots \times \mathrm{P}(C_m)$. Now, to be able to compute Equation (20), we only need to specify how to compute $P(C_i)$ $(i = 1, 2, \ldots, m)$.

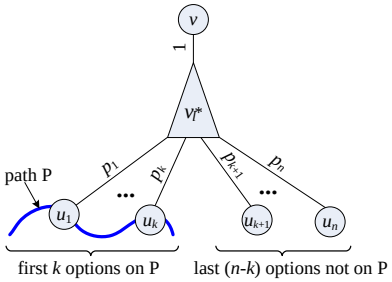

Figure 42: Intra choice dependence.

Computing $P(C_i)$. Figure 42 shows choice node v^*_l with n options $u_1, u_2, \ldots, u_n$. Each (v^*_l, u_j) edge $(j = 1, 2, \ldots, n)$ is labeled with probability p_i. As before, to specify which edge is present and which is

absent, each option edge has variable a_i associated with it. Variable $a_i = 1$ if and only if the edge labeled with p_i is present, otherwise $a_i = 0$. That is, $\mathrm{P}(a_i = 1) = p_i$ and $p_1 + p_2 + \cdots + p_n = 1$.

Let us assume, without loss of generality, that the first k $(2 \le k \le n)$ options $u_1, u_2, \ldots, u_k$ of v_l^* belong to path $\mathscr{P}$ while the other $(n-k)$ options $u_{k+1}, u_{k+2}, \ldots, u_n$ do not belong to $\mathscr{P}$, as shown in Figure 42. In the context of Figure 42, computing $P(C_l)$ is equivalent to computing the probability a particular instantiation of vector $(a_1, a_2, \ldots, a_k)$ to occur.

Notice, only one a_i among $a_1, a_2, \ldots, a_k, a_{k+1}, a_{k+2}, \ldots, a_n$ can be 1, the rest are zeroes. First let us compute the probability of instantiation $a_1 = a_2 = \cdots = a_k = 0$. For that case one of $a_{k+1}, a_{k+2}, \ldots, a_n$ should be equal to 1. Thus $\mathrm{P}(a_1 = a_2 = \cdots = a_k = 0) = p_{k+1} + p_{k+2} + \cdots + p_n$.

The second case is when one of $a_1, a_2, \ldots, a_k$ is 1. Assume $a_j = 1$, where $1 \le j \le k$, then $\mathrm{P}(a_j = 1) = p_l$. To summarize:

$$\mathrm{P}(a_1, a_2, \ldots, a_k) = \begin{cases} p_j & \text{if } \exists j \in [1,k] : a_j = 1 \\ p_{k+1} + p_{k+2} + \cdots + p_n & \text{if } \forall j \in [1,k] : a_j = 0 \end{cases}$$

Now we know how to compute $P(C_i)$ $(i = 1, 2, \ldots, m)$, thus we can compute $\mathrm{P}(\mathbf{d})$. Therefore we have specified how to compute path connection strength using Equation (20).

A.4 Computing the total connection strength.

The connection strength between nodes u and v is computed as a sum of connection strengths of all simple paths between u and v: $c(u,v) = \sum_{\mathscr{P} \in \mathcal{P}_L(u,v)} c(\mathscr{P})$. Based on this connection strength the weight of the corresponding edge will be determined. This weight will be treated as the probability of the edge to exist.

Let us give the motivation of why the *summation* of individual simple paths is performed. We associate the connection strength between two nodes u and v with probability of reaching v from u via only L-short simple paths. Let us name those simple paths $\mathscr{P}_1, \mathscr{P}_2, \ldots, \mathscr{P}_k$, Let us call $\mathcal{G}(u,v)$ the subgraph comprised of the union of those paths: $\mathcal{G}(u,v) = \mathscr{P}_1 \cup \mathscr{P}_2 \cup \cdots \cup \mathscr{P}_k$. Subgraph $\mathcal{G}(u,v)$ is a subgraph of the complete graph $G(V,E)$, where V is the set of vertices $V = \{v_i : i = 1, 2, \ldots, |V|\}$ and E is the set of edges $E = \{E_i : i = 1, 2, \ldots, |E|\}$. Let us define a_i as follows: $a_i = 1$ if and only if edge E_i is present, otherwise $a_i = 0$. Let $\mathbf{a}$ denote vector $(a_1, a_2, \ldots, a_{|E|})$ and let $\mathcal{A}$ be the set of all possible instantiations of $\mathbf{a}$.

We need to compute the probability to reach v from u via subgraph $\mathrm{P}(\mathcal{G}(u,v)^{\rightarrow})$ which we treat as the measure of the connection strength. We can represent $\mathrm{P}(\mathcal{G}(u,v)^{\rightarrow})$ as:

$$\mathrm{P}(\mathcal{G}(u,v)^{\rightarrow}) = \sum_{\mathbf{a} \in \mathcal{A}} \mathrm{P}(\mathcal{G}(u,v)^{\rightarrow} | \mathbf{a}) \cdot \mathrm{P}(\mathbf{a}) \tag{21}$$

Notice, when computing $\mathrm{P}(\mathcal{G}(u,v)^{\rightarrow} | \mathbf{a})$ we assume a particular instantiation of $\mathbf{a}$. So the complete knowledge of which edges are present and which are absent is available, as if all the edges were "fixed". Assuming one particular instantiation of $\mathbf{a}$, there is no dependence among edge existence events any longer: each edge is either present with 100% probability or absent with 100% probability. Thus

$$\mathrm{P}(\mathcal{G}(u,v)^{\rightarrow} | \mathbf{a}) = \sum_{i=1}^{k} \mathrm{P}(\mathscr{P}_i^{\rightarrow} | \mathbf{a}) \tag{22}$$

and

$$\begin{aligned}
\mathrm{P}(\mathcal{G}(u,v)^{\rightarrow}) &= \sum_{\mathbf{a}\in\mathcal{A}} \mathrm{P}(\mathcal{G}(u,v)^{\rightarrow}|\mathbf{a}) \cdot \mathrm{P}(\mathbf{a}) \\
&= \sum_{\mathbf{a}\in\mathcal{A}} \left[\left(\sum_{i=1}^{k} \mathrm{P}(\mathscr{P}_i^{\rightarrow}|\mathbf{a}) \right) \cdot \mathrm{P}(\mathbf{a}) \right] \\
&= \sum_{i=1}^{k} \left[\sum_{\mathbf{a}\in\mathcal{A}} \Big(\mathrm{P}(\mathscr{P}_i^{\rightarrow}|\mathbf{a}) \cdot \mathrm{P}(\mathbf{a}) \Big) \right] \\
&= \sum_{i=1}^{k} \mathrm{P}(\mathscr{P}_i^{\rightarrow})
\end{aligned} \tag{23}$$

Equation (23) shows that the total connection strength is the sum of the connection strength of all L-short simple paths.

B Optimizations of RelDC

In this section we present several optimizations of RelDC. These optimizations make RelDC 1–2 orders of magnitude more efficient.

RelDC spend most of its time discovering paths between nodes. This is not always true for Solv-RelDC: in certain situations the solver can spend relatively significant portion of the execution time solving the equations. However, we use an off-the-shelf math solver and cannot optimize it. So the bottleneck of RelDC that can be optimized is the path discovering (a.k.a. AllPaths) part of RelDC. Hence, all of the optimizations discussed in this section are for AllPaths part of RelDC.

Since our problem and the well-studied problem of the maximum network flow (MNF) bear some similarity, we have considered utilizing MNF for our algorithm. Unfortunately, MNF treats "weights" in a principally different way and cannot be applied. Thus we had to devise our own optimizations. Even though in theory the number of L-short simple paths between two nodes can be large, in practice it is not so. However, a naïve algorithm can make many *redundant traversals* 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 important optimizations. We classify optimizations in three large categories summarized below:

1. Constraining (simplifying) the problem
2. Algorithmic optimizations of RelDC
   (a) Preprocessing optimizations
   (b) Optimization for all (or only the first) iterations
   (c) Speeding up the subsequent iterations (only Iter-RelDC)
3. Exploiting parallelism

A parallel implementation of Iter-RelDC is possible because, when processing choices during a particular iteration, Iter-RelDC uses information (e.g. values of variables) only from the previous iteration. Therefore,

all *choice* nodes can be divided into several groups and each CPU will be responsible for resolving all *choice* nodes of one of the groups.[17]

B.1 Constraining the problem

The problem can be simplified by adding additional constraints. 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: (i) *domain dependence* and (ii) *purpose*. Rules can be *general* (domain independent) or *ad-hoc* (domain specific). In the *purpose* dimension we distinguish *resolution* and *speedup* rules, though rules that serve both resolution and speed-up purposes exist. *Resolution* rules guide the choice resolution process, while *speed-up* rules are needed to speed-up the resolution process. We have discussed several rules the analyst can use to constrain the problem in Section 5.1.

B.2 Algorithmic optimizations of AllPaths

This section presents optimizations for AllPaths algorithm. All optimizations are divided into three categories. Section B.2.1 describes one *preprocessing* optimization which is applied before the first invocation of AllPaths. Section B.2.2 presents optimizations applicable to *all* iterations. Finally, optimizations for speeding up the *subsequent* iterations (i.e., iterations number 2, 3,...) of Iter-RelDC are presented in Section B.2.3.

B.2.1 Preprocessing optimization

Before looking for *all* L-short paths, one can first try to determine whether there is *at least one* L-short 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 AllPaths. ShortestPath for two nodes u and v operates by carefully expanding both neighborhoods of u and v. ShortestPath does not maintain any intermediate paths and it does not return the shortest path. It returns whether or not there is a path between u and v, by analyzing whether or not the neighborhoods of u and v overlap. If ShortestPath determines there cannot be a *path* between u and v, then AllPaths will not find any paths either. Thus the corresponding option-edge will be permanently assigned weight of zero and the connection strength between u and v will not be recomputed on the next iterations.

B.2.2 Optimizations for all iterations

1-to-N implementation of AllPaths. When disambiguating reference $x_i.r_k$, RelDC needs to compute the connection strength not only for one pair (x_i, y_j) but rather for N pairs (x_i, y_j) $(j = 1, 2, \ldots, N)$. Thus far we have used the approach which computes all simple paths for those pairs in a pair by pair fashion. A small modification of an implementation of AllPaths algorithm allows to achieve (more efficient) 1-to-N path discovery in one run of AllPaths algorithm. When checking if the destination node y_j is reached the algorithm now instead checks if any one of $y_1, y_2, \ldots, y_N$ is reached. To achieve that, all $y_1, y_2, \ldots, y_N$ are beforehand inserted into a hash table so the check of whether a destination is reached still takes $O(1)$ time.

Utilizing neighborhoods for path pruning (NBH optimization). See Section 5.3.

[17] A parallel version of Iter-RelDC has been implemented but rarely used. That version needs to be improved: performance on 3,4,... CPUs is worse than on 2 CPUs due to the caching and synchronization issues.

Utilizing graph reachability for path pruning (GraphReach optimization). Graph reachability (GraphReach) optimization is similar to the optimization that utilizes neighborhoods (NBH): it speeds up the algorithm by pruning certain paths. GraphReach optimization is based on the fact that often the graph on which the algorithm works is not random: it is either structured or can be made structured. This is especially the case for graphs constructed from relational tables of relational databases.

Structured in this context means that a node of a certain type $type_1$ can be reached from a different node of $type_2$ only via *at least* k edges. For instance, consider Figure 11. A traversal of *at least* 3 edges is needed to reach any node of type *paper* from a node of type *university*.

The information about the minimum number of edges needed to be traversed from any node of one type to reach any node of the second type can be stored in a table for all types and looked up when needed. It is possible to exploit the reachability information to prune certain paths. For example, assume the algorithm looks for L-short $u \rightsquigarrow v$ paths. Assume the algorithm currently examines an intermediate path $p_1 : u \overset{p_1}{\rightsquigarrow} x$ of length m. From the reachability table the algorithm can look up the minimum number of edges d needed to be traversed from a node of type $type(x)$ to a node of type $type(v)$. Then if d is such that $(m + d) > L$, then there cannot be an L-short path of type $u \overset{p_1}{\rightsquigarrow} x \overset{p_2}{\rightsquigarrow} v$ for any path $p_2 : x \overset{p_2}{\rightsquigarrow} v$.

Comparing GraphReach and NBH. GraphReach and NBH are quite similar, let us take up the differences between them. The advantage of GraphReach optimization over NBH optimization is that GraphReach 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 GraphReach is that neighborhoods in general contain more information for pruning than the knowledge of reachability. GraphReach provides very general information about a lower bound of the minimum number of edges the algorithm will need to traverse from current node before it might reach a node of *type* equal to the *type* of the destination node. Neighborhoods, if available, provides information about the exact minimum distance (in number of edges) to the *destination node*. Thus the algorithm is likely to make more redundant traversals with GraphReach than with NBH. In our tests NBH optimization has shown significantly better results than GraphReach optimization. However GraphReach optimization improves performance by an order of magnitude when compared with the unoptimized version of RelDC.

Greedy implementation of AllPaths algorithm. See Section 5.2.

B.2.3 Optimizations for the subsequent iterations

(1) Storing discovered paths explicitly. See Section 5.4

(2) Path coloring. In this section we present a solution which on the first iteration of Iter-RelDC marks the discovered paths in graph G in a certain way and then uses this information later on to speed up the subsequent iterations of Iter-RelDC. The solution requires $O(|V|)$ space overhead for graph $G(V, E)$. Let us first explain the general idea behind the path coloring technique and then show how it can be implemented in practice.

During the first iteration, all L-short simple paths $\mathcal{P}_L(x_i, y_j)$ between $v[x_i]$ and $v[y_j]$ $(j = 1, 2, \ldots, N)$ will be found. The path coloring technique picks one color, $color(x_i, y_j)$, among C available colors per each (x_i, y_j) pair. It then marks all paths in $\mathcal{P}_L(x_i, y_j)$ with the same color $color(x_i, y_j)$. It also tries to assign different colors to different pairs (x_i, y_j), however this is often not possible since often the number of such pairs is greater than the number of colors C.

For example, the only path between $v[x_i]$ and $v[y_2]$ in Figure 43 is assigned red color, and paths between $v[x_i]$ and $v[y_N]$ are assigned blue color. Coloring of a path with a certain color means that all nodes on the path are assigned this color. Each node can have multiple colors. For instance, a node can have colors of blue and red at the same time, such as node v_1 in Figure 43.

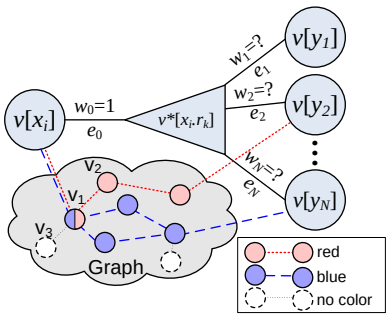

Figure 43: Path coloring

During the subsequent iterations the coloring information can be utilized to prune certain paths by utilizing the coloring that was used to mark the paths on the first iteration as follows. Assume the algorithm looks for L-short $u \rightsquigarrow v$ paths. Suppose that on the first iteration all the discovered paths of type $u \rightsquigarrow v$ have been marked with the red color. Assume the algorithm currently examines an intermediate path $p_1 : u \overset{p_1}{\rightsquigarrow} x$ all nodes of which have the red color so far. Suppose the algorithm examines the direct neighbors of x and determines that path $u \overset{p_1}{\rightsquigarrow} x \rightarrow y$ is legitimate. However, if node y does not have the red color, then there cannot be an L-short path $u \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} v$ for any path $p_2 : y \overset{p_2}{\rightsquigarrow} v$. This is because otherwise that path would have been discovered on the first iteration and y would have been marked red.

For instance, in Figure 43 all $v[x_i] \rightsquigarrow v[y_2]$ path have been assigned the red color by RelDC on the first iteration. When looking for $v[x_i]$ to $v[y_2]$ paths on the second iteration the algorithm will first retrieve v_1. Next, when deciding where to go next, it can skip all the nodes except for v_2 and so on. That way the algorithm is capable of pruning of many redundant traversals.

Implementation of path coloring. Implementing the path coloring technique is surprisingly simple: it requires only around 30–40 lines of codes. Path coloring requires each node to have an Cbit *color* field. Consequently, it will require $C \cdot |V|$ bits of additional storage, so the number of colors C is limited in practice. The *color* field of a node is implemented as an array of $\lceil \frac{C}{8} \rceil$ bytes. If all bits are set to zero, then the node *has no color*. If i^{th} bit of the *color* fields is set to 1, then the node *has color* i. To test if a node has color i (or to mark the node with color i), the algorithm computes $\lfloor \frac{C}{8} \rfloor$ to determine the location of the right byte in the *color* field and (*color* mod 8) to determine the location of the corresponding bit in that byte.

Picking colors. When resolving references $x_i.r_k$'s the color is picked for each (x_i, y_j) pair once during the first iteration. All L-short simple paths between x_i and y_j will be marked with the chosen color. The colors should be picked such that in the end each color is used roughly equal number of times. That is, the extreme case is when every node has the same single 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 for each (x_i, y_j) pair. One way to implement that, is to make the first iteration store this color explicitly, e.g. in a table, for each (x_i, y_j) pair. Our implementation instead utilizes a hash function which maps the unique identifiers of x_i and y_j to an integer between 0 and $C - 1$ to create one of the C 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 equal number of times. The (symmetric) hash function we use to pick color for pair (x_i, y_j) is as follows

$$hash(u, v, C) = (u.id + v.id) \mod C,$$

where id's are *unique* node id's.

Limited number of colors. Because the number of colors is limited to C, the same color, in general, will be assigned to multiple (x_i, y_j) pairs. Consequently, the algorithm with the coloring optimization, might still explore paths which will not lead to the destinations. The larger the number of colors C the more efficient the optimization. Naturally, this improvement is constrained since having more colors than the number of (x_i, y_j) pairs is of little value.

Removing not colored nodes. If, after the first iteration of Iter-RelDC, a node does not have any color, this means the node 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 RelDC does not even have to retrieve the node to see that it does not have the right color (has no color). However one should be careful: removing nodes means decreasing the degree of its neighbor nodes, which can affect the system which measure the connection strength. Thus it is imperative to store in nodes the weights of removed edges.

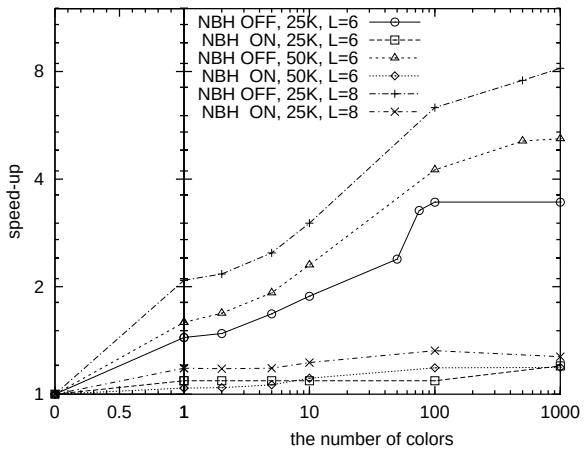

Figure 44: RealPub: Speedup vs. number of colors

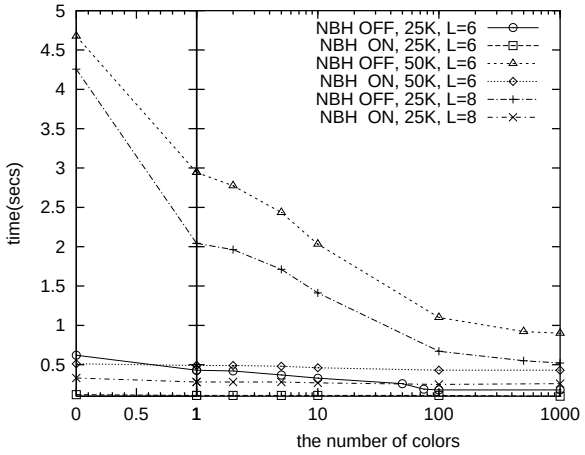

Figure 45: RealPub: Time of second invocation of AllPaths vs. number of colors

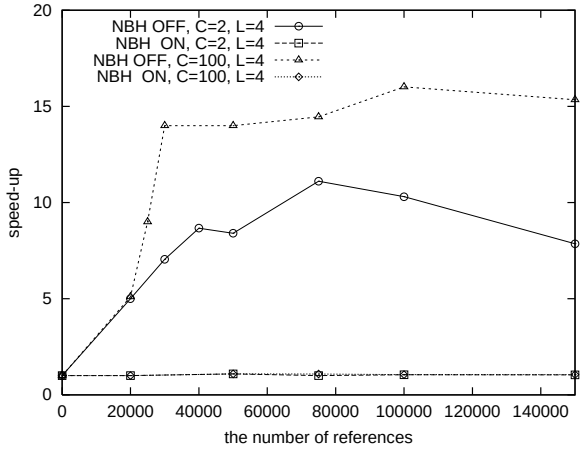

Figure 46: RealPub: Speed-up vs. the number of references in a subset of RealPub

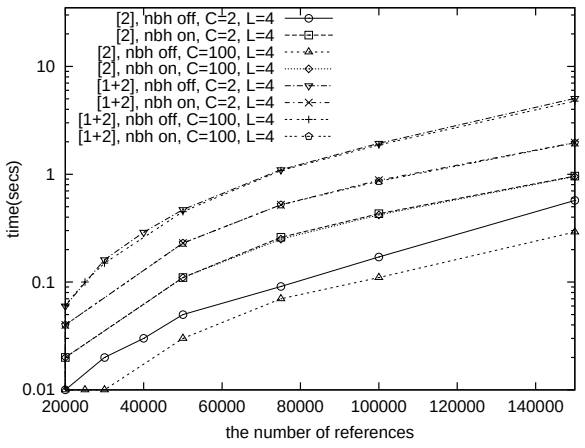

Figure 47: RealPub: The execution time of AllPaths vs. the number of references in a subset of RealPub

Experiment 12 (The path coloring optimization). In this experiment we study the improvement of

performance of AllPaths (not the complete RelDC) algorithm when path coloring optimization is used. All experiments are on subsets of RealPub. The first invocation of AllPaths will color paths using C colors in total. The second (and the rest of) invocation(s) of AllPaths will utilize the color information to rediscover the paths more efficiently.

Figure 44 shows the effect of the number of colors on the speedup achieved by the path coloring technique. The speedup is measured as the execution time of AllPaths on the first iteration of Iter-RelDC divided by that of the second iteration. The figure has six curves for six series of experiments. "NBH ON/OFF" shows whether the NBH optimization was turned on or off. Notation 25K (50K) shows that the size of the tested subset of RealPub was such that it contained only 25K (50K) of 573K *auth_name_ref*'s. When NBH optimization is OFF, the path coloring technique shows significant improvement which, as expected, increases when the number of colors increases but stabilizes when no more colors are needed. When NBH optimization is ON the effect of the path coloring optimization is much less significant. This is because NBH optimization performs very effective path pruning. As a result, when the other pruning techniques which we have discussed are applied concurrently with NBH, they lead to only a small improvement.

Figure 45 shows the effect of the number of colors on the execution time of the second invocation of AllPaths algorithm. When NBH optimization is off, the path coloring technique shows significant improvement. However, when NBH is on, it cannot improve the efficiency much further, thus the curves remain mostly flat.

Figure 46 studies the achieved speedup as the size of the subset of RealPub increases. In this figure x-axis show the number of references in the subset of RealPub being tested. Notation "$C = 2$" means two colors were used. That is, each node can be in 4 states: (a) have no color, (b) have color 0, (c) have color 1, and (d) have two colors: 0 and 1. When NBH optimization is on, the path coloring does not lead to a significant improvement: the curves are flat and close to 1. When NBH optimization is OFF, initially, when the dataset size is small (the number of references is small) the graph is sparse and there are not many path alternatives. Thus the path coloring technique leads to small speedup. The speedup improves as graph becomes larger and denser. It then fluctuates: for $C = 2$ it fluctuates in $[7.5, 12]$, for $C = 100$ in $[14, 16.5]$.

Figure 47 studies the efficiency of the path coloring optimization on AllPaths as the size of subsets of RealPub increases. Notation "[1+2]" means the time of both the first and second invocations of AllPaths is reported whereas notation "[2]" means the time of only second invocation is reported. The figure shows that when NBH optimization is on, the number of colors has little effect on the performance and thus the corresponding curves for 2 and 100 colors coincide. Also the figure shows that if both the first and second invocations of AllPaths are considered, when L is 4, having 100 colors instead of 2 only marginally improves performance, so the two curves coincide as well. This is because for $L = 4$ having just two colors already significantly improves performance and the combined cost is largely determined by the cost of the first invocation of AllPaths. □

B.3 Summary: the most important optimizations.

Appendix B presents many optimizations. Some of those optimizations, like the path coloring and storing paths explicitly optimizations, are mutually exclusive. Some of the optimizations lead to significant improvement under variety of conditions, while others work well only under certain conditions. Some do not work well in combination with other optimizations. Naturally, the question arises: which are the most important optimizations that preferably work well under variety of conditions, specifically for large datasets? Next we list the most important optimizations in the order of their significance. The optimization that limits path length is important but rather straightforward, and thus it is not listed below.

1. *NBH optimization* – the most important optimization. It allows to prune paths very effectively. In fact, it does that so effectively that if other pruning-based optimizations are used, the additional speed-up they achieve is typically under 20%. While it is possible that for certain cases using another pruning-based optimization, called GraphReach, might be more efficient than using NBH optimization, in general NBH is significantly better than GraphReach.

2. *Storing paths explicitly on disk.* This optimization allows to significantly reduce the processing cost of iterations $2, 3, \ldots$ of Iter-RelDC. However, the path coloring technique is often a better option for smaller datasets.

3. *ShortestPath* and *Weight cut-off optimizations.* These optimizations allow to quickly filter out unlikely options from choice sets.

C Alternative WM formulae

C.1 Addressing drawbacks of Equation (3)

One could argue that the formula in Equation (3) does not address properly the situation illustrated in Figure 48. In the example in Figure 48, when disambiguating references $x_i.r_k$ the choice set for this reference $CS[x_i.r_k]$ has three elements y_1, y_2, and y_3. In Figure 48(a) the connection strengths $c_j = c(x_i, y_j)$

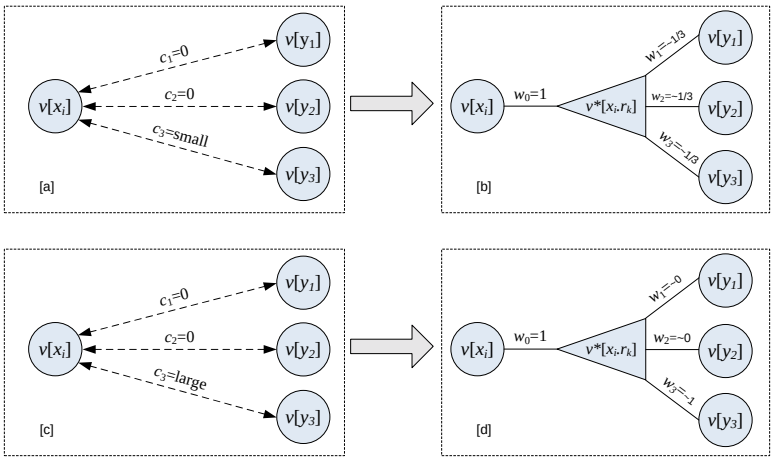

Figure 48: Motivation for *Normalization method 2*

($j = 1, 2, 3$) are as follows: $c_1 = 0$, $c_2 = 0$, and c_3 is a nonnegative value which is small. That is, RelDC has not been able to find any evidence that $d[x_i.r_k]$ is y_1 or y_2 and found insubstantial evidence that $d[x_i.r_k]$ is y_3. However Equation (3) will compute $w_1 = 0$, $w_2 = 0$, and $w_3 = 1$, one interpretation of which might be that the algorithm is 100% confident y_3 is $d[x_i.r_k]$.

One can argue that in such a situation, since the evidence that $d[x_i.r_k]$ is y_3 is very weak, w_1, w_2, and w_3 should be roughly equal. That is, their values should be close to $\frac{1}{3}$ in this case, as shown in Figure 48(b), and w_3 should be slightly greater than w_1 and w_2.

Figure 48(c) is similar to Figure 48(a), except for c_3 is large with respect to other connection strengths in the system. Following the same logic, weights w_1 and w_2 should be close to zero. Weight w_3 should be close to 1, as in Figure 48(d).

We can correct those issues with Equation (3) and achieve the desired weight assignment as follows. We will assume that since y_1, y_2, and y_3 are in the choice set $CS[x_i.r_k]$ of reference $x_i.r_k$ (whereas other

entities are not in the choice set), in such situations there is always a very small default connection strength α between each x_i and y_j. That is, Equation (3) is modified and the weights are assigned as follows.

$$w_j = \frac{(c_j + \alpha)}{\sum_{l=1,2,\ldots,N}(c_l + \alpha)} \tag{24}$$

where α is a small positive weight: $\alpha \in \mathbb{R}^+$. Equation (24) corrects the mentioned drawbacks of Equation (3).