Similarity Flooding: A Versatile Graph Matching Algorithm (Extended Technical Report)

Sergey Melnik* Hector Garcia-Molina Erhard Rahm

Stanford University CA 94305 University of Leipzig, Germany

{melnik,hector}@db.stanford.edu rahm@informatik.uni-leipzig.de

Abstract

Matching elements of two data schemas or two data instances plays a key role in data warehousing, e-business, or even biochemical applications. In this paper we present a matching algorithm based on a fixpoint computation that is usable across different scenarios. The algorithm takes two graphs (schemas, catalogs, or other data structures) as input, and produces as output a mapping between corresponding nodes of the graphs. Depending on the matching goal, a subset of the mapping is chosen using filters. After our algorithm runs, we expect a human to check and if necessary adjust the results. As a matter of fact, we evaluate the 'accuracy' of the algorithm by counting the number of needed adjustments. We conducted a user study, in which our accuracy metric was used to estimate the labor savings that the users could obtain by utilizing our algorithm to obtain an initial matching. Finally, we discuss how our matching algorithm is deployed as one of several high-level operators in an implemented testbed for managing information models and mappings.

Keywords: Matching, Model Management, Heterogeneous Databases, Semistructured Data

1 Introduction

Finding correspondences between elements of data schemas or data instances is required in many application scenarios. This task is often referred to as *matching*. Consider a comparison shopping

*On leave from the University of Leipzig

website that aggregates product offers from multiple independent online stores. The comparison site developers need to match the product catalogs of each store against their combined catalog. For instance, the 'product code' field in one catalog may match the 'product ID' and 'store ID' fields in the combined catalog. Or, think of a merger between two corporations, both of which need to consolidate their relational databases deployed by different departments. In this integration scenario, and in many data warehousing applications, matching of relational schemas is required. Schema matching is utilized for a variety of other types of schemas including UML class taxonomies, ER diagrams, and ontologies.

Matching of data instances is another important task. For example, consider two CAD files or program scripts that have been independently modified by several developers. In this scenario, matching helps to identify moved or modified elements in these complex data structures. In bioinformatics, matching has been used for network analysis of molecular interactions [Kan00]. In this domain, data instances represent e.g. metabolic networks of chemical compounds, or molecular assembly maps. Matching of molecular networks and biochemical pathways helps to predict metabolism of an organism given its genome sequence. Finally, as we will demonstrate, matching may be utilized for computing schema correspondences using instance data, and for finding related elements in a data instance.

While this paper focuses on matching, the broader goal of our work is to design a generic tool that helps to manipulate and maintain schemas, instances, and match results. With this tool, matching is not done entirely automatically. Instead, the tool assists human developers in matching by suggesting plausible match candidates for the elements of a schema. Using a graphical interface, the user adjusts the proposed match result by removing or adding lines connecting the elements of two schemas. Often, the correct match depends on the information only available or understandable by humans. For example, even matches as plausible as `ZipCode` to `zip_code` can be doomed as incorrect by a data warehouse designer who knows that zip codes from a given relational source should not be collected due to poor data quality. In such cases, the mappings suggested by the tool may be incorrect or incomplete.

Matching problems often differ a lot. So do the approaches to matching. For example, for matching relational schemas one could use SQL data types to determine which columns are closely

related. On the other hand, in XML schema matching, hierarchical relationships between schema elements can be exploited. Because of this diversity, applications that rely on matching are often built from scratch and require significant amount of thought and programming. We address this problem by proposing a matching algorithm that allows quick development of matchers for a broad spectrum of different scenarios. We are not trying to outperform custom matchers that use highly tuned, domain-specific heuristics.

In this paper we suggest a simple structural algorithm that can be used for matching of diverse data structures. Such data structures, which we call *models*, may be data schemas, data instances, or a combination of both. The elements of models represent artifacts like relational tables and columns, or products and customers. The algorithm that we suggest is based on the following idea. First, we convert the models to be matched into directed labeled graphs. These graphs are used in an iterative fixpoint computation whose results tell us what nodes in one graph are similar to nodes in the second graph. For computing the similarities, we rely on the intuition that elements of two distinct models are similar when their adjacent elements are similar. In other words, a part of the similarity of two elements propagates to their respective neighbors. The spreading of similarities in the matched models is reminiscent to the way how IP packets flood the network in broadcast communication. For this reason, we call our algorithm the *similarity flooding* algorithm. We refer to the result produced by the algorithm as a *mapping*. Depending on the particular matching goal, we then choose a subset of the resulting mapping using adequate filters. After our algorithm runs, we expect a human to check and if necessary adjust the results. As a matter of fact, in Section 6 we evaluate the 'accuracy' of the algorithm by counting the number of needed adjustments.

Our algorithm is targeted at binary, undirectional matching. An expanded classification of different goals and approaches to matching is presented in [RB01]. In this paper we are making the following contributions:

- We introduce a generic matching algorithm that is usable across application areas (Section 3), and discuss approaches for selecting relevant subsets of match results (Section 4).
- We demonstrate the applicability of the algorithm for diverse matching tasks (Section 5).
- We suggest an accuracy metric for evaluating automatic schema matching algorithms (Sec-

CREATE TABLE *Personnel* (
  Pno int,
  Pname string,
  Dept string,
  Born date,
  UNIQUE *perskey*(*Pno*)
)

(S_1)

CREATE TABLE *Employee* (
  EmpNo int **PRIMARY KEY**,
  EmpName varchar(50),
  DeptNo int **REFERENCES** *Department*,
  Salary dec(15,2),
  Birthdate date
)

CREATE TABLE *Department* (
  DeptNo int **PRIMARY KEY**,
  DeptName varchar(70)
)

(S_2)

Figure 1: Matching two relational schemas: Personnel and Employee-Department

tion 6) and evaluate the effectiveness of our algorithm on the basis of a user study that we conducted (Section 7).

- We describe the testbed that we developed, in which high-level algebraic operations are deployed for manipulating models and mappings (Section 8).

Finally, we summarize the known limitations in Section 9, and conclude with the related work.

2 Overview of the Approach

Before we go into details of our matching algorithm, let us briefly walk through an example that illustrates matching of two relational database schemas. Please keep in mind that the technique we describe is not limited to relational schemas. Consider schemas S_1 and S_2 depicted in Figure 1. The elements of S_1 and S_2 are tables and columns. Assume for now that our goal is to obtain exactly one matching element for every element in S_1. A part of the matching result could be, for example, the correspondence of column `Personnel/Pname` to column `Employee/EmpName`. A sequence of steps that allows us to determine the correspondences between tables and columns in S_1 and S_2 can be expressed as the following script:

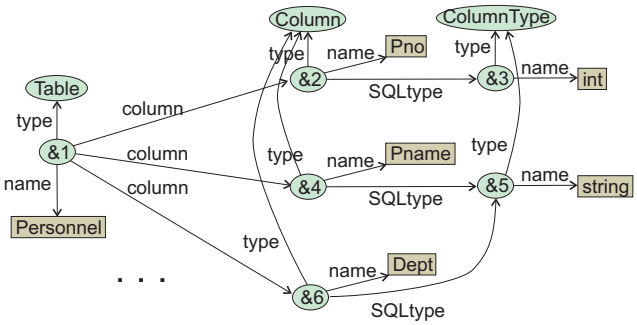

Figure 2: A portion of graph representation G_1 for relational schema S_1

1. G_1 = SQLDDL2Graph(S_1); G_2 = SQLDDL2Graph(S_2);
2. initialMap = StringMatch(G_1, G_2);
3. product = SFJoin(G_1, G_2, initialMap);
4. result = SelectThreshold(product);

As a first step, we translate the schemas from their native format into graphs G_1 and G_2. In our example, the native format of the schemas are ASCII files containing table definitions in SQL DDL. A portion of the graph G_1 is depicted in Figure 2. The translation into graphs is done using an import filter SQLDDL2Graph that understands the definitions of relational schemas. We do not insist on choosing a particular graph representation for relational schemas. The representation used in Figure 2 is based on the Open Information Model (OIM) specification [OIM99]. The nodes in the graph are shown as ovals and rectangles. The labels inside the ovals denote the identifiers of the nodes, whereas rectangles represent literals, or string values. For example, node `&1` represents the table `Personnel` in graph G_1, whereas nodes `&2`, `&4`, and `&6` denote columns `Pno`, `Pname`, and `Dept`, respectively. Column `Born` and unique key `perskey` are omitted from the figure for clarity. Tables `Employee` and `Department` from schema S_2 are represented in a similar manner in graph G_2. In our example, G_1 has a total of 31 nodes while G_2 has 55 nodes.

As a second step, we obtain an initial mapping initialMap between G_1 and G_2 using operator StringMatch. The mapping initialMap is obtained using a simple string matcher that compares common prefixes and suffixes of literals. A portion of the initial mapping is shown in Table 1. Literal nodes are highlighted using apostrophes. The second column of the table lists similarity

Line#	Similarity	Node in G_1	Node in G_2
1.	1.0	Column	Column
2.	0.66	ColumnType	Column
3.	0.66	'Dept'	'DeptNo'
4.	0.66	'Dept'	'DeptName'
5.	0.5	UniqueKey	PrimaryKey
6.	0.26	'Pname'	'DeptName'
7.	0.26	'Pname'	'EmpName'
8.	0.22	'date'	'Birthdate'
9.	0.11	'Dept'	'Department'
10.	0.06	'int'	'Department'

Table 1: A portion of initialMap obtained by string matching (10 of total 26 entries are shown)

values between nodes in G_1 and G_2 computed on the basis of their textual content. The similarity values range between 0 and 1 and indicate how well the corresponding nodes in G_1 match their counterparts in G_2. Notice that the initial mapping is still quite imprecise. For instance, it suggests mapping column names onto table names (e.g. column `Dept` in S_1 onto table `Department` in S_2, line 9), or names of data types onto column names (e.g. SQL type `date` in S_1 onto column `Birthdate` in S_2, line 8).

As a third step, operator SFJoin is applied to produce a refined mapping called product between G_1 and G_2. In this paper we propose an iterative 'similarity flooding' (SF) algorithm based on a fixpoint computation that is used for implementing operator SFJoin. The SF algorithm has no knowledge of node and edge semantics. As a starting point for the fixpoint computation the algorithm uses an initial mapping like initialMap. Our algorithm is based on the assumption that whenever any two elements in models G_1 and G_2 are found to be similar, the similarity of their adjacent elements increases. Thus, over a number of iterations, the initial similarity of any two nodes propagates through the graphs. For example, in the first iteration the initial textual similarity of strings '`Pname`' and '`EmpName`' adds to the similarity of columns `Personnel/Pname` and `Employee/EmpName`. In the next iteration, the similarity of `Personnel/Pname` to `Employee/EmpName` propagates to the SQL types `string` and `varchar(50)`. This subsequently causes increase in similarity between literals '`string`' and '`varchar`', leading to a higher resemblance of `Personnel/Dept` to `Department/DeptName` than

Similarity	Node in G_1	Node in G_2
1.0	Column	Column
0.81	[Table: Personnel]	[Table: Employee]
0.66	ColumnType	ColumnType
0.44	[ColumnType: int]	[ColumnType: int]
0.43	Table	Table
0.35	[ColumnType: date]	[ColumnType: date]
0.29	[UniqueKey: perskey]	[PrimaryKey: on EmpNo]
0.28	[Column: Personnel/Dept]	[Column: Department/DeptName]
0.25	[Column: Personnel/Pno]	[Column: Employee/EmpNo]
0.19	UniqueKey	PrimaryKey
0.18	[Column: Personnel/Pname]	[Column: Employee/EmpName]
0.17	[Column: Personnel/Born]	[Column: Employee/Birthdate]

Table 2: The mapping after applying SelectThreshold on result of SFJoin

that of `Personnel/Dept` to `Department/DeptNo`. The algorithm terminates after a fixpoint has been reached, i.e. the similarities of all model elements stabilize. In our example, the refined mapping product returned by SFJoin contains 211 node pairs with positive similarities (out of a total of $31 \cdot 55 = 1705$ entries in the G_1, G_2 cross-product).

As a last operation in the script, operator SelectThreshold selects a subset of node pairs in product that corresponds to the 'most plausible' matching entries. We discuss this operator in Section 4. The complete mapping returned by SelectThreshold contains 12 entries and is listed in Table 2. For readability, we substituted numeric node identifiers by the descriptions of the objects they represent. For example, we replaced node identifier `&2` by `[Column:Personnel/Pno]`.

As we see in Table 2, the SF algorithm was able to produce a good mapping between S_1 and S_2 without any built-in knowledge about SQL DDL by merely using graph structures. For example, table `Personnel` was matched to table `Employee` despite the lack of textual similarity. Notice that the table still contains correspondences like the one between node `Column` in G_1 to node `Column` in G_2, which are hardly of use given our goal of matching the specific tables and columns. We discuss the filtering of match results in more detail in Section 4. The similarity values shown in the table may appear relatively low. As we will explain, in presence of multiple match candidates for a given

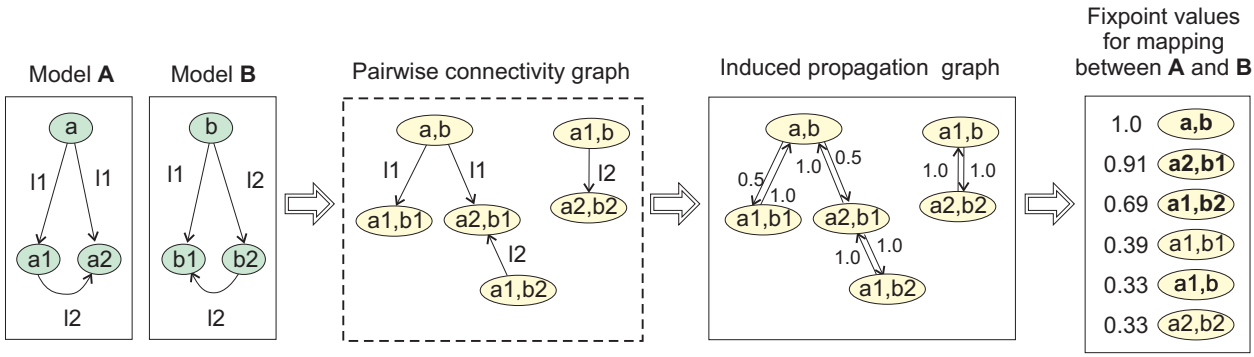

Figure 3: Example illustrating the Similarity Flooding algorithm

model element, relative similarities are often more important than absolute values.

3 Similarity Flooding Algorithm

The internal data model that we use for models and mappings is based on directed labeled graphs. Every edge in a graph is represented as a triple (s, p, o), where s and o are the source and target nodes of the edge, and the middle element p is the label of the edge. For a more formal definition of our internal data model please refer to Appendix A. In this section, we explain our algorithm using a simple example presented in Figure 3. The top left part of the figure shows two models A and B that we want to match.

Similarity propagation graph A *similarity propagation graph* is an auxiliary data structure derived from models A and B that is used in the fixpoint computation of our algorithm. To illustrate how the propagation graph is computed from A and B, we first define a *pairwise connectivity graph* (PCG) as follows:

$$((x, y), p, (x', y')) \in \mathrm{PCG}(A, B) \iff (x, p, x') \in A \text{ and } (y, p, y') \in B$$

Each node in the connectivity graph is an element from $A \times B$. We call such nodes *map pairs*. The connectivity graph for our example is enclosed in a dashed frame in Figure 3. The intuition behind arcs that connect map pairs is the following. Consider for example map pairs (a, b) and (a_1, b_1). If a is similar to b, then probably a_1 is somewhat similar to b_1. The evidence for this

conclusion is provided by the l_1-edges that connect a to a_1 in graph A and b to b_1 in graph B. This evidence is captured in the connectivity graph as an l_1-edge leading from (a, b) to (a_1, b_1). We call (a_1, b_1) and (a, b) *neighbors*.

The induced propagation graph for A and B is shown next to the connectivity graph in Figure 3. For every edge in the connectivity graph, the propagation graph contains an additional edge going in the opposite direction. The weights placed on the edges of the propagation graph indicate how well the similarity of a given map pair propagates to its neighbors and back. These so-called *propagation coefficients* range from 0 to 1 inclusively and can be computed in many different ways. The approach illustrated in Figure 3 is based on the intuition that each edge type makes an equal contribution of 1.0 to spreading of similarities from a given map pair. For example, there is exactly one l_2-edge out of (a_1, b) in the connectivity graph. In such case we set the coefficient $w((a_1, b), (a_2, b_2))$ in the propagation graph to 1.0. The value 1.0 indicates that the similarity of a_1 to b contributes fully to that of a_2 and b_2. Analogously, the propagation coefficient $w((a_2, b_2), (a_1, b))$ on the reverse edge is also set to 1.0, since there is exactly one incoming l_2-edge for (a_2, b_2). In contrast, two l_1-edges are leaving map pair (a, b) in the connectivity graph. Thus, the weight of 1.0 is distributed equally among $w((a, b), (a_1, b_1)) = 0.5$ and $w((a, b), (a_2, b_1)) = 0.5$. In Appendix B we analyze several alternative ways of computing the propagation coefficients.

Fixpoint computation Let $\sigma(x, y) \geq 0$ be the similarity measure of nodes $x \in A$ and $y \in B$ defined as a total function over $A \times B$. We refer to σ as a mapping. The similarity flooding algorithm is based on an iterative computation of σ-values. Let σ^i denote the mapping between A and B after i^{th} iteration. Mapping σ^0 represents the initial similarity between nodes of A and B, which is typically obtained using string comparisons of node labels. In our example we assume that no initial mapping between A and B is available, i.e. $\sigma^0(x, y) = 1.0$ for all $(x, y) \in A \times B$.

In every iteration, the σ-values for a map pair (x, y) are incremented by the σ-values of its neighbor pairs in the propagation graph multiplied by the propagation coefficients on the edges going from the neighbor pairs to (x, y). For example, after the first iteration $\sigma^1(a_1, b_1) = \sigma^0(a_1, b_1) + \sigma^0(a, b) \cdot 0.5 = 1.5$. Analogously, $\sigma^1(a, b) = \sigma^0(a, b) + \sigma^0(a_1, b_1) \cdot 1.0 + \sigma^0(a_2, b_1) \cdot 1.0 = 3.0$. Then, all values are normalized, i.e., divided by the maximal σ-value (of current iteration) $\sigma^1(a, b) = 3.0$.

Identifier	Fixpoint formula
Basic	$\sigma^{i+1} = normalize(\sigma^i + \varphi(\sigma^i))$
A	$\sigma^{i+1} = normalize(\sigma^0 + \varphi(\sigma^i))$
B	$\sigma^{i+1} = normalize(\varphi(\sigma^0 + \sigma^i))$
C	$\sigma^{i+1} = normalize(\sigma^0 + \sigma^i + \varphi(\sigma^0 + \sigma^i))$

Table 3: Variations of the fixpoint formula

Thus, after normalization we get $\sigma^1(a, b) = 1.0$, $\sigma^1(a_1, b_1) = \frac{1.5}{3.0} = 0.5$, etc. In general, mapping σ^{i+1} is computed from mapping σ^i as follows (normalization is omitted for clarity):

$$\sigma^{i+1}(x,y) = \sigma^i(x,y) + \sum_{(a_u,p,x)\in A,\ (b_u,p,y)\in B} \sigma^i(a_u,b_u)\cdot w((a_u,b_u),(x,y)) + \sum_{(x,p,a_v)\in A,\ (y,p,b_v)\in B} \sigma^i(a_v,b_v)\cdot w((a_v,b_v),(x,y))$$

The above computation is performed iteratively until the Euclidean length of the residual vector $\Delta(\sigma^n, \sigma^{n-1})$ becomes less than ε for some $n > 0$. If the computation does not converge, we terminate it after some maximal number of iterations. In Section 7 we study the convergence properties of the algorithm. The right part of Figure 3 displays the similarity values for the map pairs in the propagation graph. These values have been obtained after five iterations using the above equation. In the figure, the top three matches with the highest ranks are highlighted in bold. These map pairs indicate how the nodes in A should be mapped onto nodes in B.

Taking normalization into account, we can rewrite the above equation to obtain the 'basic' fixpoint formula shown in Table 3. The function φ increments the similarities of each map pair based on similarities of their neighbors in the propagation graph. The variations A, B, and C of the fixpoint formula are studied in Section 7. Our experiments suggest that formula C performs best with respect to quality of match results and convergence speed.

4 Filters

In this section we examine several filters that can be used for choosing the best match candidates from the list of ranked map pairs returned by the similarity flooding algorithm. Usually, for every element in the matched models, the algorithm delivers a large set of match candidates. Hence, the

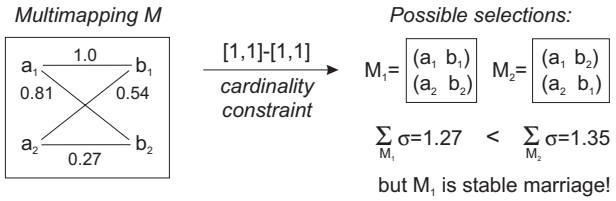

Figure 4: Cumulative similarity vs. 'stable marriage'

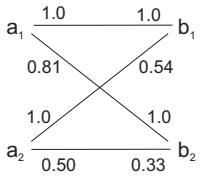

Figure 5: Relative similarities for the example in Figure 4

immediate result of the fixpoint computation (like the one shown on the right of Figure 3) may still be too voluminous for many matching tasks. For instance, in a schema matching application the choice presented to a human user for every schema element may be overwhelming, even when the presented match candidates are ordered by rank. We refer to the immediate result of the iterative computation as *multimapping*, since it contains many potentially useful mappings as subsets.

It is not evident, which criteria could be useful for selecting a desirable subset from a multimapping. An additional complication is that as many as 2^n different subsets can be formed from a set of n map pairs. To illustrate the selection problem, consider the match result obtained for two tiny models A and B that is shown on the left in Figure 4 (the models themselves are are omitted in the figure for clarity). The multimapping M contains four map pairs with similarities $\sigma(a_1, b_1) = 1.0$, $\sigma(a_2, b_1) = 0.54$, etc. From the set of 4 pairs, $2^4 = 16$ distinct subsets can be selected. Every one of these 16 subsets may be a plausible alternative for the final match result presented to the user.

We address the selection problem using a three-step approach. First, we use the available application-specific constraints to reduce the size of the multimapping. As exemplified below, typing and cardinality constraints may help to eliminate many map pairs from the multimapping. As a second step, we use selection techniques developed in context of matching in bipartite graphs to pick out the subset that is finally delivered to the user. At last, we evaluate the usefulness of particular selection techniques for a given class of matching tasks (e.g. schema matching) and choose the technique with empirically best results. In the rest of this section we discuss the first two steps in more detail. We present an evaluation of several selection techniques in Section 7.

Constraints Frequently, matching tasks include application-specific constraints that can be used for pruning of a large portion of possible selections. Recall our relational schemas S_1 (`Personnel`) and S_2 (`Employee`) from Section 2. At least two useful constraints are conceivable for this matching scenario. First, we could use a *typing* constraint to restrict the result to only those matches that hold between columns or tables, i.e., we can ignore matches of keys, data types etc. Second, if our goal were to populate the `Personnel` table with data from the `Employee` table, we could deploy a *cardinality* constraint that requires exactly one match candidate for every element of schema S_1. In this case, the cardinality of the resulting mapping would have to satisfy the restriction $[0, n] - [1, 1]$ (using the UML notation). The right expression $[1, 1]$ limits the number of S_2-elements that may match each element of S_1 to exactly one (between a lower limit of 1 and an upper limit of 1). Conversely, the left expression $[0, n]$ specifies the valid number of S_1-match candidates (between 0 and n) for each element of S_2, i.e. elements of S_2 may remain unmatched or may have one or more match candidates.

Unfortunately, in many matching tasks typing or cardinality constraints do not narrow down the match result sufficiently. To illustrate, consider the multimapping in Figure 4. If the definition of the matching task implies a cardinality constraint $[0, n] - [1, 1]$ (i.e. the mapping is required to contain exactly one match candidate for every element in A), 4 of 16 selections remain possible. A stricter cardinality constraint $[1, 1] - [1, 1]$ (i.e. one-to-one mapping) limits our choice to two sets of map pairs M_1 and M_2 shown on the right in Figure 4. Even after applying tight constraints in this simple matching task we are still left with more than one choice. Below we examine several strategies for making the decision between the remaining alternatives M_i.

Selection metrics To make an educated choice between M_i's we need an intuition of what constitutes a 'better' mapping. Fortunately, our selection dilemma is closely related to well-known matching problems in bipartite graphs, so that we can build on intuitions and algorithms developed for solving this class of problems (see e.g. [LP86, GI89]). In the graph matching literature, a *matching* is defined as a mapping with cardinality $[0, 1] - [0, 1]$, i.e., a set of edges no two of which are incident on the same node. A *bipartite* graph is one whose nodes form two disjoint parts such that no edge connects any two nodes in the same part. Thus, a mapping can be viewed as an undirected

weighted bipartite graph.

A helpful intuition that we will predominantly use for explaining alternative selection strategies for multimappings is provided by the so-called *stable marriage* problem. To remind, in an instance of the stable marriage problem, each of n women and n men lists the members of the opposite sex in order of preference. The goal is to find the best match between men and women. A stable marriage is defined as a complete matching of men and women with the property that there are no two couples (x, y) and (x', y') such that x prefers y' to y and y' prefers x to x'. For obvious reasons, such a situation would be regarded as unstable. Imagine that in Figure 4 elements a_1 and a_2 correspond to women. Then, men b_1 and b_2 would be the primary and the secondary choice for woman a_1. Obviously, mapping M_1 satisfies the stable marriage condition, whereas M_2 does not. In M_2, woman a_1 and man b_1 favor each other over their actual partners, which puts their marriages in jeopardy.

The stable-marriage property provides a plausible criterion for selecting a desired mapping from a multimapping. Further candidates for desired mappings can be drawn from the following selection criteria and well-known matching problems:

- The *assignment problem* consists in finding a matching M_i in a weighted bipartite graph M that maximizes the total weight (cumulative similarity) $\sum_{(x,y)\in M_i} \sigma(x, y)$. Viewed as a marriage, such matching maximizes the total satisfaction of all men and women. In Figure 4, $\sum_{M_2} \sigma = 0.81 + 0.54 = 1.35$, whereas $\sum_{M_1} \sigma = 1.0 + 0.27 = 1.27$. Thus, M_2 maximizes the total satisfaction of all men and women even though M_2 is not a stable marriage.

- Another group of selection candidates are maximal, maximum and perfect matchings. A *maximal* matching is a matching that is not properly contained in any other matching. A *maximum* matching is a matching of maximum cardinality, i.e. with the most number of married couples. A *perfect* (or *complete*) matching is one containing an edge incident of every node, i.e. the one in which every man and woman is married. Obviously, a perfect matching is achievable only if both parts of a mapping contain the same number of elements. M_1 and M_2 in Figure 4 are maximal, maximum and perfect matchings. All of the above-mentioned matching problems produce $[0, 1] - [0, 1]$ mappings, i.e. monogamous marriages, and can be solved using polynomial-time algorithms [LP86, MR95].

- Under *polygamy*, multiple matching counterparts for every element are allowed. Polygamy is useful for matching tasks in which many-to-many mappings are desirable. In schema matching, for instance, an element of one schema may have multiple counterparts in another schema. A polygamous variant of perfect matching corresponds to an *outer match*, i.e. a minimal mapping in which every element in both models has at least one counterpart. When multiple partners are allowed, the number of candidates for every element can be used as an additional factor for selecting the desired subset. For example, we may favor a subset M_i of the multimapping that maximizes function $\sum_{M_i} \frac{\sigma(x,y)}{\|(x,?)\| \cdot \|(?,y)\|}$, analogously to the optimum function used in the assignment problem. Terms $\|(x,?)\|$ and $\|(?,y)\|$ denote the number of partners for woman x and man y in M_i.

- The flooding algorithm produces at most one similarity value for any map pair (x, y). We call this value *absolute* similarity. Absolute similarity is symmetric, i.e. x is similar to y exactly as y to x. Under the marriage interpretation, this means that any two prospective partners like each other to the same extent. Considering *relative* similarities suggests a more diversified interpretation. Relative similarities are asymmetric and are computed as fractions of the absolute similarities of the best match candidates for any given element. In the example in Figure 4, b_1 is the best match candidate for a_2, so we set $\vec{\sigma}_{rel}\ (a_2, b_1) := 1.0$. The relative similarity for all other match candidates of a_2 is computed as a fraction of $\sigma(a_2, b_1)$. Thus, $\vec{\sigma}_{rel}\ (a_2, b_2) := \frac{\sigma(a_2,b_2)}{\sigma(a_2,b_1)} = \frac{0.27}{0.54} = 0.5$. All relative similarities for this example are summarized in Figure 5. A multimapping based on relative similarities corresponds to a *directed* weighted bipartite graph. The previously mentioned selection strategies can be adapted to relative similarities in a straightforward way.

- Some matching tasks require finding a connected subgraph in the target model that matches best the one in the source model. In such case, the number of edit operations needed to transform one subgraph to another may be included in the selection metric.

- Similarity *thresholds* are the last criteria that we discuss. For a given absolute-similarity threshold t_{abs} we select a subset of a multimapping, in which all map pairs carry an absolute similarity value of at least t_{abs}. For example, for $t_{abs} = 0.5$, Figure 4 suggests that woman

a_2 finds man b_1 acceptable (0.54), and would rather not go out with man b_2 at all (0.27). The relative-similarity threshold t_{rel} is used analogously. In the same example, for a relative-similarity threshold $t_{rel} = 0.5$ woman a_2 would still accept man b_2 as a partner, but man b_2 would reject woman a_2 since $\overleftarrow{\sigma}_{rel}(a_2, b_2) = 0.33 < 0.5$.

To summarize, the filtering problem can be characterized by providing a set of constraints and a selection function that picks out the 'best' subset of the multimapping under a given selection metric. Conceptually, the selection function assigns a value to every subset of the multimapping. The subset for which the function takes the largest/smallest value is selected as the final result. For example, using the assignment problem as selection metric, we can construct a filter that applies a cardinality constraint $[0, 1] - [0, 1]$ and utilizes a selection function $\sum_{(x,y)\in M_i} \sigma(x, y)$ to choose the best subset. Some selection metrics (e.g. threshold-based ones) can be described in terms of a boolean selection function that assigns the value 1 for one subset of the multimapping, and 0 to all others. In concrete implementations of selection functions, we can often find algorithms that avoid enumerating all subsets of the multimapping and determine the desired subset directly.

In the remainder of this section we describe a filter that produced empirically best results in a variety of schema matching tasks, as we show later in Section 7. This approach is implemented in our testbed as the SelectThreshold operator. The intuition behind this approach is based on a *perfectionist egalitarian polygamy*, which means that no male or female is willing to accept any partner(s) but the best. This criterion corresponds to using relative-similarity threshold $t_{rel} = 1.0$.

SelectThreshold operator selects a subset of the multimapping which is guaranteed to satisfy the stable-marriage property. However, this selection strategy sacrifices the happiness of those individuals who are not number one on the list of at least one person of the opposite sex. Such individuals are left unmarried, i.e. excluded from the mapping. Most of the time, SelectThreshold with $t_{rel} = 1.0$ yields matchings, or monogamous societies. In a less picky version of the operator with $t_{rel} < 1.0$, more persons have a chance to find a partner, and polygamy is more likely. In the examples presented in the following section we demonstrate the impact of threshold value t_{rel} in several practical scenarios.

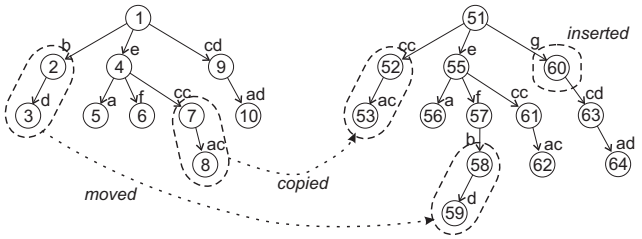

Figure 6: Matching of semistructured data

5 Features of the Algorithm by Example

In this section we discuss the features and limitations of the similarity flooding algorithm using four matching problems. In Section 2 we demonstrated how the algorithm performs on two sample relational schemas encoded as directed labeled graphs. Our next example deals with matching of semistructured data instances. After that, we illustrate matching of XML schemas. The third example addresses matching XML schemas using XML instance data. The last example deals with the task of finding related data elements in a database. The goal of our discussion in this section is to illustrate the usefulness of the algorithm and the threshold-based filter defined in the previous section for different application scenarios.

Semistructured data Detecting changes by comparing data snapshots in an important task in difference queries, version and configuration management. Figure 6 shows an example borrowed from [CGM97] that illustrates change detection in two labeled trees. The numbers inside the circles are node identifiers. The tree T_2 on the right has been obtained from the tree T_1 on the left by applying a series of transformation operations. First, all node identifiers have been replaced. In addition, some subtrees have been copied and moved, and a new node (60) has been inserted. In this example, we are interested in finding a best match candidate for every node of T_2 (i.e. a mapping between T_2 and T_1 that satisfies the cardinality constraint $[0, n] - [1, 1]$). We can express the matching procedure using the following script:

1. product = SFJoin(T_2, T_1);
2. result = SelectLeft(product);

Since no initial mapping is passed to SFJoin, the initial similarities between all nodes are set to

Similarity	Node $t_2 \in T_2$	Node $t_1 \in T_1$	Operation	$\vec{\sigma}_{rel}\ (t_2, t_1)$	$\overleftarrow{\sigma}_{rel}\ (t_2, t_1)$
1.0	55	4		1	1
0.63	61	7		1	1
0.58	51	1		1	1
0.48	56	5		1	1
0.48	57	6		1	1
0.30	62	8		1	1
0.07	52	7	copied	1	0.11
0.07	58	2	moved	1	1
0.07	63	9	moved	1	1
0.05	53	8	copied	1	0.16
0.05	59	3	moved	1	1
0.05	60	1	inserted	1	0.09
0.05	64	10	moved	1	1

Table 4: The mapping after applying SelectLeft $\circ$ SFJoin to semistructured data in Figure 6

1.0. We are using operator SelectLeft instead of SelectThreshold to ensure that all nodes of T_2 are present in the resulting mapping. For every 'left' node of the mapping, SelectLeft returns the match candidate with the highest absolute similarity. The result of matching is shown in Table 4. The fourth column in the table describes the transformation operations performed on the nodes (this information it is not part of the resulting mapping and is provided for illustration only). As the table suggests, the algorithm could correctly map every node in the modified tree T_2 to its previous version in T_1. Notice a heavy drop in similarity for copied, moved and inserted nodes. This result supports the intuition that exact structural matches should yield higher similarity values.

The right-most columns of the table show the relative similarities of the nodes in T_1 and T_2. For instance, node 62 is the top candidate for node 8, so $\overleftarrow{\sigma}_{rel}\ (62, 8) = 1$. For node 53, i.e. the second best candidate, $\overleftarrow{\sigma}_{rel}\ (53, 8) = \frac{\sigma(53,8)}{\sigma(62,8)} = \frac{0.05}{0.30} = 0.16$. If instead of SelectLeft we applied SelectThreshold with any $t_{rel} \in (0.16, 1]$ to the result of SFJoin, we would get all map pairs for T_1-nodes that have been either just renamed or moved. Lowering t_{rel} to 0.10 causes all copied nodes to appear additionally in the result. Finally, setting t_{rel} to a value like 0.05 includes the inserted node (but still filters out the rest of total 130 map pairs returned by SFJoin). This example illustrates that in certain scenarios undesired results can be pruned quickly by modifying threshold values

```
<Schema name="Schema 1"
  xmlns="urn:schemas-microsoft-com:xml-data">
<ElementType name="AccountOwner">
    <element type="Name"/>
    <element type="Address"/>
    <element type="Birthdate"/>
    <element type="TaxExempt"/>
</ElementType>
<ElementType name="Address">
    <element type="Street"/>
    <element type="City"/>
    <element type="State"/>
    <element type="ZIP"/>
</ElementType>
</Schema>
```

```
<Schema name="Schema 2"
  xmlns="urn:schemas-microsoft-com:xml-data">
<ElementType name="Customer">
    <element type="Cname"/>
    <element type="CAddress"/>
    <element type="CPhone"/>
</ElementType>
<ElementType name="CustomerAddress">
    <element type="Street"/>
    <element type="City"/>
    <element type="USState"/>
    <element type="PostalCode"/>
</ElementType>
</Schema>
```

Figure 7: Matching of two XML schemas: AccountOwner (S_1) vs. Customer (S_2)

interactively.

XML schemas The next example that we discuss illustrates how our algorithm copes with different choices of graph-based representation for the models to be matched. Consider two XML schemas in Figure 7. The schemas are specified using the XML schema language deployed on the website `biztalk.org` designed for electronic documents used in e-business.

As in the example of matching relational schemas (Section 2), both XML data structures are first converted algorithmically into graphs. Figure 8 shows portions of two different graph-based representations that are frequently used for manipulating XML data structures. The XML graph representation on the left corresponds to that of OEM/Lore [PGMW95], while the representation on the right is based on the XML/DOM standard. In the OEM representation, element tags are treated as edge labels, whereas in DOM representation hierarchical relationships between elements are captured using a uniform edge labels `child`.

The result of matching `AccountOwner` and `Customer` schemas is depicted in Table 6. Two left-

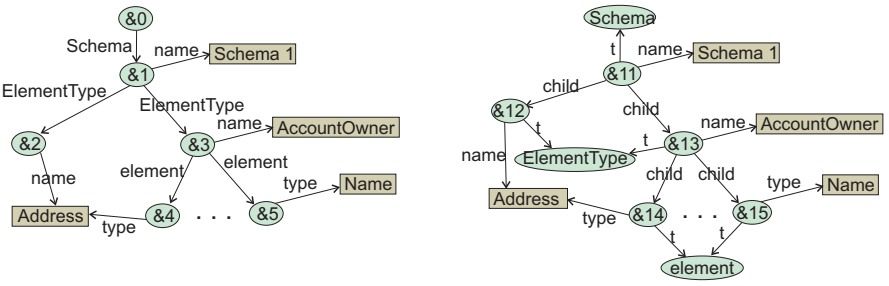

Figure 8: Two different representations of XML data: OEM/Lore-like vs. XML/DOM-like

most columns show the similarity values for computed map pairs. Omitted values indicate that the corresponding map pair does not appear in the match result. For readability, we substituted numeric node identifiers by the descriptions of the objects they represent (in square brackets). The mapping for the OEM representation was obtained by executing the script

1. G_1 = XML2OEMGraph(S_1); G_2 = XML2OEMGraph(S_2);
2. initialMap = StringMatch(G_1, G_2);
3. product = SFJoin(G_1, G_2, initialMap);
4. result = SelectThreshold(product);

For exploiting the DOM representation, the first line is replaced by

1. G_1 = XML2DOMGraph(S_1); G_2 = XML2DOMGraph(S_2);

This example illustrates two features of the algorithm. First, the algorithm produces similar results for different choices of graph-based representation. Second, the example shows that graph-based representations for models that use a wider spectrum of edge labels contributes to a faster iterative computation. The sizes of the graphs in both representations are presented in Table 5. Notice that although the graphs for S_1 and S_2 have similar sizes in both representations, the propagation graph in the OEM representation is 50% smaller than that of the DOM-like representation. Thus, every fixpoint iteration takes less time (we discuss the complexity of the algorithm in detail in Appendix D). Also note that the only extra code required for adapting the algorithm for matching XML schemas is the implementation of the XML2OEMGraph or XML2DOMGraph operator.

Matching XML schemas using instance data Two previous examples illustrated matching of instance data and matching of schema data. The third example that we discuss in this section

Nodes in S_1	Nodes in S_2	Nodes in propagation graph	Iterations
37	39	128	7
40	38	267	6

Table 5: Parameters of the fixpoint computation for S_1 and S_2

σ using OEM	σ using DOM	Node in S_1	Node in S_2
1.0		XMLARC	XMLARC
0.82	1.0	[Schema: Schema M1]	[Schema: Schema M2]
0.81	0.55	[ElementType: Address]	[ElementType: CustomerAddress]
0.40	0.25	[element: Street]	[element: Street]
0.40	0.25	[element: City]	[element: City]
0.24	0.33	[ElementType: AccountOwner]	[ElementType: Customer]
0.15	0.13	[element: Name]	[element: Cname]
0.14	0.11	[element: State]	[element: USState]
0.11	0.10	[element: Address]	[element: CAddress]
0.05	0.06	[element: ZIP]	[element: PostalCode]
	0.75	element	element
	0.40	ElementType	ElementType
	0.32	XMLDOM	XMLDOM
	0.32	urn:schemas-microsoft-com:xml-data	urn:schemas-microsoft-com:xml-data
	0.32	Schema	Schema

Table 6: Match results for XML schemas in Figure 7 using two different graph representations

```
<amazon>
  <item>
    <title>Sony DCR-PC100 Digital HandyCam
           Camcorder</title>
    <listPrice>1899.99</listPrice>
    <ourPrice>1699.00</ourPrice>
    <youSave>200.00</youSave>
    <review>
       <avgReview>4.5</avgReview>
       <numOfReviews>20</numOfReviews>
    </review>
    <availability>On Order; usually ships
                  within 1-2 weeks</availability>
    <features>
       <zoom>10x optical zoom</zoom>
       <zoom>120x digital zoom</zoom>
       <lcd>2.5 inch LCD</lcd>
       <other>4 MB Memory Stick included</other>
    </features>
  </item>
</amazon>
```

```
<yahoo>
  <productInfo>
    <id>Sony DCR-PC100</id>
    <merchantPrice>1799.94</merchantPrice>
    <rating>
       <userRating>3.5</userRating>
       <userReviews>7</userReviews>
    </rating>
    <description>
       <LCDScreenSize>2.5in</LCDScreenSize>
       <opticalZoom>10 X</opticalZoom>
       <special>4MB Memory Stick</special>
    </description>
  </productInfo>
</yahoo>
```

Figure 9: Matching of two XML schemas using instance data in DOM graph representation

deals with yet another matching problem, matching XML schemas using instance data. Consider two XML instances depicted in Figure 9. The data on the left contains information about a Sony camcorder on the `amazon.com` website. The data on the right shows similar information from the `yahoo.com` website. XML tag names for both schemas were derived from the actual vocabulary terms used on both sites. For example, Amazon site uses term `review`, whereas Yahoo site talks about `rating`. Notice that many text pieces in both XML files are different.

Table 7 shows how XML tags used in `amazon` and `yahoo` match. This result was determined by running our algorithm on XML/DOM graphs corresponding to both data instances. After that, the match candidates that do not correspond to XML tags were filtered out using a custom operator

Similarity	Tag in db1	Tag in db2
0.27	item	productInfo
0.20	amazon	yahoo
0.18	zoom	opticalZoom
0.12	features	description
0.11	ourPrice	merchantPrice
0.11	listPrice	merchantPrice
0.09	title	id
0.08	numOfReviews	userReviews
0.07	other	special
0.06	lcd	LCDScreenSize
0.05	review	userReviews
0.04	avgReview	userReviews
0.04	review	rating
0.03	youSave	id
0.03	avgReview	userRating
...	...	...

Table 7: Match results for XML element tags in Figure 9 using similarity threshold 0.05

XMLMapFilter:

1. G_1 = XML2DOMGraph(db_1); G_2 = XML2DOMGraph(db_2);
2. initialMap = StringMatch(G_1, G_2);
3. product = SFJoin(G_1, G_2, initialMap);
4. result = XMLMapFilter(product, G_1, G_2);

Setting the minimal similarity t_{abs} to 0.05 returns a set of correspondences shown above the horizontal bar in the table. Notice that the only additional code required for using the algorithm for matching XML schemas on the basis of instance data was the implementation of operator XMLMapFilter.

Find related One last application that we illustrate in this section deals with finding related data instances. The relatedness information can be computed using the same instance graph for both inputs of the algorithm. Consider the instance graph in Figure 10. This graph captures a piece

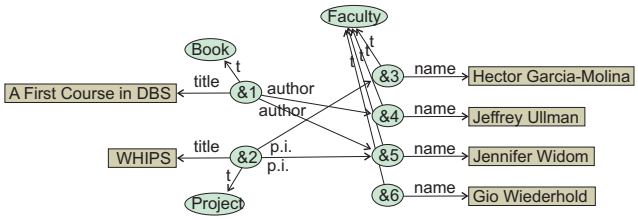

Figure 10: Excerpt of relationships in the Stanford DB Group

Faculty	Relative similarity ($\vec{\sigma}_{rel}$)	Faculty
Hector	0.40	Jennifer
	0.14	Jeff, Gio
Jeff	0.40	Jennifer
	0.14	Hector, Gio
Jennifer	0.32	Jeff, Hector
	0.11	Gio
Gio	0.19	Hector, Jennifer, Jeff

Table 8: Relatedness of faculty members in the DB group based on data in Figure 10

of information about four faculty members of the Stanford Database Group. The data says that Jennifer works with Hector on the project WHIPS and that she wrote a textbook together with Jeff. Table 8 shows the relative similarities between the faculty members. The match result was obtained using the trivial script result = SFJoin(G, G). 'Perfect' match candidates with $\vec{\sigma}_{rel}$= 1 like (Gio, Gio) are omitted in the table for brevity. Also, we substituted the identifiers of the faculty members by their names, e.g. &5 by Jennifer. Since relative similarity is not symmetric, Jeff is related to Jennifer closer than Jennifer to Jeff.

Due to space limitations, we cannot discuss further applications of the algorithm in this paper. Other applications that we used in our experiments include matching of ER, UML and RDFS schemas, comparing product catalogs, approximate queries, matching of service invocations, and matching of mappings. To summarize, notice that the examples that we discussed in this section differ quite a lot from each other. They illustrate diverse application scenarios, the semantics of the nodes in the respective graph representations is different, even the matching goals vary. Common to all these examples is, however, that different matching tasks could be addressed in a uniform fashion

using a very limited amount of custom code. In all scenarios, the similarity flooding algorithm could be deployed by providing converters into graph representation for native formats and selecting the desired subsets from the result of SFJoin. In the next section we provide a quantitative estimate of the quality of match results delivered by the algorithm.

6 Assessment of Matching Quality

In this section, we suggest a metric for measuring the quality of automatic matching algorithms. A crucial issue in evaluating matching algorithms is that a precise definition of the desired match result is often impossible. In many applications the goals of matching depend heavily on the intension of the users, much like the users of an information retrieval system have varying intensions when doing a search. Typically, a user of an information retrieval system is looking for a good, but not necessarily perfect search result, which is generally not known. In contrast, a user performing say schema matching is often able to determine the perfect match result for a given match problem. Moreover, the user is willing to adjust the result manually until the intended match has been established. Thus, we feel that the quality metrics for matching tasks that require tight human quality assessment need to have a slightly different focus than those developed in information retrieval.

The quality metric that we suggest below is based upon *user effort needed to transform a match result obtained automatically into the intended result*. We assume a strict notion of matching quality i.e. being close is not good enough. For example, imagine that a matching algorithm comes up with five equally plausible match candidates for a given element, then decides to return only two of them, and misses the intended candidate(s). In such case, we give the algorithm zero points despite the fact that the two returned candidates might be very similar to what we are looking for. Moreover, our metric does not address semiautomatic matching, in which the user iteratively adjusts the result and invokes repeatedly the matching procedure. Thus, the accuracy results we obtain here can be considered 'pessimistic', i.e., our matching algorithm may be 'more useful' that what our metric predicts.

Matching accuracy Our goal is to estimate how much effort it costs the user to modify the proposed match result $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ into the intended result $I = \{(a_1, b_1), \ldots, (a_m, b_m)\}$.

The user effort can be measured in terms of additions and deletions of map pairs performed on the proposed match result P. One simplified metric that can be used for this purpose is what we call *match accuracy*. Let $c = \|P \cap I\|$ be the number of correct suggestions. The difference $(n - c)$ denotes the number of false positives to be removed from P, and $(m - c)$ is the number of false negatives, i.e. missing matches that need to be added. For simplicity, let us assume that deletions and additions of match pairs require the same amount of effort, and that the verification of a correct match pair is free. If the user performs the whole matching procedure manually (and does not make mistakes), m add operations are required. Thus, the portion of the manual clean-up needed after applying the automatic matcher amounts to $\frac{(n-c)+(m-c)}{m}$ of the fully manual matching.

We approximate the labor savings obtained by using an automatic matcher as accuracy of match result, defined as $1 - \frac{(n-c)+(m-c)}{m}$. In a perfect match, $n = m = c$, resulting in accuracy 1. Notice that $\frac{c}{m}$ and $\frac{c}{n}$ correspond to recall and precision of matching [LC00]. Hence, we can express match accuracy as a function of recall and precision as follows:

$$\text{Accuracy} = 1 - \frac{(n-c)+(m-c)}{m} = \frac{c}{m}(2 - \frac{n}{c}) = \text{Recall}\left(2 - \frac{1}{\text{Precision}}\right)$$

In the above definition, the notion of accuracy only makes sense if precision is not less than 0.5, i.e at least half of the returned matches are correct. Otherwise, the accuracy is negative. Indeed, if more than a half of the matches are wrong, it would take the user more effort to remove the false positives and add the missing matches than to do the matching manually from scratch. As expected, the best accuracy 1.0 is achieved when both precision and recall are equal to 1.0. Notice that accuracy is biased towards precision. For example, recall/precision measure (0.7, 0.9) corresponds to accuracy 0.62. This accuracy value is higher than that for (0.9, 0.7), which amounts to 0.51.

Intended match result Accuracy, as well as recall and precision, are relative measures that depend on the *intended match result*. For a meaningful assessment of match quality, the intended match result must be specified precisely. Recall our example dealing with relational schemas that we examined in Section 2. Three plausible match results for this example (that we call Sparse, Expected, and Verbose) are presented in Table 9. A plus sign (+) indicates that the map pair shown on the right is contained in the corresponding desired match result. For example, map pair

Sparse	Expected	Verbose	Node in G_1	Node in G_2
	+	+	[Table: Personnel]	[Table: Employee]
		+	[Table: Personnel]	[Table: Department]
	+	+	[UniqueKey: perskey]	[PrimaryKey: on EmpNo]
+	+	+	[Column: Personnel/Dept]	[Column: Department/DeptName]
		+	[Column: Personnel/Dept]	[Column: Department/DeptNo]
		+	[Column: Personnel/Dept]	[Column: Employee/DeptNo]
+	+	+	[Column: Personnel/Pno]	[Column: Employee/EmpNo]
+	+	+	[Column: Personnel/Pname]	[Column: Employee/EmpName]
+	+	+	[Column: Personnel/Born]	[Column: Employee/Birthdate]

Table 9: Three plausible intended match results for matching problem in Figure 1

(`[Table: Personnel]`, `[Table: Employee]`) belongs to both Expected and Verbose intended results. The Expected result is the one that we consider the most natural one. The Verbose result illustrates a scenario where matches are included due to additional information available to the human designer. For example, the data in table `Personnel` is obtained from both `Employee` and `Department`, although this is not apparent just by looking at the schemas. Similarly, the Sparse result is a matching where some correspondences have been eliminated due to application-dependent semantics. Keep in mind that in the Sparse and Verbose scenarios, the human selecting the 'perfect' matchings has more information available than our matcher. Thus, clearly we cannot expect our matching algorithm to do as well as in the Expected case.

Accuracy, precision, and recall obtained for all three intended results using version C of the flooding algorithm (see Table 3) are summarized in Figure 11. For each diagram, we executed a script like the one presented in Section 2. The SelectThreshold operator was parameterized using t_{rel}-threshold values ranging from 0.6 to 1.0. As an additional last step in the script, we applied operator SQLDDLMapFilter that eliminates all matches except those between tables, columns, and keys. As shown in the figure, match accuracy 1.0 is achieved for $0.95 \leq t_{rel} \leq 1.0$ in the Expected match, i.e., no manual adjustment of the result is required from the user. In contrast, if the intended result is Sparse, the user can save only 50% of work at best. Notice that the accuracy quickly becomes negative (precision goes below 0.5) with decreasing threshold values. Using no threshold filter at all, i.e. $t_{rel} = 0$, yields recall of 100% but only 4% precision, and results in a desastrous accuracy

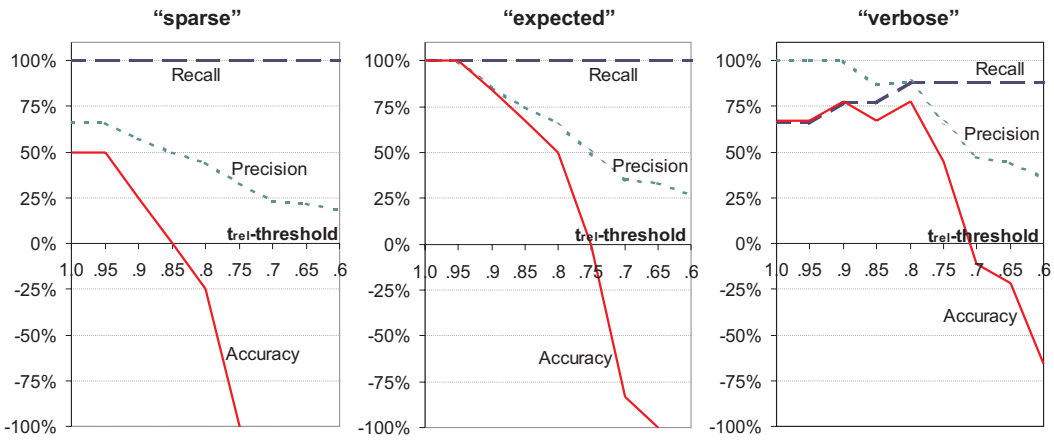

Figure 11: Matching accuracy as a function of t_{rel}-threshold for intended match results Sparse, Expected, and Verbose from Table 9

value of –2144% (not shown in the figure). Increasing threshold values corresponds to the attempt of the user to quickly prune undesired results by adjusting a threshold slider in a graphical tool.

Figure 11 indicates that the quality of matching algorithms may vary significantly in presence of different matching goals. As mentioned earlier, our definition of accuracy is pessimistic, i.e., the user may save more work as indicated by the accuracy values. The reason for that is twofold. On the one hand, if accuracy goes far below zero, the user will probably scrap the proposed result altogether and start from scratch. In this case, no additional work (in contrast to that implied by negative accuracy) is required. On the other hand, removing false positives is typically less labor-intensive than finding the missing match candidates. For example, consider the data point $t_{rel} = 0.75$ in the Expected diagram. The matcher found all 6 intended map pairs (100% recall), and additionally returned 6 false positives (50% precision) resulting in an accuracy of 0.0. Arguably, removing these false positives requires less work as compared to starting with a blank screen.

7 Evaluation of algorithm and filters

To evaluate the performace of the algorithm for schema matching tasks, we conducted a user study with help of eight volunteers in the Stanford Database Group. The study also helped us to examine how different filters and parameters of the algorithm affect the match results. For our study we used

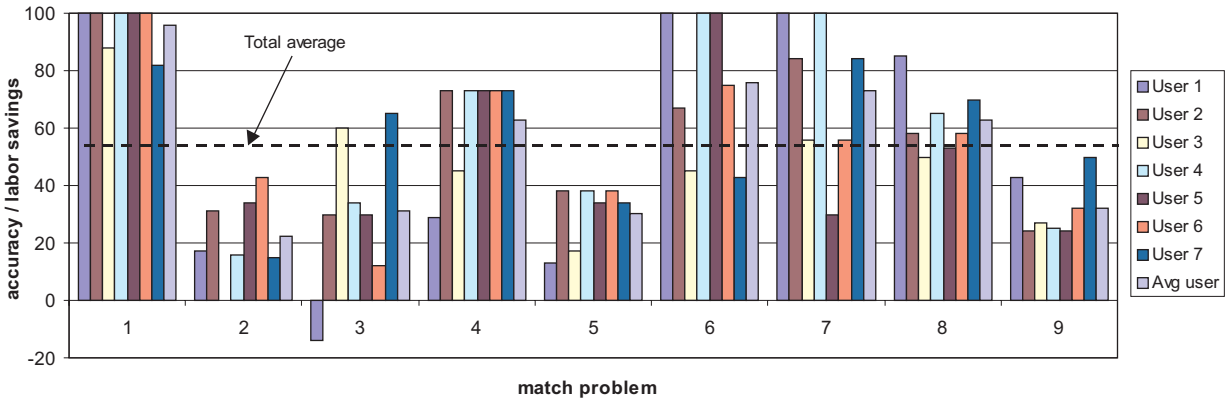

Figure 12: Average matching accuracy for 7 users and 9 matching problems

nine relatively simple match problems[1]. Some of the problems were borrowed from reseach papers [MHH00, DDH01, RB01]. Others were derived from data used on the websites like Amazon.com or Yahoo.com. Every user was required to solve tasks of three different kinds (shown along the x-axis of Figure 12):

1. matching of XML schemas (Tasks 1,2,3)
2. matching of XML schemas using XML data instances (Tasks 4,5,6)
3. matching of relational schemas (Tasks 7,8,9)

The information provided about the source and target schemas was intentionally vague. The users were asked to imagine a plausible scenario and to map elements in both schemas according to the scenario they had in mind. No cardinality constraints were given (any $[0,n]-[0,n]$ mapping was accepted). Noteworthy is that almost no two users could agree on the intended match result for a given matching task, even when examples of data instances were provided (tasks 4,5,6). Therefore, we could hardly expect any automatic procedure to produce excellent results. From eight users, one outlier (i.e. the user with highly deviating results) was eliminated. The accuracy in percent achieved by our algorithm (using fixpoint formula C) for each of the seven users and every task is summarized

[1]The complete specification of the match tasks handed out to the users is available at `http://www-db.stanford.edu/~melnik/mm/sfa/`

Task	Edges in propagation graph	Edges/arcs in left model	Edges/arcs in right model
T_1	128	35/39	32/37
T_2	313	37/43	40/46
T_3	376	46/46	49/52
T_4	383	55/62	39/44
T_5	309	36/41	48/48
T_6	571	66/55	54/45
T_7	339	33/31	69/55
T_8	1222	113/78	66/51
T_9	594	113/78	32/30

Table 10: Sizes of graphs in the user study

in Figure 12. The accuracy metric was used to estimate the amount of work that a given user could save by using our algorithm. The accuracy data was obtained after applying SelectThreshold operator with $t_{rel} = 1$. Negative accuracy of –14% in Task 3 indicates that User 1 would have spent 14% more work adjusting the automatic match result than doing the match manually.

Note that in Task 1 the algorithm performed very well, while in Task 2 the results were poor. It turned out that the models used in Task 2 had very simple structure, so that the algorithm was mainly driven by the initial textual match. We did not use any dictionaries for string matching in any of the experiments reported in this paper. Hence, the synonyms used in Task 2 were considered as plausible matches by humans but were not recognized by the algorithm. The matching accuracy over 7 users and 9 problems averaged to 52%. Hence, our study suggests that for many matching tasks, as much as a half of manual work can be saved using very little application-specific code. This figure is typically even higher in simpler tasks, e.g. when matching two XML documents conforming to the same DTD. Using synonyms may further improve the results of matching. For completeness, the sizes of graphs obtained from schemas used in the study are summarized in Table 10.

Using matching accuracy as the quality measure, we utilized the data collected in the user study to drive our evaluation and tuning of the algorithm for schema matching. As a result of this evaluation, we determined the parameters of the algorithm and the filter that performed best on average for all users and matching problems in our study. The variations of the fixpoint formula that we used are depicted in Table 3 (compare Appendix B). Using distinct fixpoint formulas results

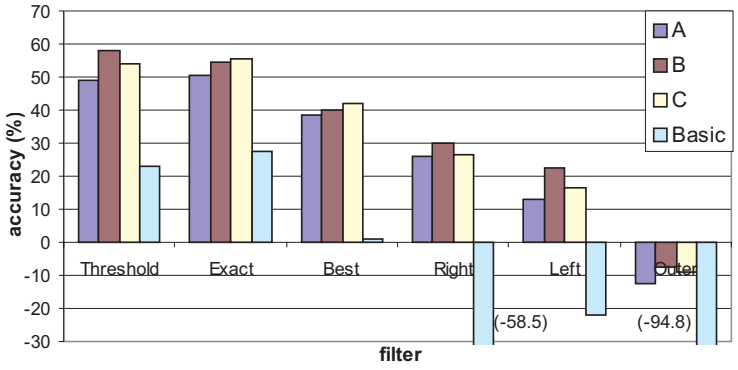

Figure 13: Matching accuracy for different filters and four versions of the algorithm

in different multimappings produced by the algorithm as well as different convergence speed. We then applied different filters to choose the best subsets of multimappings. Figure 13 summarizes the accuracy (averaged over all tasks) obtained for every version of the algorithm and filter that we used. The filters were defined as follows:

- *Threshold* filter corresponds to the SelectThreshold operator described in Section 4. It produces mappings of cardinality $[0, n] - [0, n]$ using relative-similarity threshold $t_{rel} = 1.0$.
- *Exact* is a $[0, 1] - [0, 1]$ version of *Threshold*, which yields monogamous societies.
- *Best* returns a $[0, 1] - [0, 1]$ mapping using a selection metric that corresponds to the assignment problem. The implementation of the filter uses a greedy heuristic. For the next unmatched element, a best available candidate is chosen that maximizes the cumulative similarity.
- *Left* yields a $[0, 1] - [1, 1]$ mapping, in which every node on the left is assigned a match candidate that maximizes the cumulative similarity. *Right* is a $[1, 1] - [0, 1]$ counterpart of *Left*.
- *Outer* filter delivers a $[1, n] - [1, n]$ mapping, in which every node on the left and on the right is guaranteed to have at least one match candidate.

As suggested by Figure 13, the best overall accuracy of 57.9% was achieved using *Threshold* filter with the fixpoint formula B. The accuracy of *Threshold* and *Exact* filters lie very close to each other. This is not surprising, since *Threshold* with $t_{rel} = 1.0$ typically produces $[0, 1] - [0, 1]$

mappings. In our study, *Right* consistently outperforms *Left*, since in most matching tasks the right schemas were smaller; nodes in right schemas were therefore more likely to appear in the intended match results supplied by the users. *Outer* performed worst, since in many tasks only small portions of schemas were intended to have matching counterparts.

We tried to estimate the usefulness of other filters, which are either hard to implement or require extensive computation, by using sampling. For example, a filter that returns a maximal matching (a $[0,1]-[0,1]$ mapping with the most map pairs) is apparently not an optimal one for schema matching. Under formula B, the total number of map pairs in all tasks after applying the *Best* filter is 101, with associated accuracy of 40%. This accuracy value is lower than 54% obtained using the *Exact* filter that yields only 73 map pairs. Overall, our study suggests that preserving the stable-marriage property is desirable for selecting subsets of multimappings.

Notice that the fixpoint formulae A, B, and C yield comparable matching accuracy for each filter. However, formula C has much better convergence properties, as suggested by Table 11. The table shows the number n of iterations that were required in every task to obtain a residual vector $\|\Delta(\sigma^n, \sigma^{n-1})\| < 0.05$. For every fixpoint formula, we executed the algorithm in two versions, 'as is' and 'strongly connected'. Strongly connected versions guarantee convergence. This effect is achieved by making σ^0 contain positive similarity values (e.g. at least 0.001) for each map pair in the cross-product of nodes of left and right schemas. We found experimentally that the strongly connected versions of the algorithm yielded approximately the same overall accuracy for the filters that preserve the stable-marriage property (*Threshold*, *Exact*, and *Best*). In contrast, enforcing convergence had a substantial negative impact on accuracy for the filters *Left*, *Right*, and *Outer*. For a detailed dicussion of convergence criteria please refer to Appendix D.

The formula for computing the propagation coefficients in the induced propagation graph is another important configuration parameter of the flooding algorithm. We experimented with seven distinct formulae and determined the one that performed best in our user study. For the details of this experiment please refer to Appendix B. The best-performing formula is based on the *inverse average* of equilabeled edges in the graphs to be matched. This approach is similar to the one illustrated in Section 3, which corresponds to inverse product, and performs only slightly better.

Fixpoint formula	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	Total
A (as is)	18	48	122	78	∞	12	37	25	25	∞
A (strongly connected)	15	56	89	81	1488	18	48	25	31	1851
B (as is)	8	428	17	39	8	13	10	24	21	568
B (strongly connected)	7	268	21	32	13	15	14	21	53	444
C (as is)	7	9	9	11	7	7	9	10	9	78
C (strongly connected)	7	9	8	11	7	5	9	7	9	72

Table 11: Illustration of convergence properties of variations of fixpoint formula for tasks $T_1, \ldots, T_9$ in the user study. Shows iterations needed until length of residual vector got below 0.05.

As a last experiment in this section, we study the impact of the initial similarity values (σ^0) on the performance of the algorithm. For this purpose, we randomly distorted the initial values computed by the string matcher. The initial similarities were computed using two versions of a string matcher, one of which took term frequencies into account. Figure 14 depicts the influence of randomization on matching accuracy across all users and matching tasks. For example, randomization of 50% means that every initial similarity value was randomly increased or decreased by x percent, $x \in [-50\%, 50\%]$. Negative similarity was adjusted to zero. It is noteworthy that a randomization factor of 100% introduced accuracy penalty of just about 15%. This result indicates that the similarity flooding algorithm is relatively robust against variations in seed similarities. The dotted lines show another radical modification of initial similarities, in which each non-zero value in σ^0 was set to the same number computed as the average of all positive similarity values. In this case, the accuracy dropped to 30%, which still saves the users on average one third of the manual work.

To summarize, the main results of our study were the following:

- For an average user, overall labor savings across all tasks were above 50%. Recall from Section 6 that our accuracy metric gives a pessimistic estimate, i.e. actual savings may be even higher.
- A quickly converging version of the fixpoint formula (C) did not introduce accuracy penalties.
- *Threshold* filter performed best.
- The best formula for computing the propagation coefficients was the one based on inverse average (Appendix B).

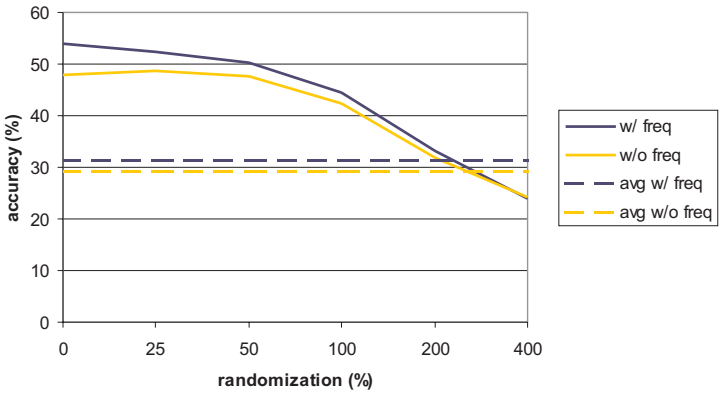

Figure 14: Impact of randomizing initial similarities on matching accuracy

- The flooding algorithm is relatively insensitive to 'errors' in initial similarity values.

8 Architecture and Implementation

The architecture of our testbed is depicted in Figure 15. A central component of this architecture is the interpreter that executes scripts like those listed throughout the paper. The script interpreter uses the operators defined in the operator library. In the current implementation, every operator corresponds to a Java class compliant to a certain interface. The semantics of the script language we are using now is extremely simple. Every operator takes a list of models as input and produces a list of models as output. Load/store and import/export operators are an exception, since they accept additional parameters that are not models. Recall that mappings are models and therefore can be used whenever model is expected as a parameter. For compactness of scripts, operators can be nested. Schemas and instances in the native format like XML or SQL DDL files are stored in the local file system. The models can additionally be loaded and stored in a (remote) metadata repository. The repository is an SQL-compatible database. The interpreter communicates with the repository via JDBC.

Covering all implemented operators and their usage goes beyond the scope of this paper. Here we are providing only some examples. SFJoin operator implements the similarity flooding algorithm and is the subject of the paper. StringMatch provides a hint how literal nodes in one graph match

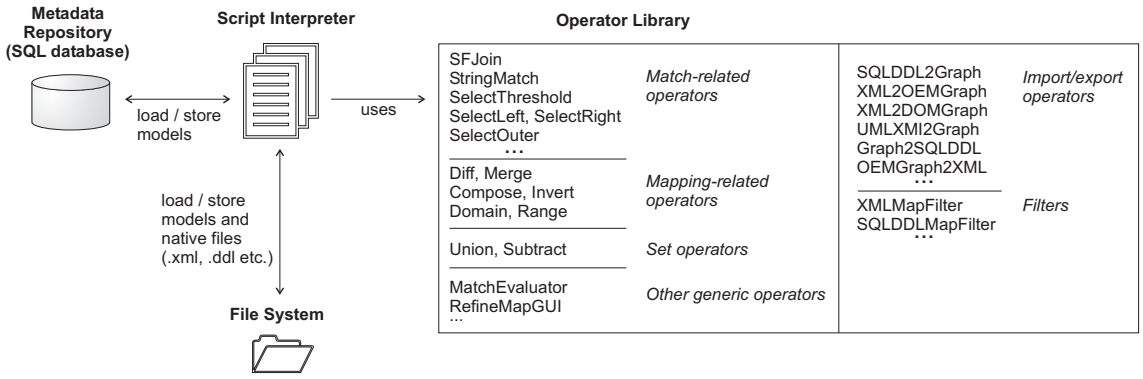

Figure 15: Architecture and implemented operators

those in another. The string matcher we are currently using splits a text string into a set of words and compares the word in two sets pairwise. In word comparison, we examine only common prefix and suffix. Optionally, term frequencies are used to reduce the impact of common terms in large schemas. An example of string matching was given in Table 1. Operator SQLDDLMapFilter is an application-specific operator that filters out all matches from a mapping except column, table and key matches. MatchEvaluator is an operator that implements a tool for evaluating the quality of match results. Operator RefineMapGUI provides a general-purpose user interface that supports quick pruning of match results using several threshold sliders. Mapping-related operators like Compose or Merge are implemented similarly to the definitions suggested in [BLP00] and are work in progress.

9 Open Issues and Limitations

Below we summarize the limitations of the algorithm and several open issues that need to be investigated. This list is by no means exhaustive:

1. The algorithm works for directed labeled graphs only. It degrades when labeling is uniform or undirected, or when nodes are less distinguishable. For example, the algorithm does not perform well for solving the graph isomorphism problem on undirected graphs having no edge labels.

2. Applicability of the algorithm is limited to equityped models. While matching of an XML schema against another XML schema delivers usable results, matching of a relational schema

against an XML schema fails.

3. An important assumption behind the algorithm is that adjacency contributes to similarity propagation. Thus, the algorithm will perform unexpectedly in cases when adjacency information is not preserved. For example, in HTML pages nodes that are structurally far away from each other may be displayed visually close. Thus, two cells in an HTML table that are vertically adjacent may be far apart in the document and won't contribute to similarity propagation.

4. The algorithm tends to favor superstructures. Consider graph A containing subgraph A_1. Let graph B contain a superstructure B_1 such that $A \subset B_1$ and a substructure B_2 such that $B_2 \subset A$. The algorithm would favor B_1 as a match candidate for A, i.e. similarity values between nodes in A and B_1 will be higher that those between A and B_2.

5. Currently, we do not consider order and aggregation in the algorithm. Matching of certain types of data like XML could benefit from taking this information into account.

6. It is unlikely that a standalone version of the algorithm could outperform custom matchers developed for a particular domain. Custom matchers may deploy domain-specific heuristics that are not available to the similarity flooding algorithm (e.g. value ranges, cardinalities, classifiers etc.). However, the algorithm can make use of custom import and export filters like XMLMapFilter mentioned in Section 5 to prototype a first-cut version of a specialized matcher quickly.

10 Related Work

Our work was inspired by model management scenarios presented in the vision paper [BLP00] by Bernstein et al. In particular, our scripts use similar high-level operations on models. Such an approach can significantly simplify the development of metadata-based tasks and applications compared to the use of current metadata repositories and their low-level APIs.

A recent classification and review of matching techniques can be found in [RB01]. Most of the previously proposed approaches lack genericity and are tailored to a specific application domain

such as schema or data integration, and specific schema types such as relational or XML schemas. Moreover, most approaches are restricted to finding 1:1 matching correspondences. A few promising approaches not only use schema-level but also instance-level information for schema matching [LC00, DDH01]. Unfortunately, their use of neural networks [LC00] or machine learning techniques [DDH01] introduces additional preparing and training effort. Concurrently and independently to the work reported in this paper, a generic schema matching approach called Cupid was developed at Microsoft Research [MBR01]. It uses a comprehensive name matching based on synonym tables and other thesauri as well as a new structural matching approach considering data types and topological adjacency of schema elements. Many other studies have used more sophisticated linguistic (name/text) matchers compared to our very simple string matcher, e.g. WHIRL [Coh98]. [MHH00] has addressed the related problem of determining mapping expressions between matching elements.

In general, match algorithms developed by different researchers are hard to compare since most of them are not generic but tailored to a specific application domain and schema types. Moreover, as we have discussed in Section 6, matching is a subjective operation and there is not always a unique result. Previously proposed metrics for measuring the matching accuracy [LC00, DDH01] did not consider the extra work caused by wrong match proposals. Our accuracy metric is similar in spirit to measuring the length of edit scripts as suggested in [CGM97]. However, we are counting the edit operations on mappings, rather than those performed on models to be matched.

In designing our algorithm and the filters, we borrowed ideas from three research areas. The fixpoint computation corresponds to random walks over graphs [MR95], as explained in Appendix D. A well-known example of using fixpoint computation for ranking nodes in graphs is the PageRank algorithm used in the Google search engine [BP98]. Unlike PageRank, our algorithm has two source graphs and extensively uses and depends on edge labeling. The filters that we proposed for choosing subsets of multimappings are based on the intuition behind the class of stable marriage problems [GI89]. General matching theory and algorithms are comprehensively covered in [LP86]. Finally, the quality metric that we use for evaluating the algorithm is related to the precision/recall metrics developed in the context of information retrieval.

The data model used in this paper is based on the RDF model [LS98]. For transforming native data into graphs we use graph-based models defined for different application e.g.[OIM99], [PGMW95]

or [DOM98].

11 Conclusion

In this paper we presented a simple structural algorithm based on fixpoint computation that is usable for matching of diverse data structures. We illustrated the applicability of the algorithm to a variety of scenarios. We defined several filtering strategies for pruning the immediate result of the fixpoint computation. We suggested a novel quality metric for evaluating the performance of matching algorithms, and conducted a user study to determine which configuration of the algorithm and filters performs best in chosen schema matching scenarios. We discussed the convergence and complexity of the algorithm, and summarized the known limitations.

References

[BLP00] P. A. Bernstein, A. Levy, and R. Pottinger. A Vision for Management of Complex Models. *SIGMOD Record*, pages 55–63, 2000.

[BP98] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proc. WWW7 Conf.* Computer Networks, 1998.

[CGM97] S. S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *Proc. SIGMOD'97*, pages 26–37, 1997.

[Coh98] W. W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. In *Proc. SIGMOD 1998*, pages 201–212, 1998.

[DDH01] A. Doan, P. Domingos, and A. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *Proc. SIGMOD 2001*, 2001.

[DOM98] XML Document Object Model (DOM), W3C Recommendation. `http://www.w3.org/TR/REC-DOM-Level-1/`, October 1998.

[GI89] D. Gusfield and R. Irving. *The Stable Marriage Problem: Structure and Algorithms.* MIT Press, Cambridge, MA, 1989.

[Kan00] M. Kanehisa. *Post-Genome Informatics.* Oxford University Press, 2000.

[LC00] W.-S. Li and C. Clifton. SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Trans. on Data & Knowledge Engineering*, pages 49–84, 2000.

[LP86] L. Lovàsz and M. Plummer. *Matching Theory.* North-Holland, Amsterdam, 1986.

[LS98] O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. `http://www.w3.org/TR/REC-rdf-syntax/`, 1998.

[MBR01] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. Proc. VLDB 2001 (to appear), 2001.

[MHH00] R. J. Miller, Laura M. Haas, and Mauricio A. Hernandez. Schema Mapping as Query Discovery. In *Proc. VLDB 2000*, 2000.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[OIM99] Open Information Model, Version 1.0, Meta Data Coalition. `http://mdcinfo.com/oim/oim10.html`, August 1999.

[PGMW95] Y. Papakonstantinou, H. García-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proc. of the 11th IEEE Int. Conf. on Data Engineering (ICDE)*, pages 251–260, Taipe, Taiwan, March 1995.

[RB01] E. Rahm and P. A. Bernstein. On Matching Schemas Automatically. Technical Report MSR-TR-2001-17, http://www.research.microsoft.com/pubs/, February 2001.

A Internal Data Model

In this section we present a more formal definition of models and mappings. Let $\mathcal{U}$ be the Unicode alphabet and $\mathcal{U}^*$ the set of strings defined over $\mathcal{U}$. The set of entities $\mathcal{E}$ and the set of statements $\mathcal{V}$ are defined using the following recursive definition:

1. $\mathcal{U}^* \times \mathcal{U}^* \subset \mathcal{E}$ (any tuple consisting of two strings is an entity; the first string of the tuple is called *type* or *namespace* of the entity, the second string is referred to as *name* of the entity)
2. $\mathcal{E} \times \mathcal{E} \times \mathcal{E} \subset \mathcal{V}$ (every tuple of three entities constitutes a *statement*)
3. $\mathcal{V} \subset \mathcal{E}$ (every statement is an entity)
4. $\mathcal{V}$ and $\mathcal{E}$ are the smallest sets with the above properties.

A subset of $\mathcal{V}$ is called *model*. The above definition and terminology are based on the RDF standard [LS98]. As the recursive definition of $\mathcal{V}$ and $\mathcal{E}$ suggests, statements can be nested, i.e. a statement can be used as an element of another statement. In our internal data model nested statements are used for representing ordered relationships and aggregation. Currently, we are not using any of these aspects in the matching algorithm presented in this paper. Therefore, we omit further discussion of nested statements. We can make a simplifying assumption $\mathcal{E} = \mathcal{U}^* \times \mathcal{U}^*$ and $\mathcal{V} = \mathcal{E}^3$. Thus, a model is a subset of $\mathcal{E}^3$.

Models can be represented graphically as shown in Figure 2. In this representation, entities correspond to graph nodes, whereas statements are depicted as edges. For any statement (s, p, o) the middle element p of the tuple (called *predicate*) is depicted as the label of the edge. A set of statements with the same predicate defines a binary relationship over entities.

In Figure 2 we distinguished two different kinds of nodes, those represented as ovals and those shown as rectangles. Nodes shown as rectangles are called *literals* and belong to the subset of entities $\mathcal{L} = \{\text{"literal"}\} \times \mathcal{U}^*$. There is no principal distinction between literals and other entities. We distinguish literals graphically mainly for better readability.

The relation *belongsTo* describes the containment of entities in a model, and is defined as shown below. Notice that this simplified definition can be used only if we disallow nested statements:

$$\forall M \subset \mathcal{V}, n \in \mathcal{E} : \mathit{belongsTo}(n, M) \iff \exists (s, p, o) \in M : n \in \{s, p, o\}$$

A *mapping* between models M_1 and M_2 can be viewed conceptually as a set of tuples (n_1, n_2, σ) such that $belongsTo(n_1, M_1), belongsTo(n_2, M_2)$ and σ is a real number that serves as a similarity measure. When M_1 and M_2 share no elements in common, a mapping can be defined as a weighted undirected bipartite graph. To treat mappings as models, a mapping is represented as a set of statements. For every tuple $t = (n_1, n_2, \sigma)$ we create four statements:

1. $(node(t), \mathsf{type}, \mathsf{MapEntry})$
2. $(node(t), \mathsf{src}, n_1)$
3. $(node(t), \mathsf{dest}, n_2)$
4. $(node(t), \mathsf{similarity}, \sigma)$

where type, MapEntry, src, dest, similarity are constants from $\mathcal{E}$, $node()$ is a Skolem function that returns a unique element from $\mathcal{E}$, and $\sigma \in \{\text{"real"}\} \times \mathcal{E}$. Notice that the ability of viewing mappings as models is essential for the orthogonality of the internal data model. Thus, algorithms that work on models can be applied to mappings without changes. In this paper, we favor the graph-oriented terminology 'nodes' and 'edges' over 'entities' and 'statements'. For brevity, we use the abbreviation $n \in M$ for $belongsTo(n, M)$.

B Generalized version of the algorithm

The core of the formal definition of the algorithm is based on the function φ that takes a mapping σ as input parameter and produces mapping θ as output. For any two given models A and B, φ is defined as follows:

$$\varphi(\sigma) = \theta \iff \forall (a,b) \in A \times B: \quad \theta(a,b) = \sum_{(a,p,x)\in A,(b,q,y)\in B} \sigma(x,y) \cdot \pi_r(\langle x,p,A\rangle, \langle y,q,B\rangle) + \sum_{(x,p,a)\in A,(y,q,b)\in B} \sigma(x,y) \cdot \pi_l(\langle x,p,A\rangle, \langle y,q,B\rangle)$$

Function φ describes how the similarity of the neighbor pairs of (a, b) 'flows' into the similarity of (a, b). Function π defines the propagation coefficients for a map pair (x, y) with respect to p-labeled edges in A and q-labeled edges in B. The π-function that corresponds to the example described in

Section 3 is based on inverse-product number of equilabeled edges in A and B computed for each map pair:

$$\pi_{\{l,r\}}(\langle x,p,A\rangle,\langle y,q,B\rangle) = \begin{cases} \frac{1}{card_{\{l,r\}}(x,p,A)\cdot card_{\{l,r\}}(y,q,B)}, & \text{if } p=q \\ 0, & \text{if } p\neq q \end{cases}$$

where $card(x,p,M)$ delivers the number of outgoing or incoming edges of node x that carry label p in model M:

$$\begin{aligned} \forall M\in\mathcal{V}^3,\forall x\in\mathcal{E},\forall p\in\mathcal{E}: \quad card_l(x,p,M) &= \|\{(x,p,t)\mid \exists t:(x,p,t)\in M\}\| \\ card_r(x,p,M) &= \|\{(t,p,x)\mid \exists t:(t,p,x)\in M\}\| \end{aligned}$$

The definitions of functions φ and π use A and B directly without relying on the pairwise connectivity graph. This is a more general approach, since the propagation graph typically contains more information than the connectivity graph. For example, the propagation coefficients obtained using a π-function based on inverse average (described below) cannot be computed using just the connectivity graph. Finally, in the definition of our algorithm we rely on summation and normalization of mappings. These two operations are defined as follows. The sum of mappings σ and ν is a mapping θ such as:

$$\forall(x,y)\in A\times B: \quad \theta(x,y)=\sigma(x,y)+\nu(x,y)$$

The function *normalize* projects all similarity values of a mapping into the range $[0,1]$. That is, normalization corresponds to dividing vector σ by a scalar value that represents the highest similarity value in σ:

$$\theta = normalize(\sigma) \iff \forall(a,b)\in A\times B: \quad \theta(a,b)=\frac{\sigma(a,b)}{max\{s\mid \exists x,y:\sigma(x,y)=s\}}$$

Now we can define the main iteration step of our algorithm. In the version of the algorithm illustrated in Section 3, on every iteration, a set of new similarity values is computed as follows:

$$\sigma^{i+1} = normalize(\sigma^i+\varphi(\sigma^i))$$

The above computation is performed iteratively until $\Delta(\sigma^n,\sigma^{n-1})$ satisfies a chosen precision goal for some $n>0$. To ensure convergence and efficiency (compare Table 11), we use a variation of the algorithm shown below. Our user study suggests that the faster converging version of the algorithm

Approach	$p = q$	$p \neq q$
inverse average	$\frac{2}{card_{\{l,r\}}(x,p,A) + card_{\{l,r\}}(y,q,B)}$	0
inverse product	$\frac{1}{card_{\{l,r\}}(x,p,A) \cdot card_{\{l,r\}}(y,q,B)}$	0
inverse total average	$\frac{2}{card_{\{l,r\}}(p,A) + card_{\{l,r\}}(q,B)}$	0
inverse total product	$\frac{1}{card_{\{l,r\}}(p,A) \cdot card_{\{l,r\}}(q,B)}$	0
combined inverse average	$\frac{4}{(card_{\{l,r\}}(p,A) + card_{\{l,r\}}(q,B)) \cdot (card_{\{l,r\}}(x,p,A) + card_{\{l,r\}}(y,q,B))}$	0
stochastic	$\frac{1}{\sum_{\forall p'}(card_{\{l,r\}}(x,p',A) \cdot card_{\{l,r\}}(y,p',B))}$	0
constant	1.0	0

Table 12: Different approaches to computing the propagation coefficients $\pi_{\{l,r\}}(\langle x,p,A\rangle, \langle y,q,B\rangle)$

does not negatively impact the quality of the results. The rationale behind this modification is discussed in Appendix D:

$$\boxed{\sigma^{i+1} = normalize(\sigma^0 + \sigma^i + \varphi(\sigma^0 + \sigma^i))}$$

C Propagation coefficients

The similarity flooding algorithm offers several tuning parameters. One such parameter is the definition of the function π that computes the propagation coefficients in the propagation graph. Above we presented a product-based definition of π that we used to illustrate the algorithm in Section 3. In our user study we found empirically that an average-based definition of π slightly outperformed the product-based one. The average-based π-formula as well as another six approaches to computing the propagation coefficients that we examined are summarized in Table 12.

For example, the stochastic formula ensures that the sum of propagation coefficients on all edges originating from each map pair in the propagation graph is 1.0. Hence, the transition matrix that corresponds to the propagation graph (see Appendix D) becomes a stochastic matrix, i.e. the entries in each column sum to 1. We evaluated the performance of each π-function listed in the table using the data obtained in the user study. Figure 16 summarizes the accuracy values obtained using different π-functions. In this experiment, we used the fixpoint formula B of Table 3 and filters *Threshold* and *Best* to determine the overall average accuracy. We found that the constant

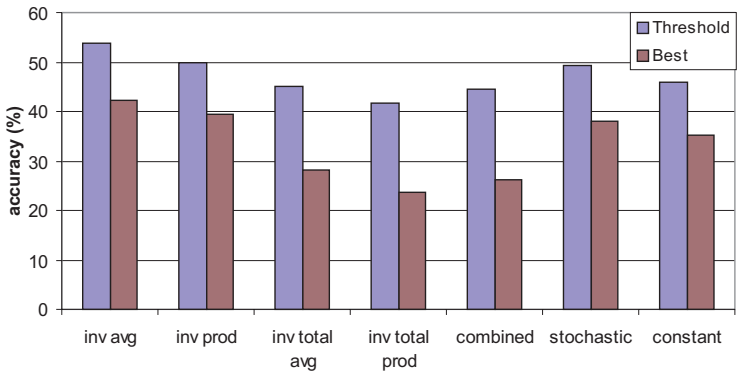

Figure 16: Impact of different ways of computing propagation coefficients on overall matching accuracy in the user study

π-function, which places weights of 1.0 on each edge of the propagation graph, performs surprisingly well as compared to more sophisticated approaches. We did not extensively examine other π-functions that take into account edge-label similarities, i.e. those that return a non-zero value when $p \neq q$.

D Convergence and complexity of the algorithm

The fixpoint computation of the similarity flooding algorithm can be expressed as the following eigenvector computation. Let T be the square matrix corresponding to the similarity propagation graph G obtained from models A and B. If there is an edge going from map pair $j = (x, y)$ to $i = (x', y')$ with propagation coefficient c, then let the matrix entry t_{ij} have the value c. Let all other entries have the value 0. Notice that the propagation coefficients in G correspond to transition probabilities if T is a transition matrix.

The fixpoint computation converges when T is an aperiodic, irreducible matrix (Ergodic theorem). Matrix T is irreducible if and only if the associated graph G is strongly connected (every node is reachable from every other node). To ensure these properties, we can introduce self-loops in G by including the summand σ^0 in the fixpoint equation, for example as $\sigma^{i+1} = normalize(\sigma^0 + \varphi(\sigma^i))$. This approach is also referred to in the literature as dampening. If σ^0 assigns a non-zero value to each map pair in $A \times B$, then adding σ^0 is equivalent to modifying G into G' in which all nodes are

interconnected with certain propagation coefficients. Let T' be the matrix associated with G'.

Now the eigenvector computation can be expressed as follows. Let S be a map pair vector that at every position contains a similarity value from σ for a fixed order of map pairs. One iteration of the fixpoint computation corresponds to the matrix-vector multiplication $T' \times S$. Repeatedly multiplying S by T' yields the dominant eigenvector S^* of the matrix T' such as $T' \times S^* = \lambda S^*$, where λ is the dominant eigenvalue of T'. In the fixpoint equation, normalization corresponds to dividing $T' \times S^*$ by λ.

The fixpoint computation corresponds to computing Markov chains over T. This fact provides an interesting insight into the algorithm. Because T corresponds to the transition matrix over the graph G, the obtained similarity measure can be viewed as the stationary probability distribution over map pairs induced by a random walk from pair to pair. This random walk corresponds to the manual matching process performed by a human designer on models A and B. Starting with a given map pair, the designer infers the similarity of another map pair based on the structural properties of A and B. Consider that A and B are models of relational schemas. If the designer concludes that table t_1 in A matches table t_2 in B, then there is a certain probability that his or her next step will be matching the columns of t_1 to those of t_2.

The conversion rate of the fixpoint computation depends on the ratio between the dominant and the second eigenvalue of T, which are determined by the structural properties[2] of G'. Higher dampening values contribute to a faster conversion rate of the matrix. For a given precision, using both σ^0 and σ^i in the variation $\sigma^{i+1} = normalize(\sigma^0 + \sigma^i + \varphi(\sigma^0 + \sigma^i))$ of the fixpoint formula improves the convergence speed by up to a factor of 5 without impeding the quality of the result.

The convergence of the iterations can be measured using the residual vector $R_i = \dfrac{T' \times S_i}{\lambda_i} - S_i$. We can treat $\|R_i\|$ as an indicator for how well S_i approximates S^*. For many practical purposes we are only interested in the resulting order of map pairs and not in the absolute values of the similarity coefficients. In such cases, the iterations can be interrupted when the order in a certain subset of a mapping with the highest similarity values has stabilized, i.e. does not change from σ^{n-1} to σ^n. In many practical scenarios, this criterion is already satisfied when $\|R_i\| < 0.05$.

Let us now turn to the complexity of the algorithm. The number of operations in every iteration

[2] Asymptotic rate of convergence coincides with the so-called spectral radius of the matrix T'

of the fixpoint computation is proportional to the number of edges in the propagation graph G. This number is in turn proportional to the product of edge numbers in models A and B. Let N_A and N_B be the number of nodes in A and B, respectively. If nodes in A and B are fully interconnected (every node is directly connected to every other node), the edge numbers in A and B are $O(N_A^2)$ and $O(N_B^2)$. If all these edges are equilabeled, the number of edges in G is $O(N_A^2 \cdot N_B^2)$. That means, the worst case complexity of every iteration is $O(N_A^2 \cdot N_B^2)$, or $O(\|A\| \cdot \|B\|)$, where $\|A\|$ and $\|B\|$ are the numbers of edges in A and B. However, in many common scenarios, the average complexity of every iteration is $O(N_A \cdot N_B)$. For typical relational or XML schemas the fixpoint computation converges within 5-30 iterations. That means that the running time of the flooding algorithm is comparable to that of a nested loop join in relational databases (multiplied with a small factor).

A straight-forward implementation of the fixpoint computation requires two occurrences of σ-vectors in memory besides σ^0. The memory usage is important for very large models that may contain parts of dictionaries or classification schemas.