Querying Websites Using Compact Skeletons*

Anand Rajaraman
Cambrian Ventures
201 San Antonio Circle
Mountain View, CA 94040
anand@cambrianventures.com

Jeffrey D. Ullman
Department of Computer Science
Stanford University
Stanford, CA 94305
ullman@cs.stanford.edu

ABSTRACT

Several commercial applications, such as online comparison shopping and process automation, require integrating information that is scattered across multiple websites or XML documents. Much research has been devoted to this problem, resulting in several research prototypes and commercial implementations. Such systems rely on wrappers that provide relational or other structured interfaces to websites. Traditionally, wrappers have been constructed by hand on a per-website basis, constraining the scalability of the system.

We introduce a website structure inference mechanism called *compact skeletons* that is a step in the direction of automated wrapper generation. Compact skeletons provide a transformation from websites or other hierarchical data, such as XML documents, to relational tables. We study several classes of compact skeletons and provide polynomial-time algorithms and heuristics for automated construction of compact skeletons from websites. Experimental results show that our heuristics work well in practice. We also argue that compact skeletons are a natural extension of commercially deployed techniques for wrapper construction.

1. INTRODUCTION

Several commercial applications, such as online comparison shopping and process automation, require integrating information that is scattered across multiple websites or hierarchically structured documents (e.g., in XML). Much research has been devoted to this problem, resulting in several research prototypes and commercial implementations (e.g., [14, 22, 18, 34]). Such systems rely on components called *wrappers* that provide relational or other structured interfaces to websites. Traditionally, wrappers have been constructed by hand on a per-website basis. Constructing these wrappers has been the main bottleneck preventing such systems from scaling; most systems today integrate tens rather than thousands of websites.

*Work partially supported by NSF grant IIS-9811904.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2001 ACM 0-89791-88-6/97/05 ...$5.00.

We introduce a website structure inference mechanism called *compact skeletons* that is a step in the direction of automated wrapper construction. We focus on an important special case: given a single relation scheme, we need to piece together information elements scattered across a website to constitute the relation. For example, suppose our application integrates job listings from websites on the Internet (WhizBang! Labs' FlipDog.com [33] is such a system). Let us say that each job listing has a job title T, a salary S, and an address A for candidates to send their resumes. The relation scheme then is $R(TSA)$.

Figure 1 shows a portion of a corporate website that lists job openings. For the moment, assume that each node corresponds to a web page where we have eliminated all the irrelevant text, leaving behind only the data element of interest. The website is oriented towards a human reader, and to a person it is fairly obvious what the tuples in the relation are. However, even this extremely simple website illustrates some of the difficulties encountered by a program trying to materialize the relation R, such as superfluous information (the job location) and incomplete information (the CEO's compensation package is negotiable).

Given a website such as that in Figure 1 and a target relation scheme, we break up the problem of constructing a wrapper into three subproblems:

1. Identifying the data elements in the scheme, such as addresses, job titles, and salaries.

2. Deducing the principles that have been followed by the person who put together the website (in effect, "reverse engineering" the website).

3. Constructing the relation corresponding to the website. In practice, we need only materialize the portion of the relation that is relevant to the query at hand.

This three-step approach is commonly used in the industry. For example, Whizbang! Labs [34] calls it the "C^4 technique" (where the 4 C's are *crawl, classify, capture, and compile*; we do not include crawling in our taxonomy), while Junglee's Virtual Database Management System [18] has components called *extractors*, *wrappers*, and *mappers* corresponding to these three steps.

A simple way to tackle problem (1) is to use a library of patterns (such as regular expressions). There are several approaches to constructing such patterns: "by hand" by studying several examples [17]; machine learning techniques; and more novel pattern extraction techniques [7]. Our work deals with problems (2) and (3). Once we have identified the

patterns of interest on the pages of a website, we can model the website as a directed graph with data elements at the nodes. There are arcs in the graph corresponding both to structure within a web page (in the case where we identify multiple data elements within a web page) and to hyperlinks between pages. We call such a graph a *data graph*; Figure 1 is an example. In the rest of the of the paper, we model websites as data graphs. Data graphs have been used extensively in the literature to model semistructured data, e.g., in [10, 8, 9, 23, 1, 27, 6, 11].

Compact skeletons are labeled trees that function as transformations between data graphs and relations. Intuitively, a compact skeleton describes the hierarchical layout of the corresponding website: for example, the IBM site groups jobs first by division (D), and each listing includes a job id (I), a job title (T), a job category (C), and the state where the job is in (S). The job title is hyperlinked to details about the job (J) and an address to send resumes to apply for the job (A). Compact skeletons are a natural extension of Junglee's Site Description Language (SDL) [17], which has been used to construct thousands of wrappers for Junglee's VDBMS [18]. We describe the relationship between SDL and compact skeletons in Section 8.

The rest of this paper is organized as follows. In Section 2 we introduce compact skeletons and analyze the properties of *perfect compact skeletons* (PCS), which apply when the data graph has complete information. In Section 3 we relax the completeness condition and introduce *partially perfect compact skeletons* (PPCS) that apply when the data graph has incomplete information, corresponding to null values in relations as in Figure 1. We show that both the PCS and the PPCS possess a desirable property we call *locality*. For a given data graph, we show that the PCS is unique but the PPCS is not; we introduce the notions of *minimal* and *maximal* PPCS that provide upper and lower bounds on the relation associated with the data graph. We describe polynomial-time algorithms to compute the PCS and the minimal and maximal PPCS. In Section 4 we present algorithms for querying websites given a compact skeleton; a special case is to materialize the entire relation corresponding to the website.

Real-life websites often contain *noise* (i.e., superfluous information) in addition to incomplete information. In Sections 5 and 6 we study *best-fit skeletons* (BFS) that apply in such cases. It turns out that computing the BFS is an NP-complete problem. We examine two simple polynomial-time heuristics, the *greedy* and the *weighted greedy*. Experimental results show that the heuristics work well in practice. Section 7 extends the theory to *graph skeletons*; we show that the PCS remains unique and provide a non-deterministic polynomial-time algorithm to compute it. Section 8 examines related work, and Section 9 concludes with some directions for further investigation. The extended version of the paper [28] contains proofs of theorems as well as examples illustrating the algorithms and several details omitted here for conciseness.

2. DATA GRAPHS AND SKELETONS

2.1 Data Graphs

We model websites using data graphs as shown in Figure 1. We restrict our attention to data graphs that are DAGs; we relax this restriction in Section 4 when we discuss generalized skeletons. For simplicity, we assume that each node of the graph has at most one information element (in our example, an instance of A, T, or S). In practice, if a page contains multiple elements of information, we create new nodes corresponding to each element and add an arc from the node corresponding to the page to these newly created nodes. We also ignore information in nodes other than that corresponding to the schema attributes. We assume without loss of generality that if a data graph includes a node, then there is some information element that is reachable from the node. In particular, this assumption means that a node with no outgoing arc must contain an information element.

Let X be the set of attributes in our schema. For attribute A in X, the domain of A, denoted by $Dom(A)$, is the (possibly infinite) set of all possible values that can appear in column A. For simplicity, we assume that domains of attributes in the scheme are pairwise disjoint. We use uppercase letters $A, B, C, \ldots$ for attribute names and corresponding lowercase letters (often with subscripts) for data values corresponding to those attributes, e.g., $a_1, a_2, \ldots \in Dom(A)$.

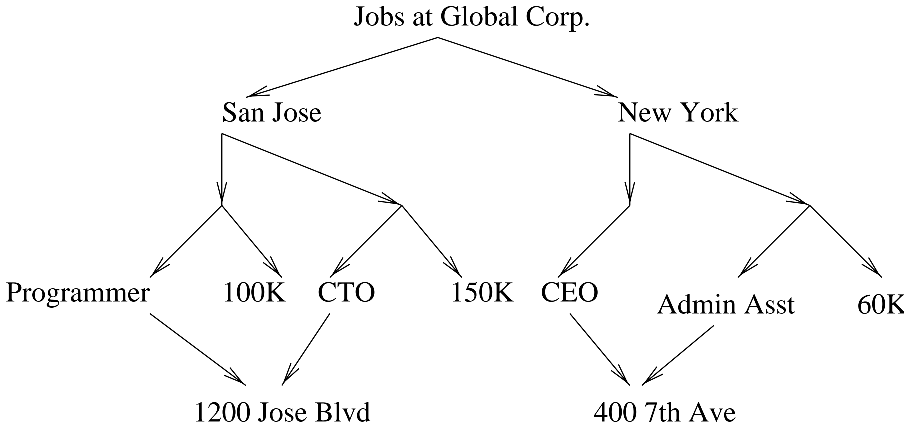
Given a graph G, we use V_G and E_G to denote respectively the node set and the arc set of G. If v is in V_G, we denote by $val(v)$ the value at node v. If there is no value at v, then $val(v) = \perp$, where $\perp$ is a special value that is not in the domain of any attribute. If $val(v)$ is in $Dom(A)$, we say that A is the attribute at node v, written as $attr(v) = A$. We use the convention that if $val(v) = \perp$, then $attr(v) = \perp$.

2.2 Skeletons

Suppose we are given a relation R over attribute set X, and we wish to construct a data graph that incorporates the contents of R. One way to do so is to work incrementally, using the following imaginary process. We construct the data graph corresponding to the first tuple in R, with some nodes and edges. The structure formed by these nodes and edges depends on how we intend to lay out the web site. We then add nodes and edges corresponding to the second tuple, and continue this process until we have the data graph corresponding to the entire relation. If we have been consistent about this process (e.g., as is increasingly common, the website was generated by a program), the manner in which we interconnected the values of the attributes corresponding to the first tuple will be identical to that for the second tuple, and so on. We formalize this notion as follows.

A *skeleton* K is a tree some of whose nodes are labeled with attribute names from X such that each attribute labels exactly one node. If node v in V_K is labeled with attribute A, we say that $attr(v) = A$. There may be nodes of the skeleton that mention no attributes at all. If i is such a node, we say $attr(i) = \perp$. For any tuple t in R, the tree $K(t)$ is obtained by constructing a tree isomorphic to K, replacing each attribute with the corresponding value from t. For tuples t_1 and t_2, nodes u_1 and u_2 in $K(t_1)$ and $K(t_2)$ are *similar* if they correspond to the same node of K, and $val(u_1) = val(u_2)$. We now describe the incremental process to construct a data graph for R. We assume without loss of generality that either the root of K is unlabeled, or the roots of all the $K(t_i)$, $t_i \in R$, have the same value.

- For the first tuple, t_1, the data graph $G_1 = K(t_1)$.
- Suppose we have constructed G_{r-1}, corresponding to the first $r - 1$ tuples of R. To construct G_r, iden-



Title	Salary	Address
Programmer	100K	1200 Jose Blvd
CTO	150K	1200 Jose Blvd
CEO	$\perp$	400 7th Ave
Admin Asst	60K	400 7th Ave

Figure 1: Data graph of a portion of a corporate website advertising job openings.

tify the root of $K(t_r)$ with the root of G_{r-1}. In addition, choose some arbitrary subset of nodes from $K(t_r)$ and identify each node with a similar node from some $K(t_i)$, where $1 \leq i < r$. When both ends of a pair of edges get identified, identify the edges.

We may view a skeleton as a transformation from a relation to a data graph. Given a relation R and a skeleton K, there are several data graphs corresponding to the two, depending on which sets of nodes and edges we choose to identify. If G is any such data graph, we write $R \xrightarrow{K} G$. Note that even though a skeleton is a tree, data graphs constructed using the skeleton need not be trees; tree skeletons can give rise to DAG data graphs. We can generalize the construction process to allow for data graphs with more than one root. All our results generalize to such data graphs.

2.3 Compact Skeletons

Given a data graph G and a skeleton K, let ϕ be an isomorphism between K and some subgraph G' of G. We say that ϕ is an *overlay* from K to G if the following conditions are satisfied:

- ϕ maps the root of G to the root of K.
- Suppose u is a node of G' and v is the corresponding node in K. Then $attr(u) = attr(v)$.

If ϕ is such an overlay, we say that ϕ *includes* all the nodes and edges in G'. For node u in G', if $attr(u) = A$ and $val(u) = a$, then we say that $\phi(A) = a$. The tuple $t = \phi(X)$ corresponding to the overlay ϕ is defined in the natural manner as the list of the values $\phi(A)$ for all A in X. The relation $R(G, K)$ induced by the skeleton K is the set of all tuples t such that there is an overlay ϕ from K to G with $\phi(X) = t$. We say that a skeleton K is *perfect* for data graph G if for every arc e in E_G, there is at least one overlay that includes e.

Example 2.1. *Figure 2 shows a simple data graph G and two perfect skeletons over the attribute set ATS; let us call Figure 2(b)* K_1 *and Figure 2(c)* K_2*. In Figure 2, there are 3 possible overlays of the skeleton* K_1*, and the corresponding relation is given by*

$$R_1 = R(G, K_1) = \{a_1t_1s_1, a_1t_2s_2, a_2t_3s_3\}$$

There are 4 possible overlays of the skeleton K_2*, and the corresponding relation is*

$$R_2 = R(G, K_2) = \{a_1t_1s_3, a_1t_2s_3, a_2t_3s_1, a_2t_3s_2\}$$

Note that ϕ *is an isomorphism, so it is not possible for two nodes of the skeleton to correspond to the same node of the data graph. This observation explains why we do not find tuples such as* $a_1t_1s_1$ *in* R_2*.* □

Given a data graph, there may be several perfect skeletons corresponding to it, each inducing a different relation. In many cases, these relations do not conform to our commonsense notion of the relation corresponding to the data graph. In Example 2.1, the relation corresponding to the first skeleton seems intuitively to be "right" while that corresponding to the second skeleton does not. It appears that the second skeleton violates some notion of locality: we expect information elements that are "closer" to each other to combine to produce tuples in favor of combining elements that are "far away" from each other. More formally, it can be verified that while $R_1 \xrightarrow{K_1} G$, it is not true that $R_2 \xrightarrow{K_2} G$; in fact, it can be verified that there is no relation R satisfying $R \xrightarrow{K_2} G$.

We now formalize this notion. A *compact skeleton* K for data graph G is a skeleton that satisfies the following condition: for every node u in G, there is a node v in K such that in every overlay ϕ from K to G in which node u participates, u is mapped to v. We call v the *K-image* of u, denoted by $image_K(u) = v$. A skeleton K is a *perfect compact skeleton* (PCS) for a data graph G if K is compact for G and K is perfect for G.

Example 2.2. *The skeleton in Figure 2(b) is a compact skeleton, while that in Figure 2(c) is not. To verify the latter, consider the overlays that result in the tuples* $a_1t_1s_3$ *and* $a_2t_3s_1$*: the children of the root of G get mapped to different nodes of the skeleton by the two overlays.* □

Not every data graph has a perfect compact skeleton (PCS). However, every data graph constructed in the manner described in the foregoing section has a PCS. In addition, perfect compact skeletons possess two desirable properties: locality and uniqueness. The locality property reflects the observation that large websites are typically constructed so that local fragments make sense independent of the global

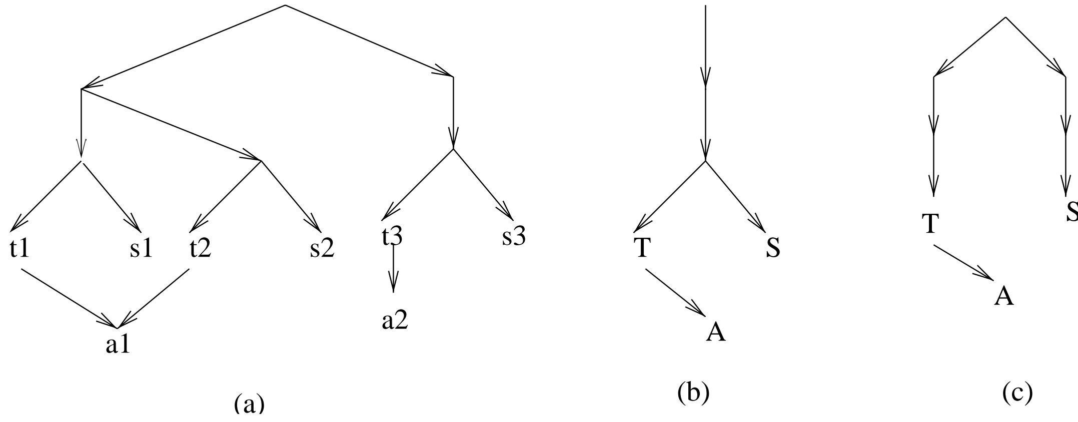


Figure 2: A data graph and two skeletons corresponding to it

picture. The following theorem summarizes our results. We use the notation that if $u \in V_G$, then G_u is the subgraph of G that is reachable from u.

THEOREM 1. *For any relation R, data graph G, and skeleton K:*

- *If $R \xrightarrow{K} G$, then K is the unique PCS for G.*
- *Conversely, if K is a PCS for G, there is a relation R such that $R \xrightarrow{K} G$.*
- *If K is a PCS for G and $u \in V_G$, then there is a subtree K' of K that is a PCS for G_u.*

2.4 Computing the PCS

Given a data graph G, there is a simple algorithm to determine if it has a PCS and to compute one if it exists. If $u \in V_G$, the *attribute set* associated with u, denoted by $attrset(u)$, is the set of all attributes whose values appear in nodes reachable from u. In what follows, for trees T_1 and T_2, $T_1 \equiv T_2$ if T_1 and T_2 are isomorphic.

Algorithm *ComputePCS*

1. For each sink u of G, T_u is the single-node skeleton labeled with $attr(u)$.
2. Process G bottom-up by successive elimination of sinks. Suppose u is the current node. Let $v_1, \ldots, v_m$ be the children of u, and let $X_i = attrset(v_i)$ and $T_i = T_{v_i}$ for $1 \leq i \leq m$. Process u as follows:
   (a) If there is a pair i, j such that $X_i \cap X_j \neq \emptyset$ and $T_i \not\equiv T_j$, then then G has no PCS.
   (b) If $attr(u) \neq \perp$ and $attr(u) \in X_i$ for some $i, 1 \leq i \leq m$, then G has no PCS.
   (c) Construct T_u as follows: the root of T_u is a node labeled with $attr(u)$; the subtrees of the root are given by $\{T_i \mid \forall j < i, T_i \not\equiv T_j\}$.
3. Let r be the root of G. Then T_r is the unique PCS for G.

EXAMPLE 2.3. *Figure 3 shows a portion on the website from Figure 2 with nodes labeled using numeric identifiers, and the trees constructed by Algorithm* ComputePCS *after processing each node of the website. The tree that results after node 1 (the root of the website) is processed is the unique PCS for the website.* □

THEOREM 2. *For any website G, Algorithm ComputePCS either computes the PCS for G or determines that G has no PCS, and runs in time $O(km|V_G|)$, where k is the number of attributes in the relation scheme and m is the number of nodes in the PCS of the largest subgraph of G that has a PCS.*

3. PARTIALLY PERFECT COMPACT SKELETONS

In the real world, it often happens that a data graph has no PCS because it has incomplete information. For example, the data graph in Figure 1 is missing a salary for the CEO, and has no PCS. Incomplete information can arise even when the relation R underlying a data graph G is complete. It may happen that our algorithms for identifying data values in the website produce false negatives e.g., we may not identify a job title because it doesn't match our patterns.

We model missing information in a data graph as follows. We start from a relation R that has nulls, constructing data graph G using skeleton K in the usual manner. Given tuple t, possibly containing nulls, the graph $K(t)$ is obtained by replacing attribute names in K with the corresponding values from t and then deleting redundant nodes, where a node is redundant if no node with a non-null value is reachable from it. With this modification, the procedure for constructing a data graph from a relation remains the same as before, and we use the same notation $R \xrightarrow{K} G$.

For data graphs with incomplete information, we relax the notion of PCS as follows. Let G be a data graph and K a skeleton over the same schema. Let ϕ be an isomorphism between a connected subgraph K' of K and a subgraph G' of G. We say that ϕ is a *partial overlay* from K to G if the following conditions are satisfied:

- ϕ maps the root of K to the root of G.
- Suppose u is a node of G and v is the corresponding node of K. If $val(u) \neq \perp$, then $attr(u) = attr(v)$. Note that it is legal for $attr(v)$ to be be non-null but

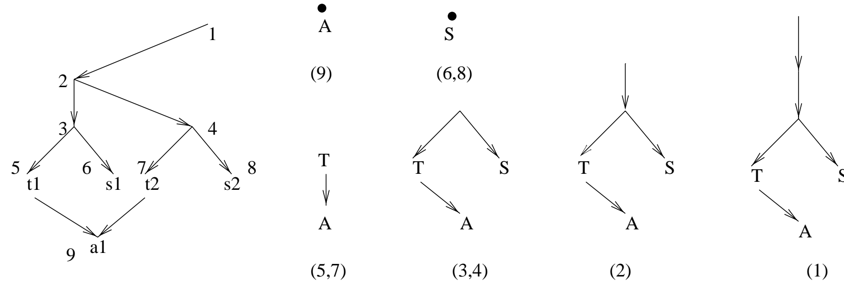


Figure 3: Running Algorithm ComputePCS

$val(u)$ to be null, which is different from the definition for overlays.

If ϕ is such a partial overlay, we say that ϕ *includes* the nodes and edges of G', and that the subgraph K' *covers* the nodes and edges of G'. The tuple t corresponding to ϕ is defined in the natural manner as the list of the information elements in G', padded with nulls in those attributes of K that don't appear in K'. Note that there are two sources of null values in t: attributes that appear in K' and are mapped to null values in G', and attributes that don't appear in K' at all. We call ϕ a *minimal partial overlay* if there is no partial overlay ϕ' that uses a strict subset of the nodes in K and derives the same tuple t.

The relation $R(G, K)$ is obtained by taking the union of the tuples produced by any partial overlay of K on G and then performing tuple subsumption, defined in the usual manner. We say that tuple t_1 *subsumes* tuple t_2 if the non-null attribute values in t_1 are the same as those in t_2; the subsumption is *strict* if t_1 has at least one additional non-null value. In $R(G, K)$ we eliminate a tuple t if there is another tuple t' that strictly subsumes t.

We define K to be *partially compact* for G if for each node u in V_G, there is a node v in V_K such that every minimal partial overlay maps u to v, and every node of K is mapped by some minimal partial overlay. The latter condition eliminates skeletons containing spurious nodes that do not appear in any minimal partial overlay, but which technically would still be compact otherwise. A skeleton K is *partially perfect* for G if for every edge e in E_G, there is a partial overlay of K that includes e. Combining these two defintions, K is a *partially perfect compact skeleton* (PPCS) for G if K is partially perfect for G and partially compact for G.

The following theorem generalizes Theorem 1 for partially perfect compact skeletons. Note that while the locality property holds for partially perfect compact skeletons, uniqueness does not (Figure 4).

THEOREM 3. *For any relation R, possibly containing null values, and data graph G:*

- *If $R \xrightarrow{K} G$, then K is a PPCS for G.*
- *Conversely, if K is a PPCS for G, then there is a relation R such that $R \xrightarrow{K} G$.*
- *If K is a PPCS for G, and $u \in V_G$ such that $image_K(u) = x$, then K_x is a PPCS for G_u.*

3.1 Minimal and Maximal Skeletons

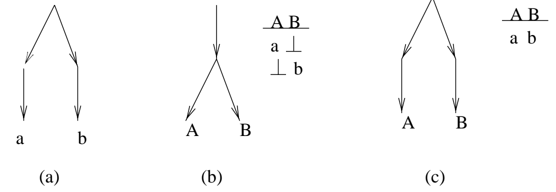


Figure 4: A data graph and two partially perfect compact skeletons

A data graph can have more than one PPCS; Figure 4 shows a data graph and two partially perfect compact skeletons corresponding to it. One PPCS induces a relation with no nulls while the other induces a relation containing null values. There are two interesting cases:

- A PPCS K is *minimal* for data graph G if for every other PPCS K' of G, $R(G, K')$ subsumes $R(G, K)$.
- A PPCS K is *maximal* for data graph G if for every other PPCS K' of G, $R(G, K)$ subsumes $R(G, K')$.

Anticipating Theorem 4, the maximal and minimal PPCS for a given data graph G, if they exist, are unique. For the data graph in Figure 4(a), the skeleton in Figure 4(b) is the unique minimal PPCS and the skeleton in Figure 4(c) is the unique maximal PPCS. The minimal PPCS is "conservative" in the sense that its relation contains just those tuples that are in the relations for every PPCS for G, while the maximal PPCS is "aggressive" in the sense that it contains every tuple in the relation for every PPCS for G. Thus the minimal and maximal PPCS give us upper and lower bounds on the relation corresponding to G.

3.2 PPCS algorithms

A DAG data graph G is *regular* if it satisfies the following conditions:

- Let p_1 and p_2 be paths from the root of G to the same node u. Then p_1 and p_2 are of the same length.
- Let $u, v \in V_G$ such that $attr(u) = attr(v)$. Then u and v are at the same distance from the root of G.
- Let u and v be nodes at the same distance from the root of G. If $attrset(u) \cap attrset(v) \neq \emptyset$, then either $attr(u) = \perp$, $attr(v) = \perp$, or $attr(u) = attr(v)$.

An interesting characterization of partially perfect compact skeletons is that a data graph G has a PPCS if and only

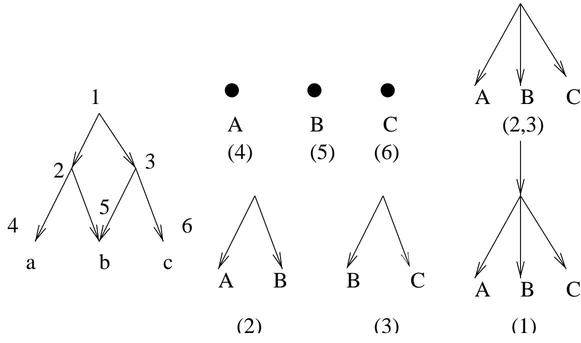


Figure 5: Algorithm ComputeMaximalPPCS

G is regular (Theorem 4). We now present an algorithm to compute the maximal partially perfect compact skeleton for a regular data graph G. The algorithm to compute the minimal partially perfect compact skeleton is similar in flavor and is in the extended version of the paper.

Algorithm *ComputeMaximalPPCS*

1. If G is not regular, then G has no PPCS.
2. Process nodes of G in decreasing order of their distances from the root. Let S_d be the set of nodes at distance d from the root.
3. Process node $u \in S_d$ as follows:
   (a) If u is a leaf, set T_u to be the tree with a single node with label $attr(u)$.
   (b) Otherwise let $T_1, \ldots, T_m$ be the set of distinct trees corresponding to the children of u. Set T_u to be the tree with root labeled $attr(u)$ and subtrees $T_1, \ldots, T_m$.
4. For each pair of nodes $u, v \in S_d$ with $attrset(u) \cap attrset(v) \neq \emptyset$, do the following:
   (a) If $attr(u) \neq \perp$, set $A = attr(u)$. Otherwise set $A = attr(v)$.
   (b) Construct T as follows: the root of T is a node labeled A; each distinct subtree of the root of either T_u or T_v is a child of the root of T.
   (c) Set $T_u := T$ and $T_v := T$.
5. Let r be the root of T. T_r is a maximal PPCS for G.

EXAMPLE 3.1. *Figure 5 shows a data graph and the partial skeletons constructed at different stages of processing. The figure labeled (2,3) is obtained after running Step 4 on the skeletons labeled (2) and (3). The final skeleton, corresponding to the root of the data graph, is a maximal PPCS for the website, and the corresponding relation is* $\{ab\perp, \perp bc\}$. □

THEOREM 4. *For any relation R, possibly containing null values, data graph G, and skeleton K:*

- *G has a PPCS if and only if G is regular.*
- *If G has a PPCS, then it has a unique maximal PPCS and a unique minimal PPCS.*
- *There is a polynomial-time algorithm that determines whether G has a PPCS, and if so, computes the minimal and maximal PPCS for G.*

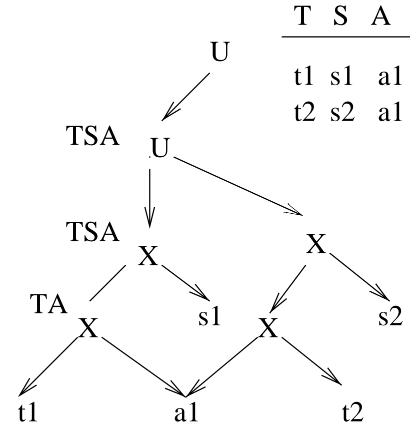


Figure 6: Running Algorithm ComputeRelation

4. ANSWERING QUERIES USING COMPACT SKELETONS

Suppose data graph G has PCS or PPCS K, we now show how to answer queries over the data graph (and the underlying website). We start with the important special case of materializing the entire relation. For simplicity we present our algorithms in the context of perfect compact skeletons; it is straightforward to extend them to partially perfect compact skeletons and also to best-fit skeletons (to be described in Section 6). Our algorithm to materialize the relation $R(G, K)$ works bottom-up on G.

Algorithm *ComputeRelation*

1. For every node u of G such that $attr(u) \neq \perp$, R'_u is the relation over the single attribute $attr(u)$ that contains the single tuple $val(u)$.
2. For every sink u of G, $R_u := R'_u$.
3. Process the nodes of G bottom up by successive elimination of sinks. Suppose we are currently processing node u.
   (a) Let $v = image_K(u)$, and let $v_1, \ldots, v_n$ be the children of v in K. Process each v_i in sequence, for $1 \leq i \leq n$:
      - Let $C_i = \{x \mid (u, x) \in E_G \text{ and } image_K(x) = v_i\}$
      - Set $S_i := \bigcup_{x \in C_i} R_x$.
   (b) Set $R_u := S_1 \times \ldots \times S_n$.
   (c) If $attr(u) \neq \perp$, set $R_u := R'_u \times R_u$.
4. If s is the root of G, then R_s contains the desired result.

A useful way to visualize Algorithm *ComputeRelation* is as a transformation that converts a data graph G into a relational operator DAG. Figure 6 shows the operator DAG and result relation obtained by running Algorithm ComputeRelation on a portion of the data graph in Figure 2(a) using the compact skeleton shown in Figure 2(b).

THEOREM 5. *Algorithm ComputeRelation computes the relation R corresponding to a data graph G and runs in time $O(k|V_G||R| \log |R|)$ where k is the number of attributes in the schema.*

In an earlier section, we had asked whether we can precisely characterize those data graphs G for which there is a unique relation R such that $R \xrightarrow{K} G$. We now provide such a characterization.

COROLLARY 1. *Given data graph G with PCS K, there is a unique relation R such that $R \xrightarrow{K} G$ if and only if whenever Algorithm ComputeRelation is required to compute a cross product, at most one relation in the cross product is non-singleton.*

4.1 Answering Queries

Often in a mediation scenario, the query to a wrapper does not materialize the entire relation but some subset of it. It would be helpful to have output-size sensitive algorithms to answer arbitrary queries on a data graph. Suppose G is a data graph with PCS K, corresponding to relation R over attribute set X. We now consider query plans for select-project queries on R, that is, queries of the form $\pi_Y(\sigma_{A=a}R)$, where $Y \subseteq X$ and A is in X. The most expensive operation in such cases is fetching web pages, so we look for query plans that fetch as few pages as possible.

A select-project query on R can be seen as a transformation on the PCS K. Consider the project query $\pi_Y(R)$. Let $K[Y]$ be the unique connected subgraph of K that includes the root of K and a path from the root to every node labeled with some A in Y. Let Y' be the attribute set of $K[Y]$; clearly $Y \subseteq Y'$ and $K[Y] = K[Y']$. It can be verified that $\pi_{Y'}(R)$ is the relation corresponding to subgraph of G that has $K[Y']$ as its PCS. To construct this relation, we can avoid fetching pages that do not contain an information element from Y'. We then project out attributes in $Y' - Y$ to obtain $\pi_Y(R)$.

The query $\sigma_{A=a}(R)$ can be seen as adding the constraint $A = a$ to the node in K labeled by attribute A; call the resulting constrained skeleton K'. We extend the definition of overlays to skeletons with constraints in the natural manner. Now the answer to the query $\sigma_{A=a}(R)$ is the relation corresponding to the constrained skeleton K'. We may modify Algorithm ComputeRelation to first fetch pages containing values of attribute A and avoid fetching pages corresponding to cross product operands when we can determine that one of the operands of the cross product is the empty relation. An equivalent view is that we push the selection condition $A = a$ down the operator DAG induced on G by K.

To answer a general select-project query, we combine the techniques for selections and projections.

5. NOISY DATA GRAPHS

In addition to incomplete information, real websites contain *noise*, i.e., superfluous information. Such a noisy website may not have a PPCS. Noise in websites can be purely random, but can also result from false-positive matches from the patterns used to identify data elements. For example, a pattern to identify US states might match the "MS" in the text "MS Word," and a pattern to identify salaries of the form "70K" might match the text "401K." In addition, there are sometimes links in websites that do not correspond to skeleton links. For example, consider a website for an online retailer, with products organized by category, then by subcategory. There might be non-skeleton links directly from the home page to some of the product pages because these are "featured products" on a given day.

When confronted with a noisy website, we look for a skeleton that covers as much of the data graph as possible. In doing so we must relax both the compactness condition and the perfect coverage condition for skeletons. We relax the compactness condition as follows. Given a data graph G and a skeleton K, a set S of overlays is *consistent* if for every node u in G that is included in some overlay in S, there is a unique node v in K such that every overlay $\phi \in S$ that includes u maps u to v. The *cover* of set S is the set of nodes and edges of G that are included in some overlay $\phi \in S$, and the *coverage* of S is some metric that measures the goodness of the cover. The coverage of skeleton K, denoted $coverage(G, K)$, is the maximum coverage across all consistent sets of overlays. A skeleton K is a *best-fit skeleton* for a data graph G if for any other skeleton K', $coverage(G, K) \geq coverage(G, K')$.

Possible coverage metrics include the cardinality (i.e., the number of nodes and edges) of the cover, the number of nodes in the cover, and the number of non-null data values in the cover. In the rest of this paper, we define the coverage as the number of nodes in the cover; our results, however, extend to the other metrics as well. Unfortunately, as we show in [28], the problem of finding a best-fit skeleton for a given data graph is NP-complete, even when the data graph is restricted to be a tree of depth no greater than 3. Given this negative result, there are two approaches to constructing a "good" skeleton for a a data graph:

1. *Semi-automatic.* Given a data graph, a human uses a visualization technique to "eyeball" the data graph and guesses a skeleton. The system computes the coverage of the skeleton. If the coverage is large enough as a ratio of the data graph size, the skeleton is accepted as a good skeleton.

2. *Automatic.* Given a data graph, use heuristics to compute a good skeleton that is as close to the best-fit skeleton as possible.

In this section, we explore computing the coverage of a skeleton to aid the semi-automatic approach. Section 6 consider heuristics to compute best-fit skeletons.

5.1 Computing the coverage of a skeleton

Given a data graph G, the best-fit skeleton is not necessarily unique; this observation follows from our experience with partially perfect compact skeletons, which are a special case of best-fit skeletons with perfect coverage. It turns out that we can restrict our attention to a class of skeletons that we call *canonical skeletons*. A skeleton K is canonical if each labeled node in K has at most one unlabeled child. Canonical skeletons are a natural generalization of minimal partially perfect compact skeletons, and are important because of the following result.

LEMMA 1. *Given a data graph G and skeleton K, there is a canonical skeleton K' with $coverage(G, K') \geq coverage(G, K)$.*

Intuitively, given a non-canonical skeleton we can identify unlabeled siblings to construct a canonical skeleton with the same or better coverage. It can be shown that every partial overlay of the original skeleton is also a partial overlay of the modified skeleton. The constructive proof appears in the extended version of the paper [28].

It turns out that the problem of computing the coverage of a skeleton for a data graph is NP-complete in the general case [28]. However, we can impose some practical restrictions on the structure of the data graph and the skeleton to yield tractable cases. We consider two important tractable cases:

1. Data graphs that are trees. In this case, there is a simple linear algorithm to compute the coverage.

2. A class of DAG data graphs called *unambiguous data graphs* that we define in Section 5.2. In this case, there is a polynomial-time algorithm to compute the coverage of a canonical skeleton, while the computing the coverage of an arbitrary skeleton is still NP-complete. This case is interesting because in practice many data graphs corresponding to real websites are unambiguous.

5.2 Unambiguous data graphs

Given a DAG data graph G, we define a process called *labeling* that will be used in subsequent algorithms. We assume without loss of generality that the root of G has a data value r corresponding to attribute R. The *label* function recursively associates a set of labels with each node u in G as follows:

- If u is the root of G, $label(u) = \{R_0\}$.
- If $attr(u) = A \neq \perp$, $label(u) = \{A_0\}$.
- If $attr(u) = \perp$, let P be the set of parents of u in G. Then for each node $v \in P$, if $A_i \in label(v)$, then A_{i+1} is in $label(u)$.

We denote by L the set of labels that were created by the labeling process. A data graph G is *unambiguous* if for each node u in G, $label(u)$ is a singleton set. As a special case, a data graph G is unambiguous if all its nodes have non-null values. Note that a tree data graph can be ambiguous, while a DAG data graph can be unambiguous; the sets of tree data graphs and unambiguous data graphs are incomparable.

5.3 Complexity

The table in Figure 7 summarizes the complexity of the coverage problem. Interestingly, relaxing either the ambiguity condition on data graphs or the canonical condition on skeletons makes the problem NP-complete [28]. The full version of the paper describes the algorithms for the polynomial-time cases.

6. COMPUTING BEST-FIT SKELETONS

We turn now to the following problem: given a data graph G, find a best-fit skeleton for G. As we mentioned earlier, this problem is NP-complete. It is natural to ask whether we can restrict the structure of G as in the case of the coverage problem to come up with tractable cases. Unfortunately, such is not the case. The best-fit problem is NP-complete even when the data graph G is restricted to be a tree of depth 3 with no unlabeled nodes [28].

We look therefore for polynomial-time heuristics that can approximate a best-fit skeleton. For simplicity we restrict our attention to unambiguous data graphs. We also make a further simplifying assumption, as follows. Let G be a data graph. The *ancestor* relation among the attributes in the schema is defined in the following manner: attribute A is an ancestor of attribute B if there are nodes $u, v \in V_G$ with $attr(u) = A$, $attr(v) = B$, and u is an ancestor of v in G. The ancestor relation on G is the transitive closure of such pairwise ancestor relationships. If the ancestor relation is acyclic (i.e., it is a partial order), we call G *label-acyclic*. In what follows we restrict our attention to unambiguous, label-acyclic data graphs. In practice, this assumption is reasonable because many websites of interest have data graphs that fit this model. Our heuristics can be modified for more general data graphs. The best-fit skeleton problem remains NP-complete even when restricted to unambiguous, label-acyclic data graphs [28].

Given a data graph G, our algorithms to construct best-fit skeletons work in three steps:

1. Transform G into a *fully-labeled data graph* G' (as defined in Section 6.1).

2. Construct a "good" skeleton K' for G'.

3. Transform K' into a "good" skeleton K for G.

We first describe the label transformation process involved in Steps (1) and (3). We then propose two different heuristics for Step (2) and study how well they perform in practice.

6.1 Labeled data graphs

Let G be an unambiguous data graph over schema X. Label each node in G using the *label* function defined in Section 5.2. Let L be the set of labels. It is useful to think of L as the set of attributes in a new schema. We construct a new data graph $G' = label(G)$ as follows: G' is a copy of G, but its attributes are the set of labels in L. For each node u in G with $label(u) = A_i$, for some $A \in X$ and $i \geq 0$, the corresponding node u' in G' has $attr(u') = A$.

We also define a reverse, unlabeling transformation on skeletons. Let K' be a skeleton on schema L. Then $K = unlabel(K')$ is the skeleton over schema X whose node and edge set are copies of K', and whose nodes are labeled as follows:

- For each node u' in K' with $attr(u) = A_0 \in L$, the corresponding node u in K has $attr(u) = A$.
- For each node u' in K' with $attr(u) = A_i \in L$, $i > 0$, the corresponding node u in K is unlabeled.

Lemma 1 allows us to restrict our attention to canonical skeletons; the following lemma takes us one step further and allows us to work with fully-labeled data graphs.

LEMMA 2. *Let G be an unambiguous data graph, and let $G' = label(G)$. If K' is a canonical skeleton for G' and $K = unlabel(K')$, then $coverage(G, K) = coverage(G', K')$.*

6.2 The greedy heuristic

Let G be an unambiguous, label-acyclic, fully-labeled data graph over schema (label set) L. We now present a simple polynomial-time heuristic that computes a canonical skeleton K for G. Since K is a tree all of whose nodes are labeled, it can be specified completely by the *parent* function on labels, as follows: suppose u is a node in K with $attr(u) = A_i \in L$, and v is the parent of u in K with $attr(v) = B_j \in L$, then $parent(A_i) = B_j$. As a special case, if R is the label of the root of K, then $parent(R) = \perp$.

	Tree data graphs	Unambiguous DAGs	General DAGs
Canonical skeletons	P	P	NP-complete
Arbitrary skeletons	P	NP-complete	NP-complete

Figure 7: Complexity of the coverage problem

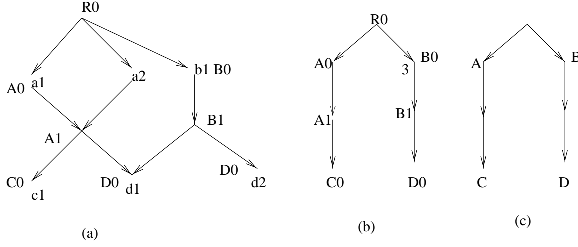


Figure 8: Running Algorithm GreedySkeleton

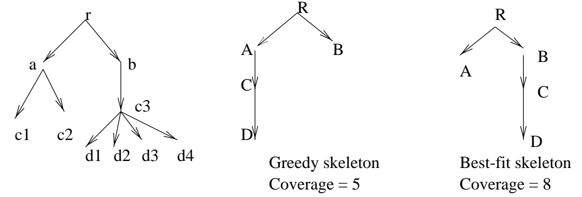


Figure 9: A data graph where the greedy algorithm does badly

Algorithm *GreedySkeleton*

1. For all pairs of labels $X, Y \in L$, set $parentCount(X, Y) = 0$
2. Traverse G and and process each arc (u, v) as follows: if $attr(u) = X$ and $attr(v) = Y$, set
$$parentCount(X, Y) := parentCount(X, Y) + 1$$
3. Process each label in $X \in L$ as follows:
   (a) Let $Y \in L$ be the label with the largest value of $parentCount(Y, X)$
   (b) Set $parent(X) = Y$

Since G is label-acyclic, the *parent* function computed by Algorithm GreedySkeleton is guaranteed to be acyclic. The following example illustrates the algorithm.

EXAMPLE 6.1. *Figure 8(a) shows a fully-labeled data graph G. The nonzero parent counts are as follows:*

- $parentCount(A_1, C_0) = 1$, *and* $parent(C_0) = A_1$.
- $parentCount(A_1, D_0) = 1$, $parentCount(B_1, D_0) = 2$, *so* $parent(D_0) = B_1$.
- $parentCount(A_0, A_1) = 2$, $parent(A_1) = A_0$.
- $parentCount(B_0, B_1) = 1$, $parent(B_1) = B_0$.
- $parentCount(R_0, A_0) = 2$, $parent(A_0) = R_0$.
- $parentCount(R_0, B_0) = 1$, $parent(B_0) = R_0$.

The resulting skeleton K' is shown in Figure 8(b). Unlabeling K' yields the skeleton K in Figure 8(c). □

6.3 The weighted greedy heuristic

Figure 9 shows an example of a data graph in which the greedy algorithm does not pick the optimal skeleton. Intuitively, the reason why greedy does badly on the data graph in Figure 9 is that all its decisions are made independently, so that the effects of a prior decision are not factored into subsequent decisions. We can tweak the greedy heuristic to avoid some of these situations; at each step, we compute the *benefit* of a decision taking into account all prior decisions. The benefit is measured by data graph coverage, the metric we seek to maximize. At each stage, we greedily pick the decision with largest benefit. We present below this *weighted greedy* heuristic.

Algorithm *WeightedGreedy*

1. Process labels in L "bottom-up," so that label X is processed after all labels $Y \in L$ with $X \in ancestor(Y)$.
2. Suppose we are currently processing label X.
   (a) For each label $Y \in ancestor(X)$:
      i. Set H to be the empty graph
      ii. Traverse G and visit each arc (u, v) such that $label(v) = X$ and $label(u) = Y$.
      iii. Let G' be the subgraph of G reachable from node v.
      iv. Set $H = H \cup G'$.
      v. Set $benefit(X, Y) = |V_H|$, where $|V_H|$ is number of nodes in H.
   (b) Let $Z \in ancestor(X)$ be the label with largest value of $benefit(X, Z)$
   (c) Set $parent(X) = Z$
   (d) Traverse G and delete all edges of the form (u, v) where $label(v) = X$ and $label(u) \neq Z$. Recursively delete all nodes and edges that are disconnected from the root of G by this deletion.

EXAMPLE 6.2. *For the data graph G in Figure 9, we process the labels in the order DCBAR. We compute the following non-zero benefits:*

- $benefit(D, C) = 4$, *and* $parent(D) = C$.
- $benefit(C, A) = 2$, $benefit(C, B) = 5$, *so* $parent(C) = B$.
- $benefit(B, R) = 6$, *and* $parent(B) = R$.
- $benefit(A, R) = 1$, *and* $parent(A) = R$

The resulting skeleton is the best-fit skeleton with coverage 8. □

6.4 Theoretical performance of heuristics

How badly can our heuristics perform, as a ratio of the optimal skeleton's coverage (called the *competitive ratio*)? We can modify the data graph in Figure 9 to create data graphs where the greedy algorithm's competitive ratio is arbitrarily close to zero; that is, the greedy heuristic can perform arbitrarily badly compared to the optimal result. The weighted greedy heuristic works well for the data graph in Figure 9, but we can construct data graphs where it does not pick the optimal skeleton. It turns out that the weighted greedy can do no worse than half-optimal for data graphs of depth 2. More generally, weighted greedy can be no worse than half-optimal at each level of the data graph, so that it can be no worse than $(\frac{1}{2})^{d-1}$ of the optimal skeleton for a data graph of depth d. In practice, however, we observe that the weighted greedy does much better than the theoretical worst-case bound, as shown by the experimental results of the following section.

6.5 Experimental results

For our experiments, we chose as our application a system that integrates job listings from multiple corporate websites into a single database (WhizBang! Labs' FlipDog.com [33] is such a system). We chose a relation schema with a dozen attributes such as Job Id, Job Title, Job Category, Division, Location, and Job Description. Not all websites include information on all these attributes. We developed regular expression patterns that could identify these patterns on a small set of websites, including those of IBM [20] and Sun Microsystems [31] (a real system such as [18] or [34] incorporates more extensive heuristics to identify attributes).

The data graphs we constructed had between 5000 and 10,000 nodes. The Sun website uses a form that is filled in at the top level; we constructed the data graph for this site using the technique for forms outlined in the extended version of the paper. The skeletons were constructed using the greedy and weighted greedy heuristics, which both constructed the same skeleton. We also verified manually that these skeletons are the optimal (best-fit) skeletons for these data graphs.

To systematically study how the greedy and weighted greedy heuristics performed as the noise-level in the data graph is increased, we came up with a technique to randomly mutate the data graphs. We picked a single parameter, the *error rate* p, to model both the *precision* and the *recall* of the pattern matching functions: p is both the probability of a false positive and the probability of a false negative. We randomly mutated our data graphs assuming different values of p ranging from 0 (perfect precision and recall) to 1.0 (random data graphs). In each case, we ran the greedy and weighted greedy heuristics on the mutated data graph, and we also ran an exhaustive enumeration algorithm that computes a best-fit skeleton.

Figures 10 and 11 summarize the results of our experiments. We note that both heuristics compute the best-fit skeleton, for error rates below 0.2. For error rates larger than 0.2, the greedy heuristic decays linearly (we show a least-squares fit). Weighted greedy is always competitive to with a factor of 0.95 of the optimal, and thus is a very good heuristic in practice. There is a big payoff for the additional complexity of using weighted greedy versus the greedy heuristic, while there is very little payoff in using the much more expensive exhaustive search algorithm to compute the optimal skeleton.

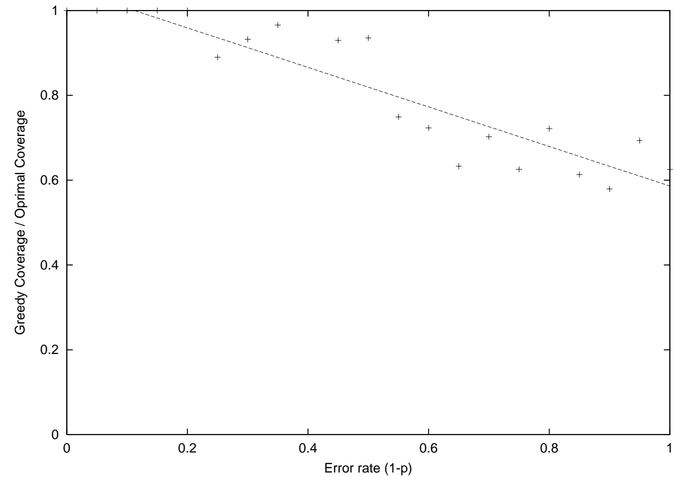


Figure 10: Performance of greedy heuristic

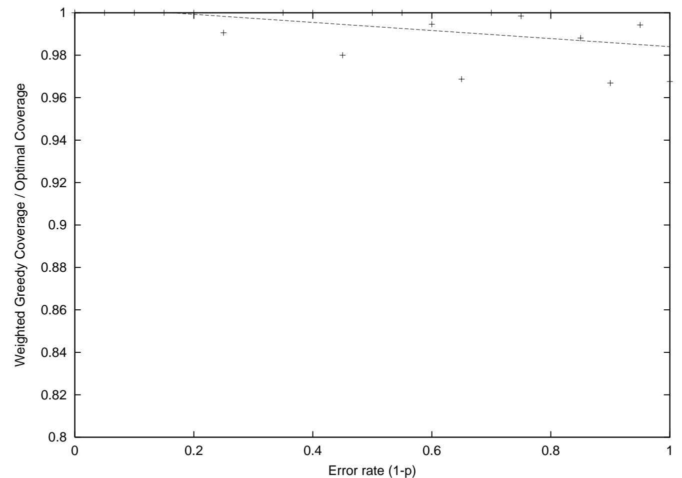


Figure 11: Performance of weighted greedy heuristic

7. GRAPH SKELETONS

Thus far we have restricted our attention to skeletons that are trees. There are websites that can be generated using more general skeletons, such as DAGs or even cyclic graphs. It turns out that theory of perfect compact skeletons generalizes to such skeletons: in particular, Theorem 1 generalizes so that every data graph has either a unique PCS or no PCS, even when the PCS is a cyclic graph. In the full version of the paper, we provide an algorithm to construct a PCS for a data graph G if it has one, and a proof that the PCS is unique. The algorithm is exponential in the size of G and it is unlikely that we can do better: determining whether a data graph has a DAG PCS turns out to be an NP-complete problem.

We define a skeleton K to be *irreducible* if there is no skeleton K' with fewer nodes and edges than K that is a PCS for K when K is treated as a data graph. The following

theorem is the counterpart of Theorem 1.

THEOREM 6. *For any relation R, data graph G, and irreducible skeleton K:*

- *If $R \xrightarrow{K} G$, then K is the unique PCS for G.*
- *Conversely, for any data graph G with PCS K, there is a relation R such that $R \xrightarrow{K} G$.*
- *There is a nondeterministic polynomial-time algorithm to determine whether G has a generalized PCS.*

The following theorem shows that we cannot hope to find a polynomial-time algorithm for the PCS problem, even for DAG-structured data graphs with an information element at each node (i.e., all nodes have non-null attribute values). To show that the problem is NP-hard, we provide a reduction from the problem of determining whether a natural join of several relations is non-empty, which is known to be NP-complete [32].

THEOREM 7. *The problem of determining whether a DAG-structured data graph, all of whose nodes have non-null values, has a DAG PCS is NP-complete.*

8. RELATED WORK

Integrating data across websites is an active area of research, with several research prototypes and commercial implementations [14, 22, 18, 34, 12]. Constructing wrappers has been the major bottleneck in most of these systems. Hammer et al. [19], one of the earliest systems, describes a toolkit that helps the user manually code wrappers. The toolkit provides many constructs, especially for HTML processing, that make it easier in many cases than writing parsers using Lex and Yacc.

There has been much work in automating aspects of wrapper construction [2, 3, 4, 5, 21, 13, 18, 24, 30]. Most of this work has focused on extracting document structure (as a grammar or a finite state automaton) from HTML and text documents. Adelberg [2] and Garofalakis et al. [15] describe systems that can infer structure from HTML and XML documents, respectively, in a semi-automated manner. Ashish and Knoblock [3, 4] present a technique called *wrapper induction* that can infer simple grammars that combine HTML elements and data elements on web pages. Kushmerick et al. [21], Muslea et al. [24] and Soderland [30] describe machine learning approaches. In contrast to these works, which infer structure within web pages, compact skeletons describe web site structure within and across web pages, and in addition also enable automatic transformation between relational and web data.

Gupta et al. [17] describes a system in which wrappers are expressed in a specialized language called *Site Description Language* (SDL). It can be shown that SDL corresponds to a a restricted form of canonical skeletons we call *skinny skeletons*: skeletons that are trees consisting of a single root-to-leaf *distinguished path* such that every node is at distance at most 1 from this path. The VDBMS system described in [18] has a wrapper toolkit that facilitates semi-automated wrapper construction, using algorithms similar to ours (but simpler because skeletons are restricted to be skinny).

There has been much work on querying documents, semistructured data, and unstructured data [10, 8, 9, 23, 1, 27, 6, 11, 29]. Many of these systems model semistructured data using data graphs. Some consider schemas that are similar in spirit to compact skeletons. The focus of these works is on query languages, testing whether a database conforms to a given schema, schema subsumption, query optimization, and data translation, in contrast to our focus on schema inference.

Nestorov et al. [25] presents a different approach to extract structure from semistructured data. They model semistructured data using data graphs and construct a *typing* that fits the data within an error threshold. Typings are defined in terms on monadic datalog programs, and nodes in the data graph can play multiple "roles." Compact skeletons by contrast provide a simple graph-theoretic approach to schema inference that works well for data on the web, and lend themselves to simple algorithms that perform well on large websites.

Representative Objects [26] and Data Guides [16] provide structural summaries of hierarchical semistructured data in a manner similar to skeletons. The purpose of data guides is to facilitate browsing and query optimization rather than mapping data into the relational model. Therefore the data guide model allows more general structures than skeletons: data guides are not restricted to be trees and the same label can appear more than once in a data guide. The different intended applications lead to problems and algorithms of a different flavor from those presented here.

9. CONCLUSION

Compact skeletons are a simple and effective model for "reverse-engineering" websites to construct wrappers that expose relational interfaces. The model can equally well be applied to XML documents where the DTD is not specified in advance. We are exploring several extensions to the basic model:

- Feedback between feature (i.e., data element) extraction and structure extraction.
- Using domain knowledge about the relation e.g., functional dependencies, to refine our heuristics for skeleton construction.
- Data graphs corresponding to multiple compact skeletons. In such cases, we must construct a "small" set of skeletons that cover as much of the data graph as possible, trading off between coverage and the number of skeletons.

10. REFERENCES

[1] S. Abiteboul. Querying semistructured data. In *Proceedings on the International Conference on Database Theory*, 1997.

[2] B. Adelberg. Nodose – a tool for semi-automatically extracting structured and semistructured data from text documents. In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, 1998.

[3] N. Ashish and C. Knoblock. Semi-automatic wrapper generation for internet information sources. In *Proceedings of CoopIS '97*, 1997.

[4] N. Ashish and C. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, 1997.

[5] P. Atzeni and G. Mecca. Cut and paste. In *Proceedings of the Sixteenth ACM Symposium on Principles of Database Systems*, pages 144–153, 1997.

[6] C. Beeri and T. Milo. Schemas for integration and translation of structured and semistructured data. In *Proceedings of the International Conference on Database Theory*, 1999.

[7] S. Brin. Extracting patterns and relations from the world-wide web. In *International WebDB Workshop, Valencia, Spain*, pages 172–183, 1998.

[8] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings on the International Conference on Database Theory*, 1997.

[9] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, 1996.

[10] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, 1994.

[11] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, 1998.

[12] R. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the First International Conference on Autonomous Agents*, 1997.

[13] D. Embley, D. Campbell, Y. Jiang, Y.-K. Ng, R. Smith, S. Liddle, and D. Quass. A conceptual-modeling approach to extracting data from the web. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER '98)*, 1998.

[14] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[15] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, 2000.

[16] R. Goldman and J. Widom. Data guides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, 1997.

[17] A. Gupta, V. Harinarayan, D. Quass, and A. Rajaraman. Method and apparatus for structuring the querying and interpretation of semistructured information. United States Patent number 5,826,258, 1998.

[18] A. Gupta, V. Harinarayan, and A. Rajaraman. Virtual database technology. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 297–301. IEEE Computer Society, 1998.

[19] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *Workshop on management of semistructured data*, 1997.

[20] IBM Corp. Job listings at IBM corporate website. `http://www.ibm.com/employment/us/html/location.html`.

[21] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, 1997.

[22] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 251–262, 1996.

[23] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proceedings of the 24th International Conference on Very Large Data Bases*, 1998.

[24] I. Muslea, S. Minton, and C. Knoblock. Stalker: Learning extraction rules for semistructured, web-based information sources. In *Proceedings of AAAI '98: Workshop on AI and Information Integration*, 1998.

[25] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD International Conference of Mangement of Data*, 1998.

[26] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concide representations of semistructured hierarchical data. In *Proceedings of the Thirteenth International Conference on Data Engineering*, 1997.

[27] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured, heterogeneous information. In *Proceedings of the Fourth International Conference on Deductive and Object Oriented Databases*, 1995.

[28] A. Rajaraman and J. Ullman. Querying websites using compact skeletons. `http://www-db.stanford.edu/~anand/pub/skeleton.ps`, 2001.

[29] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[30] S. Soderland. Learning to extract text-based information from the world-wide web. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, 1997.

[31] Sun Microsystems. Job listings at Sun Microsystems website. `http://www.sun.com/jobs`.

[32] J. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies.* Computer Science Press, Rockville, MD, 1989.

[33] WhizBang! Labs. Flipdog.com job search website. `http://www.flipdog.com/home.html`.

[34] WhizBang! Labs. WhizBang! Labs corporate website. `http://www.whizbanglabs.com`.