Results of Using an Efficient Algorithm to Query Disjunctive Genealogical Data

Lars E. Olson
David W. Embley
Department of Computer Science
Brigham Young University
Provo, UT 84602
{olsonle,embley}@cs.byu.edu

1. Introduction

In doing genealogical research, it is common to find multiple sources of information that provide contradictory or uncertain data. For example, suppose that for a particular person we find two possible dates of birth (such as "4 or 5 December 1394") or an imprecise date of birth (such as "December 1394," with a month and year but no day). There may be no way of ever finding out the true date of birth, although we may be more confident in one date than another. Since we typically want to have as much identifying and certainty information as possible, the database should store as much as it can, even though doing so may not fit the data model.

Most simple databases require the conflicts and uncertainty to be resolved, and are therefore not well suited for genealogy data. Disjunctive databases (databases that allow the "OR" of a set of values in place a single value) are more well-suited for this type of problem, but it has been proven in [IV89] that queries on disjunctive databases in general have CoNP-complete time complexity (i.e. all known algorithms require an exponential amount of time based on the problem size). Thus, using an unrestricted disjunctive database makes querying the data intractable.

Based on an algorithm presented in [LYY95], however, we present a way to handle genealogical data so that many common queries become tractable. Furthermore, the theorem also gives us a way to determine which queries remain intractable, which gives us the opportunity to handle these queries heuristically (i.e. using a quick estimate) or under reasonably bounded extents. An in-depth project is under way [O03] to implement a data storage and query engine using this algorithm, and this paper presents some preliminary results.

This material is based upon work supported by the National Science Foundation under Grant #0083127

2. Disjunctive Graphs

The solution to this problem is based on the notion of disjunctive graphs; that is, graphs with hyperarcs that represent disjunctions. Figure 1 shows an example of a disjunctive graph, along with one of its possible interpretations (where each hyperarc is replaced by one of the possible arcs it can represent). Disjunctions can occur on the head side of the arc (such as the arc from *a* to {*b*,*c*}) or on the tail side (such as the arc from {*c*,*d*} to *f*) or on both sides. Since only one of the disjunctive heads or tails can hold, each disjunction gives rise to a different interpretation. There are multiplicatively many interpretations, which is why disjunctive data naturally leads to intractability.

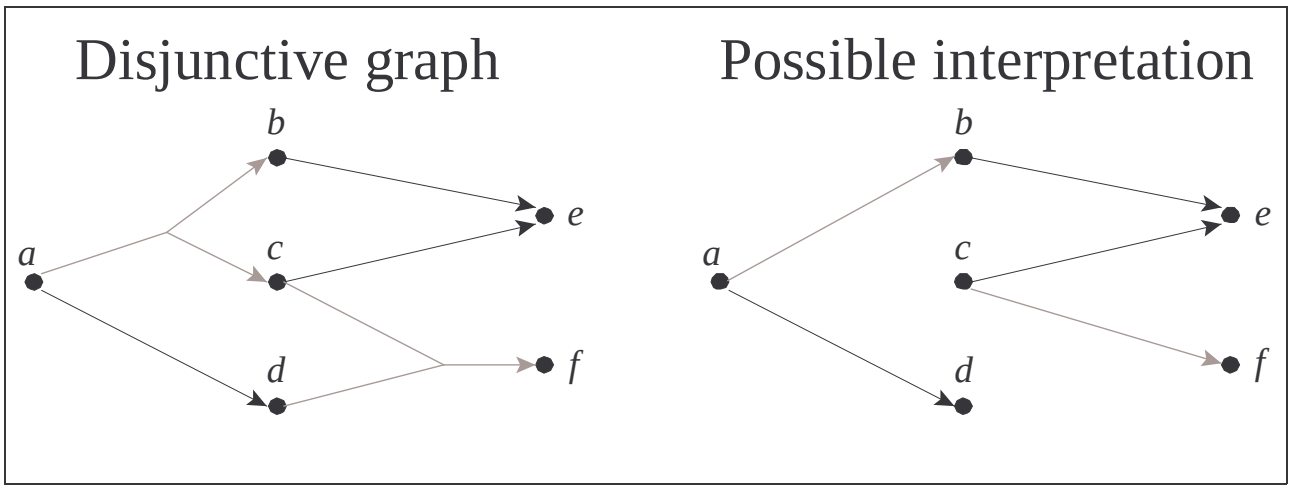

Figure 1: An example of a disjunctive graph (left) containing two disjunctive arcs, and one of the four possible interpretations of the graph (right).

[LYY95] considers the problem of computing the transitive closure of a node *x* in a disjunctive graph, which is defined as the set of all nodes *y* such that in every possible interpretation of the disjunctive graph, there exists a path from *x* to *y*. For the graph in Figure 1, for example, the transitive closure of node *a* is the set {*a*, *d*, *e*}. The theorem states that if each disjunctive arc of the graph contains a disjunction only in the head of the arc (rather than in the tail), computing the transitive closure is a polynomial-time algorithm; otherwise, it is CoNP-complete.

Table *Person*:

ID#	Name	Birth Date	Birth Place ID#	Marriage Date
1	John Doe	12 Mar. 1840 or 12 Mar. 1841	1 or 2	15 Jun. 1869 or 16 Jun. 1869
⋮	⋮	⋮	⋮	⋮

Table *Place*:

ID#	City	State
1	Commerce or Nauvoo	Illinois
2	Quincy	Illinois
⋮	⋮	⋮

Figure 2: Genealogy database to be converted to a disjunctive graph. Note that attribute Birth Place ID# is a foreign key referencing table *Place*.

In order to use this algorithm to perform queries on a disjunctive database, we must first convert the database into a disjunctive graph in such a way that solving the transitive closure of a node (or set of nodes) is equivalent to solving the query. Consider the database in Figure 2. The values in this database correspond to nodes in the graph we will build. The arcs in the graph are made by connecting each of the key values to their corresponding attributes, using disjunctive arcs where necessary. We also add arcs representing foreign keys by drawing arcs to the actual nodes in the table being referenced. We can attach labels to each arc to represent the attributes. Figure 3 shows the transformed graph.

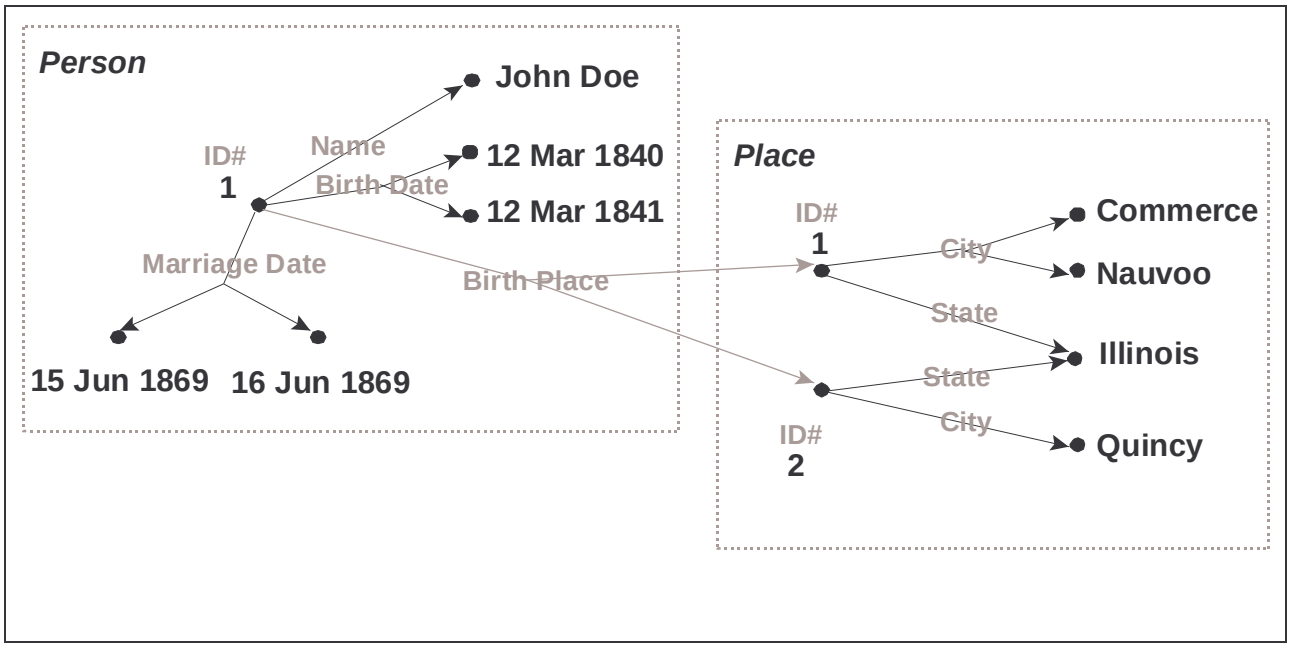

Figure 3: Database from Figure 2 transformed into a disjunctive graph.

3. Query Processing with Disjunctive Graphs

Consider the query, "In what state was the person with ID#1 born?" (which can be written as $\pi_{State}(\sigma_{ID=1}\text{Person} \bowtie \text{Place})$ in relational algebra notation.) To answer this query, we compute the transitive closure of the node labeled ID#1 and return the node corresponding to the attribute "State," which in this case is Illinois. We can guarantee that no arcs with disjunctive tails will appear for a database such as this if we insist that (1) no disjunctions for primary-key attributes occur (e.g. the relation will never contain "ID#1 or ID#2" for a single object) and (2) that foreign keys only reference primary-key attributes in other tables. Since the only arcs created in the transformation originate from these object identifiers, we will never have a disjunctive tail, and therefore this type of query can be answered in polynomial time.

Not all queries can be answered this easily, however. If we consider the query, "In what city and state was the person with ID#1 born?" (which can be written as $\pi_{City,State}(\sigma_{ID=1}\text{Person} \bowtie \text{Place})$), we still return Illinois as the state, but we find no city in the closure (as [LYY95] defines it). The correct response to this query depends on what the user really means by the query. If the user wants the values that we know without a doubt, then since we do have doubt about the correct city, the transitive closure does indeed give the correct solution. [OE02] discusses other possible "correct" answers for this type of query.

When a query requires selection on non-key attributes, such as, "Find the names of all people born in Nauvoo between 1841 and 1844" (which can be written as $\pi_{\text{Name}}\ \sigma_{\text{BirthDate} \geq 1841 \wedge \text{BirthDate} \leq 1844 \wedge \text{City} = \text{Nauvoo}}$ (Person $\bowtie$ Place)), rather than on a key attribute, such as the queries already discussed, we can still guarantee polynomial running time. We compute the closure for every possible ID# (which is bounded by the number of nodes *n* in the graph, and thus the running time is bounded by $n * P(n)$ where $P(n)$ represents the time required to compute the closure for one node). For each ID#, if the Date falls within the specified range and the City attribute is "Nauvoo," we add the Name attribute to the answer.

Most genealogical queries can be handled in this manner to achieve polynomial running time, but there are some exceptions, such as queries that require joins on attributes other than the primary key attributes. We can offer partial answers by simply removing all the disjunctive arcs entirely, or ask the user to limit the search space. If a partial answer is not acceptable, we can at least detect when such a query is CoNP-complete, warn the user if the size of the graph is large, and ask how we should proceed.*

4. Implementation and Results

The project in progress described in [O03] uses an XML-based database system. XML is not necessarily required for representing disjunctive data; however, it has the advantage of being compatible with newer standards such as [GED02]. These XML data formats must be slightly modified to include some way to label sets of data values as disjunctions in order to be usable with our query algorithm. For this purpose we introduce a <Disj> tag, as Figure 4 shows.

Currently, we use as a database backend the University of Wisconsin's Shore database [SHORE] with its Niagara XML interface [NIAGARA], which uses Apache.org's Xerces C++ XML parser [XERCES]. The underlying theories, however, should be applicable to any data storage system.

Figure 5 shows a graph of some timing comparisons using a brute-force algorithm (checking every possible valid model of the database and returning the answers that appear in each one), a backtracking algorithm (attempting to calculate the closure normally and only expanding possible models when a disjunctive edge is encountered), and the polynomial algorithm. It also includes a heuristic to ignore all disjunctive edges, as a comparison with the time required for a normal database. This heuristic only provides a partial answer.

---

*[OE02] gives more query examples showing more different kinds of disjunctions demonstrating this. It also addresses the possibility of adding weighted edges representing the confidence we have in each possible value. In this paper, however, we wish to focus on the implementation and on measuring running times.

```
<?xml version= “1.0”?>
<GEDCOM>
...
      <IndividualRec Id=“1”>
            <IndivName>
                  <Disj>
                        <NamePart Type=“given name” Level=“3”>
                              Allen
                        </NamePart>
                        <NamePart Type=“given name” Level=“3”>
                              Albert
                        </NamePart>
                  </Disj>
                  <NamePart Type=“surname” Level=“1”>
                        Jones
                  </NamePart>
            </IndivName>
      </IndividualRec>
...
</GEDCOM>
```

Figure 4: Sample GEDCOM XML file containing a <Disj> tag, shown in bold print. This particular example means that the given name of the individual is either “Allen” or “Albert.”

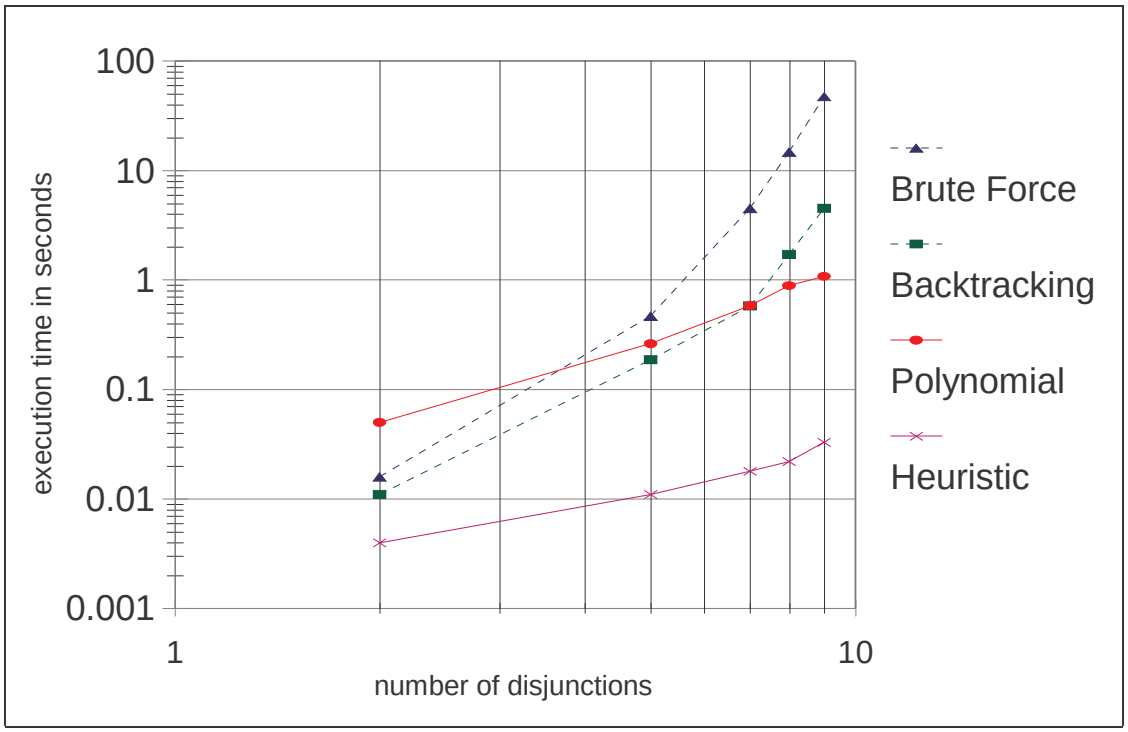

Figure 5: Timing results for brute-force, backtracking, polynomial, and heuristic algorithms. Note that the scales for the x-axis and the y-axis are both logarithmic.

The query executed by these algorithms retrieves all known information about a certain individual's grandparents, based on databases of varying complexity. The time does not include disk accesses; all relevant information is initially loaded into memory. In the graph we compare execution time against the number of disjunctions in the database. Figure 6 shows the number of nodes and edges of the databases used in the test.

	Number of disjunctions	Number of possible interpretations	Number of nodes	Total number of edges
Database 1	2	4	24	33
Database 2	5	32	60	89
Database 3	7	128	78	144
Database 4	8	256	101	183
Database 5	9	512	111	195

Figure 6: Sizes of the tested databases

Because both axes of the graph have a logarithmic scale, a straight line indicates a polynomial-time algorithm, while a curved line shows an exponential algorithm. The graph shows two interesting conclusions. First, for the test with only two disjunctive arcs, both the brute-force and the backtracking algorithms out-perform the polynomial-time algorithm. This is because with such a low number of disjunctions, there are only four possible models, so calculating every possible model of the database takes about the same time as four normal queries, whereas the [LYY95] algorithm has enough overhead to incur more cost than benefit for such a small number of possible interpretations. The polynomial algorithm begins to have a significant time advantage after about 8 disjunctive arcs. Second, we can see a rapidly increasing benefit to the polynomial-time algorithm. If we extrapolate the graph to 12 disjunctions, we would expect the brute-force algorithm to require more than 23 minutes, the backtracking algorithm to require almost 50 seconds, and the polynomial-time algorithm to require only about 1.8 seconds.[*]

[*] In a test performed separately, the polynomial-time algorithm required 1.5 seconds for 12 disjunctions.

5. Conclusion

In a genealogy application, it is possible to model disjunctive data in such a way that most queries on the data require only polynomial execution time. This makes a practical disjunctive database feasible. Some queries are still intractable, but we can detect these queries and either execute them when the problem size is reasonably small, or otherwise offer heuristics to give partial answers.

Bibliography

[GED02] "GEDCOM XML Specification, Release 6.0, Beta Version," Family and Church History Department of The Church of Jesus Christ of Latter-Day Saints, http://www.familysearch.org/GEDCOM/GedXML60.pdf, December 6, 2002.

[IV89] T. Imielinski and K. Vadaparty. "Complexity of Query Processing in Databases with OR-Objects," *Proceedings of the Eighth ACM Symposium on Principles of Database Systems (PODS)*, March 29-31, 1989, Philadelphia, Pennsylvania, pp. 51-65.

[LYY95] J. Lobo, Q. Yang, C. Yu, G. Wang, and T. Pham, "Dynamic Maintenance of the Transitive Closure in Disjunctive Graphs," *Annals of Mathematics and Artificial Intelligence*, Vol. 14, 1995, pp. 151-176.

[NIAGARA] "Niagara Query Engine," University of Wisconsin, http://www.cs.wisc.edu/niagara/.

[O03] L. Olson, "Permitting Constraint Violations in Data Storage for Integrated Data Repositories," Master's Thesis, 2003, in progress.

[OE02] L. Olson and D. Embley, "Efficiently Querying Contradictory and Uncertain Genealogical Data," 2nd Annual Workshop on Technology for Family History and Genealogical Research, April 4, 2002, Provo, Utah.

[SHORE] "Shore Object Repository," University of Wisconsin, http://www.cs.wisc.edu/shore/.

[XERCES] "Xerces C++ Parser," The Apache XML Project, http://xml.apache.org/xerces-c/index.html.