Querying Disjunctive Databases in Polynomial Time*

Lars E. Olson
Department of Computer Science
University of Illinois
leolson1@uiuc.edu

David W. Embley
Department of Computer Science
Brigham Young University
embley@cs.byu.edu

Abstract

One of the major hurdles to implementing a practical disjunctive database is the CoNP-completeness of executing queries. We demonstrate, however, that this problem is similar to the problem of computing the transitive closure of a node in a disjunctive graph. While this problem is also CoNP-complete in general, we show in this paper that there is a polynomial-time solution that solves it for certain cases. We show how to apply this solution to the problem of querying disjunctive databases, allowing us to implement a practical disjunctive database for applications that satisfy the assumptions required for our polynomial-time algorithm.

1 Introduction

Query answering on disjunctive databases is, in general, a CoNP-complete problem [IV89]. The general transitive-closure problem is also CoNP-complete, but under certain conditions it can be solved in polynomial time [LYY95]. Can a similar approach be used in disjunctive databases, and will these "certain conditions" still apply?

We can interpret values in a database as nodes in a disjunctive graph, and relationships between values as edges (with relationships to uncertain values represented as disjunctive edges). Computing the transitive closure of a node in this graph will return all related values that appear in every interpretation of the database, although the algorithm described by [LYY95] will not keep track of the path information to these values. Since path information is necessary for answering queries, this algorithm is insufficient to solve the problem. For instance, if a query asks for the year in which a person is born using the path expression Birth.Date.year[1], it is not enough to say that the value "1800" appears in every interpretation of the data, we also need to know that it is reachable through a path that answers the query, namely through the path Birth.Date.year.

It might be tempting to extend the algorithm in [LYY95] by simply keeping track of the path information. Unfortunately, as we show in this paper, even in a graph in which the algorithm can be applied in polynomial time, the number of possible paths between a pair of nodes is exponential. Thus, in order to retain a polynomial running time, we must limit the paths we check to those paths that will help answer a query. In addition, our algorithm also has to make a stronger guarantee than the [LYY95] algorithm: not only that the values returned are always reachable in any possible interpretation, but also that they are always reachable through the particular path specified in a query. Our contribution in this paper is to describe an extension to the [LYY95] algorithm that makes these guarantees and still only requires polynomial time.

The remainder of this paper is organized as follows. Section 2 describes work that has previously been done in areas related to this problem, including why this work has not fully solved this problem. It also

*This material is based upon work supported by the National Science Foundation under grant IIS-0083127.

[1] We use a sans-serif font to represent constant path expressions and edge names. For variable path expressions, we use italics.

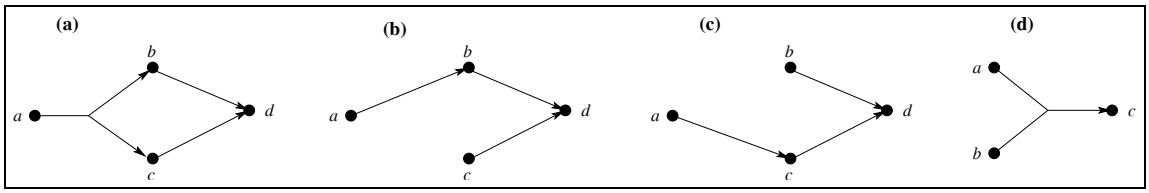

Figure 1: (a) An example disjunctive graph, with (b) and (c) its two interpretations; and an example of an edge (d) with a disjunctive tail.

introduces the concept of disjunctive graphs and the problem of solving the transitive closure of nodes in these graphs. Section 3 introduces the algorithms we use to answer queries on disjunctive data. Section 4 analyzes the performance of our implementation of the algorithm for a prototype database and query engine. It also includes an analysis of the algorithm applied to a real-world data set, namely genealogical data with uncertain information. Section 5 presents our conclusions.

2 Related Work

Disjunctive databases have been examined in a few other publications [AG85, KW85, IV89]. Most notably in [IV89] it is proved that answering queries is a CoNP-complete problem, meaning that all known algorithms for calculating the correct answer to some queries require exponential time.[2] The authors of [IV89] do, however, describe a certain set of queries that can be solved in polynomial time, although these queries require us to mark the attributes in the database under which disjunctions can occur and those under which disjunctions cannot occur. Depending on the marks of these attributes and how they combine in a query, the authors of [IV89] describe criteria to determine when a query is intractable and when it is tractable. Since we wish to allow disjunctions in any part of the database, this type of query processing is insufficient.

A similar problem is discussed in [LYY95]: transitive closure in disjunctive graphs. A disjunctive graph is defined as a graph containing disjunctive edges, an example of which is shown in Figure 1a. The disjunctive edge in Figure 1a represents an edge either from node a to node b or from node a to node c. An interpretation of a disjunctive graph is defined as a copy of the graph with all disjunctive edges replaced by one of the possible edges it represents. Figures 1b and 1c show the two possible interpretations of the graph in Figure 1a. Since each disjunctive edge represents at least two possible simple edges, it is easy to show that for a graph with n disjunctive edges, there are at least 2^n possible interpretations.

The transitive closure of a node a is defined as the set of nodes that are always reachable from a in any interpretation. In Figure 1a, for example, the transitive closure of node a is $\{a,d\}$. Since the number of possible interpretations is exponential, we cannot determine this answer in polynomial time by simply calculating the intersection of reachable nodes in every interpretation. This problem is also CoNP-complete in general; however, [LYY95] gives an algorithm that will compute the answer in polynomial time *if* all the disjunctive edges have disjunctions only in the head of the edge, rather than in the tail. The edge in Figure 1d, for example, has a disjunctive tail, and thus the algorithm would not work on a graph containing this edge.

3 Query Complexity Issues

As a motivating application, we consider genealogy in which much uncertain data exists because of conflicting historical records. Figure 2 shows an example. One record might claim that a person's birth date is 12

[2] This, of course, assumes that $P \neq NP$. We make this assumption for all similar claims throughout this paper.

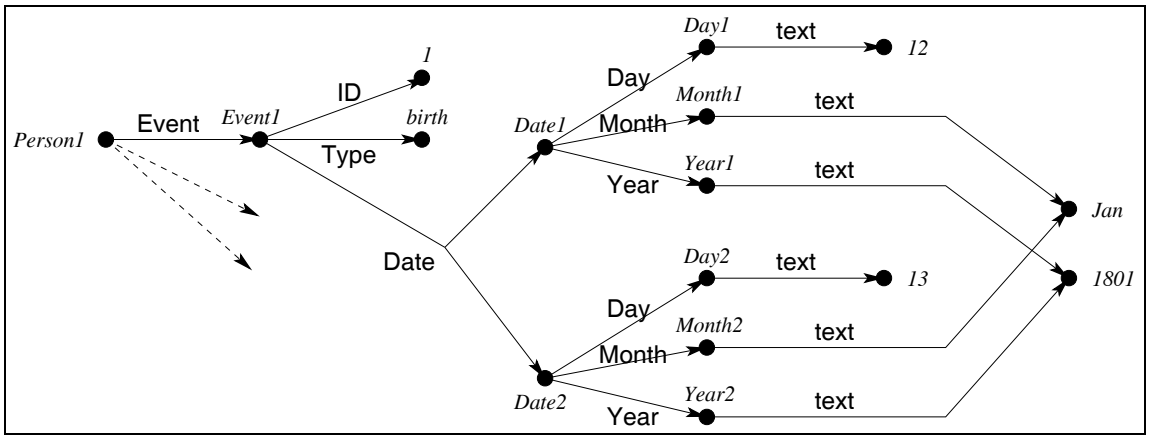

Figure 2: Example disjunctive graph.

Jan 1801, while another record might claim that the person's birth date is 13 Jan 1801. We can represent this uncertain information in a disjunctive graph by a disjunctive edge labeled Date that has two heads *Date1* and *Date2*, one for each possible date, as Figure 2 shows. In the graph the Day values are different, but the Month and Year values are the same. Thus, the transitive closure gives us the certain values, including *Jan* and *1801* in Figure 2. In all cases, we need the path information to interpret the data. Thus, for Figure 2 to find the data present in all possible interpretations for the query,[3] *Select Person.Event.Date.* Where Person = 'Person1' ∧ Person.Event.Type = 'birth'* should return *Person.Event.Date.Month.text = 'Jan'* and *Person.Event.Date.Year.text = '1801'*.

3.1 Foundations for a Polynomial-Time Solution

The algorithm in [LYY95] computes the set of nodes that make up the transitive closure of a particular node, but it does not compute the paths needed to reach those nodes. The modifications we make to the algorithm (1) keep track of path information, and (2) ensure that all relevant paths will be computed.

To help explain the modifications, we first introduce some formal definitions. We then establish the need for a polynomial-time solution by showing that the number of possible paths is exponential.

Definition 1. *A* node *is any string value designated as representing a database value.*

Definition 2. *An* edge *is an ordered triple (T,H,n) where T and H are non-empty sets of nodes and n is a string. We call T the* tail *of the edge, H the* head *of the edge, and n the* name *of the edge. We say that T is a* disjunctive tail *if $|T| > 1$. Similarly, we say that H is a* disjunctive head *if $|H| > 1$. The edge is a* disjunctive edge *if either the head or the tail is disjunctive; otherwise it is a* simple edge.

For example, ({*Day1*}, {*12*}, text) is a simple edge in Figure 2, and ({*Event1*}, {*Date1*, *Date2*}, Date) is a disjunctive edge (since the head is disjunctive).

Definition 3. *A* disjunctive graph *is an ordered pair (N,E) where N is a set of nodes and E is a set of edges such that for each $(T,H,n) \in E$, $T \subseteq N$ and $H \subseteq N$.*

Definition 4. *A* path *between two nodes x and y in a disjunctive graph (N,E) is a sequence $(p_1, p_2, \ldots, p_k)$ of edge names such that:*

[3] This example query is expressed in an SQL-like or Lorel-like [AQM96] syntax; this is simply for readability. There are many issues involved with querying disjunctive data that are beyond the scope of this paper. It is not our our purpose here to study disjunctinve query languages, but rather time and space complexity of query processing for disjunctive databases.

1. *If the length of the sequence is 1, then there exists an edge (T,H,p_1) in E such that $x \in T$ and $y \in H$.*

2. *If the length of the sequence is greater than 1, then for some $z \in N$, there exists an edge (T,H,p_1) in E such that $x \in T$ and $z \in H$, and there exists a path $(p_2,\ldots,p_k)$ from z to y.*

An equivalent representation of a path $(p_1,p_2,\ldots,p_k)$ is $p_1.p_2.\cdots.p_k$. The length *of a path $p_1.p_2.\cdots.p_k$ is k, the length of the sequence. We can also represent the length of a path $p_1.p_2.\cdots.p_k$ by $|p_1.p_2.\cdots.p_k|$.*

For example, in the graph in Figure 2, Month.text is a path from node *Date1* to node *Jan*; it is also a path from *Date2* to *Jan*. Note that this definition makes a path potentially ambiguous (i.e. makes it possible for a path to refer to several different segments of the graph) and makes no guarantees about whether the path exists in every interpretation of a disjunctive graph.

Definition 5. *An edge (T,H,n)* matches *a path $p_1.p_2.\cdots.p_k$ at position i for $1 \le i \le k$ if $n = p_i$. We can equivalently say that the name of the edge n matches $p_1.p_2.\cdots.p_k$ at position i.*

For example, the edge ($\{Month1\}$, $\{Jan\}$, text) matches the path Month.text in the second position.

Definition 6. *A* subpath *of a path $p_1.p_2.\cdots.p_k$ is a sequence $p_i.p_{i+1}.\cdots.p_{i+j}$ where $1 \le i \le i+j \le k$. The* suffix *of a path $p_1.p_2.\cdots.p_k$ for $k \ge 2$ is the subpath $p_2.\cdots.p_k$. Note that the suffix is not defined for a path with length less than 2.*

For example, Date.Month is a subpath of Event.Date.Month.text. Date.Month.text is another subpath which is also the suffix of Event.Date.Month.text.

Definition 7. *A node y is* definitely reachable *through path p from node x in graph G if every interpretation of G contains a path p from x to y.*

In Figure 2, for example, the node *Jan* is definitely reachable from *Event1* through path Date.Month.text, but node *Date1* is not definitely reachable from *Event1* through path Date because there exists an interpretation of the graph where there is no path from *Event1* to *Date1* through Date (namely, the interpretation where the Date edge goes from *Event1* to *Date2* instead of *Date1*).

Definition 8. *A* node of interest to a path *p* from a node *x* in the context of *p′, where p′ is a subpath of p, is a node y that is definitely reachable from x through p′.*

For example, the node *Jan* is a node of interest to path Date.Month.text from node *Event1* in the context of the full path Date.Month.text in Figure 2. The node *Month1* is a node of interest to path Month.text from node *Date1* in the context of the subpath Month.

To obtain a polynomial-time solution, we must limit the number of paths we store and work with in the algorithm because the number of possible paths between a pair of nodes can be exponential. Consider a graph such that the maximum number of distinct paths between a pair of nodes is n, and let b and c be those nodes, as Figure 3 shows. Let "$p_{1,1}.\cdots.p_{1,k_1}$", ..., "$p_{n,1}.\cdots.p_{n,k_n}$" be the paths between b and c. If we add another node a and add two edges A and B from a to b, then for paths from a to c, we prepend each of the n paths with A (i.e. $\mathsf{A}.p_{1,1}.\cdots.p_{1,k_1}$, ..., $\mathsf{A}.p_{n,1}.\cdots.p_{n,k_n}$) and each of the n paths with B (i.e. $\mathsf{B}.p_{1,1}.\cdots.p_{1,k_1}$, ..., $\mathsf{B}.p_{n,1}.\cdots.p_{n,k_n}$) to obtain all paths from a to c, for a total of $2n$ paths. Since adding only two edges and one node to a graph could double the maximum number of distinct paths between a pair of nodes, the number of paths in a graph is, in the worst case, exponential. Thus, a naïve solution of modifying the [LYY95] algorithm to keep track of all of the paths between each pair of nodes would require exponential running time.

We solve this problem by only storing the paths that are relevant to a given query—namely, subpaths of the path expressions in the query. We do this by stepping through the [LYY95] algorithm, adding only the

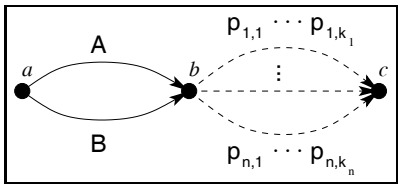

Figure 3: Graph to demonstrate why the number of paths is exponential.

edges that match a path in the query in the order in which they match the path.[4] In this way, we do not need to consider any nodes reachable (whether definitely reachable or not) beyond the current edge being added.

For example, consider Figure 2 and suppose we are determining reachability from *Person1* through the path Event.Date.Year.text. After adding the edges labeled Event, we know that node *Event1* is definitely reachable from node *Person1*. At this position in the path, we need not consider any other nodes beyond *Event1*; neither need we consider any of the possible paths to those nodes. Similarly, when adding the edges labeled Date, we can ignore any nodes beyond *Date1* and *Date2*. Next we add the edges labeled Year. When we add the first edge we can ignore any nodes beyond *Year1*, and when we add the second edge we can ignore any nodes beyond *Year2*. Finally, we add the edges labeled text. In our example, there are six such edges, but since most of them cannot be extended into subpaths that are relevant to the query, we need not keep track of them. (The edge leading to the node *Jan*, for example, can only be extended to the subpath Month.text, which is not relevant to the query path.)

At this point, it is intuitive to see that we have added enough edges to determine that the node *1801* is definitely reachable from the node *Person1* through the path Event.Date.Year.text. How is the algorithm able to arrive at this same conclusion? As we show in our first theorem, discussed next, knowing that a node is definitely reachable through a given path reduces to building a structure called a *path-connection* from the graph. Using these limitations on the number of paths to search for path-connections, we can correctly answer the query and guarantee a polynomial running time.

3.2 Path-connections and Reachability

We now introduce the concept of a *path-connection*, and show that, under certain conditions, finding all path-connections is equivalent to finding the set of all definitely reachable nodes.

Definition 9. *A* path-connection *through path p from node x to node y in a disjunctive graph (N,E) is defined as a tuple (x,S,y,p) where $x,y \in N$ and $S \subseteq N$ and either:*

1. *$|p| = 1$, $S = \{y\}$, and there exists an edge $(\{x\},S,p)$ (i.e. a simple edge from x to y) in E; or*
2. *$|p| > 1$ (in which case, let p_1 be the first edge of p and p_s be the suffix of p), and there exists an edge $(\{x\},S,p_1)$ in E, and for each $u \in S$ there is a path-connection through p_s from u to y.*

For example, in Figure 2, (*Month1*, {*Jan*}, *Jan*, text) is a path-connection through the path text from *Month1* to *Jan*, since there is an edge ({*Month1*}, {*Jan*}, text) in the graph, thus satisfying Case 1. (*Date1*, {*Month1*}, *Jan*, Month.text) is a path-connection through the path Month.text from *Date1* to *Jan*, since there is a path-connection through text from *Month1* to *Jan*, and there is an edge ({*Date1*}, {*Month1*}, Month) in the graph, thus satisfying Case 2. Likewise, (*Event1*, {*Date1*, *Date2*}, *Jan*, Date.Month.text) is a path-connection through the path Date.Month.text from *Event1* to *Jan*, since both *Date1* and *Date2* have path-connections through Month.text to *Jan* and the edge ({*Event1*}, {*Date1*, *Date2*}, Date) exists.

[4] If an edge matches at more than one position in the path, we must add it once for each position. Since we only do this at most once at each position of the path and since paths are finite, we are still within a polynomial time bound.

Theorem 1. *Let $G = (N,E)$ be a graph such that no edge in E has a disjunctive tail, and let p be a path such that no disjunctive edge in E matches p at more than one position. For any nodes x and y in G, there is a path-connection through p from x to y if and only if y is definitely reachable from x through path p in every interpretation of G.*

Proof. (We prove the ONLY IF part by induction on the length of path p.)

Basis: If the length of p is 1, then the path-connection from x to y must satisfy Case 1 of Definition 9; that is, $S = \{y\}$ and there exists an edge $(\{x\},\{y\},p)$ in G. Since this is not a disjunctive edge, it must appear in every interpretation of G, and thus y is definitely reachable from x through this edge in every interpretation.

Induction: Assume the length of p is greater than 1. Thus the path-connection from x to y must satisfy Case 2 of Definition 9; that is, for each $u \in S$ there is a path-connection through p_s from u to y, and there exists an edge $(\{x\},S,p_1)$ in G, where p_1 is the first edge of p and p_s is the suffix of p. Since $p = p_1.p_s$, $|p_s| = |p| - 1$. Thus, by our inductive hypothesis, for each $u \in S$, y is definitely reachable from u through p_s in every interpretation of G. Given any interpretation of G, we know that the interpretation must contain an edge $(\{x\},\{u\},p_1)$ for some $u \in S$ because to construct an interpretation, we must replace a disjunctive edge with one of the simple edges it might represent. (If $|S| = 1$, then $(\{x\},S,p_1)$ represents a simple edge, and the edge itself appear in any interpretation of G. The argument remains the same.) Since we already know that any interpretation must contain a path from u to y labeled p_s, we can concatenate the paths p_1 and p_s to get a path from x to y labeled $p_1.p_s = p$.

(For the IF part, we prove by contradiction that if there is no path-connection through p from x to y, then there exists an interpretation I in which y is not definitely reachable from x through path p.)

Given graph G, construct an interpretation I as follows: for each simple edge $(\{u\},\{v\},n)$ in G, add a corresponding edge $(\{u\},\{v\},n)$ to I. For each disjunctive edge $(\{u\},S,q_1)$ in G, by the hypothesis there is at most one position in p that q_1 can match. If q_1 does not match at any position in p then replace $(\{u\},S,q_1)$ with $(\{u\},\{v\},q_1)$ for any $v \in S$. Otherwise, let q be the sub-path of p that begins at q_1 and ends at the last edge of p. If there is a path connection through q from u to y, then replace $(\{u\},S,q_1)$ with $(\{u\},\{v\},q_1)$ for any $v \in S$. Otherwise, there is no path connection through q from u to y, so by definition, we have two possibilities:

1. $|q| = 1$, and thus $q = q_1$. Then either $S \neq \{y\}$ or the edge $(\{u\},S,q_1)$ does not exist. Since we know this edge exists, we are left with $S \neq \{y\}$. Since an edge cannot have an empty tail, S must contain some element $v \neq y$. Replace $(\{u\},S,q_1)$ with $(\{u\},\{v\},q_1)$ in I.

2. $|q| > 1$, and thus $q = q_1.q_s$ where q_s is the suffix of q. Then either there exists some $v \in S$ such that there is no path-connection through q_s from v to y, or the edge $(\{u\},S,q_1)$ does not exist. Since we know this edge exists, there must exist some such $v \in S$. Replace $(\{u\},S,q_1)$ with $(\{u\},\{v\},q_1)$ in I.

Since we have replaced each edge in G with an edge in I, I is a valid interpretation of G. Now, assume that y is definitely reachable from x through path p in I. Let z be the last node on the path before y, and let q be the name of the edge from z to y on this path. Then there must be a path-connection through q from z to y in G, because either $(\{z\},\{y\},q)$ was created from a simple edge $(\{z\},\{y\},q)$ in G, in which case $(\{z\},\{y\},q)$ exists in any interpretation of G, or it was created from a disjunctive edge $(\{z\},S,q)$, and since $|q| = 1$, (from Case 1, above), if there were not a path-connection through q, this disjunctive edge in G would have been replaced by edge $(\{z\},\{v\},q)$ for some $v \neq y$.

Since there is at least one node in the path (namely, z) that has a path-connection to y through a subpath of p, and at least one node in the path (namely, x) that does not, there must be two consecutive nodes u and v on the path such that there is a path-connection from v to y through a subpath of p in G (let q_s be this subpath), but no such path-connection from u to y. Let $(\{u\},\{v\},q_1)$ be the edge connecting u to v along

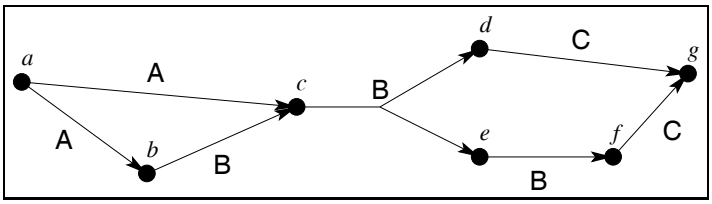

Figure 4: Graph that does not satisfy the conditions of Theorem 1.

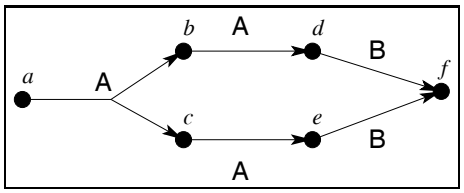

Figure 5: Graph that does not satisfy the conditions of Theorem 1, but would still work with the modified algorithm.

path p in I. Since u and v are consecutive nodes, $q_1.q_s$ is also a subpath of p. If this edge had been generated from a simple edge $(\{u\},\{v\},q_1)$ in G, then since there is a path-connection through q_s from v to y, by definition there is a path-connection through $q_1.q_s$ from u to y, contradicting our assumption. Thus, the edge must have been generated from a disjunctive edge $(\{u\},S,q_1)$ with $v \in S$. We have already established that there is no path-connection from u to y through $q_1.q_s$, and since $|q_s| > 0$, $|q_1.q_s| > 1$, so Case 2 applies. But that means that since $(\{u\},S,q_1)$ was replaced by edge $(\{u\},\{v\},q_1)$, there must not be a path-connection through q_s from v to y, contradicting our assumption that there is a path-connection through q_s from v to y. Therefore, there must be no such path p in I. □

Note that this theorem adds a condition that no disjunctive edge in the graph G can match at more than one position in the path. Since this condition is not required in the transitive closure algorithm in [LYY95], this means that there are graphs that the original [LYY95] algorithm can process but this algorithm cannot. It is important to explain why. The graph in Figure 4 has only one disjunctive edge, and therefore only two valid interpretations. It is easy to verify that node g is definitely reachable from node a through path A.B.B.C in both interpretations; however, there is not a path-connection through A.B.B.C from a to g in the original graph. This is because our definition of path-connection would require a path-connection through either B.B.C or B.C from c to g, which would in turn require path-connections from e to g and from d to g through either B.C or C, respectively. While each interpretation does contain one of these two paths, neither path alone appears in both interpretations.

It is also important to note that while the absence of disjunctive edges that match at more than one position is a sufficient condition for the algorithm to work, it is not a necessary condition. The disjunctive edge in the graph in Figure 5 matches both in the first position and in the second position in the path A.A.B; however, we can say that there is a path-connection from a to f through A.A.B, since both b and c have path-connections to f through A.B. The difference is that while the disjunctive edge *could* match both the first and the second A in the path, it only participates in the first. We leave the development of a stronger condition for graphs that will work with the modified algorithm as future work.

Theorem 2. *The running time of the modified path-connection algorithm is bounded by a polynomial on n, the number of nodes in the graph, e, the number of edges, p, the number of paths mentioned in the query clauses, and q, the length of the longest path mentioned in the query clauses.*

Proof. We only sketch the proof here; however, we note that it is similar to the analysis of the algorithm in [LYY95], using our path-connection definition instead of the connection definition in [LYY95].

For each of the p paths mentioned in the query clauses, we examine each position in the path and add each edge in the graph that matches at that position. This is bounded by eq. Each time we add an edge E, we only need to check the node(s) in the head of E, based on the observation stated at the end of Section 3.1. We start with the basis case of the path-connection definition: the path consisting of just E. If this case is not satisfied, we are done with E and can go on to add the next edge because any other new path-connections would have to be built from new, shorter path-connections.

If the basis case is satisfied, however, we check to see whether we can extend this path from the tail of E to a longer path-connection using any other edges we already added, of which there are at most e. If we cannot extend it, we are done and can go on to the next edge. Otherwise, we recursively try to extend this new path from the node in its tail. Overall, there are at most n such nodes, and we only need to check them at most once for each of the q positions in the path, so there will be at most nq recursive calls.

Since we have satisfied both cases of the path-connection definition, and we have examined all the possible subpaths that could compose a proper path-connection, this algorithm works correctly. In the worst case, adding an edge will require us to look at fewer than e edges at each recursive step, so it requires $O(enq)$ time. Since we add at most eq edges overall, each path requires $O(e^2nq^2)$. We run the algorithm for each of the p paths, so the entire process is bounded by $O(e^2npq^2)$. □

4 Analysis and Results

We programmed a prototype database to handle disjunctive data and perform queries. The host machine for the database was a dual-processor P3 running at 850MHz, with a 256KB cache and 512 MB of RAM, running Debian Linux (using the 2.4.20 SMP kernel). The compiler we used was the GNU C++ Compiler (gcc) version 2.95.4. The data was stored on a RAID-5 system with 205.1GB of storage space, using the Adaptec I2O 3210S version 2.4 drivers. The database backend was the Niagara XML interface for the Shore data repository [NIAGARA, SHORE].

We programmed 4 algorithms for a timing comparison. The first was a brute-force algorithm. Given a particular query, this algorithm constructs every possible interpretation of the graph, calculates the set of nodes that answer the query in each interpretation, and returns the intersection of all the sets. Since this algorithm always checks every interpretation, its running time follows an exponential curve with respect to the number of disjunctive edges.

The second algorithm was a backtracking algorithm. It performs a depth-first search on the graph. If it encounters a disjunctive edge, it replaces the edge with one of the simple edges it represents and finishes calculating the set of nodes that answer the query (similarly replacing other disjunctive edges it encounters). It then backtracks and replaces the edge with another one of the edges it represents and again calculates the set of nodes that answer the query and so forth until all possible edges have been checked. It then computes the intersection of the sets it calculated. While this algorithm avoids calculating every possible interpretation of the graph, it may run into the same disjunctive edges multiple times. In a highly connected graph with e edges, each step could in the worst case lead us to examine all $e-1$ of the other edges in the next step, which could lead to $e-2$ in the next step, and so forth. This gives us a worst-case running time[5] of $e \cdot (e-1) \cdot (e-2) \cdot \ldots = e!$.

The third algorithm executed the query using the modified algorithm we developed. It runs in polynomial time.

In the fourth algorithm we removed all disjunctive edges from the graph and answered the query based on the remaining edges. Since the disjunctive edges are removed, it does not give a complete answer;

[5] Caching results could improve this running time, at the expense of using more memory.

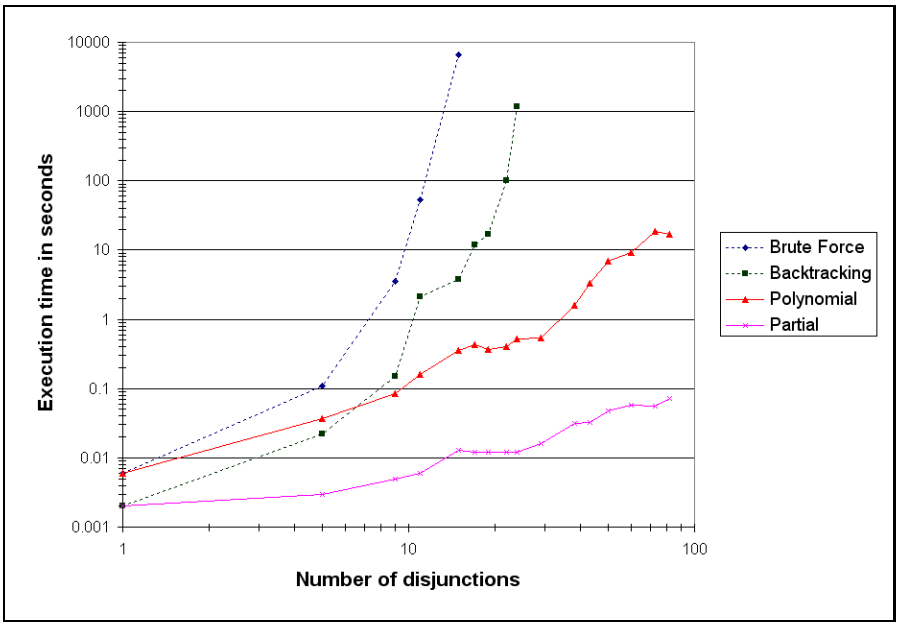

Figure 6: Comparison of query algorithms on randomly generated data.

however, it provides a comparison to typical query engines for databases that do not contain disjunctive data.

We tested the algorithms on randomly generated data, represented as a disjunctive graph. The number of nodes in each graph varied from 5 to 18, and the number of edges was set to $\frac{n(n-1)}{2}$ where n is the number of nodes in the graph. The number of head nodes in each edge followed a Poisson distribution with a minimum of one node and an average of 1.6 nodes. The path in each query had a length of 3.

Figure 6 contains the results of this test. 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. Since the edges of the graph are randomly generated, the search space in the graph varies greatly. This prevents the lines in the graph from following a smooth pattern. They do, however, generally follow the predicted behavior.

Figure 7 contains the results of running the same four algorithms on a sample genealogy data set. This data set was created by hand and was therefore not very large. Although representative of real data, it was not taken from an actual data set. Our test does, however, demonstrate that the algorithm can be useful for a real-world application.

5 Summary

We have converted an algorithm to compute the transitive closure of a node in a disjunctive graph to an algorithm to answer queries on disjunctive databases. The main modification was to keep track of path information. We have shown that if all disjunctive edges have only head disjunctions and match in at most one position in a path query, this algorithm works correctly and runs in polynomial time. Using an XML-based prototype database, we have verified with empirical tests that actual running times agree with theoretical predictions. We also applied our algorithm for querying a genealogy database where data is often uncertain and therefore disjunctive. We thus found that the algorithm can help solve existing, real-world problems.

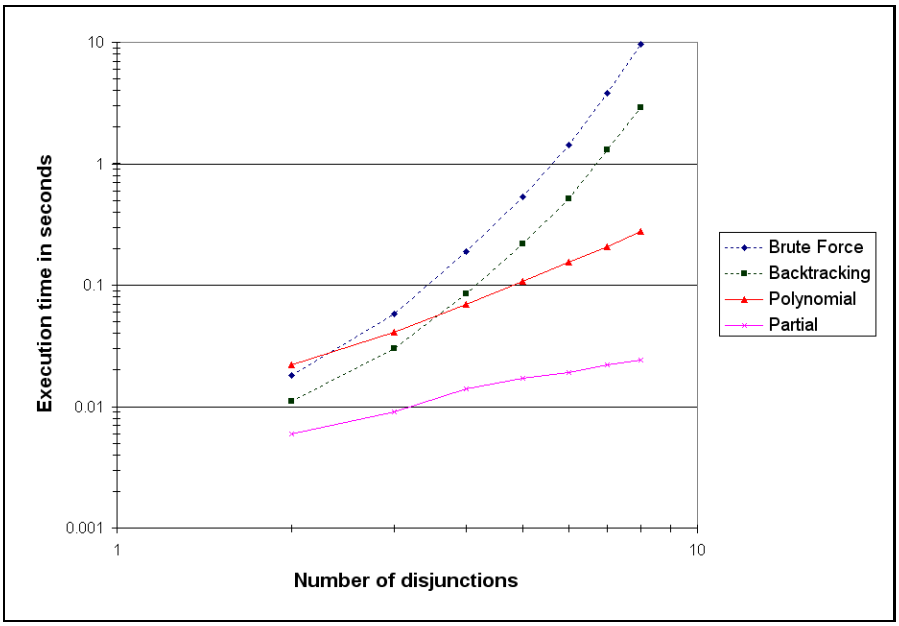

Figure 7: Comparison of query algorithms on sample genealogy data.

References

[AG85] S. Abiteboul and G. Grahne, "Update Semantics for Incomplete Databases," *Proceedings of the 11th International Conference on Very Large Databases (VLDB)*, Aug. 21-23, 1985, Stockholm, Sweden, pp. 1-12.

[AQM96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, "The Lorel query language for semi-structured data," *International Journal of Digital Libraries*, Vol. 1, No. 1, Nov. 1996, pp. 68-88.

[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)*, Mar. 29-31, 1989, Philadelphia, Pennsylvania, pp. 51-65.

[KW85] A. M. Keller and M. W. Wilkins, "On the Use of an Extended Relational Model to Handle Changing Incomplete Information," *IEEE Transactions on Software Engineering*, Vol. 11, No. 7, Jul. 1985, pp. 620-633.

[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/.

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