CS 784: Foundations of Data Management **Fall 2019**

Lecture 4: Acyclic Conjunctive Queries

Instructor: Paris Koutris

Previously we showed that the *combined complexity* of evaluating Conjunctive Queries is NP-complete. In this lecture, we show how to obtain polynomial time algorithms for some classes of CQs. Let us start with an example.

Example 4.1. *Consider the boolean k-path query:*

$$P^k() : -R_1(x_1, x_2), R_2(x_2, x_3), \dots, R_k(x_k, x_{k+1})$$

Assume that the size of every relation is $|R_i| = N$, hence the input is of size $k \cdot N$. Consider the following algorithm: we first project R_1 on the attribute x_2, then compute the semi-join with R_2, project the result on attribute x_3, and so on. Observe that at every point the size of the intermediate result is at most N. Hence, we can implement the algorithm with running time time $O(kN)$. The running time in this case is polynomial (in combined complexity)!

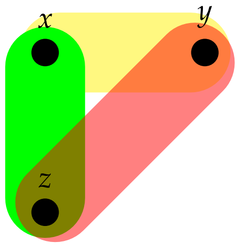
Let's see now how we can generalize this intuition to compute a larger class of queries. Notice that the output of a query may be in the worst case exponential in the input size; in this case, designing an algorithm that runs in polynomial time in the size of the input is not possible. Thus, we want to design an algorithm that is polynomial with respect to the *input and the output size*. We will show that such an algorithm is possible for the class of Conjunctive Queries called *acyclic CQs*.

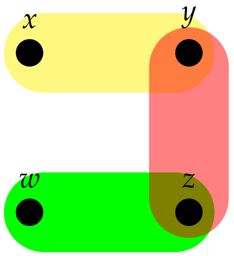
4.1 Acyclic CQs: Definition

We start by defining an alternative description of a CQ as a hypergraph.

Definition 4.2 (Query Hypergraph)**.** *The hypergraph $\mathcal{H}(q) = (V, E)$ associated to a conjunctive query q is defined as follows. The set of vertices V consists of all variables in the body of q, $\mathbf{vars}(q)$, while the set E of hyperedges contains, for each atom in the query, the set of variables that appear in this atom.*

Below are the hypergraphs for $q_0(x, y, z) : -R(x, y), S(y, z), T(z, x)$ and $q_1() : -R(x, y), S(y, z), T(z, w)$.





Intuitively, a CQ is acyclic if the hypergraph contains no cycles. If the hypergraph is a graph (which happens if every relation in the query is binary), then acyclicity coincides with the graph being a tree. But the characterization is not as clear when the hyperedges have arity more than 2. To formally define acyclicity, we will apply the following algorithm, called the **GYO algorithm** (from Graham-Yu-Ozsoyoglu).

Definition 4.3. *An* ear *in a hypergraph $\mathcal{H}$ is a hyperedge e for which there exists another hyperedge f with the following property: for every node x that does not appear exclusively in e, it must be that $x \in f$. In this case, f is called a* witness *of e.*

For example, if a hyperedge is isolated (does not intersect with other hyperedges), or if it is contained within another hyperedge, it is by definition an ear.

Example 4.4. *For the query $q_1() : -R(x,y), S(y,z), T(z,w)$, both R and T are ears, but S is not. The query $q_0(x,y,z) : -R(x,y), S(y,z), T(z,x)$ contains no ears.*

```
Algorithm 1: GYO algorithm
  Input: hypergraph $\mathcal{H}$
  Output: is $\mathcal{H}$ acyclic?
  while $\mathcal{H}$ has ears do
      $e \leftarrow$ ear of $\mathcal{H}$ ;
      remove all vertices that are exclusively in $e$ ;
      remove $e$;
  end
  if $\mathcal{H}$ is the empty hypergraph then
      return YES ;
  end
  return NO ;
```

Example 4.5. *If we apply the GYO algorithm to $\mathcal{H}(q_0)$, since no hyperedge is an ear, the algorithm will not remove anything from the hypergraph, and thus it will return that it is not acyclic. For q_1, the GYO algorithm will run as follows:*

$$\begin{array}{rll}
V = \{x,y,z,w\} & E = \{\{x,y\},\{y,z\},\{z,w\}\} & ear = T \\
V = \{x,y,z\} & E = \{\{x,y\},\{y,z\}\} & ear = R \\
V = \{y,z\} & E = \{\{y,z\}\} & ear = S \\
V = \{\} & E = \{\} &
\end{array}$$

Since the resulting hypergraph is the empty one, the algorithm returns that the query is acyclic.

The original GYO algorithm is a slight variation of the above algorithm. Instead of removing ears one at time, it repeats the following 2 steps as long as they can be applied: (i) delete a vertex v that belongs to a single hyperedge, and (ii) delete a hyperedge that is contained in another hyperedge.

This variation guarantees that the hypergraph at the end of the while loop will always be the same, independent of the sequence with which we remove hyperedges or vertices.

This above definition of acyclicity is called *α-acyclicity*. There are other definitions of acyclicity for hypergraphs (Berge-acyclicity, β-acyclicity, γ-acyclicity), but they are more restrictive.

Next, we discuss some equivalent definitions of acyclic CQs. Recall that a *forest* is a graph that is a collection of disjoint trees.

Definition 4.6 (Join Forest)**.** *A* join forest *for a CQ q is a forest $F = (V, E)$ whose vertices are the atoms in q and such that for each pair of atoms R, S having variables in common the following conditions hold:*

1. *R, S belong in the same connected component of F; and*
2. *all variables common to R and S occur on the unique path from R to S.*

In the case where F is a tree, it is called a *join tree* of the query q. It turns out that we can equivalently characterize acyclicity by looking at whether a CQ admits a join forest.

Proposition 4.7. *A Conjunctive Query q is acyclic if and only if it has a join forest.*

We next show a third equivalent characterization of acyclic CQs. Say that, given a CQ q, we want to remove from a relation R the tuples that we are certain that they do not participate in the final result of q, called *dangling tuples*. In other words, t is a dangling tuple if no valuation that leads to an output uses t.

Example 4.8. *Consider again query $q_1() : -R(x, y), S(y, z), T(z, w)$, along with the following instance: $I = \{R(a, b), S(b, c), T(c, d), R(a, b'), S(b'', c'), T(c', d')\}$. In this example, $R(a, b'), S(b'', c'), T(c', d')$ are all dangling tuples.*

We can attempt to remove dangling tuples using the *semijoin* operator. A semijoin between relations R, S is defined as:

$$R \ltimes S = \pi_{\mathbf{att}(R)}(R \bowtie S)$$

In other words, the result is the set of tuples from R that we are certain that they join with a tuple from S. Semijoins are often used in distributed databases to reduce the communication cost of joining two relations. An instance without any dangling tuples is called *globally consistent*. If we can remove all dangling tuples by using only a (finite) sequence semijoin operations, such a sequence is called a *full reducer*.

Example 4.9. *Here is one full reducer for the query q_1. You can check that it removes all dangling tuples!*

$$S := S \ltimes R$$
$$T := T \ltimes S$$
$$S := S \ltimes T$$
$$R := R \ltimes S$$

The triangle query T does not admit any full reducer. For example, consider the following instance: $\{R(a, a), R(b, b), S(a, a), S(b, b), T(a, b), T(b, a)\}$.

Lemma 4.10. *A Conjunctive Query q is acyclic if and only it admits a full reducer.*

To prove this equivalence, the idea is to compute semi-joins in the order in which we remove the ears in the GYO algorithm (and in the reverse order as well).

4.2 Acyclic CQs: PTIME Algorithm

We will now use join forests and full reducers to evaluate an acyclic CQ q over a database instance in polynomial time (combined complexity). Without loss of generality, we can assume that q is connected (since we can compute each connected component separately and then take the cartesian product). The algorithm works as follows:

Algorithm 2: Yannakakis algorithm

```
Input: Conjunctive Query $q$, instance $I$
Output: $q(I)$
apply a full reducer to $I$ to obtain a globally consistent instance $J$ ;
construct a join tree $T$ for $q$ and choose a root node ;
Let $R_1, \dots, R_n$ be a post-order traversal of the tree ;
for $i = 1, \dots, n$ do
    $S_1, \dots, S_k$ are the children of $R_i$ ;
    for $j = 1, \dots, k$ do
        $R_i^J \leftarrow \pi_{\mathbf{vars}(R_i) \cup \mathbf{head}(q)}(R_i^J \bowtie S_j^J)$ ;
    end
end
```

Theorem 4.11. *The Yannakakis algorithm evaluates an acyclic Conjunctive Query q in time polynomial in the size of the query, the input and the output.*

Proof. Why is this algorithm correct? The key property is that if we have a variable x that appears in a node and not its parent in the join tree, the variable will not appear at any ancestor (so it will not be needed further in the evaluation).

To show that we obtain the desired complexity, we will prove that no intermediate result can grow more than a polynomial in the size of the input and output. Consider some node with atom R and children $S_1, \dots, S_n$. Then the algorithm will replace this node with a new node $T' = \pi_{Y \cup Z}(T)$, where $T = R \bowtie S_1 \bowtie \dots S_n$, Y are the variables in R, and Z are the variables in $\mathbf{head}(q) \setminus Y$. Since Y, Z are disjoint, $\pi_{Y \cup Z}(T) \subseteq \pi_Y(T) \times \pi_Z(T)$. Now notice that $\pi_Y(T)$ is at most the size of R. Also, $\pi_Z(T)$ is bounded by the size of the output (but only because we took care of dangling tuples by applying the full reducer). □

Exercise 4.12. *How does Yannakakis algorithm work for the following query?*

$$q(x,t) : -R(x,y,z), S(y,v), T(y,z,u), U(z,u,w), V(u,w,t).$$

Run the GYO algorithm, find the join tree and explain what is the query plan produced by the polynomial-time algorithm.

From Yannakakis algorithm, we can obtain stronger results if we consider CQs that are boolean or full (no projections).

Proposition 4.13. *Let q be an acyclic CQ, and I an instance of size N. Then:*

- *If q is boolean, we can evaluate the query in time $O(N)$.*
- *If q is full, we can evaluate the query in time $O(N + OUT)$, where OUT is the size of the output.*

References

[Alice] S. Abiteboul, R. Hull and V. Vianu, “Foundations of Databases.”

[Y81] M. Yannakakis, “Algorithms for acyclic database schemes,” *VLDB 1981*.