3. Evaluation and Containment of Conjunctive Queries

Instructions: Work in groups of 2 students. You can write up your answers by hand (provided your handwriting is legible) or use a word processing system like Latex or Word. Note that experience has shown that Word is in general difficult to use for this kind of task. If you prefer to write up your solution by hand, submit a scanned electronic version. Please, include name and email address in your submission.

1. Evaluation of Special Types of Conjunctive Graph Queries

In this exercise we want to analyze the difficulty to evaluate certain queries over directed graphs. The formulation of the exercise is pretty long, but that should not frighten you. It is long because it contains many detailed hints.
We can store a directed graph in a database using a binary relation $edge/2$.

- The k-th *circle query* $C_k/1$ asks for all nodes x that lie on a circle of length k, that is, a path consisting of k edges that starts and ends at x. Thus, C_k is defined as

$$C_k(x) \text{ :- } edge(x, z_1), edge(z_1, z_2), \ldots, edge(z_{k-1}, x).$$

- Intuitively, the k-th *star query* $S_k/1$ asks for all nodes x that are in the center of a star with k rays. More precisely, S_k is defined as

$$S_k(x) \text{ :- } edge(x, z_1), edge(x, z_2), \ldots, edge(x, z_k).$$

- The k-th *spider-web query* asks for nodes x that are in the center of a graph that has a form similar to a spider web, that is, a star with k rays that are connected by edges forming a circle. Formally, W_k is defined as

$$\begin{aligned} W_k(x) \text{ :- } & edge(x, z_1), edge(x, z_2), \ldots, edge(x, z_k), \\ & edge(z_1, z_2), edge(z_2, z_3), \ldots, edge(z_k, z_1). \end{aligned}$$

- The k-th *clique query* $Cl_k/0$ asks whether the graph stored in the relation *edge* has a clique of size k, that is, a subgraph of size k with an edge from each node to each other node. Formally, Cl_k is defined as

$$\begin{aligned} Cl_k() \text{ :- } & edge(z_1, z_2), edge(z_1, z_3) \ldots, edge(z_1, z_k), \\ & edge(z_2, z_1), edge(z_2, z_3), \ldots, edge(z_2, z_k), \\ & \ldots \\ & edge(z_k, z_1), edge(z_k, z_2), \ldots, edge(z_k, z_{k-1}). \end{aligned}$$

To get a better idea for the pattern that these queries ask for, draw the pattern as a graph.

1. For each type of query, design an efficient algorithm to compute the set of answers $Q(\mathbf{I})$ for a query Q applied to an instance of size $\mathbf{I}$.
2. For each of the algorithms, assess the running time, expressed in $|\mathbf{I}|$ and $|Q|$, using $O(\cdot)$ ("big Ohhh" :-)) notation? Explain your assessment.

(12 Points)

Hint 1: You can describe your algorithms in natural language or in pseudo-code. It may help you to formulate your algorithm as a sequence of relational algebra operators applied to the relation *edge*. For example, an algorithm for evaluating the k-the path query P_k, defined in the lecture as

$$P_k(x) \text{ :- } edge(x, y_1), edge(y_1, y_2), \ldots, edge(y_{k-1}, y_k),$$

one can define the algorithm

$R := \pi_{edge.1}(edge)$
for $i := 1$ **to** $k - 1$ **do**
    $R := \pi_{edge.1}(edge \bowtie_{edge.2=R.1} R)$
return R,

which uses a the integer variables k and i and the relation variable R.
An assignment to a relation variable is executed in the same way as an assignment to a numerical variable: first, the expression on the rhs is evaluated, then the result is assigned to the variable on the lhs.
In the algebra expressions in the algorithm, we have referred to the arguments of a relation by numbers that are disambiguated by the relation name. For instance, "*edge*.2" referes to the second argument of the relation *edge*.

Of course, this is not the only approach one can take. Formulating the algorithm in terms of operations on graphs (e.g., labeling of nodes) is another option.

Hint 2: The running time of the algorithm above can be roughly assessed as follows.
Let $n := |\mathbf{I}|$.

1. The projection in line 1 requires duplicate elimination. This can be achieved by sorting *edge* according to the first argument and by a subsequent scan that eliminates duplicates of the first argument. This operation requires $O(n \log n)$ steps. The resulting relation R is unary and has size at most n.

2. The join in the loop (line 3) joins two relations of size at most n. The join can be computed by sorting *edge* on the second argument and then merging the result with R. This requires $O(n \log n + 2n) = O(n \log n)$ many steps. The resulting relation has size at most n.

3. The projection can be computed as above, in time $O(n \log n)$.

4. Thus, all in all the expression in the loop can be computed with $O(n \log n + n \log n) = O(n \log n)$ many steps.

5. Initially, the relation R is computed in time $O(n \log n)$. The loop is executed $k-1$ times, at a cost of $O(n \log n)$ per execution. Thus, the whole execution costs
$$O(k\, n \log n) = O(|Q| \times |\mathbf{I}| \times log\, |\mathbf{I}|).$$

2. Evaluation of Conjunctive Queries with Unary Relation Symbols

Recall that relational conjunctive queries[1] have only relational atoms in their body, and no equalities or inequalities. We know that the combined complexity of evaluating relational conjunctive queries is NP-complete. However, the reduction used queries with binary relation symbols.

What can you say about the difficulty of evaluating relational boolean conjunctive queries that have only unary relations in their body (that is, relations of arity 1)? Is this an NP-hard, a coNP-hard, or a polynomial time problem?

[1] sometimes also called "simple" conjunctive queries

Distinguish between the problem for

1. relational queries (that is, queries without built-in predicates)
2. general conjunctive queries, which may contain the built-in predicates "$<$", "$\leq$", and "$\neq$".

To prove NP-hardness or coNP-hardness of a problem, provide a reduction from a known NP-hard or coNP-hard problem to the new one. To prove that it is in polynomial time, give an algorithm, show that it solves the problem, and explain why it runs in polynomial time.

(12 Points)

3. Containment of Relational Conjunctive Queries without Self Joins

A conjunctive query has a *self join* if its body contains two relational atoms with the same relation symbol. Thus, in the body of a query without self join, any two relational atoms have distinct relation symbols.

We know that containment is NP-complete for arbitrary relational conjunctive queries.

Question: How difficult is it to decide containment of relational conjunctive queries that have no self join? Can this problem be solved in polynomial time? Or is it NP-complete?

(6 Points)

Submission: Wed, 9 May 2018, 16:00 hrs, by email to

`werner DOT nutt AT unibz DOT it.`