Principles of Managing Uncertain Data

Lecture 3: Querying Complexity
Table of Contents

1. Introduction
2. Data and Combined Complexity
3. Parameterized Complexity
4. Input-Output Complexity
1. Introduction

2. Data and Combined Complexity

3. Parameterized Complexity

4. Input-Output Complexity
Complexity Measures for Database Querying

- Classical complexity theory considers two types of problems:
  - Decision: given $x$, decide whether $x$ is a yes/no input
  - Function: given $x$, compute the $f(x)$ for some function $f$
Classical complexity theory considers two types of problems:
- **Decision**: given $x$, decide whether $x$ is a yes/no input
- **Function**: given $x$, compute the $f(x)$ for some function $f$

Database queries are typically so general that there are no "easy" (e.g., polynomial-time) problems.
Complexity Measures for Database Querying

- Classical complexity theory considers two types of problems:
  - Decision: given $x$, decide whether $x$ is a yes/no input
  - Function: given $x$, compute the $f(x)$ for some function $f$
- Database queries are typically so general that there are no “easy” (e.g., polynomial-time) problems
- There are certain general parameters of a query-evaluation problem that have a major impact on the complexity, and allow to isolate significant “islands of tractability”
Complexity Measures for Database Querying

- Classical complexity theory considers two types of problems:
  - Decision: given $x$, decide whether $x$ is a yes/no input
  - Function: given $x$, compute the $f(x)$ for some function $f$
- Database queries are typically so general that there are no “easy” (e.g., polynomial-time) problems
- There are certain general parameters of a query-evaluation problem that have a major impact on the complexity, and allow to isolate significant “islands of tractability”
- Hence, we often adopt finer notions of complexity
Database vs. Query Size

- The most important feature of query evaluation is that databases are typically large, whereas queries/schemas are tiny
The most important feature of query evaluation is that databases are typically large, whereas queries/schemas are tiny. This gives rise to various notions of complexity:

- *Data complexity*
- *Parameterized complexity*
Queries may be asked to compute huge answers (e.g., Cartesian products)
Queries may be asked to compute huge answers (e.g., Cartesian products)

Is a query hard because it is asked to compute a huge object? Or it is hard even for a small output?

- What is the complexity *per output bit*?
Queries may be asked to compute huge answers (e.g., Cartesian products)

- Is a query hard because it is asked to compute a huge object? Or it is hard even for a small output?
  - What is the complexity *per output bit*?

- This gives rise to additional notions of complexity:
  - *Input-output complexity*
  - And in particular, *enumeration complexity*
We will learn the aforementioned notions of complexity
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Introduction</td>
</tr>
<tr>
<td>2.</td>
<td>Data and Combined Complexity</td>
</tr>
<tr>
<td>3.</td>
<td>Parameterized Complexity</td>
</tr>
<tr>
<td>4.</td>
<td>Input-Output Complexity</td>
</tr>
</tbody>
</table>
We consider computational problems that involve one or more of the following components:

- Schema $\mathcal{S}$
- A set $\Sigma$ of constraints
- A query $Q$
- A database $I$
We consider computational problems that involve one or more of the following components:

- Schema $S$
- A set $\Sigma$ of constraints
- A query $Q$
- A database $I$

- **Combined complexity:** everything is given as input
- **Data complexity:** $I$ is given as input, everything else is fixed

  Formally, we consider infinitely many computational problems $P_{S,\Sigma,Q}$, one per combination of $S$, $\Sigma$ and $Q$.
Example: Complexity of CQ Answering

**Problem Def. (Boolean CQ Evaluation)**

Given a schema $S$, a Boolean CQ $Q$ over $S$ and an instance $I$ over $S$, determine whether $Q(I) = \text{true}$. 

We will show that this problem is NP-complete under combined complexity, by reduction from the Clique problem.
Example: Complexity of CQ Answering

**Problem Def. (Boolean CQ Evaluation)**

Given a schema $S$, a Boolean CQ $Q$ over $S$ and an instance $I$ over $S$, determine whether $Q(I) = \text{true}$.

We will show that this problem is NP-complete under *combined complexity*, by reduction from the Clique problem.
Problem Def. (Boolean CQ Evaluation)

Given a schema $S$, a Boolean CQ $Q$ over $S$ and an instance $I$ over $S$, determine whether $Q(I) = \text{true}$.

We will show that this problem is NP-complete under combined complexity, by reduction from the Clique problem.

Problem Def. (Clique)

Given a graph $G = (V, E)$ and a number $k$, determine whether $G$ contains a clique of size $k$, that is, a subset $U$ of $V$ such that $|U| = k$ and every two nodes in $U$ are neighbours.
Reduction

- Given $G = (V, E)$ with $V = \{1, \ldots, n\}$, and $k$, construct:
  - $S = \{R_E/2\}$
  - $I_G = \{R_E(i, j) \mid \{i, j\} \in E \text{ and } i < j\}$
  - $Q_k$ is a CQ with existential variables $X_1, \ldots, X_k$, and an atom $R_E(X_i, X_j)$ for every $i$ and $j$ with $1 \leq i < j \leq k$
Reduction

- Given $G = (V, E)$ with $V = \{1, \ldots, n\}$, and $k$, construct:
  - $S = \{R_E/2\}$
  - $I_G = \{R_E(i, j) \mid \{i, j\} \in E \text{ and } i < j\}$
  - $Q_k$ is a CQ with existential variables $X_1, \ldots, X_k$, and an atom $R_E(X_i, X_j)$ for every $i$ and $j$ with $1 \leq i < j \leq k$

- For example, suppose that $G$ is the following graph:

```
1 -- 2
|   |
|   |
3 -- 4
```
Given $G = (V, E)$ with $V = \{1, \ldots, n\}$, and $k$, construct:

- $S = \{R_E/2\}$
- $I_G = \{R_E(i, j) \mid \{i, j\} \in E \text{ and } i < j\}$
- $Q_k$ is a CQ with existential variables $X_1, \ldots, X_k$, and an atom $R_E(X_i, X_j)$ for every $i$ and $j$ with $1 \leq i < j \leq k$

For example, suppose that $G$ is the following graph:

For

$Q_3() :\neg R_E(X_1, X_2), R_E(X_1, X_3), R_E(X_2, X_3)$
The reduction is correct since the following two are equivalent:

1. $G$ has a clique of size at least $k$
2. $Q_k(I_G) = \text{true}$
Correctness

- The reduction is correct since the following two are equivalent:
  1. $G$ has a clique of size at least $k$
  2. $Q_k(I_G) = \text{true}$

- Hence, determining whether $Q(I) = \text{true}$, given $S$, $Q$ and $I$, is NP-hard
  - Membership in NP is straightforward, hence, the problem is NP-complete
Correctness

- The reduction is correct since the following two are equivalent:
  1. $G$ has a clique of size at least $k$
  2. $Q_k(I_G) = \text{true}$

- Hence, determining whether $Q(I) = \text{true}$, given $S$, $Q$ and $I$, is NP-hard
  - Membership in NP is straightforward, hence, the problem is NP-complete

- Note: The schema $S$ does not depend on the input $(G, k)$, but the size of $Q$ is quadratic in $k$
What is the data complexity of answering a query in RA?
What is the data complexity of answering a query in RA?

- We consider the problem $P_{S,Q}$ of computing the answers for a query $Q$ in RA (Relational Algebra) over a given input instance $I$ over $S$
**Data Complexity**

*What is the data complexity of answering a query in RA?*

- We consider the problem $P_{S,Q}$ of computing the answers for a query $Q$ in RA (Relational Algebra) over a given input instance $I$ over $S$.
- The naive way of straightforwardly executing $Q$ runs in polynomial time!
  - *What is the degree of the polynomial?*
What is the data complexity of answering a query in RA?

- We consider the problem $P_{S,Q}$ of computing the answers for a query $Q$ in RA (Relational Algebra) over a given input instance $I$ over $S$.
- The naive way of straightforwardly executing $Q$ runs in polynomial time!
  - What is the degree of the polynomial?
- As a special case, CQ evaluation is in polynomial time under data complexity.
  - Note that data complexity is insensitive to the representation of the query.
Under *combined complexity*, CQ evaluation is intractable

- Boolean CQ evaluation is NP-complete
- The non-emptiness problem for CQ evaluation (i.e., is there at least one tuple in the result?) is NP-complete
Summary for CQs

- Under *combined complexity*, CQ evaluation is intractable
  - Boolean CQ evaluation is NP-complete
  - The non-emptiness problem for CQ evaluation (i.e., is there at least one tuple in the result?) is NP-complete
- Under *data complexity*, CQ evaluation is solvable in polynomial time
Summary for CQs

- Under *combined complexity*, CQ evaluation is intractable
  - Boolean CQ evaluation is NP-complete
  - The non-emptiness problem for CQ evaluation (i.e., is there at least one tuple in the result?) is NP-complete
- Under *data complexity*, CQ evaluation is solvable in polynomial time
  - That is, for every CQ $Q$ there exists a polynomial-time algorithm $A_Q$ to compute $Q(I)$ on a given instance $I$
Under *combined complexity*, CQ evaluation is intractable
  
  - Boolean CQ evaluation is NP-complete
  - The non-emptiness problem for CQ evaluation (i.e., is there at least one tuple in the result?) is NP-complete

Under *data complexity*, CQ evaluation is solvable in polynomial time
  
  - That is, for every CQ $Q$ there exists a polynomial-time algorithm $A_Q$ to compute $Q(I)$ on a given instance $I$
  - The naive way gives a polynomial running time where the degree depends on the query (next: *Is it necessary?*)
Table of Contents

1. Introduction
2. Data and Combined Complexity
3. Parameterized Complexity
4. Input-Output Complexity
Parameterized Complexity

- Parameterized complexity provides a yardstick of efficiency somewhere between data complexity and combined complexity
Parameterized Complexity

- *Parameterized complexity* provides a yardstick of efficiency somewhere between *data complexity* and *combined complexity*

- Intuitively, we would like to have evaluation in polynomial time in the size of the database, but we allow the query to affect *only the coefficient* of the polynomial; not the *degree* of the polynomial
Parameterized Complexity

- **Parameterized complexity** provides a yardstick of efficiency somewhere between data complexity and combined complexity.

- Intuitively, we would like to have evaluation in polynomial time in the size of the database, but we allow the query to affect only the coefficient of the polynomial; not the degree of the polynomial.

- This is formalized and explored in the framework of parameterized complexity.
  - Where the parameter here is the size of the query.
Formal Definition

- Recall: a decision problem is a set of strings (representing problem instances)
- A decision problem $D$ is solvable in polynomial time if there exists an algorithm $A$ and a polynomial $p$ such that $A$:
  - solves $D$ (i.e., decides whether a given input string $x$ is in $D$)
  - terminates in at most $p(|x|)$ steps on every input $x$
Formal Definition

- Recall: a decision problem is a set of strings (representing problem instances).
- A decision problem $D$ is solvable in polynomial time if there exists an algorithm $A$ and a polynomial $p$ such that $A$:
  - solves $D$ (i.e., decides whether a given input string $x$ is in $D$)
  - terminates in at most $p(|x|)$ steps on every input $x$
- A parameterized decision problem is a set of pairs $(x, k)$, where $x$ is a string and $k$ is a natural number—the parameter
Formal Definition

- Recall: a *decision problem* is a set of strings (representing problem instances).
- A decision problem $D$ is *solvable in polynomial time* if there exists an algorithm $A$ and a polynomial $p$ such that $A$:
  - solves $D$ (i.e., decides whether a given input string $x$ is in $D$)
  - terminates in at most $p(|x|)$ steps on every input $x$
- A *parameterized decision problem* is a set of pairs $(x, k)$, where $x$ is a string and $k$ is a natural number—the *parameter*.
- A parameterized decision problem $D$ is *Fixed Parameter Tractable*, or $FPT$, if there exists an algorithm $A$, a (computable) function $f$ and a polynomial $p$ such that $A$:
  - solves $D$ (decides whether a given $(x, k)$ is in $D$)
  - terminates in at most $f(k) \cdot p(|x|)$ steps on every input $(x, k)$, where $f$ is computable and $p$ is a polynomial.
Vertex Cover

**Input:** Graph $g$, natural number $k$

**Goal:** Determine whether there is a *vertex cover* of size $k$
**Vertex Cover**

**Input:** Graph $g$, natural number $k$

**Goal:** Determine whether there is a *vertex cover* of size $k$

- Recall: a *vertex cover* is a set of nodes that hits all edges
Vertex Cover

**Input:** Graph \( g \), natural number \( k \)

**Goal:** Determine whether there is a *vertex cover* of size \( k \)

- Recall: a *vertex cover* is a set of nodes that hits all edges
- *Why is this problem in polynomial time for every fixed \( k \)?*
Parameterized Vertex Cover

Vertex Cover

**Input:** Graph $g$

**Parameter:** $k$

**Goal:** Determine whether there is a *vertex cover* of size $k$
Notation

- Let $G = (V, E)$ be a graph
- Let $v \in V$ be a node of $G$
Notation

- Let $G = (V, E)$ be a graph
- Let $v \in V$ be a node of $G$
- We denote by $G - v$ that graph $G'$ that is obtained from $G$ by removing $v$ and all of its incident edges
Notation

- Let $G = (V, E)$ be a graph
- Let $v \in V$ be a node of $G$
- We denote by $G - v$ that graph $G'$ that is obtained from $G$ by removing $v$ and all of its incident edges
- That is, $G - v$ is the graph $G' = (V', E')$ where

$$V' = V \setminus \{v\} \quad E' = \{ e \in E \mid v \notin e \}$$
FPT Algorithm

\textbf{VertexCover}(g, k)

1. if \( k < 0 \) then
2. \hspace{1em} return false
3. if \( k \geq 0 \) and \( g \) has no edges then
4. \hspace{1em} return true
FPT Algorithm

VertexCover\((g, k)\)

1 if \(k < 0\) then
2 | return false
3 if \(k \geq 0\) and \(g\) has no edges then
4 | return true
5 select an arbitrary edge \(e = \{u, v\}\);
6 if VertexCover\((g - u, k - 1)\) then
7 | return true
8 if VertexCover\((g - v, k - 1)\) then
9 | return true
10 return false;
FPT Algorithm

VertexCover\((g, k)\)

1. if \(k < 0\) then
2. \hspace{1em} return false
3. if \(k \geq 0\) and \(g\) has no edges then
4. \hspace{1em} return true
5. select an arbitrary edge \(e = \{u, v\}\);
6. if VertexCover\((g - u, k - 1)\) then
7. \hspace{1em} return true
8. if VertexCover\((g - v, k - 1)\) then
9. \hspace{1em} return true
10. return false;

Why is this algorithm FPT?
Hardness in Parameterized Complexity

- Like classical complexity, in parameterized complexity there are also problems that are strongly assumed to be hard
  - That is, not FPT
Like classical complexity, in parameterized complexity there are also problems that are strongly assumed to be hard.

- That is, not FPT.

This is captured by the $W$-hierarchy (that we do not define formally here).

- $W[1]$-hard is not likely to be FPT.
Hardness in Parameterized Complexity

- Like classical complexity, in parameterized complexity there are also problems that are strongly assumed to be hard
  - That is, not FPT
- This is captured by the *W-hierarchy* (that we do not define formally here)
  - *W[1]-hard* is not likely to be FPT
  - *W[2]-hard* is harder than *W[1]*, etc.
- Examples of *W[1]-hard* problems:
  - Independent set: \( \{(g, k) \mid g \text{ has an ind. set of size } k \} \)
  - Clique: \( \{(g, k) \mid g \text{ has a clique of size } k \} \) (same problem)
  - We will see another one next
Hardness in Parameterized Complexity

- Like classical complexity, in parameterized complexity there are also problems that are strongly assumed to be hard
  - That is, not FPT
- This is captured by the $W$-*hierarchy* (that we do not define formally here)
  - $W[1]$-hard is not likely to be FPT
- Examples of $W[1]$-hard problems:
  - Independent set: $\{(g, k) \mid g$ has an ind. set of size $k$\}
  - Clique: $\{(g, k) \mid g$ has a clique of size $k$\} (same problem)
  - We will see another one next
- Example of a $W[2]$-hard problem:
  - Dominating set: $\{(g, k) \mid g$ has a *dominating set* of size $k$\}
    - Dominating set: each node is there or has a neighbor there
Parameterized CQ Evaluation

**Input:** Boolean CQ $Q$, instance $I$

**Parameter:** Size of $Q$

**Goal:** Compute $Q(I)$
W[1]-Hardness of Boolean CQ Evaluation

Recall our reduction from maximum clique to Boolean CQ evaluation

\[ G = Q \left( R_E(X_1, X_2), R_E(X_1, X_3), R_E(X_2, X_3) \right) \]
Recall our reduction from maximum clique to Boolean CQ evaluation

\[ Q_3() \vdash R_E(X_1, X_2), R_E(X_1, X_3), R_E(X_2, X_3) \]

\[ I_G = \begin{array}{c|ccc}
1 & 3 \\
2 & 3 \\
2 & 4 \\
3 & 4 \\
\end{array} \]
In the reduction, the size of the CQ was determined only by $k$.
In the reduction, the size of the CQ was determined only by $k$.

In formal terms, our reduction is a so called *FTP reduction*.
In the reduction, the size of the CQ was determined only by $k$.

In formal terms, our reduction is a so called \textit{FTP reduction}.

Hence, Boolean CQ evaluation is W[1]-hard when the size of the CQ is the parameter.
In the reduction, the size of the CQ was determined only by $k$.

In formal terms, our reduction is a so-called FTP reduction.

Hence, Boolean CQ evaluation is W[1]-hard when the size of the CQ is the parameter.

Hence, no hope for FPT without further assumptions; the query necessarily determines the degree of the polynomial data complexity.
Table of Contents

1 Introduction

2 Data and Combined Complexity

3 Parameterized Complexity

4 Input-Output Complexity
In this section we adopt the *combined complexity*, hence nothing is fixed.
In this section we adopt the *combined complexity*, hence nothing is fixed.

Some queries evaluate to a super-polynomial (e.g., exponential) number of tuples in the worst case.

Hence, no evaluation in polynomial time...
In this section we adopt the *combined complexity*, hence nothing is fixed.

Some queries evaluate to a super-polynomial (e.g., exponential) number of tuples in the worst case.

Hence, no evaluation in polynomial time... *But:*
In this section we adopt the *combined complexity*, hence nothing is fixed.

Some queries evaluate to a super-polynomial (e.g., exponential) number of tuples in the worst case.

Hence, no evaluation in polynomial time... *But*:
- What if on some instance there are just a few tuples?
In this section we adopt the *combined complexity*, hence nothing is fixed.

Some queries evaluate to a super-polynomial (e.g., exponential) number of tuples in the worst case.

Hence, no evaluation in polynomial time... *But:*

- What if on some instance there are just a few tuples?
- Is high complexity only due to #tuples?
In this section we adopt the *combined complexity*, hence nothing is fixed.

Some queries evaluate to a super-polynomial (e.g., exponential) number of tuples in the worst case.

Hence, no evaluation in polynomial time. . . *But:*

- What if on some instance there are just a few tuples?
- Is high complexity only due to \#tuples?
- What about incremental evaluation (produce as much as we have time for)?
In this section we adopt the *combined complexity*, hence nothing is fixed.

Some queries evaluate to a super-polynomial (e.g., exponential) number of tuples in the worst case.

Hence, no evaluation in polynomial time... *But:*

- What if on some instance there are just a few tuples?
- Is high complexity only due to #tuples?
- What about incremental evaluation (produce as much as we have time for)?

*Input-output complexity* measures the time as a function of both the input and the output.
In this section we adopt the *combined complexity*, hence nothing is fixed.

Some queries evaluate to a super-polynomial (e.g., exponential) number of tuples in the worst case.

Hence, no evaluation in polynomial time... *But:*

- What if on some instance there are just a few tuples?
- Is high complexity only due to #tuples?
- What about incremental evaluation (produce as much as we have time for)?

*Input-output complexity* measures the time as a function of both the input and the output.

Next, we make it more formal.
Notation

If \( S \) is a (possibly infinite) set, then we denote by \( \mathcal{P}_{\text{fin}}(S) \) the set of all finite subsets of \( S \).
An *enumeration problem* $E$ has an *input space* $\text{In}(E)$, an *output space* $\text{Out}(E)$, and it maps every input $x \in \text{In}(E)$ into a finite subset $E(x)$ of $\text{Out}(E)$

$$E : \text{In}(E) \to \mathcal{P}_{\text{fin}}(\text{Out}(E))$$
An enumeration problem $E$ has an input space $\text{In}(E)$, an output space $\text{Out}(E)$, and it maps every input $x \in \text{In}(E)$ into a finite subset $E(x)$ of $\text{Out}(E)$

$$E : \text{In}(E) \rightarrow \mathcal{P}_{\text{fin}}(\text{Out}(E))$$

Examples:
- $\text{In}(E)$: pairs (query, instance); $\text{Out}(E)$: tuples of values
- $\text{In}(E)$: graphs; $\text{Out}(E)$: node sets
An enumeration problem $E$ has an input space $\text{In}(E)$, an output space $\text{Out}(E)$, and it maps every input $x \in \text{In}(E)$ into a finite subset $E(x)$ of $\text{Out}(E)$

$$E : \text{In}(E) \rightarrow \mathcal{P}_{\text{fin}}(\text{Out}(E))$$

Examples:
- $\text{In}(E)$: pairs (query,instance); $\text{Out}(E)$: tuples of values
- $\text{In}(E)$: graphs; $\text{Out}(E)$: node sets

Computational task for $E$: Given $x \in \text{In}(E)$, compute (or enumerate) the items of $E(x)$
Let $E$ be an enumeration problem

A *solver* for $E$ is an algorithm $A$ that, given $x \in \text{In}(E)$, produces (or *prints*) a sequence of elements in $\text{Out}(E)$ during its execution, and has the following properties:

- **Soundness**: every produced answer is in $E(x)$
- **Completeness**: every answer in $E(X)$ is produced
- **Nonrepeating**: no answer is produced more than once
Johnson, Papadimitriou and Yannakakis [JPY88] introduced several different notions of efficiency for enumeration algorithms.

- Let $E$ be an enumeration problem, and let $A$ be solver for $E$.
- **Polynomial total time**: the total execution time of $A$ is polynomial in $|x| + |E(x)|$.
- **Polynomial delay**: the time between every two executive outputs is polynomial in $|x|$.
- **Incremental polynomial time**: after producing $N$ elements, the time to produce the next element is polynomial in $|x| + N$. 
Implications among Measures

Polynomial delay

\[ \Downarrow \]

Incremental polynomial time

\[ \Downarrow \]

Polynomial total time
Example: Path CQ

- We now look at an example of an algorithm that enumerates in polynomial total time.
Example: Path CQ

- We now look at an example of an algorithm that enumerates in polynomial total time
- Problem: evaluate a CQ of the following form over $R/2$:

\[
Q_n(x_1, \ldots, x_n) :- R(x_1, x_2), R(x_2, x_3), \ldots, R(x_{n-1}, x_n)
\]
Example: Path CQ

- We now look at an example of an algorithm that enumerates in polynomial total time.
- Problem: evaluate a CQ of the following form over $R/2$:

$Q_n(x_1, \ldots, x_n) \leftarrow R(x_1, x_2), R(x_2, x_3), \ldots, R(x_{n-1}, x_n)$

- That is, compute all length-$n$ paths of a given directed graph.
  - The directed graph is represented by an instance $I$ over $R$.
  - Not necessarily *simple* paths.
First Attempt

1 \( A_2 := I; \)
2 \[ \text{for } i = 3, \ldots, n \text{ do} \]
3 \[ \hspace{1em} /* \text{Join previous with } I */ \]
4 \[ A_i := \{(a_1, \ldots, a_i) \mid (a_1, \ldots, a_{i-1}) \in A_{i-1}, (a_{i-1}, a_i) \in I\}; \]
5 \[ /* \text{Print the output */} \]
6 \[ \text{forall } t \in A_n \text{ do} \]
7 \[ \text{print } t; \]

Given: \( Q_n, I \) \hspace{1em} Compute: \( Q_n(I) \)

\( Q_n(x_1, \ldots, x_n) := R(x_1, x_2), R(x_2, x_3), \ldots, R(x_{n-1}, x_n) \)
First Attempt

1. $A_2 := I$;
2. for $i = 3, \ldots, n$ do
   /* Join previous with $I$ */
   $A_i := \{(a_1, \ldots, a_i) \mid (a_1, \ldots, a_{i-1}) \in A_{i-1}, (a_{i-1}, a_i) \in I\}$;
   /* Print the output */
3. forall $t \in A_n$ do
4. print $t$;

Given: $Q_n, I$  Compute: $Q_n(I)$

$Q_n(x_1, \ldots, x_n) := R(x_1, x_2), R(x_2, x_3), \ldots, R(x_{n-1}, x_n)$

Is the algorithm correct (sound, complete, nonrepeating)?
First Attempt

Given: $Q_n, I$  
Compute: $Q_n(I)$

$$Q_n(x_1, \ldots, x_n) \leftarrow R(x_1, x_2), R(x_2, x_3), \ldots, R(x_{n-1}, x_n)$$

Is the algorithm correct (sound, complete, nonrepeating)?

Does the algorithm guarantee polynomial total time?
Example of a Problematic Case

\[ n = 7 \]
Revised Algorithm

1  \( I_n := I \);
2  \textbf{for} i = n - 1, \ldots, 2 \textbf{ do} \\
3  \quad I_i := \{ (a, b) \in I \mid \exists c [(b, c) \in I_{i+1}] \} ; \quad /* \text{semijoin} */ \\
4  \quad /* \text{Now join, as in the previous (slow) algorithm} */ \\
5  \quad \textbf{for} i = 3, \ldots, n \textbf{ do} \\
6  \quad \quad /* \text{Join previous with} I_i */ \\
7  \quad \quad A_i := \{ (a_1, \ldots, a_i) \mid (a_1, \ldots, a_{i-1}) \in A_{i-1}, (a_{i-1}, a_i) \in I_i \} ; \\
8  \quad \quad /* \text{Print the output} */ \\
9  \quad \textbf{forall} t \in A_n \textbf{ do} \\
10 \quad \quad \text{print} t ;

Given: \( Q_n, I \); Compute: \( Q_n(I) \)
Revised Algorithm

1. $I_n := I$;
2. for $i = n-1, \ldots, 2$ do
3.     $I_i := \{(a, b) \in I | \exists c[(b, c) \in I_{i+1}]\}$; /* semijoin */
4.     /* Now join, as in the previous (slow) algorithm */
4. for $i = 3, \ldots, n$ do
5.     /* Join previous with $I_i$ */
5.     $A_i := \{(a_1, \ldots, a_i) | (a_1, \ldots, a_{i-1}) \in A_{i-1}, (a_{i-1}, a_i) \in I_i\}$;
6.     /* Print the output */
6. for all $t \in A_n$ do
7.     print $t$;

Given: $Q_n, I$; Compute: $Q_n(I)$

Why is it correct? Is it polynomial time? Polynomial total time?
We have seen an algorithm for computing all the paths of a given length $n$ in polynomial total time.
We have seen an algorithm for computing all the paths of a given length $n$ in polynomial total time.

What about all *simple* paths of length $n$?
We have seen an algorithm for computing all the paths of a given length $n$ in polynomial total time.

What about all *simple* paths of length $n$?

Problem: Deciding whether a graph $g$ has a simple path of length $n$, given $g$ and $n$, is NP-complete.

- Generalizes the *Hamiltonian path* problem.
We have seen an algorithm for computing all the paths of a given length $n$ in polynomial total time. What about all *simple* paths of length $n$?

Problem: Deciding whether a graph $g$ has a simple path of length $n$, given $g$ and $n$, is NP-complete.

- Generalizes the *Hamiltonian path* problem

Assuming $P \neq NP$, can there be an enumeration algorithm for all simple paths, of a given length, that runs in:
We have seen an algorithm for computing all the paths of a given length $n$ in polynomial total time.

What about all *simple* paths of length $n$?

Problem: Deciding whether a graph $g$ has a simple path of length $n$, given $g$ and $n$, is NP-complete.

- Generalizes the *Hamiltonian path* problem.

Assuming $P \neq NP$, can there be an enumeration algorithm for all simple paths, of a given length, that runs in:
  - Polynomial delay?
We have seen an algorithm for computing all the paths of a given length $n$ in polynomial total time.

What about all *simple* paths of length $n$?

Problem: Deciding whether a graph $g$ has a simple path of length $n$, given $g$ and $n$, is NP-complete.

- Generalizes the *Hamiltonian path* problem

Assuming $P \neq NP$, can there be an enumeration algorithm for all simple paths, of a given length, that runs in:
  - Polynomial delay?
  - Polynomial total time?
The Emptiness Problem

- Let $E$ be an enumeration problem
Let $E$ be an enumeration problem

The *emptiness problem* for $E$ is the following:

Given $x \in \text{In}(E)$, is $E(x)$ empty?
The Emptiness Problem

- Let $E$ be an enumeration problem
- The *emptiness problem* for $E$ is the following:
  
  Given $x \in \text{In}(E)$, is $E(x)$ empty?

- We say that $E$ has *tractable verification* if:
  1. Deciding whether $x \in \text{In}(E)$, given $x$, is in polynomial time
  2. Every $y \in E(x)$ is of length polynomial in that of $x$
  3. Deciding whether $y \in E(x)$, given $x$ and $y$, is in polynomial time
Let \( E \) be an enumeration problem

- The *emptiness problem* for \( E \) is the following:
  
  Given \( x \in \text{In}(E) \), is \( E(x) \) empty?

- We say that \( E \) has *tractable verification* if:
  1. Deciding whether \( x \in \text{In}(E) \), given \( x \), is in polynomial time
  2. Every \( y \in E(x) \) is of length polynomial in that of \( x \)
  3. Deciding whether \( y \in E(x) \), given \( x \) and \( y \), is in polynomial time

- If \( E \) has tractable verification, then the emptiness problem of \( E \) is in coNP
The Emptiness Problem

- Let $E$ be an enumeration problem
- The *emptiness problem* for $E$ is the following:
  
  Given $x \in \text{In}(E)$, is $E(x)$ empty?
- We say that $E$ has *tractable verification* if:
  1. Deciding whether $x \in \text{In}(E)$, given $x$, is in polynomial time
  2. Every $y \in E(x)$ is of length polynomial in that of $x$
  3. Deciding whether $y \in E(x)$, given $x$ and $y$, is in polynomial time
- If $E$ has tractable verification, then the emptiness problem of $E$ is in coNP  *Why?*
PROPOSITION

Let $E$ be an enumeration problem with tractable verification, and assume that $P \neq NP$. If the emptiness problem of $E$ is coNP-complete, then $E$ cannot be solved in polynomial total time.
**Proposition**

Let $E$ be an enumeration problem with tractable verification, and assume that $P \neq NP$. If the emptiness problem of $E$ is coNP-complete, then $E$ cannot be solved in polynomial total time.

Proof: discussion + home assignment
Next, we will see an interesting example of a polynomial-delay algorithm.
Example of Polynomial Delay

- Next, we will see an interesting example of a polynomial-delay algorithm.
- Let $g$ be an undirected graph.
- Recall: a *clique* of $g$ is a set $C$ of nodes of $g$ such that every two nodes in $C$ are connected by an edge.
Next, we will see an interesting example of a polynomial-delay algorithm.

Let $g$ be an undirected graph.

Recall: a clique of $g$ is a set $C$ of nodes of $g$ such that every two nodes in $C$ are connected by an edge.

A clique $C$ is maximal if there is no clique $C'$ such that $C \subsetneq C'$.

Do not mix with a maximum clique that has a maximal number of nodes among all cliques.
Next, we will see an interesting example of a polynomial-delay algorithm

Let $g$ be an undirected graph

Recall: a *clique* of $g$ is a set $C$ of nodes of $g$ such that every two nodes in $C$ are connected by an edge

A clique $C$ is *maximal* if there is no clique $C'$ such that $C \subsetneq C'$

- Do not mix with a *maximum clique* that has a maximal number of nodes among all cliques

Next, we will see a polynomial-delay algorithm for enumerating *all maximal cliques* of a graph
Discussion on Enumerating Maximal Cliques

- What is the complexity of the emptiness problem?
Discussion on Enumerating Maximal Cliques

- What is the complexity of the emptiness problem?
- How would you generate one maximal clique?
Discussion on Enumerating Maximal Cliques

- What is the complexity of the emptiness problem?
- How would you generate one maximal clique?
- How would you generate two maximal cliques?
Discussion on Enumerating Maximal Cliques

- What is the complexity of the emptiness problem?
- How would you generate *one* maximal clique?
- How would you generate *two* maximal cliques?
- How would you generate *three* maximal cliques?
Discussion on Enumerating Maximal Cliques

- What is the complexity of the emptiness problem?
- How would you generate one maximal clique?
- How would you generate two maximal cliques?
- How would you generate three maximal cliques?
- How would you generate \( n \) maximal cliques for a given \( n \)?
Generating a Single Max Clique

1. $C := \emptyset$;
2. \textbf{forall} nodes $v$ of $g$ \textbf{do}
3. \hspace{1em} \textbf{if} $v$ is connected to every node in $C$ \textbf{then}
4. \hspace{2em} $C := C \cup \{v\}$;
5. \textbf{return} $C$

Given: $g$ \hspace{1em} Compute: a maximal clique
Generating a Single Max Clique

1. \( C := \emptyset; \)
2. \( \text{forall nodes } v \text{ of } g \text{ do} \)
3. \( \text{if } v \text{ is connected to every node in } C \text{ then} \)
4. \( C := C \cup \{v\}; \)
5. \( \text{return } C \)

**Given:** \( g \)  \hspace{1cm} **Compute:** a maximal clique

*Why is the returned \( C \) a clique? Why maximal?*
Maximizing a Clique

MaximizeClique\( (g, B) \)

1 \( C := B; \)
2 \( \text{forall nodes } v \text{ of } g \text{ do} \)
3 \( \quad \text{if } v \text{ is connected to every node in } C \text{ then} \)
4 \( \quad \quad C := C \cup \{v\}; \)
5 \( \text{return } C \)

Given: \( g, \) clique \( B \)  
Compute: a max. clique \( C \) such that \( B \subseteq C \)
Enumerating the Maximal Cliques [CFK^+06]

Given: \( g \) \hspace{1cm} \text{Compute: all maximal cliques}

1. \( C := \text{MaximizeClique}(g, \emptyset) \); 
2. \( Q := \{C\} \); \hspace{1cm} /* Assume log-time ops */
3. \( \mathcal{O} := \emptyset \); \hspace{1cm} /* Printed answers, assume log-time ops */
Enumerating the Maximal Cliques [CFK+06]

Given: $g$  Compute: all maximal cliques

1. $C := \text{MaximizeClique}(g, \varnothing)$;
2. $Q := \{C\}$;     /* Assume log-time ops */
3. $O := \varnothing$; /* Printed answers, assume log-time ops */
4. while $Q \neq \varnothing$ do
5.   $C := Q$.remove();
6.   print $C$;        /* Enumeration op */
7.   $O$.insert($C$);  /* $O(\log|O|)$ */
8.   /* Previous slide */
9.   $C' := \text{MaximizeClique}(g, B)$;
10. if $C' \notin Q \cup O$;  /* $O(\log Q + \log O)$ */
11. then $Q$.insert($C'$); /* $O(\log Q)$ */
Enumerating the Maximal Cliques [CFK+06]

Given: $g$  
Compute: all maximal cliques

1. $C := \text{MaximizeClique}(g, \emptyset)$;  
2. $Q := \{C\}$;  
   /* Assume log-time ops */  
3. $\mathcal{O} := \emptyset$;  
   /* Printed answers, assume log-time ops */  
4. while $Q \neq \emptyset$ do  
   5. $C := Q.\text{remove}()$;  
   6. print $C$;  
      /* Enumeration op */  
   7. $\mathcal{O}.\text{insert}(C)$;  
      /* $O(\log |\mathcal{O}|)$ */  
8. forall nodes $v$ of $g$ do  
   9. $B := \{v\} \cup \{u \in C \mid u \text{ is connected to } v\}$;  
10. $C' := \text{MaximizeClique}(g, B)$;  
    /* Previous slide */
Enumerating the Maximal Cliques \([\text{CFK}^+06]\)

**Given:** \(g\)  
**Compute:** all maximal cliques

1. \(C := \text{MaximizeClique}(g, \emptyset)\);  
2. \(Q := \{C\} \);  
   \hspace{1cm} /* Assume log-time ops */  
3. \(O := \emptyset \);  
   \hspace{1cm} /* Printed answers, assume log-time ops */  
4. **while** \(Q \neq \emptyset\) **do**  
5. \(C := Q\).remove();  
6. **print** \(C\);  
   \hspace{1cm} /* Enumeration op */  
7. \(O\).insert\((C)\);  
   \hspace{1cm} /* \(O(\log|O|)\) */  
8. **forall** nodes \(v\) **of** \(g\) **do**  
9. \(B := \{v\} \cup \{u \in C \mid u\ \text{is connected to} \ v\}\);  
10. \(C' := \text{MaximizeClique}(g, B)\);  
   \hspace{1cm} /* Previous slide */  
11. **if** \(C' \notin Q \cup O\);  
   \hspace{1cm} /* \(O(\log|Q| + \log|O|)\) */  
12. **then**  
13. \(Q\).insert\((C')\);  
   \hspace{1cm} /* \(O(\log|Q|)\) */
Correctness and Efficiency

- Why is the algorithm *sound* (printing only maximal cliques)?
- Why is the algorithm *nonrepeating*?
- Why is the algorithm running with *polynomial delay*?
- Why is the algorithm *complete*?
Proof of Completeness

- Suppose, by way of contradiction, that some maximal clique $D$ is not printed.
Proof of Completeness

- Suppose, by way of contradiction, that some maximal clique $D$ is not printed.
- Let $D'$ be a maximal subset of $D$ that is printed as part of some maximal clique, say $C$. 
Proof of Completeness

- Suppose, by way of contradiction, that some maximal clique $D$ is not printed
- Let $D'$ be a **maximal subset** of $D$ that is printed as part of some maximal clique, say $C$
- Let $v$ be a node in $D \setminus D'$
  - Why does $v$ exist?
Proof of Completeness

- Suppose, by way of contradiction, that some maximal clique $D$ is not printed
- Let $D'$ be a maximal subset of $D$ that is printed as part of some maximal clique, say $C$
- Let $v$ be a node in $D \setminus D'$
  - Why does $v$ exist?
- Consider the iteration where $C$ and $v$ are selected
Proof of Completeness

- Suppose, by way of contradiction, that some maximal clique \( D \) is not printed
- Let \( D' \) be a maximal subset of \( D \) that is printed as part of some maximal clique, say \( C \)
- Let \( v \) be a node in \( D \setminus D' \)
  - Why does \( v \) exist?
- Consider the iteration where \( C \) and \( v \) are selected
- In that iteration, \( B \) contains \( D' \cup \{v\} \)
Proof of Completeness

- Suppose, by way of contradiction, that some maximal clique $D$ is not printed
- Let $D'$ be a **maximal subset** of $D$ that is printed as part of some maximal clique, say $C$
- Let $v$ be a node in $D \setminus D'$
  - *Why does $v$ exist?*
- Consider the iteration where $C$ and $v$ are selected
- In that iteration, $B$ contains $D' \cup \{v\}$
- \ldots and $C'$ contains $B$, hence $D' \cup \{v\}$
Proof of Completeness

- Suppose, by way of contradiction, that some maximal clique $D$ is not printed.
- Let $D'$ be a maximal subset of $D$ that is printed as part of some maximal clique, say $C$.
- Let $v$ be a node in $D \setminus D'$.
  - *Why does $v$ exist?*
- Consider the iteration where $C$ and $v$ are selected.
- In that iteration, $B$ contains $D' \cup \{v\}$.
- ... and $C'$ contains $B$, hence $D' \cup \{v\}$.
- ... and $C'$ is printed at some point.
Proof of Completeness

- Suppose, by way of contradiction, that some maximal clique \( D \) is not printed.
- Let \( D' \) be a maximal subset of \( D \) that is printed as part of some maximal clique, say \( C \).
- Let \( v \) be a node in \( D \setminus D' \).
  - Why does \( v \) exist?
- Consider the iteration where \( C \) and \( v \) are selected.
- In that iteration, \( B \) contains \( D' \cup \{v\} \).
- \( \ldots \) and \( C' \) contains \( B \), hence \( D' \cup \{v\} \).
- \( \ldots \) and \( C' \) is printed at some point.
- Hence, a contradiction to our choice of \( D' \).
References I


End of lecture 3

Querying Complexity