Review: Query Complexity

Query answering as decision problem
〜 consider Boolean queries

Various notions of complexity:
- Combined complexity (complexity w.r.t. size of query and database instance)
- Data complexity (worst case complexity for any fixed query)
- Query complexity (worst case complexity for any fixed database instance)

Various common complexity classes:

\[ L \subseteq NL \subseteq P \subseteq NP \subseteq PSpace \subseteq \text{ExpTime} \]
**Theorem 4.1** The evaluation of FO queries is PSpace-complete with respect to combined complexity.

We have actually shown something stronger:

**Theorem 4.2** The evaluation of FO queries is PSpace-complete with respect to query complexity.

This also holds true when restricting to domain-independent queries.
The algorithm showed that FO query evaluation is in L

\[ L \subseteq N L \subseteq P \ldots \]

\[ \Rightarrow \text{can we do any better?} \]
The algorithm showed that FO query evaluation is in $L$

～ can we do any better?

**What could be better than $L$?**

$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \ldots$
The algorithm showed that FO query evaluation is in $L$

$\leadsto$ can we do any better?

**What could be better than $L$?**

$? \subseteq L \subseteq NL \subseteq P \subseteq \ldots$

$\leadsto$ we need to define circuit complexities first
**Definition 5.1:** A Boolean circuit is a finite, directed, acyclic graph where

- each node that has no predecessors is an **input node**
- each node that is not an input node is one of the following types of **logical gate**: AND, OR, NOT
- one or more nodes are designated **output nodes**

→ we will only consider Boolean circuits with exactly one output

→ propositional logic formulae are Boolean circuits with one output and gates of fanout $\leq 1$
Example

A Boolean circuit over an input string $x_1 x_2 \ldots x_n$ of length $n$
Example

A Boolean circuit over an input string $x_1x_2 \ldots x_n$ of length $n$

Corresponds to formula $(x_1 \land x_2) \lor (x_1 \land x_3) \lor \ldots \lor (x_{n-1} \land x_n)$

$\sim$ accepts all strings with at least two 1s
Circuits as a Model for Parallel Computation

Previous example:

\[ x_1 x_2 x_3 x_4 x_5 \ldots x_n \]

\[ \text{(} n^2 \text{ gates) } \]

\[ \sim n^2 \text{ processors working in parallel} \]

\[ \sim \text{ computation finishes in 2 steps} \]

- **size**: number of gates = total number of computing steps
- **depth**: longest path of gates = time for parallel computation

\[ \sim \text{ circuits as a refinement of polynomial time that takes parallelizability into account} \]

Markus Krötzsch, 8th May 2018

Database Theory

slide 7 of 20
Observation: the input size is “hard-wired” in circuits
~ each circuit only has a finite number of different inputs
~ not a computationally interesting problem

How can we solve interesting problems with Boolean circuits?
Observation: the input size is “hard-wired” in circuits

⇒ each circuit only has a finite number of different inputs

⇒ not a computationally interesting problem

How can we solve interesting problems with Boolean circuits?

Definition 5.2: A uniform family of Boolean circuits is a set of circuits $C_n \ (n \geq 0)$ that can easily be computed from $n$.

A language $L \subseteq \{0, 1\}^*$ is decided by a uniform family $(C_n)_{n \geq 0}$ of Boolean circuits if for each word $w$ of length $|w|$: $$ w \in L \text{ if and only if } C_{|w|}(w) = 1 $$

\(^a\)We don’t discuss the details here; see course Complexity Theory.
How to measure the computing power of Boolean circuits?

**Relevant metrics:**

- **size** of the circuit: overall number of gates
  (as function of input size)
- **depth** of the circuit: longest path of gates
  (as function of input size)
- **fan in:** two inputs per gate or any number of inputs per gate?
Measuring Complexity with Boolean Circuits

How to measure the computing power of Boolean circuits?

**Relevant metrics:**

- **size** of the circuit: overall number of gates
  (as function of input size)
- **depth** of the circuit: longest path of gates
  (as function of input size)
- **fan in**: two inputs per gate or any number of inputs per gate?

Important classes of circuits: **small-depth circuits**

**Definition 5.3:** \((C_n)_{n \geq 0}\) is a family of small-depth circuits if

- the size of \(C_n\) is polynomial in \(n\),
- the depth of \(C_n\) is poly-logarithmic in \(n\), that is, \(O(\log^k n)\).
The Complexity Classes NC and AC

Two important types of small-depth circuits:

**Definition 5.4:** $\text{NC}^k$ is the class of problems that can be solved by uniform families of circuits $(C_n)_{n \geq 0}$ of fan-in $\leq 2$, size polynomial in $n$, and depth in $O(\log^k n)$.

The class NC is defined as $\text{NC} = \bigcup_{k \geq 0} \text{NC}^k$.

(“Nick’s Class” named after Nicholas Pippenger by Stephen Cook)

**Definition 5.5:** $\text{AC}^k$ and AC are defined like $\text{NC}^k$ and NC, respectively, but for circuits with arbitrary fan-in.

(A is for “Alternating”: AND-OR gates alternate in such circuits)
family of polynomial size, constant depth, arbitrary fan-in circuits $\sim$ in $\text{AC}^0$
Example

family of polynomial size, constant depth, arbitrary fan-in circuits \( \leadsto \) in AC\(^0\)

We can eliminate arbitrary fan-ins by using more layers of gates:

family of polynomial size, logarithmic depth, bounded fan-in circuits \( \leadsto \) in NC\(^1\)

Markus Krötzsch, 8th May 2018
The previous sketch can be generalised:

$$NC^0 \subseteq AC^0 \subseteq NC^1 \subseteq AC^1 \subseteq \ldots \subseteq AC^k \subseteq NC^{k+1} \subseteq \ldots$$

Only few inclusions are known to be proper: $NC^0 \subset AC^0 \subset NC^1$
Relationships of Circuit Complexity Classes

The previous sketch can be generalised:

\[ NC^0 \subseteq AC^0 \subseteq NC^1 \subseteq AC^1 \subseteq \ldots \subseteq AC^k \subseteq NC^{k+1} \subseteq \ldots \]

Only few inclusions are known to be proper: \( NC^0 \subset AC^0 \subset NC^1 \)

Direct consequence of above hierarchy: \( NC = AC \)

Interesting relations to other classes:

\[ NC^0 \subset AC^0 \subset NC^1 \subseteq L \subseteq NL \subseteq AC^1 \subseteq \ldots \subseteq NC \subseteq P \]

Intuition:

- Problems in NC are parallelisable (known from definition)
- Problems in \( P \setminus NC \) are inherently sequential (educated guess)

However: it is not known if \( NC \neq P \)
Theorem 5.6: The evaluation of FO queries is complete for (logtime uniform) $AC^0$ with respect to data complexity.

Proof:

- **Membership:** For a fixed Boolean FO query, provide a uniform construction for a small-depth circuit based on the size of a database.
- **Hardness:** Show that circuits can be transformed into Boolean FO queries in logarithmic time (not on a standard TM . . . not in this lecture).
From Query to Circuit

Assumptions:

- query and database schema is fixed
- database instance (and thus active domain) are variable

Construct circuit uniformly based on size of active domain
From Query to Circuit

Assumptions:

• query and database schema is fixed
• database instance (and thus active domain) are variable

Construct circuit uniformly based on size of active domain

Sketch of construction:

• one input node for each possible database tuple (over given schema and active domain) 
  ~ true or false depending on whether tuple is present or not
• Recursively, for each subformula, introduce a gate for each possible tuple (instantiation) of this formula
  ~ true or false depending on whether the subformula holds for this tuple or not
• Logical operators correspond to gate types: basic operators obvious, ∀ as generalised conjunction, ∃ as generalised disjunction
• subformula with \( n \) free variables ~ \(|\text{adom}|^{n}\) gates
  ~ especially: \(|\text{adom}|^{0} = 1\) output gate for Boolean query
We consider the formula

$$\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$$

Over the database instance:

<table>
<thead>
<tr>
<th>R:</th>
<th>S:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a</td>
<td>b b</td>
</tr>
<tr>
<td>a b</td>
<td>b c</td>
</tr>
</tbody>
</table>

Active domain: \{a, b, c\}
Example: $\exists z.(\exists x.\exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$

<table>
<thead>
<tr>
<th></th>
<th>$R(a, a)$</th>
<th>$R(a, b)$</th>
<th>$R(a, c)$</th>
<th>...</th>
<th>$S(a, a)$</th>
<th>...</th>
<th>$S(b, a)$</th>
<th>$S(b, b)$</th>
<th>$S(b, c)$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: \( \exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z) \)
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: $\exists z.(\exists x.\exists y.\: R(x, y) \land S(y, z)) \land \neg R(a, z)$
Example: \( \exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z) \)
Example: $\exists z. (\exists x. \exists y. R(x, y) \land S(y, z)) \land \neg R(a, z)$
Summary and Outlook

The evaluation of FO queries is

- PSpace-complete for combined complexity
- PSpace-complete for query complexity
- AC⁰-complete for data complexity

Circuit complexities help to identify highly parallelisable problems in P

Open questions:

- Are there query languages with lower complexities? (next lecture)
- Which other computing problems are interesting?
- How can we study the expressiveness of query languages?