TECHNISCHE UNIVERSITÄT DRESDEN

DATABASE THEORY

Lecture 3: Complexity of Query Answering

Markus Krötzsch
Knowledge-Based Systems

TU Dresden, 16th Apr 2019

Review: The Relational Calculus

What we have learned so far:

- There are many ways to describe databases:
  $\leadsto$ named perspective, unnamed perspective, interpretations, ground fracts, (hyper)graphs
- There are many ways to describe query languages:
  $\leadsto$ relational algebra, domain independent FO queries, safe-range FO queries, actice domain FO queries, Codd's tuple calculus
  $\leadsto$ either under named or under unnamed perspetive

All of these are largely equivalent: The Relational Calculus

Next question: How hard is it to answer such queries?

How to Measure Complexity of Queries?

- Complexity classes often for decision problems (yes/no answer)
  $\leadsto$ database queries return many results (no decision problem)
- The size of a query result can be very large
  $\leadsto$ it would not be fair to measure this as "complexity"
- In practice, database instances are much larger than queries
  $\leadsto$ can we take this into account?

Query Answering as Decision Problem

We consider the following decision problems:

- Boolean query entailment: given a Boolean query q and a database instance $\mathcal{I}$, does $\mathcal{I} \models q$ hold?
- Query of tuple problem: given an n-ary query q, a database instance $\mathcal{I}$ and a tuple $\langle c_1, \ldots, c_n \rangle$, does $\langle c_1, \ldots, c_n \rangle \in M[q](\mathcal{I})$ hold?
- Query emptiness problem: given a query q and a database instance $\mathcal{I}$, does $M[q](\mathcal{I}) \neq \emptyset$ hold?

$\leadsto$ Computationally equivalent problems (exercise)

The Size of the Input

Combined Complexity
Input: Boolean query q and database instance $\mathcal{I}$
Output: Does $\mathcal{I} \models q$ hold?

$\rightsquigarrow$ estimates complexity in terms of overall input size
$\rightsquigarrow$ "2KB query/2TB database" = "2TB query/2KB database"
$\rightsquigarrow$ study worst-case complexity of algorithms for fixed queries:

Data Complexity
Input: database instance $\mathcal{I}$
Output: Does $\mathcal{I} \models q$ hold? (for fixed q)

$\rightsquigarrow$ we can also fix the database and vary the query:

Query Complexity
Input: Boolean query q
Output: Does $\mathcal{I} \models q$ hold? (for fixed $\mathcal{I}$)

Review: Computation and Complexity Theory

The Turing Machine (1)

Computation is usually modelled with Turing Machines (TMs)
$\rightsquigarrow$ "algorithm" = "something implemented on a TM"

A TM is an automaton with (unlimited) working memory:

- It has a finite set of states Q
- Q includes a start state q_{start} and an accept state q_{acc}
- The memory is a tape with numbered cells $0, 1, 2, \ldots$
- Each tape cell holds one symbol from the set of tape symbols Γ
- There is a special symbol ␣ for empty tape cells
- The TM has a transition relation $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{l, r, s\})$
- Δ might be a partial function $(Q \times \Gamma) \to (Q \times \Gamma \times \{l, r, s\})$
  $\rightsquigarrow$ deterministic TM (DTM); otherwise nondeterministic TM

There are many different but equivalent ways of defining TMs.

The Turing Machine (2)

TMs operate step-by-step:

- At every moment, the TM is in one state $q \in Q$ with its read/write head at a certain tape position $p \in \mathbb{N}$, and the tape has a certain contents $\sigma_0\sigma_1\sigma_2\cdots$ with all $\sigma_i \in \Gamma$
  $\rightsquigarrow$ current configuration of the TM
- The TM starts in state q_{start} and at tape position 0.
- Transition $\langle q, \sigma, q', \sigma', d\rangle \in \Delta$ means:
  if in state q and the tape symbol at its current position is σ,
  then change to state q', write symbol σ' to tape, move head by d (left/right/stay)
- If there is more than one possible transition, the TM picks one nondeterministically
- The TM halts when there is no possible transition for the current configuration (possibly never)

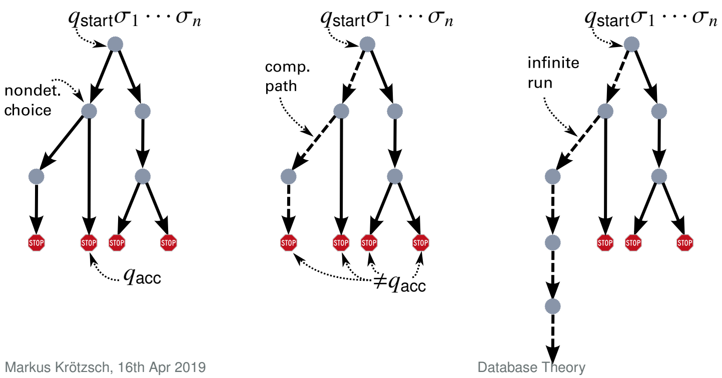
A computation path (or run) of a TM is a sequence of configurations that can be obtained by some choice of transition.

Languages Accepted by TMs

The (nondeterministic) TM accepts an input $\sigma_1 \cdots \sigma_n \in (\Gamma \setminus \{␣\})^*$ if, when started on the tape $\sigma_1 \cdots \sigma_n ␣␣ \cdots$,

(1) the TM halts on every computation path and
(2) there is at least one computation path that halts in the accepting state $q_{\text{acc}} \in Q$.

accept: reject: reject (not halting):



Solving Computation Problems with TMs

A decision problem is a language $\mathcal{L}$ of words over $\Sigma = \Gamma \setminus \{␣\}$
$\rightsquigarrow$ the set of all inputs for which the answer is "yes"

A TM decides a decision problem $\mathcal{L}$ if it halts on all inputs and accepts exactly the words in $\mathcal{L}$

TMs take time (number of steps) and space (number of cells):

- Time($f(n)$): Problems that can be decided by a DTM in $O(f(n))$ steps, where f is a function of the input length n
- Space($f(n)$): Problems that can be decided by a DTM using $O(f(n))$ tape cells, where f is a function of the input length n
- NTime($f(n)$): Problems that can be decided by a TM in at most $O(f(n))$ steps **on any of its computation paths**
- NSpace($f(n)$): Problems that can be decided by a TM using at most $O(f(n))$ tape cells **on any of its computation paths**

Some Common Complexity Classes

$$\text{P} = \text{PTime} = \bigcup_{k\geq 1} \text{Time}(n^k) \qquad \text{NP} = \bigcup_{k\geq 1} \text{NTime}(n^k)$$

$$\text{Exp} = \text{ExpTime} = \bigcup_{k\geq 1} \text{Time}(2^{n^k}) \qquad \text{NExp} = \text{NExpTime} = \bigcup_{k\geq 1} \text{NTime}(2^{n^k})$$

$$\text{2Exp} = \text{2ExpTime} = \bigcup_{k\geq 1} \text{Time}(2^{2^{n^k}}) \qquad \text{N2Exp} = \text{N2ExpTime} = \bigcup_{k\geq 1} \text{NTime}(2^{2^{n^k}})$$

$$\text{ETime} = \bigcup_{k\geq 1} \text{Time}(2^{nk})$$

$$\text{L} = \text{LogSpace} = \text{Space}(\log n) \qquad \text{NL} = \text{NLogSpace} = \text{NSpace}(\log n)$$

$$\text{PSpace} = \bigcup_{k\geq 1} \text{Space}(n^k)$$

$$\text{ExpSpace} = \bigcup_{k\geq 1} \text{Space}(2^{n^k})$$

NP

NP = Problems for which a possible solution can be verified in P:

- for every $w \in \mathcal{L}$, there is a certificate $c_w \in \Sigma^*$, such that
- the length of c_w is polynomial in the length of w, and
- the language $\{w\#\#c_w \mid w \in \mathcal{L}\}$ is in P

Equivalent to definition with nondeterministic TMs:

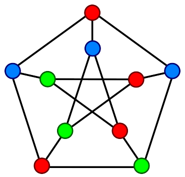
- $\Rightarrow$ nondeterministically guess certificate; then run verifier DTM
- $\Leftarrow$ use accepting polynomial run as certificate; verify TM steps

NP Examples

Examples:

- Sudoku solvability (certificate: filled-out grid)
- Composite (non-prime) number (certificate: factorization)
- Prime number (certificate: see Wikipedia “Primality certificate”)
- Propositional logic satisfiability (certificate: satisfying assignment)
- Graph colourability (certificate: coloured graph)

5		3				7		
			8					6
	7			6			4	
	4		1					
7		8		5		3		9
					9		6	
	5			1			7	
6					4			
		2				5		3



p	q	r	$p \to q$
f	f	f	w
f	w	f	w
w	f	f	f
w	w	f	w
f	f	w	w
f	w	w	w
w	f	w	f
w	w	w	w

NP and coNP

Note: Definition of NP is not symmetric

- there does not seem to be any polynomial certificate for Sudoku **un**solvability or logic **un**satisfiability
- converse of an NP problem is coNP
- similar for NExpTime and N2ExpTime

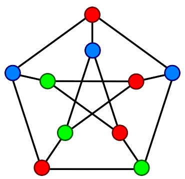
Other classes are symmetric:

- Deterministic classes (coP = P etc.)
- Space classes mentioned above (esp. coNL = NL)

Reductions

Observation: some problems can be reduced to others

Example: 3-colouring can be reduced to propositional satisfiability



Encoding colours in propositions:

- r_i means "‘vertex i is red"’
- g_i means "‘vertex i is green"’
- b_i means "‘vertex i is blue"’

Colouring conditions on vertices: $(\mathsf{r}_1 \wedge \neg \mathsf{g}_1 \wedge \neg \mathsf{b}_1) \vee (\neg \mathsf{r}_1 \wedge \mathsf{g}_1 \wedge \neg \mathsf{b}_1) \vee (\neg \mathsf{r}_1 \wedge \neg \mathsf{g}_1 \wedge \mathsf{b}_1)$
(and so on for all vertices)

Colouring conditions for edges:
$\neg(\mathsf{r}_1 \wedge \mathsf{r}_2) \wedge \neg(\mathsf{g}_1 \wedge \mathsf{g}_2) \wedge \neg(\mathsf{b}_1 \wedge \mathsf{b}_2)$ (and so on for all edges)

Satisfying truth assignment ⇔ valid colouring

Defining Reductions

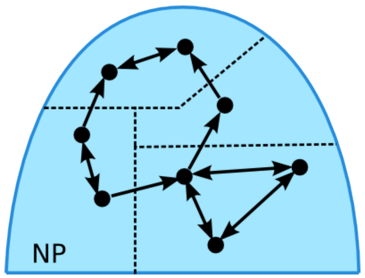
Definition 3.1: Consider languages $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Sigma^*$. A computable function $f : \Sigma^* \to \Sigma^*$ is a many-one reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$ if:

$$w \in \mathcal{L}_1 \quad \text{if and only if} \quad f(w) \in \mathcal{L}_2$$

$\rightsquigarrow$ we can solve problem $\mathcal{L}_1$ by reducing it to problem $\mathcal{L}_2$
$\rightsquigarrow$ only useful if the reduction is much easier than solving $\mathcal{L}_1$ directly
$\rightsquigarrow$ polynomial many-one reductions

The Structure of NP

Idea: polynomial many-one reductions define an order on problems



NP-Hardness und NP-Completeness

Theorem 3.2 (Cook 1971; Levin 1973): All problems in NP can be polynomially many-one reduced to the propositional satisfiability problem (SAT).

Stephen Cook

- NP has a maximal class that contains a practically relevant problem
- If SAT can be solved in P, all problems in NP can
- Karp discovered 21 further such problems shortly after (1972)
- Thousands such problems have been discovered since . . .

Leonid Levin

Definition 3.3: A language is
- NP-hard if every language in NP is polynomially many-one reducible to it
- NP-complete if it is NP-hard and in NP

Richard Karp

Comparing Complexity Classes

Is any NP-complete problem in P?
- If yes, then P = NP
- Nobody knows $\rightsquigarrow$ biggest open problem in computer science
- Similar situations for many complexity classes

Some things that are known:

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSpace} \subseteq \text{ExpTime} \subseteq \text{NExpTime}$$

- None of these is known to be strict
- But we know that P $\subsetneq$ ExpTime and NL $\subsetneq$ PSpace
- Moreover PSpace = NPSpace (by Savitch's Theorem)

(see TU Dresden course complexity theory for many more details)

Comparing Tractable Problems

Polynomial-time many-one reductions work well for (presumably) super-polynomial problems $\rightsquigarrow$ what to use for P and below?

Definition 3.4: A LogSpace transducer is a deterministic TM with three tapes:
- a read-only input tape
- a read/write working tape of size $O(\log n)$
- a write-only, write-once output tape

Such a TM needs a slightly different form of transitions:
- transition function input: state, input tape symbol, working tape symbol
- transition function output: state, working tape write symbol, input tape move, working tape move, output tape symbol or ␣ to not write anything to the output

The Power of LogSpace

LogSpace transducers can still do a few things:

- store a constant number of counters and increment/decrement the counters
- store a constant number of pointers to the input tape, and locate/read items that start at this address from the input tape
- access/process/compare items from the input tape bit by bit

Example 3.5: Adding and subtracting binary numbers, detecting palindromes, comparing lists, searching items in a list, sorting lists, ... can all be done in L.

Joining Two Tables in LogSpace

Input: two relations R and S, represented as a list of tuples

- Use two pointers p_R and p_S pointing to tuples in R and S, respectively
- Outer loop: iterate p_R over all tuples of R
- Inner loop for each position of p_R: iterate p_S over all tuples of S
- For each combination of p_R and p_S, compare the tuples:
  - Use another two loops that iterate over the columns of R and S
  - Compare attribute names bit by bit
  - For matching attribute names, compare the respective tuple values bit by bit
- If all joined columns agree, copy the relevant parts of tuples p_R and p_S to the output (bit by bit)

Output: $R \bowtie S$

$\leadsto$ Fixed number of pointers and counters
(making this fully formal is still a bit of work; e.g., an additional counter is needed to move the input read head to the target of a pointer (seek))

LogSpace reductions

LogSpace functions: The output of a LogSpace transducer is the contents of its output tape when it halts $\leadsto$ a partial function $\Sigma^* \to \Sigma^*$

Note: the composition of two LogSpace functions is LogSpace (exercise)

Definition 3.6: A many-one reduction f from $\mathcal{L}_1$ to $\mathcal{L}_2$ is a LogSpace reduction if it is implemented by some LogSpace transducer.

$\leadsto$ can be used to define hardness for classes P and NL

From L to NL

NL: Problems whose solution can be verified in L

Example: Reachability

- Input: a directed graph G and two nodes s and t of G
- Output: accept if there is a directed path from s to t in G

Algorithm sketch:

- Store the id of the current node and a counter for the path length
- Start with s as current node
- In each step, increment the counter and move from the current node to one of its direct successors (nondeterministic)
- When reaching t, accept
- When the step counter is larger than the total number of nodes, reject

Beyond Logarithmic Space

Propositional satisfiability can be solved in linear space:
$\leadsto$ iterate over possible truth assignments and check each in turn

More generally: all problems in NP can be solved in PSpace
$\leadsto$ try all conceivable polynomial certificates and verify each in turn

What is a "typical" (that is, hard) problem in PSpace?
$\leadsto$ Simple two-player games, and other uses of alternating quantifiers

Example: Playing "Geography"

A children's game:

- Two players are taking turns naming cities.
- Each city must start with the last letter of the previous.
- Repetitions are not allowed.
- The first player who cannot name a new city looses.

A mathematicians' game:

- Two players are marking nodes on a directed graph.
- Each node must be a successor of the previous one.
- Repetitions are not allowed.
- The first player who cannot mark a new node looses.

Question: given a certain graph and start node, can Player 1 enforce a win (i.e., does he have a winning strategy)?

$\leadsto$ PSpace-complete problem

Example: Quantified Boolean Formulae (QBF)

We consider formulae of the following form:

$$\mathsf{Q}_1 X_1.\mathsf{Q}_2 X_2.\cdots \mathsf{Q}_n X_n.\varphi[X_1, \ldots, X_n]$$

where $\mathsf{Q}_i \in \{\exists, \forall\}$ are quantifiers, X_i are propositional logic variables, and φ is a propositional logic formula with variables $X_1, \ldots, X_n$ and constants $\top$ (true) and $\bot$ (false)

Semantics:

- Propositional formulae without variables (only constants $\top$ and $\bot$) are evaluated as usual
- $\exists X_1.\varphi[X_1]$ is true if either $\varphi[X_1/\top]$ or $\varphi[X_1/\bot]$ are
- $\forall X_1.\varphi[X_1]$ is true if both $\varphi[X_1/\top]$ and $\varphi[X_1/\bot]$ are

Question: Is a given QBF formula true?

$\leadsto$ PSpace-complete problem

A Note on Space and Time

How many different configurations does a TM have in space $(f(n))$?

$$|Q| \cdot f(n) \cdot |\Gamma|^{f(n)}$$

$\leadsto$ No halting run can be longer than this
$\leadsto$ A time-bounded TM can explore all configurations in time proportional to this

Applications:

- $\mathsf{L} \subseteq \mathsf{P}$
- PSpace $\subseteq$ ExpTime

Summary and Outlook

The complexity of query languages can be measured in different ways

Relevant complexity classes are based on restricting space and time:

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSpace} \subseteq \mathrm{ExpTime}$$

Problems are compared using many-one reductions

$\rightsquigarrow$ see TU Dresden course **Complexity Theory** for further details and deeper insights

Open questions:

- Now how hard is it to answer FO queries? (next lecture)
- We saw that joins are in LogSpace – is this tight?
- How can we study the expressiveness of query languages?