TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

KNOWLEDGE GRAPHS

Lecture 7: Expressive Power and Complexity of SPARQL

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 23th Nov 2021

Review

Semantics of each feature is defined by specific algebra operators

- $\text{Join}(M_1, M_2)$: join compatible mappings from M_1 and M_2
- $\text{Filter}_G(\varphi, M)$: remove from multiset M all mappings for which φ does not evaluate to EBV "true"
- $\text{Union}(M_1, M_2)$: compute the union of mappings from multisets M_1 and M_2
- $\text{Minus}(M_1, M_2)$: remove from multiset M_1 all mappings compatible with a non-empty mapping in M_2
- $\text{LeftJoin}_G(M_1, M_2, \varphi)$: extend mappings from M_1 by compatible mappings from M_2 when filter condition is satisfied; keep remaining mappings from M_1 unchanged
- $\text{Extend}(M, v, \varphi)$: extend all mappings from M by assigning v the value of φ.
- $\text{OrderBy}(L, \text{condition})$: sort list by a condition
- $\text{Slice}(L, \text{start}, \text{length})$: apply limit and offset modifiers

Further operators exist, e.g., $\text{Distinct}(L)$.

Translating SPARQL to nested algebra expressions is mostly straightforward (we saw an algorithm for a subset of features).

Complexity of SPARQL

Finding BGP solutions

How can we compute solutions to BGPs?

Finding BGP solutions

How can we compute solutions to BGPs?

Possible approach:

1. Find solutions to triple patterns
2. Compute joins of partial solutions

By Theorem 6.6, $\text{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

(Blank nodes might need to be replaced by variables that are projected away later.)

Finding BGP solutions

How can we compute solutions to BGPs?

Possible approach:

1. Find solutions to triple patterns
2. Compute joins of partial solutions

By Theorem 6.6, $\text{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.
(Blank nodes might need to be replaced by variables that are projected away later.)

How hard is this? (on a graph with n edges)

Finding BGP solutions

How can we compute solutions to BGPs?

Possible approach:

1. Find solutions to triple patterns
2. Compute joins of partial solutions

By Theorem 6.6, $\text{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.

(Blank nodes might need to be replaced by variables that are projected away later.)

How hard is this? (on a graph with n edges)

1. Can be solved by iterating over all edges: $O(n)$ (linear)

Finding BGP solutions

How can we compute solutions to BGPs?

Possible approach:

1. Find solutions to triple patterns
2. Compute joins of partial solutions

By Theorem 6.6, $\text{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.
(Blank nodes might need to be replaced by variables that are projected away later.)

How hard is this? (on a graph with n edges)

1. Can be solved by iterating over all edges: $O(n)$ (linear)
2. We defined
   $\text{Join}(\Omega_1, \Omega_2) = \{\mu_1 \uplus \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$, and μ_1 and μ_2 are compatible$\}$.

Finding BGP solutions

How can we compute solutions to BGPs?

Possible approach:

1. Find solutions to triple patterns
2. Compute joins of partial solutions

By Theorem 6.6, $\text{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.
(Blank nodes might need to be replaced by variables that are projected away later.)

How hard is this? (on a graph with n edges)

1. Can be solved by iterating over all edges: $O(n)$ (linear)
2. We defined
   $\text{Join}(\Omega_1, \Omega_2) = \{\mu_1 \uplus \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$, and μ_1 and μ_2 are compatible$\}$.
   Therefore $\text{Join}(\Omega_1, \Omega_2)$ is of size $O(|\Omega_1| \times |\Omega_2|) \in O(n^2)$ (quadratic)

Finding BGP solutions

How can we compute solutions to BGPs?

Possible approach:

1. Find solutions to triple patterns
2. Compute joins of partial solutions

By Theorem 6.6, $\text{BGP}_G(P)$ is the join of the solution multisets of all individual triple patterns in P.
(Blank nodes might need to be replaced by variables that are projected away later.)

How hard is this? (on a graph with n edges)

1. Can be solved by iterating over all edges: $O(n)$ (linear)
2. We defined
   $\text{Join}(\Omega_1, \Omega_2) = \{\mu_1 \uplus \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$, and μ_1 and μ_2 are compatible$\}$.
   Therefore $\text{Join}(\Omega_1, \Omega_2)$ is of size $O(|\Omega_1| \times |\Omega_2|) \in O(n^2)$ (quadratic)
   But joining results of k triple patterns is in $O(n^k)$ (exponential)!

$\leadsto$ worst-case exponential-time query answering algorithm

Review: Computational complexity

Computational complexity provides tools for estimating

- how hard a problem is
- based on the effort an algorithm needs to solve it

To classify algorithms, we distinguish:

- computational models: deterministic, non-deterministic, probabilistic, quantum, ...
- constrained resources: time (steps), space (memory), ...
- resource bounds: polynomial, exponential, ... (measured wrt. to input size)

Such complexity classifications are rather robust measures of a problem's "difficulty" and do not depend on implementation details.

Review: Some complexity classes

deterministic		non-deterministic
$\mathsf{P} = \mathsf{PTime} = \bigcup_{d\geq 1} \mathsf{DTime}(n^d)$	polynomial time	$\mathsf{NP} = \bigcup_{k\geq 1} \mathsf{NTime}(n^k)$
$\mathsf{Exp} = \mathsf{ExpTime} = \bigcup_{d\geq 1} \mathsf{DTime}(2^{n^d})$	exponential time	$\mathsf{NExp} = \mathsf{NExpTime} = \bigcup_{k\geq 1} \mathsf{NTime}(2^{n^k})$
$\mathsf{L} = \mathsf{LogSpace} = \mathsf{DSpace}(\log n)$	logarithmic space	$\mathsf{NL} = \mathsf{NLogSpace} = \mathsf{NSpace}(\log n)$
$\mathsf{PSpace} = \bigcup_{d\geq 1} \mathsf{DSpace}(n^d)$	polynomial space	

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

$\neq$ $\neq$ $\neq$ $\neq$

Review: The class NP

NP is an extremely common class for challenging problems in practice.
It can be defined in two ways:

Nondeterministic polynomial time

- Problems in NP can be solved by a non-deterministic algorithm
- In time bounded by a polynomial

Polynomial verification

- All problems in NP have polynomial “solutions”: short certificates that prove all “yes” answers
- The correctness of such certificates can be verified in polynomial time

NP problems are search problems – searching for a right solution among the exponentially many potential solutions – but even the best known algorithms may take exponential time.

Finding BGP solutions

Observation: It is easy to check if a given mapping of bnodes and variables produces a solution:

- Simply verify that the mapped triples are contained in the given graph
- Can be done in quadratic time (# triples in pattern $\times$ # edges in graph)

Finding BGP solutions

Observation: It is easy to check if a given mapping of bnodes and variables produces a solution:

- Simply verify that the mapped triples are contained in the given graph
- Can be done in quadratic time (# triples in pattern $\times$ # edges in graph)

In other words: the problem (as a decision problem) is in NP.

Finding BGP solutions

Observation: It is easy to check if a given mapping of bnodes and variables produces a solution:

- Simply verify that the mapped triples are contained in the given graph
- Can be done in quadratic time (# triples in pattern $\times$ # edges in graph)

In other words: the problem (as a decision problem) is in NP.

It turns out this is the best we can do:

Theorem 7.1: Determining if a BGP has solution mappings over a graph is NP-complete (with respect to the size of the pattern).

Finding BGP solutions

Observation: It is easy to check if a given mapping of bnodes and variables produces a solution:

- Simply verify that the mapped triples are contained in the given graph
- Can be done in quadratic time (# triples in pattern $\times$ # edges in graph)

In other words: the problem (as a decision problem) is in NP.

It turns out this is the best we can do:

Theorem 7.1: Determining if a BGP has solution mappings over a graph is NP-complete (with respect to the size of the pattern).

Proof:

- Inclusion: guess mapping for bnodes and variables; check if guess was correct.
- Hardness: by reduction from a known NP-hard problem

Review: Polynomial many-one reductions

To compare the hardness of problems, we ask which problems can be reduced to others.

Definition 7.2: A language $\mathbf{L_1} \subseteq \Sigma^*$ is polynomially many-one reducible to $\mathbf{L_2} \subseteq \Sigma^*$, denoted $\mathbf{L_1} \leq_p \mathbf{L_2}$, if there is a polynomial-time computable function f such that for all $w \in \Sigma^*$

$$w \in \mathbf{L_1} \qquad \text{if and only if} \qquad f(w) \in \mathbf{L_2}.$$

Review: Polynomial many-one reductions

To compare the hardness of problems, we ask which problems can be reduced to others.

Definition 7.2: A language $\mathbf{L_1} \subseteq \Sigma^*$ is polynomially many-one reducible to $\mathbf{L_2} \subseteq \Sigma^*$, denoted $\mathbf{L_1} \leq_p \mathbf{L_2}$, if there is a polynomial-time computable function f such that for all $w \in \Sigma^*$

$$w \in \mathbf{L_1} \qquad \text{if and only if} \qquad f(w) \in \mathbf{L_2}.$$

Intuition: If $\mathbf{L_1} \leq_p \mathbf{L_2}$, then:

- We can solve a problem of $\mathbf{L_1}$, by reducing it to a problem of $\mathbf{L_2}$
- Therefore $\mathbf{L_1}$ is "at most as difficult" as $\mathbf{L_2}$ (modulo polynomial effort)

Review: Polynomial many-one reductions

To compare the hardness of problems, we ask which problems can be reduced to others.

Definition 7.2: A language $\mathbf{L_1} \subseteq \Sigma^*$ is polynomially many-one reducible to $\mathbf{L_2} \subseteq \Sigma^*$, denoted $\mathbf{L_1} \leq_p \mathbf{L_2}$, if there is a polynomial-time computable function f such that for all $w \in \Sigma^*$

$$w \in \mathbf{L_1} \qquad \text{if and only if} \qquad f(w) \in \mathbf{L_2}.$$

Intuition: If $\mathbf{L_1} \leq_p \mathbf{L_2}$, then:

- We can solve a problem of $\mathbf{L_1}$, by reducing it to a problem of $\mathbf{L_2}$
- Therefore $\mathbf{L_1}$ is "at most as difficult" as $\mathbf{L_2}$ (modulo polynomial effort)

Definition 7.3: A problem $\mathbf{C}$ is NP-complete if $\mathbf{C} \in$ NP and, for every problem $\mathbf{L} \in$ NP, we find $\mathbf{L} \leq_p \mathbf{C}$.

Intuition: NP-complete problems are the "hardest" problems in NP since they hold the key to solving all other problems in NP. For a more refined understanding, see course "Complexity Theory".

From 3-colourability to BGP matching

The problem of graph 3-colourability (**3Col**) is defined as follows:
Given: An undirected graph G
Question: Can the vertices of G be assigned colours red, green and blue so that no two adjacent vertices have the same colour?

It is known that this problem is NP-complete (and in particular NP-hard).

From 3-colourability to BGP matching

The problem of graph 3-colourability (**3Col**) is defined as follows:

Given: An undirected graph G

Question: Can the vertices of G be assigned colours red, green and blue so that no two adjacent vertices have the same colour?

It is known that this problem is NP-complete (and in particular NP-hard).

We can find a polynomial many-one reduction from **3Col** to BGP matching:

- A given graph G is mapped to a BGP P_G by introducing, for each undirected edge `e−f` in G, two triples `?e <edge> ?f` and `?f <edge> ?e`.
- We consider the RDF graph C given by

```
<red> <edge> <green>, <blue> .
<green> <edge> <red>, <blue> .
<blue> <edge> <green>, <red> .
```

Then P_G has a solution mapping over C if and only if G is 3-colourable. □

NP-hardness another way

A typical NP-complete problem is satisfiability of propositional logic formulae:

The problem of propositional logic satisfiability (**SAT**) is defined as follows:
Given: An propositional logic formula φ
Question: Is it possible to assign truth values to propositional variables in φ such that the formula evaluates to true?

NP-hardness another way

A typical NP-complete problem is satisfiability of propositional logic formulae:

The problem of propositional logic satisfiability (**SAT**) is defined as follows:
Given: An propositional logic formula φ
Question: Is it possible to assign truth values to propositional variables in φ such that the formula evaluates to true?

Exercise: Give a direct reduction from **SAT** to SPARQL query answering, without using BGPs.

This shows (in another way) that SPARQL query answering is NP-hard. However, it is actually harder than that.

Beyond NP

In complexity theory, space is usually more powerful than time
(intuition: space can be reused; time, alas, cannot)

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

Space restrictions can also be used for non-deterministic algorithms, but by Savitch's Theorem, this often does not give additional expressive power: PSpace = NPSpace

Completeness again is defined by polynomial reductions:

Definition 7.4: A problem **C** is PSpace-complete if **C** $\in$ PSpace and, for every problem **L** $\in$ PSpace, we find **L** $\leq_p$ **C**.

Quantified Boolean Formulae

A QBF is a formula of the following form:

$$\mathsf{Q}_1 X_1.\mathsf{Q}_2 X_2.\cdots\mathsf{Q}_\ell X_\ell.\varphi[X_1,\ldots,X_\ell]$$

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

Semantics:

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

(where $\varphi[X/\top]$ is "φ with X replaced by $\top$, and similar for $\bot$)

Hardness of QBF Evaluation

TrueQBF is the following problem:

Given: A quantified boolean formula φ

Question: Does φ evaluate to true?

Hardness of QBF Evaluation

TrueQBF is the following problem:
Given: A quantified boolean formula φ
Question: Does φ evaluate to true?

This is a rather difficult question:

Example 7.5: A propositional formula φ with propositions $p_1, \ldots, p_n$ is satisfiable if $\exists p_1 \ldots \exists p_n.\varphi$ is a true QBF, i.e., **SAT** reduces to **TrueQBF** (so it is NP-hard).

The QBF φ is a tautology if $\forall p_1 \ldots \forall p_n.\varphi$ is a true QBF, i.e., tautology checking reduces to **TrueQBF** (so it is coNP-hard).

Hardness of QBF Evaluation

TrueQBF is the following problem:
Given: A quantified boolean formula φ
Question: Does φ evaluate to true?

This is a rather difficult question:

Example 7.5: A propositional formula φ with propositions $p_1, \ldots, p_n$ is satisfiable if $\exists p_1 \ldots \exists p_n.\varphi$ is a true QBF, i.e., **SAT** reduces to **TrueQBF** (so it is NP-hard).

The QBF φ is a tautology if $\forall p_1 \ldots \forall p_n.\varphi$ is a true QBF, i.e., tautology checking reduces to **TrueQBF** (so it is coNP-hard).

In fact, it is known that **TrueQBF** is harder than both NP and coNP:

Theorem 7.6: **TrueQBF** is PSpace-complete.

(without proof; see course “Complexity Theory”)

Universal quantifiers in SPARQL

To show NP-hardness, we used the fact that SPARQL can naturally express existential quantifiers, since we always ask "does a match for this query exist"?

Can we also express universal quantifiers?

Universal quantifiers in SPARQL

To show NP-hardness, we used the fact that SPARQL can naturally express existential quantifiers, since we always ask "does a match for this query exist"?

Can we also express universal quantifiers? — Yes:

Example 7.7: In Wikidata, find bands all of whose (known) members are female.

```
SELECT ?band
WHERE {
 ?band wdt:P31 wd:Q215380 . # ?band instance of: band
 ?band wdt:P527 [] . # ?band has part: [] (at least one known member)
 FILTER NOT EXISTS {
   ?band wdt:P527 ?member . # ?band has part: ?member
   FILTER NOT EXISTS {
     ?member wdt:P21 wd:Q6581072 # ?member sex or gender: female
   }
 }
}
```

SPARQL is PSpace-hard

The PSpace-hardness of **TrueQBF** + the encoding universal quantifiers yield:

Theorem 7.8: Deciding whether a SPARQL query has any results is PSpace-hard, even over an empty RDF graph.

Proof: We reduce QBF formulae to SPARQL queries.

SPARQL is PSpace-hard

The PSpace-hardness of **TrueQBF** + the encoding universal quantifiers yield:

Theorem 7.8: Deciding whether a SPARQL query has any results is PSpace-hard, even over an empty RDF graph.

Proof: We reduce QBF formulae to SPARQL queries. A QBF $\mathsf{Q}_1 X_1.\mathsf{Q}_2 X_2.\cdots\mathsf{Q}_\ell X_\ell.\varphi[X_1,\ldots,X_\ell]$ is transformed to SPARQL in the following steps:

SPARQL is PSpace-hard

The PSpace-hardness of **TrueQBF** + the encoding universal quantifiers yield:

Theorem 7.8: Deciding whether a SPARQL query has any results is PSpace-hard, even over an empty RDF graph.

Proof: We reduce QBF formulae to SPARQL queries. A QBF $\mathsf{Q}_1 X_1.\mathsf{Q}_2 X_2.\cdots\mathsf{Q}_\ell X_\ell.\varphi[X_1,\ldots,X_\ell]$ is transformed to SPARQL in the following steps:

1. Replace every sub-formula of the form $\forall X_i.\psi$ by $\neg\exists X_i.\neg\psi$.

SPARQL is PSpace-hard

The PSpace-hardness of **TrueQBF** + the encoding universal quantifiers yield:

Theorem 7.8: Deciding whether a SPARQL query has any results is PSpace-hard, even over an empty RDF graph.

Proof: We reduce QBF formulae to SPARQL queries. A QBF $\mathsf{Q}_1 X_1.\mathsf{Q}_2 X_2.\cdots\mathsf{Q}_\ell X_\ell.\varphi[X_1,\ldots,X_\ell]$ is transformed to SPARQL in the following steps:

1. Replace every sub-formula of the form $\forall X_i.\psi$ by $\neg\exists X_i.\neg\psi$.
2. Replace the innermost boolean formula (φ or $\neg\varphi$) by an expression `FILTER` $(\hat{\varphi})$ where $\hat{\varphi}$ is $(\neg)\varphi$ written using SPARQL Boolean functions $\&\&$, ||, and !, and with each propositional variable X_i replaced by a unique SPARQL variable `?Xi`.

SPARQL is PSpace-hard

The PSpace-hardness of **TrueQBF** + the encoding universal quantifiers yield:

Theorem 7.8: Deciding whether a SPARQL query has any results is PSpace-hard, even over an empty RDF graph.

Proof: We reduce QBF formulae to SPARQL queries. A QBF $\mathsf{Q}_1 X_1.\mathsf{Q}_2 X_2.\cdots\mathsf{Q}_\ell X_\ell.\varphi[X_1,\ldots,X_\ell]$ is transformed to SPARQL in the following steps:

1. Replace every sub-formula of the form $\forall X_i.\psi$ by $\neg\exists X_i.\neg\psi$.
2. Replace the innermost boolean formula (φ or $\neg\varphi$) by an expression `FILTER` $(\hat{\varphi})$ where $\hat{\varphi}$ is $(\neg)\varphi$ written using SPARQL Boolean functions $\&\&$, ||, and !, and with each propositional variable X_i replaced by a unique SPARQL variable `?Xi`.
3. Replace every sub-expression of the form $\neg\exists X_i.\psi$ with `FILTER NOT EXISTS { VALUES ?Xi {true false}` ψ `}`

SPARQL is PSpace-hard

The PSpace-hardness of **TrueQBF** + the encoding universal quantifiers yield:

Theorem 7.8: Deciding whether a SPARQL query has any results is PSpace-hard, even over an empty RDF graph.

Proof: We reduce QBF formulae to SPARQL queries. A QBF $\mathcal{Q}_1 X_1.\mathcal{Q}_2 X_2.\cdots\mathcal{Q}_\ell X_\ell.\varphi[X_1,\ldots,X_\ell]$ is transformed to SPARQL in the following steps:

1. Replace every sub-formula of the form $\forall X_i.\psi$ by $\neg\exists X_i.\neg\psi$.
2. Replace the innermost boolean formula (φ or $\neg\varphi$) by an expression **FILTER** $(\hat{\varphi})$ where $\hat{\varphi}$ is $(\neg)\varphi$ written using SPARQL Boolean functions $\&\&$, $||$, and !, and with each propositional variable X_i replaced by a unique SPARQL variable `?Xi`.
3. Replace every sub-expression of the form $\neg\exists X_i.\psi$ with **FILTER NOT EXISTS { VALUES** `?Xi {true false}` ψ **}**
4. Replace every sub-expression of the form $\exists.\psi$ with **FILTER EXISTS { VALUES** `?Xi {true false}` ψ **}**

SPARQL is PSpace-hard

The PSpace-hardness of **TrueQBF** + the encoding universal quantifiers yield:

Theorem 7.8: Deciding whether a SPARQL query has any results is PSpace-hard, even over an empty RDF graph.

Proof: We reduce QBF formulae to SPARQL queries. A QBF $\mathsf{Q}_1 X_1.\mathsf{Q}_2 X_2.\cdots\mathsf{Q}_\ell X_\ell.\varphi[X_1,\ldots,X_\ell]$ is transformed to SPARQL in the following steps:

1. Replace every sub-formula of the form $\forall X_i.\psi$ by $\neg\exists X_i.\neg\psi$.
2. Replace the innermost boolean formula (φ or $\neg\varphi$) by an expression `FILTER (`$\hat{\varphi}$`)` where $\hat{\varphi}$ is $(\neg)\varphi$ written using SPARQL Boolean functions $\&\&$, ||, and !, and with each propositional variable X_i replaced by a unique SPARQL variable `?Xi`.
3. Replace every sub-expression of the form $\neg\exists X_i.\psi$ with
   `FILTER NOT EXISTS { VALUES ?Xi {true false}` ψ `}`
4. Replace every sub-expression of the form $\exists.\psi$ with
   `FILTER EXISTS { VALUES ?Xi {true false}` ψ `}`

From the resulting SPARQL expression `P`, create the query:

```
SELECT * WHERE { VALUES ?x {"QBF is true!"} P }
```

SPARQL is PSpace-hard (2)

It is not hard to see that this transformation works as desired: the resulting query has a solution mapping $\{x \mapsto \texttt{"QBF is true!"}\}$ if and only if the QBF is true.

Example 7.9: Consider the QBF $\forall p.\exists q.((\neg p \wedge q) \vee (p \wedge \neg q))$. Eliminating $\forall$ yields $\neg\exists p.\neg\exists q.((\neg p \wedge q) \vee (p \wedge \neg q))$. We then obtain the following SPARQL query:

```
SELECT * WHERE {
 VALUES ?x {"QBF is true!"}
 FILTER NOT EXISTS { VALUES ?p {true false}
    FILTER NOT EXISTS { VALUES ?q {true false}
       FILTER ( (! ?p && ?q) || (?p && ! ?q) )
    }
 }
}
```

Is SPARQL practical?

PSpace-hard problems are highly intractable and hard to implement in practice.

Is SPARQL practically feasible at all?

Is SPARQL practical?

PSpace-hard problems are highly intractable and hard to implement in practice.

Is SPARQL practically feasible at all?

Apparently yes:

- We have seen implementations
- Other widely used query languages, such as SQL, have similar complexities

Is SPARQL practical?

PSpace-hard problems are highly intractable and hard to implement in practice.

Is SPARQL practically feasible at all?

Apparently yes:

- We have seen implementations
- Other widely used query languages, such as SQL, have similar complexities

Is complexity theory useless?

Is SPARQL practical?

PSpace-hard problems are highly intractable and hard to implement in practice.

Is SPARQL practically feasible at all?

Apparently yes:

- We have seen implementations
- Other widely used query languages, such as SQL, have similar complexities

Is complexity theory useless?

No, but we should measure more carefully:

- Our proofs (for NP and PSpace) turn hard problems into hard queries
- We hardly need RDF data at all

In practice, databases grow very big, while queries are rather limited!

(Wikidata has billions of triples; typical Wikidata query have less than 100 triple patterns [Malyshev et al., ISWC 2018])

More fine-grained complexity measures

Combined Complexity
Input: Query Q and RDF graph G
Output: Does Q have answers over G?

$\leadsto$ estimates complexity in terms of overall input size
$\leadsto$ "2KB query/2TB database" = "2TB query/2KB database"

More fine-grained complexity measures

Combined Complexity
Input: Query Q and RDF graph G
Output: Does Q have answers over G?

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

Data Complexity
Input: RDF graph G
Output: Does Q have answers over G? (for fixed query Q)

More fine-grained complexity measures

Combined Complexity
Input: Query Q and RDF graph G
Output: Does Q have answers over G?

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

Data Complexity
Input: RDF graph G
Output: Does Q have answers over G? (for fixed query Q)

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

Query Complexity
Input: SPARQL query Q
Output: Does Q have answers over G? (for fixed RDF graph G)

Below P

Our previous proofs show high query complexity (hence also high combined complexity). For data complexity, we get much lower complexities, starting below polynomial time.

Definition 7.10: The class NL of languages decidable in logarithmic space on a non-deterministic Turing machine is defined as $\mathrm{NL} = \mathrm{NSpace}(\log(n))$.

Note: When restricting Turing machines to use less than linear space, we need to provide them with a separate read-only input tape that is not counted (since the input of length n cannot fit into $\log(n)$ space itself).

Intuition: The memory of a logspace-bounded Turing machine (deterministic or not) is just enough for the following:

- Store a fixed number of binary counters (with at most polynomial value)
- Store a fixed number of pointers to positions in the input
- Compare the values of counters and target symbols of pointers

It is known that $\mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP}$ (and all inclusions are believed to be strict, though this remains unproven)

Data complexity of SPARQL

The problem of directed graph reachability (also known as s-t-reachability) is defined as follows:
Given: A directed graph G and two vertices s and t
Question: Is there a directed path from s to t?

This can be solved in NL:

- Starting from s, non-deterministically move to a successor vertex
- Terminate when moving to t (success) or after making more moves than vertices in the graph (failure)

This runs in logarithmic space: one pointer to current vertex, one counter

Data complexity of SPARQL

The problem of directed graph reachability (also known as s-t-reachability) is defined as follows:
Given: A directed graph G and two vertices s and t
Question: Is there a directed path from s to t?

This can be solved in NL:

- Starting from s, non-deterministically move to a successor vertex
- Terminate when moving to t (success) or after making more moves than vertices in the graph (failure)

This runs in logarithmic space: one pointer to current vertex, one counter

Directed graph reachability is furthermore known to be NL-hard, so we get:

Theorem 7.11: Deciding if a SPARQL query has any solutions is NL-hard in terms of data complexity.

Data complexity of SPARQL

The problem of directed graph reachability (also known as s-t-reachability) is defined as follows:

Given: A directed graph G and two vertices s and t

Question: Is there a directed path from s to t?

This can be solved in NL:

- Starting from s, non-deterministically move to a successor vertex
- Terminate when moving to t (success) or after making more moves than vertices in the graph (failure)

This runs in logarithmic space: one pointer to current vertex, one counter

Directed graph reachability is furthermore known to be NL-hard, so we get:

Theorem 7.11: Deciding if a SPARQL query has any solutions is NL-hard in terms of data complexity.

Proof: Directed graph reachability is easily reduced: encode graph in RDF, and use a single property path pattern with * to check reachability. □

Upper bounds

Important note: All of our results so far were lower bounds, showing that SPARQL is at least as hard as the given class. We have not shown that SPARQL queries can actually be answered in the given bounds.[1]

[1] We have not even shown that SPARQL query answers are computable at all. SQL query answers, e.g., are not, if all SQL features are allowed.

Upper bounds

Important note: All of our results so far were lower bounds, showing that SPARQL is at least as hard as the given class. We have not shown that SPARQL queries can actually be answered in the given bounds.[1]

How to obtain upper bounds?

- Give an algorithm
- Show that it can run within the required bounds (with respect to query size and/or data size)

[1] We have not even shown that SPARQL query answers are computable at all. SQL query answers, e.g., are not, if all SQL features are allowed.

Upper bounds

Important note: All of our results so far were lower bounds, showing that SPARQL is at least as hard as the given class. We have not shown that SPARQL queries can actually be answered in the given bounds.[1]

How to obtain upper bounds?

- Give an algorithm
- Show that it can run within the required bounds (with respect to query size and/or data size)

Problem: SPARQL has a large number of features that an algorithm would need to consider, making algorithms rather complex and harder to verify

$\leadsto$ sketch algorithms for basic cases only

[1] We have not even shown that SPARQL query answers are computable at all. SQL query answers, e.g., are not, if all SQL features are allowed.

Answering queries in PSpace

Note: A single query can have exponentially many solutions, so the result does not fit into polynomial space. But a polynomial space algorithm could still discover all solutions (and stream them to an output).

Answering queries in PSpace

Note: A single query can have exponentially many solutions, so the result does not fit into polynomial space. But a polynomial space algorithm could still discover all solutions (and stream them to an output).

Algorithm sketch:

- Iterate over all possible variable and bnode bindings, storing them one by one (possible in polynomial space)
- Verify query conditions for the given binding (possible in polynomial space for most features, e.g., triple patterns, property path patterns, filters, union, minus, . . .)

Answering queries in PSpace

Note: A single query can have exponentially many solutions, so the result does not fit into polynomial space. But a polynomial space algorithm could still discover all solutions (and stream them to an output).

Algorithm sketch:

- Iterate over all possible variable and bnode bindings, storing them one by one (possible in polynomial space)
- Verify query conditions for the given binding (possible in polynomial space for most features, e.g., triple patterns, property path patterns, filters, union, minus, ...)

Where this sketch is lacking:

- We should check complexity of all filter conditions and functions
- We did not clarify how to handle subqueries and aggregates
- Result values can become exponentially large (e.g., by repeated string doubling using **`BIND`**), so a smarter representation of values has to be used

Answering queries in NL for data

We can use the same approach for worst-case optimal query answering with respect to the size of the RDF graph (data complexity):

Algorithm sketch:

- Iterate over all possible variable and bnode bindings, storing one at a time
- Verify query conditions for the given binding

$\leadsto$ If the query is fixed, the bindings can be stored using a fixed number of pointers.

$\leadsto$ For most operations, it is again clear that they are possible to verify in NL

This includes many numeric aggregates and arithmetic operations.

Again, we omit many details here that would need careful discussion.

Note: In terms of the size of the data, values can not be exponentially but merely polynomially large, since the query is constant now; but one still needs to explain how to represent this.

Summary

SPARQL is PSpace-complete for query and combined complexity[1]

SPARQL is NL-complete for data complexity, hence practically tractable and well parallelisable[1]

What's next?

- The limits of SPARQL
- Querying graphs with rules
- Property graph and Cypher

[1] The matching upper bound has not been proven with the full set of features.