Background on Decidability and Complexity

Decidability (in logic)

Instead of defining the terms "decidable" and "undecidable" in a formal way, we give some examples.

The problem whether

$$\mathcal{T} \models C \sqsubseteq D$$

for $\mathcal{EL}$-TBoxes $\mathcal{T}$ and $\mathcal{EL}$-concepts C, D, is decidable because there exists an algorithm (e.g., the one given in the lectures) that terminates after finitely many steps for any $\mathcal{EL}$-TBox $\mathcal{T}$ and $\mathcal{EL}$-concepts C, D and

- outputs YES if $\mathcal{T} \models C \sqsubseteq D$;
- outputs NO if $\mathcal{T} \models C \sqsubseteq D$.

Decidability (in logic)

Instead of defining the terms "decidable" and "undecidable" in a formal way, we give some examples.

The problem whether

$$\mathcal{T} \models C \sqsubseteq D$$

for $\mathcal{EL}$-TBoxes $\mathcal{T}$ and $\mathcal{EL}$-concepts C, D, is decidable because there exists an algorithm (e.g., the one given in the lectures) that terminates after finitely many steps for any $\mathcal{EL}$-TBox $\mathcal{T}$ and $\mathcal{EL}$-concepts C, D and

- outputs YES if $\mathcal{T} \models C \sqsubseteq D$;
- outputs NO if $\mathcal{T} \models C \sqsubseteq D$.

For all the description logics discussed in this module (logics in the DL-Lite family, $\mathcal{ALC}$ and its extension by number restrictions, inverse, and nominals) the basic reasoning problems are decidable.

Decidability

One can argue that decidability of basic reasoning tasks is a **necessary** condition for an ontology language to be useful in practice. If basic query answering (e.g., deciding $\mathcal{T} \models C \sqsubseteq D$) is undecidable, then it is impossible to implement an algorithm that gives a correct answer to every possible query. Thus, it is not possible to implement software for which it is guaranteed that a user will always get a correct answer to a query.

Decidability

One can argue that decidability of basic reasoning tasks is a **necessary** condition for an ontology language to be useful in practice. If basic query answering (e.g., deciding $\mathcal{T} \models C \sqsubseteq D$) is undecidable, then it is impossible to implement an algorithm that gives a correct answer to every possible query. Thus, it is not possible to implement software for which it is guaranteed that a user will always get a correct answer to a query.

For **first-order predicate logic**, the basic reasoning problems are **undecidable**: there does not exist an algorithm that decides whether a first-order predicate logic sentence follows from a finite set of first-order predicate logic sentences.

This explains why unrestricted first-order predicate logic should not be used as an ontology language.

Complexity

Decidability alone does not guarantee that one can implement an algorithm that gives correct answers within a reasonable amount of time/space.

Complexity

Decidability alone does not guarantee that one can implement an algorithm that gives correct answers within a reasonable amount of time/space.

To analyse whether one can implement an algorithm that gives correct answers within a reasonable amount of time, one can investigate how the time/space required to solve an instance of the problem grows with the size of the instance.

Example: Propositional Logic

- A propositional formula is constructed from propositional variables $\boldsymbol{p_1, p_2, \ldots}$ using the connectives $\wedge$, $\vee$, and $\neg$.

Example: Propositional Logic

- A propositional formula is constructed from propositional variables $p_1, p_2, \ldots$ using the connectives $\wedge$, $\vee$, and $\neg$.
- A propositional formula P is **satisfiable** if there exists an assignment a of truth values $0, 1$ to the propositional variables in P such that $a(P) = 1$.

Example: Propositional Logic

- A propositional formula is constructed from propositional variables $p_1, p_2, \ldots$ using the connectives $\wedge$, $\vee$, and $\neg$.
- A propositional formula P is **satisfiable** if there exists an assignment a of truth values $0, 1$ to the propositional variables in P such that $a(P) = 1$.
- An (inefficient) way of checking satisfiability of P is to compute for all assignments a the value $a(P)$ and output "Yes" if $a(P) = 1$ for at least one a. Otherwise output "No".

Example: Propositional Logic

- A propositional formula is constructed from propositional variables $p_1, p_2, \ldots$ using the connectives $\wedge$, $\vee$, and $\neg$.
- A propositional formula P is **satisfiable** if there exists an assignment a of truth values $0, 1$ to the propositional variables in P such that $a(P) = 1$.
- An (inefficient) way of checking satisfiability of P is to compute for all assignments a the value $a(P)$ and output "Yes" if $a(P) = 1$ for at least one a. Otherwise output "No".
- In the worst case, the algorithm requires at least 2^n steps, where n is the number of variables in P.

Example: Propositional Logic

- A propositional formula is constructed from propositional variables $p_1, p_2, \ldots$ using the connectives $\wedge$, $\vee$, and $\neg$.
- A propositional formula P is **satisfiable** if there exists an assignment a of truth values $0, 1$ to the propositional variables in P such that $a(P) = 1$.
- An (inefficient) way of checking satisfiability of P is to compute for all assignments a the value $a(P)$ and output "Yes" if $a(P) = 1$ for at least one a. Otherwise output "No".
- In the worst case, the algorithm requires at least 2^n steps, where n is the number of variables in P.
- This is an exponential time algorithm.

Example: our $\mathcal{EL}$ subsumption algorithm

Let us assume the size of the input "$\mathcal{T} \models A \sqsubseteq B$?" is the length of the corresponding word. We assume $\mathcal{T}$ is in normal form. Let a single rule application (updating S or R) be a step of the algorithm.

Example: our $\mathcal{EL}$ subsumption algorithm

Let us assume the size of the input "$\mathcal{T} \models A \sqsubseteq B$?" is the length of the corresponding word. We assume $\mathcal{T}$ is in normal form. Let a single rule application (updating S or R) be a step of the algorithm.

- We apply the rule corresponding to an inclusion in $C \sqsubseteq D$ in $\mathcal{T}$ at most once to update $S(A)$ for a fixed A and as most once to update $R(r)$ for a fixed r.

Example: our $\mathcal{EL}$ subsumption algorithm

Let us assume the size of the input "$\mathcal{T} \models A \sqsubseteq B$?" is the length of the corresponding word. We assume $\mathcal{T}$ is in normal form. Let a single rule application (updating S or R) be a step of the algorithm.

- We apply the rule corresponding to an inclusion in $C \sqsubseteq D$ in $\mathcal{T}$ at most once to update $S(A)$ for a fixed A and as most once to update $R(r)$ for a fixed r.

- Let M_C be the number of concept names in $\mathcal{T}$, M_R be the number of role names in $\mathcal{T}$, and N be the number of inclusions in $\mathcal{T}$.

Example: our $\mathcal{EL}$ subsumption algorithm

Let us assume the size of the input "$\mathcal{T} \models A \sqsubseteq B$?" is the length of the corresponding word. We assume $\mathcal{T}$ is in normal form. Let a single rule application (updating S or R) be a step of the algorithm.

- We apply the rule corresponding to an inclusion in $C \sqsubseteq D$ in $\mathcal{T}$ at most once to update $S(A)$ for a fixed A and as most once to update $R(r)$ for a fixed r.
- Let M_C be the number of concept names in $\mathcal{T}$, M_R be the number of role names in $\mathcal{T}$, and N be the number of inclusions in $\mathcal{T}$.
- Then at most $(M_C + M_R) \times N$ rule applications are possible.

Example: our $\mathcal{EL}$ subsumption algorithm

Let us assume the size of the input "$\mathcal{T} \models A \sqsubseteq B$?" is the length of the corresponding word. We assume $\mathcal{T}$ is in normal form. Let a single rule application (updating S or R) be a step of the algorithm.

- We apply the rule corresponding to an inclusion in $C \sqsubseteq D$ in $\mathcal{T}$ at most once to update $S(A)$ for a fixed A and as most once to update $R(r)$ for a fixed r.

- Let M_C be the number of concept names in $\mathcal{T}$, M_R be the number of role names in $\mathcal{T}$, and N be the number of inclusions in $\mathcal{T}$.

- Then at most $(M_C + M_R) \times N$ rule applications are possible.

- Let n be the size of $\mathcal{T}$. We have $M_C + M_R \leq n$ and $N \leq n$.

Example: our $\mathcal{EL}$ subsumption algorithm

Let us assume the size of the input "$\mathcal{T} \models A \sqsubseteq B$?" is the length of the corresponding word. We assume $\mathcal{T}$ is in normal form. Let a single rule application (updating S or R) be a step of the algorithm.

- We apply the rule corresponding to an inclusion in $C \sqsubseteq D$ in $\mathcal{T}$ at most once to update $S(A)$ for a fixed A and as most once to update $R(r)$ for a fixed r.
- Let M_C be the number of concept names in $\mathcal{T}$, M_R be the number of role names in $\mathcal{T}$, and N be the number of inclusions in $\mathcal{T}$.
- Then at most $(M_C + M_R) \times N$ rule applications are possible.
- Let n be the size of $\mathcal{T}$. We have $M_C + M_R \leq n$ and $N \leq n$.
- Thus at most n^2 rule applications are possible.

Example: our $\mathcal{EL}$ subsumption algorithm

Let us assume the size of the input "$\mathcal{T} \models A \sqsubseteq B$?" is the length of the corresponding word. We assume $\mathcal{T}$ is in normal form. Let a single rule application (updating S or R) be a step of the algorithm.

- We apply the rule corresponding to an inclusion in $C \sqsubseteq D$ in $\mathcal{T}$ at most once to update $S(A)$ for a fixed A and as most once to update $R(r)$ for a fixed r.
- Let M_C be the number of concept names in $\mathcal{T}$, M_R be the number of role names in $\mathcal{T}$, and N be the number of inclusions in $\mathcal{T}$.
- Then at most $(M_C + M_R) \times N$ rule applications are possible.
- Let n be the size of $\mathcal{T}$. We have $M_C + M_R \leq n$ and $N \leq n$.
- Thus at most n^2 rule applications are possible.
- This is a polynomial time algorithm.

Tractable Problems

For most applications, a problem can be regarded as **tractable** if there exists an algorithm that solves the problem in polynomial time:

Tractable Problems

For most applications, a problem can be regarded as **tractable** if there exists an algorithm that solves the problem in polynomial time:

there exists a polynomial function $p(n)$ (e.g. the linear function n, the quadratic function n^2, the cubic function n^3) such that for any input of some size $n > 0$, the algorithm terminates with the correct answer after at most $p(n)$ steps (we use an intuitive notion of "step" of a computation here without making it precise.

Tractable Problems

For most applications, a problem can be regarded as **tractable** if there exists an algorithm that solves the problem in polynomial time:

there exists a polynomial function $p(n)$ (e.g. the linear function n, the quadratic function n^2, the cubic function n^3) such that for any input of some size $n > 0$, the algorithm terminates with the correct answer after at most $p(n)$ steps (we use an intuitive notion of "step" of a computation here without making it precise.

Typically, a problem can be regarded as non-tractable if there exists no such polynomial function. In practice, this means that the best algorithm solving the problem requires more than polynomial time in **some case**.

Classifying non-tractable Problems

Problems that are decidable but not in polytime can be further classified according to the time/space it takes to solve them. Important classes of problems are

- NP-complete problems
- ExpTime-complete problems
- and problems even harder than ExpTime.

For our purposes, it is enough to know that any algorithm solving such a problem requires exponential time (roughly 2^n) for some inputs of size n (“in the worst case”).

Classifying non-tractable Problems

Problems that are decidable but not in polytime can be further classified according to the time/space it takes to solve them. Important classes of problems are

- NP-complete problems
- ExpTime-complete problems
- and problems even harder than ExpTime.

For our purposes, it is enough to know that any algorithm solving such a problem requires exponential time (roughly 2^n) for some inputs of size n ("in the worst case").

We distinguish here between NP-complete and ExpTime: For NP-complete problems there is currently no proof that no polytime algorithm exists (P equals NP problem). For ExpTime-complete and harder problems we can actually prove that no polytime algorithm exists.