Symbolic Artificial Intelligence (COMP3008) Lecture 5: Complexity of Reasoning 27 October 2025

Daniel Karapetyan
daniel.karapetyan@nottingham.ac.uk

Decidability

Idealised reasoning systems

Idealised reasoning system:

- Given knowledge base S
- Given sentence α (the hypothesis)
- Need to determine if $S \models \alpha$
  - In other words, check if every interpretation $\mathfrak{I}$ that satisfies S also satisfies α

Recall that testing entailment can be reduced to testing satisfiability:

$$S \models \alpha \qquad \text{if and only if} \qquad S \wedge \neg\alpha \text{ is unsatisfiable}$$

Semi-decidability of FOL

Assume that we want to check if $S \models \alpha$

FOL is *semi-decidable*:

- If $S \models \alpha$ then it is possible to prove it in finite time
- If $S \not\models \alpha$ then there is no procedure that is guaranteed to ever prove it

Since FOL is semi-decidable, we cannot talk about its computational complexity

Real reasoning systems

- There exist decidable subsets of FOL
  - For example, propositional logic and CSP for finite domains
- There exist reasoning systems for FOL such that they either give correct answers or never terminate (or terminate with an error)
  - For example, Z3

Computational complexity of propositional logic

- Testing entailment or satisfiability in propositional logic can be reduced to SAT
- SAT is NP-complete
- NP-complete means that, for a satisfiable case, there is a polynomial-time proof of satisfiablity, however proving unsatisfiability is ‘hard’
- ‘Hard’ means that it is as hard as solving many other search problems, for which it is widely believed that they cannot be solved in polynomial time

Questions

Decidable or not: prove that $S \models \alpha$ for propositional sentences S and α?

Questions

Decidable or not: prove that $S \models \alpha$ for propositional sentences S and α? Decidable

Questions

Decidable or not: prove that $S \models \alpha$ for propositional sentences S and α? Decidable

Decidable or not: testing if $S \wedge \neg\gamma$ is satisfiable if S and γ are in *FOL*?

Questions

Decidable or not: prove that $S \models \alpha$ for propositional sentences S and α? Decidable

Decidable or not: testing if $S \wedge \neg\gamma$ is satisfiable if S and γ are in *FOL*? Undecidable

Questions

Decidable or not: prove that *S* $\models \alpha$ for propositional sentences *S* and α? Decidable

Decidable or not: testing if *S* $\wedge \neg\gamma$ is satisfiable if *S* and γ are in *FOL*? Undecidable

Computational complexity of testing that *S* $\models \alpha$ in the FOL case?

Questions

Decidable or not: prove that $S \models \alpha$ for propositional sentences S and α? Decidable

Decidable or not: testing if $S \wedge \neg\gamma$ is satisfiable if S and γ are in *FOL*? Undecidable

Computational complexity of testing that $S \models \alpha$ in the FOL case? Undefined (the problem is undecidable)

Computational complexity of testing that S is valid for any FOL sentence S?

Questions

Decidable or not: prove that $S \models \alpha$ for propositional sentences S and α? Decidable

Decidable or not: testing if $S \wedge \neg\gamma$ is satisfiable if S and γ are in *FOL*? Undecidable

Computational complexity of testing that $S \models \alpha$ in the FOL case? Undefined (the problem is undecidable)

Computational complexity of testing that S is valid for any FOL sentence S? Undefined (the problem is undecidable)

Practical Complexity

Vocabulary: problem vs problem instance

Problem is an abstract description of a mathematical question

- Example: the Latin square problem

Problem instance is the specific input data

- Example: $n = 3$ and the known values are $M_{1,3} = 1$ and $M_{2,1} = 2$
- Sometimes also called ‘instance’ or ‘problem’

Problem can be seen as a collection $\mathcal{I}$ of all instances of that problem

Vocabulary: solution approach vs problem solution

Solution approach is your method to solve the problem:

- The formulation
- The solver software (e.g. Z3, OR-Tools)

Problem solution is the data that we are seeking:

- Timetable
- Values in the Latin square
- Assignment of pigeons to pigeon holes

What is complexity?

Complexity of an algorithm – how ‘long’ it takes

Complexity of a problem – class of problems that are similarly difficult

- More theoretical

The time complexity of a problem vaguely defines the minimum time complexity of the best possible algorithm

Note: we are talking about asymptotic behaviour

Worst-case complexity

- We have been talking about *worst-case complexity*
  - Formally speaking,
    $$\max_{I \subset \mathcal{I}} T(I)$$
    where $T(I)$ is the time taken to solve instance I and $\mathcal{I}$ is the set of all instances
- Worst-case complexity may only be relevant to very special cases
- Can we measure 'practical' complexity?

Practical complexity

- The concept is vague
- Could measure the average case complexity (average over all instances)

$$\frac{\sum_{I \subset \mathcal{I}} T(I)}{|\mathcal{I}|}$$

    - Somewhat meaningless for practical use
- Practical complexity is hard to formalise
    - Many instances are trivial
    - Want to consider only practical instances
    - Makes sense to focus on the hardest of practical instances

Undersubscribed vs Oversubscribed

What makes an instance hard?

(We will use the vocabulary of CSP but these concepts apply to any other logic)

What makes an instance hard?

- Obvious: number of variables
- What about the number of constraints?
- Also, are satisfiable instances easier or harder than unsatisfiable?

Number of constraints

Without constraints, any combination of variable values is a solution

For example, if we have variables $x_1, x_2 \in \{1, 2, 3\}$ and no constraints then

$$
\begin{aligned}
&x_1 = 1,\ x_2 = 1,\\
&x_1 = 1,\ x_2 = 2,\\
&x_1 = 1,\ x_2 = 3,\\
&x_1 = 2,\ x_2 = 1,\\
&\dots
\end{aligned}
$$

are all feasible solutions

As we keep adding constraints to a CSP instance, the number of solutions decreases

Eventually, the instance will become unsatisfiable

Undersubscribed instances

- If an instance has few/loose constraints, it is likely to have many solutions
- We call such instances *undersubscribed*, meaning that they are clearly satisfiable (sat)
- With only a few constraints, there will be little backtracking (we will study reasoning algorithms such as DPLL in a future lecture)
- As a result, undersubscribed instances are easy to solve

Oversubscribed instances

- If an instance has many/tough constraints, it is likely to be unsatisfiable
- We call such instances *oversubscribed*, meaning that they are clearly unsatisfiable (unsat)
- There are perhaps many ‘conflicts’ within an oversubscribed instance
- It is usually easy to arrive to one of the conflicts

The middle is hard

The hardest region is in between undersubscribed and oversubscribed instances

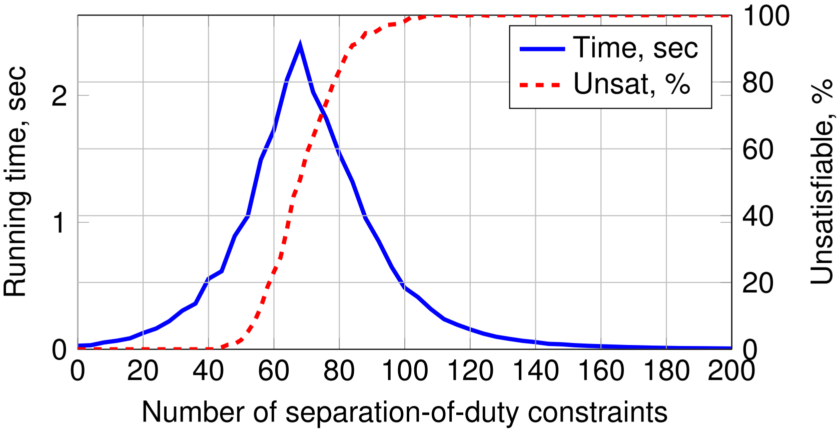
- No ‘obvious’ answers

In this region, an instance may be sat or unsat

- Sat instances have few solutions
- Unsat instances have few ‘conflicts’

It is called *phase transition region*

Number of constraints vs probability of unsatisfiability



Left	all instances satisfiable (undersubscribed) and easy
Middle	some instances sat, some unsat; hard
Right	all instances unsat (oversubscribed) and easy

What Can Make Reasoning Faster?

What can we do to improve performance of reasoning

Reasoning is generally computationally hard

It is not unusual that it is prohibitively slow

Some options to make reasoning faster:

- Decompose the problem into subproblems
- Break the symmetry of the problem
- Change the formulation
- Improve the solver

Decomposing the problem

Consider the following CSP instance:

$$
\begin{aligned}
&x_1 + x_2 < 5 \\
&x_3 \neq x_4 \\
&x_1, x_2, x_3, x_4 \in \{0, 1, 2, 3\}
\end{aligned}
$$

Can we split this into two independent problems?

Decomposing the problem

Observe that x_1 and x_2 do not depend on x_3 and x_4, and vice versa

The original problem is equivalent to the following two subproblems:

$$x_1 + x_2 < 5$$
$$x_1, x_2 \in \{0, 1, 2, 3\}$$

and

$$x_3 \neq x_4$$
$$x_3, x_4 \in \{0, 1, 2, 3\}$$

If either of these two subproblems is unsatisfiable, the original problem is also unsatisfiable

Otherwise the solutions of the two subproblems form a solution to the original problem

Decomposing the problem

Solving two subproblems is *much* easier than solving the original problem

- The size of the search space of the original problem is $4^4 = 256$
- The size of the search space of each of the subproblems is $4^2 = 16$
- Combined, the subproblems have the search space of size $16 \cdot 2 = 32$, which is much smaller than the original search space

Lessons:

- If the problem can be decomposed, decompose it!
- Solvers can identify simple cases automatically

Symmetry

Consider the following CSP instance:

$$\mathit{AllDiff}(\{x_1, x_2, x_3\})$$
$$x_1 + x_2 + x_3 < 20$$
$$x_1, x_2, x_3 \in \{0, 1, \ldots, 10\}$$

What can you tell about the solutions without solving the problem?

Symmetry

$$\mathit{AllDiff}(\{x_1, x_2, x_3\})$$
$$x_1 + x_2 + x_3 < 20$$
$$x_1, x_2, x_3 \in \{0, 1, \ldots, 10\}$$

Let us assume that, for some values y_1, y_2 and y_3, the following is a solution: $x_1 = y_1$, $x_2 = y_2$, $x_3 = y_3$

Then

$x_1 = y_2$, $x_2 = y_1$, $x_3 = y_3$ is also a solution

$x_1 = y_2$, $x_2 = y_3$, $x_3 = y_1$ is also a solution

...

Any permutation of values y is a solution

Symmetry

Symmetry generally makes reasoning unnecessarily hard

- For a human, it is obvious that all these solutions are equivalent
- The solver may, however, spend a lot of time traversing all these equivalent branches of the search tree

Symmetry breaking

Knowing about the symmetry, we can *break* it, e.g.

$$
\begin{aligned}
&\mathit{AllDiff}(\{x_1, x_2, x_3\})\\
&x_1 + x_2 + x_3 < 20\\
&x_1 \leq x_2 \leq x_3\\
&x_1, x_2, x_3 \in \{0, 1, \ldots, 10\}
\end{aligned}
$$

We can guarantee that

- Any solution to this new problem is a solution to the original problem
- If the original problem is satisfiable, this new problem is also satisfiable

Symmetry breaking can make a drastic effect on performance

- But symmetry is not always this obvious

Improving formulation

- There can be many formulations of the same problem

  For example, assuming that $x_i \in \{1, 2, \ldots, n\}$ for $i = 1, 2, \ldots, n$, there exist several ways to request that $x_1, x_2, \ldots, x_n$ have distinct values:

  - $\mathit{AllDiff}(\{x_1, x_2, \ldots, x_n\})$
  - $x_i \neq x_j$ for every $i \neq j$
  - $\forall i. \exists j. x_j = i$
- Good formulations enable intensive propagation
- How to design a formulation that enables intensive propagation?
  - It helps to have understanding of the solver's working
  - Experience is crucial
  - Trial-and-error

What can the solvers do?

To be effective, solvers have to exploit the structure of the problem instance

- Sometimes, they can identify symmetry or problem decompositions automatically
- They can remember solutions to fragments of the problem and then reuse these solutions in other branches of the search
- They prioritise variables that are most important/enable intensive propagation (recall DPLL)

Variable prioritisation

Consider the following problem:

$$
\begin{aligned}
x_1 \cdot x_4 &= 0 \\
x_2 \cdot x_4 &= 0 \\
x_3 \cdot x_4 &= 0 \\
x_1, x_2, x_3, x_4 &\in \{0, 1\}
\end{aligned}
$$

Possible reasoning process (inefficient):

- Try $x_1 = 0$; this satisfies the first constraint; no propagation
- Try $x_2 = 0$; this satisfies the second constraint; no propagation
- Try $x_3 = 0$; this satisfies the third constraint; problem solved

Alternative reasoning process (efficient):

- Try $x_4 = 0$; this satisfies all the constraints

Prioritising x_4 significantly improved the performance

How prioritisation works?

How does the solver decide which variables to prioritise?

- It analyses which variables will enable most propagation
  - In the example above, it was easy to assume that x_4 is most important because it participates in all the constraints
- It uses numerous heuristics ('hacks') to predict which variables will make the search quick
- The user may be able to supply the solver with insights about most important variables

Concluding notes

- While the worst-case computational complexity is pessimistic for most of the reasoning problems, real-world problem instances are usually not that hard
- It is difficult to formalise the realistic complexity of a problem
- The hardness of a problem significantly depends on whether it is undersubscribed, oversubscribed or in the phase transition region (in the middle between those)
- It is sometimes possible to exploit the structure of the problem instance to significantly speed up reasoning; the most interesting examples of this are problem decomposition and symmetry breaking
- Solvers spend significant effort on identifying such special structures but they may not be able to identify the more sophisticated cases; it is best to do this work for the solver when designing the formulation