Knowledge Representation and Reasoning

Isa-Ali Kirca

Contents

1 Basics

1.1 Symbols

Symbol	Explanation
$\neg x$	negation (not x)
$x \wedge y$	conjunction (x and y)
$x \vee y$	disjunction (x or y)
$x \rightarrow y$ **or** $x \supset y$	conditional/implies (if x then y)
$x \leftrightarrow y$ **or** $x \equiv y$	equivalence/iff (x if and only if y)
$a \models p$	a satisfies p
$\forall x$	for all x
$\exists x$	there exists x
$\exists ! x$	there is a unique x

Truth tables:

x	¬(x)
F	T
T	F

x	y	∧(x, y)	∨(x, y)	→ (x, y)	↔ (x, y)
F	F	F	F	T	T
F	T	F	T	T	F
T	F	F	T	F	F
T	T	T	T	T	T

1.2 Tautology and contradiction

Tautology: A compound proposition that is always true for all possible truth values of the propositions.
Example: $x \vee \neg x$ is a tautology.

x	¬(x)	x ∨ ¬ x
T	F	T
F	T	T

Contradiction: A compound proposition that is always false.
Example: $x \wedge \neg x$ is a contradiction.

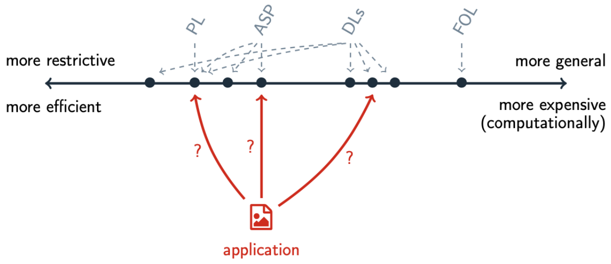
x	¬(x)	x ∧ ¬ x
T	F	F
F	T	F

Contingency: A proposition that is **neither** a tautology nor contradiction.

2 Week 1: Introduction

A variety of tools in KRR:

- Propositional logic (PL)
- First-order logic (FOL)
- Answer-Set Programming (ASP)
- Description logics (DL)



2.1 Propositional Logic (PL) - $\vee$

Propositional variables represent statements that can be **true** or **false**.
Logical operators are used to build more complex statements: $\wedge, \vee, \neg, \rightarrow$.

- **Syntax of propositional logic**
  - Take some (infinite) set P of propositional variables → P = $\{p_1, p_2, \cdots\}$
  - Propositional logic formulas construction:
    - Each p $\in$ P is a formula.
    - If φ is a formula, then $\neg\varphi$ is a formula too.
    - If φ_1, φ_2 are formulas, then so are $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$.
  - *Example propositional logic formula:* $(p_1 \vee p_2 \vee p_3) \leftrightarrow (\neg p_1 \rightarrow (p_2 \vee p_3))$
- **Semantics of propositional logic**
  - Let φ be a prop. logic formula that contains variables $p_1, \cdots, p_n$
  - Consider truth assignments $\alpha : P \rightarrow \{0, 1\}$, where 0 = false and 1 = true.
  - When a truth assignment α makes a formula φ true, **or** when it satisfies φ, written $\alpha \models \varphi$

    iff = if and only if

    - $\alpha \models p$ iff $\alpha(p) = 1$
    - $\alpha \models \varphi_1 \wedge \varphi_2$ iff both $\alpha \models \varphi_1$ and $\alpha \models \varphi_2$
    - $\alpha \models \varphi_1 \wedge \varphi_2$ iff $\alpha \models \varphi_1$ or $\alpha \models \varphi_2$ (or both)
    - $\alpha \models \neg\varphi$ iff $\alpha \not\models \varphi$
    - $\alpha \models \varphi_1 \rightarrow \varphi_2$ iff $\alpha \models (\neg\varphi_1) \vee \varphi_2$
    - $\alpha \models \varphi_1 \leftrightarrow \varphi_2$ iff $\alpha \models (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$

- **Conjunctive Normal Form (CNF)**
    - Useful to consider only formulas of a particular (**normal**) form.
    - A **CNF formula** is the conjunction ($\wedge$) of several clauses
        - A literal is a propositional variable p or the negation $\neg p$ of a variable
        - A clause is the disjunction ($\vee$) of several literals
        - *Example:* $(p_1 \vee \neg p_2) \wedge (p_1 \vee p_2) \wedge (\neg p_1 \vee p_3 \vee \neg p_4)$
        - *Every propositional logic formula can be equivalently expressed as a **CNF formula**.*
- **Reasoning tasks**
    - Satisfiability: Given a formula φ, does there exist a truth assignment α that satisfies φ?
        - Satisfiability can be **reduced to** non-entailment:
          φ is satisfiable iff $\varphi \not\models (p_1 \wedge \neg p_2)$
    - Entailment: Given two formulas φ_1, φ_2, does it hold that φ_1 entails φ_2, written as $\varphi_1 \models \varphi_2$?

      $\varphi_1 \models \varphi_2$ iff for all assignments α such that $\alpha \models \varphi_1$ and $\alpha \models \varphi_2$.
      (*Intuitively:* if one assumes that φ_1 is true, then φ_2 must also be true).
        - Entailment can be **reduced to** unsatisfiability:
          $\varphi_1 \models \varphi_2$ iff $\varphi_1 \wedge \neg\varphi_2$ is not satisfiable.

2.2 First-order Logic (FOL) - $\forall$

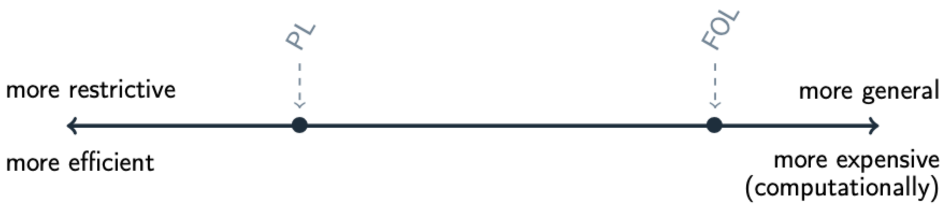
Extends capabilities of Propositional Logic.
Express properties about objects, their properties, and relations between objects:

- **Main ingredients**
    - Language constructs for objects
        - Object constants $c_1, c_2, \cdots$, function constants $f_1, f_2, \cdots$
        - Variables (for objects) and quantifiers $\forall, \exists$
        - Relation symbols $R_1, R_2, \cdots$
    - Objects in the notion of interpretations

- **Syntax first-order logic**
  - Fix sets of:
    - Function constants $f_1, f_2, \cdots$, each with an arity $k \in \mathbb{N}$
    - Relation symbol $R_1, R_2, \cdots$, each with an arity $k \in \mathbb{N}$
    - (Object) variables $x_1, x_2, \cdots, y_1, y_2, \cdots, z_1, z_2, \cdots$
  - Terms are constructed as follows:
    - Each variable x is a term.
    - If $t_1, \cdots, t_k$ are terms and f is a function constant of arity k, then $f(t_1, \cdots, t_k)$ is a term.
      (If c is a function constant of arity 0, then c is a term by itself.)
  - Formulas are constructed as follows:
    - If t_1, t_2 are terms, then $(t_1 = t_2)$ is a formula.
    - If R is a relation symbol of arity k and $t_1, \cdots, t_k$ are terms, then $R(t_1, \cdots, t_k)$ is a formula.
      (If R is a relation symbol of arity 0, then R is a formula by itself).
    - If φ_1, φ_2 are formulas, then so are: $\neg\varphi_1$, $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$, $(\varphi_1 \leftrightarrow \varphi_2)$
    - If x is a variable and φ is a formula, then so are: $\forall x.\varphi$ and $\exists x.\varphi$
  - Example: If R is a unary relation symbol, then the following are formulas:
    (unary = 1-ary = arity 1)
    $R(x), \quad \exists x.R(x), \quad \exists x.\forall y.(R(x) \vee (x = y))$
  - The subformulas Sub(φ) of a formula φ are all formulas that appear as a part of φ:
    $$\begin{aligned} \mathrm{Sub}(\varphi_1 \circ \varphi_2) &= \{\varphi_1 \circ \varphi_2\} \cup \mathrm{Sub}(\varphi_1) \cup \mathrm{Sub}(\varphi_2) \\ \mathrm{Sub}(\neg\varphi) &= \{\neg\varphi\} \cup \mathrm{Sub}(\varphi) \\ \mathrm{Sub}(R(t_1, \cdots, t_k)) &= \{R(t_1, \cdots, t_k)\} \\ \mathrm{Sub}(t_1 = t_2) &= \{(t_1 = t_2)\} \end{aligned}$$
  - An occurence of a variable x in a formula φ is bound if it appears in a subformula of φ of the form $\forall x.\psi$ or $\exists x.\psi$
  - An occurrence of a variable is free if it is not bound
  - Free(φ) denotes the set of all free variables of φ: variables that have a free occurrence in φ
  - *Example:* There is one bound and one free occurrence of x and one bound occurrence of y in the formula $\varphi = R_1(\mathrm{x}) \vee \exists \mathrm{x}.\forall \mathrm{y}.R_2(x, y)$

- **Semantics first-order logic**
  - Instead of truth assignments, we consider interpretations $(l, \cdot^l)$ for the meaning of first-order logic formulas, consisting of:
    - a (possibly infinite) set l, called the domain;
    - a function $\cdot^l$, called the interpretation function, that:
      - assigns each function constant f of arity k to a function $f^l : l^k \to l$ (and thus assigns each object constant c to an object $c^l \in l$)
      - assigns each relation symbol R of arity k to a relation $R^l \subseteq l^k$ (and thus assigns each 0-ary relation symbol R to true, $\{()\} \subseteq l^0$, or false, $\emptyset \subseteq l^0$)
  - We define when an interpretation $(l, \cdot^l)$ and a variable assignment $\mu : \mathrm{Free}(\varphi) \to l$ **satisfy** a formula φ, written $l, \mu \models \varphi$.

$$
\begin{array}{ll}
x^{l,\mu} & = \mu(x) \\
(\mathrm{c})^{l,\mu} & = \mathrm{c}^l \\
(\mathrm{f}\,(t_1, \ldots, t_k))^{l,\mu} & = \mathrm{f}'\left((t_1)^{l,\mu}, \ldots, (t_k)^{l,\mu}\right) \\
l, \mu \models (t_1 = t_2) & \text{iff } (t_1)^{l,\mu} = (t_2)^{l,\mu} \\
l, \mu \models \mathrm{R}\,(t_1, \ldots, t_k) & \text{iff } \left((t_1)^{l,\mu}, \ldots, (t_k)^{l,\mu}\right) \in \mathrm{R}^l \\
l, \mu \models \neg\varphi & \text{iff } l, \mu \not\models \varphi \\
l, \mu \models \varphi_1 \wedge \varphi_2 & \text{iff} \quad \text{both } l, \mu \models \varphi_1 \text{ and } l, \mu \models \varphi_2 \\
l, \mu \models \varphi_1 \vee \varphi_2 & \text{iff} \quad \text{at least one of } l, \mu \models \varphi_1 \text{ and } l, \mu \models \varphi_2 \\
l, \mu \models \exists x \cdot \varphi & \text{iff} \quad \text{there is some } d \in l \text{ such that } l, \mu' \models \varphi \\
 & \quad \text{where } \mu' = \mu \cup \{x \mapsto d\}, \\
l, \mu \models \forall x \cdot \varphi & \text{iff } l, \mu' \models \varphi \text{ for all } \mu' = \mu \cup \{x \mapsto d\} \text{ where } d \in l
\end{array}
$$



Tradeoff propositional logic and first-order logic

- **Extensions (make modelling more convenient)**
  - Sorts
    - Intuitively: types of 'variables'
    - E.g. numbers ($\forall x \in \mathbb{N}.\varphi$), sets, lists
  - Additional (abbreviations for) quantifiers
    - E.g. 'there exists exactly one x such that ..":

$$\exists! x.\varphi(x) = \exists x.(\varphi(x) \wedge \forall y.(\varphi(y) \rightarrow (x = y)))$$

- **Noteworthy**: reasoning over computer programs can be expressed using FOL
  - Equivalence: Given two computer programs P_1, P_2 do they compute the same outcome for each possible input?
  - Termination: Given a computer program P, for each possible input does it eventually terminate?
  - Correctness: Given a computer program P and a specification of a function f, does P compute the function f?
- For example, Turing machines (math. model of computation) can be encoded into First-order logic

2.3 Second-order Logic (SOL)

- **SOL** has **quantification** over relations: e.g., $\exists R.\forall x.\exists y.R(x, y)$
- Monadic Second-order Logic (MSOL) is an often considered variant where second-order quantification is only over unary predicates

2.4 Uncomputability

- Main notion in the field of **Recursion theory**.
- A function $f : X \rightarrow Y$ is computable if there exists a computer program P that computes this function f
  - For each input $x \in X$, running P on input x:
    1. terminates, and
    2. gives as output $f(x)$

2.4.1 Halting Problem (Alan Turing, 1936)

Example of an uncomputable function:

> **Input**: a computer program P (Turing machine) and an input x
>
> **Output**: does P, when run on x as input, eventually terminate (halt)?

- Satisfiability of First-order Logic:
  - > **Input**: a first-order logic formula φ
  - > **Output**: does there exist an interpretation $(l, \cdot^l)$ that satisfies φ?
  - > **Uncomputable!**
  - > e.g. there exists no algorithm that for each FOL formula φ correctly tells us whether φ is satisfiable (and that always gives an answer).
  - > Similarly for the problems of validity, equivalence and logical entailment

FOL validities can be enumerated:

- We can enumerate all valid first-order logic sentences

2.4.2 Gödel's Completeness Theorem (1929)

- ∘ All valid first-order logic formulas can be proven
  - ∗ A formula φ is valid if every interpretation $(l, \cdot^l)$ satisfies φ.
    In other words, φ is valid iff $\neg\varphi$ is **not satisfiable**.
- ∘ In an appropriate proof system

- We can go over all possible proofs one-by-one.
  If there is a proof of φ, then we will find it after some finite amount of time.

Restrictions of FOL:

- FOL is too powerful to effectively reason with algorithmically, in general.
- To enable algorithms that always give the correct answer, restrictions are considered, e.g.:
  - ∘ Only allow formulas of a particular form (e.g., the guarded fragment of FOL)
  - ∘ Languages that contain only some operators from FOL (and that sometimes look a bit different; e.g., description logics)

2.5 Computational complexity

Computational Complexity is a part of theoretical computer science and studies how much resources (e.g., time, space) you need to solve computational problems.

- ”Is there an algorithm for this problem that is guaranteed to be efficient?”
- Measures the amount of time/space in terms of the length of the input.

Main method: categorizing computational problems into complexity classes

- **Polynomial-time solvability** is the typical notion of "efficiently solvable"
  - An algorithm runs in polynomial time if there is a polynomial function $p(n)$ such that on **any** input (of length n), the algorithm terminates within at most $p(n)$ time steps. E.g., $p(n) = 2n$, or $p(n) = 10n^2 + 100n$
- The complexity class P contains problems that are solvable in polynomial time.
- The complexity class NP contains problems where:
  - the goal is to find a solution in a search space that is of exponential size (in the input size n);
  - for each candidate solution, you can efficiently (in polynomial time) check whether or not it is a correct solution.
- An example of an NP problem is satisfiability for propositional logic (SAT).
- Best alg. for hard problems in NP take **exponential time**, in the **worst case**.
- **More complexity classes:**
  - □ **PSPACE**: problems that can be solved with polynomial memory usage (possibly exponential time).
  - □ **EXP**: problems that can be solved within exponential time (possibly exponential memory usage).
  - □ **EXPSPACE**: problems that can be solved with exponential memory usage (possibly doubly-exponential time).

2.6 Monotonicity

Propositional logic and first-order logic are **monotonic** in the following sense:

- Take some formula φ_1
- Suppose that I can derive some conclusion from it: $\varphi \models \psi$
- Now if I add further information to my formula: $\varphi_1 \wedge \varphi_2$
- Then I can always still derive the original conclusion: $\varphi_1 \wedge \varphi_2 \models \psi$

For some types of applications, **non-monotonic reasoning** is **desirable**:

- **Non-monotonicity**: "new information can cause me to retract previous conclusions".
- It can be that $\varphi_1 \models \psi$ but $\varphi_1 \wedge \varphi_2 \not\models \psi$.

For example:

- "Given evidence E, the most likely explanation of something is X"
- "With new evidence, so $E \cup E'$, the most likely explanation now is $X' \neq X$ ".

Non-monotonicity is another feature on the wish list for knowl. repr. languages.

3 Week 2: SAT and DPLL

3.1 SAT

The satisfiability problem for propositional logic (SAT):

- **Input**: a propositional logic formula φ (in CNF)
- **Output**: is there a truth assignment α that satisfies φ?

Reminder solving SAT allows us to perform various reasoning tasks:

- φ_1 logically entails φ_2 ($\varphi_1 \models \varphi_2$)
  iff $\varphi_1 \land \neg\varphi_2$ is **not satisfiable**.
- φ_1 and φ_2 are logically equivalent ($\varphi_1 \equiv \varphi_2$)
  iff $(\varphi_1 \land \neg\varphi_2) \lor (\varphi_2 \land \neg\varphi_1)$ is **not satisfiable**.
- φ is valid
  iff $\neg\varphi$ is **not satisfiable**.

Algorithms for SAT:

- Most naive algorithm: iterate over all 2^n truth assignments
  (n is number of propositional variables in formula φ)
- All algorithms for SAT take **exponential time** in the **worst case**, but perform very well in practice.
- Most algorithms work only for propositional logic formulas in CNF, but translation to CNF often leads to **exponential blow-up** in the formula.

That's where the **Tseytin transformation** comes in handy.

3.2 Tseytin Transformation

- We can translate an arbitrary propositional logic formula φ efficiently to a **equisatisfiable CNF** formula χ-so φ and χ do not have to be logically equivalent
  - **Equisatisfiable**: φ is satisfiable iff χ is satisfiable.

- The **Tseytin transformation**:
  - Take a formula φ and call its variables $p_1, \cdots, p_n$.
  - Take all the subformulas $\psi_1, \cdots, \psi_m$ of φ, where $\psi_1 = \varphi$.
  - Introduce new propositional variables $q_1, \cdots, q_m$ -each q_i will represent whether the subformula ψ_i is satisfied.
  - For each subformula ψ_i, add some clauses to χ:
    - If $\psi_i = p_j$, add $(\neg q_i \vee p_j)$ and $(\neg p_j \vee q_i)$; "$(q_i \leftrightarrow p_j)$"
    - If $\psi_i = \neg\psi_j$, add $(\neg q_i \vee \neg q_j)$ and $(q_j \vee q_i)$; "$(q_i \leftrightarrow \neg q_j)$"
    - If $\psi_i = (\psi_j \wedge \psi_k)$, add $(\neg q_i \vee q_j)$, $(\neg q_i \vee q_k)$, and $(\neg q_j \vee \neg q_k \vee q_i)$; "$(q_i \leftrightarrow q_j \wedge q_k))$"
    - If $\psi_i = (\psi_j \vee \psi_k)$, add $(\neg q_i \vee q_j \vee q_k)$, $(\neg q_j \vee q_i)$, and $(\neg q_k \vee q_i)$; "$(q_i \leftrightarrow (q_j \vee q_k))$"
    - (and similarly for subformulas built using $\rightarrow, \leftrightarrow, ..$)
  - **Finally**, add the unit clause (q_1).
  - Then, χ is satisfiable iff φ is satisfiable.

3.3 Algorithms

3.3.1 Backtracking search

Main idea of **Backtracking search**:

- Pick some truth value for a variable x, and try if that leads to a solution. If so, great!
- Else (if not), backtrack and try the other value.

Useful notation: $\varphi|_\ell$ ('plugging in' ℓ into φ).

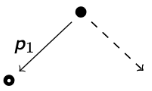
- Let φ be a CNF formula and ℓ a literal;
- To obtain $\varphi|_\ell$ from φ:
  1. Remove all clauses that contain ℓ (already satisfied);
  2. Remove $\neg\ell$ from all remaining clauses ($\neg\ell$ cannot be satisfied anymore)

Algorithm 1: Backtracking search (BS)

```
BS($\varphi, \alpha$):
Input  : a propositional CNF formula $\varphi$,
         and a partial truth assignment $\alpha$ ;            # $\alpha$ is initially empty
Output: a satisfying truth assignment $\beta$ for $\varphi$ if it exists, otherwise "unsat"
begin
    if $\varphi$ contains the empty clause then
        return "unsat" ;                  # empty clause means contradiction
    if $\varphi$ has no clauses then
        return $\alpha$ ;            # no clauses means the formula is satisfied

    $\ell \leftarrow$ a literal in $\varphi$ not assigned by $\alpha$ ;     # the branching choice:  which $\ell$
    if BS($\varphi|_\ell, \alpha \cup \{\ell\} = \beta$) then
        return $\beta$ ;                                           # first try $\ell$
    else
        return BS($\varphi|_{\neg\ell}, \alpha \cup \{\neg\ell\}$) ;               # if $\ell$ did not work, try $\neg\ell$
```

Ass: p_1

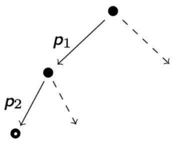


$(\neg p_1 \vee \neg p_2)$

$(\neg p_1 \vee p_2)$

✓ $(p_1 \vee \neg p_2)$

Ass: p_1 p_2

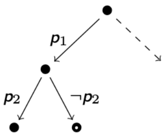


✗ $(\neg p_1 \vee \neg p_2)$

✓ $(\neg p_1 \vee p_2)$

✓ $(p_1 \vee \neg p_2)$

Ass: p_1 $\underline{\neg p_2}$

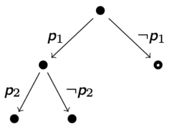


✓ $(\neg p_1 \vee \neg p_2)$

✗ $(\neg p_1 \vee p_2)$

✓ $(p_1 \vee \neg p_2)$

Ass: $\underline{p_1}$

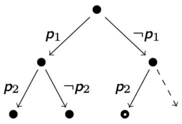


✓ $(\neg p_1 \vee \neg p_2)$

✓ $(\neg p_1 \vee p_2)$

$(p_1 \vee \neg p_2)$

Ass: $\underline{\neg p_1}$ p_2

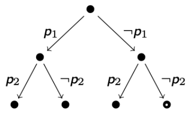


✓ $(\neg p_1 \vee \neg p_2)$

✓ $(\neg p_1 \vee p_2)$

✗ $(p_1 \vee \neg p_2)$

Ass: $\underline{\neg p_1 \neg p_2}$



✓ $(\neg p_1 \vee \neg p_2)$

✓ $(\neg p_1 \vee p_2)$

✓ $(p_1 \vee \neg p_2)$

In the example above, we have no empty or no clauses. We start with true, then false and go in ascending order (order picking). Furthermore the dashed line means that we still have to expand this branche later maybe. The colors indicate whether the assignment matches the literal(s). In case the color is red, the literal is removed from the clause. If all the literals are removed and we have a ✗, that branch does not give us a solution, so we need to backtrack to the last point. A ✓ means that the clause is removed. The underlining in the assignment, means that we have no other choice anymore for that literal(s). Once all the clauses have a ✓, we have a solution.

3.3.2 Propagation: unit propagation (UP)

Propagation: simplifying the formula at each node in the search tree.
Unit Propagation is a particular propagation method:

- If φ contains a unit clause (ℓ), then set the literal ℓ to true.
- Repeat this until there are no more unit clauses (or until the formula contains the empty clause)

DPLL makes use of unit propagation.

Algorithm 2: DPLL

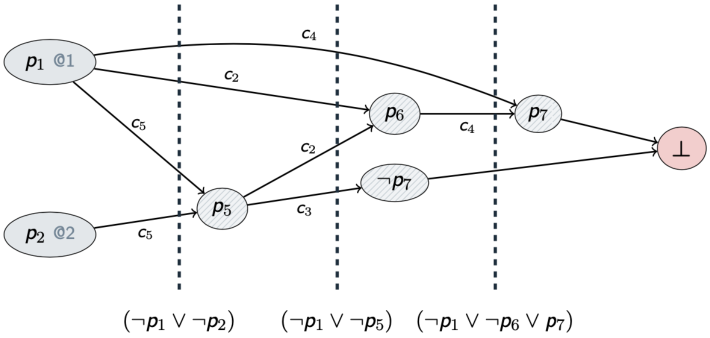
```
DPLL($\varphi, \alpha$):
Input  : a propositional CNF formula $\varphi$,
         and a partial truth assignment $\alpha$
Output: a satisfying truth assignment $\beta$ for $\varphi$ if it exists, otherwise "unsat"
begin
    $(\varphi, \alpha) \leftarrow \mathbf{UP}(\varphi, \alpha)$ ;            # update $(\varphi, \alpha)$ with unit propagation
    if $\varphi$ contains the empty clause then
        return "unsat"
    if $\varphi$ has no clauses then
        return $\alpha$
    $\ell \leftarrow$ a literal in $\varphi$ not assigned by $\alpha$ if $\boldsymbol{BS}(\varphi|_{\ell}, \alpha \cup \{\ell\} = \beta)$ then
        return $\beta$
    else
        return $\boldsymbol{BS}(\varphi|_{\neg\ell}, \alpha \cup \{\neg\ell\})$

UP($\varphi, \alpha$):
begin
    while $\varphi$ contains no empty clause but some unit clause $\ell$ do
        $\varphi \leftarrow \varphi|_{\ell}$ ;          # $\ell$ must be true, so 'plug in' $\ell$ into $\varphi$
        $\alpha \leftarrow \alpha \cup \{\ell\}$ ;                          # and add $\ell$ to $\alpha$
    return $(\varphi, \alpha)$
```

The algorithm will still do some things over and over again.
A way to **deal** with this: **conflict analysis** and **clause learning**.

3.3.3 Conflict graph (implication graph)

Example of a **conflict graph**:



p_1 @1
c_4
c_2
c_5
p_6
c_4
p_7
$\bot$
c_2
$\neg p_7$
p_5
c_3
p_2 @2
c_5
$(\neg p_1 \vee \neg p_2)$
$(\neg p_1 \vee \neg p_5)$
$(\neg p_1 \vee \neg p_6 \vee p_7)$

The **implication graph** is a directed graph that is constructed as follows:

- Add a node for each decision literal—i.e., literals that can still be backtracked on.
- Every time unit propagation has been used with some clause $(\ell_1 \vee \cdots \vee \ell_k)$ to conclude, say, ℓ_k draw an edge from the nodes for $\neg\ell_1, \cdots, \neg\ell_{k-1}$ to the node for ℓ_k (and label these edges with the clause number).
- Add a node for the conflict (with label $\bot$), and for any two nodes for negated literals (i.e., ℓ and $\neg\ell$), draw an edge from these nodes to the conflict node.

For each state of the algorithm, there is a unique implication graph—that might contain several pairs of conflicting literals.

A **conflict graph** is a subgraph of the implication graph that contains exactly one conflicting pair of literals:

- Pick some conflicting pair $\ell, \neg\ell$ of literals in the implication graph, together with the conflict node $\bot$.
- Keep only those nodes that have a path to either ℓ or $\neg\ell$, and remove all other nodes from the implication graph.

3.3.4 Further additions to DPLL

- **Pure literal elimination (PL)**: if φ contains a literal ℓ but not $\neg\ell$, then you can safely set ℓ to true.
- **Backjumping**: learning a clause can enable backtracking several choices at once
- **Branching heuristics**: pick a literal ℓ that immediately satisfies as many clauses as possible
- **Random restarts**: Restart search every now and then (keep learned clauses).

3.4 SAT and NP-completeness

A problem Q is **NP-complete** if every other problem Q' in the class NP can be efficiently (in polynomial-time) be reduced to it (and if Q is in NP).

- A reduction is an algorithm p that maps an input x' for Q' to an input x for Q such that the outputs are the same—e.g., $Q(x) = Q'(x')$.

SAT is **NP-complete**.

4 Week 3: Logic programming and ASP

In **logic programming** you can express facts and if-then rules. Example: *Prolog*. **Goal**: get computationally useful language to represent knowledge and reason with it.

Positive logic programs consist of rules and facts (no negations and nots).

- **Rules** are of the form (can all be true or false):

  ```
  a :- b, c, d, ..., z.
  ```

  which represents the implication $(b \wedge c \wedge d \wedge \ldots \wedge z) \rightarrow a$
  - **a** is **head** of the rule.
  - $b, c, d, \ldots, z$ is **body** of the rule.
- **Facts** are of the form:

  ```
  a.
  ```

  which represents the positive literal a.

In **logic programming**, database semantics is used:

- Objects mentioned are the only objects (**domain closure**)
- Objects with different names are different objects (**unique-names assumption**)
- (Atomic) statements that are not mentioned are false (**close-world assumption**)

So we do not have to explicitly say what function symbols mean. We can represent objects simply by the terms that point to them. We can represent the meaning of a relation R by the set of atoms (over R) that are true-and then all atoms that are not in this set are false.

For **positive logic programs**, there is a **unique minimal model**:

- Minimal in terms of subset-inclusion
- Model is an interpretation that makes all rules true

For example the minimal model for the program below is: $\{a, b, c\}$:

```
a :- b, c.
b :- c.
c.
d :- e.
```

We can find this **minimal model** M with the following procedure:

1. Start by putting all facts of the program into M.
2. Repeat until M does not change anymore:
   If there is a rule $b \leftarrow c_1, \cdots, c_n$ where $c_1, \cdots, c_n \in M$ put b in M too.

Convention to: write variables starting with capital letters and relation and function symbols starting with small letters. Variables are always universally quantified ($\forall$).

In **positive logic programs** it is **not allowed** to use negation in the body of rules (not, which is $\neg$). However, some extensions allow this: **Datalog**.

In **Datalog**, rules are of the following form:

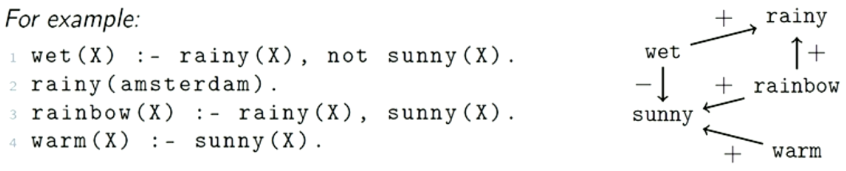
a :— b_1 , . . . , b_n , not c_1 , . . . , not c_m .

There are some restrictions on programs:

- Function symbols (with arity > 0) are not allowed
- Every variable that appears in the head of a rule, must also appear in a non-negated atom in the body.
- Every variable that appears in a negated atom in the body of a rule, must also appear in a non-negated atom in the body.
- Negation must be stratified.

Negation is stratified in a program P if the following holds:

- Draw a directed graph, where the nodes are relation symbols appearing in P.
- For every rule a :- $b_1, \ldots, b_n$, not $c_1, \ldots,$ not c_m in P:
  - Draw an edge labelled with - (a negative edge) from the relation symbol in a to the relation symbol in c_i, for each $1 \leq i \leq m$.
  - Draw an edge labelled with + (a positive edge) from the relation symbol in a to the relation symbol in b_i, for each $1 \leq i \leq n$.
- If this graph has no cycles involving negative edges, then negation is stratified.



For example:

```
wet(X) :- rainy(X), not sunny(X).
rainy(amsterdam).
rainbow(X) :- rainy(X), sunny(X).
warm(X) :- sunny(X).
```

Datalog programs have a unique minimal model as well. We can find it with the following procedure:

1. Assign a positive integer to each relation symbol such that:
   When there is a negative edge from a to b, then the number assigned to a is strictly largert than that assigned to b (since there are no cycles with negated edges).
2. Start by putting all facts of the program into M.
3. Proceed in stages, one stage for each assigned integer i, going from low to high. In each stage i:
   - Repeat until M does not change anymore:
     If there is a rule $b \leftarrow c_1, \ldots, c_n$ not $d_1, \ldots,$ not d_m in P where $c_1, \ldots, c_n \in M$ and $d_1, \ldots, d_m \notin$, and b is assigned the number i, then put b in M too.

4.1 Answer Set Programming (ASP)

What if we want to use negation, but it is not stratified in our knowledge base (unstratified negation)?

For example:

```
low :- not high.
high :- not low.
```

Then there is not always a unique minimal model. In this example, the following are (subset-) minimal models: {low} and {high}.
Answer Set Programming (ASP) assigns the so-called answer set semantics to logic programs with negations.

In the basic language of ASP, programs may contain facts and rules of the form:

$\texttt{a :- } b_1\texttt{,} \ \ldots\texttt{,} \ b_n\texttt{,} \ \texttt{not } c_1\texttt{,} \ \ldots\texttt{,} \ \texttt{not } c_m\texttt{.}$

- Negation does not have to be stratified.
- It may contain variables, that must appear safely:
  - Every variable that appears in the head of a rule, must also appear in a non-negated atom in the body
  - Every variable that appears in a negated atom in the body of a rule, must also appear in a non-negated atom in the body.

Semantics of ASP (intuitions):

1. Consider an interpration M as an assumption for which atoms are true and which are false.
2. Construct a (positive) variant P^M of the program P that takes into account this assumption.
3. Check whether M is the (unique) minimal model of P^M.

Take a normal logic program P and an interpretation M.

- The reduct P^M of P w.r.t. M is obtained from P by:
  1. removing rules with not a in the body, for $a \in M$
  2. removing literals not b from all rules, for $b \notin M$
- An answer set of P is a set M that is the minimal model of P^M.

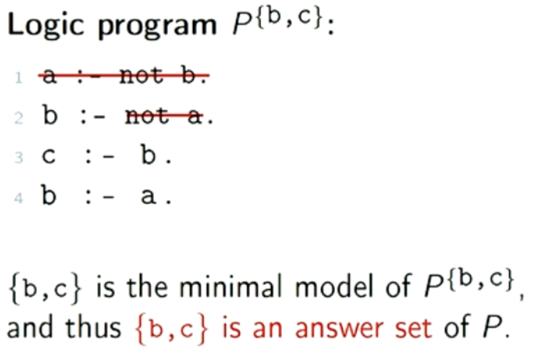
For example, take P to be:

low :- not high.
high :- not low.

and $M = \{\text{high}\}$. Then P^M is:

high.

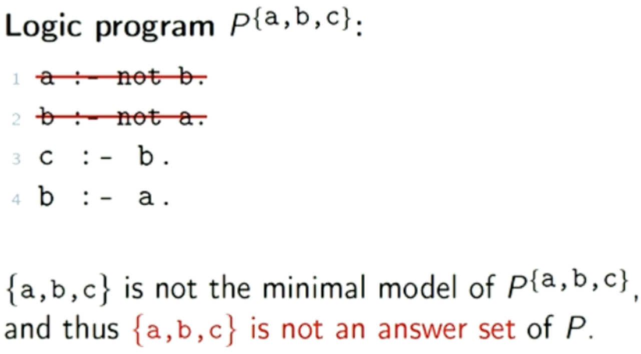
and M is the minimal model of P^M.



Logic program $P^{\{b,c\}}$:

~~a :- not b.~~
b :- ~~not a~~.
c :- b.
b :- a.

{b,c} is the minimal model of $P^{\{b,c\}}$, and thus {b,c} is an answer set of P.



Logic program $P^{\{a,b,c\}}$:

~~a :- not b.~~
~~b :- not a.~~
c :- b.
b :- a.

{a,b,c} is not the minimal model of $P^{\{a,b,c\}}$, and thus {a,b,c} is not an answer set of P.

In the figures above: For any 2 answer sets of a program, they cannot be subsets of each other. The left figure shows that $\{b, c\}$ is an answer set, which means that $\{a, b, c\}$ cannot be an answer set anymore.

Some clarifications:

- The first rule in the **left figure** gets thrown out completely. This is because when building the reduct, we only look at the literals with not. So for every not *something*, where *something* is in the set P^M, that rule gets thrown out completely.
- In the second rule of the **left figure** only the right part of the rule gets striked, since we again look at not *something*, but *something* is not in the set P^M. Thus we strike **not *something*** in that rule.
- Thereafter we do the iterative procedure and see that $\{b, c\}$ is the answer set of P.
- In the **right figure** we go through the same procedure, except that $\{a, b, c\}$ is not an answer set, since it not the minimal model of $P^{\{a,b,c\}}$

4.2 Building blocks of ASP (- check other PDF)

Logic program with constraint:

```
num(1).
num(2).

 left(X) :- not right(X), num(X).
 right(X) :- not left(X), num(X).

:- left(1), left(2).    # left(1) and left(2) cannot be true both.
```

Answer sets (because of constraint):

```
num(1) num(2) right(1) left(2)
num(1) num(2) right(1) right(2)
num(1) num(2) left(1) right(2)
```

Helpful feature: # show:
If you want to see a specific part (subset) of the answer set (not changing the problem or answer).

Helpful feature: abbreviations:
num(1..3) is equivalent to: num(1). num(2). num(3).

num(a;c) is equivalent to: num(a). num(c). **So not up to!**

Helpful feature: #const:
A constant. So for example #const k=2 and thereafter num(1, k) will give num(1) up to num(k).

Choice rules:
Logic program:

```
{ a; b }.
```

Possible translation:

```
a :- not na.
na :- not a.
b :- not nb.
nb :- not b.
```

Answer sets:

```
na nb
a nb
na b
a b
```

Choice rule with lower bound (choose at least 2 of them):

```
2 { a; b; c }
```

So the anwser set becomes: a b, a c, b c and a b c.

Choice rule with upper bound (choose at most 1 of them):

```
{ a; b; c } 1
```

So the anwser set becomes: {empty answer set}, a, and b.

Choice rule with lower and upper bound (choose exactly 2 of them):

```
2 { a; b; c } 2
```

So the anwser set becomes: a b, a c, and b c.

Conditional literals (1):

Logic program:

```
p(1..3).
q(1..2).
s :- p(X) : q(X)
```

Answer sets:

```
p(1) p(2) p(3) q(1) q(2) s
.
```

Conditional literals (2):

Logic program:

```
p(1..3).
q(1..2).
r(1..3).
s :- p(X) : q(X), r(X)     .
```

Answer sets:

```
p(1) p(2) p(3) q(1) q(2) r(1) r(2) r(3) s
```

Conditional literals (3):

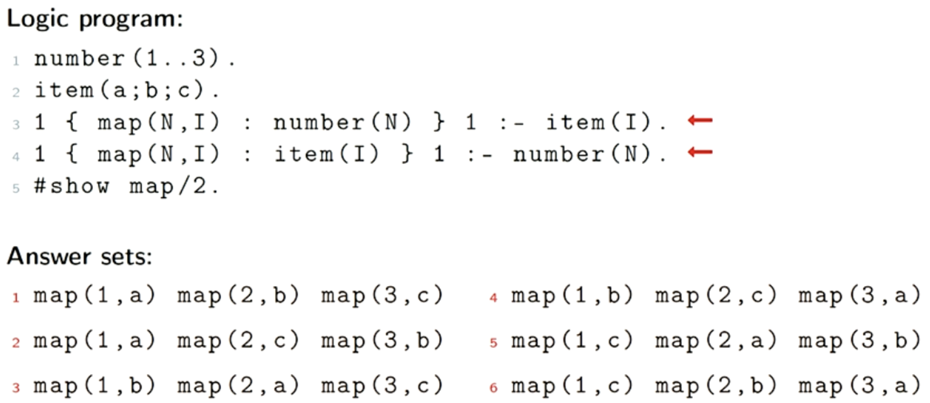
Logic program:

```
p(1..3).
q(1..2).
r(1..3).
t.
s :- p(X) : q(X), r(X) ; t.
```

Answer sets:

```
p(1) p(2) p(3) q(1) q(2) r(1) r(2) r(3) s t
```

One-to-one mappings:



Logic program:

```
number(1..3).
item(a;b;c).
1 { map(N,I) : number(N) } 1 :- item(I). ←
1 { map(N,I) : item(I) } 1 :- number(N). ←
#show map/2.
```

Answer sets:

```
1 map(1,a) map(2,b) map(3,c)
2 map(1,a) map(2,c) map(3,b)
3 map(1,b) map(2,a) map(3,c)
4 map(1,b) map(2,c) map(3,a)
5 map(1,c) map(2,a) map(3,b)
6 map(1,c) map(2,b) map(3,a)
```

4.3 Generate and test

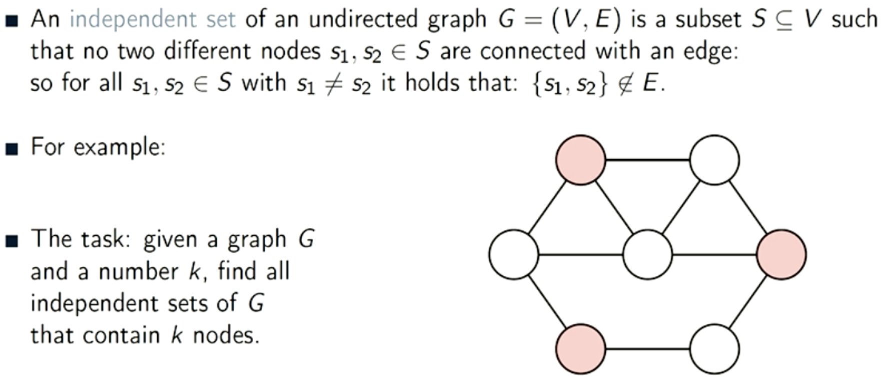
Some useful general guidelines for modelling a problem in ASP:

- Seperate the input (the encoding of the problem input) from the problem (the encoding of the solution requirements).
- The generate and test approach.

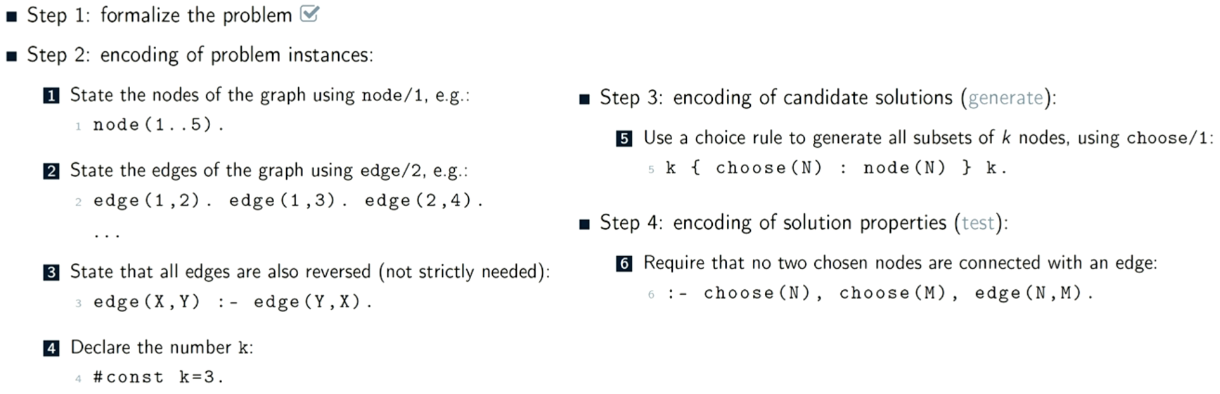
Both have 4 steps:

1. **Formalize the problem**
    - What is the problem input?
    - What kind of objects are (candidate) solutions?
    - What properties do solutions need to have?
2. **Establish encoding of problem instances**
    - What predicates to use to represent the problem input? What is their arity?
    - What constants to use to represent the problem input?
    - How to translate the problem input to facts that use these predicates and constants.
3. **Establish encoding of candidate solutions (generate)**
    - What predicates to use to represent candidate solutions? What is their arity?
    - What rules/constraints to add so that the answer sets of the program correspond to all candidate solutions?
    - Do you need any auxiliary predicates/rules to express some properties that candidate solutions should have?
    - How to obtain from any answer set the corresponding candidate solution?
4. **Establish encoding of solution properties (test)**
    - What rules/constraints to add so that only the answer sets remain that correspond to actual solution?
    - Do you need any auxiliary predicates/rules to express some properties that actual solutions should have?

Example:



- An independent set of an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that no two different nodes $s_1, s_2 \in S$ are connected with an edge: so for all $s_1, s_2 \in S$ with $s_1 \neq s_2$ it holds that: $\{s_1, s_2\} \notin E$.
- For example:
- The task: given a graph G and a number k, find all independent sets of G that contain k nodes.



- Step 1: formalize the problem ☑
- Step 2: encoding of problem instances:

1 State the nodes of the graph using node/1, e.g.:

```
node(1..5).
```

2 State the edges of the graph using edge/2, e.g.:

```
edge(1,2). edge(1,3). edge(2,4).
...
```

3 State that all edges are also reversed (not strictly needed):

```
edge(X,Y) :- edge(Y,X).
```

4 Declare the number k:

```
#const k=3.
```

- Step 3: encoding of candidate solutions (generate):

5 Use a choice rule to generate all subsets of k nodes, using choose/1:

```
k { choose(N) : node(N) } k.
```

- Step 4: encoding of solution properties (test):

6 Require that no two chosen nodes are connected with an edge:

```
:- choose(N), choose(M), edge(N,M).
```

4.4 Grounding

Grounding algorithm:

- Keep track of a set D of atoms that appear in the head of some ground rule. Initially, D contains the facts of the program P.
- Apply the following procedure, until D does not change anymore:
  - If there is a ground version of some rule in P where all non-negated atoms in the body appear in D, add the head of the rule to D.
- For example:

```
path(a,b).
path(b,c).
path(X,Z) :- path(X, Y), path(Y,Z).
```

  - initially, $D = \{$ path(a, b). path(b, c) $\}$
  - One instantiation with atoms in D: path(a, c) :- path(a, b), path(b, c).
  - Then, $D = \{$ path(a, b). path(b, c). path(a, c) $\}$, and the algorithm stops.

Safety:

To carry out this 'on demand' grounding, all rules and constraints need to have a property of safety:

- A rule/constraint is called safe if every variable appearing in it appears at least once in the body in an unnegated literal.

Examples:
Safe:

```
a(X,Y,Z) :- b(X,Y), c(X,Z), not d(X,Y,Z)
```

Not safe:

```
a(X,Y,Z) :- b(X,Y) not d(X,Y,Z)
```

Not safe (be careful with negated statements):

```
a(X,Y,Z) :- b(X,Y), X + Y != Z     # Z is unsafe here
```

5 Week 4: More ASP features

More convenient way to sum and max:

Logic Program:

```
#const k=4. num(1,5). num(2,2). num(3,6). num(4,2).

sum(S) :- S = #sum { N, num(I,N) : num(I, N) }
max(S) :- M = #max { N, num(I,N) : num(I, N) }

#sum/1. #max/1.
```

Projected answer sets:

```
sum(15) max(6)
```

Min and count (count is unique, other_count not):

```
#const k=4. num(1,5). num(2,2). num(3,6). num(4,2).

min(S) :- S = #min { N, num(I,N) : num(I, N) }
count(S) :- S = #count { N : num(I,N) }
other_count(S) :- S = #count { N, num(I,N) : num(I, N) }


#min/1. #count/1. #other_count/1.
```

Projected answer sets:

```
min(2) count(3) other_count(4)
```

Computing degree of a graph:

Suppose we have an undirected graph $G = (V, E)$, represented as follows:

```
node(1..5).
edge(1,2). edge(1,3). edge(2,4).
...
edge(X,Y) :- edge(Y,X).
```

- The degree of a node in the graph is the number of nodes that it is connected to.
- The degree of the graph is the maximum degree of any node in the graph.

2 ways:

```
degree(N, D) :- node(N), D = #count { M : edge(N, M) }
degree(D) :- D = #max { E : degree(N,E), node(N) }
```

Alternative correct way:

```
degree(N, D) :- node(N), D = #count {edge(N,M) : edge(N, M) }
degree(D) :- D = #max { E : degree(N,E) }
```

Warning on using aggregates:
They are translated by clingo to rules involving regular predicates. The larger the numbers involved, the more rules are needed for the translation, which takes a lot of memory (impacts efficiency of solver). Only use if other constructions provide no solution.

5.1 Optimization

These statements allow to select between different answer sets. Where we can express what is to be maximized or minimized. Optimal answer sets are those that maximize or minimize this criterion.

```
#minimize { N : something(N) }
```

expresses that the sum of all N is to be minimized for which something(N) is true (similarly for #maximize).

Be careful: the part after #minimize/#maximize is a set. So they do not contain duplicates (only counted once). For example:

```
item(1..3).
cost(1,3). cost(2,3). cost(3,2).
2 { choose(I) : item(I) } 2.
#minimize { C,I : choose(I), cost(I,C) }
```

Projected answer sets:

```
choose(1) choose(3)
choose(2) choose(3)
```

Finding all minimum-size dominating sets of an undirected graph G:

A dominating set of an undirected graph $G = (V, E)$ is a subset $D \subseteq V$ of nodes that dominates all nodes in the graph: for each $v \in V$ either (1) $v \in D$ or (2) $\{u, v\} \in E$ for some $u \in D$.

Using the 4 steps again (with an addition of optimization):

1. **Formalize the problem**
2. **Encoding of problem instances**:
   a) State the nodes of the graph using node/1, e.g.:
   ```
   node(1..5).
   ```
   b) State the edges of the graph using edge/2, e.g.:
   ```
   edge(1,2). edge(1,3). edge(2,4).
   ```
   c) State that all edges are also reversed:
   ```
   edge(X,Y) :- edge(Y,X).
   ```
3. **Encoding of candidate solutions (generate)**:
   a) Use a choice rule to generate all subsets of nodes, using choose/1:
   ```
   { choose(N) : node(N) }.
   ```
4. **Encoding of solution properties (test)**:
   a) Define when a node is dominated:
   ```
   dominated(N) :- choose(N).
   dominated(N) :- choose(M), edge(N,M).
   ```
   b) Require that there are no undominated nodes:
   ```
   :- node(N), not dominated(N).
   ```
5. **Add optimization statements**:
6. Define when a node is dominated:
   ```
   #minimize { 1, choose(N) : choose(N) }
   ```

Recursion:

Suppose we have an undirected graph $G = (V, E)$, represented as follows:

```
node(1..5).
edge(1,2). edge(1,3). edge(2,4).
...
edge(X,Y) :- edge(Y,X).
```

Always start by having a **basecase**. Check whether a node is reachable from another, by some path (**recursively**):

```
reachable(N,N) :- node(N).
reachable(N,M2) :- reachable(N,M1), edge(M1,M2).
```