Artificial Intelligence Fundamentals

Federico Matteoni

A.A. 2021/22

Index

0.1 Introduction . . . 2
0.2 CSP . . . 8
0.3 Knowledge Based Systems . . . 17
0.3.1 Knowledge Representation and Reasoning . . . 17
0.3.2 Nonmonotonic reasoning . . . 23
0.3.3 Knowledge and Beliefs . . . 26
0.3.4 Autoepistemic Logic . . . 30
0.3.5 Graph representation and structured representation . . . 30
0.3.6 Object Oriented Representations and Frames . . . 32
0.3.7 Description Logics . . . 34
0.4 Reasoning Under Uncertainty . . . 35
0.4.1 Quantifying Uncertainty . . . 35
0.4.2 Probabilistic Reasoning . . . 37
0.4.3 Probabilistic Reasoning Over Time . . . 40
0.5 Rule-Based Systems . . . 44
0.5.1 Forward Reasoning . . . 45
0.5.2 Backward Reasoning . . . 47
0.5.3 Forms of Reasoning . . . 49
0.5.4 Prolog . . . 51
0.5.5 Constraint Logic Programming . . . 53
0.5.6 Meta Interpreters . . . 53
0.5.7 Answer Set Programming . . . 53
0.6 Planning . . . 56
0.6.1 Classical Planning . . . 56
0.6.2 Planning Graph . . . 60
0.6.3 From PDDL to Boolean SAT . . . 64
0.6.4 Partial Order Planning . . . 64

0.1 Introduction

Prof.: Maria Simi, Vincenzo Lomonaco
AI is taking over the world. Formalizing common sense is a lot more difficult. We can formalize knowledge in very specific and small domains. But is deep learning the final solution to AI? "*It will transform many industries, but it's not magic. Almost all of AI's recent progress is based on one type of AI, in which some input is used to quickly generate simple response.*" (Andrew Ng)
This AI can do supervised learning, but requires huge amount of data (tens of thousands of pictures to build a photo tagger, for example). The rule of thumb of Ng is: if a person can do a mental task with less than one second of thought, we can automate it using AI either now or in the near future.
Software is not a problem, the community is open and the software can be replicated. So the challenges are:

> **Data is exceedingly difficult to get access to**. Data is the defensible barrier for many businesses
>
> **Talent**, because downloading and applying open-source software to your data won't work. AI needs to be customized to context and data, that's why there's a war for the scarce AI talent that can do this work.
>
> **Computational resources** are also very important.

Deep Learning DL is only one approach inside the much wider field of ML and ML is only one approach in the wider field of AI. Book: *Thinking Fast and Slow*, Kahneman. Two systems: system 1 does perceptual tasks, simple computations, system 2 instead does complex computation, recalling from memory... this is a distinction in our brains.

Machine Learning Is AI all about machine learning? Possible arguments against ML are:

> Explanation and accountability: ML systems are not (yet?) able to justify in human terms their results. For some applications this is essential: must knowledge be meaningful to humans to be able to generate explanations? Some regulations requires the right to an explanation in decision-making, and seek to prevent discrimination based on race, opinions, sex... (see GDPR)
>
> ML systems learn what's in the data, **without understanding what's true or false, real or imaginary, fair or unfair**. It is possible to develop unfair, bad models. People are generally more critical about information.

Building AI systems is a goal far from being solved, still quite challenging. Complex AI systems requires the combination of several techniques and approaches, not only ML.

AI Fundamentals Is mostly about reasoning and *slow thinking*. Different approaches, "good old-fashioned artificial intelligence" or "symbolic AI": teaching about the foundations of the discipline, now 60 years old.

Symbolic AI High-level human readable representations of problems, the general paradigm of searching for a solution, knowledge representation and reasoning, planning. Dominant paradigm from the mid 1950s until late 1980s.
Central to the building of AI systems is the **Physical Symbol Systems Hypothesis** (**PSSH**), formulated by Newell and Simon (*Computer Science as Empirical Inquiry: Symbols and Search*)
The approach is based on the assumption that many aspects of intelligence can be achieved by the manipulation of symbols (the PSSH): *a physical symbol system has the necessary and sufficient means for general intelligent action.*
Human thinking is a king of symbol manipulation system (so a symbol system is **necessary** for intelligence), and machine can be intelligent (a symbol system is **sufficient** for intelligence). This cannot be proven, we can only collect empirical evidence: observation and experiments on human behavior in tasks requiring intelligence, and solving tasks of increasing complexity.

Strong and Weak AI The Chinese room argument, by John Searle, introduced the following distinction: strong AI relies on the *strong* assumption that human intelligence can be reproduced in all its aspects (general AI), including adaptivity, learning, consciousness..., while weak AI is the simulation of human-like behavior, without effetctive thinking or understanding, no claim that it works like the human mind. The latter is the dominant approach today, fragmented AI.
One strong argument against strong AI is the lack of needs by the systems: biological needs, safety, relationships, self esteem, self-actualization (**Maslow's hierarchy of needs**).
What stands in the way of all-powerful AI is not a lack of smarts: it's that computers don't have needs, cravings or desires.

Artificial Intelligence is the enterprise of building intelligent computational agents

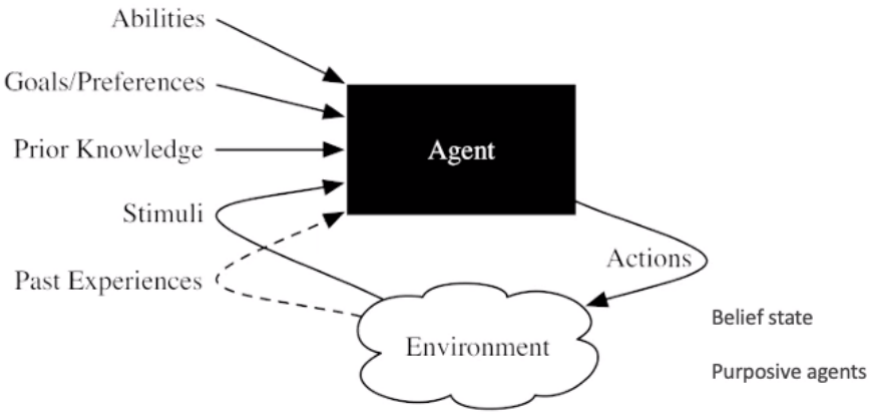
Agents An agent is something that acts in an environment. We are interested in what an agent does, that is: **how it acts**. We judge an agent by its action. An agent acts intelligently when: what it does is appropriate given the circumstances and its goals, it is flexible to changing environments and changing goals, learns from experience, makes appropriate choices given its perceptual and computational limitations.
Computational agent is an agent whose decisions about its actions can be explained in terms of computation and implemented on a physical device.

Scientific Goal: understand the principles that make intelligent behavior possible in natural or artificial systems

Engineering Goal: design and synthesis of useful, intelligent artifacts, agents that are useful in many applications

Artificial Intelligence Artificial intelligent is not the opposite of real intelligence. Intelligence cannot be *fake*: if an artificial agent behaves intelligently, it is intelligent. It is only the external behavior that defines intelligence, according to the **Turing Test** (weak AI). So **artificial intelligence is real intelligence created artificially**.
More sophisticated and upgraded tests are the **Winograd schemas**.
Human intelligence: biology (surviving various habitats), culture (language, tools, concepts, wisdom passed from parents and teachers to children) and life-long learning experience (learning throughout life). Another form is social intelligence, exhibited by communities and organizations.

So agents are situated in environments, inputs are abilities, goals, prior knowledge, stimuli and past experiences, and outputs actions which affect the environment.



Design process

design time computation, that goes into the design

offline computation, that the agent can do before acting in the world (ex: specializing the model)

online computation, done by the agent that is acting

Designing an intelligent agent that can adapt to complex environments and changing goals is a major challenge. Two strategies: simplify environments and build strong reasoning systems for these simple environments, or build simple agents for natural/complex environments simplifying the task.
Steps in the design process:

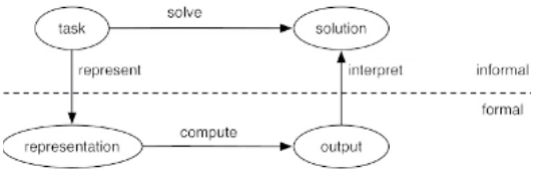
Define the task in natural language, what need to be computed

Define what is a solution and its qualities: optimal, satisfying, approximately optimal, probable...

Formal representation for the task, choosing how to represent knowledge for the task, including representations suitable for learning.

Compute an output

Interpret output as solution



Levels of abstraction A model of the world is a symbolic representation of the beliefs of the agents. It is necessarily an abstraction: more abstract representations are simpler and human-readable but they may not be effective enough. Low level descriptions are more detailed and accurate but more complex too. Multiple level of abstractions are possible (hierarchical design). Two levels always present in the design: knowledge level (what the agent knows and its goals, not in terms of how we represent) and the symbol level (internal representation and reasoning system).
Modularity is the extent to which a system/task can be decomposed:

Flat: not modular

Modular: interacting modules that can be understood on their own

Hierarchical: modules are decomposed into simpler modules

Planning horizon is how far ahead in time the agent plans

Non planning agent

Finite horizon planner: looks for a fixed amount of stages, greedy if only one step ahead

Indefinite horizon planner: finite but not predetermined number of stages

Infinite horizon planner: keeps planning forever (ex: stabilization module of a legged robot)

Representation concerns how the state of the world is described

Atomic states

Feature-based representation: set of propositions that are true or false (PROP, CSP, most ML)

Individuals and relations, or **relational representation**

Computational limits that determines whether an agent has

Perfect rationality, reasons about the best actions without constraints

Bounded rationality, decides on the best action that it can find given its limits

An **anytime algorithm** is an algorithm where the solution improves with time. The more it computes, the more accurate is the solution it provides.
Learning dimensions determines whether

knowledge is **given in advance**, or

knowledge is **learned** (from data or past experience)

Learning typically means finding the best model that fits the data and produces a good predictive model.
Uncertainty, which can be

in **sensing** (fully/partially observable states)

or about the **effects** of the actions (deterministic/stochastic)

Preference is the dimension which considers whether the agent has

Goal (achievement goal a proposition true in a final state, or maintenance goal, proposition true in all possible states)

Complex preferences, involving trade-offs among the desirability of various outcomes, perhaps at different times. **Ordinal preference** when only the ordering of the preferences is important, while **cardinal preference** where the magnitude of the values matters. States are evaluated by **utility functions**.

Number of agents

Single agent reasoning

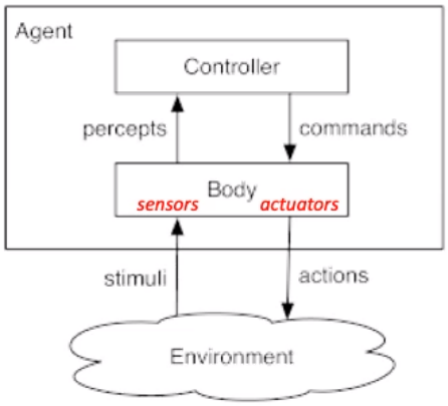
Multi agent reasoning

Interaction considers whether the agent does:

Offline reasoning or

Online reasoning

Agent Architectures An agent interacts with an environment, receives informations with sensors and acts in the world with actuators. Robot: physical body. Program: software agent, digital environment.
Agent is made of body and controller (brain of the agent), which receives percepts from the body and sends command to the body. A body includes sensors that converts stimuli into percepts and actuators that convert commands into actions. Bot sensors and actuators can be a source of uncertainty.



Agents act in time. T is a set of time points which starts at 0, totally ordered, discrete and each t has a next time $t+1$.

Percept trace/stream: function of time into percepts (past, present, future)

Command trace: a function of time into commands (past, present, future)

History at time t: percepts up to t and commands up to $t-1$

Causal Transduction Function from history to commands. "Transduction" comes from *finite state transducers*, where both new states and commands are emitted. "Causal" because only previous and current percepts and previous commands can be considered. A controller ideally implements a causal transduction.
But complete history is usually unavailable, only the memory of it. The belief state of an agent at time t is all the information that the agent remembers from the previous times. The behavior of an agent can be described by two functions:

Belief State function $remember : S \times P \to S$ with S being the set of belief states and P the set of percepts

Command function $command : S \times P \to C$ with C being the set of commands.

The controller implements both, an approximation of the causal transduction.

Problem Solving as search The dominant approach to AI is formulating a task as a search in a state space. The paradigm is as follows:

Define a goal (a set of states, a boolean test function...)

Formulate the task as a search problem: define a representation for states and define legal actions and transition functions

Find a solution (a sequence of actions) by means of a search process

Execute the plan

This is a basic technique in AI: search happens inside the agent, it's the planning stage before acting. It's different from searching the world, when an agent may have to act in the world and interleave an action with planning.
Search is a general paradigm, underlying much of the artificial intelligence field. An agent is usually given only a description of what it should achieve, not an algorithm to solve it. The only possibility is to search for a solution. Searching can be computationally very hard (NP-Complete).
Humans are able to solve specific instances by using their knowledge about the problem. This extra knowledge is called **heuristic knowledge**.

Assumptions in classic problem solving Problem solving agents are goal driven agents, that work under simplified assumptions made in the design process.

States are treated as black boxes: we only need to know the heuristic value and whether they are a goal by applying the boolean goal function. The internal structure doesn't matter from the point of view of search algorithms. **Atomic representations**.

The agent has **perfect knowledge** of the state (full accessibility), no uncertainty in sensors.

Actions are **deterministic**, so that the agent know the consequences of its actions.

The state space is generated incrementally: can be infinite, so may not fit in memory.

Problem formulation A problem is defined formally by five components:

Initial state

Possible actions in state s, $Actions(s)$

Transition model: a function $Result : State \times Action \to State$
$Result(s, a) = s'$, a **successor state**

Goal states are defined by a boolean function
$Goal\text{-}Test(s) \to \{true, false\}$

Path-cost function, that assigns a numeric cost to each path. The sum of the cost of the actions on the path $c(s, a, s')$

Graphs for searching A (directed) graph consists of a set N of nodes and a set A of arcs, which are ordered pairs of nodes. Node n_2 is a neighbor/successor of n_1 if $\exists\,(n_1, n_2) \in A$, and a path is a sequence of nodes $(n_0, \ldots, n_k)$ such that $(n_{i-1}, n_i) \in A$ with length k.
The cost of the path is the sum of the costs of its arcs $cost((n_0, \ldots, n_k)) = \sum_{i=1}^{k} cost((n_{i-1}, n_i))$.
A **solution** is a path from a start node to a goal node and an **optimal solution** is one with minimum cost.

Search algorithms A problem is given as input to a search algorithm. A solution to a problem is a path (actions sequence) that leads from the initial state to a goal state.
Solution quality is measured by the path cost function and an optimal solution has the lowest path cost among all solutions. Different strategies (algorithms) for searching the state space may be characterized by:

Their time and space complexity, completeness, optimality...

Uninformed search methods vs informed/heuristic search methods, which use an heuristic evaluation function of the nodes

Direction of search (forward or backwards)

Global vs local search methods

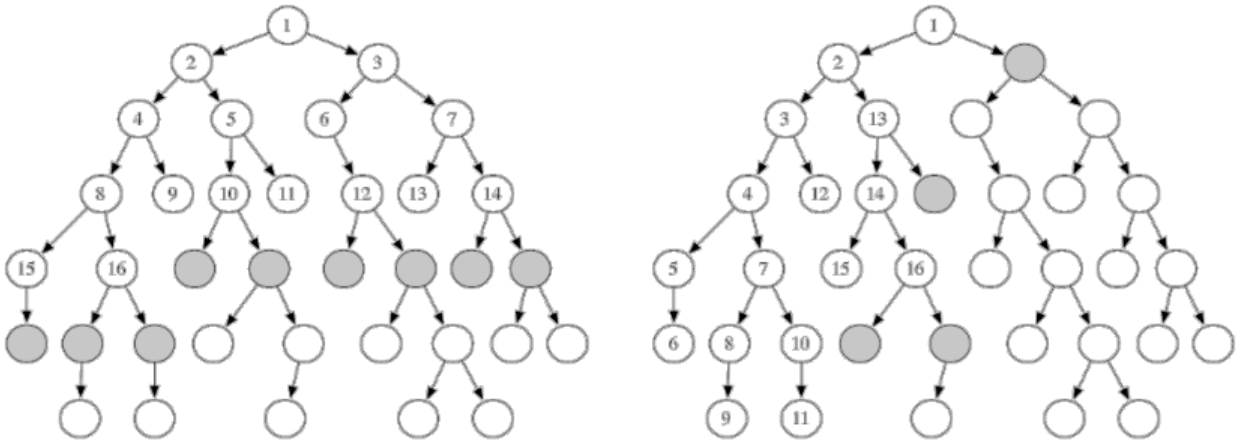
Generic search algorithm

```
input:
a graph
a set of nodes
boolean function goal(n) that tests if n is a goal node

frontier := {s | s is a start node}
while frontier is not empty:
select and remove path (n0, ..., nk) from frontier
if goal(nk):
return (n0, ..., nk)
for each neighbor n of nk
add (n0, ..., nk, n) to frontier
end while
return fail
```

Other algorithms With b max number of successors, d depth of solution and m max distance of solution.

Breadth and Depth search Respectively:



Breadth search: complete (if a solution exists, it finds it), optimal (if it finds a solution, it's the one with minimum cost), time $O(b^d)$, space $O(b^d)$
Depth search: not complete, time $O(b^m)$, space $O(bm)$

Depth bounded search: supposes to know the distance of the solution, performs depth-first up to a limit without giving up completeness

Iterative deepening: tries depth limit 1, then 2, then 3 and so on, freeing memory from one iteration to the next.

Uniform cost search: at each stage, selects a path on the frontier with lowest cost.
The frontier is priority queue ordered by path cost, so the first path to goal is the least-cost path. When arc costs are equal is equivalent to breadth-first search.
This strategy is complete, provided that the branching factor is finite and there is some $\epsilon > 0$ such that all the costs are $> \epsilon$. It's also optimal, since it guarantees that the paths with lower costs are found first.

Heuristic search The idea is to not ignore the goal when selecting the paths. Often there's extra knowledge that can be used to guide the search: **heuristics**, provided by an heuristic function $h : N \to R \Rightarrow h(n)$ is the estimate of the cost of the shortest path from node n to the goal node.
h needs to be efficiently to compute. An **admissible heuristic** $h^*(n)$ is a non-negative heuristic function that **under-estimates** the minimum cost of a path from a node n to a goal: $\forall\, n \;\; h*(n) \leq \text{cost}(n)$

Best-first search selects the most promising node on the frontier according to the heuristic function.

A^* **search** With an heuristic function in the form $f(n) = g(n) + h(n)$ with:

$g(n)$ being the cost of path leading to n (so the previous path up until n, $cost(n)$)

$h(n)$ is an admissible heuristic (so, $h(n) \geq 0$)

Then $f(n)$ estimates the total path cost of going from a start node to a goal via n. The special cases are $h = 0$ (lowest cost search) and $g = 0$ (greedy best first).
Properties of A^*:

Complete

Always finds an optimal solution, if the branching factor is finite and arc costs are bounded above 0 (which means that $\exists\, \epsilon > 0 \mid$ arc costs are $> \epsilon$)

Optimiziations are possibile when searching graphs

They operate some sort of graph pruning:

Cycle pruning: doesn't add nodes to the frontier with states already encountered along the path (easy)

Multiple-path pruning: maintains an explored set of nodes that are at the end of paths that have been expanded. When an n is selected, if its state is already in the explored set, it's discarded.

Memory requirement is exponential ($O(b^d)$). Can be mitigated in some ways:

IDA^*: performs repeated depth-bounded searches with value of $f(n)$ used as bound

Recursive best-first, similar to branch & bound

SMA^* (simplified memory-bounded A^*)

Beam search, keeps in frontier only the best k paths, with k being the beam width (gives up optimality)

Consistent heuristics An heuristic that statisfies the monotone restriction guarantees consistency $h(n) \leq cost(n, n') + h(n')$
Consistency $\Rightarrow$ admissibility. With the monotone restriction, the f-values of the paths selected from the frontier are monotonically non-decreasing.

Features Often better to describe states in terms of features: **factored representation**, more natural and efficient than explicitly enumerating states. Often, features are not independent and there are constraints that specify legal combinations of assignments. We can exploit these constraints to solve tasks.
Constraint satisfaction is about generating assignments that satisfy a set of hard constraints and how to optimize a collection of soft constraints (preferences).

0.2 CSP

Constraint Satisfaction Problem, formal definition. A Constraint Satisfaction Problem $CSP = \langle X, D, C\rangle$ consists of three components:

A finite set of **variables**, $X = \{x_1, \ldots, x_n\}$

A **finite domain** for each variable, $D = \{D_1, \ldots, D_n\}$ with each $D_i = \{v_1, \ldots, v_k\}$ containing values assignable to x_i.
Dom is a function that maps every variable in X to a set of objects of arbitrary type. $Dom(x) = D_x$

A **set of constraints** that restrict the values the variables can simultaneously take, C

Task: assign a value from the associated domain to each variable satisfying all the constraints. **NP-hard** in worst cases, but general heuristics exist and structures can be exploited for efficiency.

A **(partial) assignment** of values to a set of variables (**compound label**) is a set of pairs $A = \{\langle x_i, v_i\rangle, \ldots\}$ with $v_i \in D_{x_i}$. A **complete assignment** is an assignment to all the variables of the problem. Can be projected to a smaller partial assignment by restricting the variables to a subset (projection, with the following notation: $\pi_{x_1,\ldots,x_k} A$, with π being the projection operator of relational algebra)
Each constant in C can be represented as a pair ⟨scope, rel⟩: scope is a tuple of variables participating in the constraint,

and rel is a relation that defines the allowable combinations of values for those variables, taken from the respective domains. The relation can be represented as: an explicit list of all tuples of values that satisfy the constraint (explicit relation), or an implicit relation (an object that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation).
We also use $C_{x_1,\ldots,x_k} = \text{rel}$ to denote a constraint with scope $= x_1, \ldots, x_k$, so the constraint $C = \langle (x_1, \ldots, x_k), \text{rel} \rangle$

CSP solution To solve a CSP problem seen as a search problem, we need to define a state space and the notion of a solution.

> **State**: assignment of values to some or all the variables.
> Partial if values are assigned only to some of the variables, complete if every variable is assigned.
>
> **Solution**: a complete and consistent assignment.
> An assignment is consistent if it satisfies all the constraints: $\mathit{Satisfies}(\{\langle x_1, v_1 \rangle, \ldots, \langle x_k, v_k \rangle\}, C_{x_1,\ldots,x_k})$ for any constraint in C

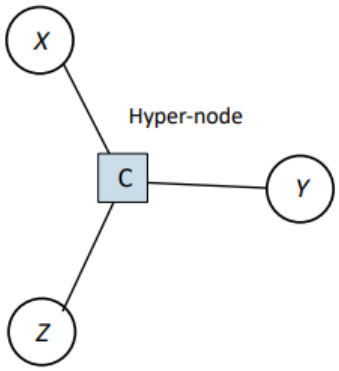
Problem characteristics

> Number of solutions required (one or all)
>
> Problem size (number of variables and constraints)
>
> Type of variables and constraints
>
> Structure of the constraint graph
>
> Tightness of the problems (measured in terms of the solution tuples over the number of all distinct compund labels of all variables)
>
> Quality of solutions
>
> Partial solutions

CSP solving techniques Problem reduction techniques/inference/constraint propagation: techniques for transforming CSP into an equivalent problem easier to solve or recognizable as insoluble.
Searching efficiently: heuristics, intelligent backtracking...
Exploiting the structure of the problem: independent sub-problems, tree structured constraints, tree decomposition, exploiting symmetry...

Constraint hyper-graphs Binary CSP = CSP with unary and binary constraints only. May be represented as an undirected graph (V, E): nodes corresponds to variables V and edges corresponds to binary constraints between variables ($E = V \times V$). Edges are undirected arcs (can be seen as pair of arcs).
Node x is adiacent to node y is $(x, y) \in E$. A graph is connected if there's a path among any two nodes.

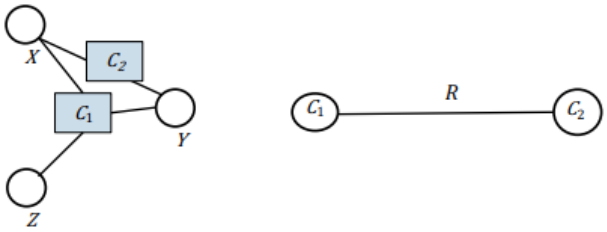
In general, every CSP is associated with a constraint hyper-graph, a generalization of graphs: an hyper-node may connect more than two nodes. The constraint hyper-graph of a CSP $\langle X, D, C \rangle$ is a hyper-graph in which each node represent a variable in C and each hyper-node represents a higher order constraint in C



Dual Graph transformation Alternate way to convert a n-ary CSP to a binary one:

1. Create a new graph in which there is one variable for each constraint in the original graph
2. If two constraints share variables, they are connected by an arc corresponding to the constraint that the shared variables receive the same value

Example: $Dom(x) = Dom(y) = Dom(z) = \{1, 2, 3\}$ with $C_1 = \{(x, y, z), x + y = z\} = \{(1, 2, 3), (2, 1, 3), (1, 1, 2)\}$ and $C_2 = \{(x, y), x < y\} = \{(1, 2), (1, 3), (2, 3)\}$. This will become $Dom(C_1) = \{(1, 2, 3), (2, 1, 3), (1, 1, 2)\}, Dom(C_2) = \{(1, 2), (1, 3), (2, 3)\}$ and $R_{x,y}$ = constraint that x and y will receive the same values



Related concepts

Problem reduction techniques: techniques for transforming CSP into an equivalent problem easier to solve or recognizable as insoluble

Enforcing local consistency: the process of enforcing local consistency properties in a constraint graph causes inconsistent values to be eliminated. Different types of local consistency properties have been studied.

Constraint propagation/inference: constraints are used to reduce the number of legal values for a variable, which in turn can reduce the legal value for another variable and so on...

Problem reduction Reducing a problem means removing those constraints which appear in no solution tuples. A CSP problem P_1 is reduced to P_2 when P_1 is equivalent to P_2, domains of variables in P_2 are subsets of those in P_1 and the constraints in P_2 are at least as restrictive as those in P_1.
These conditions guarantees that a solution in P_2 is also a solution in P_1.
Problem reduction strategies are of two types: removing redundant values from the domains of the variables or tightening the constraints so that fewer compound labels satisfy them (examples: if $x < y$ and $D_x = \{3, 4, 5\}, D_y = \{1, 2, 4\}$ then those can be reduced to $D_x = \{3\}, D_y = \{4\}$.
Constraints are sets, this means removing redundant compound labels from the set. If the domain of any variable or any constraint is reduced to an empty set, then the problem is **unsolvable**.
Problem reduction is also called consistency checking or maintenance, since it relies on establishing local consistency properties.

Local consistency properties: node consistency, arc consistency, path consistency, k-consistency, forward checking.
All these operations do not change the set of solutions, do not necessarily solve a problem but, used with search, will make the search more efficient by pruning the search tree.

Node/Domain consistency: a node is consistent if all the values in its domain satisfy unary constraints on the associated variable. A constraint network is node-consistent if all the nodes are consistent.
Given a unary constraint on x_i, $C_i = \langle (x_i), R_i \rangle$ then node consistency $D_i \subseteq R_i$ and can be enforced by reducing the domains of the variables $D_i \leftarrow D_i \cap R_i$ (this is the NC-1 algorithm, $O(d \cdot n)$)

Arc consistency: a variable is arc-consistent if every value in its domain satisfies the binary constraints of this variable with another variable. x_i is arc-consistent with respect to x_j if for every value in its domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (x_i, x_j)
Example, $X = \{x, y\}$, $D_x = D_y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and constraint $\langle (x, y), x = y^2 \rangle$. Considering arc $x \rightarrow y$, it can be made consistent by reducing D_x to $\{0, 1, 4, 9\}$. Considering $y \rightarrow x$, can be made consistent by reducing D_y to $\{0, 1, 2, 3\}$, making the entire edge consistent.

Arc Consistency Algorithm (AC-3) It maintains a queue of arcs to consider, initially all the arcs in CSP. An edge produces two arcs. AC-3 pops off an arc (x_i, x_j) from the queue and makes x_i arc-consistent respect to x_j:

If this step leaves D_i unchanged, the algorithm just moves on to the next arc

If D_i is made smaller, then we need to add to the queue all arcs (x_k, x_j) where x_k is a neighbor of $x_i \neq x_j$

If D_i becomes empty, then we conclude that the CSP has no solution

When there are no more arcs to consider, we have finished: we are left with a CSP equivalent to the original but smaller.
With a CSP of n variables, each with domain size at most d and c binary constraints (arcs):

Checking consistency of an arc can be done in $O(d^2)$ time

Each arc (x_i, x_j) can be inserted in the queue only d times (because x_i has at most d values to delete)

We have c arcs to consider, so complexity is $O(c \cdot d^3)$, polynomial time

The AC-4 algorithm is an improved version of AC-3, based on the notion of support that doesn't need to consider all the incoming arcs. More information is kept, but complexity is $O(c \cdot d^2)$.

Example $A, B, C \in \{1, 2, 3, 4\}, A < B, A > C$

Queue	Arc	Arc domain
$\{(A,B),(B,A),(A,C),(C,A)\}$		
$\{(B,A),(A,C),(C,A)\}$	(A,B)	$A \in \{1,2,3,\not{4}\}$
$\{(A,C),(C,A)\}$	(B,A)	$B \in \{\not{1},2,3,4\}$
$\{(C,A)\}$	(A,C)	$A \in \{\not{1},2,3\}$
$\{(B,A),(C,A)\}$		
$\{(C,A)\}$	(B,A)	$B \in \{\not{2},3,4\}$
$\{\}$	(C,A)	$C \in \{1,2,\not{3},\not{4}\}$

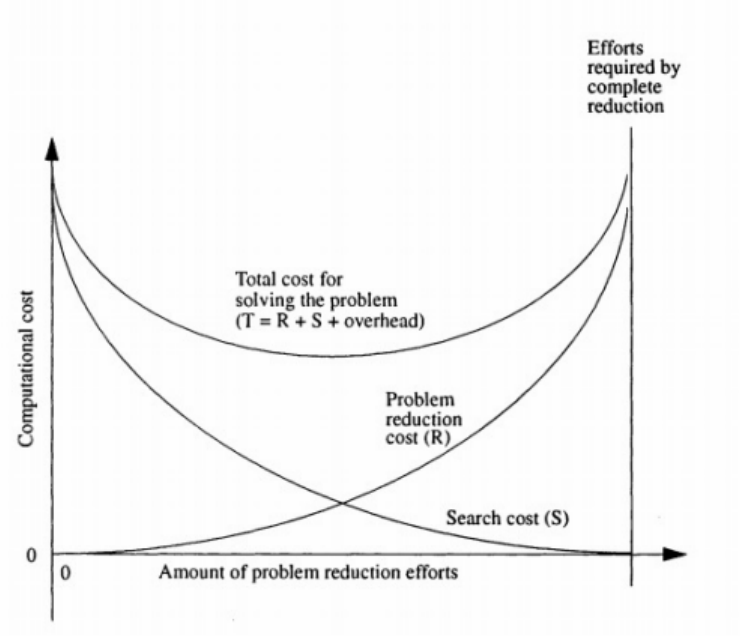
At the end $A \in \{2,3\}, B \in \{3,4\}, C \in \{1,2\}$

Directional Arc Consistency DAC is defined with reference to a total ordering of the variables. A CSP is DAC under an ordering of the variables if and only if for every label $\langle x, a\rangle$ which satisfies the constraints on x there exists a compatible label $\langle y, b\rangle$ for every label y which is after x according to the ordering.
In the algorithm for establishing DAC (DAC-1) each arc is examined exactly once, by proceeding from last to the first in the ordering, so the complexity is $O(c \cdot d^2)$. AC cannot always be achieved by running DAC-1 in both directions.

Generalized Arc Consistency GAC, extension of AC-3 to handle n-ary constraints rather than just binary. A variable x_i is GAC with respet to a n-ary constraint if for every value v in the domain of x_i there exists a tuple of values that is a member of the constraint and has its x_i component equal to v.
For example, if $X, Y, Z \in \{0, 1, 2, 3\}$ and $X < Y < Z$, to make C consistent we would have to eliminate $2, 3$ from the domain of X because the constraint cannot be satisfied with $X = 2$ or $X = 3$.

Path Consistency Arc consistency tightens down the domains using the arcs (binary constraints). Path consistency is a stronger notion: tightens the binary constraints by using implicit constraints that are inferred by looking at the triples of variables. A path of length 2 between variables x_i, x_j is path-consistent with respect to a third intermediate variable x_m if for every consistent assignment $\{x_i = a, x_j = b\}$ there is an assignment to x_m that satisfies the constraints on (x_i, x_m) and (x_m, x_j). In relational algebra $R_{i,j} \subseteq \pi_{i,j}(R_{i,m} \bowtie D_m \bowtie R_{m,j})$
To achieve path consistency, $R_{i,j} \leftarrow R_{i,j} \cap \pi_{i,j}(R_{i,m} \bowtie D_m \bowtie R_{m,j})$ (algorithm name: PC-2). If all path of length 2 are made consistent, then all paths of any length are consistent.
Called path consistency because you can think of it as a path from x_i to x_j with x_m in the middle.

Combining search and problem reduction Problem reduction techniques are used in combination with search. The more effort one spends on problem reduction, the less effort one needs in searching.



Most problems cannot be resolved by reduction alone, we must search for solutions and combine problem reduction with search. In this context we talk about **constraint propagation** or **inference**. A classical incremental formulation of CSP as a search problem is:

States are partial assignments

Initial state: empty assignment

Goal state: complete assignment that satisfy all constraints

Actions: assign to a specific unassigned variable x_i a value $\in D_i$

Branching factor: d, with d maximum cardinality of the domains. Number of leaves: d^n, with n number of variables. n finite $\Rightarrow$ space graph finite.

CSP as search: simplifications We can exploit **commutativity**. A problem is commutative if the order of application of any given set of action has no effect on the outcome. In this case, the order of the variable assignments does not change the result.
We can consider a single variable for assignment at each step, so the branching factor is d and the number of leaves id d^n. We can also exploit depth limited search: **backtracking search** with depth limit n.
Search strategies:

Generate and Test: generate a full solution and test it, not the best

Anticipated Control: after each assignment we check the constraint, if some is violated we backtrack to previous choices (undoing the assignment)

Backtracking search algorithm `todo code`

Heuristics and search strategies

`SELECT-UNASSIGNED-VARIABLE`: which variable should be assigned next?

`ORDER-DOMAIN-VALUES`: in which order should the values be tested?

`INFERENCE`: what inference should be performed at each step?
Techniques for **constraint propagation** (local consistency enforcement) can be used

`BACKTRACKING`: where to back up to? When the search ends up in an assignment that violates a constraint, can the search avoid repeating this failure? Forms of **intelligent backtracking**.

Choosing the next variable

MRV (minimum remaining values): variable with fewest "legal" remaining values

Degree heuristic: variable involved in the largest number of constraints

Choosing value

Least constraint variable: prefer the variable rules out the fewest choices

Note that **in choosing the variable, a fail-first strategy helps in reducing the amount of search** by pruning large parts of the tree earlier. **In the choice of value, a fail-last approach works best in CSP where the goal is to find *any* solution**. This is **not effective if we are looking for all solutions or no solution exists**.

Interleaving search and inference One of the simplest form of inference propagation is **forward checking**: efficient constraint propagation, weaker than other forms. Whenever X is assigned, FC process establishes arc consistency of X for the arcs connecting neighbor nodes. For each unassigned Y connected to X, delete from Y's domain any value inconsistent with the value assigned to X.

Constraint learning When the search is at a contradiction, we know that some subset of the conflict set is responsible. Constraint learning is the idea of finding a minimum set of variables from the conflict set that causes the problem: the no-good set. We record the no-good set either by adding a new constraint to CSP or by keeping a separate cache of no-goods. This way we do not repeat the no-good state.

Local search Requires a complete state formulation, keep in memory only current state to improve it iteratively and does not guarantee to find a solution even if it exists (**not complete**).
Used when space too large for systematic search and we need to be very efficient. Also when we need to provide a solution but it's not important to produce solution path. Also when we know in advance that a solution exists.

Local search methods for CSP Complete state formulation: we start with a complete random assignment, and we try to fix it until all the constraints are satisfied.
Local methods are very efficient for large scale problems where the solutions are densely distributed in the space. A basic algorithm is:

```
def Local_search(V, Dom, C):  # returns a complete & consistent assignment
  # Inputs: V: a set of variables
  # Dom: a function such that Dom(x) is the domain of variable x
  # C: set of constraints to be satisfied
  # Local: A (complete assignment) an array of values indexed by variables in V

  repeat until termination
    for each variable x in V do  # random initialization or random restart
      A[x] := a random value in Dom(x)
      while not stop_walk( ) & A is not a satisfying assignment do  # local search
        Select a variable y and a value w in Dom(y), w != A[y]  # a successor
        A[y] := w  # change a variable
        if A is a satisfying assignment then return A  # solution found
```

this can be specialized, two extreme versions are: **random sampling** (no walking done to improve solution, so `stop_walk` always `true`, just generating random assignments and testing them) or **random walk** (no restarting is done, so `stop_walk` always `false`)

Heuristic Local Search Inject heuristics in the selection of the variable and the value by the means of an evaluation function (in CSP, $f = \#$ violated constraints or conflicts, perhaps with weights).
Iterative best improvement: choose the successor that most improves the current state according to an evaluation function f. Moves to best successor even if worse than current state, may be stuck in loops, not complete.

Stochastic Local Search Adds randomness, escapes local minima:

Random restart is a global random move: the search starts from a completely different part of the search state

Random walk is a local random move: random steps are taken interleaved with the optimizing steps

There are possible variants

Most improving step: selects a variable-value pair that makes the best improvement. Needs to evaluate all of them, needing strategies for efficient computation

Two stage choice: select the variable that participates in most conflicts, then the value that minimizes conflicts (or a random value)

Any conflict: choose a conflicting variable at random, select the value that minimizes conflicts (or a random value)

CSP: Min-Conflict Heuristic All the local search techniques are candidate for CSP, some have proved especially effective. Min-conflict heuristics is widely used and also quite simple:

select a variable at random among the conflicting variables

select the value that results in the **minimum number of conflicts** with other variables

Improvements Usually many plateaus, possible improvements:

Tabu Search: local search has no memory, the idea is keeping a small list of the last t steps and forbidding the algorithm to change the value of a variable whose value was changed recently. This is meant to prevent cycling among assignments, and t is called **tenure**

Constraint Weighting: concentrates the search on important constraints. We assign a numeric weight to each constraint, initially 1. The weight is incremented each time the constraint is violated: goal is choose the variable and value which minimizes the weights of the violated constraints.

Alternatives

Simulated Annealing: allows downhill moves at the beginning of the algorithm and slowly *freezes* this possibility

Population based methods

Local Beam Search: proceed with the k best successors according to the evaluation function

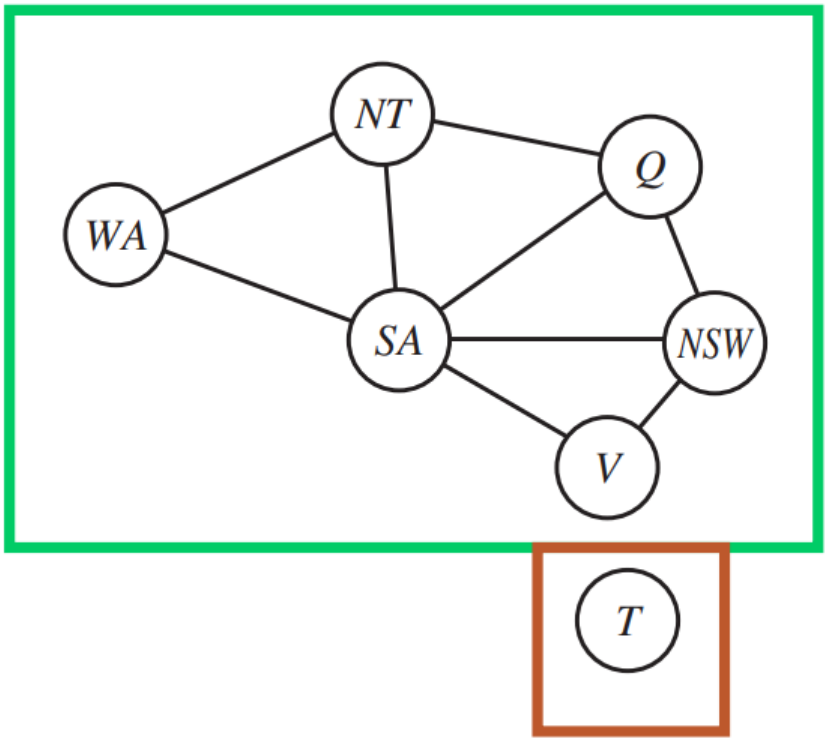
Stochastic Local Beam Search: selects k of the individuals at random with a probability that depends on the evaluation function

Genetic Algorithms...

Online search Another advantage of local search methods is that they can be used in an online setting when the problem changes dynamically.

Evaluating randomized algorithms Randomized algorithms are difficult to evaluate since they output a different result and a different execution time each time they run. We take the runtime distribution, which shows the number of runs in which the algorithm solved the problem within a given number of steps.
A randomized algorithm can be run multiple times with random restart, increasing the probability of success. An algorithm that succeeds with probability p run n times will found a solution with probability $1-(1-p)^n$ with $(1-p)^n$ the probability of failing n times.

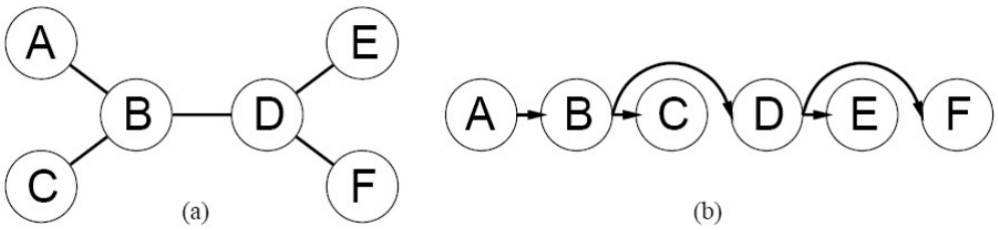
Independent sub-problems When problems have a specific structure, which is reflected in properties of the constraint graph, there are strategies for improving the process of finding a solution. A first obvious case is that of independent subproblems, for examples in the map coloring problem, a country which is not connected to another country (in the image, T) can be colored *independently* from the others, and any solution for that country combined with any solution for the other countries yields a solution for the whole map.



Formally, each **connected component** of the constraint graph corresponds to a subproblem CSP_i. If an assignment S_i is a solution of CSP_i then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$.

Complexity The saving in computational time is dramatic. With n variables and c variables for subproblems, we have $\frac{n}{c}$ independent subproblems. Given d size of the domain, solving one subproblem costs $O(d^c)$ and solving all of them costs $O(d^c \frac{n}{c})$, **linear** in the number of variables n rather than $O(d^n)$ exponential!
Dividing a boolean CSP with 80 variables into 4 subproblems reduces the worst case solution time from the lifetime of the Universe down to less than a second.

The structure of problems: trees



In a tree-structured graph, two node are connected by only one path: we can choose any variable as root of the tree. Chosen a variable as root, for example A, the tree induces a topological sort on the variables: children of a node are listed after their parent.

Directional Arc Consistency A CSP constraint graph is defined to be directionally arc-consistent under an ordering of variables $X_1, \ldots, X_n \Leftrightarrow$ every X_i is arc-consistent with each X_j for $j > i$.
We can make a tree-like graph directionally arc-consisten in one pass over the n variables: each step must compare up to d possible domain values for two variables (so d^2) for a total time of $O(nd^2)$

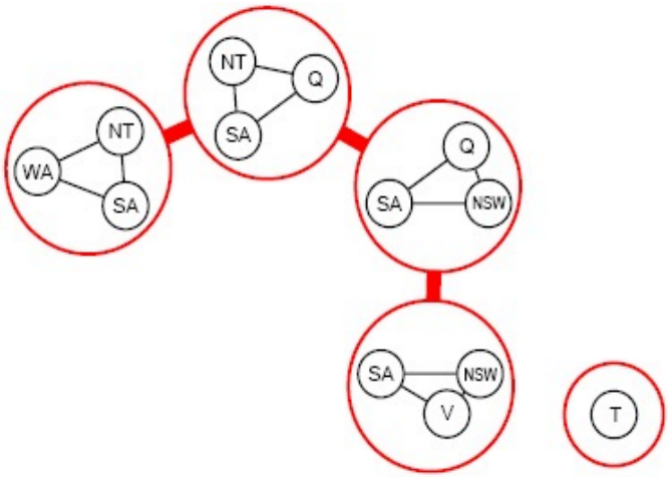
Tree-CSP solver

1. Proceeding from X_n to X_2, make the arcs $X_i \to X_j$ DAC by reducing the domain of X_i if necessary. This can be done in one pass
2. Proceeding from X_1 to X_n, assign values to variables. There's **no need for backtracking** since each value for a father has at least one legal value for the child

Reducing graphs to trees For example, assigning a value to the node we want to remove and removing inconsistent values for other variables, then solve with tree-CSP solver.
In general not easy.

Cutset conditioning Domain splitting strategy, trying with different assignments: choose subset S of CSP's variables such that the constraint graph becomes a tree $\Rightarrow$ S is called **cycle cutset**. For each consistent assignment to variables $\in S$: remove from the domains of remaining vars any inconsistent value, if the remaining CSP has a solution, return it together with the current assignment of S.
Time $O(d^c(n-c)d^2)$ where c is size of the cycle cutset and d size of the domain. We have to try each of the d^c combinations of values for the variables $\in S$ and for each combination we must solve a tree problem of size $(n-c)$

Tree decomposition The approach consists in a tree decomposition of the constraint graph into a set of connected sub-problems. Each of them is solved independently and the resulting solutions are combined cleverly.



Every variable in the original problem must appear in at least one of the sub-problems

If two variables are connected by a constraint, they must appear together in at least one of the subproblems, along with the constraint

If a variable appears in two subproblems of the tree, it must appear in every subproblem along the path connecting those subproblems

1-2 ensure that all variables and constraints are represented. Condition 3 ensure that any given variable must have same value in every subproblem.

Solving a decomposed problem We solve each subproblem independently. If any subproblem has no solution then the original problem has no solution. For putting the solutions together, we solve a "meta-problem" as follows:

Each subproblem is a "mega-variable", whose domain is the set of all solutions for the subproblem.
Es, Dom(X_1) = {(WA=r, SA=b, NT=g), ...}

The constraints ensure that the subproblem solutions assign the same values to the variables they share

The tree width of the decomposition is the size of the largest subproblem -1. Ideally we should find, among many possible ones, a tree decomposition with minimal tree width. NP-hard but heuristics exists.

Symmetry Important factor for reducing the complexity of CSP problems. Value symmetry: the values does not really matter, for example different colors but there are 6 equivalent ways of satisfying the constraints. If S is a solution to coloring n var, then there are $n!$ solutions.
Symmetry breaking constraints: we impose an arbitrary ordering constraints that requires the values to be in alphabetical order. Breaking value symmetry has proved to be important and effective on many problems.

Three approaches

Reformulate the problem so that it has a reduced amount of symmetry, or none at all.

Add symmetry breaking constraints before the search begins, making some symmetric solutions unacceptable while leaving at least one solution in each symmetric equivalence class

Break symmetry dynamically during search

It's an active area of research.

0.3 Knowledge Based Systems

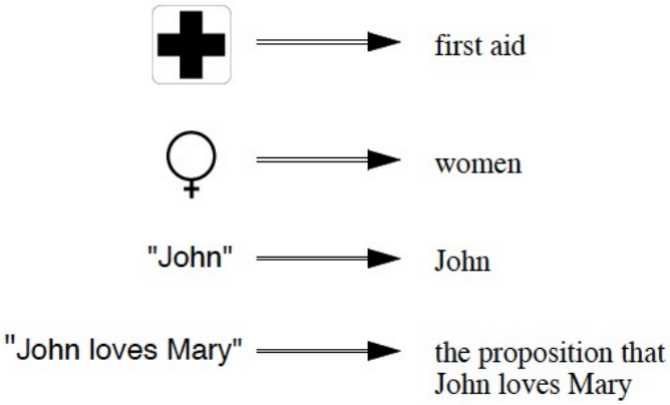
0.3.1 Knowledge Representation and Reasoning

Introducing an additional level of complexity in the representation of states. Knowledge based systems have rich representation languages and the ability to do inference (derive new knowledge). These languages rely on classical logic.
Other representation languages are proposed for inherent limitations in classical logic or for improving efficiency of inference.

Knowledge Representation and Reasoning (KR&R) is the field of AI dedicated to representing information about the world in a form that a computer system can use to solve complex tasks. The class of systems that derive from this is called **knowledge-based** systems/**agents**. A knowledge-based agent maintains a knowledgebase (KB) of facts expressed in a declarative language, and is able to perform automated reasoning to solve complex tasks.
The KB is the agent's representation of the world which is responsible for its intelligent behavior.

We will deal on how knowledge is represented and reasoned about, to derive new knowledge or decide actions. A separate important issue is how knowledge is acquired: hardcoded, obtained automatically... and how it evolves maintaining its anchorage to the world.

Knowledge What kind of knowledge? The emphasis is the relation between an agent and **facts that may be true or false in the world**. With p proposition, something true or false: "John knows p", "John believes p", "John desires p", "John is confident that p"...
Contrast with non-factual knowledge: knowing how ("John knows how to play the piano"), when, where ("John knows where the party is"), a person ("John knows Bill very well")...
Represented knowledge is given a propositional account. Knowledge representation is about the use of formal symbolic structures to represent a collection of propositions, believed by some agent. Not necessarily all of them.
A representation is a surrogate.



Reasoning Is the formal manipulation of the symbols representing a collection of beliefs, to produce representations of new ones. Analogy with arithmetic. **Logical deduction** is a very well known example of reasoning.
Why reasoning? We would like the system to depend on what it believes and not what was explicitly stored. It's a question of economy of the representation.
Usually we need more than just DB-styled retrieval of facts in the KB. Explicit and implicit beliefs, logical entailment (KB $\vDash \alpha$).

Other forms of reasoning: **abductive reasoning** (given a causal relation $a \Rightarrow b$, from observing b we can conjecture a: it's a way of providing an explanation) or **inductive reasoning** (from specific observation to a general rule)

The Knowledgebase Representation hypothesis *Any mechanically embodied intelligent process will be comprised of structural ingredients that*

> *we, as external observers, naturally take to represent a propositional account of the knowledge that the overall process exhibits, and*
>
> *independent of such external semantic attribution play a formal but causal and essential role in engendering the behavior that manifests that knowledge*

Brian C. Smith, 1985

Putting it simply, we want AI systems that contain symbolic representations with two important properties:

> We can understand those symbolic structures as propositions
>
> These symbolic structures determine the behavior of the system

Knowledge based systems have these properties.

Competing approaches: **procedural approach** (knowledge is embedded in programs) and **connectionist approach** (avoids symbolic representation and reasoning, instead models intelligent behavior by computing with networks of weighted links between artificial neurons)

Advantages of KB systems

> They try solving **open-ended tasks**, not pre-compiled kind of behavior for specific tasks
>
> **Separation** of knowledge and "inference engine"
>
> **Extensibility**: we can extend the existing behavior by simply adding new propositions. **Knowledge is modular**, the reasoning mechanism does not change.
>
> **Understandability**: the system can be understood at knowledge level. Important for debugging, so we can debug faulty behavior by changing erroneous beliefs, and **accountability**, so that the system can explain and justify current behavior in terms of the knowledge used.

Representation and reasoning are intimately connected, in AI research. **The representation scheme must be expressive enough** to describe many aspects of complex worlds with symbolic structures. The reasoning mechanism needs to ensure that **reasoning can be performed efficiently enough**. There is a trade-off between these two concerns.
Fundamental trade-off in knowledge representation and reasoning: the more expressive the representation language is, the more complex is the reasoning. We want the best compromise.
The expressivity of representation language doesn't concern what *can be said*, but what *may be left unsaid*: it's related to the possibility of expressing uncertainty and incomplete information.
The complexity of inference regards the computational cost of deciding entailment ($\text{KB} \vDash \alpha$)

Knowledge representation and classical logic

Classical logic is propositional calculus (PROP) and first order predicate logic (FOL). We can understand KB systems at two different levels:

> **Knowledge level**: representation language and its semantics, what can be expressed and what can be inferred
>
> **Symbol level**: computational aspects, efficiency of encoding, data structures and efficiency of reasoning procedure, including their complexity

DPLL

Requires a formula in clausal form (conjunctive normal form, a conjunction of disjunctions of atomic formulas meaning "and outside or inside")
It enumerates, with a depth first strategy, all interpretations looking for a model (an interpretation that makes a set of formulas true), with three strategies:

Anticipated control: if one clause is false it backtracks, if one literal is true then the clause is satisfied

Pure symbols heuristics: first assign pure symbols (symbols that appear everywhere with the same sign)

Unit clauses heuristics: first assign unit clauses (only one literal)

Unification algorithm

Unification Fundamental operation in FOL, which computes a **substitution** that makes two expressions identical. For example, $P(A, y, z)$ and $P(x, B, z)$ can be made identical by the substitution $\tau = \{x/A, y/B, z/C\}$. $\sigma = \{x/A, y/B\}$ is also a substitution and is **more general** than τ.

Algorithm It returns the Most General Unifier (MGU) or FAIL. Computes the MGU by means of a rule-based equation-rewriting system. Initially the working memory (WM) contains the equality of the two expressions to be unified and the rules modify the equations in the WM. The algorithm terminates with a failure or when there are no applicable rules (success), so that the WM contains the MGU. The rules are:

$f(s_1, \ldots, s_n) = f(t_i, \ldots, t_n) \rightarrow s_1 = t_1, \ldots, s_n = t_n$

$f(s_1, \ldots, s_n) = g(t_i, \ldots, t_m) \rightarrow$ fail when $f \neq g$ or $n \neq m$

$x = x \rightarrow$ remove equation

$t = x \rightarrow x = t$ (bring variable to the left)

$x = t$, x doesn't occur in $t \rightarrow$ apply $\{x/t\}$ to other equations

$x = t$, t is not x, x occur in $t \rightarrow$ fail (**occur check**)

Note: when comparing two different constants, we use the second rule as a special case where $n = m = 0$ and we fail.

Knowledge Engineering & Ontological Engineering

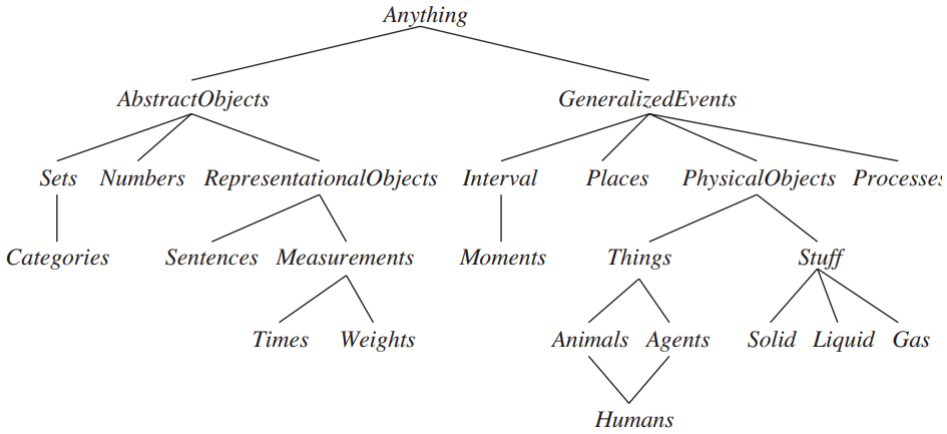
It's possible to discuss representation issues at two levels:

Knowledge Engineering is the activity to formalize a specific problem or task domain. It involves decisions about: what are the relevant facts and objects relations, which is the right level of abstraction and what are the queries to the KB (inferences)

Ontological Engineering seeks to build general-purpose ontologies which can be reused in any special-purpose domain (with additional domain-specific axioms).

Sometimes it's useful to reduce n-ary predicates to 1-place predicates and 1-place functions: this involves creating new individuals and new functions for properties/roles and it's typical of description logics/frame languages.
For example, with Purchase(john, sears, bike, \$200) we can introduce individuals for purchase objects and functions for roles (**reification**): Purchase(p23) $\wedge$ Agent(p23)=john$\wedge$ Object(p23)=bike $\wedge$ Source(p23)=sears $\wedge$ Amount(p23)=\$200...
This allows Purchase to be described at different levels of detail.

Representing common sense The use of KR languages and logic in AI is representing common sense knowledge about the world, rather than mathematics or properties of programs. Common sense knowledge is difficult since it comes in different varieties. It requires formalisms able to represent actions, events, time, physical objects, beliefs... categories that occur in many different domains. We will explore FOL as a tool to formalize different kinds of knowledge.



A general ontology organizes everything in the world into a hierarchy of categories.

Properties A general-purpose ontology should be applicable in any special-purpose domain, with the addition of domain-specific axioms. In any non-trivial domain, different areas of knowledge must be combined because reasoning and problem solving could involve several areas simultaneously.
It's difficult to construct one single best ontology: every ontology is a treaty, a social agreement, among people with some common interest in sharing. An upper ontology is like an object oriented programming framework (reuse).

Categories and objects Most reasoning takes place at the level of categories: we can infer category membership from the perceived properties of an object, and the use category information to derive specific properties of the object. There are two choices for representing categories in first-order logic:

Predicates: categories are unary predicates that we assert of individuals
Example: WinterSport(Ski), $\forall\, x$ WintersSport(x) $\Rightarrow$ Sport(x)

Objects: categories are objects that we talk about (**reification**)
Examples: Ski $\in$ WinterSports, WinterSports $\subseteq$ Sports

This way we can organize categories into taxonomies, define disjoint categories, partitions... and use specialized inference mechanisms, such as **inheritance**. Description logic takes this approach.

PartOf PartOf to say that one thing is part of another. Composite objects can be seen as part-of hierarchies, similar to the Subset hierarchy.
PartOf is transitive, PartOf$(x, y) \wedge$ PartOf$(y, z) \Rightarrow$ PartOf(x, z), and reflexive, PartOf(x, x)

BunchOf BunchOf is a composite object with definite parts but no particular structure. BunchOf({Apple1, Apple2, Apple3}) not to be confused with the set of the 3 apples.
BunchOf$(\{x\}) = x$ and each element of category S is part of BunchOf(S). Also BunchOf(S) is part of any object that has all the elements of S as parts $(\forall\, y[\forall\, x\ \ x \in S \Rightarrow \text{PartOf}(x, y)] \Rightarrow \text{PartOf}(\text{BunchOf}(S), y))$

Qualitative Measures Measures (weight, mass, cost...) are represented as unit functions that take the number as argument (for example: Centimeters(2.54))
The measures can be ordered, which enables us to do qualitative inference and is typical of the field of qualitative physics.

Objects vs Stuff Objects are countable, while stuff are mass objects: water, energy, butter...
Any part of a stuff is still stuff, example with Butter: $b \in \text{Butter} \wedge \text{PartOf}(p, b) \Rightarrow p \in \text{Butter}$
Also stuff has a number of intrinsic properties (color, density, high-fat content...) shared by all its subparts, but no extrinsic properties (weight, length, shape...). It is a substance.

Situation Calculus

Representation and reasoning about states and actions. The situation calculus is a specific ontology in FOL dealing with actions and change:

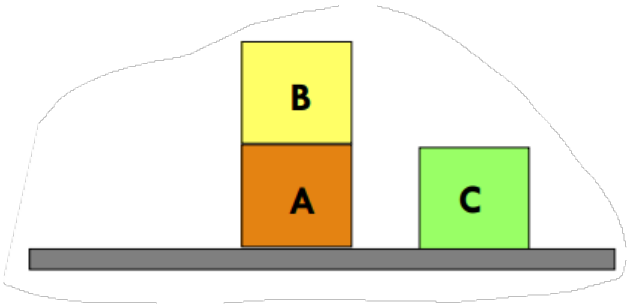
Situations: snapshots of the world at a given instant in time, the result of an action

Fluents: time dependent properties and relations

Actions: performed by an agent, but also events

Change: how the worlds changes as a result of an action

The situation calculus is the formalization in FOL of this ontology. An example: the blocks world, there are blocks on a table and the goal is to reach a given arrangement of the blocks by stacking them on top of each other.



States: arrangements of blocks on a table

Initial State and **Goal State**: a specific arrangement of blocks

Actions:

Move: move block x from block y to block z, provided x and z are free

Unstack: move block x from y to the table, x must be free

Stack: move block x from the table to y, y must be free

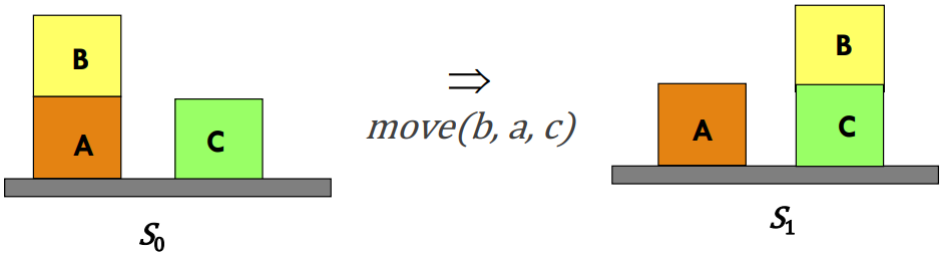
Can be formalized as:

Situations: constants $s, s_0, s_1, \ldots$ and functions denoting situations

Fluents: predicates or functions that vary from a situation to another.
On, Table, Clear... are Fluents, so for example $\text{On}(a, b)$ becomes $\text{On}(a, b, s)$

Actions: modeled as functions
$\text{Move}(a, b, c)$ is a function representing the action of moving block A from B to C. It's an instance of the generic operator/function Move. Same thing for $\text{Unstack}(a, b)$ and $\text{Stack}(a, b)$

Situations as results of actions: function Result : $A \times S \to S$, so $s_1 = \text{Result}(\text{Move}(b, a, c), s_0)$ denotes the situation resulting from the action $\text{Move}(b, a, c)$ executed in s_0. Then we can assert, for example, $\text{On}(b, c, \text{Result}(\text{Move}(b, a, c), s_0))$



This can be applied to sequences of actions, too. Result : $[A^*] \times S \to S$

$\text{Result}([], s) = s$

$\text{Result}([a \mid \text{seq}], s) = \text{Result}(\text{seq}, \text{Result}(a, s))$

In general $\text{Result}([a_1, a_2, \ldots, a_n], s_0) = \text{Result}(a_n, \text{Result}(a_{n-1}, \ldots \text{Result}(a_2, \text{Result}(a_1, s_0)) \ldots))$

Formalizing actions We need **possibility axioms**, with the structure preconditions ⇒ possibility, for example

$$\text{On}(x, y, s) \wedge \text{Clear}(x, s) \wedge \text{Clear}(z, s) \wedge x \neq z \Rightarrow \text{Poss}(\text{Move}(x, y, z), s)$$

We also need **effect axioms** such as

$$\text{Poss}(\text{Move}(x, y, z), s) \Rightarrow \text{On}(x, z, \text{Result}(\text{Move}(x, y, z), s)) \wedge \text{Clear}(y, \text{Result}(\text{Move}(x, y, z), s))$$

This is a specification of the direct effects of the action, what changes. But it's not enough: is y on the table in the new situation? Is x free?
We have a big problem: in the new situation we don't know anything about properties that were not influenced at all by the action, and these properties are the majority. This is the **frame problem**.

Frame problem and frame axioms The frame problem is one of the most classical AI problems. The name comes from an analogy with the animation world, where the problem is to distinguish the background (the fixed part) from the foreground (the things that change) from one frame to the other.
Let's fix the problem with frame axioms: Frame axioms for Clear with respect to Move

A block stays free unless the Move action is putting something on it
$\text{Clear}(x, s) \wedge x \neq w \Rightarrow \text{Clear}(x, \text{Result}(\text{Move}(y, z, w), s))$

A block remains not free unless it is not freed by a Move action
$\neg\text{Clear}(x, s) \wedge x \neq z \Rightarrow \neg\text{Clear}(x, \text{Result}(\text{Move}(y, z, w), s))$

And similarly for each pair Fluent-Action: too many axioms, **representational frame problem**.

Successor-State axioms We can combine preconditions, effects and frame axioms to obtain more compact representation for each fluent f. The schema is this:

f true after ⇔	[*preconditions before* and	*preconditions*
	an action made f true] or	*effect*
	[*f was true before and no action made it false*]	*frame axioms*

An example with Clear:

$\text{Clear}(y, \text{Result}(a, s)) \Leftrightarrow$	$[\text{On}(x, y, s) \wedge \text{Clear}(x, s) \wedge \text{Clear}(z, s) \wedge x \neq z \wedge a = \text{Move}(x, y, z)] \vee$	*effect*
	$[\text{On}(x, y, s) \wedge \text{Clear}(x, s) \wedge (a = \text{Unstack}(x, y,))] \vee$	*effect*
	$[\text{Clear}(y, s) \wedge (a \neq \text{Move}(z, w, y)) \wedge (a \neq \text{Stack}(z, y))]$	*frame*

Related problems The **representational frame problem** is considered more or less solved.

Qualification problem In real situations it is almost impossible to list all the necessary and relevant preconditions.

Ramification problem Among derived properties, which ones persist and which ones change?
Objects on a table are in the room where the table is. If we move the table from one room to another, object on the table must also change their location. Frame axioms could make the old location persist for objects.

Uses of situations calculus

Planning: finding a sequence of actions to reach a certain goal condition G
$\text{KB} \models \exists a\ G(\text{Result}(a, s_0))$ where $a = [a_1, \ldots, a_n]$

Projection: given a sequence of actions and some initial situation, determine what it would be true in the resulting situation.
Given $\phi(s)$ determine whether $\text{KB} \models \phi(\text{Result}(a, s_0))$ where $a = [a_1, \ldots, a_n]$

Legality test: checking whether a given sequence of actions $[a_1, \ldots, a_n]$ can be performed starting from an initial situation.
$\text{KB} \models \text{Poss}(a_i, \text{Result}([a_1, \ldots, a_{i-1}], s_0))$ for each $1 \leq i \leq n$
For example $\text{Result}(\text{Pickup}(b_2), \text{Result}(\text{Pickup}(b_1), s_0))$ is not a legal situation because the robot can hold only one object.

Non-Monotonic approach What we would need is the ability to formalize a notion of **persistence**: *in the absence of information to the contrary, things remain as they were*".
This leads out of classical logic, because it violates the **monotonicity property** of classical logic. The **closure assumption** we used is already an ad hoc form of completion and we will see more of this strategy in non-monotonic reasoning.
In planning, specialized languages that makes more assumptions and are more limited in their expressivity.

Limits of situation calculus Single agent, actions are discrete and instantaneous (no duration in time), they happen one at a time (no concurrency, no simultaneous actions) and only primitive actions (no way to combine, conditionals, iterations,...)
The **event calculus** is introduced to handle such cases: it's based on events, points in time, intervals rather than situations.

Event Calculus

A Fluent is an object (represented by a function)
To assert that a Fluent is true at some point in time t, we use the predicate T (true)

$T(\text{At(Shankar, Berkley)}, t)$ Where At(Shankar, Berkley) is a term, t a time

$T(\text{At(Shankar, Berkley)}, i)$ With $i = (t_1, t_2)$ being a time interval

Events are described as instances of event categories. The event E_1 of Shankar flying from San Francisco to Washington is described as $E_1 \in$ Flyings $\wedge$ Flyer(E_1, Shankar)$\wedge$ Origin(E_1, SF)$\wedge$ Destination(E_1, W)

To assert that an event happens during an extended period of time, we say Happens(e, i)

0.3.2 Nonmonotonic reasoning

Classical entailment is monotonic: if KB $\vDash a$ then KB $\cup \{b\} \vDash a$ (KB $\wedge\ b \vDash a$)
Failures of monotonicity are widespread in commonsense reasoning. It seems that humans often "jump to conclusions" when they think it's safe to do so (when they lack information to the contrary). These conclusions are only "reasonable" given what you know, rather than classically entailed.
Most of the inference we do is defeasible: additional information may lead to retract those tentative conclusions. Anytime the set of beliefs does not grow monotonically when new evidence arrives, the monotonicity property is violated. Including defeasible reasoning leads us to consider **nonsound inferences**.

Common instances of nonmonotonic reasoning

Default reasoning: reasonable assumptions unless evidence of the contrary
Car parked on the street: you assume it has for wheels even if you can only see two
Also, prototypes: birds fly, tomatoes are red...

Persistence: things stay the same, according to the principle of inertia, unless we know they change

Economy of representation: only true facts are stored, false facts are only assumed

Reasoning about knowledge: if you have $\neg$Know(p) and you learn p...

Abductive reasoning: most likely explanations to known facts

Strictness of FOL universals Universal rules (example: $\forall x\ (P(x) \Rightarrow Q(x))$) express properties that apply to all instances: all or nothing. But most of what we learn about the worlds is in terms of generics rather than universals. Properties are not strict for all instances: genetic/manufacturing varieties, borderline cases (early ferry wheels vs modern ones, or violins vs toy violins...), cases in exceptional circumstances...
Listing all exceptions is not a viable solution: qualification problem in enumerating all exceptions, and similarly for general properties of individuals. The goal is to be able to say a P is a Q in general, normally, but not necessarily. It is reasonable to conclude $Q(a)$ given $P(a)$ unless there's a good reason not to.
This is what we call a default, and **default reasoning** the tentative conclusion. Three ways to approach the problem:

Model Theoretic Formalizations (CWA, Circumscription)
Consist in a restriction to the possible interpretations, redefining the notion of entailment.

Proof Theoretic Formalizations (Default logic, Autoepistemic logic)
A proof system with non-monotonic inference rules. Autoepistemic logic (under the heading "logics for knowledge and beliefs")

Systems Supporting Belief Revision (TMS, ATMS)

Closed Worlds Assumpion (CWA) *There are usually many more negative facts than positive facts*! Under CWA, only positive facts are stored, and any other basic fact is assumed false. It's used in deductive databases and in logic programming with negation as failure. Corresponds to a new version of entailment $\models_C$:

KB $\models_C a \Leftrightarrow$ CWA(KB) $\models a$
Where CWA(KB) = KB $\cup \{\neg p \mid p$ ground atom and KB $\neq p\}$, the set of assumed beliefs

CWA(KB) is the completion under CWA of KB. CWA is a form of theory completion and is nonmonotonic.

Consistent and complete knowledge KB with consistent knowledge (satisfiable): $\nexists\, a \mid \text{KB} \models a \wedge \text{KB} \models \neg a$, so there's no contradiction.

Complete theory $\forall\, a$ KB $\models a \vee$ KB $\models \neg a$
Normally a KB has incomplete knowledge:

Let KB = $\{p \vee q\}$, then KB $\models (p \vee q)$ but KB $\not\models p$ and KB$\not\models \neg p$
Also, for any ground atom not mentioned in KB, KB $\not\models r$ and KB $\not\models \neg r$

CWA can be seen as an **assumption about complete knowledge**, or a way to make a theory complete.
Theorem: $\forall\, a$ within the language, KB $\models_C a \vee$ KB $\models_C \neg a \Leftrightarrow$ CWA(KB) $\models a \vee$ CWA(KB) $\models \neg a$

CWA(KB) isn't always consistent when KB is consistent. Problems with disjunctions, for example: KB = $\{p \vee q\}$, CWA(KB) = KB $\cup \{\neg p, \neg q\}$ since KB $\not\models p$ and KB $\not\models q$, but KB $\cup \{\neg p, \neg q\} \models \neg(p \vee q)$ then CWA(KB) is inconsistent.
The solution is to restrict CWA to atoms that are "uncontroversial": p, q are controversial, r isn't.
CWA limited in such a way is called Generalized CWA (GCWA), is a weaker form of completion than unrestricted CWA (the assumed beliefs are less)

GCWA: if KB $\models \{p \vee q_1, \ldots, \vee q_n\}$ and KB $\neg \models p$ then add $\neg p$, provided at least one ground atom q_i is entailed by KB
Theorem consistency of CWA: CWA(KB) is consistent $\Leftrightarrow$ whenever KB $\models (q_1 \vee \ldots \vee q_n)$ then KB $\models q_i$ for some i.

Since it may be difficult to test the conditions of this theorem, the following **corollary** (which restricts the application of CWA) is also of practical importance: if the KB is made of Horn clauses and it's consistent, then CWA(KB) is consistent.
A clause is a disjuction of atomic formulas (positive and negative literals), and a Horn clause has *at most* one positive literal.

Vivid knowledgebase A model is a vivid representation of the world. In a vivid KB we store a unique interpretation of the world (a consistent and complete set of positive literals) and answer questions retrieving from it. A vivid KB has the CWA built in.
If positive atoms are stored as a table, deciding if KB $\models_C a$ is like a DB retrieval. Instead of reasoning with sentences we reason about an analogical representation of the world, with these properties:

For each object of interest in the world, there is exactly one constant in KB that stands for that object.

For each relationship of interest in the world, there is a corresponding predicate in the KB such that the relationship holds among certain objects in the world if and only if the predicate with the constants as arguments is an element of the KB.

Extension to quantifiers The application of the theorem of consistency depends on the terms that we allow as part of the language. The **Domain Closure Assumption** (DCA) may be used to restrict the constants to those explicitly mentioned in the KB. Under this restriction, quantifiers can be replaced by finite conjunctions and disjunctions.
The **Unique Names Assumption** (UNA) can be used to deal with terms equality.

CWA in synthesis CWA is the assumption that atomic formulas not entailed by the KB are assumed to be false. This is a normal assumption in databases. Formally, KB $\models_C a \Leftrightarrow$ KB $\cup\{\neg p \mid p$ ground atom and KB $\not\models p\} \models a$
The KB so augmented is **complete**: $\forall\, a$ KB $\models_C a$ or KB $\models_C \neg a$
Consistency requires a more restricted formulation of the assumed belief GCWA. If KB $\models \{p \vee q_1 \vee \ldots \vee q_n\}$ and KB $\not\models p$ then add $\neg p$ if at least one ground literal q_i is entailed.
Query processing can be reduced as a combination of atomic queries.
Vivid knowledgebases store the plus part of a complete interpretation and make reasoning efficient (the augmentation reduces the possible models to one)

Circumscription A more powerful and precise version of CWA, working also for FOL. The idea is to specify special **abnormality predicates** for dealing with exceptions.
For example, suppose we want to assert the default rule "birds fly":

All normal birds fly: $\forall\, x$ Bird$(x) \wedge \neg$Ab$_f(x) \Rightarrow$ Flies(x)

We also have Bird(Tweety), Bird(Chilly), Chilly $\neq$ Tweety, $\neg$Flies(Chilly)
We want to infer Flies(Tweety), but Tweety could satisfy Ab$_f$ in some model

The idea is to **minimize abnormality**.
Circumscription: given the unary predicate Ab, consider only interpretation where I[Ab$_f$] is **as small as possible** relative to KB.

Minimal Entailment Let P be a set of unary abnormality predicates. Let I_1 and I_2 two interpretations that agree on the values of constants and functions.

Ordering on interpretations
$I_1 < I_2 \Leftrightarrow$ same domain $\wedge \forall\, p \in P\ \ I_1[p] \subset I_2[p]$ holds

Minimal Entailment
KB $\models_{\leq} a \Leftrightarrow a$ is true in I in every minimal model I
Note: model (I[KB] = true) and minimal (there is no other interpretation $I' < I$ such that I'[KB] = true)
a doesn't need to be true in all interpretations satisfying KB but only in all those that minimize abnormalities.

Issues Although the default assumptions made by circumscription are usually weaker than those of the CWA, there are cases where they appear too strong.
A partial fix is to distinguish between P (variable predicates) and Q (fixed predicates), and ordering on interpretations differently for the two sets so that only predicates in P are allowed to be minimized:

$\forall\, p \in P\ \ I_1[p] \subset I_2[p]$ holds

$\forall\, q \in Q\ \ I_1[q] = I_2[q]$ holds

The problem is that we need to decide what to allow to vary.

Default Logic We use rules to specify implicit beliefs. We distinguish explicit beliefs (axioms) from implicit beliefs (theorems).
Default logic KB uses two components: KB = (F, D)

F is a set of sentences (**facts**)

D is a set of **default rules** $\frac{\alpha:\beta}{\gamma}$
Read it as: if you can infer α and it's consistent to assume β, then infer γ:

α prerequisite

β justification

γ conclusion

Default rules where $\beta = \gamma$ are called normal defaults

Extensions How to characterize theorems/entailments? Cannot write a derivation, since don't know when to apply default rules, and no guarantee a unique set of theorems.
Extensions: set of sentences that are "reasonable" beliefs, given explicit facts and default rules. E is an extension of (F, D) $\Leftrightarrow$ for every sentence π, E satisfies the following:

$$\pi \in E \Leftrightarrow F \cup \Delta \vDash \pi \quad \text{where } \Delta = \left\{ \gamma \mid \frac{\alpha : \beta}{\gamma} \in D, \alpha \in E, \neg\beta \notin E \right\}$$

An extension E is the set of entailments of $F \cup \Delta$ where the Δ is the "suitable" set of assumptions given D.
Note that α has to be in E, not in F. This has the effect of allowing the prerequisite to be believed as the result of other default assumptions. Note also that this definition is not constructive.

Theorem: an extension of a default theory is inconsistent $\Leftrightarrow$ the original F is inconsistent.
In the following example the extension is unique, but in general a default theory can have multiple extensions.
Suppose KB is

F = {Bird(Chilly), Bird(Tweety), ¬Flies(Chilly)}

D = $\left\{ \frac{\text{Bird}(x):\text{Flies}(x)}{\text{Flies}(x)} \right\}$

then the unique possible extension is Δ = {Flies(Tweety)}, since Bird(Tweety)$\in E$ and ¬Flies(Tweety)$\notin E$.

Properties

If a default theory has distinct extensions, they are **mutually inconsistent**.
Example: F={A∨B}, D=$\left\{ \frac{:\neg A}{\neg A}, \frac{:\neg B}{\neg B} \right\}$ then E_1={A∨B, ¬A}, E_2={A∨B, ¬B} are mutually inconsistent

There are default theories with no extensions.
Example: with D=$\left\{ \frac{:\neg A}{\neg A} \right\}$, if F={} then E={}

Every normal default theory has an extension

Adding new normal default rules does not require the withdrawal of beliefs, even if adding new beliefs might. Normal default theories are semi-monotonic.

Grounded Extensions We have a problem that leads to a more complex definition of extension. Suppose F = {} and D=$\left\{ \frac{:p}{\neg p} \right\}$. Then E = entailments of $\{p\}$ is an extension since $p \in E$ and $\neg p \notin E$, but we have no good reason to believe p. The only support for p is the default rule, which requires p itself as a prerequisite. So the default should have no effect.
Desirable extension is only E = entailments of {}, that is to say all valid formulas. It's necessary a revision of the definition.
Grounded extensions: $\forall$ set S, let $\Gamma(S)$ be the least set containing F, closed under entailment and satisfying the default rules

$$\frac{\alpha : \beta}{\gamma} \in D, \alpha \in \Gamma(S) \wedge \neg\beta \notin S \Rightarrow \gamma \in \Gamma(S)$$

instead of $\neg\beta \notin \Delta(S)$
A set E is an extension of (F, D) $\Leftrightarrow E = \Gamma(E)$, i. e. E is a fixed point of the Γ operator.

0.3.3 Knowledge and Beliefs

Human intelligence is social: we need to negotiate and coordinate with others. In multi-agent scenarios we need methods for one agent to model **mental states** of other agents: high level representations of other agent's belief, intentions and goals may be relevant for acting.
By mental states we mean the relation of an agent to a proposition. **Propositional Attitudes** that an agent can have include Believes, Knows, Wants, Intends, Desires, Inform... so called because the argument is a proposition. Propositional attitudes do not behave as regular predicates.

Referential Transparency Suppose we try to assert "Lois knows that Superman can fly". Knows(Lois, CanFly(Superman))

What is "CanFly"? A predicate? A term?

Since Superman = Clark, then we can reason as follows:
(Superman = Clark) $\wedge$ Knows(Lois, CanFly(Superman)) $\vDash$ Knows(Lois, CanFly(Clark))

This property is called **referential transparency**: what matters is the object that the term names, not the form of the term. Important property for reasoning in classical logic.
Propositional attitudes like Believes and Knows require **referential opacity**: the terms used do matter, because an agent may not be aware of which terms are co-referential.

Three approaches

Reification: we remain within FOL, as we did for the situation calculus, using terms to represent propositions.
Ex: Bel(a, On(b, c))

Meta-Linguistic representation: we remain within FOL and represent proposition as strings.
Ex: Bel(a, "On(b, c)")

The problem in both of the previous approaches is connecting the reified version of the proposition (a function or a string) and the proposition itself

Modal Logics: propositional attitudes are represented as **modal operators** in specialized modal logics, with alternative semantics. Modal operators are an extension of classical logical operators.
Ex: B(a, On(b, c)) or B_a(On(b, c)), K_a(On(b, c)) to say that the agent a believes/knows that block b is on c.
Classical logic has only one modality (modality of truth): P is the same as saying "P is true"

Modal Logics Modal logic is about necessity and possibility.

$\Box A$ it is necessary that $A\ldots$ (with $\Box$ being some sort of universal quantification over interpretations)

$\Diamond A$ it is possible that $A\ldots$ (with $\Diamond$ being some sort of existential quantification over interpretations)

The simplest logic is called K, and results from adding the following to the principles of PROP:

Necessitation Rule: If A is a theorem of K, then so is $\Box A$

Distribution Axiom: $\Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$

Logic T adds axiom

(M) $\Box A \Rightarrow A$

Can be relevant for some modal operators, but not for others. For example, Knows $A \Rightarrow A$ seems plausible, while Believes $A \Rightarrow A$ is not.
Logic $S4$ adds

$\Box A \Rightarrow \Box\Box A$

Logic $S5$ adds

$\Diamond A \Rightarrow \Box\Diamond A$

World semantics Semantics for modal logics is defined with reference to: a set W of possible worlds and an **accessibility relation** R between worlds.
A formula A is now given an interpretation with reference to a possible world w: we write $I(A, w)$

$\neg$ $I(\neg A, w) = T \Leftrightarrow I(A, w) = F$

$\Rightarrow$ $I(A \Rightarrow B, w) = T \Leftrightarrow I(A, w) = F \vee I(B, w) = T$

$\vdots$

$\Box$ $I(\Box A, w) = T \Leftrightarrow \forall\, w' \in W \mid wRw'$ we have $I(A, w') = T$

$\Diamond$ $I(\Diamond A, w) = T \Leftrightarrow \exists\, w' \in W \mid wRw'$ we have $I(A, w') = T$

Different modal logics are defined according to the properties of the accessibility relation R and corresponding axioms.

Referential Transparency Modal logics address this problem, since the truth of a complex formula does not depend on the truth of the components in the same world/interpretation. **Modal operators are not compositional**. Modal logics for knowledge are easier than those of beliefs.

Syntax for modal logic for knowledge

All the well formed formulas of ordinary FOL are also well formed formulas of the modal language

If ϕ is a closed well formed formula of the modal language and a is an agent, then Knows(a, ϕ) is a formula of the modal language

If ϕ and ψ are well formed formulas so are the formulas that can be constructed form them with the usual logic connectives.

Some examples:

Knows(A_1, Knows(A_2, On(B, C))), A_1 knows that A_2 knows that B is on C

Knows(A_1, On(B, C)) $\vee$ Knows(A_1, $\neg$On(B, C)), A_1 knows whether B is on C or not

$\neg$Knows(A_1, On(B, C)), A_1 doesn't know that B is on C

Properties of knowledge

One agent can hold false beliefs but **cannot hold false knowledge**.
If an agent knows something then that must be true. **Knowledge is justified true belief**.

An agent **doesn't know all the truths**: something may be true without the agent knowing it.

If two formulas ϕ and ψ are equivalent, not necessarily Knows(A, ϕ) implies Knows(A, ψ)

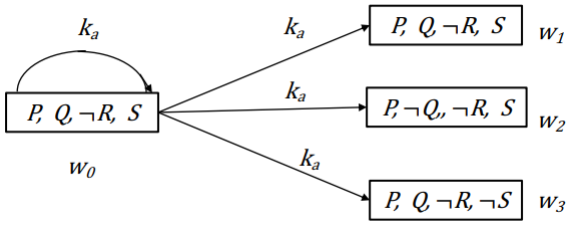
The semantics of modal logic is given in terms of possible worlds, and specific accessibility relations among them, one for each agent.
An agent know a proposition just when that proposition is true in all the worlds accessible from the agent's worlds (those that the agent consider possible).
Possible worlds roughly correspond to contexts within interpretations. An accessibility relation is defined for agents and connects possible worlds: if Knows(a, w_i, w_j) is satisfied, then world w_j is accessible form world w_i for agent a.
Possible world semantics:

Regular well formed formulas (with no modal operators) are not simply true or false but they are true or false with reference to a possible world.
$I(w_1, \phi)$ may be different from $I(w_2, \phi)$

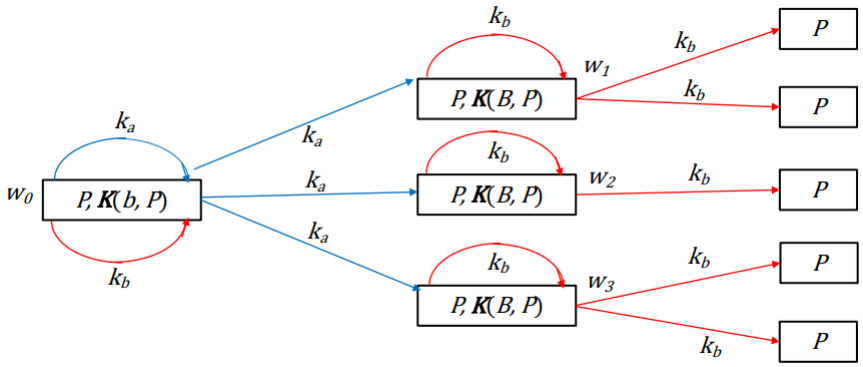
A modal formula Knows(a, ϕ) is true in $w \Leftrightarrow \phi$ is true in all the worlds accessible from w for agent a.

The semantics of complex formulas is determined by regular truth recursive rules.

Knows(a, ϕ) means that an agent a knows the proposition denoted by ϕ. "Not knowing ϕ" in w_0, a specific world, is modeled by allowing some worlds, accessible by w_0, in which ϕ is true and some worlds in which ϕ is false.
For example, in this scenario Knows(a, P) and Knows$(a, \neg R)$ in w_0 since P and $\neg R$ are true in worlds w_0, w_1, w_2, w_3, but Knows(a, Q) is false in w_0.



The accessibility relation also accounts for nested knowledge statements, also involving different agents. Knows(a, Knows(b, P)) holds in w_0 since Knows(b, P) holds in w_0, w_1, w_2 and w_3 accessible to a.



Properties and axioms for knowledge Many of the properties that we desire for the knowledge can be achieved by imposing constraints to the accessibility relation.

1. Agents should be able to reason with the knowledge they have
   $\text{Knows}(a, \alpha \Rightarrow \beta) \Rightarrow (\text{Knows}(a, \alpha) \Rightarrow \text{Knows}(a, \beta))$, **distribution axiom**
   Implicit in possible worlds semantics

2. Agents cannot have false knowledge
   $\text{Knows}(a, \alpha) \Rightarrow \alpha$, **knowledge axiom**
   Satisfied if the accessibility relation is **reflexive**, i.e. $\text{Knows}(a, w, w)$ for every a and every w. An implication is that $\neg\text{Knows}(a, \text{false})$
   Moreover, it implies that the relation is also **serial**, i.e. there's at least a world accessible from w.

3. It's reasonable to assume that if an agent knows something, then it knows that it knows
   $\text{Knows}(a, \alpha) \Rightarrow \text{Knows}(a, \text{Knows}(a, \alpha))$, **positive introspection**
   The accessibility relation must be **transitive**, i.e. $\text{Knows}(a, w_1, w_2) \wedge \text{Knows}(a, w_2, w_3) \Rightarrow \text{Knows}(a, w_1, w_3)$

4. In some axiomatization we also assume that if an agent doesn't know something, then it knows that it doesn't know
   $\neg\text{Knows}(a, \alpha) \Rightarrow \text{Knows}(a, \neg\text{Knows}(a, \alpha))$, **negative introspection**
   The accessibility relation must be **euclidean**, i.e.$\text{Knows}(a, w_1, w_2) \wedge \text{Knows}(a, w_1, w_3) \Rightarrow \text{Knows}(a, w_2, w_3)$

5. It's also intrinsic in possible worlds semantic that an agent knows all the logical theorems, including the ones characterizing knowledge
   From $\vdash \alpha$ infer $\text{Knows}(a, \alpha)$, **epistemic necessitation rule**

6. From the first and fifth properties, we can also get the rules

   From $\alpha \vdash \beta$ and $\text{Knows}(a, \alpha)$ infer $\text{Knows}(a, \beta)$

   From $\vdash \alpha \Rightarrow \beta$ infer $\text{Knows}(a, \alpha) \Rightarrow \text{Knows}(a, \beta)$

   This is called **logical omniscience**, is considered problematic: we are assuming unbounded reasoning capabilities. As a corollary of logical omniscience we have $\text{Knows}(a, \alpha \wedge \beta) \equiv \text{Knows}(a, \alpha) \wedge \text{Knows}(a, \beta)$ (Knows distribution over $\wedge$)
   It's however not the case that $\text{Knows}(a, \alpha \vee \beta) \equiv \text{Knows}(a, \alpha) \vee \text{Knows}(a, \beta)$

Modal epistemic logics are obtained with various combination of axioms 1-4 plus inference rule 5:

System K: axiom 1

System T: axioms 1-2

Logic S4: axioms 1-3

Logic S5: axioms 1-4 (perfect reasoner)

Not any combination is possible since the properties of accessibility are interdependent, for example: reflexive implies serial, a reflexive and euclidean relation is also transitive (axioms 2 and 4 impliy 3), ...

Properties and axioms for beliefs Since an agent can hold wrong beliefs, the knowledge axiom is not appropriate. Instead, we include as an axiom the following: ¬Believe(a, False), **lack of contradictions**.
The distribution axiom and the necessitation rule are controversial, since an agent cannot realistically believe all the logical consequences of its beliefs but only those that he is able to derive (**limited/bounded rationality**).

Believe$(a, \alpha) \Rightarrow$ Believe(a, Believe(a, α)), **positive introspection**

Believe$(a, \alpha) \Rightarrow$ Knows(a, Believe(a, α)) is also reasonable

Negative introspection is problematic, while the following special case of the knowledge axioms is safe: Believe(a, Believe$(a, \alpha)) \Rightarrow$ Believe$(a, \alpha) \ldots$

0.3.4 Autoepistemic Logic

At the intersection between Modal Logics for Beliefs and Nonmonotonic Logic.
$\frac{\alpha:\beta}{\gamma}$, in default logic, is read as "if α and *it is consistent to believe* β then γ"
A different approach to nonmonotoninc reasoning is to model "*it is consistent to believe* β" as the lack of belief in $\neg\beta$ with a suitable logic for belief.

Autoepistemic Logic It uses a belief operator B. $B\alpha$ stands for "α is believed to be true". The B operator could be used to represent defaults, for example:

$\forall\, x$ Bird$(x) \wedge \neg B\neg$Flies$(x) \Rightarrow$ Flies(x)
Any bird not believed to be unable to fly, does fly.

Note that $B\neg$Flies(x) is different from ¬Flies(x)

Given a KB that contains sentences with the B "autoepistemic" operator, what is a reasonable set of beliefs E to hold? Minimal properties of a set of beliefs E to be considered stable:

Closure under entailment: if $E \vDash \alpha$ then $\alpha \in E$

Positive introspection: if $\alpha \in E$ then $B\alpha \in E$

Negative introspection: if $\alpha \notin E$ then $\neg B\alpha \in E$

This leads to a the definition of a stable expansion of a KB (minimal set satisfying all three properties)

Stable Expansion of the KB A set E is a stable expansion of KB $\Leftrightarrow$ $\forall$ sentence π we have

$$\pi \in E \Leftrightarrow KB \cup \{B\alpha \mid \alpha \in E\} \cup \{\neg B\alpha \mid \alpha \notin E\} \vDash \pi$$

with $\{B\alpha \mid \alpha \in E\} \cup \{\neg B\alpha \mid \alpha \notin E\}$ being the set of assumed beliefs. The **implicit beliefs** E are those sentences entailed by KB plus the assumptions deriving from positive and negative introspection.

0.3.5 Graph representation and structured representation

Psychological-linguistic approach to KR&R

Logical Approach Suitable to model rational reasoning: extension to model commonsense reasoning (frame problem, nonmonotonic reasoning, propositional attitudes...), FOL contradictions to control computational complexity of reasoning (description logics, logic programming...)

Cognitive-Linguistic Approach More concerned with understanding the mechanisms for knowledge acquisition, representation and use in human minds. Synergies with other fields, such as cognitive psychology and linguistics.

Associationist theories of KR In logic systems, symbolic expressions are modular and are syntactically transformed without paying attention to the symbols used or to their "meaning". The symbol themselves are arbitrary: it's all a matter of writing axioms to restrict interpretations.
Associationist theories are instead concerned with the connection among symbols, and the meaning that emerges from these connections. The idea is that the meaning of a symbol emerges as a result of the connections to other symbols. The connections are shaped by experience.

Semantic Networks

Semantic Memory The problems is how the meaning of the symbols/words is acquired, represented and used. The memory itself is distinguished in:

Episodic memory: specific facts and events

Semantic memory: abstract and general knowledge

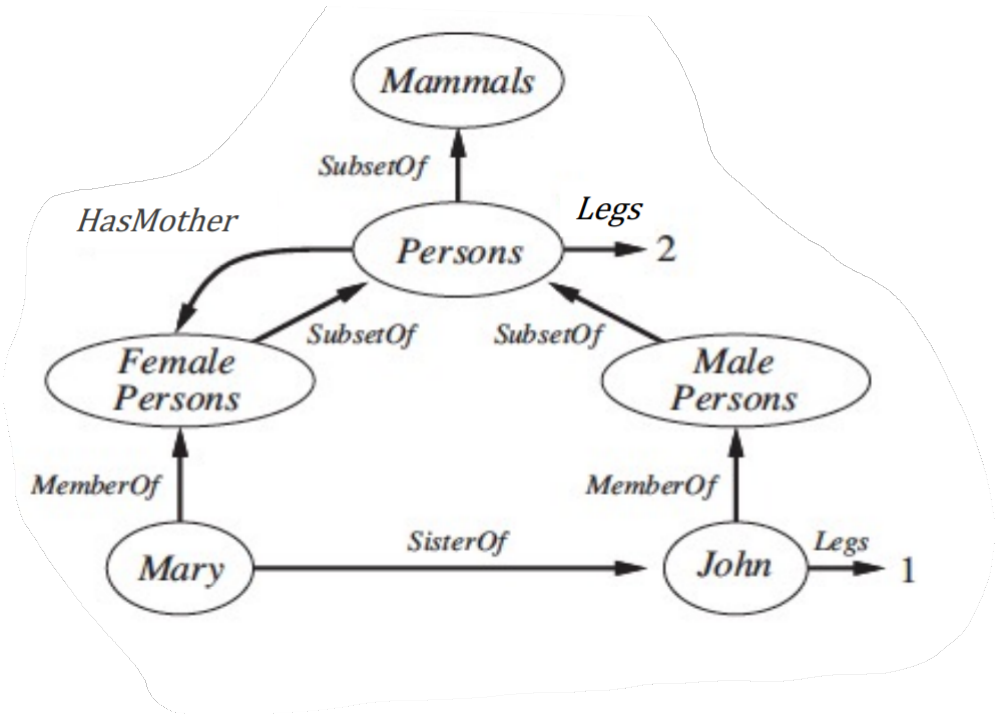
A **semantic network** is a graphical model proposed for semantic memory, with two kinds of knowledge:

Concepts: semantic counterpart of words, represented as nodes

Propositions: relations among concepts, represented as labelled arcs

No account for dynamic aspects of memory and learning. Other models are: distributional models (example: word embeddings) or connectionist models for learning.

Essence of semantic networks It's a large family of graphical representation schemes. A semantic network is a graph where: labeled **nodes** correspond to **concepts** and labeled and directed **arcs** correspond to **binary relations between concepts**.
Node come in two flavors: **generic concepts**, corresponding to categories/classes, and **individual concepts** corresponding to individuals.
Two special kind of links are always present, with different names: **Is-A** (subclass, between two generic concepts) and **Inst-Of** (member of, between and individual concept and a class).
An example:



Inheritance Implemented as link traversal: find the first occurrence of the property that you seek.

Logic accounts We may look at semantic networks as a convenient implementation for a part of FOL: representation and mechanism at the symbol level rather than at the knowledge level

$A \overset{\text{Is-A}}{\longrightarrow} B$
$\forall x \ A(x) \Rightarrow B(x)$
All members of class A are also members of class B

$a \overset{\text{Inst-Of}}{\longrightarrow} B$
$B(a)$
a belongs to class B

$A \xrightarrow{\text{R}} b$

$\forall\, x \;\; x \in A \Rightarrow R(x, b)$

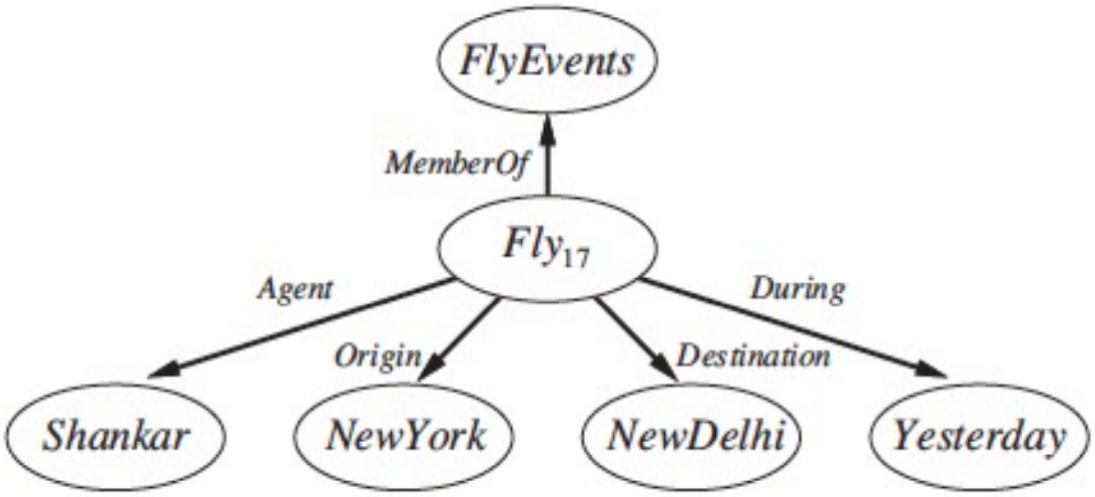
All members of class A are in relation R with b

$A \xrightarrow{\text{R}} B$

$\forall\, x \;\; x \in A \Rightarrow \exists\, y \;\; y \in B \wedge R(x, y)$

For all members of class A, there is a R-related element in B

An example of "Shankar flew from New York to New Deli yesterday"



Limited expressive power Even if they can express n-ary predicates like in the example above, semantic networks do not have the same expressive power of FOL: Existential, $\vee$, $\Rightarrow$,... are not expressible or are expressible only in special cases. Not necessarily a bad case, since it suggests that a subset of FOL with interesting computational properties, explored in description logics.
More expressive assertional networks ere proposed in the past. A well-known example in AI are Pierces's Existential Graphs, which inspired Sowa's Conceptual Graphs: they candidate as an intermediate schema for representing natural language.

0.3.6 Object Oriented Representations and Frames

It's very natural to think of knowledge not as a flat collection of sentences, but rather as a structured and organized collection in terms of the objects the knowledge is about. Complex objects have attributes, part constrained in various ways. Objects might have a behavior that is better expressed as procedures (like in OOP).

Frames

Nothing to do with frame problems, proposed by Minsky in 1975: idea of using a structured representation of objects. Knowledge is organized in complex mental structures called **frames**. When one encounters a new situation (or makes a substantial change in one's view of the present problem) one selects from memory a structure called a frame: this is a remembered framework to be adapted to fit reality by changing details as necessary.
A **frame** is a **data structure for representing a stereotypical object or situation**. There are two types:

Individual frames, used to represent single objects
Is a collection of slot-filler pairs:

```
(Frame-Name
<slot1 filler1>
<slot2 filler2>
...)
```

Fillers can be: values (usually default values), constraints on values, names of other individual frames, the special slot INSTANCE-OF. Example:

```
(toronto
<:INSTANCE-OF CanadianCity>
<:Province ontario>
<:Population 4.5M>
...)
```

Generic frames, used to represent categories or classes of objects.
Similar to individual frames, where fillers can be: the special slot IS-A or procedures (IF-ADDED, activated when the slot receives a value, or IF-NEEDED, activated when the value is requested)
These procedures are calles **procedural attachments** or demons. An example:

```
(CanadianCity
<:IS-A City>
<:Province CanadianProvince>
<:Country Canada>
...)

(Lecture
<:DayOfWeek WeekDay>
<:Date [IF-ADDED ComputeDayOfWeek]>
...)

(Table
<:Clearance [IF-NEEDED ComputeClearanceFromLegs]>
...)
```

The INSTANCE-OF and IS-A slots organize frames in frame systems: they have the special role of activating inheritance of properties and procedures. In frames, all values are understood as default values, which can be overwritten.

Reasoning with frames Attached procedure provide a flexible, organized framework for computations: reasoning has a procedural flavor.
A basic reasoning loop in a frame system has three steps:

Recognition: a new object or situation is recognized as instance of a generic frame

Inheritance: any slot fillers that are not provided explicitly but can be inherited by the new frame instance are inherited

Demons: for each slot with a filler, any inherited IF-ADDED procedure is run, possibly causing new slots to be filled, or new frames to be instantiated, until the system stabilizes.
Then the cycle repeats.

When the filler of a slot is requested:

If there's a value stored, it's returned

Otherwise, any inherited IF-NEEDED procedure is run to compute the filler for the slot. This may cause other slots to be filled, or new frames to be instantiated.

Frames and OOP Frame-based representation languages and OOP systems were developed at the same time. They look similar and certainly one could implement a frame system in OOP. The **important difference** is that frame systems tend to work in a cycle:

Instantiate a frame and declare some slot fillers

Inherit values from more general frames

Trigger appropriate forward-chaining procedures

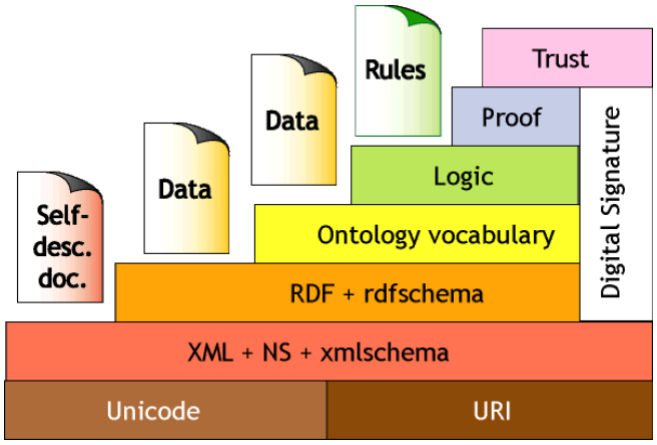
When the system is quiescent, stop and wait for the next input

The design can control the amount of "forward" reasoning that should be done (in a data-directed fashion) or "backward" (in a goal-directed fashion)

0.3.7 Description Logics

Categories and Objects Most of the reasoning takes place at the level of categories rather than on individuals. If we organize knowledge in categories and subcategories (in a hierarchy), it's enough to classify an object according to its perceived properties, in order to infer properties of the categories to which it belongs. **Inheritance** is the common form of inheritance, which exploits structures. **Ontologies** will play a crucial role, providing a source of shared and precisely defined terms that can be used in metadata of digital objects and real-world objects.

Domain ontologies The formalization of the ideas coming from semantic networks and frames resulted in **specialized logics**. These logics, called **terminological logics** and later **description logics**, find an important application in describing "domain ontologies" and represent the theoretical foundations for adding reasoning capabilities to the **semantic web**. **Ontology** is a **formal model of an application domain**, a **conceptualization**.
the **semantic web** is the vision to gradually develop alongside the "syntactic web" (or web of documents), for communication among people, a "semantic web" (or web of data), form communication among the machines. It's a huge distributed network of linked data, which can be used by programs as well, provided their semantics is shared and made clear (role of formal ontologies).



Description logics Can be seen as:

> Logical counterparts of network knowledge representation schemas, frames and semantic networks.
> In this formalization effort, defaults and exceptions are abandoned. The ideas and terminology (concept, roles, inheritance) are very similar (to KLOne in particular)
>
> Contractions of FOL, investigated to obtain better computational properties. Attention to computational complexity and decidability of the inference mechanisms.

An example: "paper3 has exactly two authors"

> (**and** Paper (**atmost** 2 hasAuthor) (**atleast** 2 hasAuthor)) [paper3]
>
> We'll use the alternative, **mathematical notation**
> paper3: Paper $\sqcap$ ($\leq$ 2 hasAuthor) $\sqcap$ ($\geq$ 2 hasAuthor)
>
> The corresponding in FOL is
> Paper(paper3) $\wedge$ $\exists$ x hasAuthor(paper3, x) $\wedge$ $\exists$ y hasAuthor(paper3, y) $\wedge$ $x \neq y$ $\wedge$ hasAuthor(paper3, z) $\Rightarrow$ $(z = x) \vee (z = y)$

Concepts, roles, individuals Each DL is characterized by operators for the construction of terms.
Terms (description) are of three types:

> **Concepts**, corresponding to unary relations, with operators for the construction of complex concepts: and ($\sqcap$), or ($\sqcup$), not ($\neg$), all ($\forall$), some ($\exists$), atleast ($\geq n$), atmost ($\leq n$), ...
>
> **Roles**, corresponding to binary relations, possibily together with operators for construction of complex roles
>
> **Individuals**, only used in assertions

Assertions are kept separate and can be of two types:

$i : C$, where i is an individual and C a concept

$(i, j) : R$, where i, j are individuals and R a role

0.4 Reasoning Under Uncertainty

0.4.1 Quantifying Uncertainty

Agents are inevitably forced to reason and make decisions based on incomplete information. They need a way to handle uncertainty deriving from:

Uncertainty in sensors: partial observability

Uncertainty in actions: nondeterministic actions

A partial answer would be to consider a set of possible worlds (a **belief set**, worlds which are considered possible by the agent), but planning by anticipating all possible contingencies can be really complex. Moreover, if no plan guarantees the achievement of the goal but the agent still needs to act, how can he evaluate the relative merits of alternative plans? **Probability theory** offers a clean way to quantify uncertainty (common sense reduced to calculus).
With a logical approach, it is difficult to anticipate everything "that could go wrong" (**quantification problem**). The rational decision depends on both the **relative importance** of the various goals and the **likelihood** that they will be achieved. When there are conflicting goals, the agent may express **preferences** by the means of a utility function. Utilities are combined with probabilities in the general theory of rational decisions called **decision theory**. An agent is rational $\Leftrightarrow$ it chooses **the action that yields the maximum expected utility, averaged over all the possible outcomes of the action**. This is called the principle of **Maximum Expected Utility** (**MEU**).

Probability Theory Logic and probability theory both talk about a world made of propositions which are *true* or *false*. They share the **ontological commitment**.
What's different is the **epistemological commitment**: a logical agent believes each sentence to be true or false or has no opinion, whereas a probabilistic agent may have a numerical **degree of belief** between 0 (certainly false) and 1 (certainly true). The uncertainty is not in the world, but in the beliefs of the agent (**state of knowledge**). If the knowledge about the world changes (we learn more information) the probability changes, but there is no contradiction.

Probabilities In AI terms: probabilistic assertions are assertions about possible worlds stating how probable a world is. The set of possible worlds is the **sample space** Ω: they are mutually exclusive and exhaustive. For example, the 36 possible outcomes of rolling two dices.
A fully specified probability model associates a probability $P \in R, 0 \leq P \leq 1$ to each possible world $\omega \in \Omega$.
The **basic axiom of probability** is

$$0 \leq P(\omega) \leq 1 \ \forall\, \omega \in \Omega \text{ and } \sum_{\omega \in \Omega} P(\omega) = 1$$

Usually we deal with a subset of possible worlds (**events**) which can be described by an expression (**proposition**) in a formal language. The **events** are the possible worlds where the proposition holds. Also for any event Φ we have that $P(\Phi) = \sum_{\omega \in \Phi} P(\omega)$

Conditional probabilities Often we have evidence that restricts the number of possible worlds, conditioning the probability of an event. These probabilities are called **conditional**/posterior probabilities: $P(a \mid b) = \frac{P(a,b)}{P(b)}$ with $P(b) > 0$. Note that $P(a, b) = P(a \mid b)P(b)$

Probability distribution A full specification of the probability for all values of a variable X is a **probability distribution**.

Discrete For example

$P(\text{Weather} = \text{sunny}) = 0.6$
$P(\text{Weather} = \text{rain}) = 0.1$
$P(\text{Weather} = \text{cloudy}) = 0.29$
$P(\text{Weather} = \text{snow}) = 0.01$

can be expressed with a vector P to represent the distribution: $P(\text{Weather}) = \langle 0.6, 0.1, 0.29, 0.01 \rangle$
$P(X \mid Y)$ is the table of values $P(X = x_1 \mid Y = y_i)$ one value for each pair i, j.

Continuous For continuous variables we can define the probability that a random variable takes some value x as a function of x, called **probability density function** or **PDF**.
If f is the density function, then $P(a \leq x \leq b) = \int_a^b f(x)\, dx$, $\int_{-\infty}^{+\infty} f(x)\, dx = 1$ for $f(x) \geq 0$

A **joint probability distribution** is a distribution over a set of variables.

Semantics A possible world is defined to be an assignment of values to all the random variables under consideration. It follows that possible worlds are mutually exclusive and exhaustive.
The truth of a complex proposition in a world can be determined using the same recursive definition of truth used for formulas in PROP.
If we have a joint probability distribution of all the variables (**full joint probability distribution**), then we can perform an inference. Given that

A proposition identifies a set of possible worlds

An entry in the table gives the probability of a possible world

For any event $\Phi, P(\Phi) = \sum_{\omega \in \Omega} P(\omega)$

we can compute the probability of any proposition by taking the sum of the probabilities of the relevant possible worlds in the distribution.

Properties of discrete variables Known as **Kolmogorov's axioms**:

$0 \leq P(\omega) \leq 1 \ \forall\, \omega \in \Omega$ and $\sum_{\omega \in \Omega} P(\omega) = 1$

For any proposition $\Phi, P(\Phi) = \sum_{\omega \in \Omega} P(\omega)$

$P(\neg a) = 1 - P(a)$

$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$ (**inclusion-exclusion principle**)

Plausibility If an agent holds an inconsistent set of beliefs, then if it bets according to this set of beliefs against another agent, then the other agent can devise a strategy to make it always lose money. De Finetti's theorem implies that no rational agent can have beliefs that violate the axioms of probability.

Probabilistic Inference

Assuming we have a full joint distribution as the knowledge base, we will use the axioms of probability and show how to:

Compute **marginals**, prior probability of a single variable

Compute the probability of complex propositions $(\wedge, \vee, \neg)$

Compute the **posterior probability** for a proposition given observed evidence

Computing marginals Probability of a variable from the full joint distribution. Given a joint distribution $P(X, Y)$ over variables X and Y, the distribution of the single variable X is given by

$$P(X) = \sum_{y \in dom(Y)} P(X, y) = \sum_{y} P(X, y)$$

Also called **marginalization** or **summing out**. In general, if Y and Z are sets of variables such that $Y \cup Z$ are all the variables in the full joint distribution, then $P(Y) = \sum_{z \in Z} P(Y, z)$

Conditioning Variant of the marginalization, involving conditional probabilities instead of joint probability: can be obtained from the previous using the product rule

$$P(Y) = \sum_{z \in Z} P(Y \mid z) P(z)$$

General inference procedure If the query involves a single variable X, e is the list of the observed values (the evidence) and Y the rest of unobserved variables, then

$$P(X \mid e) = \alpha P(X, e) = \alpha \sum_y P(X, e, y)$$

However it's intractable due to the complexity of the joint distribution table: with n boolean variables, it requires an input table of size $O(2^n)$ and $O(2^n)$ time to process.

Independence Independence of propositions a, b: $P(a \mid b) = P(a)$, $P(b \mid a) = P(b)$, $P(a, b) = P(a)P(b)$
Independence of variables X, Y: $P(X \mid Y) = P(X)$, $P(Y \mid X) = P(Y)$, $P(X, Y) = P(X)P(Y)$
Independence assumptions can reduce the size of the representation and the complexity of the inference problems. A notation for the independence $X \perp Y$.

Bayes theorem

$$P(Y \mid X) = \frac{P(X \mid Y)P(Y)}{P(X)}$$

This tells us how to update the agent's believes in hypothesis h as new evidence e arrives, given the background knowledge k

$$P(h \mid e, k) = \frac{P(e \mid h, k)P(h \mid k)}{P(e \mid k)}$$

X and Y are **conditionally independent given** Z when $P(X, Y \mid Z) = P(X \mid Z)P(Y \mid Z)$. Alternatively, when

$$P(X \mid Y, Z) = P(X \mid Z)$$

$$P(Y \mid X, Z) = P(Y \mid Z)$$

Naive Bayes model $P(\text{Cause}, \text{Effect}_1, \ldots, \text{Effect}_n) = P(\text{Cause}) \prod_i P(\text{Effect}_i \mid \text{Cause})$

0.4.2 Probabilistic Reasoning

Bayesian Networks Also called **belief networks**: graphs for representing dependencies among variables. The network makes specific conditional dependencies: the intuitive meaning of an arc from X to Y is typically that X has a direct influence on Y.
Bayesian Networks are **directed acyclic graphs** (DAG) so defined:

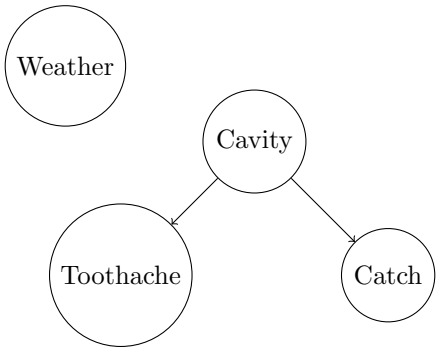
Each node corresponds to a random variable, which may be discrete or continuous.

Directed links or arcs connect pairs of nodes. If there's an arc from node X to node Y, X is said to be a parent of Y: Parents(Y) is the set of variables directly influencing Y.

Each node X has an associated **conditional probability distribution table** $P(X \mid \text{Parents}(X))$ that quantifies the effect of the parents on the node.

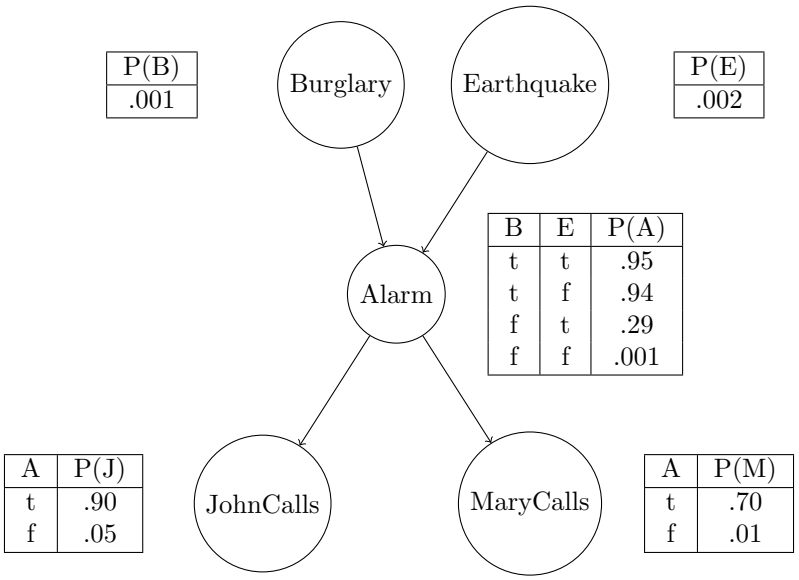
It's easier to decide what direct influences exist in the domain than actually specifying the probability themselves.

Cavity & Weather example



Weather is independent from the other variables, and Toothache and Catch are **conditionally dependent** given Cavity. The **lack of an arc between two variables** is interpreted as **independence**.

Alarm example *The burglary alarm goes off very likely on burglary and occasionally on earthquakes. John and Mary are neighbors who agreed to call when the alarm goes off. Their reliability is different...*



The tables are called **Condition Probability Tables**.

Semantic of Bayesian Networks

Two alternative and equivalent ways to look at the semantic of Bayesian Networks:

Distributed representation of the full joint probability distribution
Suggests a metodology for constructing networks

Encoding of a collection of conditional independence statements
Helps in designing inference procedures

Representation of the full joint distribution A joint probability distribution can be expressed in terms of conditional probabilities

$P(x_1, \ldots, x_n) = P(x_n \mid x_{n-1}, \ldots, x_1) \cdot P(x_{n-1}, \ldots, x_1)$, product rule

And by iterating the process we get the **chain-rule**

$$P(x_1, \ldots, x_n) = P(x_n \mid x_{n-1}, \ldots, x_1) P(x_{n-1} \mid x_{n-2}, \ldots, x_1) \ldots P(x_2 \mid x_1) P(x_1) = \prod_{i=1}^{n} P(x_i \mid x_{i-1}, \ldots, x_1)$$

So we assumed an order of the variables, we computed posterior probabilities of each variable, given all previous variables and we took the product of these posteriors.

Representing the joint probability distribution Given that $P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid x_{i-1}, \ldots, x_1)$ and assuming

Each variable appears after its parents in the ordering, meaning $\text{Parents}(X_i) \subseteq \{X_{i-1}, \ldots, X_1\}$

We simplify the computation by conditioning the computation only to values of the parent variable (assuming the others are independent)

The numbers in the Condition Probability Tables are actually conditional probabilities.

Each entry in the joint distribution can be computed as the product of the appropriate entries in the conditional probability tables (CPTs) in the Bayesian network. We get the **simplified rule** for computing joint distributions:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid \text{Parents}(x_i))$$

Given the Alarm example and the ordering (J, M, A, B, E) we can compute, by the chain rule

$$P(j, m, a, \neg b, \neg e) = P(j \mid m, a, \neg b, \neg e)P(m \mid a, \neg b, \neg e)P(a \mid \neg b, \neg e)P(\neg b \mid \neg e)P(\neg e)$$

And taking into account only direct dependencies, we can simplify as

$$P(j, m, a, \neg b, \neg e) = P(j \mid \cancel{m}, a, \cancel{\neg b, \neg e})P(m \mid a, \cancel{\neg b, \neg e})P(a \mid \neg b, \neg e)P(\neg b \mid \cancel{\neg e})P(\neg e)$$

And using the probability values from the CPTs we get $P(j \mid a)P(m \mid a)P(a \mid \neg b, \neg e)P(\neg b)P(\neg e) = 0.9 \cdot 0.7 \cdot 0.001 \cdot 0.999 \cdot 0.998 = 0.000628$

Procedure for building the network

Nodes: determine the set of variables required to model the domain, and order them $\{X_1, \ldots, X_n\}$
Ordering influences the result: the resulting network will be more compact if the variables are ordered such that causes precedes effects.

Liks: for $i = 1$ to n do

Choose, from $X_1, \ldots, X_{i-1}$ a minimal set of parents for X_i such that the equation $P(X_i \mid X_{i-1}, \ldots, X_1) = P(X_i \mid \text{Parents}(X_i))$ is satisfied

For each parent insert an arc from the parent to X_i

CPTs: write down the conditional probability table $P(X_i \mid \text{Parents}(X_i))$

Note that the parents of X_i should contain all the nodes that **directly** influence X_i. Also the network is a DAG by construction and contains no redundant probability values.
In **locally structured systems** each subcomponent interacts directly only with a bounded number of other components. They are usually associated with linear rather than exponential growth in complexity. In the case of Bayesian networks, it is reasonable to suppose that in most domains each random variable is directly influenced by at most k others, for some constant k.
If we assume n boolean variables, then each conditional probability table will have at most 2^k numbers, and the complete network can be specified by $n \cdot 2^k$ numbers. In contrast, the joint distribution contains 2^n numbers.

Conditional Independence We can also extract the independence assumptions encoded in the graph to do inference. The topological semantics specifies that each variable is conditionally independent of its non-descendants, given its parents. In our example, the network tells us that JohnCalls is independent of Burglary, Earthquake and MaryCalls given the value of Alarm. This autorizes the following semplification $P(J, B, E, M \mid A) = P(J \mid A)P(M \mid A)P(B)P(E)$
A node is conditionally independent of all other nodes in the network given its parents, children and children's parents.

Efficient construction of CPTs Often, the relationship between a node and its parent follows a canonical distribution and the construction of the CPTs can be simplified. Two examples:

Deterministic Nodes: nodes whose value is specified exactly by the values of their parents. For example, $\text{Price}_1, \ldots, \text{Price}_k$ (the parents) determine $min(\text{Price}_1, \ldots, \text{Price}_k)$ (the best price)

Noisy-OR Relations: a generalization of the logical OR. We can specify inibiting factors, so that A∨B⇔C but not always. The probability of ¬C is the product of the inibition probabilities q for each parent that is true. It is 1 when none of the causes are true. $P(x_i \mid \text{Parents}(X_i)) = 1 - \prod_{\{j \mid X_j = \text{true}\}} q_j$

Continuous variables One possible way to handle them is to avoid them and use discrete intervals. For example, temperature into three intervals: < 0°C, $[0, 100]$°C, > 100°C.
The most common solution is to define standard families of probability density function that are specified by a finite number of parameters, for example a Gaussian (normal) distribution $N(\mu, \sigma^2)(x)$ with parameters the mean μ and the variance σ^2.
A network with both discrete and continuous variables is called hybrid Bayesian network.

Exact inference in Bayesian Networks

The basic task of probabilistic inference. Given:

X: the query variable (assuming just one)

E: the set of evidence variables $\{E_1, \ldots, E_m\}$

Y: the set of unknown variables, i.e. the **hidden** non-evidence variables $\{Y_1, \ldots, Y_l\}$

Thus $X = \{X\} \cup E \cup Y$

A typical query asks for the posterior probability distribution, given some evidence $P(X \mid e)$. In the alarm example, a query could be P(Burglary | JohnCalls=true, MaryCalls=true) with E = {JohnCalls, MaryCalls} and Y = {Earthquake, Alarm}

Inference by enumeration We defined a procedure for the task as $P(X \mid e) = \alpha P(X, e) = \alpha \sum_y P(X, e, y)$ with α being a normalizing factor.
The query can be answered using a Bayesian network by computing sums of products of conditional probabilities from the network. In the example: $P(B \mid j, m) = \alpha P(B, j, m) = \alpha \sum_e \sum_a P(B, j, m, e, a)$ summing over $y = \{E, A\}$
We can now use the CPTs for computing $P(B, j, m, e, a)$, i.e. the case of Burglary = true:

$$P(B \mid j, m) = \alpha \sum_e \sum_a P(B, j, m, e, a) = \alpha \sum_e \sum_a P(B)P(e)P(a \mid B, e)P(j \mid a)P(m \mid a)$$

And simplifying

$$P(B \mid j, m) = \alpha P(B) \sum_e P(e) \sum_a P(a \mid B, e)P(j \mid a)P(m \mid a)$$

The tree of computation is evaluated depth first, but the computation can be improved substantially by storing and reusing the results of repeated computations. The variable elimination algorithm proceeds right to left (bottom-up in the tree). Conditional probabilities are represented as factors, matrices resulting from conditional probabilities. In the example, it becomes

$$P(B \mid j, m) = \alpha f_1(B) \sum_e f_2(E) \sum_a f_3(A, B, E) f_4(A) f_5(A)$$

Given a variable and some evidence, a factor can be built by looking at the variable part of the CPT's in the network: `make-factor(var, e)`

> `make-factor(M, [j,m]) = f5(A)` because M only depends on A, m is an evidence
> $f_5(A) = \langle P(m \mid a), P(m \mid \neg a) \rangle = \langle 0.7, 0.01 \rangle$

The pointwise product of two factors f_1, f_2 yields a new factor f_3 such that

> variables are the union of the variables in f_1 and f_2
>
> elements are given by the product of the corresponding values in the two factors

So $f_1(A, B) \cdot f_2(B, C)$ gives $f_3(A, B, C)$

The ordering of variables can produce differences in efficiency.

Complexity Depends strongly on the structure of the network. Singly connected networks or polytrees are such that there is at most one undirected path between any two nodes (the Alarm network, for example). The time and space complexity in polytrees is linear in the size of network (the number of CPT entries). For multiply connected networks, variable elimination can have exponential time and space complexity in the worst case. In fact, inference in Bayesian networks includes as a special case propositional inference, which is NP-complete.

0.4.3 Probabilistic Reasoning Over Time

Reasoning in a changing world So far, we've been reasoning in a static world. In an evolving world an agent needs:

> A **belief state**: the states of the world which are possible
>
> A **transition model**: to predict how the world will evolve
>
> A **sensor model**: to update the belief state from perceptions

A changing world is modeled using a variable for each aspect of the world state **at each point in time** (**fluents**), and we use probabilities to quantify how likely a world is.
The transition and sensor model themselves may be uncertain:

> The transition model gives the probability distribution of the variables at time t, given the state of the world at past times
>
> The sensor model describes the probability of each percept at time t, given the current state of the world

The umbrella example *You are the security guard stationed at a secret underground installation. You want to know whether it's raining today, but your only access to the outside world occurs each morning when you see the director coming in with, or without, an umbrella.*

States and observations We view the world as a series of snapshots, or **time slices**, each of which contains a set of random variables, some observable and some not.

x_t will denote the set of **state variables**, unobservable (hidden) at time t

$E_t = e_t$ will denote the set of observations (**evidence variables**) at time t, with e_t being their values

The sequence starts at $t = 0$ (state variables starts at 0, evidences at 1), and the distance between time slices is fixed. With $a : b$ we will denote the sequence of integers from a to b, inclusive, and with $x_{a:b}$ the set of variables from x_a to x_b.
The umbrella world is represented by a sequence of evidence variables $E_t = \{U_t\}$ (whether the umbrella appears at time t) and state unobservable variables $x_t = \{R_t\}$ (it's raining):

State variables $R_0, R_1, R_2, \ldots$

Evidence variables $U_1, U_2, \ldots$

Transition model and simplifying absumptions The transition model describes how the world evolves, i.e. the probability distribution over the latest state variables, given the previous values starting from time 0: $P(x_t \mid x_{0:t-1})$, with $x_{0:t-1} = x_0, x_1, \ldots, x_{t-1}$
The sequence of states can become very large, unbounded as t increases. **Markov assumptions**: the transition model specifies the probability distribution over the lates state variables, given a **finite fixed number of previous states**.

First-Order Markov Chain: $P(x_t \mid x_{0:t-1}) = P(x_t \mid x_{t-1})$

Second-Order Markov Chain: $P(x_t \mid x_{0:t-1}) = P(x_t \mid x_{t-1}, x_{t-2})$

We additionally assume a **stationary process**, i.e. the conditional probability distribution is the same for all t: $P(x_t \mid x_{t-1})$ is the same for every t.

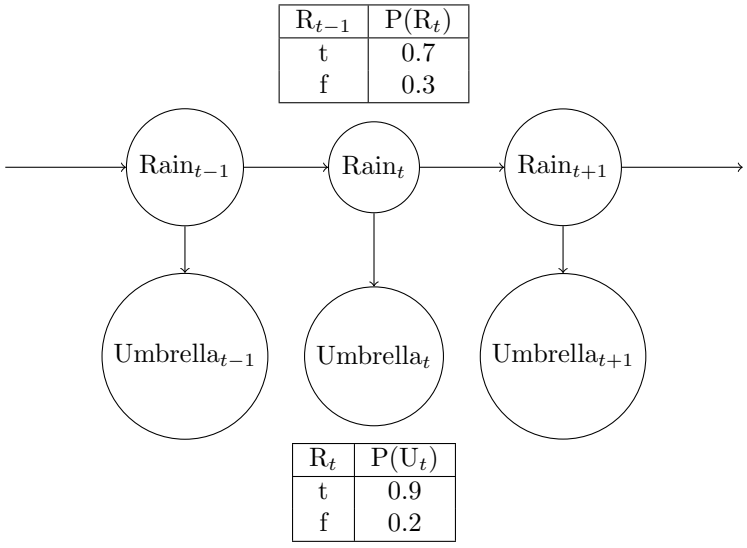
Sensor model assumptions The sensor model, under Markov sensor assumption, postulates that evidence only depends on the current state: $P(E_t \mid x_{0:t-1}, E_{0:t-1}) = P(E_t \mid x_t)$

Releasing the assumptions Assuming that "Rain" only depends on rain the previous day may be a too strong assumption. There are two ways to improve the accuracy of the approximation:

Increasing the order of the Markov chain, for example using a second-order assumption

Increasing the set of state variables: we could add Season, Temperature, Humidity, Pressure... as state variables. This may imply more computation for predicting state variables or adding new sensors.

Bayesian net for Umbrella



The transition model is $P(R_t \mid R_{t-1})$ (the rain depends on rain on the previous day, first-order Markov model) and the sensor model is $P(U_t \mid R_t)$ (the rain causes the umbrella to appear).
The **complete joint distribution** over all the variables, for any t, given the prior probability distribution at time 0 $P(x_0)$, is

$$P(x_{0:t}, E_{1:t}) = P(x_0) \prod_{i=1}^{t} P(x_i \mid x_{i-1}) P(E_i \mid x_i)$$

Which comes from $P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid \text{Parents}(x_i))$, with:

$P(x_0)$ is the initial state model over set of variables x

$P(x_i \mid x_{t-1})$ is the transition model under a first-order Markov assumption

$P(E_i \mid x_i)$ is the sensor model under Markov sensor assumption

Inference in temporal models

Basic inference task based on the temporal model:

Filtering or **state elimination**: computing the belief state (posterior probability distribution of state variables) given evidence from previous and current states. A subtask is the **likelihood of the evidence sequence** $P(x_t \mid e_{1:t})$

Prediction: computing the posterior distribution over a **future** state, given all the evidence to date $P(x_{t+k} | e_{1:t})$ with $k > 1$

Smoothing: computing the posterior distribution over a **past** state, given all the evidence up to the present. Looking back with the knowledge of today provides a more accurate estimate $P(x_k \mid e_{1:t})$ with $0 \leq k < t$

Most likely explanation/sequence: given a sequence of observations, we might wish to find the sequence of states that is most likely to have generated those observations

Learning: the transition and sensor models can be learned from observations

Filtering Estimation of the next state. A good filtering algorithm maintains a current state estimate and updates it, rather than going back over the entire history of percepts for each time. The filtering function f takes into account the state estimation computed up to the present and the new evidence

$$P(x_{t+1} \mid e_{1:t+1}) = f(e_{t+1}, P(x_t \mid e_{1:t}))$$

This process is called **recursive estimation** and is made of two parts:

Prediction: the current state distribution is projected forward from t to $t+1$: $P(x_{t+1} \mid e_{1:t})$

Update: then it's updated using the new evidence e_{t+1}: $P(e_{t+1} \mid x_{t+1})$

So it can be computed as

$$P(x_{t+1} \mid e_{1:t+1}) = \alpha \cdot P(e_{t+1} \mid x_{t+1}) \cdot \sum_{x_t} P(x_{t+1} \mid x_t) P(x_t \mid e_{1:t})$$

Prediction The task of prediction can be seen simply as filtering without the contribution of new evidence. The filtering process already incorporates a one-step prediction (the sum part).
In general, looking ahead k steps, at time $t + k + 1$, given evidence up to t

$$P(x_{t+k+1} \mid e_{1:t}) = \sum_{x_{t+k}} P(x_{t+k+1} \mid x_{t+k}) P(x_{t+k} \mid e_{1:t})$$

This involves only the transition model.
In the Umbrella, the predicted distribution of Rain converges to a fixed point $\langle 0.5, 0.5 \rangle$, after which remains constant for all time (**stationary distribution**). The time it takes to reach the fixed point is called **mixing time**: the more uncertainty there is in the transition model the shorter is the mixing time.

Smoothing Is the process of computing the posterior distribution of the state at some past time k, given a complete sequence of observations up to the present t: $P(x_k \mid e_{1:t})$ with $0 \leq k < t$. To compute it:

$$P(x_k \mid e_{1:t}) = \alpha f_{1:k} \cdot b_{k+1:t} = \alpha P(x_k \mid e_{1:k}) \cdot P(e_{k+1:t} \mid x_k)$$

where $\cdot$ is the pointwise multiplication, $f_{1:k} = P(x_k \mid e_{1:k})$ is the **forward message** and $b_{k+1:t} = P(e_{k+1:t} \mid x_k)$ is the **backward message** which can be computed by a recursive process that runs backward from t. This can be computed as:

$$P(e_{k+1:t} \mid x_k) = \sum_{x_{k+1}} P(e_{k+1} \mid x_{k+1}) P(e_{k+2:t} \mid x_{k+1}) P(x_{k+1} \mid x_k)$$

Viterbi Algorithm There is a recursive relationship between the most likely path to each state x_{t+1} and most likely paths to each previous state x_t. We can write a recursive equation similar to the one for filtering:

$$\max_{x_1,\ldots,x_t} P(x_1, \ldots, x_t, x_{t+1} \mid e_{1:t+1}) = \alpha P(e_{t+1} \mid x_{t+1}) \max_{x_t} (P(x_{t+1} \mid x_t) \max_{x_1,\ldots,x_{t-1}} P(x_1, \ldots, x_{t-1}, x_t \mid e_{1:t}))$$

With the forward message being $m_{1:t} = \max_{x_1,\ldots,x_{t-1}} P(x_1, \ldots, x_{t-1}, x_t \mid e_{1:t})$, the probabilities of the best path to each state x_t.
From those we can compute the probabilities of the extended paths at time $t+1$ and take the max. The **most likely sequence** can be computed in one pass. For each state, the best state that leads to it is recorded, so that the optimal sequence is identified.

Other approaches

Despite the disadvantage of the many probability values, Probability Theory and Bayesian Networks are the dominant approaches today. Other approaches that have been used are:

Rule-Based methods for uncertain reasoning in expert systems
Three good properties of classical logic-based rules:

Locality: in logical systems, from A and $A \Rightarrow B$ we can conclude B without worrying about any other rules. In probabilistic systems we have to consider all evidence.

Detachment: once B is proven, it can be used regardless of how it was derived, it *becomes detached from its justification.* Probabilities cannot be detached from evidence.

Truth-Functionality: the truth of complex sentences can be computed from the truth of the components. Probability combination doesn't work that way.

The idea is to attach a degree of belief to facts and rules and to combine and propagate them. The most famous examples is the **certainty factors model**.

Representing vagueness: fuzzy sets and fuzzy logic The theory of fuzzy sets is a way to specify how well an object satisfies a vague description property, like "being tall".
Fuzzy Logic is a way of reasoning about membership in fuzzy sets. A fuzzy predicate implicitly defines a fuzzy set. The standard rules are:

$T(A \wedge B) = \min(T(A), T(B))$

$T(A \vee B) = \max(T(A), T(B))$

$T(\neg A) = 1 - T(A)$

Fuzzy Control is a methodology for constructing control systems in which the mapping between real-valued input and output parameters is represented by fuzzy rules.

Fuzzication: continuous variables mapped into a set of linguistic variables with fuzzy values

Reasoning with rules expressed in terms of these fuzzy variables

Defuzzication: mapping the results into numerical values

Representing ignorance: Dempster-Shafer Theory

0.5 Rule-Based Systems

Rules are everywhere

Expert systems

Natural language processing

Rule-based programming

Decision support systems

Fuzzy control systems

Can be learned from decision trees and inductive methods...

Rules Knowledgebases As contractions of PROP and FOL
We assume: clausal form (conjunctive normal form, $\vee$ inside and $\wedge$ outside), unification (Most General Unifier), resolution method for PROP and FOL, completeness of the resolution method when used for refutation.
We will see **Horn clauses** i.e. rule-based knowledgebase, classical reasoning with rules (forward and backward), other forms of reasoning with rules.

Reasoning with Horn clauses By limiting the expressivity to a specific interesting subset of FOL, resolution procedures become much more manageable.
We limit the degree of uncertainty we can express by considering **clauses that have at most one positive literal** (**Horn clauses**). Two cases:

The knowledgebase is a set of definite Horn clauses (*exactly* one positive literal)
$\{\neg a_1, \ldots, \neg a_m, h\}$ also written as $a_1 \wedge \ldots \wedge a_m \Rightarrow h$

Rules when $m > 0$, for example: Child $\wedge$ Male $\Rightarrow$ Boy

Facts when $m = 0$, for example: Child

Goals or **queries** include only negative literals (**negative clauses**), for example $\{\neg Boy\}$

The limited expressivity is in the fact that we cannot express disjunctive conclusions. Propositional Horn clauses knowledgebases have linear time deduction algorithms.

This was discovered by looking at different strategies to perform resolution proofs in resolution theorem proving for FOL, i.e. strategies to explore the resolution graph. Looking at deduction as a search problem and at the resolution graph as your search space, a number of strategies and heuristics can be devised:

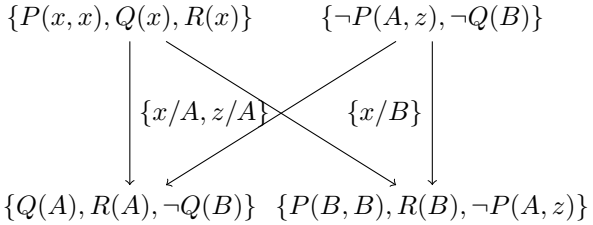
Where do we start from? (Initial state)

Can you avoid considering irrelevant clauses?

How do you select the next clauses to resolve?

It was found that effective resolution strategies are complete with respect to knowledgebase of a certain form: **Definite Horn Clauses**.

A resolution graph



Forward and backward Once we have a knowledgebase of facts and rules, the inference process can work in two directions:

Forward reasoning: use of rules from antecedent to consequent, to compute all the logical consequences of the initial facts. Also called **bottom-up**.

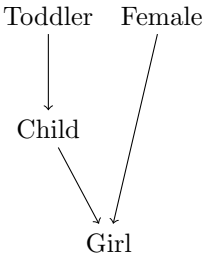
Backward reasoning: use of the rules from consequent to antecedent, until goals match initial facts. Also called **top-down**.

0.5.1 Forward Reasoning

Example of inference tree

1. Toddler
2. Baby ⇒ Child
3. Toddler ⇒ Child
4. Child, Male ⇒ Boy
5. Infant ⇒ Child
6. Child, Female ⇒ Girl
7. Female

Goal: Girl



Generalized Modus Ponens For the first-order case, inference is performed on the basis of the following rule which uses unification

$$\frac{p_1', p_2', \ldots, p_n', (p_1 \wedge p_2 \wedge p_n \Rightarrow q)}{q\gamma} \quad \text{(Generalized Modus Ponens)}$$

Where γ is a MGU $| \forall\, i\ \ p_i' = p_i\gamma$.
At each iteration new sentences are added to the knowledgebase, and it terminates when no new sentences are generated (**fixed point**) or the goal is found. This inferential process is **sound** (the rule is correct), **complete** for Definite Horn Clauses, and convergence is guaranteed for datalog databases (without function symbols, only constant) because there are a finite number of consequences.

Complexity issues The interpeter is required to perform many unification operations. If we have

r rules to try (in $\vee$)

n preconditions in each rule to be satisfied (in $\wedge$)

w facts in the knowledgebase (in $\vee$)

Everything repeated for c loops

$$(\vee \ldots (\wedge \ldots (\vee \ldots)))$$

We end up with $r \cdot n \cdot w \cdot c$ unification operations! Some ideas for possible improvements:

1. Finding all the facts that unify with a given pattern can be made more efficient with appropriate predicate indexing: hash table on the head of the predicate, or with a combined key head plus first argument, or a discrimination network on the antecedents of the rules
2. Each rule does not need to be checked again at each iteration, only the ones which are affected by recent addition

   These two rules are the strategies implemented by the RETE algorithm included in production systems and rule-based programming languages such as CLIPS
3. **Conjunct Ordering**: the order in which you list antecedents of rules is important for efficiency, each antecedent is a constraint and each rule a CSP:

   Can apply heuristics from CSP for ordering, for example MRV

Can try to write rules which correspond to CSP with a tree-structure

4. Proceeding forward may lead to derive many irrelevant facts

   Restrict the forward chaining to the relevant rules for the goal by proceeding backwards and marking them

   Magic sets in deductive databases

Production systems

Rule-based systems are one of the first paradigms for representing knowledge in artificial intelligence. Most expert systems are rule-based. Rules are used forward to produce new facts, hence the names *production* and *production systems*: general computational model guided by patterns, where the rule to be applied next is determined by the operation of pattern matching, a simplified form of unification.

Components A typical rule-based systems has four basic components:

A **list of facts** (no variables), the temporary working memory (WM)

Ordered facts: (predicate $\text{arg}_1 \ldots \text{arg}_k$)

Structured facts: (predicate (slot_1 arg_1) $\ldots$ (slot_k, arg_k))

A list of rules or **rule base**, a specific type of knowledgebase

(defrule rule_name["comment"] (pattern_1) $\ldots$ (pattern_k) $\Rightarrow$ (action_1) $\ldots$ (action_m))

An interpreter, called **inference engine**, which infers new facts or takes action based on the interaction of the working memory and the rules base.
The interpreter executes a *match-resolve-act* cycle:

Match: the Left-Hand-Sides (**LHS**) of all rules are matched against the contents of the working memory. This produces a conflict set: instantiations of all the rules that are satisfied/applicable.

Conflict-Resolution: one of the rules instantiations in the conflict is chosen for execution. If no applicable rule, the interpeter halts

Act: the actions in the Right-Hand-Side (**RHS**) of the rule selected are executed. These actions may change the contents of the working memory.
Then the cycle repeats.

A **conflict resolution strategy**

CLIPS C Language Integrated Production System, successor of OPS-5, developed by NASA and freely available on `www.clipsrules.net`
Matching rules (**activations**) are put in an **agenda** from where one will be selected:

Newly activated rules are added to the agenda and the agenda is reordered according to the salience of the rules

Among the rules of equal salience, the current conflict resolution strategy is used to determined the rule to be fired (activated, analogy with neurons)

The **depth** strategy is the standard default strategy for CLIPS: new rules on top. Different predefined conflict resolution strategies under user control:

breadth: newly activated rules placed below all rules of the same salience

simplicity: among rules of the same salience, less specific rules are preferred

complexity: among rules of the same salience, most specific rules are preferred

random: among rules of the same salience, choose at random (useful for debugging)

Advantages

Writing rules is very natural for experts or end-users

Modular architecture: the rule knowledgebase is separated from the inference engine

Modularity of rules and incremental acquisition of knowledge

Explanations: justification of conclusions is possible

General programming paradigm: extensible with user defined functions and object oriented programming for object definition and ontologies

Disadvantages

May be difficult to control the firing rules, unexpected behavior

Expressing knowledge as rules may be a bottleneck

0.5.2 Backward Reasoning

SLD Resolution Strategy A SLD derivation of a clause c from KB is a sequence S of clauses $c_1, \ldots, c_n$ such that

$c_1 \in$ KB

$c_n = c$

Each c_{i+1} is obtained by the resolution rule applied to c_i and some clause in KB

In backward reasoning systems, the SLD strategy is used by refutation, i.e. starting from the negation of the goal and trying to derive the empty clause. The SLD strategy is complete for Horn clauses

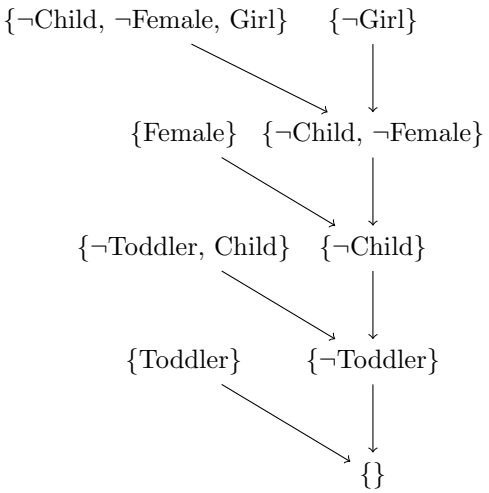
$$\text{KB} \vDash c \Rightarrow S \vdash_{SLD} c$$

Propositional Horn Clauses knowledgebases have linear-time deduction algorithms.

Example

1. {Toddler}
2. {¬Baby, Child}
3. {¬Toddler, Child}
4. {¬Child, ¬Male, Boy}
5. {¬Infant, Child}
6. {¬Child, ¬Female, Girl}
7. {Female}

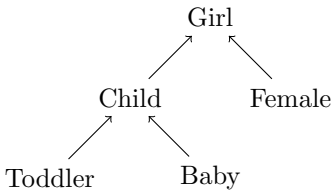
Goal: Girl ⇒ add {¬Girl}



The **SLD Goal Tree** will be

1. Toddler
2. Child ⇐ Baby
3. Child ⇐ Toddler
4. Boy ⇐ Child, Male
5. Child ⇐ Infant
6. Girl ⇐ Child, Female
7. Female

Goal: Girl



Logic Programs A logic program is a **set of definite Horn clauses** (facts and rules):

Declarative interpretation:

A

A :- $B_1, \ldots, B_n$, with A head and $B_1, \ldots, B_n$ body

Procedural interpretation:

A is true

$B_1, \ldots, B_n \Rightarrow A$, i.e. A is true if $B_1, \ldots, B_n$ are all true

The goal (query) is a negative clause whose logical meaning is $\neg(G_1 \wedge \ldots \wedge G_k)$ written as ?- G_1,...,G_k (**conjunction of subgoals**)

Example

parent(X, Y) :- father(X, Y)

parent(X, Y) :- mother(X, Y)

ancestor(X, Y) :- parent(X, Y)

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y)

father(john, mark)

father(john, luc)

mother(lia, john)

?- ancestor(lia, mark) (*negation of the goal*)

SLD Resolution Given a logic program and a goal $G_1, \ldots, G_k$, the SLD goal tree is constructed as follows. Each node of the tree correspond to a conjunctive goal to be solved:

The root is ?- $G_1, \ldots, G_k$

Let ?- $G_1, \ldots, G_k$ a node in the tree, then the node successors are obtained by considering the facts and rules in the program whose head unifies with G_1:

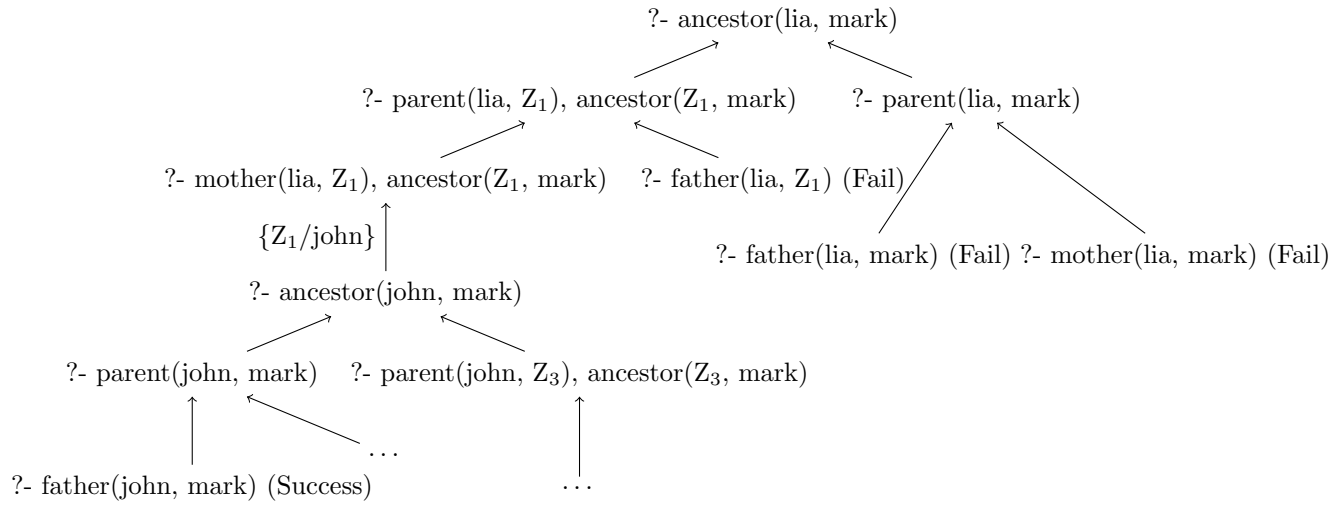
If A is a fact and $\gamma = \text{MGU}(A, G_1)$ then a descendant is the new goal ?- $(G_2, \ldots, G_k)\gamma$

If $A :- B_1, \ldots, B_m$ is a rule and $\gamma = \text{MGU}(A, G_1)$ then a descendant is the new goal ?- $(B_1, \ldots, B_m, G_2, \ldots, G_k)\gamma$

Nodes that correspond to empty clauses are **successes**

Nodes without successors are **failures**

SLD tree for the goal ancestor(lia, mark)



Computed answers In addition to yes or no goals, such as ?- ancestor(lia, mark), we can also have goals with variables such as ?- ancestor(X, mark).
The logical meaning of the query is:

KB $\models \exists\, X$ ancestor(X, mark)? (*Are there ancestors of Mark?*)

KB $\cup \neg\exists\, X$ ancestor(X, mark) unsatisfiable?

KB $\cup \{\neg$ancestor(X, mark)$\} \vdash_{RES} \{\}$?

In this case the expected answer are the values that X is bound to during the resolution proof, $X \in$ {lia, john}. Similarly, ?- ancestor(lia, Y) returns all the descendants of Lia ($Y \in$ {john, mark, luc}).

Properties The SLD resolution strategy is complete for definite Horn clauses. This means that if $P \cup \{\neg G\}$ is unsatisfiable, then at least one of the leaves of the SLD tree produces the empty clause (success). Moreover, trying to satisfy the subgoals in the order they appear is not restrictive, since in the end all of them must be satisfied.
When there are variables in the goal, the substitution that we obtain is the **computed answer**.
Completeness and efficiency are however influenced by: the order of expansion of the nodes (visit strategy of the SLD tree), the order in which we consider successor nodes at each level and the order of literals in the body.

0.5.3 Forms of Reasoning

Reasoning with Horn clauses

Knowledge engineering issues: querying the user, providing knowledge-level explanations, knowledge-level debugging

Beyond classical deduction (asking whether KN $\models a$): proving by contradiction and consistency-based diagnosis, complete knowledge assumption and non-monotonic reasoning, abductive reasoning.

Observations Observations arrive online by the user or from sensors. We cannot expect users or sensors to provide all relevant information. The solution is to introduce an "ask-the-user" mechanism into the backward procedure.

Define **askable** atoms: atoms for which the user is expected to provide an answer when unknown

In the top-down procedure, if the system isn't able to provide an atom and that atom is askable, it asks the user.

Knowledge-level Explanation Explanations are required and a strong pro of a rule-based decision support system with respect to other AI techniques. Support can be provided for three different kinds of requests:

How questions: how a fact was proven

Why questions: why the system is asking about something

Whynot questions: why a fact was not proven

Knowledge-level Debugging Given that the inference engine is correct, misbehaviors can only be caused by faulty knowledge. Debugging can be done by the domain expert, provided it's aware of the meaning of the symbols. We already discussed how why questions can be used to spot irrelevant questions, and four type of errors can be supported in debugging:

Incorrect answers

Missing answers

Infinite loops

Irrelevant questions

Proving by Contradiction We need an extension of the language to deal with integrity constraints of this form:

$$false \leftarrow a_1 \wedge \ldots \wedge a_k \equiv \neg a_1 \vee \ldots \vee \neg a_k$$

A definite Horn database is always satisfiable and a Horn database (including integrity constraints) can be unsatisfiable. Discovering a contradiction in the knowledgebase can be important in many applications. For these tasks you must declare a set of assumables, atoms that can be assumed in a proof by contradiction. The system can then collect all the assumable that are used in proving *false*. That is KB $\cup \{c_1, \ldots, c_r\} \vDash false$, with $C = \{c_1, \ldots, c_r\}$ **conflict** of KB. A possible answer is KB $\vDash \neg c_1 \vee \ldots \vee c_r$.
A **minimal conflict** is a conflict such that no strict subset is also a conflict.

Consistency-based Diagnosis The aim of consistency-based diagnosis is to determine the possible faults based on a model of the system and observations of the system.

Background knowledge provides the model of the system

You make the absence of faults assumable

Conflicts with observations can be used to derive what is wrong with the system

From set of conflicts you can derive a **minimal diagnosis**

A **consistency-based diagnosis** is a set of assumables that has at least one element in each conflict. A **minimal diagnosis** is such that no subset is a diagnosis.

Complete Knowledge Assumption What changes under a closed world assumption: everything which is true is in the knowledgebase, everything else is false. Knowledge is complete.
Suppose $a \leftarrow b_1, \ldots, a \leftarrow b_n$ (equivalent to $a \leftarrow b_1, \ldots, b_n$), noting that a is treated as $a \leftarrow true$.
Complete knowledge assumption: $a \rightarrow b_1, \ldots, b_n$
Clark's Completion: $a \leftrightarrow b_1, \ldots, b_n$
Under Clark's completion, if there are no rules for a then $a \leftrightarrow false$, i.e. $\neg a$
With this, the system is able to derive negations. We extend the language of definite Horn clauses with **not**. Note that this negation is not the same as logical $\neg$, so we use a different notation. In backward reasoning systems this **not** can be implemented by **negation as failure**.

Negation as Failure The top-down procedure for negation as failure is defined as follows: **not**p can be added to the set of consequences whenever proving p fails. The failure must occur in finite time, there is no conclusion if the proof procedure does not halt.
Negation as failure is the same as a proof of a logical negation from the KB under Clark's completion KB_c

$$\text{KB}_c \vDash \neg a \Leftrightarrow \text{KB} \nvdash a$$

It's a form of **non-monotonic reasoning**: you can express **defaults**.

Abductive Reasoning Abduction is a form of reasoning which consists in providing explanations for observations. The term is used to differentiate from deduction, which involves determining what logically follows from a set of axioms, and induction, which involves inferring general relationship from examples.
Given

a knowledgebase KB, which is a set of Horn clauses

a set A of assumable atoms, the building blocks of hypotheses

an **explanation** of g is a set $H \subseteq A$ such that:

$\text{KB} \cup H \not\models false$

$\text{KB} \cup H \models g$

There can be multiple explanations, you usually want the minimal explanation.

Example of abduction Consider the knowledgebase

Alarm $\leftarrow$ Tampering

Alarm $\leftarrow$ Fire

Smoke $\leftarrow$ Fire

If Alarm is observed, there are two possible minimal explanations: {Tampering} and {Fire}. If Alarm $\wedge$ Smoke is observed, the only possible minimal explanation is {Fire}

0.5.4 Prolog

Prolog is the most widely used **logic-programming language**. We already introduced the syntax and the execution model (SLD resolution).

The declarative semantics is given by Horn clause knowledgebases

The procedural semantics is given by a specific strategy for exploring SLD trees:

Successors are generated in the order they appear in the logic program

The SLD tree is generated left-to-right **depth-first**

The **occur check is omitted** from Prolog's unification algorithm

Therefore Prolog is neither complete (since the depth-first strategy is not complete) nor correct.
Prolog uses the database semantics rather than the first-order semantics: unique name assumption, closed world assumption (a form of nonmonotonic reasoning, i.e. negation as failure). Also Prolog is a full fledged programming language: efficient implementation of the interpreter, built-in functions for arithmetic and list manipulation, the burden of controlling the flow of execution is on the programmer.

A Prolog interpreter is similar to FOL-BC-Ask but more efficient. It relies on a global data structure: a **stack of choice points**. Logic variables in a path remember their binding in a **trail**. When failure occurs, the interpreter backtracks to a previous choice point and undoes the bindings of the variables.
Prolog compiles into an intermediate abstract machine (**Warren Abstract Machine**), and parallelism can be exploited in two ways:

OR-parallelism, a goal may unify with many different rules in the knowledge base

AND-parallelism, comes from the possibility of solving each conjunct in the body of an implication in parallel but with consistent substitutions

Redundant Computations In forward chaining we do not have the problem of repeated computation since, once a solution to a smaller problem is computed, it's remembered as part of the solution to the bigger problem. We can obtain a similar saving in backward systems with a technique called **memoization**, which consists in caching solutions to subgoals and reusing them. Tabled logic programming systems use efficient storage and retrieval mechanisms to perform memoization.

Data Types

Atoms: identifiers with lowercase initial (tom, x-1,...), string of characters ("Tom", "Sarah Johnes",...), string of special characters (<==>, ::=, ...)

Numbers: integers and reals

Variables: identifiers with uppercase initial (Tom, X1, Result,...), anonymous variables (_ as in hasChild(X) :- parent(X,_))

Structured Objects Functions with arguments (functor applied to arguments) and lists.

Arithmetic Basic arithmetic operations and comparison operators.

Procedural Control of Reasoning Declarative encoding of knowledge and general mechanisms for deduction can be very inefficient. For efficiency we must have some domain dependent control on the reasoning process. Given a knowledgebase of facts and rules, how to make the most effective use of the rules? The choice on the order of the rules makes a difference in performance. Prolog takes ordering of clauses and subgoals very seriously: **the burden is on the programmer**.

Controlling backtracking Prolog will automatically backtrack if this is necessary to satisfy a goal. Uncontrolled backtracking however may cause inefficiency in a Prolog program. With CUT we can control backtracking.
In a case where given a clause of the form "G :- T,R", the goal T is needed only as a test for the applicability of a subgoal R. If R fails, we do not want to backtrack to T nor try any other alternative for G. The pattern

G :- T, !, R

G :- S

Is equivalent to `if T then R` $\Rightarrow$ `G else S` $\Rightarrow$ `G` and is more efficient than

G :- T, R

G :- $\overline{T}$, S

with $\overline{T}$ a goal mutually exclusive with T.
In general, G :- $T_1, \ldots, T_m$, !, $G_1, \ldots, G_n$ means that once $T_1, \ldots, T_m$ have been established we can commit to the rest of goals without looking for alternatives.
CUT has advantages and drawbacks:

With CUT we can often improve the efficiency of the program. The idea is to explicitly tell Prolog: do not try other alternatives because they are bound to fail.

Using CUT we can specify mutually exclusive rules, so we can add expressivity to the language.

The main disadvantage is that we can lose the correspondence between the declarative and procedural meanings of the programs.
Green cuts do not change the meaning, red cuts do and we have to be careful.

p :- a, b. p :- c.	meaning $p \Leftarrow (a \wedge b) \vee c$	p :- a, !, b. p :- c.	meaning $p \Leftarrow (a \wedge b) \vee (\neg a \wedge c)$	p :- c. p :- a, !, b.	meaning $p \Leftarrow c \vee (a \wedge b)$

Negation as failure The unary predicate **not** is such that not(P) is true if P fails. This new type of goal, not(G), is understood to succeed when the goal G fails and to fail when the goal G succeed. Failure must occur in a finite number of steps.

0.5.5 Constraint Logic Programming

CLP combines the constraint satisfaction approach with logic programming, creating a new language where a logic program works along a specialized constraint solver.
The basic Prolog can be seen as a very specific constraint satisfaction language, where the constraints are of a limited form. Prolog is extended introducing other types of constraints: CLP(X) differ in the domain and type of constraints they can handle

CLP(R) constraints on real numbers

CLP(Z) for integers

CLP(Q) rational numbers

CLP(B) booleans

CLP(FD) finite domains

A CLP solution is the most specific set of constraints on the variables that can be derived from the knowledgebase. A specific solution if the constraints are tight enough.

0.5.6 Meta Interpreters

A meta interpreter for a language is an interpreter that is written in the language itself. Prolog has powerful features for writing meta programs because Prolog treats programs and data both as terms. A program can be input to another program.
One can write meta interpreters for various applications, extending the implementation of Prolog in different directions:

exploring different execution strategies for the interpreter (breadth-first, limited depth search...)

generating proof trees, expert system shells, explanations...

implementing new languages

0.5.7 Answer Set Programming

Answer Set programming/Prolog is a language for knowledge representation and reasoning based on the stable model semantics of logic programs (an alternative, more declarative, semantics). In includes, in a coherent framework, ideas from:

declarative programming, expressive knowledge representation language

deductive databases (forward reasoning) and also disjunctive databases

syntax and semantics of standard Prolog as a special case, but also disjunction, "classical" and "strong" negations, constraints...)

nonmonotonic logic (defaults, preference models, autoepistemic logic)

Balance between expressivity, ease of use and computational effectiveness. Answer Set inference engines/solvers exist (often relying on SAT solvers).
Useful real world application: decision support systems of the Space Shuttle, team building, configuration systems, bioinformatics, lingustics...

Syntax Answer sets are sets of beliefs (AS) that can be justified on the basis of the program. We assume sorted signatures from classical logic, including natural numbers and arithmetic operators.
Literals $p(t)$ and $\neg p(t)$ where t are sequences of terms (contrary literals in this case).
A **rule** is an espression of the form:

$$l_0 \text{ or } \ldots \text{ or } l_k \leftarrow l_{k+1}, \ldots, l_m, \text{ not } l_{m+1}, \ldots, \text{ not } l_n$$

$\text{head}(r) = \{l_0 \text{ or } \ldots \text{ or } l_k\}$

$\text{body}(r) = \{l_{k+1}, \ldots, l_m, \text{ not } l_{m+1}, \ldots, \text{ not } l_n\}$

$\text{pos}(r) = \{l_{k+1}, \ldots, l_m\}$

$\text{neg}(r) = \{\text{not } l_{m+1}, \ldots, \text{ not } l_n\}$

With "not" being the negation as failure or **default negation**, different from classical $\neg$ which is also used, and "or" being the **epistemic disjuction**, different from classical $\vee$.

Special cases

Normal logic programs or **nlp**: programs without $\neg$ (the strong negation) and without "or" ($k = 0$)

$$l_0 \leftarrow l_1, \ldots, l_m, \text{ not } l_{m+1}, \ldots, \text{ not } l_n$$

Constraint: when head$(r) = \{\}$

$$\leftarrow l_1, \ldots, l_m, \text{ not } l_{m+1}, \ldots, \text{ not } l_n$$

Fact: when body$(r) = \{\}$

$$l_0 \text{ or } \ldots \text{ or } l_k$$

A **ASP** (Answer Set Prolog/program) **is a pair $\{\sigma, \Pi\}$ where σ is a signature and Π is a collection of logic programming rules over σ.** The signature σ may be derived from the program, $\sigma(\Pi)$, rather than being given explicitly. It consists of all the constants which appear in the program.
Example of an ASP program, Π_0:

$q(a, 1)$

$q(b, 2)$

$p(X) \leftarrow K + 1 < 2, q(X, K)$

$r(X) \leftarrow \text{not } p(X)$

for the signature σ:

Sorts/constants: $\tau_1 = \{a, b\}$ and $\tau_2 = \{N\}$ with N natural numbers set (note: no functions)

Predicates: $p(\tau_1)$, $q(\tau_1, \tau_2)$, $r(\tau_1)$ and the standard relation $<$ on N

Ground instances of a program Terms, literals and rules are called ground if they contain no variables and no symbols for arithmetic functions.

Herbrand Universe: the set of all ground instances of terms

Herbrand Base: the set of all ground atomic sentences

A program $gr(\Pi)$ consisting of all ground instances of all rules of Π is called the ground instantiation of Π: $gr(\Pi_0)$

$q(a, 1)$

$q(b, 2)$

$p(a) \leftarrow 1 < 2, q(a, 0)$

$p(a) \leftarrow 2 < 2, q(a, 1)$

$\ldots$

$p(b) \leftarrow 1 < 2, q(b, 0)$

$p(b) \leftarrow 2 < 2, q(b, 1)$

$\ldots$

$r(a) \leftarrow \text{not } p(a)$

$r(b) \leftarrow \text{not } p(b)$

ASP Partial Interpretations A (partial) interpretation is a consistent subset of the Herbrand base.
Partial interpretation S: a consistent set of ground literals over σ:

Given two contrary literals l and l^-: l is true if $l \in S$, l is false if $l^- \in S$, otherwise l is unknown in S

An extended literal not l is true in S if $l \notin S$, otherwise not l is false in S

A set of extended literals U, understood as their conjunction $(l_0, \ldots, l_k)$, is true if all of them are true in S, false if at least one of them is false, unknown otherwise

A disjunction of literals (l_0 or ... or l_k) is true if at least one of them is true, false if all of them are false, unknown otherwise. Note that $\neg(l_0, \ldots, l_k)$ is treated as (l_0^- or ... or l_k^-) while $\neg(l_0$ or ... or $l_k)$ is treated as $(l_0^-, \ldots, l_k^-)$

Rule Satisfaction: S satisfies a rule r if S satisfies head(r) or does not satisfies its body.

ASP Semantics The semantics is given for ASP ground programs: this approach is justified by the domain closure assumption, where all objects in the domain of discourse are in teh signature of Π, no other. The answer set semantics of a logic program Π assign to Π a **collection of answer sets** S. Each answer set S obeys to the following principles:

Consistency: S must satisfy the rules of Π

Minimality: a rationality principle, "*do not believe anything you are not forced to believe*"

Semantics is given in two steps: programs without default negation (part 1) and arbitrary programs (part 2).

Consistency of logic programs A logic program is called consistent if it has an answer set.
Inconsistency may be due to improper use of **logical negation** ($\neg$). We can transform a ASP program Π to one without negation Π^+ (the positive form) as follows:

For each predicate symbol p we introduce a symbol p' with the same arity

We replace each $l = \neg p(t)$ with $p'(t)$, its positive form, l^+. If l is positive, $l = l^+$

A logic program Π can be transformed in its positive form Π^+ by replacing each rule with l_0^+ or ... or $l_k^+ \leftarrow l_{k+1}^+, \ldots, l_m^+$, not $l_{m+1}^+, \ldots,$ not l_n^+ and adding the constraints $\leftarrow p(t), p'(t)$ for each p' added.
Property 1: a set S of literals of $\sigma(\Pi)$ is an answer set of $\Pi \Leftrightarrow S^+$ is an answer set of Π^+.

A logic program can be inconsistent for the use of the **default negation**.
In general, the problem of checking inconsistency is undecidable, or decidable but very complex for finite domains. The theory helps us in making simplifying assumptions to guarantee consistency.
Level Mapping: a function $||\ ||$ which maps ground atoms in P (the Herbrand base of P) to natural numbers. If D is a disjunction or conjunction of literals, $||D||$ is defined as the minimum level of atoms occurring in literals from D' (the positive form). This also implies that $||\neg A|| = ||A||$. A logic program Π is **locally stratified** if it does not contain $\neg$, and there is a level mapping on the grounded version of Π ($gr(\Pi)$) such that for every rule:

$\forall\, l \in \text{pos}(r), ||l|| \leq ||\text{head}(r)||$

$\forall\, l \in \text{neg}(r), ||l|| < ||\text{head}(r)||$

as to impose an higher level when there is a not.
If a program is locally stratified and, in addition, for any predicate symbol p we have $||p(t_1)|| = ||p(t_2)||$ for any t_1, t_2, the program is called **stratified**. A stratified program is also locally stratified. A program without not is stratified. Properties:

A locally stratified program is consistent

A locally stratified program without disjunction has exactly one answer set

The above conditions hold by adding to a locally stratified program a collection of closed world assumptions of the form $\neg p(X) \leftarrow$ not $p(X)$

Reasoning methods for Answer Set Programming The choice of the algorithms used depend on the structure of problems and the type of queries. Two examples:

Acyclic Programs
A nlp Π is called acyclic if there's a level mapping of Π such that for every rule r of $gr(\Pi)$ and every literal l which occurs in pos(r) or neg(r), $||l|| < ||\text{head}(r)||$

If Π is acyclic then the unique answer set of Π is the unique Herbrand/classical model of Clark's completion of Π, comp(Π)

SLDNF resolution-based interpreter of Prolog will always terminate on atomic queries and produce the intended answers.

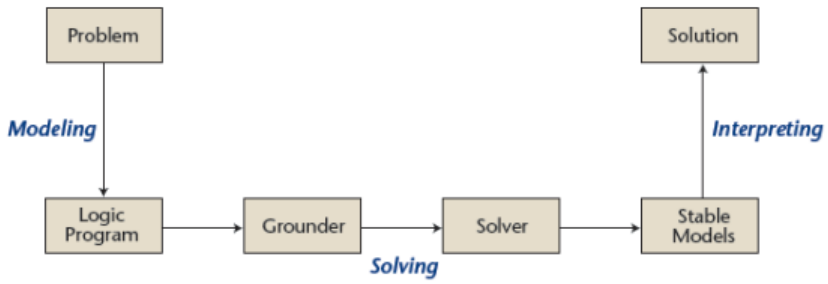
Tight Programs
A nlp Π is called tight if there is a level mapping of Π such that for every literal rule r of $gr(\Pi)$ and every literal $r \in \text{pos}(r)$, $\text{head}(r) > l$. Acyclic programs are tight but not viceversa.
If a program Π is tight then answer sets of Π are the model of the Clark's completion of Π. The problem of computing answer sets can be reduced to SAT for propositional formulas and a SAT solver can be used. Also, in the case of non tight programs there are theoretical results that allow translating into SAT program even if very large.

Answer Set Solvers These theoretical results have led to the development of answer set solvers such as ASET, CMODELS... which are based on (possibly multiple) calls to propositional solvers.
Traditional answer set solvers typically have a two level architecture:

Grounding step: compute $gr(P)$
Grounders can be integrated or provided as separated modules. The size of the grounding, even if partial, is a major bottleneck.
Smart and lazy grounding techniques have been developed.

Model search: the answer sets of grounded (propositional) program are computed

The workflow of answer set programming:



Knowledge Representation with Answer Set Programming `todo`

0.6 Planning

0.6.1 Classical Planning

Planning combines two major areas of AI: search and logic. By representing actions and change in logic (as in the situation calculus) we can cast the problem of planning as: a SAT problem in the PROP case or a theorem proving in the FOL case.
In the classical planning, we'll introduce a **restricted language** suitable for describing planning problems. This language allows to **circumvent representation and computational problems** by resorting to a **restricted FOL representation**, allowing **specific descriptions of actions and states** (including goals) so that **special algorithms and heuristics can be developed**.

Problem Solving Agent

Actions: implicitly defined by a function defining successor states

Goal: test Goal(state) → {true, false}

Planning: to obtain, by heuristic search, a path in the state graph, leading from the initial state to a Goal. In needs good domain-specific heuristics.

Planning Agent

Has an explicit representation of the goal state, of action and their effects

It is able to inspect the goal and decompose it, and make abstractions

It can work freely on the plan construction, manipulating goals and plans

Some general heuristics and algorithms for planning become possible, leveraging on this representation

A planning agent can be more efficient.

Representation for Planning in PROP Let's consider the **Wumpus world**: the agent moves in a labyrinth looking for gold, can shoot an arrow, grab the gold... trying to avoid pits and the Wumpus. We need a (huge) number of propositional sentences in clausal form, that include:

Init, a collection of assertion about the initial state. Example: $L^0_{1,1}, W_{2,3}, G_{3,1} \ldots$

Transition^1, ..., Transition^t, the succesor-state axioms for all possible fluents at each time up to some maximum time t
Examples: HaveArrow$^{t+1} \Leftarrow$ (HaveArrow$^t \wedge \neg$Shoot^t), or another example $L^{t+1}_{x+1,y} \Leftrightarrow (L^t_x \wedge \text{MoveUp}^t_y)$ replicated for 4 action, $n \times n$ locations, t times...

State constraints like non ubiquity: $L^z_{x,y} \Rightarrow \neg L^z_{x',y'}$

Action exclusion axioms: $\neg A^t_i \wedge \neg A^t_j$ for any pair of actions A_i and A_j and any time t

Finally, the assertion that the goal is achieved at time t: HaveGold$^t \wedge$ ClimbedOut^t

Give all this in input to a SAT solver in clausal form and look for a model.

```
function SATPLAN(init, transition, goal, T_max) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
          T_max, an upper limit for plan length

  for t = 0 to T_max do
    cnf ← TRANSLATE-TO-SAT(init, transition, goal, t)
    model ← SAT-SOLVER(cnf)
    if model is not null then
      return EXTRACT-SOLUTION(model)
  return failure
```

The planning problem is translated into a CNF sentence for increasing values of t until a solution is found or the upper limit to the plan length is reached.

Planning as Theorem Proving in FOL

Reminder of Situation Calculus in FOL A special ontology made of situations, fluents, actions...
Problems in defining actions and their effect (frame problem) partially solved by defining state successor axioms, one for each fluent. For example, for the blocks world

$$
\begin{aligned}
&Clear(y, Result(a, s)) \Leftrightarrow \\
&\quad [On(x, y, s) \wedge Clear(x, s) \wedge Clear(z, s) \wedge x \neq z \wedge a = move(x, y, z))] \vee \\
&\quad [On(x, y, s) \wedge Clear(x, s) \wedge (a = unstack(x, y))] \vee \\
&\quad [Clear(y, s) \wedge (a \neq move(z, w, y)) \wedge (a \neq stack(z, y))]
\end{aligned}
$$

Effect of a sequence of actions Result : $[A^*] \times S \to S$

Result($[], s$) = s

Result($[a \mid \text{seq}], s$) = Result(seq,Result(a, s))

Planning is the generation of a sequence of actions, a plan p, to reach the goal G. This amounts to proving that $\exists\, p\ G(\text{Result}(p, s_0))$
The task is made complex by different sources of non-determinism:

The length of the sequence of actions is not known in advance

Frame axioms may infer many things that are irrelevant

We need to resort to ad hoc strategies

A general theorem prover is inefficient and semi-decidable, completeness is not guaranteed

We do not have any guarantee of the efficiency of the generated plan

Definition of Classical Planning In classical planning we assume **fully observable**, **deterministic** and **static** environments with **single agents**. We also assume a **factored representation**: a state of the world is represented by a collection of variables.

PDDL

The **Planning Domain Definition Language** is a specialized language for describing planning problems and in particular:

States

Applicable actions (Actions(s)) and transition model (Result(s, a)) through action schemas

Initial and goal states

States Conjunction/set of fluents, ground positive atoms with no variables and no functions.
Examples: At(Truck$_1$, Melbourne) $\wedge$ At(Truck$_2$, Sydney)
The database semantics is used:

Closed World Assumption: fluents that are not mentioned are considered false

Unique Name Assumption: distinct name to refer to distinct individuals

Actions Defined by a set of action schemas that implicitly define the Actions(s) and their Result(s, a). Actions are defined in a way that avoids the frame problem, since they require that you carefully specify all the changes. All **the rest it is assumed to persist**. This because in most problems the things that change are very few compare to the ones that do not.

Action Schemas Correspond to parametric actions or operators (a lifted representation). Example of action schema: flying from a location to another one:

Action(Fly(p, from, to), % p (a plane); from (departing airport); to (destination)
PRECOND: At(p, from) $\wedge$ Plane(p) $\wedge$ Airport(from) $\wedge$ Airport(to)
EFFECT: $\neg$At(p, from) $\wedge$ At(p, to))

p, from, to are universally quantified variables that need to be instantiated. An instance:

Action(Fly(P1, SFO, JFK),
PRECOND: At(P1, SFO) $\wedge$ Plane(P1) $\wedge$ Airport(SFO) $\wedge$ Airport(JFK)
EFFECT: $\neg$At(P1, SFO) $\wedge$ At(P1, JFK))

Restriction: any variable in the effect must also appear in the precondition.

PRECOND: a list of preconditions to be satisfied in s for the action to be applicable.

$a \in \text{Actions}(s) \Leftrightarrow s \vDash \text{PRECOND}(a)$

I.e. positive literals are in s and negated literals are not in s

EFFECT: the successor state s' is obtained from s as a result of the action by:

Adding positive effects to s: add list, ADD(a)

Removing negative effects from s: delete list, DEL(a)

Result$(s, a) = (s-$ DEL$(a)) \cup$ ADD(a)
In the example, Fly(P1, SFO, JFK) would remove At(P1, SFO) and add At(P1, JFK).

Initial State A collection of ground positive atoms.

Goal A conjunction of literals (positive or negative) that may contain variables, for example At(p, SFO) $\wedge$ Plane(p) is the goal of having any plane to SFO.
Variables are treated as existentially quantified.

Solution The problem is solved when we can find a sequence of actions that end in a state s that entails the goal.

Algorithms and Heuristics

Decision Problems for Planning, Complexity PlanSAT (does a plan exist?) and Bounded PlanSAT (is there a solution of length k or less?): both problems are decidable for classical planning

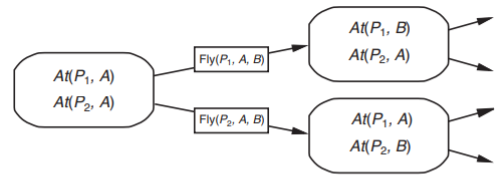
PlanSAT without functions, since the number of states is finite

Bounded PlanSAT: always decidable, also with functions

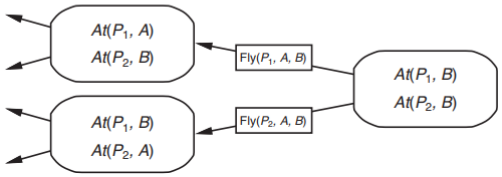
Theoretical complexity for both is very high (PSPACE). If we disallow negative effects, both problems are still NP-hard. If we also disallow negative preconditions, PlanSAT reduces to P.
This means we need sub-optimal solutions and good heuristics.

Planning as a State-Space Search Nodes in the search space are states, the arcs are the actions.

Progression Planning: a forward search from the initial state to the goal state. Believed to be inefficient: prone to exploring irrelevant actions, and planning problems often have very large state spaces



Regression Planning: backward search from the goal state to the initial state, the first approach tempted.



Regression Planning We start with the goal, a conjunction of literals, describing a set of worlds. The PDDL representation allows to regress actions, i.e. to find a state g' from where it is possible to reach the goal with some action a (achieving some preconditions of g).

$g' = (g-$ ADD$(a)) \cup$ Precond(a)

We do not say anything about DEL(a).

Relevant Actions and Relevant Search Actions that are useful to reach a goal are defined relevant. Relevant actions contribute to the goal (achieve some subgoal) but must be the last step to the solution (i.e. not have effects that negate some element of the goal).
This is achieved by computing the **Most General Unifier** (Θ). Formally: given a goal g containing a literal g_i, an action is a relevant action towards g if

action schema A has an effect literal e such that Unify$(g_i, e) = \Theta$

$a =$ SUBST(Θ, A)

there is no effect in a that is the negation of a literal in g

One of the earlier planning systems was a linear regression planner called STRIPS.

Heuristics for Planning Given the factored representation, we can devise good general heuristics for planning. Remember:

> A^* is a best first algorithm which uses an evaluation function $f(s) = g(s)+h(s)$ where h is an admissible heuristic (i.e. an heuristic functions that never overestimate the distance from the solution, $\forall\, s\;\; h(s) \leq h^*(s)$)
>
> A^* is optimal
>
> **Problem relaxation** is a common technique for finding admissible heuristics, it consists in looking at a problem with less constraints and computing the cost of the solution in the relaxed problem. This cost can be used as an admissible heuristics for the relaxed problem.

A state space is a graph, where the nodes are states and edges are actions. There are two strategies to relax the problem:

> Add more arcs to the graph, so it's easier to find a path to the goal (ignore preconditions heuristics or ignore delete list heuristics)
>
> Group multiple nodes together, forming an abstraction of the state space (with fewer states is easier to search)

Ignore Precondition Heuristic It drops all preconditions from actions. Every action is applicable in any state, any single goal literal can be satisfied in one step or there is no solution.
The number of steps to solve a goals is approximated by the number of unsatisfied subgoals, but: one action can achieve more than one subgoal (non admissible estimate) or one action may undo the effect of another one (admissible).
An accurate heuristic is the following:

> Remove all preconditions and all effects except those that are literals in the goal
>
> Count the minimum number of actions required such that the union of those actions effects satisfies the goal (a problem of set-cover)
> NP-complete but greedy approximations exist.

As an alternative, ignore only some selected preconditions.

Ignore Delete List Heuristic Assume all goals and preconditions contain only positive literals (not a very strong assumption, you can rename negative literals with new positive ones).
Remove the delete lists from all actions (i.e. removing all negative literals from effects): no action will ever undo the effects of other actions, so there is a monotonic progress towards the goal.
Use the length of the solution as an admissible heuristic.
Still NP-hard to find the optimal solution of the relaxed problem, this can be approximated in polynomial time with hill-climbing.

State Abstraction In order to reduce the number of states, we need other forms of relaxation: state abstraction reduces the number of states.
A state abstraction is a many-to-one mapping from the states in the original representation to a more abstract representation. The common strategy is to ignore some fluents.
The cost of the solution to this smaller problem can be used as an admissible heuristic.

Decomposition Divide a problem into parts, solve each one independently an then combine the subplans. The subgoals independence assumption is that the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal independently. This assumption can be: optimistic (admissible, there are negative interactions) or pessimistic (inadmissible, subplans contain redundant actions, **Sussman anomaly**)

0.6.2 Planning Graph

A data structure that can be a rich source of information:

> Can be used to give better heuristic estimates to employ in conjunction with search algorithms
>
> It's the search space of an algorithm called **GraphPlan**

A search tree is exponential in size, and a planning graph is a polynomial size approximation of the search tree that can be constructed quickly.
The planning graph can't answer definitively whether G is reachable from S_0, but

It may discover that the goal is not reachable

It can estimate how many steps it takes, in the most optimistic case, to reach G, so it can be used to derive an admissible heuristic

Definition A planning graph works just for propositional planning problems with no variables.
A planning graph is a directed graph which is **built forward** and is **organized into levels**:

A level S_0 for the initial state, representing each fluent that holds in S_0

A level A_0 consisting of nodes for each ground action applicable in S_0

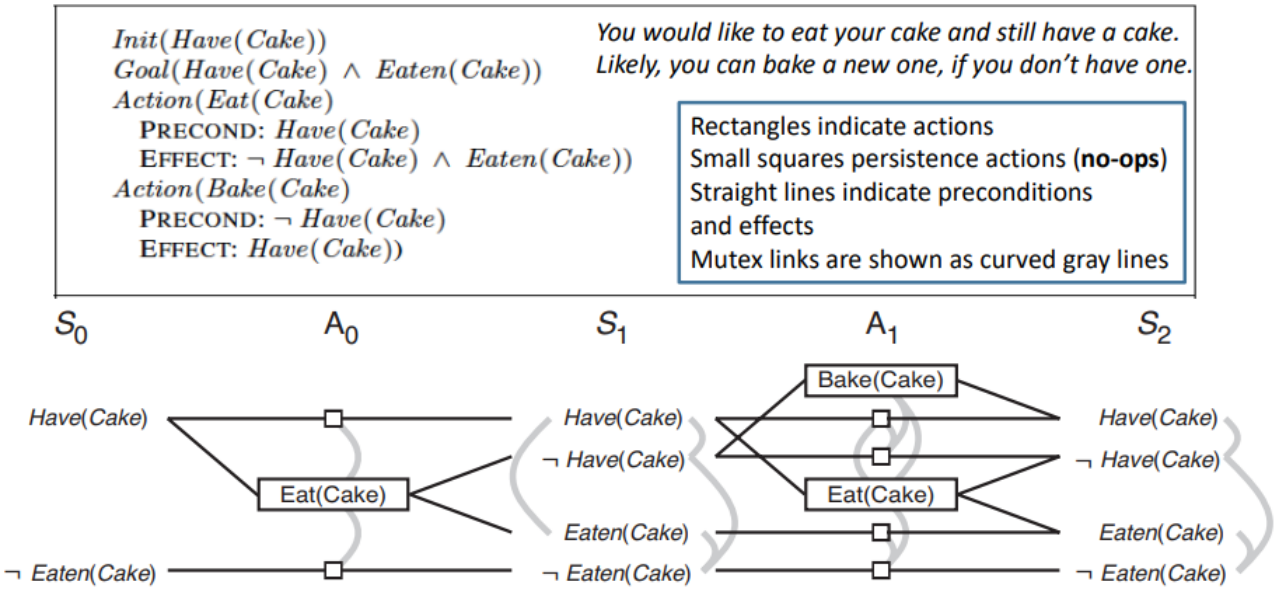
Alternating levels S_i followed by A_i are built until we reach a terminating condition.

S_i contains all the **literals that could hold** at time i (even contrary literals, like P and $\neg P$). A_i contains all the **actions that could have their precondition satisfied** at time i.
Mutual exclusion links (**mutex**) connect incompatible pairs of literals and actions:

Mutex between literals mean that two literals cannot appear in the same belief state

Mutex between actions mean that two actions cannot occur at the same time

Example



Mutex Computation Mutex relations between actions:

Inconsistent Effects: one action negates an effect of another action.
E.g. persistence of Have(Cake) and Eat(Cake) have inconsistent effects

Interference: one of the effect of one action is the negation of a precondition of the other.
E.g. Eat(Cake) interferes with the persistence of Have(Cake)

Competing Needs: one of the preconditions of one action is mutually exclusive with a precondition of the other.
E.g. Bake(Cake) and Eat(Cake)

Mutex relations between literals at the same level:

If one is the negation of the other

Inconsistent Support: if each possible pair of actions that could achieve the two literals is mutually exclusive.
E.g. Have(Cake), produced by noOp, is mutex with Eaten(Cake), produced by Eat(Cake).

Properties of the Planning Graph

Each level S_i represent a set of possible belief states. Two literals connected by a mutex belong to different belief states

The levels, alternating Ss and As, are computed until we reach a point where two consecutive levels are identical. The graph for the cake **levels off** at S_2

The level j at with a literal first appears is never greater than the level at which it can be achieved. We cal this the **level cost** of a literal/goal

The process of constructing the planning graph does not require choosing among actions and is very fast

A planning graph is polynomial in the size of the planning problem: an entire graph with n levels, a actions, l literals has size $O(n(a+l)^2)$. The same for time complexity.

Using the Planning Graph for Heuristic Estimation The information that can be extracted from the planning graph is:

If any goal literal fails to appear in the final level of the graph, then the problem is unsolvable

We can **estimate the cost of achieving a goal** literal g_i by its level cost. **This estimate is admissible**

A better estimate can be obtained by **serial planning graphs**: by enforcing only one action at each level (adding mutex)

Estimating the heuristic cost of a conjunction of goals:

Max-level heuristic: the maximum level cost of any of the sub-goals (**admissible**)

Level sum heuristic: the sum of the level costs of the goals (can be inadmissible when goals are not independent, but it may work well in practice)

Set-level heuristic: finds the level at which all the literals in the goal appear together in the planning graph, without any mutex between pairs of them (admissible, accurate, but not perfect)

Planning Graph as Relaxed Problem The planning graph can be seen as a relaxed problem with the following characteristics:

If there exists a plan with i actions levels that achieves g, then g will appear at level i. If g doesn't appear then there's no plan

But not viceversa: the fact that a g appears does not mean that there's a plan

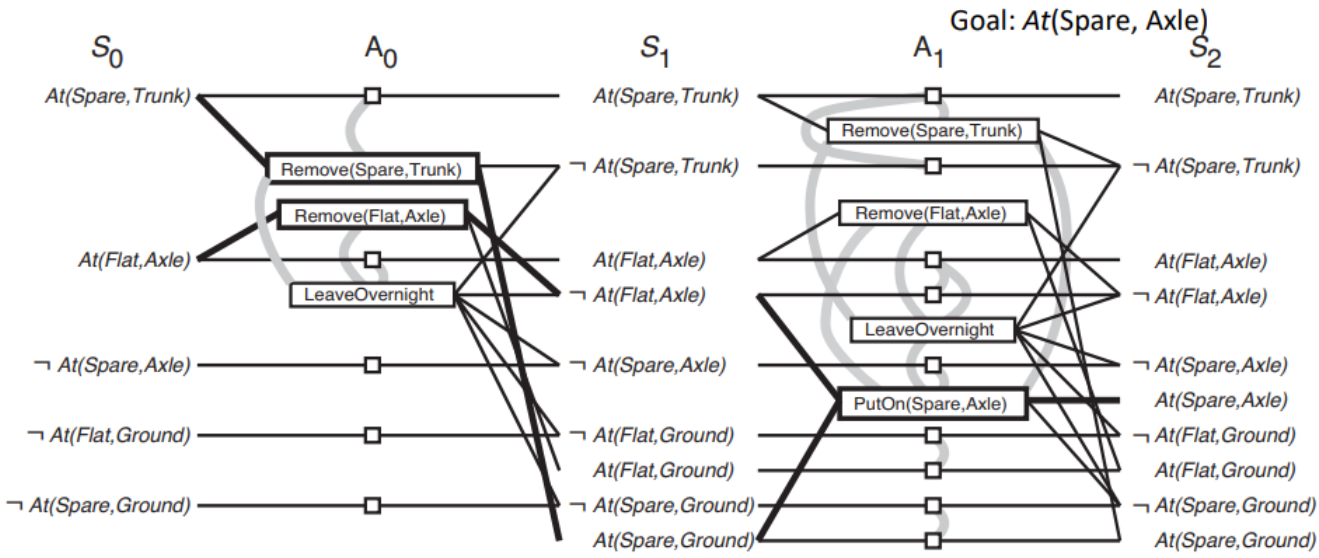
If g appears at level i, the plan possibly exists but we have to check the mutex relations. But, even so, there are plans that mutex cannot recognize as impossible. We need higher order constraints and to search over the planning graph.

GraphPlan Algorithm

The GraphPlan algorithm is a strategy for extracting a plan from the planning graph.
The planning graph is computed incrementally by the function Expand-Graph. Once a level is reached where **all the literals in the goal show up as non-mutex**, an attempt to extract a plan is made with Extract-Solution.
If Extract-Solution fails, the failure is recorded as no-good, another level is expanded and the process repeats until a termination condition is met.

```
function GRAPHPLAN(problem) returns solution or failure
    graph ← INITIAL-PLANNING-GRAPH(problem)
    goals ← CONJUNCTS(problem.GOAL)
    nogoods ← an empty hash table
    for tl = 0 to ∞ do
        if goals all non-mutex in S_t of graph then
            solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
            if solution ≠ failure then return solution
        if graph and nogoods have both leveled off then return failure
        graph ← EXPAND-GRAPH(graph, problem)
```

Example



1. The planning graph is initialized with S_0, representing the initial state. The goal At(Spare, Axle) is not present in S_0
2. Expand-Graph adds into A_0 the three applicable actions and persistence actions for all the literals in S_0. The effects of the actions are added at level S_1. Mutex relations are also added to the graph
3. The goal is not yet present in S_1, so we call Expand-Graph again adding A_1 and S_2
4. All the literals from the goal are present in S_2, and non of them is mutex with any other $\Rightarrow$ we can call Extact-Solution

Extract-Solution Two approaches:

Solve as boolean CSP: the variables are the actions at each level, the values for each variable are in or out of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition

Solve as backward search problem

Start with S_n (the last level of the planning graph) and the goals

For each level S_i select a number of non-conflicting actions in A_{i-1} whose effects cover the goals in S_i. The resulting state is S_{i-1} with goals the preconditions of the selected actions

The process is repeated until S_0, hoping all the goals are satisfied

If Extract-Solution fails, we record the pair (level, goals) as no-good so that we can avoid to repeat the computations.

Complexity and Heuristics Constructing the planning graph takes polynomial time. Solution extractions is **intractable** in the worst case. Heuristics exist.
Greedy algorithm based on the level cost of the literals:

1. Pick first the literal with the highest level cost
2. To achieve that literal, prefer actions with easier preconditions. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is the smallest

Termination We can prove that GraphPlan will terminate and return failure when there is no solutions, but we may need to expand the graph even after it levels off.
Theorem: if the graph and the no-goods have both leveled off, and no solution is found, we can safely terminate with failure. The **proof** is sketched as follows:

Literals and actions increase monotonically and are finite, we need to reach a level where they stabilize

Mutex and no-goods decrease monotonically and cannot become less than zero, so they too must level off

When we reach this stable state, if one of the goals is missing or is mutex with another goal, it will remain so. We may as well stop the computation

0.6.3 From PDDL to Boolean SAT

How to translate a PDDL description into a form that can be processed by SATPlan?

1. **Propositionalize the actions**: create ground instances of the actions (used in step 4)
2. **Define the initial state**: F_0 for every fluent F in the initial state, $\neg F_0$ for every fluent F *not* in the initial state
3. **Propositionalize the goal** by instantiating variables to a disjunction of constraints
   E.g. $On(A, x) \wedge Block(x)$ becomes $On(A, B) \vee On(A, C) \vee On(A, D) \vee \ldots$ for each block mentioned
4. **Add successor-state axioms**. For each fluent:

   $$F_{t+1} \Leftrightarrow \text{ActionCauses}F_t \vee (F_t \wedge \neg \text{ActionCausesNot}F_t)$$

   where ActionCausesF_t is a disjunction of all the ground actions that have F in their add list and ActionCausesNotF_t is a disjunction of all the ground actions that have F in their delete list
5. **Add precondition axioms**: for every ground action $A_t \Rightarrow PRE(A)_t$
6. **Add action exclusion axioms**: every actions is distinct from every other action

This resulting translation is in the form that can be given in input to SATPlan to find a solution.

0.6.4 Partial Order Planning

POP is an interesting approach since it addresses the issue of independent subgoals, that can be performed in parallel. For some specific tasks, such as operations scheduling, is the technology of choice.
It represents a change of paradigm: planning as search in the state of partial plans rather than in space of states. The plan refinement approach is also more explainable: it makes it easier for humans to understand what the planning algorithms are doing and verify that they are correct.

Ideas The driving principle is **least-commitment**. Partially ordered plans:

Do not order steps in the plan unless necessary to do so

In a partial-order plan, steps are partially ordered

Plan linearization: to impose a total order to a partially ordered plan

Partially instantiated plans:

Leave variables uninstantiated until is necessary to instantiate them

A plan without variables is said to be **totally instantiated**

Searching in the Space of Partial Plans Instead of searching in space of states as in the classical formulation, we search in the space of partial plans:

1. We start with an empty plan
2. At each step, we use operators for plan construction and refinement

   We can add actions in order to satisfy some pre-condition, i.e. fixing flaws in the plan

   We can instantiate variables

   We can add ordering constraints between steps
3. We stop when we obtain a consistent and complete plan where

   All the preconditions of all the steps are satisfied

   Ordering constraints do not create cycles

Every linearization is a solution.

Representation of Plans Partial plans are represented as:

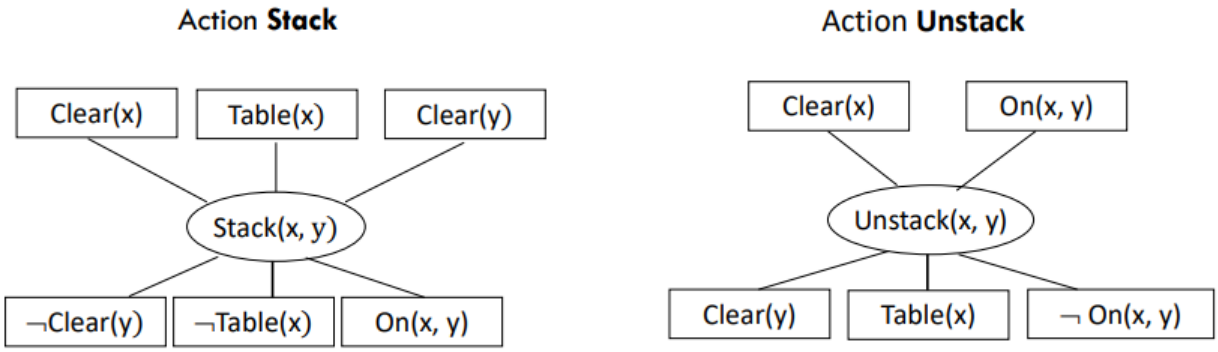
A set of actions, among them Start and Finish

A set of open preconditions

Constraints among actions of two different types

Ordering relations $S_1 < S_2$ (S_1 before S_2)
Causal links $S_1 \rightarrow_{\text{Cond}} S_2$ (S_1 achieves Cond for S_2)
Note: $S_1 \rightarrow_{\text{Cond}} S_2 \Rightarrow S_1 < S_2$ but not viceversa

Representation of Actions



POP Algorithm We start with the empty plan, with Start and Finish. At each step:

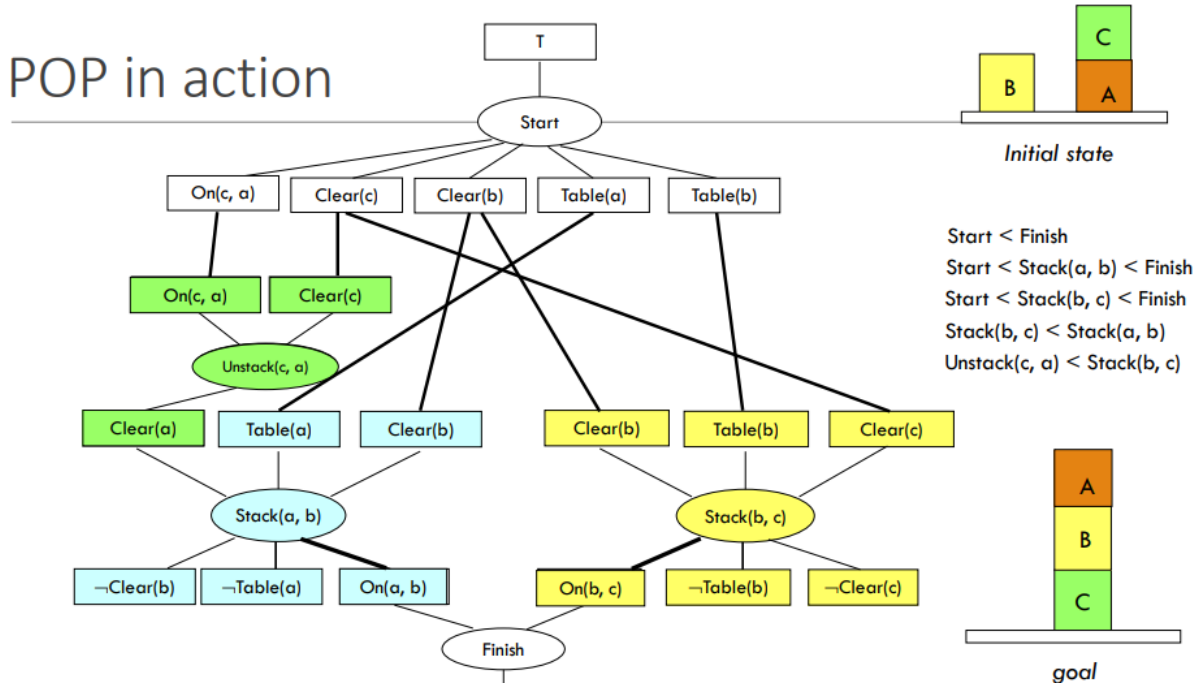
We choose a step B and one of its preconditions p, and we generate a successor plan for each action A (old or new) having p among the effects

After choosing an actions A, consistency is re-established as follows:

Add to the plan the constraints $A < B$ and $A \rightarrow_p B$

Possible actions C having $\neg p$ as effect are potential conflicts (or **threats**). They need to be anticipated or delayed, adding the constraints $C < A$ or $B < C$.
This step may fail

We stop ehn the set of open pre-conditions is empty



Credits: M. Simi

Analysis We obtained a complete and consistent partial plan. Any linearization is a solution, in this case only one possible: [Unstack(C, A), Stack(B, C), Stack(A, B)]
The POP algorithm is correct and complete: any plan computed is a solution and if a plan exists the algorithm finds it.