Artificial Intelligence Fundamentals

Federico Matteoni

A.A. 2021/22

Index

0.1 Introduction

Prof.s: 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.
The challenges are:

Software is not a problem, the community is open and the software can be replicated the software can be replicated

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 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: knowledge must 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 prove, 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. Dominant approach today, fragmented AI.
One strong argument against strong AI is the lack of needs by the systems: biological need, 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 can't have needs, cravings or desires.

AI is the enterprise of building intelligent computational agents

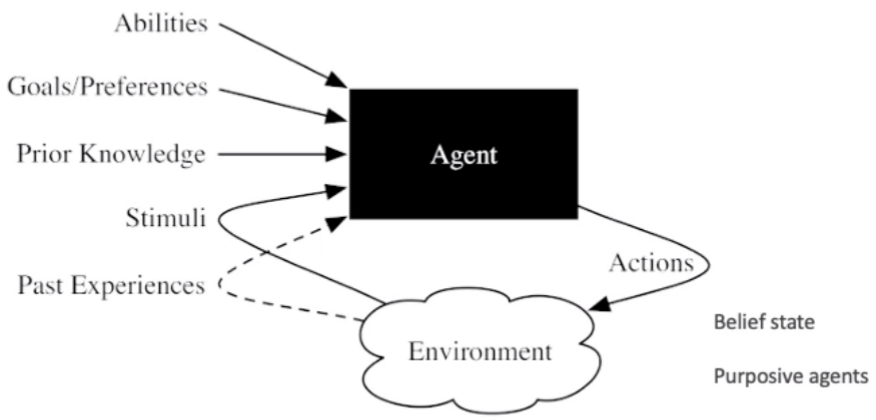
Agents An agent is something that acts in an environment. We are interested in what an agetn does, that is how it acts. We judge and agent by its action. An agent acts intelligently when: what it does is appropriate given the circumnstances and its goals, it is flexible to changing envoronments and changing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

Scientific Goal: undestrand the principles that make intelligent behavior possibile in natural or artificial systems

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

Artificial Intelligence Artificial intelligent is not the opposit of real intelligence. Intelligence cannot be *fake*: in 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 updated test: 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:

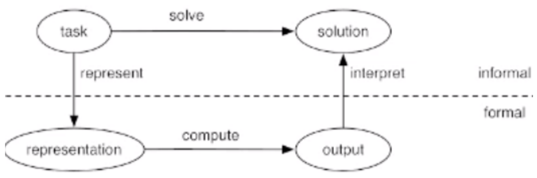
define the task in natural language, what need to be computed

define what is a solution and its quality: optimal, satisfying, aproximately 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 syumbolic rtepresentation of the beliefs of the agents. In 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 possibile (hierchical 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** 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 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.
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 th **Uncertainty**, which can be

in sensing (fully/partially observable states)

about the effects of the actions (deterministic/stochastic)

Preference dimension which considers thetehr the agent has

goal (achievemtn goal a proposition true in a final state, or mainentance goal, proposition true in all psobbile states)

complex preferences, involving trade-offs among the desiderability of various. . .

Number of agents

single agent reasoning

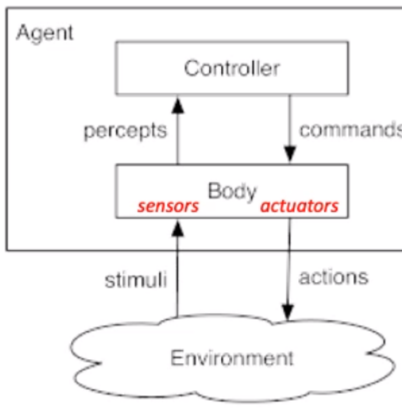
multi agent resoning

Interaction considers wheter the agent does:

offline reasoning or

online reasoning

Agent Architectures 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, which receives percepts from the body and sends command to the body. A body includes sensors that converts stimuli into percepts and actautors that convert commands into actions.



Bot sensors and Agents act in time. T is a set of time points, with start 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 Trasduction 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 $\textit{remember} : S \times P \to S$ with S being the set of belief states and P the set of percepts

Command function $\textit{command} : S \times P \to C$ 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, $\mathit{Actions}(s)$

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

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

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 $\mathit{cost}((n_0, \ldots, n_k)) = \sum_{i=1}^{k} \mathit{cost}((n_{i-1}, n_i))$.
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

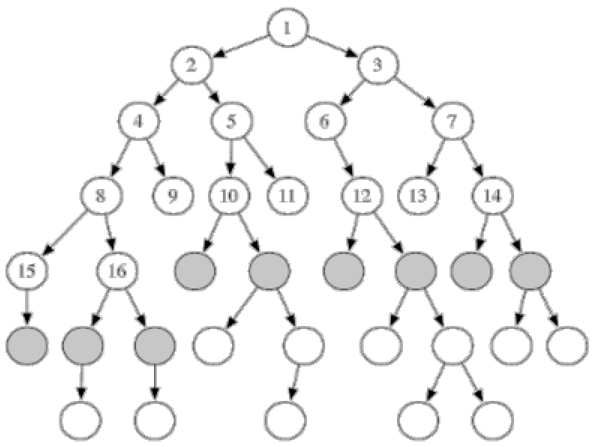
Generic search algorithm

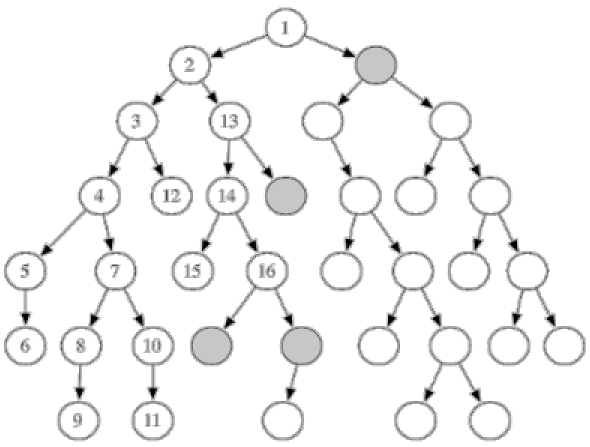
```
input:
a graph
a set of nodes
boolean function goal(n) that tests if n is a goal node

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, optimal, 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 ignoring 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 **underestimates** the minimum cost of a path from a node n to a goal: $\forall\, n \;\; h(n) \leq h^*(n)$

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

The 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 exploits 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 $\langle$ scope, rel$\rangle$: 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 $C_{x_1,\ldots,x_k} =$ 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)

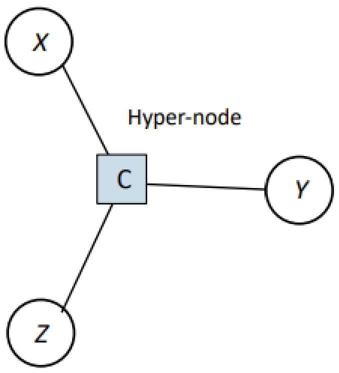
Quality of solutions

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 constraint, tere decomp, 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.

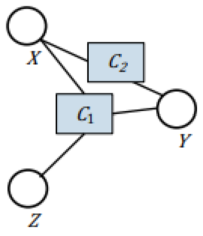
In general, every CSP is associated with a constraint hyper-graph, a generalization of graphs: an hyper-node may connect more than two nodes. The constraint hyper-graph of a CSP $\langle X, D, C\rangle$ is a hyper-graph in which each node represent a variable in C and each hyper-node represents a higher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

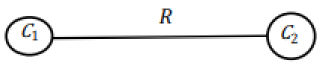


Dual Graph transformation Alternate way to convert a n-ary CSP to a binary one:

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 other 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 $x \to y$, it can be made consistent by reducing D_x to $\{0, 1, 4, 9\}$. Considering $y \to x$, 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 with 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)	$A \in \{1,2,3,\cancel{4}\}$
$\{(A,C),(C,A)\}$	(B,A)	$B \in \{\cancel{1},2,3,4\}$
$\{(C,A)\}$	(A,C)	$A \in \{\cancel{1},2,3\}$
$\{(B,A),(C,A)\}$		
$\{(C,A)\}$	(B,A)	$B \in \{\cancel{2},3,4\}$
$\{\}$	(C,A)	$C \in \{1,2,\cancel{3},\cancel{4}\}$

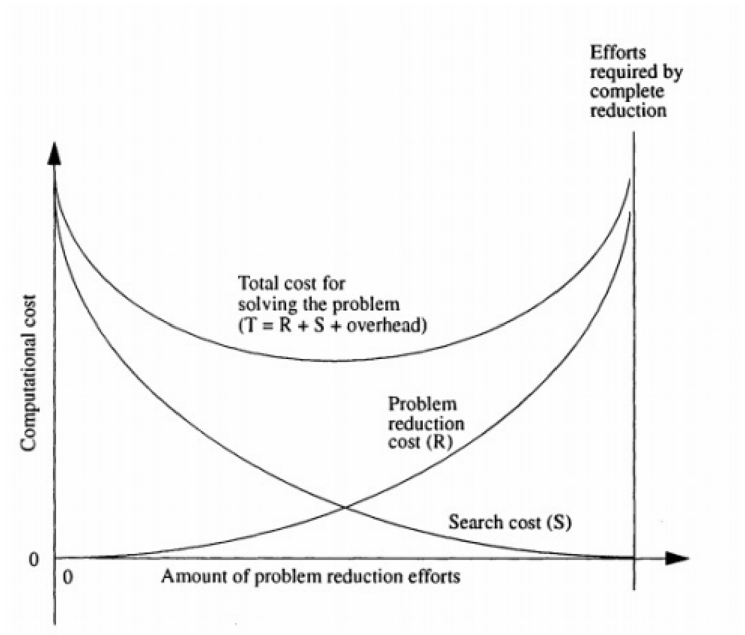
At the end $A \in \{2,3\}, B \in \{3,4\}, C \in \{1,2\}$

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 xc 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 ordiering, 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

Heuristics and search strategies

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

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

`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 (minimnum remaining values): variable with fewest "legal" remaining values

Degree heuristic: variable involved in the largest number of constraints

Choosing value

Least constaint 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:

```
function 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 assignement) 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 successors
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

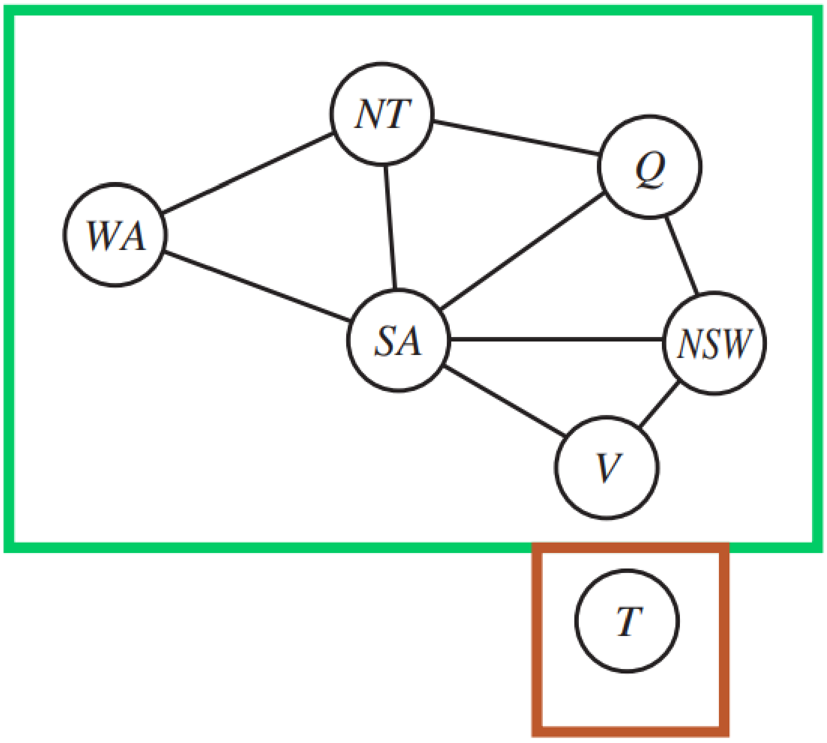
Stochastic Local Beam Search: selects k of the individuals at random with a probability that depends on the evaluation function

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.

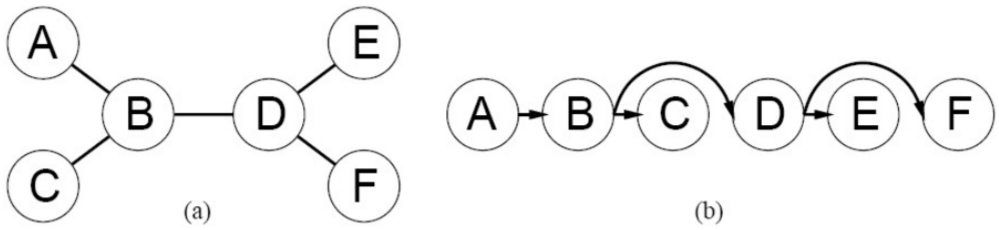
Independent sub-problems When problems have a specific structure, which is reflected in properties of the constraint graph, there are strategies for improving the process of finding a solution. A first obvious case is that of independent subproblems, for examples in the map coloring problem, a country 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 $X_1, \dots, X_n \Leftrightarrow$ 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 $O(nd^2)$

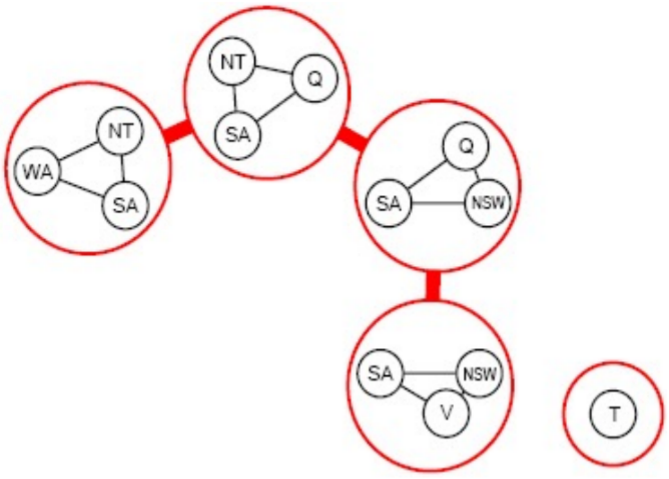
Tree-CSP solver

1. Proceeding from X_n to X_2, make the arcs $X_i \rightarrow X_j$ 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: 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 int at least one of the subproblem, 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 problem 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

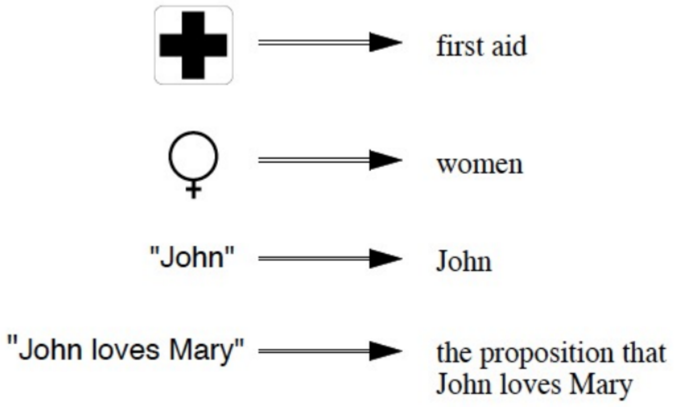
0.3.1 Knowledge Representation and Reasoning

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** (KB) systems/**agents**. A KB agent maintains a knowledgebase of facts expressed in a declarative language, and is able to perform automated reasoning to solve complex tasks.
The knowledgebase (KB) is the agent's representation of the world which is responsible for its intelligente 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 symboli structures to represent a collection o 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 know 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 $\models \alpha$).
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

- The try solving **open-ended tasks**, not per-compiled kind of behavior for specific tasks
- **Separation** of knowledge and "inference engine"

Extensibility: we can extend the existing behavior by simply adding new proposition. **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 (KB $\models \alpha$)

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

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

Unification algorithm

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:

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

$f(s_1, \dots, s_n) = g(t_i, \dots, t_m) \rightarrow$ 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.

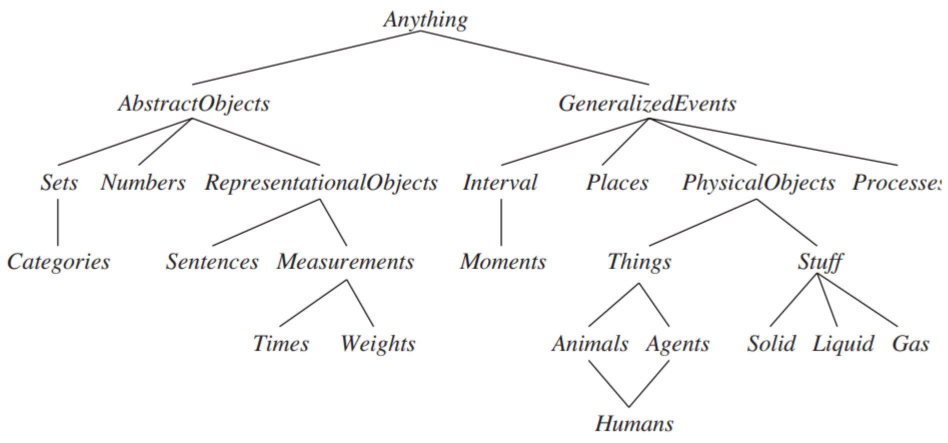
Knowledge Engineering & Ontological Engineering

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 Purchase(john, sears, bike, \$200), we can introduce individuals for purchase objects and functions for roles (**reification**): Purchase(p23)∧agent(p23)=john∧object(p23)=bike∧source(p23)=sears∧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 Much 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 trainsitive, PartOf(x, y)∧ 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:

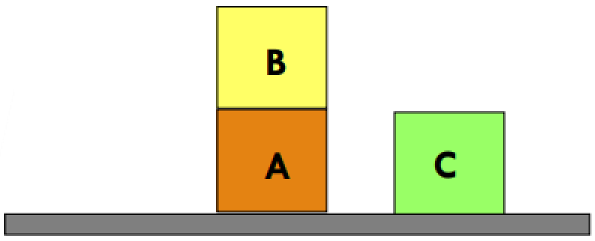
Situations: snapshots of the world at a given instant in time, the result of an action

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

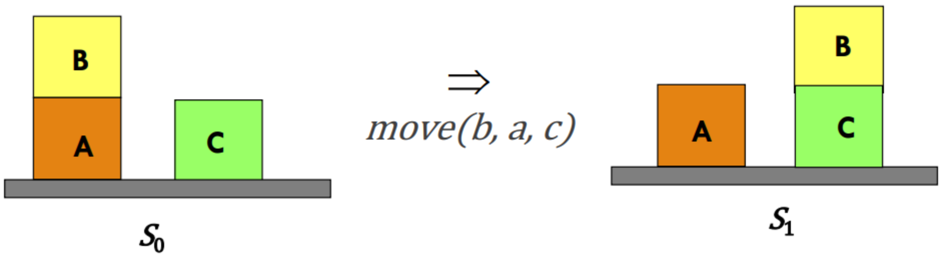
Can be formalized as:

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 On(a, b) becomes On(a, b, s)

Actions: modeled as functions
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 Unstack(a, b) and 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 Move(b, a, c) executed in s_0. Then we can assert, for example, On$(b, c,$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 $\text{Result}([a_1, a_2, \dots, a_n], s_0) = \text{Result}(a_n, \text{Result}(a_{n-1}, \dots \text{Result}(a_2, \text{Result}(a_1, s_0)) \dots))$

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:

$$\begin{array}{lll} f \text{ true after} \Leftrightarrow & [\textit{preconditions before} \text{ and} & \textit{preconditions} \\ & \textit{an action made } f \textit{ true}] \text{ or} & \textit{effect} \\ & [f \textit{ was true before and no action made it false}] & \textit{frame axioms} \end{array}$$

An example with Clear:

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

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
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 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.
KB$\models$ Poss(a_i, Result($[a_1, \ldots, a_{i-1}], s_0$)) for each $i \mid 1 \leq i \leq n$
For example Result(Pickup(b_2), Result(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(At(Shankar, Berkley), t) Where At(Shankar, Berkley) is a term, t a time

T(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$\models$ a then KB $\cup\{b\} \models a$ (KB$\wedge b \models a$)
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 ¬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): $\nexists\, a \mid$ KB$\models a \wedge$ 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$\cup\{\neq p, \neq q\}$ since KB $\neg \models p$ and KB $\neg \models q$, but KB$\cup\{\neg p, \neg q\} \models \neg(p \vee q)$ then CWA(KN) 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 $\models \{p \vee q_1, \dots, \vee q_n\}$ 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 KN 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 $\models \{p \vee q_1 \ldots \vee q_n\}$ 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$(x) \wedge \neg\text{Ab}_f(x) \Rightarrow$ Flies(x)

We also have Bird(Tweety), Bird(Chilly), Chilly $\neq$ Tweety, $\neg$Flies(Chilly)
We want to infer Flies(Tweety), but Tweetu 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 and extension of (F, D) $\Leftrightarrow$ for every sentence π, E satisfies the following:

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

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=$\{\frac{:\neg A}{\neg A}, \frac{:\neg B}{\neg B}\}$ then E_1={A∨B, ¬A}, E_2={A∨B, ¬B} are mutually inconsistent

There are default theories with no extensions.
Example: with D=$\{\frac{:\neg A}{\neg A}\}$, 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.