UNIVERSITÀ DEGLI STUDI DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
Master of Science in Artificial Intelligence Systems

Fundamentals of Artificial Intelligence

Academic Year 2021/2022
Trento, Italy

Ambrosi Giovanni, Bonomi Andrea, De Min Thomas, Laiti Francesco,
Lobba Davide, Miotto Sara, Momesso Filippo, Turri Evelyn

Preface

The goal of this work is to help studying the Fundamentals of AI course by prof. Roberto Sebastiani. A lot of people worked on this latex project and there could be errors around. We encourage you to use it and help us improving it for future students. You can find the repository of the project here: `https://github.com/unitn-drive/fundamentals-of-ai`. Please feel free to make pull requests for making it better.

Contents

Chapter 1

Artificial Intelligence

First solve AI, then use AI to solve everything else.

CEO of Google DeepMind
Demis Hassabis

1.1 Introduction to AI

Francesco Laiti

What is AI?

Definition. We define **Intelligence** the capacity for logic, understanding, self-awareness, learning, emotional knowledge, reasoning, planning, creativity, critical thinking, and problem-solving.
More generally, it can be described as the ability to perceive or infer information, and to retain it as knowledge to be applied towards adaptive behaviors within an environment or context.
Some have defined **intelligence** in terms of fidelity to human performance, while others prefer an abstract, formal definition of intelligence called rationality (doing the right thing).

Definition. Artificial Intelligence is the science of making machines do things that would require intelligence if done by men. *(thank you Prof. Casonato).*

Some consider intelligence to be a property of internal *thought processes* and *reasoning*, while others focus on intelligent *behavior*, an external characterization.

Two orthogonal dimensions:

- human *vs* rational;
- thought *vs* behavior.

create four possible combinations.

Methods used are necessarily different:

- **human-like intelligence approach**: involves observations and hypotheses about human behavior and thought process;
- **rational approach**: combination of mathematics and engineering, and connects to statistics, control theory, and economics.

The four possible combinations (they have both disparaged and helped each other):

1. **Thinking humanly**

   Problem. *How do humans think?*

   - Idea: develop a theory of the mind → express the theory as computer programs
   - Requires scientific theories of brain activities (cognitive model)
   - Inter-disciplinary field: Cognitive Science
     - combines computer models from AI and experimental techniques from psychology
     - construct precise and testable theories of the human mind
   - AI and Cognitive Science nowadays distinct
     - A.I: find an algorithm performing well on a task
     - C.S: find a good model of human performance

     although they fertilize each other (e.g. in computer vision)

2. **Acting humanly**

   Problem. *When does a system behave intelligently?*

   Turing test was designed as a thought experiment in order to answer to this question: "Can a machine think?"

   $$\text{behave intelligently} \iff \text{behave humanly}$$

   Turing test: a computer A passes the test if a human interrogator C, after posing some written questions, cannot tell whether the written responses come from a person B or from a computer A, as shown in Figure 1.1.

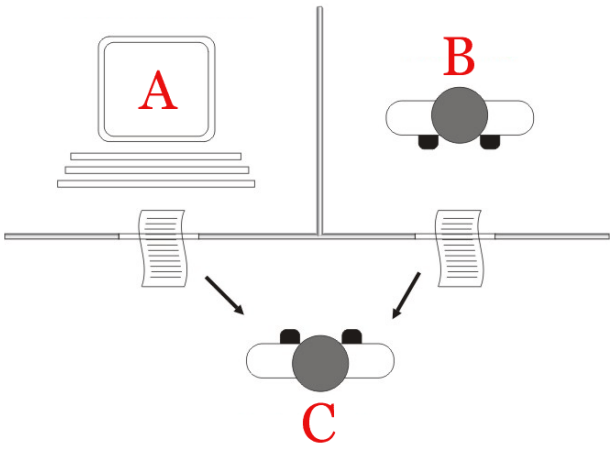


Figure 1.1: Scheme of the Turing test.

Capabilities for passing the Turing Test:

- **natural language processing** to enable it to communicate successfully in English (or other);
- **knowledge representation** to store what it knows or hears;
- **automated reasoning** to answer questions and to draw new conclusions;
- **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

For a **total** Turing Test, which requires interaction with objects and people in the real world:

- **computer vision** to perceive objects;
- **computer speech** to communicate orally;

- **robotics** to manipulate objects and move around.

Example. Some success with Turing test:

(a) (2014) a chatbot by Eugene Goostman, mimicking the answer of a 13 years old boy, has succeeded the test (chatbots are now frequently available);

(b) vocal assistants are now of common use
e.g. Alexa (Amazon), Siri (Apple), Cortana (Microsoft),...

There are also **limitations** for the Turing test:

- not reproducible, constructive or amenable to mathematical analysis;
- AI researchers devoted little effort to make systems pass the Turing Test;
- Should we really emulate humans to achieve intelligence?
Metaphorical example: Successful flight machines have not been developed by imitating birds, rather by studying engines and aerodynamics.
Kill idea: is it important to reach the intelligence of human?
- Shouldn't we study the underlying principles of intelligence instead?

3. **Thinking rationally**

Problem. *Can we capture the laws of thought?*

- Aristotle: What are correct argument and thought processes?
codify "right thinking" i.e. irrefutable reasoning processes (syllogisms):
(e.g. "all men are mortal; Socrates is a man; therefore, Socrates is mortal")
→ Logic and Logical inference
- The Logicist tradition in AI hopes to create intelligent systems using logic-based inference systems
  (a) "algorithm = logic + control";
  (b) logic programming, automated-deduction systems, ...
  (c) logics: propositional, first-order, modal and description, temporal,...
- Two main limitations:
  (a) not easy to state informal knowledge into the formal terms of logic;
  (b) problems undecidable or computationally very hard (NP-hard).
- Logical reasoning is currently part of many fields of AI
  (a) problem solving, knowledge representation and reasoning, planning;
  (b) does not exhaustively cover AI.

4. **Acting rationally**

Problem. *Can we make systems "do the right things"?*

- An agent is an entity that perceives and acts;
- A rational agent acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome;
- Rational agents need all the skills needed for the Turing Test;
- Thinking rationally is sometimes part of being a rational agent e.g. planning an action sometimes action without thinking (e.g. reflexes);
- Two advantages over previous approaches:
  (a) More general than law of thoughts approach (correct inference is just one of several possible mechanisms for achieving rationality);
  (b) More amenable to scientific development than human-emulation approaches (rationality mathematically well defined and general).

AI System Classification

1. **Weak *vs* Strong AI**
   (a) **Weak AI**: build systems that act as if they were intelligent?;
   (b) **Strong AI**: the machine would require an intelligence equal to humans. It would have a self-aware consciousness that has the ability to solve problems, learn, and plan for the future.
2. **General *vs* Narrow AI**
   (a) **General AI**: refers to systems able to cope with any generalized task which is asked of it, much like a human;
   (b) **Narrow AI**: refers to systems able to handle one particular task. AI system displays a certain degree of intelligence only in a particular narrow field to perform highly specialized tasks.
3. **Symbolic Approach *vs* Connectionist Approach**
   (a) **Top-down *or* Symbolic Approach**
      i. Symbolic representation of knowledge;
      ii. Logics, ontologies, rule based systems, declarative architecture;
      iii. Human-understandable models.
   (b) **Bottom up *or* Connectionist Approach**
      i. Based on Neural networks;
      ii. Knowledge is not symbolic and it is "encoded" into connections between neurons;
      iii. Concepts are learned by examples;
      iv. Non understandable by humans.

The State of Art AI is everywhere. It is used in search engines, route planning, logistics, medical diagnosis, automated help desk, smart devices, assistants, smart home and more.

What can AI do today?
Perhaps not as much as some of the more optimistic media articles might lead one to believe, but still a great deal.

Here are some examples of what AI can currently do:

- classify incoming e-mails as spam or not;
- predict stock price evolution;
- understanding handwriting;
- learn to gab objects;
- design a molecule with given properties;
- translate text from Chinese to English;
- convert a voice into text;
- predict traffic trajectories;
- automatically writing the caption of a figure/image;
- driving autonomously;
- win against any human at chess.

These are just a few examples of artificial intelligence systems that exist today. Not magic or science fiction—but rather science, engineering, and mathematics.

When (if ever) will AI systems achieve human-level performance across a broad variety of tasks?
Ford (2018) interviews AI experts and finds a wide range of target years, from 2029 to 2200, with a mean of 2099. In a similar survey (Grace et al., 2017) 50% of respondents thought this could happen by 2066, although 10% thought it could happen as early as 2025, and a few said "never".

How will future AI systems operate?
We can't yet say. The field has adopted several stories about itself—first the bold idea that intelligence by a machine was even possible, then that it could be achieved by encoding expert knowledge into logic, then that probabilistic models of the world would be the main tool, and most recently that machine learning would induce models that might not be based on any well-understood theory at all. The future will reveal what model comes next.

1.2 Intelligent Agents

Francesco Laiti

Agents and Environments

Definition. We define **agent** any entity that can be viewed as:

1. **Perceiving** its environment through sensors;
2. **Acting** upon that environment through actuators.

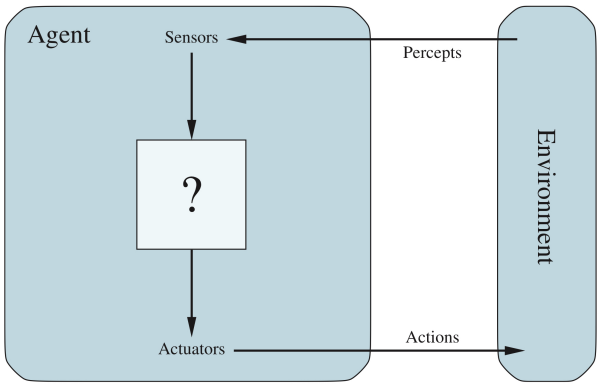


Figure 1.2: Agents interact with environments through sensors and actuators.

Definition. We define **environment** that part of the universe whose state we care about when designing an agent—the part that affects what the agent perceives and that is affected by agent's actions.

Definition. We define **percept** the collection of agent's perceptual inputs at any given instant.

Definition. We define **percept sequence** the complete history of everything the agent has ever perceived.

Definition. An **agent's choice** of action at any given instant:

1. can depend on the entire percept sequence observed to dates;
2. does not depend on anything it hasn't perceived;
3. can perceive its own actions, but not always it effects.

Definition. An agent's behavior is described by the **agent function** that maps any given percept sequence to an action. It is an abstract mathematical description, possibly-infinite description.

Definition. Internally, the agent function for an artificial agent is implemented by an **agent program**. The agent program is a concrete implementation of the agent function with a finite description and it runs on the physical architecture to produce the agent function *f*.

Example. A very-simple vacuum cleaner

1. **Environment**: squares A and B;
2. **Percepts**: location ({A, B}) and content ({Dirty, Clean});
3. **Actions**: *{left, right, suck, no-op}*.

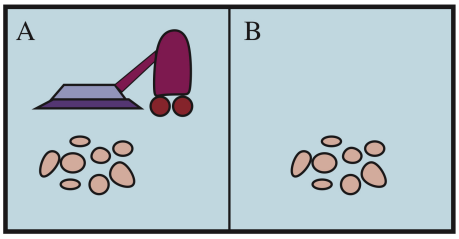


Figure 1.3: A vacuum-cleaner world with just two locations.

Rational Agents

Definition. We define **rational agent** an agent that does the right thing.

Definition. We define **right thing** the most *successful* thing.

To understand what is *successful*: Percept sequence receive → generate sequence of actions → environment go through sequence of states. If such sequence is desirable → agent has performed well.

Definition. A **performance measure** (p.m.) allows to evaluate any sequence of environment states (p.m. should be objective).

General rule: it is better to design the p.m. according to what one actually wants to be achieved in the environment, rather than according to how one thinks the agent should behave.

What is rational at any given time depends on four things:

1. **Performance measure** that defines the criterion of success;
2. **Prior knowledge** of the agent of the environment;
3. **Actions** that the agent can perform;
4. **Percept sequence** of the agent to date (from sensors).

For each possible percept sequence, a **rational agent** should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Example. Simple vacuum cleaner agent

1. **Performance measure**: one point for each clean square at each time step, over 1000 time steps;
2. **Prior knowledge**:
   - "geography" known a priori;
   - dirt distribution and agent initial location unknown;
   - clean squares cannot become dirty again.
3. **Actions**: Left, Right, Suck;
4. **Perception**: self location, presence of dirt.

Under this assumption, the simple vacuum cleaner agent is a **rational agent**.

⚠ The agent would behave poorly under different circumstances.
For instance, once all the dirt is cleaned up:
✗ give a penalty for each move.
✓ do nothing once it is sure all the squares are clean.

Definition. An **omniscient agent** knows the actual outcome of its actions and can act accordingly; omniscience is impossible in reality.

Definition. A **rational agent** may only know "up to a reasonable confidence".
Rationality is not the same as perfection. Rationality maximizes expected performance, while perfection maximizes actual performance.

$$\text{Rational} \neq \text{Omniscience}$$
$$\text{Rational} \neq \text{Perfection}$$

Rationality requires other important features:

1. **Information gathering/exploration**: the rational choice depends only on the percept sequence to date → actions needed in order to modify future percepts;
   Ex. look both ways before crossing a busy road
2. **Learning**: agent's prior knowledge of the environment incomplete → learning from percept sequences improves and augments it;
   Ex. a baby learns from trial & errors the right movements to walk
3. **Being Autonomous**: prior knowledge may be partial/incorrect or evolving → learn to compensate for partial or incorrect prior knowledge.
   Ex. a child learns how to climb a tree

Task Environments

Definition. A **task environment** is essentially the "problem" to which rational agent is the "solution".

Task environment is described by four elements, acronymically minded **PEAS**, listed below with a taxi's environment example:

1. **P**erformance measure: safety, destination, profits, comfort,...
2. **E**nviroment: streets/freeways, other traffic, pedestrians, ...
3. **A**cutators: steering, accelerator, brake, horn, speaker/display, ...
4. **S**ensors: video, sonar, speedometer, engine sensors, GPS, ...

⚠ Some goals to be measured may conflict → **tradeoff** required

Virtual environment can be just as complex as the "real" world. For instance, a software agent that trades on auction and reselling Web sites deals with millions of other users and billions of objects, many with real images.

Task-Environments Type

1. **Fully observable *vs* partially observable**:
   (a) **Fully observable**: agent's sensors give it access to the complete state of the environment at each point in time;
   (b) **Partially observable**: parts of the state are not accessible for sensors or sensors are noisy and inaccurate;
   (c) **Unobservable**: agent has no sensor.
2. **Single-agent *vs* multi-agent**:
   (a) **Single-agent**: an agent solving a crossword puzzle by itself;
   (b) **Multi-agent**: contains other agents who are also maximizing some performance measure that depends on the current agent's actions
      i. **Competitive**: other agents' goals conflict with, or even oppose to, the agent's goals.
      Ex. chess, war scenarios, taxi driving (compete for parking lot);
      Design problem: randomized behaviour often rational (unpredictable);
      ii. **Cooperative**: other agents' goals coincide in full, or in part, with the agent's goals.
      Ex. ants' nest, factory, taxi driving (avoid collisions).
      Design problem: communication with other agents often rational.

3. **Deterministic *vs* stochastic**:
   (a) **Deterministic**: the next state of the environment is completely determined by the current state and the action is executed by the agent(s);
   (b) **Stochastic**: uncertainty about outcomes is quantified in terms of probabilities.
   Ex. there's a 25% chance of rain tomorrow;
   (c) **Non-deterministic**: actions are characterized by their possible outcomes, but no probabilities are attached to them.
   Ex. there's a chance of rain tomorrow.

   ⚠ In a multi-agent environment we ignore uncertainty that arises from the actions of other agents. Ex: chess is deterministic even though each agent is unable to predict the actions of the others.
   ⚠ Most real situations are so complex that is impossible to keep track of all the unobserved aspects; for practical purposes, they must be treated as stochastic (Ex. taxi driving).

4. **Episodic *vs* sequential**:
   (a) **Episodic**: the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action → episodes do not depend on the actions taken in previous episodes, and they do not influence future episodes;
   Ex. an agent that has to spot defective parts on an assembly line;
   (b) **Sequential**: the current decision could affect future decisions → actions can have long-term consequences.
   Ex: chess, taxi driving.

   Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

5. **Static *vs* dynamic**
   (a) **Static**: agent need not keep looking at the world while it is deciding on an action.
   Ex. crossword puzzles;
   (b) **Dynamic**: the environment can change while an agent is choosing an action.
   Ex. taxi driving;
   (c) **Semi-dynamic**: the environment itself does not change with time, but the agent's performance score does.
   Ex: chess with a clock

6. **Discrete *vs* continuous**
   The discrete/continuous distinction applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agent.
   (a) **Discrete**: discrete state, time, percepts and actions.
   Ex. crossword puzzles;
   (b) **Continuous**: continuous state, time, percepts and actions.
   Ex. taxi driving.

7. **Known *vs* unknown**
   Describes the agent's (or designer's) state of knowledge about the "laws of physics" of the environment.
   (a) **Known**: the outcomes (or outcome probabilities if stochastic) for all actions are given. A known environment can be partially observable;
   (b) **Unknown**: the agent will have to learn how it works in order to make good decisions. An unknown environment can be fully observable.

$$\text{Known} \neq \text{Fully observable}$$

The **simplest environment** is fully observable, single-agent, deterministic, episodic, static and discrete (ex. simple vacuum cleaner).
Most **real-world situations** are partially observable, multi-agent, stochastic, sequential, dynamic, and continuous (ex. taxi driving).

Agent Type The job of AI is to design an **agent program** that implements the agent function → mapping from percepts to actions.

Definition. The **agent architecture** is some sort of computing device with physical sensors and actuators where this program will run on.

$$\text{Agent} = \text{Architecture} + \text{Program}$$

All agents have the same skeleton:

1. **Input**: current percepts;
2. **Output**: action;
3. **Program**: manipulates input to produce output.

Remark

1. the **agent function** takes the entire percept history as input;
2. the **agent program** takes only the current percept as input

⚠ If the actions need to depend on the entire percept sequence → the agent will have to remember the percepts.

The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from a vast table (blow-up table size problem)

Four basic kinds of agent programs (can be turned into learning agents, page 18):

1. **Simple-reflex agents**:
   - select action on the basis of the current percept, ignoring the rest of the percept history;
   - implemented through condition-action rules;
   - large reduction in possible percept/action situations due to ignoring the percept history;
   - very simple and may work only if the environment is fully observable (errors, deadlocks or infinite loops may occur otherwise → limited applicability).

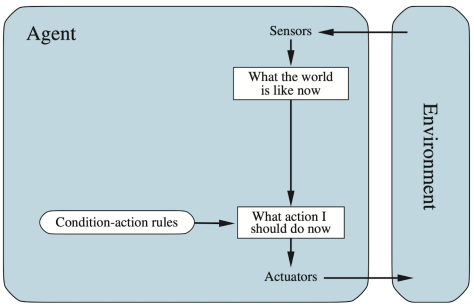


Figure 1.4: Schematic diagram of a simple reflex agent.

Algorithm 1 Simple-reflex agent program

function *Simple-Reflex-Agent*(*percept*) **return** an action
  persistent: *rules*, a set of condition-action rules
  state ← interpret-input(*percept*)
  rule ← rule-match(*state*, *rules*)
  action ← *rule*.action
  return *action*

2. **Model-based reflex agents**:

   - maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state;
   - to update internal state the agent needs a model of the world:
     (a) how the world evolves independently of the agent.
     Ex: an overtaking car will soon be closer behind than it was before;
     (b) how the agent's own actions affect the world.
     Ex: turn the steering wheel clockwise → the car turns to the right.

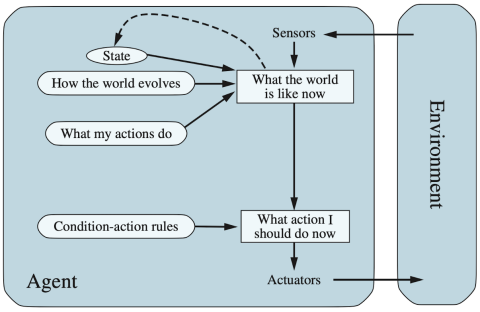


Figure 1.5: Schematic diagram of a model-based reflex agent.

Algorithm 2 Model-based agent program

function *Model-Based-Reflex-Agent*(*percept*) **return** an action
  persistent: *state*, the agent's current conception of the world state
    rules, a set of condition-action rules
    transition_model, a description of how the next state depends on the current state and action
    sensor_model, a description of how the current world state is reflected in the agent's percepts
    action, the most recent action, initially none
  state ← update-state(*state*, *action*, *percept*, *transition_model*, *sensor_model*)
  rule ← rule-match(*state*, *rules*)
  action ← *rule*.action
  return *action*

3. **Goal-based agents**:

   - agent needs goal information describing desirable situation.
     Ex: destination for a taxi driver;
   - agent program can combine the goal information with the model to choose actions that achieve the goal →search and planning research;
   - future is taken into account → rules are simple condition-action pairs, do not target a goal
   - more flexible
     (a) the knowledge that supports its decisions is represented explicitly;
     (b) such knowledge can be modified → all of the relevant behaviors to be altered to suit the new conditions.
     (Ex: If it rains, the agent can update its knowledge of how effectively its brakes operate).
     (c) the goal can be modified/updated → modify its behaviour (no need to rewrite all rules from scratch)
   - more complicate to implement
   - may require expensive computation for search and planning

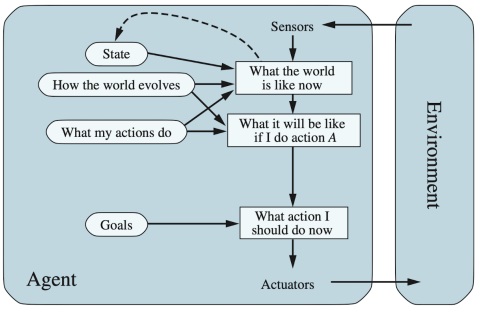


Figure 1.6: Schematic diagram of a goal-based agent.

4. **Utility-based agents**:
   - goals alone often not enough to generate high-quality behaviors.
     Crude binary distinction between *happy*:) and *unhappy*:(states;
   - certain goals can be reached in different ways, of different quality.
     Ex: some routes are quicker, safer, or cheaper than others;
   - idea: Add utility function(s) to drive the choice of actions (*happy* → utility)
     (a) maps a (sequence of) state(s) onto a real number → actions are chosen which maximize the utility function;
     (b) under uncertainty, maximize the expected utility function.
   - advantages with regards to goal-based:
     (a) with conflicting goals, utility specifies an appropriate tradeoff;
     (b) with several goals none of which can be achieved with certainty, utility selects proper tradeoff between importance of goals and likelihood of success.
   - still complicate to implement;
   - require sophisticated perception, reasoning, and learning;
   - may require expensive computation.

→ utility function = internalization of performance measure

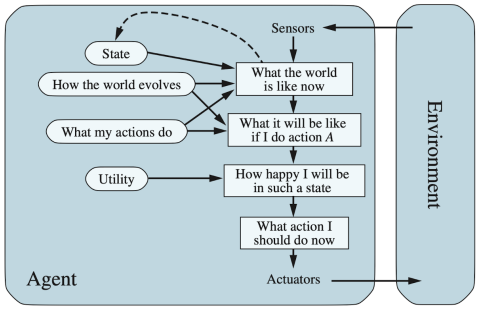


Figure 1.7: Schematic diagram of a utility-based agent.

Learning agents Turing method is now, in many areas of AI, the preferred method for creating state-of-art systems in many areas of AI.
The method → build learning machines and then teach them, rather than instruct them.

Advantages:

1. robustness of the agent program toward initially-unknown environments
2. become more competent than its initial knowledge alone might allow

Learning agent is divided into four conceptual components:

1. **Learning element**:
   (a) makes improvements based on feedback from the **critic** on how the agent is doing;
   (b) determines how the performance element should be modified to do better in the future;
   (c) makes changes to any of the "knowledge" components shown in the agent diagrams (Figure 1.4, 1.5, 1.6, 1.7).
2. **Performance element**: selects external actions (takes in percepts and decides on actions aka "agent programs");
3. **Critic**: tells the learning element how well the agent is doing with respect to a fixed performance standard;
4. **Problem generator**: suggests actions that will lead to new and informative experiences (forces exploration of new stimulating scenarios).

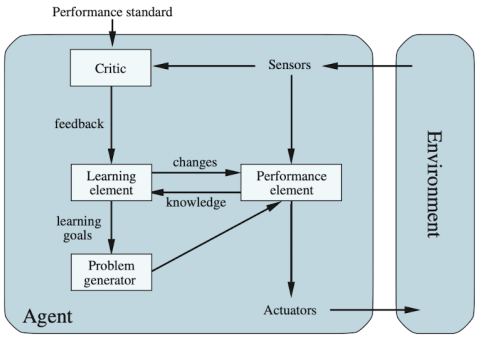


Figure 1.8: A general learning agent.

In summary, agents have a variety of components, and those components can be represented in many ways within the agent program, so there appears to be great variety among learning methods.

Example. Taxi driving

1. After the taxi makes a quick left turn across three lanes, the **critic** observes the shocking language used by other drivers;
2. From this experience, the **learning element** formulates a rule saying this was a bad action;
3. The **performance element** is modified by adding the new rule;
4. The **problem generator** might identify certain areas of behavior in need of improvement, and suggest trying out the brakes on different road surfaces under different conditions.

Environment States There are three ways to represent states and transitions between them ($\Downarrow$ increasing expressive power and computational complexity):

1. **Atomic**:
   - each state of the world is indivisible (black box) $\rightarrow$ no internal structure;
   - state: one among a collection of discrete state values.
     ⚠ Only discernible property is that of being identical to or different from another black box. Ex. find driving routes: {Trento, Rovereto, Verona, ...};
   - very high level of abstraction $\rightarrow$ lots of details ignored;
   - algorithms underlying
     (a) search and game-playing;
     (b) hidden Markov models;

(c) Markov decision processes.

all work with atomic representations.

2. **Factored**:

- a state consists of a vector of attribute values—fixed set of variables or attribute, each of which can have a value;
  Ex. $\langle zone, \{dirty, clean\}\rangle, \langle town, speed\rangle$

- state: combination of attribute values;
  Ex. $\langle A, dirty\rangle, \langle Trento, 40kmh\rangle$
- distinct states may share the values of some attribute
  ⚠ identical iff all attribute have the same values → must differ for at least one value to be different;
  Ex. $\langle Trento, 40kmh\rangle$ and $\langle Trento, 47kmh\rangle$
- can represent uncertainty
  Ex. ignorance about the amount of gas in the tank represented by leaving that attribute blank;
- lower level abstraction → less details ignored;
- many areas of AI are based on factored representations:
  (a) constraint satisfaction and propositional logic;
  (b) planning;
  (c) Bayesian networks;
  (d) (most of) machine learning.

3. **Structured**:

- a state includes objects, each of which may have attributes of its own as well as relationships to other objects
  Ex. $\forall x.(Men(x) \rightarrow Mortal(x))$,
  $Woman(Maria), Mother \equiv Woman \cap \exists hasChild.Person$;
- lowest level of abstraction → can represent reality in details;
- many areas of AI are based on structured representations:
  (a) relational databases;
  (b) first-order logic;
  (c) first-order probability models;
  (d) knowledge-based learning;
  (e) natural language understanding.

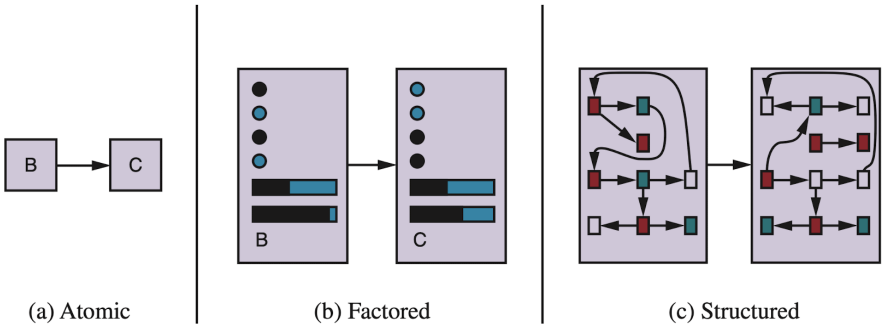


Figure 1.9: Three ways to represent states and the transitions between them.

Chapter 2

Problem Solving

Definition. We define **problem-solving agent** an agent that consider a sequence of actions that form a path to a goal state. The process it undertakes is called **search**.

Problem-Solving Agents We make a distinction between **informed** algorithms, in which the agent can estimate how far it is from the goal, and **uniformed** algorithms, where no such estimation is available. If you think about a routing problem (you are in place A and want to go to place B), the environment usually is unknown. However, for this section, you consider this example with the environment fully observable. Thus, the agent will follow a four-phase problem solving process:

1. **Goal formulation**: The agent adopts the goal of reaching a place. Here the goal limits the actions to be considered (otherwise it is nonetheless than random searching);
2. **Problem formulation**: The agent devises a description of the states and actions necessary to reach the goal. For example by considering the adjacent city at each iteration;
3. **Search**: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. This sequence (defined as a sequence of actions, or a **path**, from the initial state to the goal state both defined below) is the **solution**;
4. **Execution**: The agent execute the actions in the solution.

Algorithm 3 Problem Solving Agent Algorithm

function *Simple-Problem-Solving-Agent*(percept) **return** an action
  seq is an action sequence, initially empty
  state defines a description of the current world state
  goal is a goal, initially set to null
  problem is a problem formulation

  $state \leftarrow Update\text{-}State(state, percept)$
  if *seq* is empty **then**
    $goal \leftarrow Formulate\text{-}Goal(state)$
    $problem \leftarrow Formulate\text{-}Problem(state, goal)$
    $seq \leftarrow Search(problem)$
    if $seq = failure$ **then return** a null action
  $action \leftarrow First(seq)$
  $seq \leftarrow Rest(seq)$ **return** *action*

Definition. If we give the model full reliability (because it is correct) or the environment is fully observable, we ignore the percepts. The agent is called to be in an **open-loop** system.

Definition. If the model is incorrect or the environment non deterministic the agent will use a **closed-loop** approach that monitors also the percepts.

2.1 Search

Andrea Bonomi

Definition. A **search problem** is defined as a tuple of:

- The **state space**: a set of possible **states** (a representation of a physical configuration) in which the environment[1] can be in;
- The **initial state**: the state where the agent starts in;
- A set of one (or more) **goal states**: the state (states) to reach;
- The **actions** available to the agent. Given a state s, $Action(s)$ returns a finite set of actions that can be executed in s. Each of these actions is **applicable** in s;
- A **Transition model** that describes what each action does. Given the state s and the action a, $Result(s, a)$ returns the state that results from doing action a in state s;
- An **action cost function**, denoted either as $Action{-}Cost(s, a, s')$ (when programming) or $c(s, a, s')$ (when doing math), gives the numeric cost of applying action a in state s to reach state s'.

The optimal solution has the lowest path cost among all solutions. We can represent the state space as a graph where the vertices are states and directed edges between them are actions. We make an abstract mathematical description when formulating the routing problem of getting from place A to place B.

Definition. The process of removing detail from a representation is called **abstraction**. There are multiple **levels of abstraction** but they are not defined singularly.

We can have two type of problems: standardized, which is intended to illustrate or exercise various problem-solving methods, or real-world, which is the one that people actually use.

Example. Below we list some examples of standardized problems[2]:

1. Simple Vacuum Cleaner World:

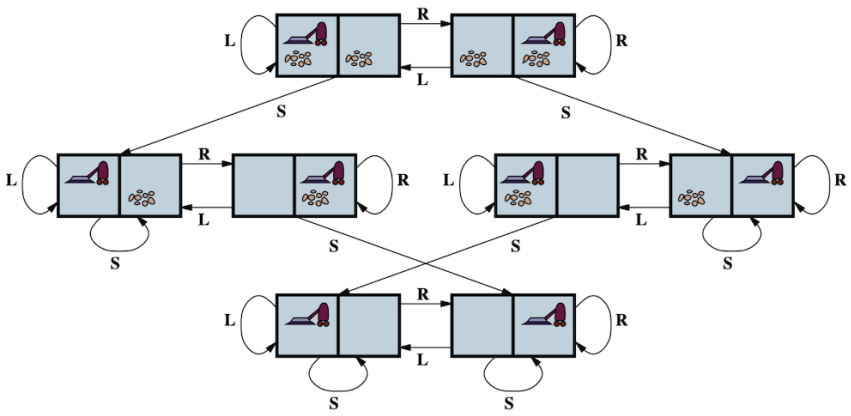


Figure 2.1: Vacuum Cleaner World Example

   - States: 2 locations, each $\{clean, dirty\}$;
   - Initial State: any;
   - Actions: $\{Left, Right, Suck\}$;
   - Transition Model: $Left$ [$Right$] if A [B], $Suck$ if $Clean \Rightarrow$ no effect;
   - Goal Test: check if squares are $Clean$;

[1] The book use word "environment", however it is more likely to be "agent".
[2] There are three examples, however just one is enough. The other two are intended to be used for exercises purposes.

- Path cost: each step costs 1 ⇒ path cost is number of steps in path.

2. Sliding-Tile Puzzle (8-Puzzle):

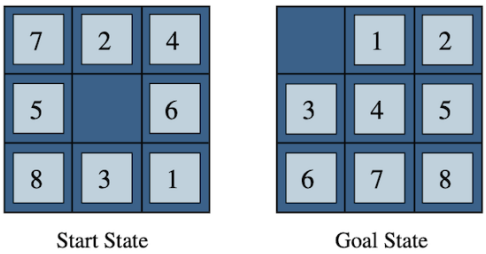


Figure 2.2: Sliding-Tile Puzzle Example

- States: Integer location of each tile;
- Initial State: any;
- Actions: moving $\{Left, Right, Up, Down\}$ the empty space;
- Transition Model: empty switched with the tile in target location;
- Goal Test: checks state corresponds with goal configuration;
- Path cost: each step costs 1 ⇒ path cost is number of steps in path.

3. 8-Queens Problem:

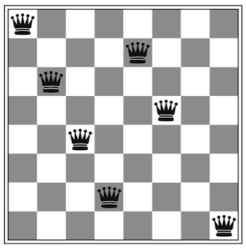


Figure 2.3: 8-Queens Problem Example

- States: any arrangement of 0 to 8 queens on the board;
- Initial State: no queens on the board;
- Actions: add a queen to any empty square;
- Transition Model: returns the board with a queen added;
- Goal Test: 8 queens on the board, none attacked by other queen;
- Path cost: none.

As Real-World Example we introduce the Robotic Assembly:

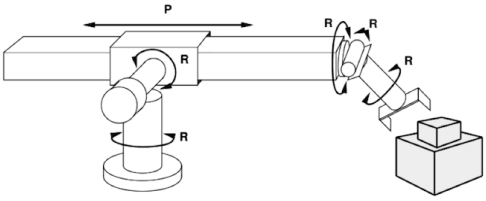


Figure 2.4: Robotic Assembly Example

- States: real-valued coordinates of robot joint angles, and of parts of the object to be assembled;
- Initial State: any arm position and object configuration;

- Actions: continuous motions of robot joints;
- Transition Model: position resulting from motion;
- Goal Test: complete assembly (without robot);
- Path cost: time to execute.

Search Algorithms A **search algorithm** takes a search problem as an input and returns either a solution or an indication of failure.

Definition. We can **expand** a node, by considering the available *Actions* for that state, using the *Result* function to see whose actions lead to, and **generating** a new node (that will be a **child node** or **successor node**) for each of the resulting states. Each child node will have also the **parent node** as the node visited before.

Definition. Search strategy is the task of selecting which state to expand.

Definition. To represent the expansion of all states, starting from the initial one (the **root**), we use a **search tree**/**DAG** (Directed Acyclic Graph).

All the leaves of the tree/dag represent either states to expand, solutions or dead-ends. First we introduce the Tree Search Algorithm: the basic idea of this algorithm is that we start from an initial state, we pick one leaf node and we expand it (we generate its successor). The algorithm ends when either a goal state is reached or no more candidates to expand are available (due to time-out or memory-out).

Algorithm 4 Tree Search Algorithm

function *Tree-Search*(*problem*) **return** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

Example. Consider the following simplified road map:

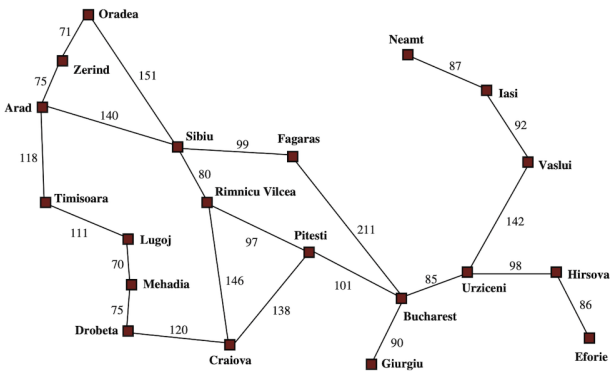


Figure 2.5: Simplified road map of Romania

What we obtain by applying the algorithm 4 is the following:

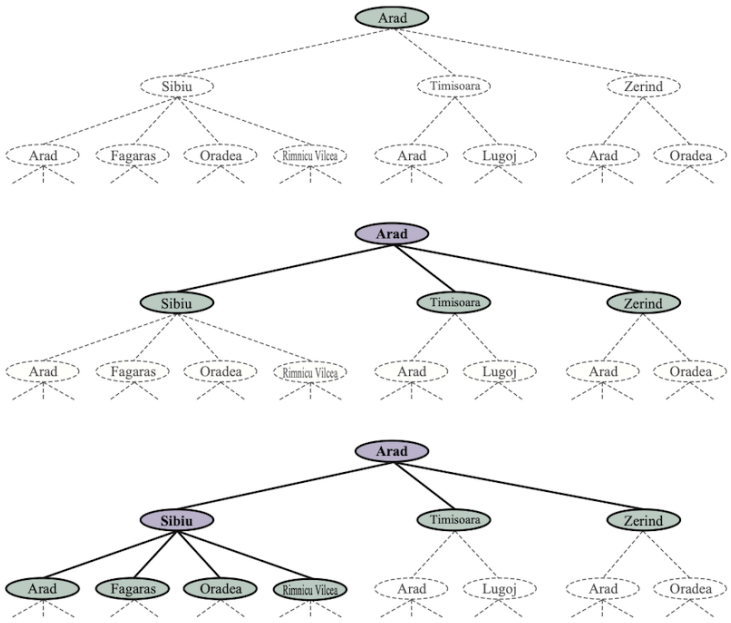


Figure 2.6: Tree search resolution of the simplified road map of Romania

We have the initial state as $\{Arad\}$. By expanding the initial state we obtain $\{Sibiu, Timisoara, Zerind\}$. Finally we pick and expand $Sibiu$ that leads to $\{Arad, Fargas, Oradea, Rimnicu\ Vicea\}$

Definition. When there is more than one way to get from one state to another (meaning that the same state is explored more than once) we are in a **redundant path**.

If a redundant path occur we are in trouble because there is no way to exit infinite loops or exponential complexity.

Graph search Looking at Figure 2.6 **Graph search** gives the solution to this problem: we add a data structure[3] which remembers every expanded node. Do not expand a node if it already occurs in explored set.

Algorithm 5 Graph Search Algorithm

function *Graph-Search*(*problem*) **return** a solution, or failure
initialize the frontier using the initial state of *problem*
initialize the explored set to be empty
loop do
if the frontier is empty **then return** failure
choose a leaf node and remove it from the frontier
if the node contains a goal state **then return** the corresponding solution
add the node to the explored set
expand the chosen node, adding the resulting nodes to the frontier only if not in the frontier or explored set

And now we clearly see that the exploration is way better:

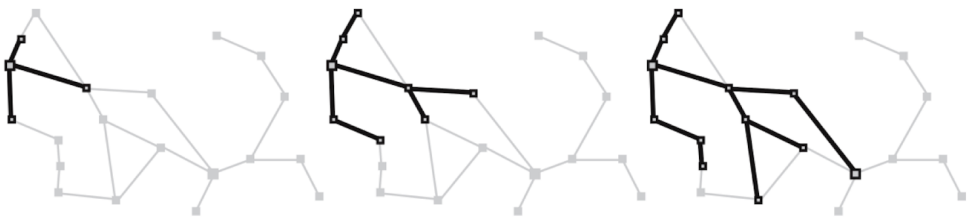


Figure 2.7: Graph search resolution of the simplified road map of Romania

[3]Typically we use a hash table because access costs $O(1)$.

One may think that a node is the same of a state, however it is not the case: a node is a data structure constituting part of a search tree.
We need a data structure to store the frontier[4]. The appropriate choice is a **queue** of some kind, because the operations on a frontier are:

- $Is\text{-}Empty(frontier)$ returns true only if there are no nodes in the frontier;
- $Pop(frontier)$ removes the top node from the frontier and returns it;
- $Insert(node, frontier)$ inserts an element into queue;

For saving the explored nodes the best way is a hash table with the primitives:

- $IsThere(Element, Hash)$ returns true iff element is in the hash;
- $Insert(Element, Hash)$ inserts element into hash.

Three kinds of queue are used in search algorithms:

- A **priority queue** first pops the node with the minimum cost according to some evaluation function, f, it has cost $O(\log(n))$. It is used in best-first search;
- A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first, it has cost $O(1)$. We use it in breadth-first-search;
- A **LIFO queue** or last-in-first-out queue (also known as a **stack**) pops first the most recently added node, it has cost $O(1)$; we shall see it is used in depth-first search.

Definition. If we do not use any domain knowledge, apply rules arbitrarily and do an exhaustive search strategy, we talk about **uninformed strategies**.

Definition. If we use domain knowledge and apply rules following heuristics we talk about **informed strategies**.

Strategies are evaluated along the following dimensions:

- Completeness: does it always find a solution if one exists?;
- Time complexity: how many steps to find a solution?;
- Space complexity: how much memory is needed?;
- Optimality: if it finds a least-cost solution?.

Time and space complexity are measured in terms of:

- b: maximum branching factor of the search tree;
- d: depth of the least-cost solution;
- m: maximum depth of the state space.

All this said we conclude with these pseudocodes:

[4]The frontier is the set of current leaves. It separates the state-space graph into the explored region and the unexplored region.

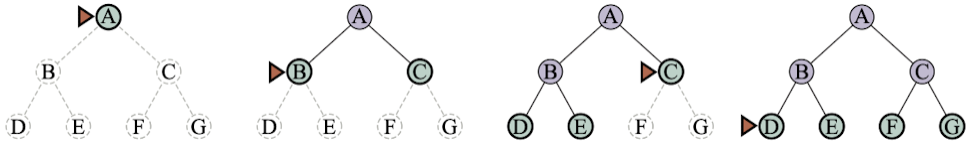


Figure 2.8: BFS Example

Algorithm 6 All Algorithms in detail

```
function Child-Node(problem, parent, action) return a node
    return a node with
      State = problem.Result(parent.State, action),
      Parent = parent, Action = action,
      Path-Cost = parent.Path-Cost + problem.Step-Cost(parent.State, action)

function Tree-Search(problem) return a solution, or failure
    fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← Remove-Front(fringe)
        if Goal-Test(problem, State(node)) then return node
        fringe ← InsertAll(Expand(node, problem), fringe)

function Expand(node, problem) return a set of nodes
    successors ← the empty set
    for each action, result in Successor-Fn(problem, State[node]) do
        s ← a new Node
        Parent-Node[s] ← node
        Action[s] ← action
        State[s] ← result
        Path-Cost[s] ← Path-Cost[node] + Step-Cost(node, action, s)
        Depth[s] ← Depth[node] + 1
        add s to successors
    return successors

function Graph-Search(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← Remove-Front(fringe)
        if Goal-Test(problem, State[node]) then return node
        if State[node] is not in closed then
            add State[node] to closed
            fringe ← InsertAll(Expand(node, problem), fringe)
```

Uninformed Search Strategies The first strategy is **breadth-first search**, the idea is to expand first the shallowest unexpanded node, the frontier is implemented as a FIFO queue.

Algorithm 7 Breadth First Search Algorithm

```
function Breadth-First Search(problem) return a solution, or failure
    node ← a node with State = problem.Initial-State
    Path-Cost = 0
    if problem.Goal-Test(node.State) then return Solution(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if Empty(frontier) then return failure
        node ← Pop(frontier)                    ▷ chooses the shallowest node in frontier
        add node.State to explored
        for each action in problem.Actions(node.State) do
            child ← Child-Node(problem, node, action)
            if child.State is not in explored or frontier then
                if problem.Goal-Test(child.State) then return Solution(child)
                frontier ← Insert(child, frontier)
```

To process all solutions takes $O(b^d)$ in time complexity and $O(b^d)$ in memory size. It is complete and optimal if all costs are 1.
Another important uninformed search is **uniform-cost** search, which allows costs > 0: the basic idea is to expand first the node with the lowest path cost $g(n)$, the frontier is a priority queue ordered by $g()$.

Algorithm 8 Uniform Cost Search Algorithm

```
function Uniform-Cost-Search(problem) return a solution, or failure
    node ← a node with State = problem.Initial-State
    Path-Cost = 0
    frontier ← a priority queue ordered by Path-Cost, with node as the only element
    explored ← an empty set
    loop do
        if Empty(frontier) then return failure
        node ← Pop(frontier)                    ▷ chooses the lowest-cost node in frontier
        if problem.Goal-Test(node.State) then return Solution(node)
        add node.State to explored
        for each action in problem.Actions(node.State) do
            child ← Child-Node(problem, node, action)
            if child.State is not in explored or frontier then
                frontier ← Insert(child, frontier)
            else if child.State is in frontier with higher Path-Cost then
                replace that frontier node with child
```

The time complexity to process all solutions is in $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ while the memory size is in $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, where C^* is the cost of the cheapest solution and ϵ is the minimum arc cost. This search strategy is complete and optimal too.
A completely different approach but still very important is **depth-first search**[5]: the idea here is to expand first the deepest unexpanded nodes, using a LIFO queue for the frontier. Figure 2.9 shows an example. The time complexity is in $O(b^m)$ while the memory size is in $O(bm)$ (for tree). It is incomplete in the case of infinite spaces and complete (in the tree version we have to prevent loops). Unfortunately this search is not optimal unless the optimal solution is the leftmost (rightmost) one.

The DFS algorithm works well with finite graphs but with infinite ones it has problems. **Depth-limited search** solves the incompleteness problems related to DFS, in fact it is nonetheless than a DFS with a depth limit l (for **l**imit). In case the goal is at depth d:

- If $d > l$ then the algorithm is incomplete;

[5]For this search type there is no algorithm provided, if there will be time left it will be added.

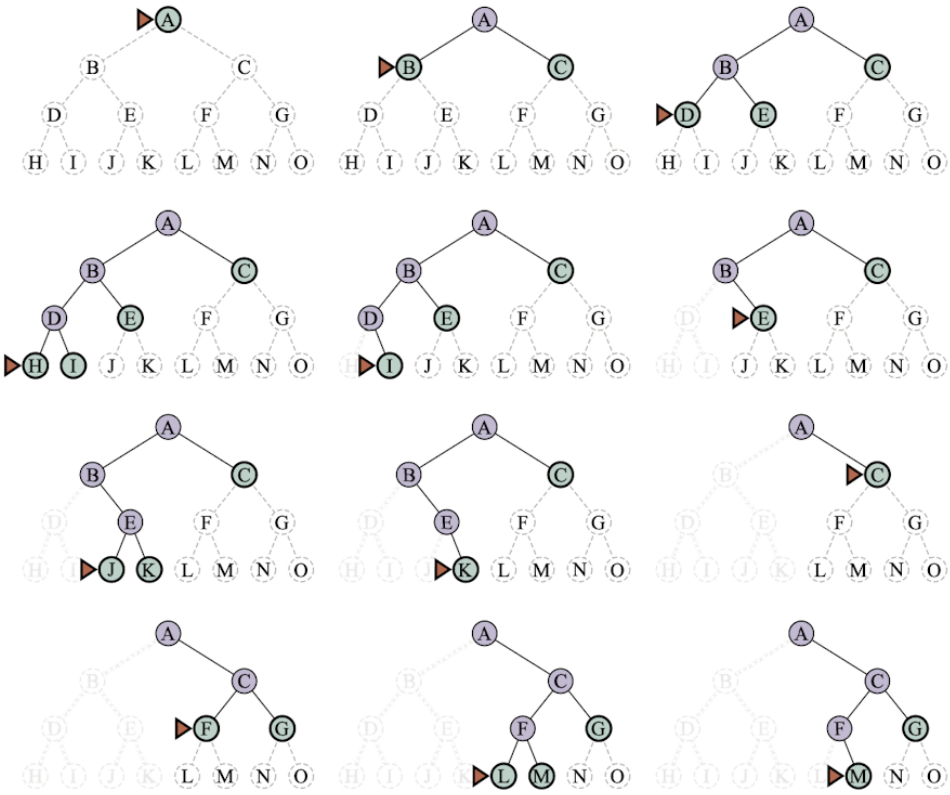


Figure 2.9: DFS example

- Otherwise it will take $O(b^l)$ steps;

Algorithm 9 Depth Limited Search Algorithm

function *Depth-Limited-Search*(*problem*, *limit*) **return** a solution, or failure/cutoff
  return *Recursive-DLS*(*Make-Node*(*problem.Initial-State*), *problem*, *limit*)

function *Recursive-DLS*(*node*, *problem*, *limit*) **return** a solution, or a failure/cutoff
  if *problem.Goal-Test*(*node.State*) **then return** *Solution*(*node*)
  else if $limit = 0$ **then return** *cutoff*
  else
    $cutoff\_occurred \leftarrow false$
    for each *action* **in** *problem.Actions*(*node.State*) **do**
      $child \leftarrow$ *Child-Node*(*problem*, *node*, *action*)
      $result \leftarrow$ *Recursive-DLS*(*child*, *problem*, $limit - 1$)
      if $result == cutoff$ **then**
        $cutoff\_occurred \leftarrow true$
      else if $result \neq failure$ **then return** *result*

    if *cutoff_ocurred* **then return** *cutoff*
    else return $failure$

The last uninformed search strategy is **Iterative Deepening Search**, here the idea is to call DLS iteratively for increasing depths. It combines these advantages:

- It is complete;
- It take $O(b^d)$ steps;
- It requires $O(bd)$ memory;

- Explores a single branch at a time;
- Optimal only if step cost is 1.

The pseudocode is very easy:

Algorithm 10 Iterative-Deepening Search Algorithm

```
function Iterative-Deepening Search(problem) return a solution, or failure
    for depth ← 0 to ∞ do
        result ← Depth-Limited-Search(problem,depth)
        if result ≠ cutoff then return result
```

And below an example to understand it better:

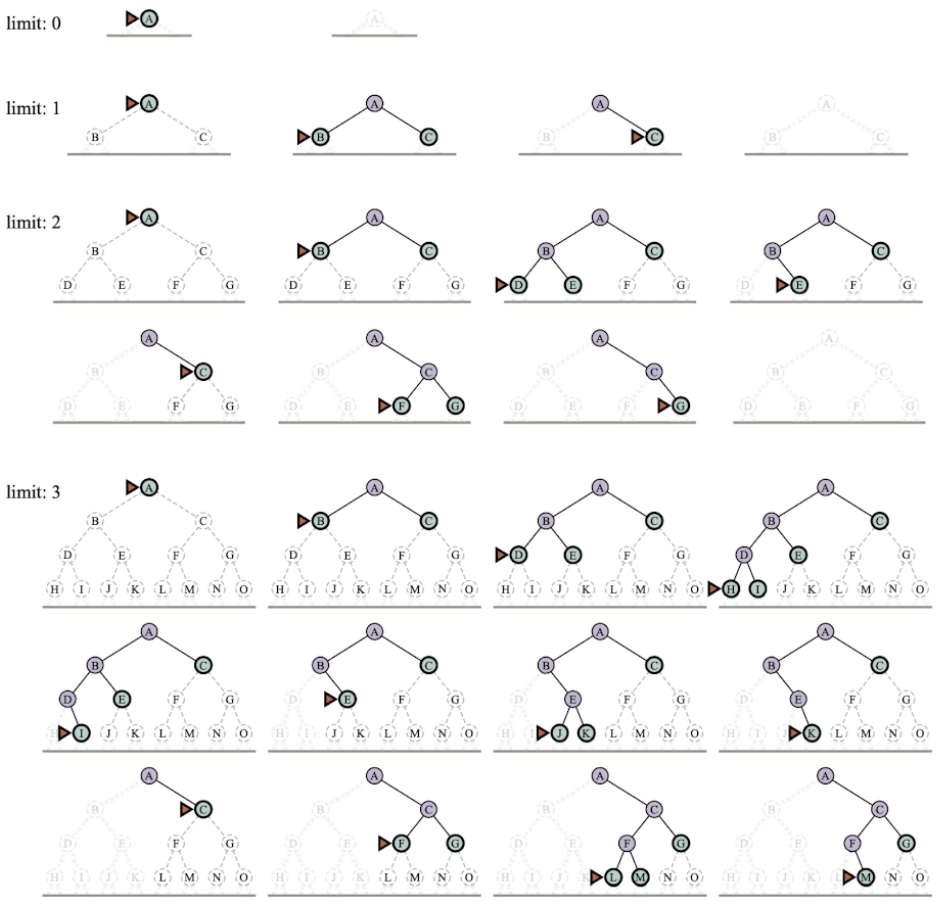


Figure 2.10: Iterative Deepening Search example

A different approach, **Bidirectional Search**, starts two simultaneous searches: forward from start node and backward from goal node until they meet each other. We end up with a cost of $O(b^{d/2}+b^{d/2}) \ll O(b^d)$.
To sum up:

Criterion	Breadth First	Uniform Cost	Depth First	Depth Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$

Table 2.1: Summary Table

1. complete if b is finite;
2. complete if step costs $\geq \epsilon$ for some positive ϵ;
3. optimal if step costs are all identical;
4. if both directions use BFS.

Informed Search Strategies With these strategies (called also **Heuristic Strategies**) it is possible to find solutions more efficiently than an uninformed strategy. We first define a **heuristic function**, denoted $h(n)$:

$$h(n) = \text{estimated cost of the cheapest path from the state at node } n \text{ to a goal state}$$

For example we can estimate the distance from the current state to a goal by computing the straight line distance on the map between two points in route-finding problems. A good strategy to start with is **Greedy Best-First Search**, here the idea is to expand first the node n with lowest estimate cost to the closest goal $h(n)$, the frontier is implemented with a priority queue ordered by $g(n) = h(n)$. Consider as example the road map of figure 2.5, we can compute $h(x)$ as the straight-line distances to $Bucarest$: We

Arad	366	Fagaras	176	Mehadia	241	Sibiu	253
Bucharest	0	Giurgiu	77	Neamt	234	Timisoara	329
Craiova	160	Hirsova	151	Oradea	380	Urziceni	80
Drobeta	242	Iasi	226	Pitesti	100	Vaslui	199
Eforie	161	Lugoj	244	Rimnicu Vilcea	193	Zerind	374

Table 2.2: Values of h_{SLD} - Straight-Line Distances to Bucharest

end up with a tree like this:

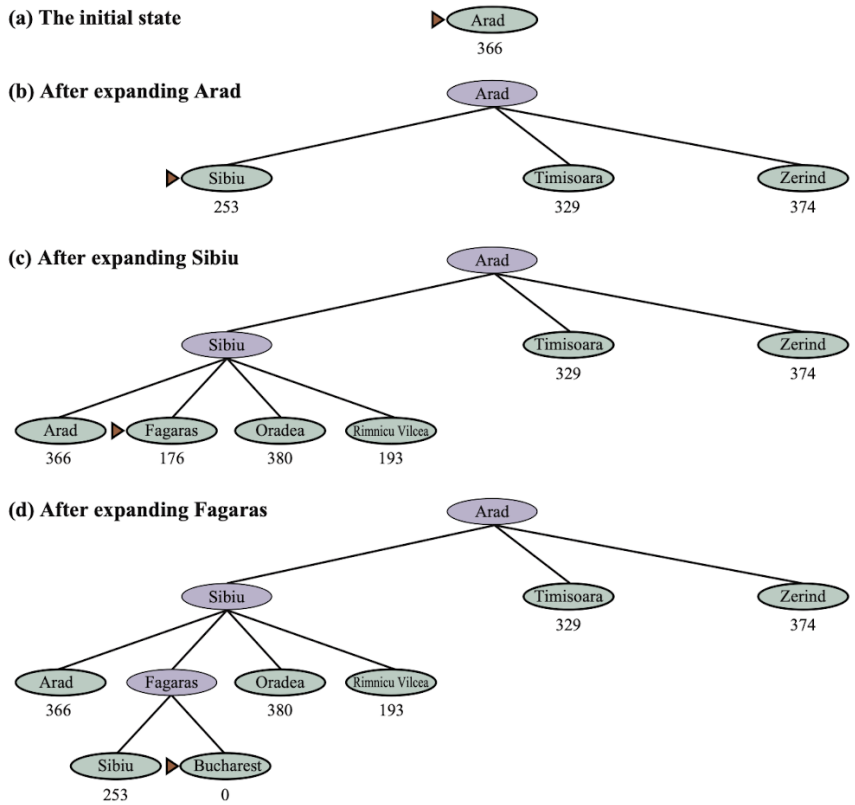


Figure 2.11: Greedy Best-First Search Example

Definition. The key property is **admissibility**: an **admissible heuristic** is the one that never overestimates the cost to reach a goal. In other words:

$$h(n) \leq h^*(n) \text{ where } h^*(n) \text{ is the true cost from } n$$

However one may notice that this path is **not** optimal, the optimal one would be $\{Arad, Sibiu, Rimnicu\ Vilcea, Pitesti\}$. This lead to an important conclusion: this algorithm is not guaranteed to find neither the best solution nor the best path toward a solution. It also takes some completeness issues that DFS have, one is very important: tree-based greedy best-first search is not complete because it may lead to infinite loops. The time complexity is in $O(b^d)$, if the heuristic is good, it may give good improvements. The memory size is in $O(b^d)$.
The best form of best-first search is A^* **Search**, the idea here is to avoid expanding paths that are already expensive and combine uniform cost with greedy search, obtaining:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$ is the cost so far to reach n;
- $h(n)$ is the estimated cost to reach the goal from n;
- $f(n)$ is the estimated total cost of the path through n to goal.

In A^* search the property of **admissibility** is also important, in addition there is another: consistency.

Definition. $h(n)$ is **consistent** (aka **monotonic**) iff, for every successor n' of n generated by any action a with step cost $c(n, a, n')$, $h(n)$ verifies the triangular inequality, meaning that:

$$h(n) \leq c(n, a, n') + h(n')$$

Theorem

If $h(n)$ is admissible, then A^* search is optimal.

Demonstration. Suppose some sub-optimal goal G_2 is in the frontier queue. n is an unexpanded node on a shortest path to an optimal goal G. Then:

$$\begin{aligned} f(G_2) &= g(G_2) \text{ since } h(G_2) = 0 \\ &> g(G) \text{ since } G_2 \text{ sub-optimal} \\ &\geq f(n) \text{ since } h \text{ is admissible} \end{aligned}$$

A^* will not pick G_2 from the frontier queue before n.

There are some properties about A^* graph search:

- If $h(n)$ is consistent, then $h(n)$ is admissible;
- If $h(n)$ is consistent, then $f(n)$ is non-decreasing along any path:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

- If A^* graph search selects a node n from the frontier, then the optimal path to that node has been found. In fact, if not so, there would be a node n' in the frontier on the optimal path to n (because of the graph separation property). Since f is non decreasing along any path, $f(n') \leq f(n)$ and since n' is on the optimal path to n, $f(n') < f(n)$ then n' would have been selected before n;

Theorem

If $h(n)$ is consistent, then A^* graph search is optimal.

Demonstration. A^* expands nodes in order of non decreasing value of f and gradually adds f-contours of nodes. If C^* is the cost of the optimal solution path, then:

1. A^* search expands all nodes such that $f(n) < C^*$;
2. A^* might expand some of the nodes on goal contour such that $f(n) = C^*$ before selecting a goal node;
3. A^* does not expand nodes such that $f(n) > C^*$ (this technique is called **pruning**).

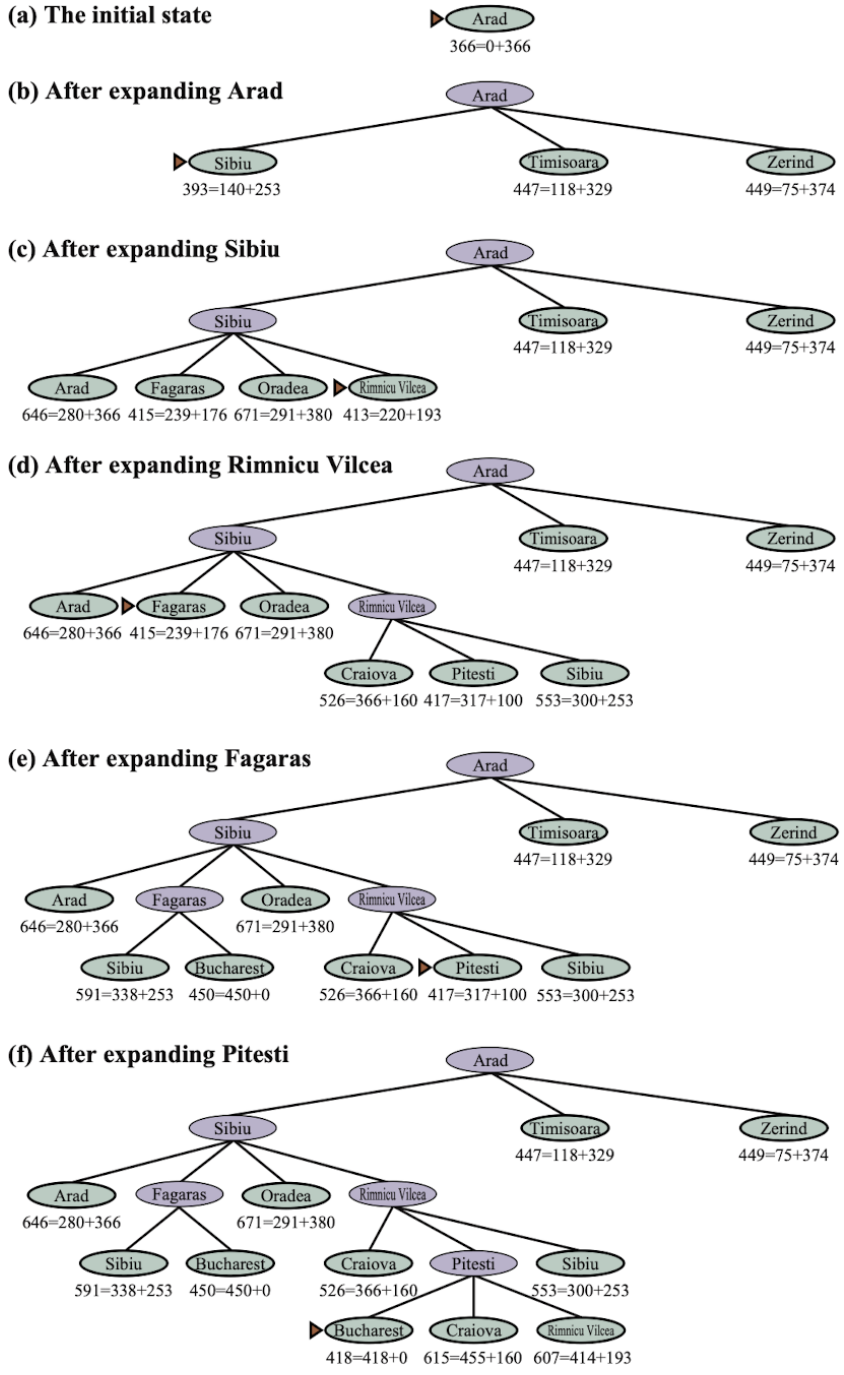


Figure 2.12: A^* Best-First Search Example

Before continue we have to define three variables:

- h^* is the true cost from n (as we have seen before);
- $\epsilon = \dfrac{h^* - h}{h^*}$ (relative error);
- b^ϵ is the effective branching factor.

A^* search takes time complexity in $O((b^\epsilon)^d)$ and with a good heuristic may give dramatic improvements, while memory size is in $O(b^d)$. A^* is both complete and optimal.
There are some memory bounded heuristic search algorithms that, thanks to A^* and best-first search, are way better than the uninformed ones:

- Iterative deepening A^*, here cutoff information is the f-cost $(g + h)$ instead of depth;
- Recursive best-first search, it attempts to mimic standard best-first search with linear space;
- Memory bounded A^*, it drops the worst leaf node when memory is full.

Last we want to understand what is the best admissible/consistent heuristic. First we need to define **dominance**.

Definition. Let $h_1(n)$, $h_2(n)$ admissible heuristics. $h_2(n)$ **dominates** $h_1(n)$ iff $h_2(n) \geq h_1(n)$ for all n. And thus, $h_2(n)$ is better for search. In addition, let h_{12} be $\max(h_1(n), h_2(n))$, then h_{12} is also admissible and h_{12} dominates both $h_1(n)$, $h_2(n)$.

In order to characterize the quality of a heuristic, it is very useful to use the **effective branching factor** b^*.

Definition. If the total number of nodes generated by A^* for a particular problem is N and the solution depth is d, then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain $N + 1$ nodes, Thus:

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

The most ideal value of b^* is 1.

Example. As example consider the 8-puzzle game:

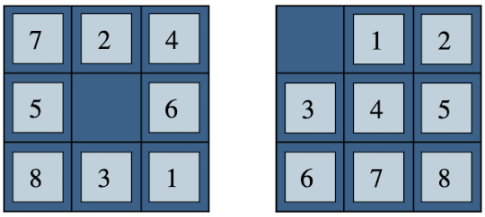


Figure 2.13: Heuristic 8-puzzle Example

- $h_1(n)$ is the number of misplaced tiles;
- $h_2(n)$ is the total of the Manhattan distance over all tiles;

Thus, $h_1(S) = 8$, $h_2(S) = = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18 \Rightarrow h^*(S)$ is 26. Both $h_1(n)$ and $h_2(n)$ are admissible but $h_2(n)$ dominates $h_1(n)$.

It is possible to compute the heuristic on a **relaxed** version of our problem.

Definition. A problem with fewer restrictions on the actions is called a **relaxed problem**.

Any optimal solution in the original problem is also a solution in the relaxed problem and the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.

Example. Again, consider the 8-puzzle above 2.13, all actions are regulated from this rule:

"*A tile can move from square A to square B if A is horizontally or vertically adjacent to B, and B is blank.*"

It is possible to generate three relaxed problems by removing one or both of the conditions:

1. "*A tile can move from square A to square B if A is adjacent to B*";
2. "*A tile can move from square A to square B if B is blank*";
3. "*A tile can move from square A to square B.*

2.2 Beyond Classical Search

Andrea Bonomi

In the section before we assume a small category of problems, all with these properties:

1. Observable;
2. Deterministic;
3. Known Environment;
4. Solution is a sequence of actions.

What if we relax these assumptions?

1. Release 4: local search;
2. Release 2: search with non deterministic actions;
3. Release 1: search with no observability or with partial observability;
4. Release 3: online search.

Local Search If we relax the fourth assumption, or else we do not keep track of the path, we talk about **Local search** algorithms that operate by searching from an initial state to neighboring states with a systematic exploration of the search space, without keeping track of the paths, nor the set of states that have been reached. In many problems the path to goal is irrelevant because the goals are expressed as conditions, not as explicit list of goal states. These algorithms can also solve optimization problems, in which the aim is to find the best state according to an objective function. Beware that a complete local search algorithm is guaranteed to always find a solution while an optimal local search algorithm is guaranteed to always find a maximum/minimum solution. To understand local search, consider the states of a problem laid out in a **state-space landscape**.

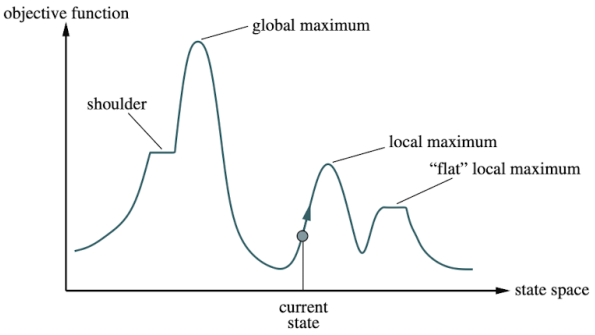


Figure 2.14: Local Search State-Space Landscape

The aim is to find the highest peak, the **global maximum**, and we call the process **hill climbing**. If elevation corresponds to cost, then the aim is to find the lowest valley, a **global minimum**, and we call it **gradient descent**.

The idea of hill-climbing is that it keeps track of one current state and on each iteration moves (following the steepest ascent) to the neighboring state with highest value. It terminates when it reaches a "peak" where no neighbour has higher value. The algorithm is pretty simple though:

Algorithm 11 Hill-Climbing Search Algorithm

function *Hill-Climbing*(*problem*) **return** a state that is a local maximum
  current ← *Make-Node*(*problem.Initial-State*)
  loop do
    neighbor ← a highest valued successor state of *current*
    if *neighbor.Value* ≤ *current.Value* **then return** *current.State*
    current ← *neighbor*

Example. Consider the 8-queen puzzle game, the neighbour states are generated by moving one queen vertically. The $Cost(h)$ is the number of queen pairs on the same row, column, or diagonal and the goal is to have $h = 0$. Considering the example below, the positions hill climbing will consider first are those

Figure 2.15: 8-Queen Puzzle Hill Climbing Example

with smaller h values, in the example above it is $h = 12$.

Hill climbing is incomplete because it can get stuck in any of the following (keep in mind Figure 2.14):

- Local maximum;
- Ridges: sequence of local maximum;
- Plateaus: can be either a flat area that has not uphill exit or a shoulder from which progress is possible.

There are variants of hill climbing like stochastic hill climbing (chooses at random from among the uphill moves), first choice hill climbing (stochastic hill climbing by generating successors randomly until it is better than the current state) and random restart hill climbing (the simplest one: "*if at first you do not succeed, try, try again*").

The only way to exit the incompleteness is to allow some "*bad*" moves, for example by picking a random move. It is important to decrease the size and frequency of these moves. Such algorithm is called **Simulated Annealing**. It is inspired from metallurgy and the idea is that a temperature parameter T slowly decreases with step *schedule*. The probability of picking a "bad move" can decreases first exponentially with the "badness" of the move $|\Delta E|$ or as T goes down.

Algorithm 12 Simulated Annealing Algorithm

function *Simulated-Annealing*(*problem*, *schedule*) **return** a solution state
problem is a problem
schedule is a mapping from time to "temperature"

$current \leftarrow Make\text{-}Node(problem.Initial\text{-}State)$
for $t = 1$ **to** ∞ **do**
$T \leftarrow schedule(t)$
if $T = 0$ **then return** *current*
next ← a randomly selected successor of *current*
$\Delta E \leftarrow next.Value$ - $current.Value$
if $\Delta E > 0$ **then** $current \leftarrow next$
else $current \leftarrow next$ only with probability $e^{\Delta E/T}$

Last of the local search algorithms is **local beam search**. This algorithm keeps track of k states rather than just one. It starts with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats. Beware that this is different from running k random restarts in parallel instead of in sequence: searches that find good states recruit other searches to join them, thus, information is shared among k search threads. The only problem is that quite often, all k nodes end up in the same local hill. Stochastic beam search solves this problem by choosing k random successors.
Another possibility are **genetic algorithms** that, instead picking random successors, it pick successor states generated by combining two parent states. States are represented as strings over a finite alphabet (e.g. {0, 1}). Initially we pick k random states. At each step we rate the parent states according to a fitness function, k parent pairs are selected at random, with probability increasing with their fitness, last, for each parent pair

1. A crossover point is chosen randomly;
2. A new state is created by crossing over the parent strings;
3. The offspring state is subject to (low-probability) random mutation.

It ends when some state is fit enough, or due to timeout.

Algorithm 13 Genetic Algorithm

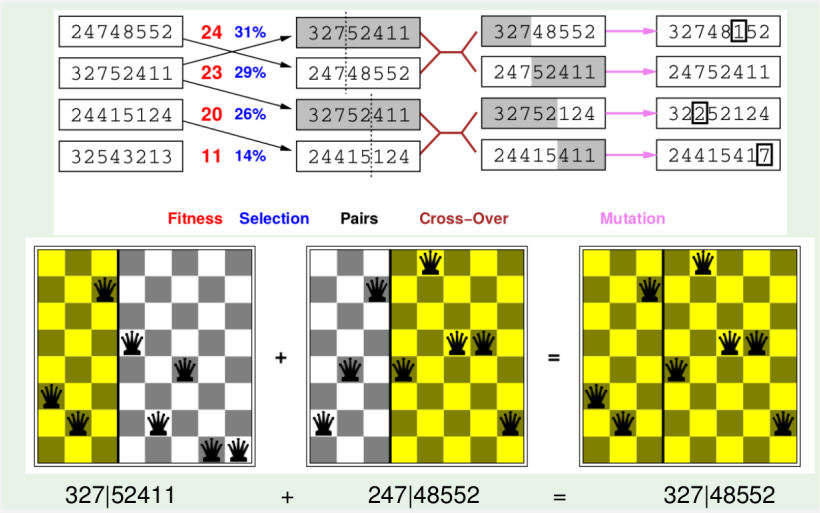
function *Genetic-Algorithm*(*population*, *Fitness-Fn*) **return** an individual
population is a set of individuals
Fitness-Fn is a function that measures the fitness of an individual

repeat
new_population ← empty set
for $i \leftarrow 1$ **to** *Size*(*population*) **do**
$x \leftarrow Random\text{-}Selection(population, Fitness\text{-}Fn)$
$y \leftarrow Random\text{-}Selection(population, Fitness\text{-}Fn)$
$child \leftarrow Reproduce(x, y)$
if (small random probability) **then** $child \leftarrow Mutate(child)$
add *child* to *new_population*
population ← *new_population*
until some individual is fit enough, or enough time has elapsed
return the best individual in *population*, according to *Fitness-Fn*

function *Reproduce*(x, y) **return** an individual
$n \leftarrow Length(x)$
$c \leftarrow$ random number from 1 to n
return $Append(Substring(x, 1, c), Substring(y, c + 1, n))$

Note

There is a slight difference between the pseudocode included here and the exercises proposed by both rseba and Dragons. In their examples, they form random couples of parents (a parent can be chosen more than once), and they make them reproduce but, instead of producing only a child, they produce two. One the opposite of the other. The main difference, at the end, is that with the pseudocode a couple can generate only one child (with its own crossover point), while in the examples couples generate two children with the same crossover point. Thus, more couples are generated in the pseudocode.



Search with non deterministic actions In partial observable and non deterministic environments, the solution to a problem is no longer a sequence, but rather a **conditional plan** that specifies what to do depending on what percepts agent receives while executing the plan.

Example. Consider the vacuum cleaner example, we introduce the erratic vacuum world, the *Suck* action works as follow:

- When applied to a dirty square the action cleans the square and sometimes cleans up in an adjacent square, too;
- When applied to a clean square the action sometimes deposits dirt on the carpet.

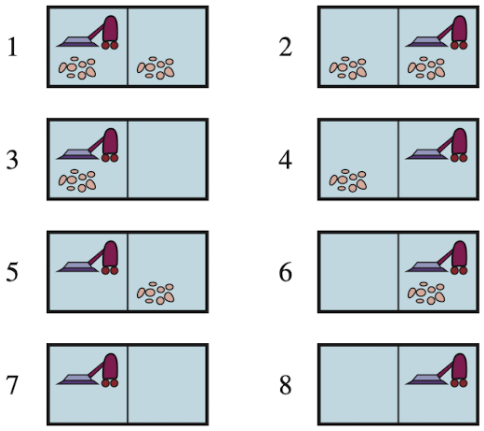


Figure 2.16: Erratic Vacuum World Example

We define a function *Result* that returns a set of possible outcome states, $Result(1, Suck) = \{5, 7\}$. We have a **conditional plan** that solves the problem and contains nested conditions on future percepts:

$$[Suck, \ \textbf{if } State = 5 \textbf{ then } [Right, Suck] \textbf{ else } [\,]]$$

How to find the contingent solution to a non deterministic problem?

Definition. In a deterministic environment, the only branching is introduced by the agent's own choices in each state: "*I can do this action or that action*". These are **OR nodes**.

Definition. In a non deterministic environment, branching is also introduced by the environment's choice of outcome for each action. These are the **AND nodes**.

Definition. Alternating this two kinds of nodes leads to an **AND-OR tree**.

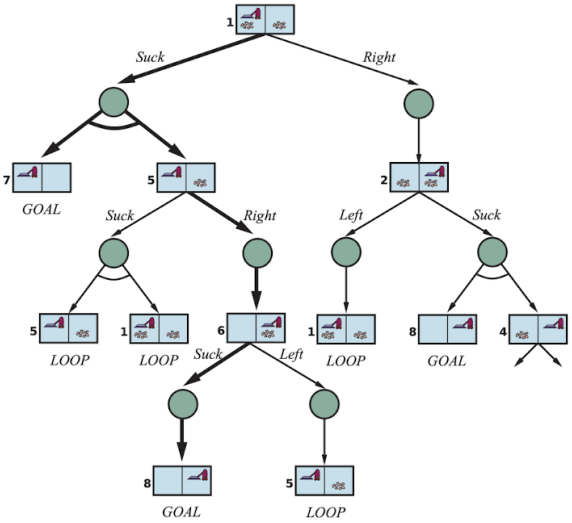


Figure 2.17: AND-OR Tree of the Erratic Vacuum World Example

Algorithm 14 And-Or Search Algorithm

function *And-Or-Search*(*problem*) **return** conditional plan, or failure
  return *Or-Search*(*problem.Initial-State*, *problem*, [ ])

function *Or-Search*(*state*, *problem*, *path*) **return** a *conditional plan*, or *failure*
  if *problem.Goal-Test*(*state*) **then return** the empty plan
  if *state* is on *path* **then return** *failure*
  for each *action* **in** *problem.Actions*(*state*) **do**
    plan ← *And-Search*(*Results*(*state*, *action*), *problem*, [*state*|*path*])
    if *plan* ≠ *failure* **then return** [*action*|*plan*]
  return *failure*

function *And-Search*(*states*, *problem*, *path*) **return** a *conditional plan*, or *failure*
  for each s_i **in** *states* **do**
    $plan_i$ ← *Or-Search*(s_i, *problem*, *path*)
    if $plan_i$ == *failure* **then return** *failure*
  return [
  if s_1 **then** $plan_1$
  else if s_2 **then** $plan_2$ ...
  else if s_{n-1} **then** $plan_{n-1}$
  else $plan_n$
  ]

Sometimes it is the case in which acyclic solutions are no longer available. Thus, *And-Or-Search* would return a failure. However sometimes it is possible to accept a cyclic solution, this happens only when every leaf is a goal state and a leaf is reachable from every point in the plan.

Search in Partially Observable Environments We have two cases: no observation or partial observation. Consider the first one, we do not have information at all. The solution to this category of problem is a sequence of actions, not a conditional plan. The idea is that the agent searches in the space of belief states rather than in physical ones.

Example. Think, again, about the vacuum cleaner that knows the geography of its world, but it does not know its location nor the distribution of dirt (take as reference figure 2.16).

- Initial state: {1, 2, 3, 4, 5, 6, 7, 8};
- After action *Right*, state is {2, 4, 6, 8};
- After action sequence [*Right*, *Suck*], state is {4, 8};
- After action sequence [*Right*, *Suck*, $Left$, *Suck*], state is {7}.

Definition. The **belief-state** is the set of all of physical states. If P has N states, then the sensorless problem has up to 2^N states.

A belief-state problem has the following components:

- States: the belief-state space contains every possible subset of the physical states. If P has N states, then the belief-state problem has 2^N belief states;
- Initial state: typically the belief state consists of all states in P;
- Actions: suppose the agent is in belief state $b = \{s_1, s_2\}$, but $Actions_p(s_1) \neq Actions_p(s_2)$; then the agent is unsure of which actions are legal. We can take all of the actions,

  $$Actions(b) = \bigcup_{s \in b} Actions_p(s)$$

  because we can suppose the illegal actions have no effect on the environment. If our supposition is wrong we take the intersections, meaning the set of legal actions in all the states;
- Transition model: for deterministic actions, the new belief state has one result state:

  $$b' = Result(b, a) = \{s' \ : \ s' = Result_p(s, a) \text{ and } s \in b\}$$

  With non determinism, the new belief state consists of all the possible results of applying the action to any of the states in the current belief state:

  $$\begin{aligned} b' = Result(b, a) &= \{s' \ : \ s' \ \in \ Results_p(s, a) \text{ and } s \in b\} \ \# \textit{ Note the s in Results.} \\ &= \bigcup_{s \in b} Results_p(s, a) \end{aligned}$$

  The size of b' will be the same or smaller than b for deterministic actions, but may be larger than b for non deterministic actions;
- Goal test: the agent possibly achieves the goal if any state s in the belief state satisfies the goal test of the underlying problem. In other words, $GoalTest(b)$ holds if and only if $GoalTest_p(s)$ holds, $\forall s \in b$;
- Action/Path cost: if the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of several values, in other words: $StepCost(a, b)$ is defined from $StepCost_p(a, s)$, $\forall s \in b$. For now we assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.

Example. Consider as example the vacuum world, in "a" the prediction is the next belief state for the sensorless vacuum world with the deterministic action *Right*. In "b" the prediction is for the same belief state and action in the slippery version of the sensorless vacuum world

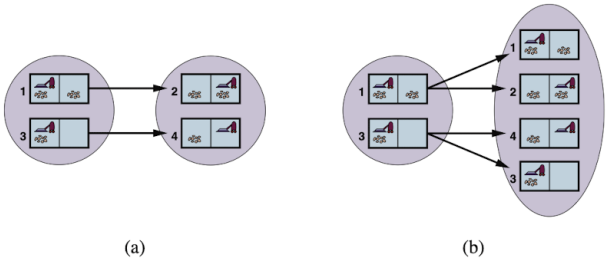


Figure 2.18: Belief State Example for the plain ("a") and slippery("b") vacuum world

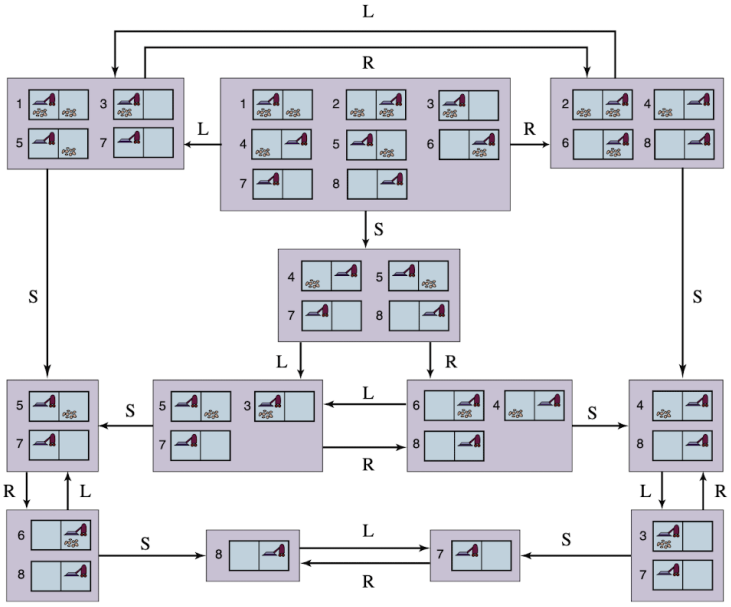


Figure 2.19: Belief State Example for the sensorless vacuum world

To sum up the belief state problem formulation:

- if $b \subseteq b'$, then $Result(b, a) \subseteq Result(b', a)$;
- if a is deterministic, then $|Result(b, a)| \leq |b|$;
- The agent might achieve the goal earlier than $GoalTest(b)$ holds, but it does not know it;
- An action sequence is a solution for b iff it leads b to a goal;
- If an action sequence is a solution for a belief state b, then it is also a solution for any belief state b' so that $b' \subseteq b$;
- We can apply to the belief-state space any search algorithm, we can discard a path reaching a belief state b if $b' \subseteq b$ has already been generated and discarded. If a solution for b has been found, then any $b' \subseteq b$ is solvable. This dramatically improves efficiency.

Now comes the partial observable environment case. For a partial observable problem, the problem specification will specify a $Percept(s)$ function that returns the percept received by the agent in a given state. If sensing is non deterministic, then we use a $Percepts$ function that returns a set of possible percepts. We can think of the transition model between belief states for partially observable problems as occurring in three stages:

1. Prediction stage: it computes the belief state resulting from the action (same as for the sensorless problem):

$$\hat{b} = Predict(b, a) = Result_{(sensorless)}(b, a) = \{s' \ : \ s' = Result_P(s, a) \text{ and } s \in b\}$$

2. Possible percepts stage/Observation prediction: it computes the set of percepts that could be observed in the predicted belief state:

$$PossiblePercepts(\hat{b}) = \{o \ : \ o = Percept(s) \text{ and } s \in \hat{b}\}$$

3. Update stage: it computes, for each possible percept, the belief state that would result from the percept:

$$b_o = Update(\hat{b}, o) = \{s \ : \ o = Percept(s) \text{ and } s \in \hat{b}\}$$

The agent needs to deal with possible percepts at planning time, because it will not know the actual percepts until it executes the plan.

By putting all the stages together we obtain:

$$Result(b, a) = \{b_o \ : \ b_o = Update(Predict(b, a), o) \text{ and } o \in PossiblePercepts(Predict(b, a))\} \quad (2.1)$$

It is the set of belief states, one for each possible percepts o. $b_o \subseteq \hat{b}, \ \forall o$ implies that sensing reduces uncertainty. If sensing is deterministic then the b_o's are disjoint.

Example. Consider the deterministic actions case of the local sensing vacuum cleaner:

- $\hat{b} = Predict(\{1, 3\}, Right) = \{2, 4\}$;
- $PossiblePercepts(\hat{b}) = \{[B, Dirty], [B, Clean]\}$;
- $Result(\{1, 3\}, Right) = \{\{2\}, \{4\}\}$.

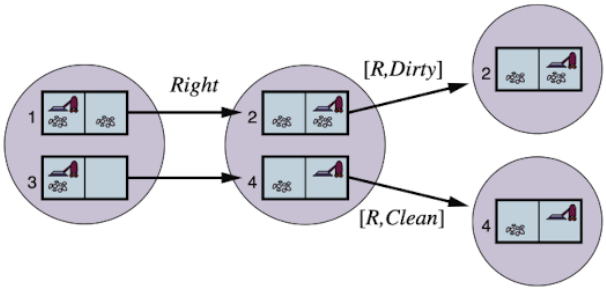


Figure 2.20: Local-Sensing Vacuum Cleaner Example

Example. Consider the non deterministic actions case of the slippery local sensing vacuum cleaner:

- $\hat{b} = Predict(\{1, 3\}, Right) = \{1, 2, 3, 4\}$;
- $PossiblePercepts(\hat{b}) = \{[B, Dirty], [A, Dirty], [B, Clean]\}$;
- $Result(\{1, 3\}, Right) = \{\{2\}, \{4\}\}$.

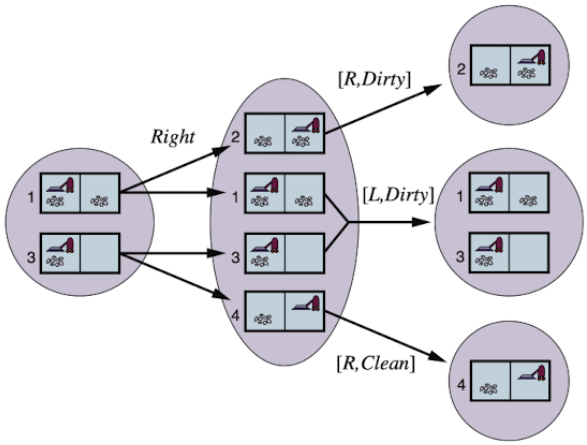


Figure 2.21: Slippery Local-Sensing Vacuum Cleaner Example

We saw the AND-OR Trees, we can appply them here and obtain a conditional plan as solution.

Example. For example, for the initial percept $[A, Dirty]$ (in deterministic case) we obtain $[Suck, Right,$ if $Bstate = \{6\}$ then $Suck$ else $[\,]]$.

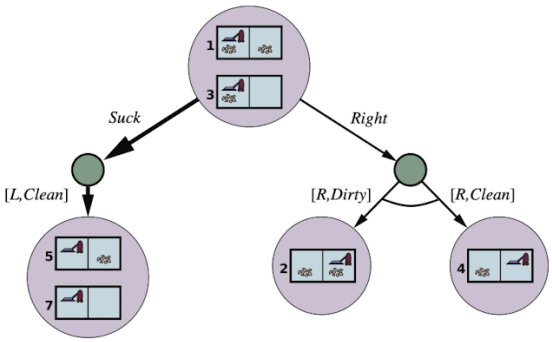


Figure 2.22: AND-OR Tree for the Local-Sensing Vacuum Cleaner Example

The agent for partially observable environments is similar to the simple problem solving agent of 2.1: it formulates a problem as a belief state, it calls a search algorithm, which is the AND-OR Graph, and execute the solution. What is the difference though? The solution is a conditional plan and not an action sequence and when executing the algorithm, the agent needs to maintain its belief state as it performs actions and receives percepts.

Online Search Until now we saw offline search: it computes a complete solution before executing it. In **Online Search** the agent interleaves computation and action. Online search is necessary in dynamic domains or unknown domains (third property) and it is useful to prevent search blowup in non deterministic domains. The agent now knows only $Action(s)$, which returns the list of actions allowed in s, the step-cost function $c(s,\ a,\ s')$ and the $GoalTest(s)$ function. The agent might know an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state. The agent's objective is to reach a goal state while minimizing the cost. The big trouble is that online search may face dead ends and there is no algorithm for avoiding them.

Definition. An **adversary argument** is an adversary constructing the state space while the agent explores it and putting the goals and dead ends wherever it chooses.

Example. To understand an adversary argument consider this example:
An online search algorithm that has visited states S and A cannot tell if it is in the top state space or the bottom one; the two look identical based on what the agent has seen. Therefore, there is no way it could know how to choose the correct action in both state spaces.

Basically the agent have to create and maintain a map of the environment. The agent will expand the node it is physically in and will backtrack physically too. Notice that this works only if actions are always reversible and that an agent can go on a long walk even if it is close to the solution. Like depth-first search, hill-climbing search has the property of locality in its node expansions, in fact, because it keeps just one current state in memory, hill-climbing search is already an online search algorithm. Unfortunately it leaves the agent stuck in local minima, and, moreover, random restart cannot be used. A possible solution comes from **random walk**, it selects randomly one available actions from the current state, gives preference to actions that have not yet been tried and eventually finds a goal or complete its exploration if space is finite. Unfortunately this algorithm is very slow.

In order to get better solutions, we can use **Learning Real-Time A***: which adds memory to hill climbing. The idea is to store the current best estimate $H(s)$ of the cost to reach the goal from each state that has been visited. The algorithm builds a map of environment in the $result[s,\ a]$ table, chooses the best move (apparently) a according to current $H()$ and updates the cost estimate $H(s)$ for the state s it has just left by using the cost estimate of the target state s':

$$H(s) = c(s, a, s') + H(s')$$

One important detail is that actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost, namely $h(s)$. This **optimism under uncertainty**

Algorithm 15 Online DFS Agent Algorithm

function *Online-DFS-Agent*(s') **return** an action
  Persistent:
    s' is a percept that identifies the current state
    result is a table indexed by state and action, initially empty
    untried is a table that lists, for each state, the actions not yet tried
    unbacktracked is a table that lists, for each state, the backtracks not yet tried
    s, a are the previous state and action respectively, initially null

  if *Goal-Test*(s') **then return** *stop*
  if s' is a new state (not in *untried*) **then**
    untried[s'] $\leftarrow$ *Actions*(s')
  if s is not null **then**
    result[s, a] $\leftarrow$ s'
    add s to the front of *unbacktracked*[s']
  if *untried*[s'] is empty **then**
    if *unbacktracked*[s'] is empty **then return** *stop*
    else $a \leftarrow$ an action b such that *result*[s', b] = *Pop*(*unbacktracked*[s'])
  else $a \leftarrow$ *Pop*(*untried*[s'])
  $s \leftarrow s'$
  return a

encourages the agent to explore new, possibly promising paths. Remember that a LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment.

Algorithm 16 LRTA* Agent Algorithm

function *LRTA*-Agent*(s') **return** an action
  Persistent:
    s' is a percept that identifies the current state
    result is a table mapping (s, a) to s', initially empty
    H is a table mapping s to a cost estimate, initially empty
    s, a are the previous state and action respectively, initially null

  if *Goal-Test*(s') **then return** *stop*
  if s' is a new state (not in H) **then**
    $H[s'] \leftarrow h(s')$
  if s is not null **then**
    result[s, a] $\leftarrow$ s'
    $H[s] \leftarrow \min_{b \in Actions(s)}$ *LRTA*-Cost*(s, b, *result*[s, b], H)
  $a \leftarrow \operatorname{argmin}_{b \in Actions(s')}$ *LRTA*-Cost*(s', b, *result*[s', b], H)
    ▷ An action b in *Actions*(s') that minimizes *LRTA*-Cost*()
  $s \leftarrow s'$
  return a

function *LRTA*-Cost*(s, a, s', H') **return** a cost estimate
  if s' is undefined **then return** $h(s)$
  else return $c(s, a, s') + H[s']$

Example. Consider the following example of five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and every link has an action cost of 1. The red state marks the location of the agent, and the updated cost estimates at each iteration have a double circle.

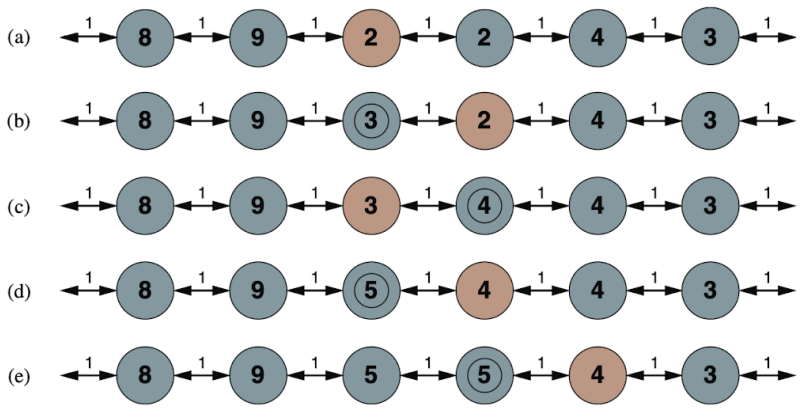


Figure 2.23: LRTA* Example

2.3 Adversarial Search

Filippo Momesso

Games Games are a form of **multi-agent environment** because the outcome of the game depend on what other agent do and how their behaviour affect the agent's actions. We saw that multi-agent environments can be *cooperative* or *competitive*, the latter category is also known as **adversarial problems** (games).

The main characteristics are:

- **easy to represent**,
- interesting due to **computational complexity's hardness**,
- **metaphors** for important real life domains like markets, sports, politics.

In games the solution is a **policy** $S \mapsto A$, which specifies an action for every possible move of the opponent, and each state is evaluated by an **utility function** which returns a score for the game position.
There are many different types of games classified by their features:

- deterministic or stochastic
- number of players
- zero-sum or general games
- perfect information or imperfect

We first consider two-players games. The players are called MAX and MIN, MAX moves first and the two alternate until the game is over.

Definition. Games are a kind of search problems defined by:

- Initial State S_0: specifies how the game is set up at the start,
- $Player(s)$: defines which player has the move in a state,
- $Actions(s)$: returns the set of legal moves in a state,
- $Result(s, a)$: the transition model, defines the result of a move,
- $TerminalTest(s)$: true iff the game is over (if so, s terminal state),
- $Utility(s, p)$: defines the final numeric value for a game ending in state s for player p.

The initial state S_0, $Actions(s)$, $Result(s, a)$ recursively define the **search-space graph**, which is search tree that follows every sequence of moves all the way down to a terminal state, and is called the **game tree** as showed in Figure 2.24.
In *zero-sum* games what is good for one player is bad for the other and there is no "win-win" outcome, in other words the agents have opposite utilities and the game is purely adversarial with no cooperation. E.g. in chess Utility values are either $1 + 0$ or $0 + 1$ or $\frac{1}{2} + \frac{1}{2}$.

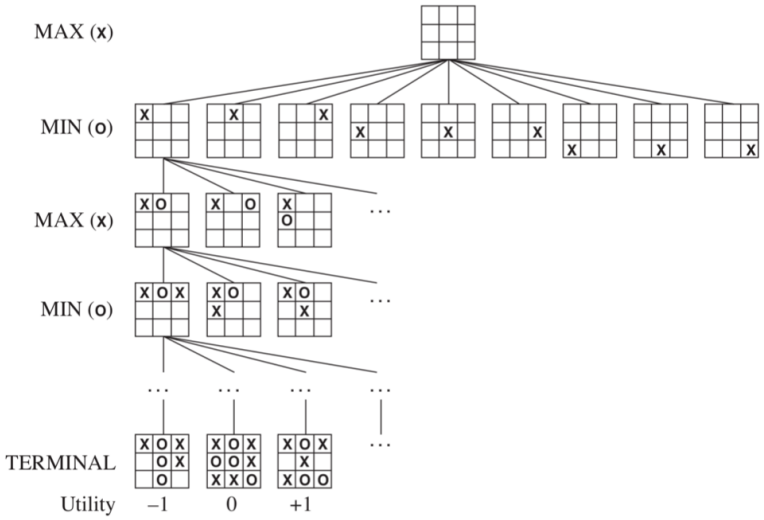


Figure 2.24: Tic-Tac Toe game tree.

Optimal Decisions in Games MAX must find a contingent strategy specifying a response to each of MIN's possible moves. For games with a win-lose outcome we could simply use AND-OR Search algorithm, but with games with multiple outcomes scores like chess we should use a more general algorithm called **Minimax Search**.

Definition. The optimal strategy can be determined by computing the **minimax value** for each state. We define a $Minimax(s)$ function which is the utility of being in that state, assuming that both players play optimally for the rest of the game. The minimax value of a terminal state is its utility. MAX player maximizes the minimax value, MIN player minimizes it.

$$Minimax(s, d) = \begin{cases} Utility(s, MAX) & \text{if } Is\text{-}Terminal(s) \\ max_{a \in Actions(s)} Minimax(Result(s, a)) & \text{if } To\text{-}Move(s) = MAX \\ min_{a \in Actions(s)} Minimax(Result(s, a)) & \text{if } To\text{-}Move(s) = MIN \end{cases} \tag{2.2}$$

Example. In Figure 2.25 a representation of the minimax tree. The $\triangle$ nodes are "MAX nodes" and the $\triangledown$ are the "MIN nodes". The terminal nodes show the utility values for MAX. The algorithm first recurses down to the three bottom-left nodes and uses the $Utility$ function on them to discover their values. Then it takes the minimum and returns it as the backed up value of node B. Similarly the backed-up values for C and D are computed. Finally the root A takes the maximum of those values.

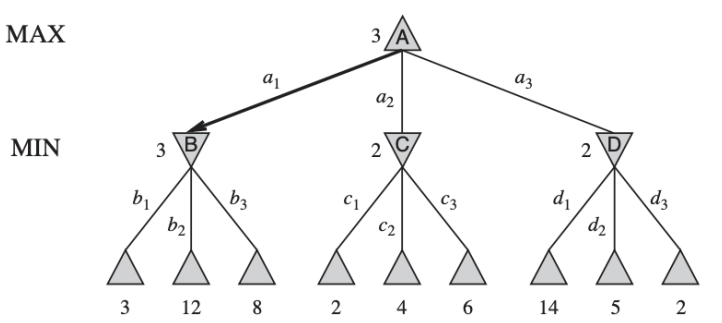


Figure 2.25: Minimax tree of a two-ply game.

From this recursive definition of the $Minimax(s)$ function it is trivial to derive the **Minimax-Search** algorithm. The algorithm proceeds all the way down to the leaves and then backs up the minimax value through the tree as the recursion unwinds.

Since it performs a **complete DFS exploration** of the minimax tree, time complexity is $O(b^m)$ and space complexity is $O(bm)$. It is **complete** and **optimal** if the opponent is optimal (if it is not it performs even better but it is not optimal in a formal sense). But, cAn wE Do bEtTeR ThAn tHiS[6]?

In the case of **multiplayer games** the solution is to replace the single value for each node with a vector of values. The terminal states will contain the utility for each agent. Each turn the agent chooses the action with the best value for itself. **Alliances** between agents are possible.

Algorithm 17 Minimax Search Algorithm

function $Minimax\text{-}Search(game, state)$ **return** *an action*
  $player \leftarrow game.To\text{-}Move(state)$
  $value, move \leftarrow Max\text{-}Value(game, state)$
  return $move$

function $Max\text{-}Value(game, state)$ **return** a $(utility, move)$ pair
  if $game.Is\text{-}Terminal(state)$ **then return** $game.Utility(state, player), null$
  $v, move \leftarrow -\infty$
  for each a **in** $game.Actions(state)$ **do**
    $v2, a2 \leftarrow Min\text{-}Value(game, game.Result(state, a))$
    if $v2 > v$ **then**
      $v, move \leftarrow v2, a$
  return $v, move$

function $Min\text{-}Value(game, state)$ **return** a $(utility, move)$ pair
  if $game.Is\text{-}Terminal(state)$ **then return** $game.Utility(state, player), null$
  $v, move \leftarrow +\infty$
  for each a **in** $game.Actions(state)$ **do**
    $v2, a2 \leftarrow Max\text{-}Value(game, game.Result(state, a))$
    if $v2 < v$ **then**
      $v, move \leftarrow v2, a$
  return $v, move$

Alpha-Beta Pruning We have seen that standard Minimax-Search is of exponential complexity. It is possible to almost cut it in half by applying a pruning approach called **alpha-beta pruning**, where pruning means "cutting parts of the tree" that make no difference on the outcome.
This algorithm is based on the two extra parameters in $Max\text{-}Value(state, \alpha, \beta)$
(and respectively $Min\text{-}Value(state, \alpha, \beta)$) function that describe bounds on the backed-up values that appear on the path.

- α = best value for MAX (highest) so far off the current path. α means "at least".
- β = best value for MIN (lowest) so far off the current path. β means "at most".

Considering a node n, it should be pruned if its value is lower than current alpha value for MAX, or respectively higher than current beta value for MIN, as showed in figure 2.26 on the following page.

[6] Sorry but it was necessary

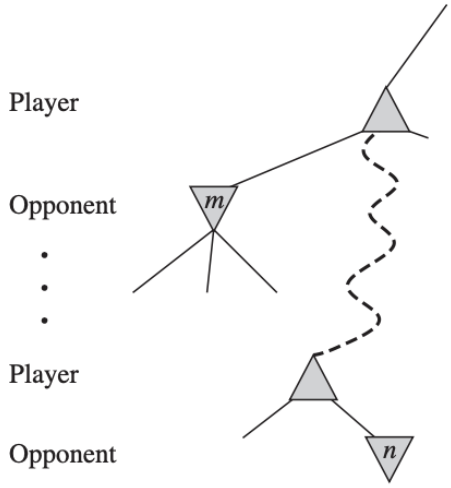


Figure 2.26: The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

Alpha-beta pruning algorithm is **correct** and reduces the complexity to $O(b^{\frac{m}{2}})$ if move ordering is "perfect", i.e. the first moves to be analyzed are the so called *killer moves* which allow to cut the rest of the sub-tree. With a random move ordering the complexity is $O(b^{\frac{3}{4}m})$. To improve performances it is possible to implement "graph versions" of alpha-beta pruning which address the problem of redundant paths in the tree by tracking explored states via hash table. Anyway even with alpha-beta pruning games like chess and go are still intractable. Full tree exploration would take too much time.

Algorithm 18 Alpha-Beta Pruning Algorithm

function *Alpha-Beta-Search*$(game, state)$ **return** *an action*
  $player \leftarrow game.To\text{-}Move(state)$
  $value, move \leftarrow Max\text{-}Value(game, state, -\infty, +\infty)$
  return move

function $Max\text{-}Value(game, state, \alpha, \beta)$ **return** a $(utility, move)$ pair
  if $game.Is\text{-}Terminal(state)$ **then return** $game.Utility(state, player), null$
  $v, move \leftarrow -\infty$
  for each a **in** $game.Actions(state)$ **do**
    $v2, a2 \leftarrow Min\text{-}Value(game, game.Result(state, a), \alpha, \beta)$
    if $v2 > v$ **then**
      $v, move \leftarrow v2, a$
      $\alpha \leftarrow Max(\alpha, v)$
    if $v \geq \beta$ **then return** v, move
  return $v, move$

function $Min\text{-}Value(game, state, \alpha, \beta)$ **return** a $(utility, move)$ pair
  if $game.Is\text{-}Terminal(state)$ **then return** $game.Utility(state, player), null$
  $v, move \leftarrow +\infty$
  for each a **in** $game.Actions(state)$ **do**
    $v2, a2 \leftarrow Max\text{-}Value(game, game.Result(state, a), \alpha, \beta)$
    if $v2 < v$ **then**
      $v, move \leftarrow v2, a$
      $\beta \leftarrow Min(\beta, v)$
    if $v \leq \alpha$ **then return** v, move
  return $v, move$

Heuristic Alpha-Beta Tree Search It is possible to cut off the search at limited depth and apply an **heuristic evaluation function** in order to treat nonterminal nodes as if they were terminal. The

Minimax(s) is modified in the following ways:

- Replace utility function $Utility(s)$ by a heuristic evaluation function $Eval(s)$, which estimates the states's utility.

- Replace the terminal test $TerminalTest(s)$ by a cutoff test $CutOffTest(s, d)$, that decides when to apply $Eval()$.

Definition. We get $H\text{-}Minimax(s, d)$ function which is defined as follows:

$$H\text{-}Minimax(s,d) = \begin{cases} Eval(s, MAX) & \text{if } Is\text{-}Cutoff(s,d) \\ max_{a \in Actions(s)} H\text{-}Minimax(Result(s,a), d+1) & \text{if } To\text{-}Move(s) = MAX \\ min_{a \in Actions(s)} H\text{-}Minimax(Result(s,a), d+1) & \text{if } To\text{-}Move(s) = MIN \end{cases} \tag{2.3}$$

The **heuristic evaluation function** $Eval(s, p)$ returns an estimate of the expected utility of state s to player p. For terminal states $Eval(s, p) = Utility(s, p)$ and for non terminal states it should be correlated with the actual chances of winning: $Utility(loss, p) \leq Eval(s, p) \leq Utility(win, p)$. Typically is computed as a weighted sum of features $Eval(s, p) = \sum_{i=1}^{n} w_i f_1(s)$ where w_i is a weight and f_i is a feature of the state (for example "number of white bishops").
For the cutoff test implementation the most straight forward method is to **set a fixed depth limit**. However, a more robust approach is to apply iterative deepening. It is even more efficient to apply $Eval()$ only to *quiescent states*, states which are unlikely to exhibit wild swings in value in the near future.

Important Remark

An important remark is that exact state values do not matter, it is only important the **order of states** in the tree. The behaviour of the heuristic alpha-beta pruning is the same under any monotonic transformation of $Eval()$, as Figure 2.27 shows.

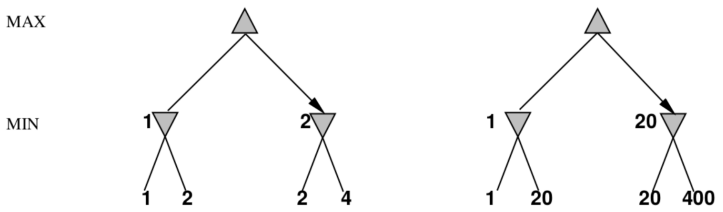


Figure 2.27: The behaviour of the heuristic alpha-beta pruning is preserved under any monotonic transformation of of $Eval()$

Stochastic Games Stochastic games mirror unpredictability by adding **random steps** (eg. dice rolls, coin flipping, card shuffles...). Due to unpredictability it is not possible to compute definite minimax values but we must resort to **expected values**. This because uncertain outcomes are controlled by chance instead of an adversary agent, so due to statistical properties it is correct to **average**.
As represented in Figure 2.28 on the following page, the basic idea is to include **chance nodes** in the game tree in addition to MAX and MIN nodes. Chance nodes represent stochastic events like a dice roll, are labeled with a **probability** and the outcoming arcs represent stochastic event outcomes.

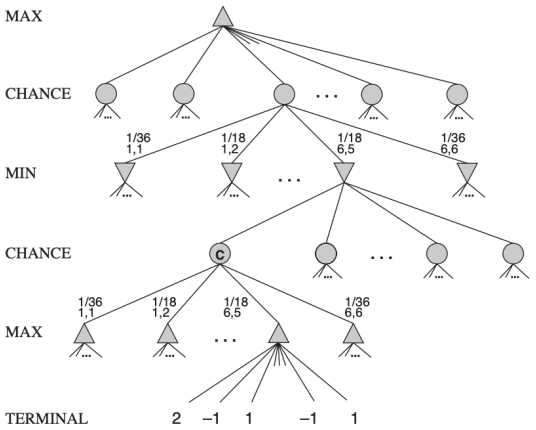


Figure 2.28: Stochastic Minimax game tree example.

Definition. We get the $ExpectMinimax(s)$ function which is defined as follows:

$$ExpectMinimax(s) = \begin{cases} Utility(s, MAX) & \text{if } Is\text{-}Terminal(s) \\ max_{a \in Actions(s)} ExpectMinimax(Result(s,a)) & \text{if } To\text{-}Move(s) = MAX \\ min_{a \in Actions(s)} ExpectMinimax(Result(s,a)) & \text{if } To\text{-}Move(s) = MIN \\ \sum_r P(r) ExpectMinimax(Result(s,r)) & \text{if } To\text{-}Move(s) = CHANCE \end{cases} \tag{2.4}$$

where r represent a possible outcome of the stochastic event and $Result(s, r)$ is the same state as s, with the additional fact that the result of the dice roll is r.

Important Remark

As shown in Figure 2.29, the behaviour of the ExpectMinimax algorithm is **not preserved** under monotonic transformations of Utility values due to the *expected value* operation. Only **positive linear transformation** preserve its behaviour. Therefore $Utility()$ should be proportional to the expected payoff for each terminal state.

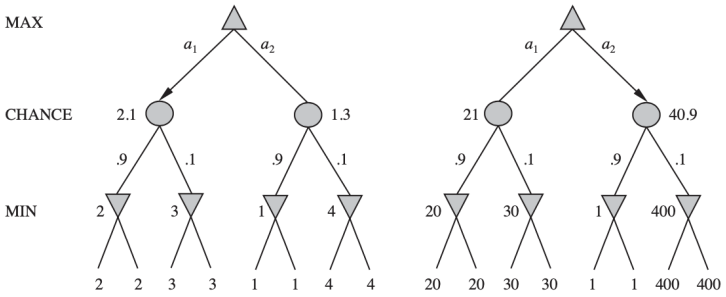


Figure 2.29: An order-preserving transformation on leaf values changes the best move.

Time complexity is $O(b^m n^m)$ where n is the number of distinct rolls. Notice that alpha-beta pruning is much less effective than with deterministic games and it is possible to use Heuristic variants of $ExpectMinimax()$ but only with low cutoff depth. This because the number of states explodes as the depth increases.

2.4 Constraint Satisfaction Problems

Giovanni Ambrosi

States representation Before introducing the main concepts of this chapter, let's define the three ways to represent states

Atomic A state is a black box **with no internal structure**.

Factored A state consists of a **vector of attribute values**.

Structured A state includes objects, each of which may have **attributes of its own as well as relationships to other objects**.

The three definitions above differ on the computational complexity and in the expressive power. Here, there is an image that gives an idea of the different representations.

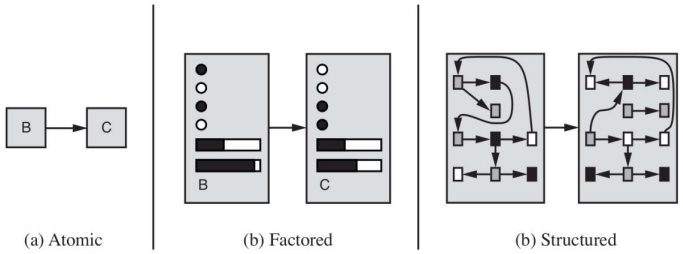


Figure 2.30: States representation

Constraint Satisfaction Problems, CSPs (aka Constraint Satisfiability Problems) In this chapter we will deal with factored representation of states. Let's have a look to the main differences that derive:

1. **state** is defined by a **set of variables values**, as mentioned above, from some domains;
2. goal test is a **set of constraints** specifying allowable combinations of values for subsets of variables. **A set of variable values is a goal iff the values verify all constraints**.

CSPs: Definitions A Constraint Satisfaction Problem consists in a tuple $< X, D, C >$, where:

1. $X \triangleq \{X_1,X_2,...,X_n\}$ is a set of variables;
2. $D \triangleq \{D_1,D_2,...,D_n\}$ is a set of (non-empty) domains;
3. $C \triangleq \{C_1,C_2,...,C_m\}$ is a set of constraints;

Notice that the constraints can be $\leq$ than domains and variables.
Each D_i is a set of allowable variables $\{v_i,...,v_k\}$ for variable X_i. Each C_i is a pair <scope,rel> where:

- scope is a tuple of variables that participate in the constraint;
- rel is a relation defining the values that such variables can take.

A relation is an explicit list of all tuples of values that satisfy the constraint (most often inconvenient), or an abstract relation. We need a language to express constraint relation (in the exercises constraints are often expressed as mathematical formula).

States, Assignments and Solutions A state in CSPs is an assignment of values to some or all of the variables $\{X_i, ..., X_n\}$ with their specific domains. An assignment can be:

- complete (or total) if every variable is given a value;
- incomplete (or partial)

An assignment that does not violate any constraint in the CSP is called a **consistent or legal assignment** and a **solution** to a CSP is a **consistent and complete assignment**. A CSP consists in finding one solution (or state if there is none). We mention also Constraint Optimization Problems (COPs) where the target is to provide a solution that maximize/minimize an objective function.

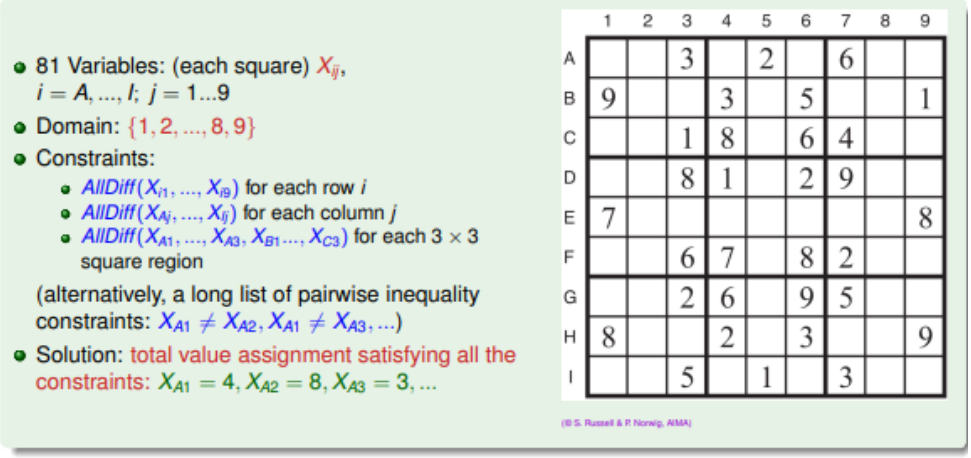


Figure 2.31

Constraint Graphs It is useful to represent CSPs as constraint graphs (aka network). The nodes of the graph correspond to variables of the problem, an edge connects any two variables that participate in a constraint as shown in the figure below.

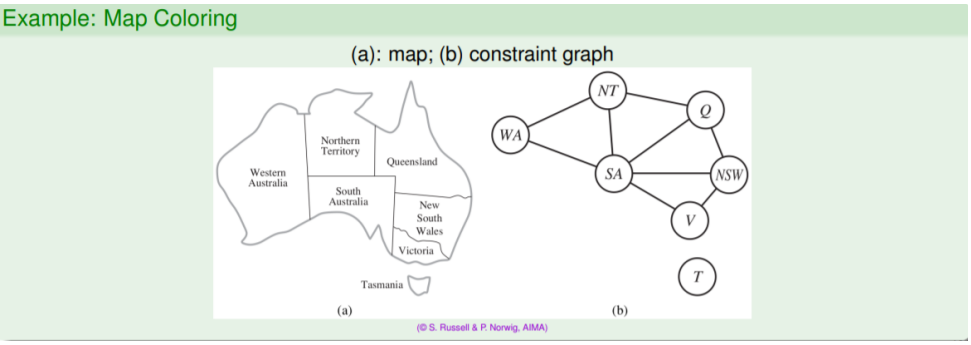


Figure 2.32: (a): map; (b): constraint graph

Varieties of Constraints We can subdivide the variables of CSPs as:

- Discrete variables:
  - Finite domains of domain size d (thus, d^n complete assignments or candidate solutions). For example Boolean CSPs. It is possible to define constraint by enumerating all combinations;
  - Infinite domains of infinite domain size (thus, infinite number of complete assingments). For example job scheduling. We need to use implicit constraints like $T_1 + d_1 \leq T_2$ rather than explicit tuples of values. There are some special solution algorithms for linear constraints on integer variables[7]. Meanwhile for non linear constraints the problem is undecidable.

[7]Linear constraints: where variables appears only in linear form.

- Continuous variables:
  - Linear constraints makes the CSP solvable in polynomial time by LP methods;
  - Non-Linear constraints make the CSP solvable but very hard.

What are these *constraints*? A complete list is this:

- **Unary** constraints: involve **one** single variable;
- **Binary** constraints: involve **pairs** of variables (most used in the exercises);
- **Higher-order** constraints: involve $\geq$ **3** variables;
- **Global** constraints: involve an arbitrary number of variables;
- **Preference** constraints (aka soft constraints): **describe preferences between/among solutions**.

Remark

k-ary contraints can be transformed into sets of binary constraints.
In real world CSPs involve often real/rational-valued variables, combinatorics and logic, optimization.

Constraint Propagation In state space an algorithm can only search.
With CSPs, an algorithm can:

Search Pick a new variable assignment.

Infer Apply constraint propagation
Use the constraints to reduce the set of legal candidate values for a variable.

Constraint propagation can either:

- be interleaved with search;
- be performed as a preprocessing step

Constraint propagation is a very important step, because we can eliminate inconsistent values (values that do not respect constraints) and simplify the structure of the graph/tree and, if used as a preprocessing step, do a cheaper search in the state space.

Types of Consistency In this paragraph are reported different types of (local) consistency: Node Consistency, Arc Consistency, Path Consistency, K-Consistency

- **Node Consistency**: X_i is node-consistent if all the values in the variable's domain satisfy its unary constraints. If every variable of a CSP is node-consistent then CSP is node-consistent too
  - **Node Consistency Propagation/1-consistency**: Node Consistency Propagation infers remove all values from the domain D_i of X_i which violate unary constraints on X_i.Unary constraints can be removed a priori by node consistency propagation.
- **Arc Consistency/2-consistency**: X_i is arc-consistent iff for every value d_i of X_i in D_i exists a value d_j for X_j in D_j which satisfy all binary constraints on $<X_i,X_j>$. A CSP is arc-consistent if every variable is arc-consistent with every other variable;
  - **Arc Consistency Propagation**: Arc Consistency Propagation consists in removing all values from the domains of every variable which are not arc-consistent with these of some other variables.
- **Path Consistency/3-consistency**: A two-variable set $\{X_i, X_j\}$ is path-consistent wrt. a third variable X_m if,for every assignment $\{ X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$;
- **K-Consistency**: A CSP is k-consistent iff for any set of $k-1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any other k_{t_h} variable.

Forward Checking

Definition. Forward checking is the simplest form of propagation (of arc-consistency). The idea is to **propagate information from assigned to unassigned variables**. Let's describe it in few steps. Assume to have a set of variables X, with X_i assigned:

- Choose an assignment for the variable X_i;
- Update remaining legal values $\forall X_j \in X s.t. X_j$ is an unassigned variable, which is connected to X_i by a constraint;
- Does not provide early detection for all failures;
- If X_i loses a value, neighbors of X_i need to be rechecked;

Example. Taking now in consideration the Map coloring example, after the previous definition, can we conclude anything?
NT and SA cannot both be blue.

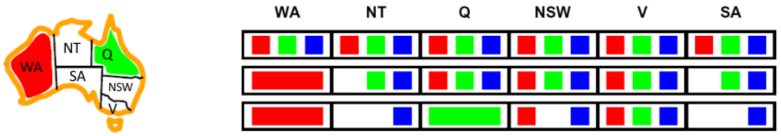


Figure 2.33

A well known arc-consistency algorithm, with Forward Checking, is AC-3 (code below), where every arc is arc-consistent, or some domain variable are empty. Arc-consistency can be interleaved with search or used as a preprocessing step. **Time and space complexity grow exponentially with k**.

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  inputs: *csp*, a binary CSP with components $(X,\ D,\ C)$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

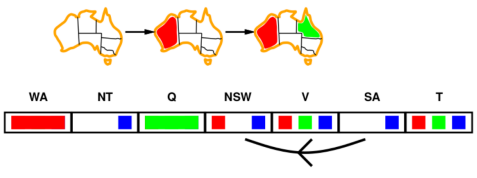
  while *queue* is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REVISE(*csp*, $X_i,\ X_j$) **then** // makes Xi arc-consistent wrt. Xj
      if size of $D_i\ =\ 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to *queue*
  return *true*

---

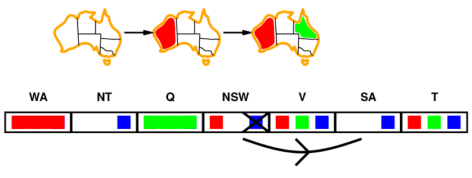
function REVISE(*csp*, $X_i,\ X_j$) **returns** true iff we revise the domain of X_i
  revised $\leftarrow$ *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised $\leftarrow$ *true*
  return *revised*

Figure 2.34: Algorithm AC-3

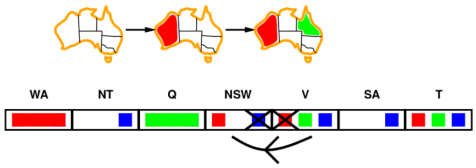
Example. (Arc-Consistency Propagation AC-3) We assume to have a set of variables X. If a variable X_i lose a value, the neighbours need to be rechecked. For example we have:



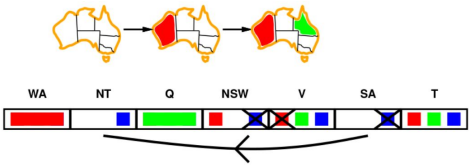
(1) $Revise(SA, NSW) \Rightarrow D_{SA}$ unchanged



(2) $Revise(NSW, SA) \Rightarrow D_{NSW}$ revised



(3) $Revise(V, NSW) \Rightarrow D_V$ revised



(4) $Revise(SA, NT) \Rightarrow D_{SA}$ revised

Example. (Sudoku) (consider AllDiff() as a set of binary constraints)
Apply arc-consistency propagation:

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 ⇒ E6=4

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7					**4**			8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

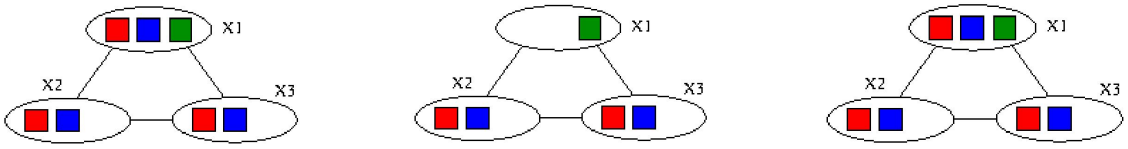
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 ⇒ I6=7

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7					**4**			8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1	**7**	3		

- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7,8,9 ⇒ A6=1

	1	2	3	4	5	6	7	8	9
A			3		2	**1**	6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7					**4**			8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1	**7**	3		

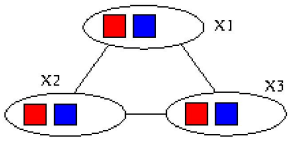
Path vs. Arc Consistency Taking the example below, can we say anything about X_1?
We can drop *red* and *blue* from D_1, and this infers the assignment $C_1 = green$.



In the end, can we say anything? Yes, the triplet is inconsistent.

Can **arc consistency** reveal it? **No**

Can **path consistency** reveal it? **Yes**



Backtracking Search Backtracking Search is a basic uninformed algorithm for solving CSPs. We explain the procedure with the following two ideas:

1. Pick one variable at a time;
2. Check constraints as long as you proceed;
   - pick only values which do not conflict with previous assignments;

- requires some computation to check the constraints;
- can detect if a partial assignments violate a goal → early detection of inconsistencies;

Definitely Backtracking Search is a DFS algorithm (example in Figure 2.36 on the following page) but with the two improvements above.

Backtracking Search Algorithm (BSA) Backtracking Search is a general purpose algorithm for generic CSPs. Eventually we can modify it adding some sophistication to the unspecified functions as i.e.

1. SelectUnassignedVariable(): which variable should be assigned next?
2. OrderDomainValues(): in what order should its values be tried?
3. Inference(): what inferences should be performed at each step?

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, assignment)
      if inferences ≠ failure then
        add inferences to csp
        result ← BACKTRACK(csp, assignment)
        if result ≠ failure then return result
        remove inferences from csp
      remove {var = value} from assignment
  return failure
```

Figure 2.35

In the next paragraphs we'll have a look at some procedures which say what variable to choose next, or what to try first in a backtracking search. (Most of them refers to Map Colouring example)

Variable Selection Heuristic The most important algorithm of this typology is Minimum Remaining Value (MRV) heuristic (aka most constrained variable or fail-first heuristic)

- Choose the variable with the fewest legal value → pick a variable that is most likely to cause a failure soon;
- If X has no legal values left, MRV heuristic selects X → failure detected immediately, avoid pointless search through other variables;
- (Otherwise) If X has one legal value left, MRV selects X → performs deterministic choices first!. Postpones nondeterministic steps as much as possible;

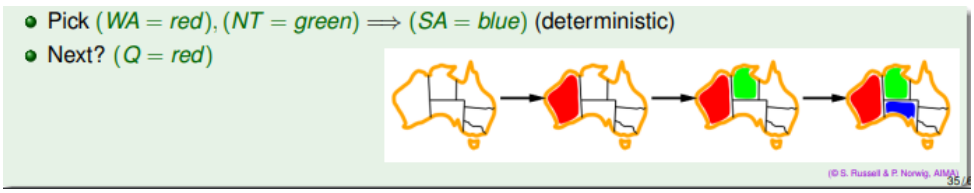


Figure 2.37

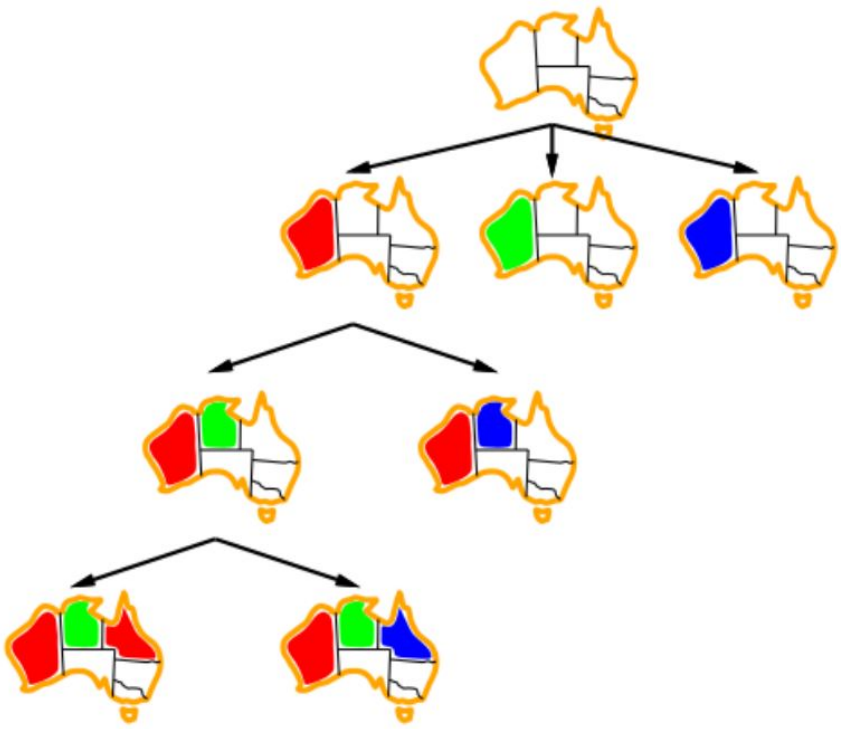


Figure 2.36: Example of Backtracking Search

We can improve the MRV by using Degree Heuristic (DH). DH is used as a tie-breaker in combination with MRV.

- Pick the variable with most constraints on remaining variables → attempts to reduce the branching factor on future choices.

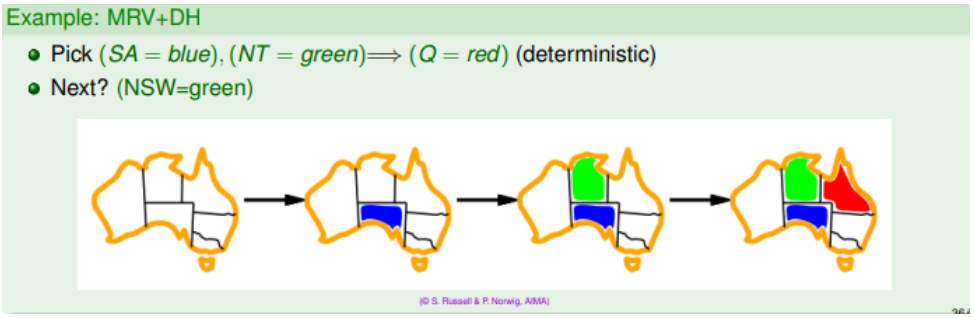


Figure 2.38

Value Selection Heuristic The most important algorithm is Least Constraining Value (LCV) heuristic

- Pick the value that rules out the fewest choices for the neighboring variables → tries maximum flexibility for subsequent variable assignments;

- Look for the most likely values first → improve chances of finding solutions earlier;

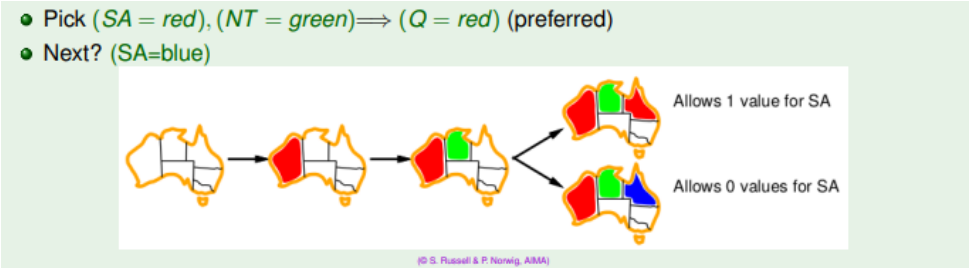


Figure 2.39: IMPORTANT! there is an error in the example: $SA = red$ is wrong, consider the example with $WA = red$

Inference After a choice, infer new domain reductions on other variables. This can help to:

- detect inconsistencies earlier;
- reduce search spaces;
- may produce unary domains (deterministic steps) → returned as assignments ("inferences").

Inference finds a tradeoff between **effectiveness and efficiency**. One of the simplest form of inference is the forward checking (see example below). Another form of inference is the AC-3 algorithm, more expensive than FC. Both of the techniques guarantee arc-consistency.

Backtracking Search with Forward Checking

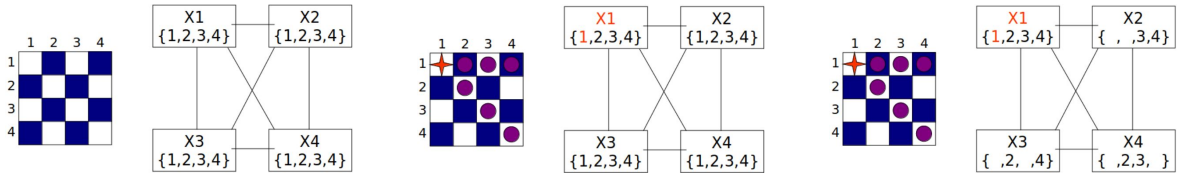
Example. Here is an example of backtracking search with FC (4-Queens problem). We report the images first and then explain them.

Notation

Please refer to the following conventions and explanations:

- X-axis has letters as coordinates (a,b,c,d);
- Y-axis has numbers as coordinates (1,2,3,4);
- Red cross is the position of the queen;
- Violet dots are the cells a queen can attack;

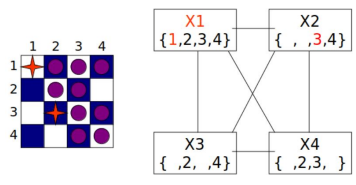
A point on the chessboard is expressed with a letter and a number, so a queen located in the high-left angle has coordinates a4.



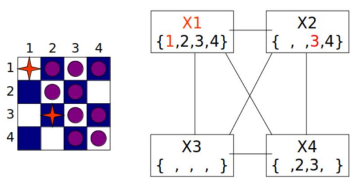
Step 1

Step 2

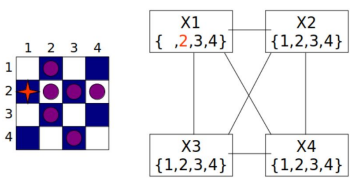
Step 3



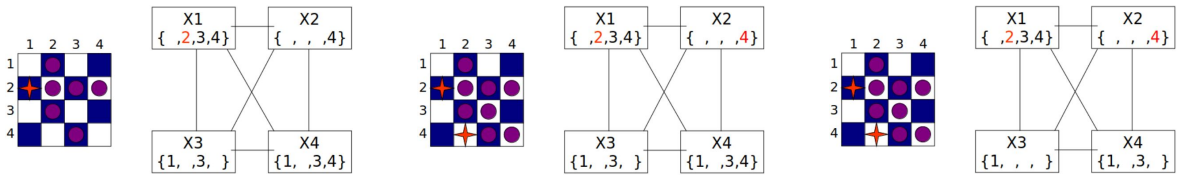
Step 4



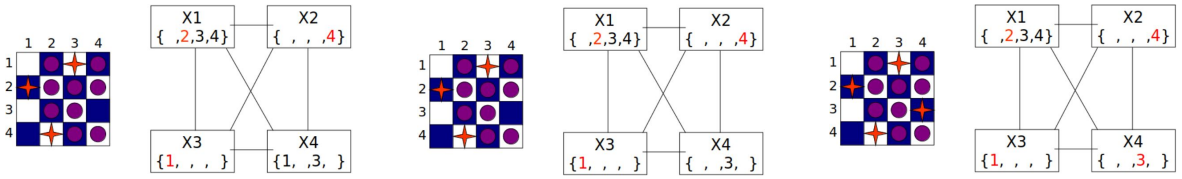
Step 5



Step 6



Step 7 Step 8 Step 9



Step 10 Step 11 Step 12

We see that at step 4, when we arrange the second queen in b2, the domain of X_3 is empty, so we have a failure. This is the moment where we have to "Backtrack" to step 2 again and move the first queen in another position, so we try for a2 (step 6). We have now one position allowed for the second queen, which is b1, then c4 for the third one and d3 for the last queen.

Remark

If we would have arranged the 2nd queen in b1, then the 3rd one would have had one position allowed, c3, and the domain of X_4 would have been empty (failure). But the algorithm wants that we backtrack as soon as we get a failure.

Standard Chronological Backtracking (SCB) It is the most common backtracking technique, which consists in:

- back up to the preceding (chronological) variable (who still has an untried value);
  - forward-propagated assignments and rightmost choices are skipped.
- try a different value for it.

A problem of this procedure is that lots of search are wasted
Now we introduce two other concepts which allow to improve the backtracking algorithm.

Example. Assume variable selection order: WA, NSW, T, NT, Q, V, SA.

- We take the **failed branch**:

	Step	*Assignment[domain]*
(1)	*pick*	$WA = r[rbg]$
(2)	*pick*	$NSW = r[rbg]$
(3)	*pick*	$T = r[rbg]$
(4)	*pick*	$NT = g[b, g]$
(5)	$\overset{fc}{\Longrightarrow}$	$Q = b[b]$
(6)	*pick*	$V = b[b, g]$
(7)	$\overset{fc}{\Longrightarrow}$	$SA = \{\}[]$

- Backtrack to (5), pick $V = g \Rightarrow$ (7) again
- Backtrack to (3), pick $NT = b \overset{fc}{\Longrightarrow} Q = g \Rightarrow$ same subtree (6)...
- Backtrack to (2), pick $T = b \Rightarrow$ same subtree (4)...

- Backtrack to (2), pick $T = g \Rightarrow$ same subtree (4)...
- Backtrack to (1), then assign NSW another value
- Lots of useless search on T and V values
- Source of inconsistency not identified: $\{WA = r, NSW = r\}$

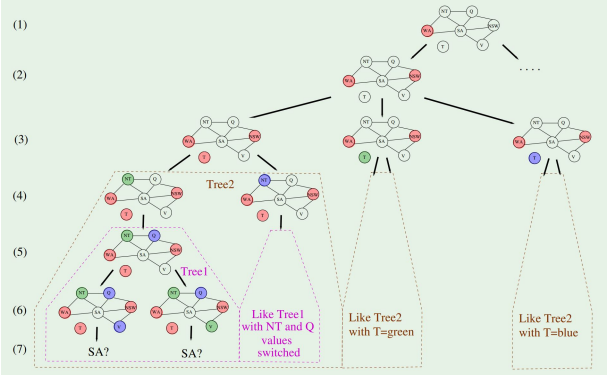


Figure 2.40: Search Tree

Nogoods and Conflict Sets

Definition. Nogood is a subassignment which cannot be part of any solution;

Definition. Conflict set for X_j (aka explanations) is a (minimal) set of value assignments which caused the reduction of D_j via forward checking (i.e., in direct conflict with some values of X_j)

Conflict-Driven Backjumping This is a backtracking technique that uses nogoods and conflict sets. The result is a more efficient algorithm, which does not waste time and memory like SCB. The main idea, when a branch fails(meaning we have an empty domain for variable X_i) is:

- identify nogood which caused the failure deterministically via forward checking;
  - take the conflict set C_i of empty-domain X_i(initial nogood)
  - backward-substitute deterministic unit assignments with their respective conflict set (until none is left)
- backtrack to the most-recently assigned element in nogood $\rightarrow$ Identify the most recent decision which caused the failure due to FC by "undoing" FC steps
- change its value;

Many different strategies and variants are available obviously.

Example. (Conflict-Driven Backjumping) Assume variable selection order: WA, NSW, T, NT, Q, V, SA.

- We take the **failed branch**:

	Step	*Assignment[domain]*	$\leftarrow \{conflictset\}$
(1)	*pick*	$WA = r[rbg]$	$\leftarrow \{\}$
(2)	*pick*	$NSW = r[rbg]$	$\leftarrow \{\}$
(3)	*pick*	$T = r[rbg]$	$\leftarrow \{\}$
(4)	*pick*	$NT = g[b, g]$	$\leftarrow \{WA = r\}$
(5)	$\overset{fc}{\Longrightarrow}$	$Q = b[b]$	$\leftarrow \{NSW = r, NT = g\}$
(6)	*pick*	$V = b[b, g]$	$\leftarrow \{WA = r\}$
(7)	$\overset{fc}{\Longrightarrow}$	$SA = \emptyset[]$	$\leftarrow \{WA = r, NT = g, Q = b\}$

- Backward-substitute assignments

$$\dfrac{\dfrac{\dfrac{\emptyset \quad (7)}{\{WA = r, NT = g, Q = b\}} \quad (5)}{\{WA = r, NT = g, NSW = r\}}}$$

- Backtrack till (3), then assign $NT = b$
- Saves useless search on V values
- We take the **new failed branch**:

	Step	*Assignment[domain]*	$\leftarrow \{conflictset\}$
(1)	*pick*	$WA = r[rbg]$	$\leftarrow \{\}$
(2)	*pick*	$NSW = r[rbg]$	$\leftarrow \{\}$
(3)	*pick*	$T = r[rbg]$	$\leftarrow \{\}$
(4)	*pick*	$NT = b[b$	$\leftarrow \{WA = r\}$
(5)	$\overset{fc}{\Longrightarrow}$	$Q = g][g]]$	$\leftarrow \{NSW = r, NT = b\}$
(6)	*pick*	$V = b[b, g]$	$\leftarrow \{WA = r\}$
(7)	$\overset{fc}{\Longrightarrow}$	$SA = \emptyset[]$	$\leftarrow \{WA = r, NT = b, Q = g\}$

- backward-substitute assignments

$$\dfrac{\dfrac{\dfrac{\emptyset \quad (7)}{\{WA = r, NT = b, Q = g\}} \quad (5)}{\{WA = r, NT = b, NSW = t\}} \quad (4)}{\{WA = r, NSW = r\}}$$

- Backtrack till (1), then assign NSW another value
- Saves useless search on T values
- Overall, saves lots of search with regard to chronological backtracking

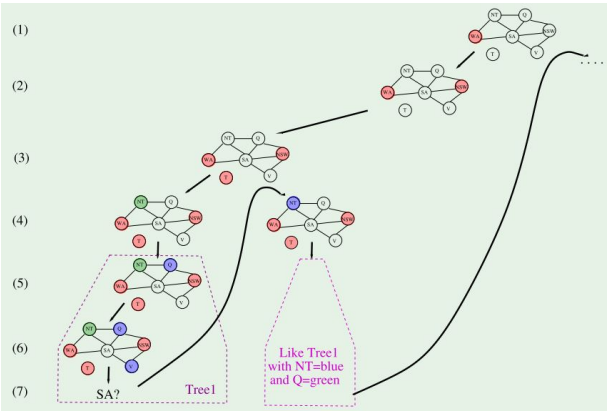


Figure 2.41: Search Tree

Learning Nogoods Nogood can be learned (stored) for future search pruning and we can add them to i.e costraints or explicit nogood list. As soon as assignment contains all but one element of a nogood, drop the value of the remaining element from variable's domain.

Example. Given nogood: $\{WA = r, NSW = r\}$ (make reference to slides 6 of the 6th Chapter), as soon as $\{NSW = r\}$ is added to assignment, red is dropped from WA domain

Nogoods can be learned either temporarily or permanently (pruning effectiveness vs. memory consumption & overhead). Many different strategies and variants are available obviously.

Local Search with CSPs Local search algorithms turn out to be very effective in solving many CSPs. They use a complete-state formulation where each state assigns a value to every variable and the search changes the value of one variable at a time. As an example, we'll use the 8-queens problem. We start on the left with a complete assignment to the 8 variables (the 8 columns, one queen per column); typically this will violate several constraints. We then randomly choose a conflicted variable, which turns out to be Q8, the rightmost column. We'd like to change the value to something that brings us closer to a solution; the most obvious approach is to select the value that results in the minimum number of conflicts with other variables, the **min-conflicts heuristic**. Eventually we can add some improvements to this strategy as i.e. random walk, simulated annealing, GAs, taboo search.

function MIN-CONFLICTS(csp, max_steps) **returns** a solution or failure
  inputs: csp, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  $current \leftarrow$ an initial complete assignment for csp
  for i = 1 to max_steps **do**
    if $current$ is a solution for csp **then return** $current$
    $var \leftarrow$ a randomly chosen conflicted variable from csp.VARIABLES
    $value \leftarrow$ the value v for var that minimizes CONFLICTS($var, v, current, csp$)
    set $var = value$ in $current$
  return *failure*

Figure 2.42: MIN − CONFLICT algorithm

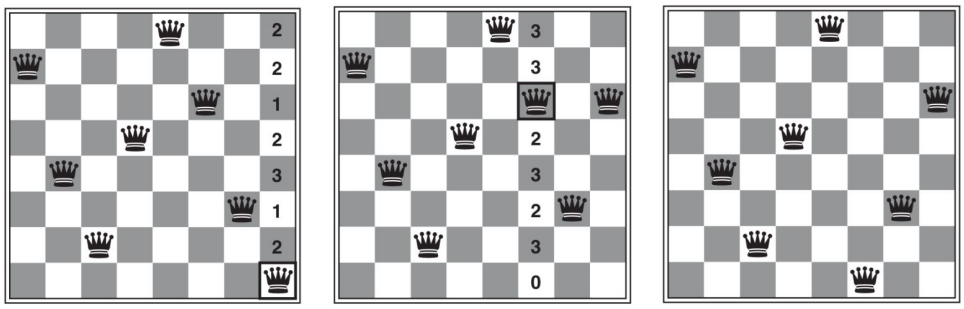


Figure 2.43: As we can see in the figure the queens in the 8th column is under attack. We have to move her to the cell with minimum conflicts which are h3 or h6. Moving her in h3 causes a failure, so our choice is reduced to h6. Now she is in conflict with the queen in f6. We repeat the reasoning for that queen and we finally find a solution to the problem.

Partitioning CSPs "Divide & Conquer" CSPs. Means partition a CSP into independent CSPs. This technique is going to be useful when identify strongly-connected components in constraint graph. Let's think about Australia. It is a country divided into 6 regions, plus an island Tasmania. It is normal to think treating Australia and Tasmania as independent sub problems.

Solving Tree-structured CSPs

Theorem. If the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time in worst case. General CSPs can be solved in $O(d^n)$

Algorithm

1. Choose a variable as root
2. Order variables from root to leaves
3. For $j \in \{n, \dots, 2\}$ apply MAKEARCCONSISTENT(PARENT(X_j), X_j)
4. For $j \in \{2, \dots, n\}$, assign X_j consistently with PARENT(X_j)

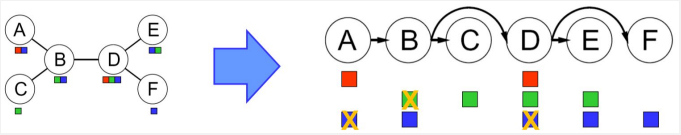


Figure 2.44

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or failure
inputs: *csp*, a CSP with components X, D, C

$n \leftarrow$ number of variables in X
assignment $\leftarrow$ an empty assignment
root $\leftarrow$ any variable in X
$X \leftarrow$ TOPOLOGICALSORT(X, *root*)
for j = n **down to** 2 **do**
MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
if it cannot be made consistent **then return** *failure*
for i = 1 **to** n **do**
assignment[X_i] $\leftarrow$ any consistent value from D_i
if there is no consistent value **then return** *failure*
return *assignment*

Figure 2.45: TREE-CSP-SOLVER algorithm

For **Solving Nearly Tree-Structured CSPs** we have **Cuteset Conditioning**:

1. Identify a (small) cycle cutset S: a set of variables s.t. the remaining constraint graph is a tree;
2. For each possible consistent assignment to the variables in S:
   (a) Remove from the domains of the remaining variables any values that are inconsistent with the assignment for S
   (b) apply the tree-structured CSP algorithm
3. If $c \triangleq |S|$, then runtime is $O(d^c \cdot (n-c)^2)$, which is much smaller than d^n if c small;

Example. We take now in consideration the following example regarding the algorithm for solving nearly tree-structured CSPs:

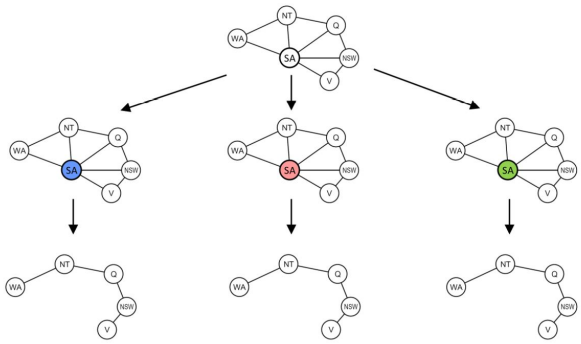


Figure 2.46: Cutset conditioning Example

Breaking Value Symmetry So far, we have looked at the structure of the constraint graph. There can also be important structure in the values of variables, or in the structure of the constraint relations themselves. Consider the map-coloring problem with d colors. For every consistent solution, there is actually a set of $d!$ solutions formed by permuting the color names. For example, on the Australia map (see Figure 2.36) we know that WA, NT, and SA must all have different colors, but there are $3! = 6$ ways to assign three colors to three regions. This is called value symmetry. We would like to reduce the search space by a factor of d! by breaking the symmetry in assignments. We do this by introducing a symmetry-breaking constraint. For our example, we might impose an arbitrary ordering constraint, NT<SA<WA, that requires the three values to be in alphabetical order. This constraint ensures that only one of the d! solutions is possible: $NT = blue$, $SA = green$, $WA = red$. For map coloring, it was easy to find a constraint that eliminates the symmetry. In general it is NP-hard to eliminate all symmetry, but breaking value symmetry has proved to be important and effective on a wide range of problems.

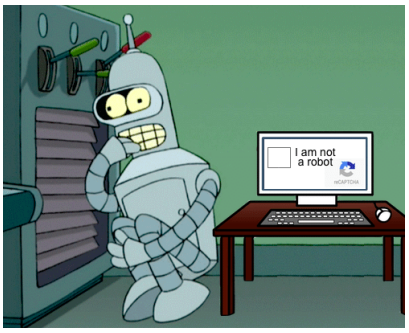
Chapter 3

Knowledge, Reasoning and Planning

3.1 Logical Agents

Thomas De Min

Propositional logic illustrates all the basic concepts of logic.



Not a robot

Figure 3.1: A logic agent

Syntax The syntax of propositional logic defines the allowable sentences. A **Propositional formula** (or Boolean formula or sentence) is:

- Both $\top, \bot$ (i.e. True, False);
- A propositional atom A_1, A_2, A_3[1];
- If φ_1 and φ_2 are formulas, then: $\neg\varphi_1, \varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \rightarrow \varphi_2, \varphi_1 \leftarrow \varphi_2, \varphi_1 \leftrightarrow \varphi_2, \varphi_1 \oplus \varphi_2$ are formulas. More precisely these are **Complex sentences**, which are constructed from simpler ones.

The function $Atoms(\varphi)$ returns the set $\{A_1, ..., A_N\}$ of atoms occurring in φ. For example if φ contains $A \wedge B$, then $Atoms(\varphi) = \{A, B\}$. A **Literal** is a propositional atom A_i (**positive literal**) or its negation $\neg A_i$ (**negative literal**).

Definition. If $l := \neg A_i$, then $\neg l := A_i$.

Definition. A **Clause** is a disjunction of literals, $\bigvee_j l_j$ (e.g. $A_1 \vee A_2 \vee A_3 \vee ...$).

Definition. A **Cube** instead is a conjunction of literals, $\bigwedge_j l_j$ (e.g. $A_1 \wedge A_2 \wedge A_3 \wedge ...$).

[1] Roughly speaking a variable.

$$
\begin{aligned}
\neg\neg\alpha &\iff \alpha \\
(\alpha \vee \beta) &\iff \neg(\neg\alpha \wedge \neg\beta) \\
\neg(\alpha \vee \beta) &\iff (\neg\alpha \wedge \neg\beta) \\
(\alpha \wedge \beta) &\iff \neg(\neg\alpha \vee \neg\beta) \\
\neg(\alpha \wedge \beta) &\iff (\neg\alpha \vee \neg\beta) \\
(\alpha \to \beta) &\iff (\neg\alpha \vee \beta) \\
\neg(\alpha \to \beta) &\iff (\alpha \wedge \neg\beta) \\
(\alpha \leftrightarrow \beta) &\iff ((\alpha \to \beta) \wedge (\alpha \leftarrow \beta)) \\
\neg(\alpha \leftrightarrow \beta) &\iff ((\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta)) \\
(\alpha \oplus \beta) &\iff \neg(\alpha \leftrightarrow \beta)
\end{aligned}
$$

Boolean logic can be expressed in terms of $\{\neg, \wedge\}$ (of $\{\neg, \vee\}$) only.

Semantics The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In PL, a model simply fixes the **truth value** for every proposition symbol. For atomic sentences is straightforward. For complex ones, we have five rules (Figure 3.2):

- $\neg P$ is true iff P is false in m [2];
- $P \wedge Q$ is true iff both P and Q are true in m;
- $P \vee Q$ is true iff either P or Q is true in m;
- $P \Rightarrow Q$ is true unless P is true and Q is false in m;
- $P \Leftrightarrow Q$ is true iff P and Q are both true or both false in m.
- $P \oplus Q$ is true iff P and Q yields different truth values.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 3.2: Truth tables for the five logical connectives.

Note

$\wedge, \vee, \leftrightarrow$[a] and $\oplus$ are commutative and associative.

[a] The eagle-eyed reader (cit. AIMA) may have noticed the use of both $\leftrightarrow$ and $\Leftrightarrow$, these two have the same meaning. The same holds for $\to, \leftarrow$.

Remark

With $\alpha \to \beta$ we implicitly say that "the **antecedent** α implies the **consequent** β", but not vice versa. **It does not require causation or relevance between α and β.** For example, "*5 is odd implies Tokyo is the capital of Japan*", "*5 is even implies Sam is smart*" and "*5 is even implies Tokyo is in Italy*" are all true.

[2] m is the considered model, we will see it later.

Tree and DAG Representations of Formulas Formulas can be represented either as trees or as DAGs. However, DAG representation can be up to exponentially smaller, in particular when $\leftrightarrow$s are involved.

Basic Definitions and Notation

Definition. A **Total truth assignment** represents a possible world or a possible state of the world. A Total truth assignment μ for φ is defined as:

$$\mu : Atoms(\varphi) \mapsto \{\top, \bot\}$$

Definition. A **Partial Truth assignment** represents 2^k total assignments, k is the number of unassigned variables. A Partial truth assignment μ for φ is defined as

$$\mu : \mathcal{A} \mapsto \{\top, \bot\}, \mathcal{A} \subset Atoms(\varphi)$$

Notation

Set and formula representations of an assignment

- μ can be represented as a set of literals: $\{A_1, \neg A_2\}$
- μ can be represented as a formula (cube): $(A_1 \wedge \neg A_2)$

Model

Definition. A **model** is a possible world.

Example. The semantics for arithmetic specifies that the sentence "$x + y = 4$" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1.

If a sentence φ is true in model μ, we say that μ **satisfies** φ or sometimes μ **is a model of** φ. $M(\varphi) \stackrel{def}{=} \{\mu | \mu \models \varphi\}$ is the set of all models of φ. A partial truth assignment μ satisfies φ iff all its total extensions satisfy φ.

Example. $\{A_1\} \models (A_1 \vee A_2)$ because $\{A_1, A_2\} \models (A_1 \vee A_2)$ and $\{A_1, \neg A_2\} \models (A_1 \vee A_2)$.

φ is satisfiable iff $\mu \models \varphi$ for some μ ($M(\varphi) \neq \emptyset$). Finally, sentence α **entails** sentence β ($\alpha \models \beta$) iff, for all μs, $\mu \models \alpha \Longrightarrow \mu \models \beta$, i.e. $M(\alpha) \subseteq M(\beta)$. φ is **valid** ($\models \varphi$) iff $\mu \models \varphi$ forall μs, i.e. $\mu \in M(\varphi)$ forall μs.

Properties and Results:

- **Property**: φ is **valid** iff $\neg\varphi$ is **unsatisfiable**.
- **Deduction Theorem**: $\alpha \models \beta$ iff $\alpha \rightarrow \beta$ is valid ($\models \alpha \rightarrow \beta$)
- **Corollary**: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ is unsatisfiable.

Validity and entailment checking can be straightforwardly reduced to (un)**satisfiability checking**.

Complexity For N variables, there are up to 2^N truth assignments to be checked. The problem of deciding the satisfiability of a propositional formula is $\mathbb{NP}$-complete. The most important logical problems (validity, inference, entailment, equivalence, ...) can be straightforwardly reduced to (un)satisfiability, and are thus (co)$\mathbb{NP}$-complete. That means no worst-case-polynomial algorithm exists.

Conjunctive Normal Form (CNF) φ is in CNF iff it is a conjunction of disjunctions of literals:

$$\bigwedge_{i=1}^{L} \bigvee_{j_i=1}^{K_i} l_{j_i} \tag{3.1}$$

the disjunctions of literals $\bigvee_{j_i=1}^{K_i} l_{j_i}$ are called **clauses**. It is easier to handle because it represents a list of lists of literals. Thus, no reasoning on the recursive structure of the formula must be done.

Every φ can be reduced into CNF by[3]:

1. expanding implications and equivalences:

$$\alpha \to \beta \Longrightarrow \neg\alpha \vee \beta$$
$$\alpha \leftrightarrow \beta \Longrightarrow (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$$

2. pushing down negations recursively:

$$\neg(\alpha \wedge \beta) \Longrightarrow \neg\alpha \vee \neg\beta$$
$$\neg(\alpha \vee \beta) \Longrightarrow \neg\alpha \wedge \neg\beta$$
$$\neg\neg\alpha \Longrightarrow \alpha$$

3. applying recursively the DeMorgan's Rule:

$$(\alpha \wedge \beta) \vee \gamma \Longrightarrow (\alpha \vee \gamma) \wedge (\beta \vee \gamma) \quad \text{Associative property}$$

In the worst case scenario, the resulting formula is **exponential** w.r.t. the initial one. $Atoms(CNF(\varphi)) = Atoms(\varphi)$. $CNF(\varphi)$ is **equivalent** to $\varphi : M(CNF(\varphi)) = M(\varphi)$.

Due to the exponential increase of length in the resulting formula, the classic conversion to CNF is rarely used in practice. A much more used conversion is the **Labeling CNF conversion** $CNF_{label}(\varphi)$ (also known as Tseitin's conversion). Every φ can be reduced into CNF by applying recursively bottom-up the rules:

$$\varphi \Longrightarrow \varphi[(l_i \vee l_j)|B] \wedge CNF(B \leftrightarrow (l_i \vee l_j))$$
$$\varphi \Longrightarrow \varphi[(l_i \wedge l_j)|B] \wedge CNF(B \leftrightarrow (l_i \wedge l_j))$$
$$\varphi \Longrightarrow \varphi[(l_i \leftrightarrow l_j)|B] \wedge CNF(B \leftrightarrow (l_i \leftrightarrow l_j))$$

l_i, l_j being literals and B being a "new" variable[4]. In the worst-case it is **linear**. $Atoms(CNF_{label}(\varphi)) \supseteq Atoms(\varphi)$. $CNF_{label}(\varphi)$ is equi-satisfiable w.r.t. φ: $M(CNF(\varphi)) \neq \emptyset$ iff $M(\varphi) \neq \emptyset$.

Example. Consider the following formula ϕ.

$$\phi := ((p \vee q) \wedge r) \to (\neg s)$$

Consider all subformulas (excluding simple variables):

$$\neg s$$
$$p \vee q$$
$$(p \vee q) \wedge r$$
$$((p \vee q) \wedge r) \to (\neg s)$$

Introduce a new variable for each subformula (here I use x instead of B for coherence with the example):

$$x_1 \leftrightarrow \neg s$$
$$x_2 \leftrightarrow p \vee q$$
$$x_3 \leftrightarrow x_2 \wedge r$$
$$x_4 \leftrightarrow x_3 \to x_1$$

Conjunct all substitutions and the substitution for ϕ:

$$T(\phi) := x_4 \wedge (x_4 \leftrightarrow x_3 \to x_1) \wedge (x_3 \leftrightarrow x_2 \wedge r) \wedge (x_2 \leftrightarrow p \vee q) \wedge (x_1 \leftrightarrow \neg s)$$

All substitutions can be transformed into CNF, e.g.

$$\begin{aligned} x_2 \leftrightarrow p \vee q &\equiv (x_2 \to (p \vee q)) \wedge ((p \vee q) \to x_2) \\ &\equiv ... \\ &\equiv (\neg x_2 \vee p \vee q) \wedge (\neg p \vee x_2) \wedge (\neg q \vee x_2) \end{aligned}$$

[3] I think this count as pseudocode or at least it is highly probable that rseba may ask this procedure (also for FOL).
[4] The | stands for substitution

Propositional Reasoning Automated Reasoning in PL is a fundamental task. Let KB be a set of sentences (or a sentence that asserts all the individual sentence) called **Knowledge Base**. $KB \models \alpha$: entail fact α from KB, also known as **Model Checking**: $M(KB) \subseteq M(\alpha)$. Model checking enumerates all possible models to check that α is true in all models in which KB is true. Typically $||KB|| \gg ||\alpha||$. All propositional reasoning tasks reduced to satisfiability (SAT). A sentence is satisfiable if it is true in, or satisfied by, some model. Validity and satisfiability are connected: α is valid iff $\neg\alpha$ is unsatisfiable; contrapositively, α is satisfiable iff $\neg\alpha$ is not valid. Then $KB \models \alpha \implies SAT(KB \wedge \neg\alpha)$ =false.

Resolution Rule Resolution is the deduction of a new clause from a pair of clauses with exactly one incompatible variable (**resolvent**):

$$\frac{(\overbrace{l_1 \vee ... \vee l_k}^{common} \vee \overbrace{l}^{resolvent} \vee \overbrace{l'_{k+1} \vee ... \vee l'_m}^{C'}) \qquad (\overbrace{l_1 \vee ... \vee l_k}^{common} \vee \overbrace{\neg l}^{resolvent} \vee \overbrace{l''_{k+1} \vee ... \vee l''_n}^{C''})}{(\underbrace{l_1 \vee ... \vee l_k}_{common} \vee \underbrace{l'_{k+1} \vee ... \vee l'_m}_{C'} \vee \underbrace{l''_{k+1} \vee ... \vee l''_n}_{C''})} \tag{3.2}$$

Example.

$$\frac{(A \vee B \vee C \vee D \vee E) \qquad (A \vee B \vee \neg C \vee F)}{(A \vee B \vee D \vee E \vee F)}$$

We have to search for a refutation of φ; in other words, is φ unsatisfiable? Basic idea: apply iteratively the resolution rule to pairs of clauses with a conflicting literal, producing novel clauses, until either:

- a false clause is generated, or
- the resolution rule is no more applicable

It is **Correct** if it returns an empty clause, then φ unsat ($\alpha \models \beta$). **Complete** if φ unsat ($\alpha \models \beta$), then it returns an empty clause.
Unfortunately, this resolution procedure is time-inefficient and memory inefficient (exponential in memory).

Note

When the two clauses contain more than one pair of complementary literals, the resolution rule can be applied (independently) for each such pair; however, the result is always a **tautology**. Moreover, it is **ILLEGAL** to apply the resolution rule to all complementary literal at once.

Example.

$$\frac{(A \vee \neg B \vee \neg C \vee D) \qquad (A \vee B \vee C \vee E)}{\text{Both } (A \vee \neg C \vee C \vee D \vee E) \text{ and } (A \vee \neg B \vee B \vee D \vee E) \text{ are correct, } (A \vee D \vee E) \text{ is illegal.}}$$

function PL-RESOLUTION(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  clauses $\leftarrow$ the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new $\leftarrow$ { }
  loop do
    for each pair of clauses C_i, C_j **in** *clauses* **do**
      resolvents $\leftarrow$ PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new $\leftarrow$ *new* $\cup$ *resolvents*
    if *new* $\subseteq$ *clauses* **then return** *false*
    clauses $\leftarrow$ *clauses* $\cup$ *new*

Figure 3.3: A simple resolution algorithm for PL. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

Improvements Alternative "set" notation (Γ clause set)[5]:

$$\frac{\Gamma, \phi_1, ..., \phi_n}{\Gamma, \phi'_1, ..., \phi'_n}$$

Improvements:

- **Clause Subsumption** (C clause):

$$\frac{\Gamma \wedge C \wedge (C \vee \bigvee_i l_i)}{\Gamma \wedge (C)}$$

- **Unit Resolution**:

$$\frac{\Gamma \wedge (l) \wedge (\neg l \vee \bigvee_i l_i)}{\Gamma \wedge (l) \wedge (\bigvee_i l_i)}$$

- **Unit Subsumption**:

$$\frac{\Gamma \wedge (l) \wedge (l \vee \bigvee_i l_i)}{\Gamma \wedge (l)}$$

- **Unit Propagation** = Unit Resolution + Unit Subsumption

"Deterministic" rule applied before other "non-deterministic" rules

DPLL The Davis-Putnam-Longemann-Loveland procedure tries to build an assignment μ that satisfies φ. At each step assigns a truth value to all instances of one atom. Performs deterministic choices (mostly unit-propagation) first. It is correct and complete and it requires polynomial space (Figure 3.4 and Figure 3.5).

Procedural note

When you get two complementary unit clauses on the same literal (e.g. $(\neg F) \wedge ... \wedge (F)$), the rule is to apply unit propagation on one of them, choosing it following the convention decided a priori (e.g. alphabetical order and true first), and then recursively apply DPLL on the resulting formula. As a consequence there will be an empty clause and the branch will be false. In this case applying a decision step (branching) is an **error**. This procedure can be seen applied in Figure 3.5 on the variable F in the leftmost branch.

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(*clauses*, *symbols*, { })

---

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

  if every clause in *clauses* is true in *model* **then return** *true*
  if some clause in *clauses* is false in *model* **then return** *false*
  P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)
  if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*})
  P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
  if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*})
  P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
  return DPLL(*clauses*, *rest*, *model* ∪ {P=*true*}) **or**
      DPLL(*clauses*, *rest*, *model* ∪ {P=*false*}))

Figure 3.4: The DPLL algorithm for checking satisfiability of a sentence in PL. Pure-Symbol rule is out of date, no more used in modern solvers. A pure symbol is a symbol that always appears with the same "sign" in all clauses (e.g. $(A \vee \neg B), (\neg B \vee C), (A \vee C)$).

[5]This paragraph is not contained in the third edition of AIMA

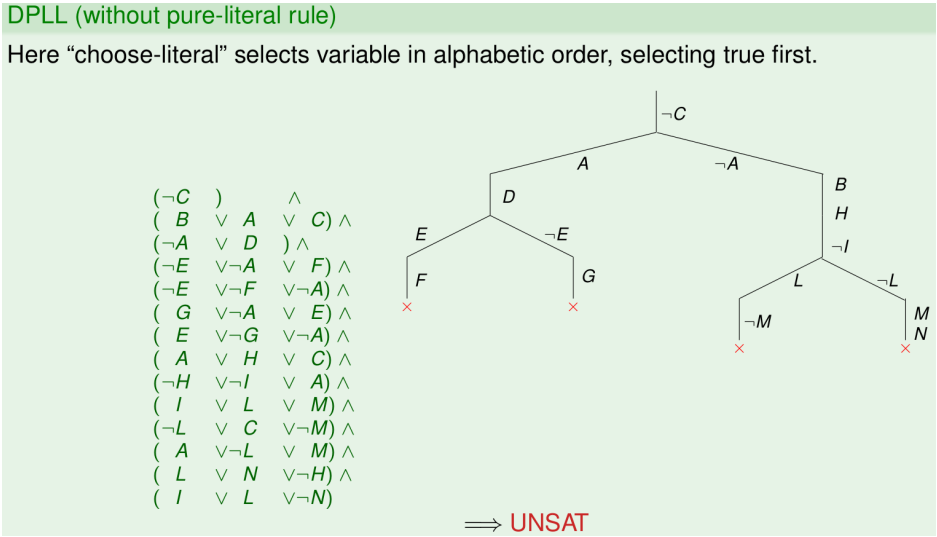


Figure 3.5: Example of DPLL execution.

Horn clauses and definite clauses Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables to use a more restricted and efficient inference algorithm. One such restricted form is the **Definite** clause, which is a disjunction of literals where exactly one is positive (e.g. $(A \vee \neg B \vee \neg C)$). A slightly more general is the **Horn** clause, which is a disjunction of literals where at most one is positive. Clauses with no positive literals are called **Goal** clauses. Therefore, Definite clauses and Goal clauses are Horn Clauses. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause.

Checking the satisfiability of Horn formulas requires polynomial time (Figure 3.6):

1. Eliminate unit clauses by propagating their value;

2. If an empty clause is generated, return unsat

3. Otherwise, every clause contains at least one negative literal. Then assign all variables to $\bot$ and return the assignment

Alternatively, run DPLL/CDCL[6], selecting negative literals first.

```
function Horn_SAT(formula ϕ, assignment & μ) {
    Unit_Propagate(ϕ, μ);
    if (ϕ == False) then
        return UNSAT;
    else {
        μ := μ U Union(A_i not in μ, {¬A_i});
        return SAT;
}   }

function Unit_Propagate(formula & ϕ, assignment & μ){
    while (ϕ != True and ϕ != False and {a unit clause (l) occurs in ϕ}) do {
        ϕ = assign(ϕ, l);
        μ := μ U {l};
}   }
```

Figure 3.6: A simple polynomial procedure for Horn-SAT.

[6]Modern SAT solver, I skipped the slide since it seems more of a summary of features.

Example. Check satisfiability of this Horn formula.

$$\begin{aligned} &\neg A_1 \vee A_2 \vee \neg A_3 \\ &A_1 \vee \neg A_3 \vee \neg A_4 \\ &\neg A_2 \vee \neg A_4 \\ &A_3 \vee \neg A_4 \\ &A_4 \end{aligned}$$

We first assign $A_4 := \top$. The previous step left us A_3 as unit clause, we assign $A_3 := \top$. $A_2 :=\bot$. But then, all it remains are two occurrences of A_1, one positive and one negative, thus it is unsat.

Local Search with SAT It can be applied directly to SAT problems provided that we choose the right evaluation function. The evaluation function counts the number of unsatisfied clauses (**Cost**). In fact this is exactly the same measure used by Local Search for CSPs. It takes as input a set of clauses and it uses total truth assignment on those. Basically it allows states with unsatisfied clauses. Two neighbour states differ for one variable truth value.

There exists many variants: **Stochastic local search** can also be applied to SAT as well; **WalkSAT** is a particular instantiation of stochastic local search (Figure 3.7): It randomly selects an unsatisfied clause C and with probability p flips variable from C at random, with probability $1 - p$ flip variable from C causing a minimum number of unsat clauses. It can only detect satisfiability but not unsatisfiability.

function WALKSAT(*clauses*, p, *max_flips*) **returns** a satisfying model or *failure*
  inputs: *clauses*, a set of clauses in propositional logic
    p, the probability of choosing to do a "random walk" move, typically around 0.5
    max_flips, number of flips allowed before giving up

  model ← a random assignment of *true*/*false* to the symbols in *clauses*
  for $i = 1$ **to** *max_flips* **do**
    if *model* satisfies *clauses* **then return** *model*
    clause ← a randomly selected clause from *clauses* that is false in *model*
    with probability p flip the value in *model* of a randomly selected symbol from *clause*
    else flip whichever symbol in *clause* maximizes the number of satisfied clauses
  return *failure*

Figure 3.7: The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables

Agents Based on Knowledge Representation and Reasoning Knowledge Representation and Reasoning (**KR&R**) is the field of AI dedicated to representing knowledge of the world in a form a computer system can utilize to solve complete tasks. The class of systems (agents) that derive from this approach are called **knowledge based (KB) systems (agents)**.
A KB agent maintains a knowledge base (KB) of facts which is a collection of domain-specific facts believed by the agent. Those are expressed in a formal language (in this particular case PL). The KB represents the agent's representation of the world. Initially it contains the background knowledge, but it can be updated through to the agent's percepts in order for it to ask queries. Updates and queries are performed by an **inference engine** which allows for inferring actions and new knowledge.

Reasoning is the formal manipulation of the symbols representing a collection of beliefs to produce representations of new ones, the fundamental operation is the **Logical entailment** ($KB \models \alpha$).

Example. Prescription of medication to patient with allergies.

- (KB acquired fact): "*Patient x is allergic to medication m.*"
- (KB general rule): "*Anybody allergic to m is also allergic to m'.*"
- (KB general rule): "*If x is allergic to m', do not prescribe m' for x.*"

- (query): "*Prescribe m' for x?*"
- (answer): No (because patient x is allergic to medication m').

Finally, a **Logic agent** combine domain knowledge with current percepts to infer hidden aspects of current state prior to selecting actions (crucial in partially observable environments). The KB Agent must be able to represent states and actions, incorporate new percepts, update its internal representation of the world, deduce hidden properties of the world and deduce appropriate actions.

An agent can be described at different levels:

- **Knowledge level** (declarative approach): the behaviour is completely described by the sentences stored in the KB;
- **Implementation level** (procedural approach): the behaviour is described as program code.

In the declarative approach, we **Tell** the KB what it needs to know and **Ask** what to do. At each percept, the agent tells the KB of the percept at time step t, it asks the KB for the best action to do at time step t and, in the end, it tells the KB that it has in fact taken that action. These three steps, are represented in Figure 3.8 through functions MAKE-PERCEPT-SENTENCE, MAKE-ACTION-QUERY, MAKE-ACTION-SENTENCE.

function KB-AGENT(*percept*) **returns** an *action*
  persistent: KB, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE($percept, t$))
  $action \leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE($action, t$))
  $t \leftarrow t + 1$
  return *action*

Figure 3.8: Tell and Ask may require complex logical inference.

Propositional Logic Agents PL Agents are another kind of logic agents, they use propositional logic to represent the KB as a set of propositional formula. Percepts and actions are collections of propositional atoms, which in practice are sets of clauses. The propositional logic inference is done with incremental calls to a SAT solver.
Reasoning Process (propositional entailment) is sound, which means that if the KB is true in the real world, then any sentence α derived from KB by a sound inference procedure is also true in the real world. Sentences are configurations of the agent. With reasoning the agent constructs new configurations from old ones, thus the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

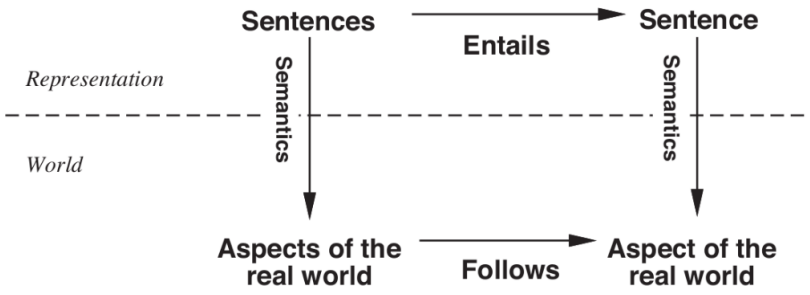


Figure 3.9: Propositional logic agent.

3.2 First Order Logic

Sara Miotto

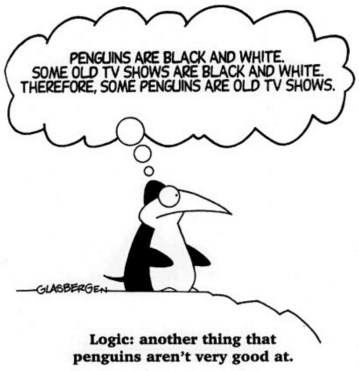


Figure 3.10: Cringe

Representation Revisited First of all, let's start talking about **logic** in general. A logic is a triple characterized by:

- **Language**: a class of sentences described by a formal grammar;
- **Inference system**: is a set of formal derivation rules over language;
- **Semantics**: a formal specification of how to assign meaning in the real world to the elements of language.

There are different **types** of logics:

- **Propositional logic (PL)**: a formal (non-ambiguous), declarative (knowledge and inference are separate), compositional (the meaning of $(A \wedge B) \Rightarrow C$ derives from the meaning of A, B, C) language. It allows for partial, disjunctive and negated information. The meaning of the sentences is context independent. However, there are some cons of it, such as the fact that it is based on atomic events, it has a very limited expressive power and it assumes that the world contains only facts;
- **First-Order logic (FOL)**: set of objects and an interpretation that maps constant symbols to objects, function symbols to functions on those objects, and predicate symbols to relations;
- **Modal Logic (MLs)**;
- **Description logics (DLs)**;
- **Temporal logics (TLs)**: facts hold at particular times and that those times are ordered;
- **Fuzzy logics**: propositions have a degree of truth between 0 and 1;
- **Probabilistic logics**: propositions can have any degree of belief ranging from 0 to 1;
- **Higher-order logic**: it views the relations and functions referred to by first-order logic as objects in themselves.

Two important features of logic are the so-called **Ontological Commitment** (what exists in the world) and the **Epistemological Commitment** (what the agent believes about facts).

Syntax and Semantics of First-Order Logic Let's focus now on our main topic, **First-Order Logic**. First-Order Logic is **structured**: a world/state includes **objects**, each of which may have attributes of its own as well as *relationships* to other objects.

Definition. A **relation** is a set of tuples of objects, and a tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.

Definition. Models are the formal structures that constitute the world we are analyzing and here they require total functions (there must be a total functions value for every input tuple). They are composed by the domain and the interpretation.

Definition. The **domain** is the set of objects, which has to be nonempty. The focus is not on the type of objects, yet on their number.

Definition. The **interpretation** is a map on elements of the signature:

- variables ⇒ domain elements: a variable x is mapped into a a particular object ;
- constant symbols ⇒ domain elements: a constant symbol C is mapped into a particular object;
- predicate symbols ⇒ domain relations: a k-ary predicate $P(\dots)$ is mapped into a subset;
- function symbols ⇒ domain functions: a k-ary function f is mapped into a domain function;
- terms: are mapped into the value $[f(t_1, \dots, t_k)]$ returned by applying the domain function to the values $[t_1]^I, ..., [t_k]^I$ obtained by applying recursively I to the terms $t_1, \dots, t_k$: $[f(t_1, \dots, t_k)]^I = [f]^I([t_1]^I, \dots, [t_k]^I)$.

Every model must provide the information required to determine if any given sentence is true or false. An **atomic formula** $P(t_1, \dots, t_k)$ is true in I if the objects into which the terms $t_1, \dots, t_k$ are mapped by I comply to the relation into which P is mapped $[P(t_1, \dots, t_k)]^I$ is true if $\langle [t_1]^I, \dots, [t_k]^I \rangle \in [P]^I$. An atomic formula $t_1 = t_2$ is true in I if the terms t_1, t_2 are mapped into the same domain element: $[t_1 = t_2]^I$ is true if $[t_1]^I$ same as $[t_2]^I$. (**Remember** that an atomic formula is a formula with no deeper propositional structure, that is, a formula that contains no logical connectives or equivalently a formula that has no strict subformulas).

Definition. Functions are a particular type of relationship where a given object must be related to exactly one object.

The basic syntactic elements of first-order logic are **symbols**, **relations**, **functions**, **propositional connectives**, **equality**, **quantifiers** and **punctuation symbols**.

Speaking of symbols, there exist different types:

- **constant symbols**: objects, $0 - ary$ function symbols;
- **predicate symbols**: relations;
- **function symbols**: functions;
- **variable symbols**.

Remember to make them beginning with an uppercase letter. And notice that not all the objects have a name.

Definition. A **term** is a logical expression that refers to an object. They can be constant, variable or function. A term with no variables is called a **ground term**.

Definition. A **complex term** (or formula) is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. A formula is closed when all variables occurring in it are quantified. Ground formulas are closed, but not vice versa.

Let's say something regarding the *polarity* of subformulas.

Definition. The **polarity** of a subformula is the number of nested negations modulo 2.

Positive/negative occurrences:

- φ occurs positively in φ;
- if $\neg\varphi_1$ occurs positively [negatively] in φ, then φ_1 occurs negatively [positively] in φ;
- if $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$ occur positively [negatively] in φ, then φ_1 and φ_2 occur positively [negatively] in φ;

- if $\varphi_1 \Rightarrow \varphi_2$ occurs positively [negatively] in φ, then φ_1 occurs negatively [positively] in φ and ϕ_2 occurs positively [negatively] in φ;
- if $\varphi_1 \Leftrightarrow \varphi_2$ or $\varphi_1 \oplus \varphi_2$ occurs in φ, then φ_1 and φ_2 occur positively and negatively in φ;
- if $\forall x\ \varphi_1$ or $\exists x\ \varphi_1$ occurs positively [negatively] in φ, then φ_1 occurs positively [negatively] in φ

We can **combine** objects and relations in order to make atomic sentences that state facts. An atomic sentence is composed by a predicate symbol followed by a parenthesized list of terms and it is proved to be true if the relation referred to by the predicate symbol holds among the objects referred to by the arguments. And so, it is written as: $\top, \bot$,proposition or predicate$(term_1, \ldots, term_n)$ or $term_1 = term_2$.

Complex sentences are constructed combining atomic sentences using **logical connectives**. They are then written as: $\neg\alpha, \alpha \wedge \beta, \alpha \vee \beta, \alpha \Rightarrow \beta, \alpha \Leftrightarrow \beta, \alpha \oplus \beta, \forall x\ \alpha, \exists x\ \alpha$.

Quantifiers are used in order to be able to express properties of the entire collection of objects we are referring to. There are two different quantifiers: the universal and the existential one.

Universal quantifier $\forall x$, where x is a variable (variables are lowercase letters). $\forall x.\alpha(x, \cdot)$ true in M if α is true in M for every possible domain value x is mapped to. Roughly speaking, can be seen as a conjunction over all possible instantiations of x in α.

One may want to restrict the domain of universal quantification to elements of some kind. A possible solution for this is using an implication, with restrictive predicate as implicant. Beware of not using $\wedge$ instead of $\Rightarrow$. Also $\forall$ distributes with $\wedge$ but not with $\vee$.

Existential quantifier $\exists x$ allows us to make statements about some object without naming it. $\exists x.\alpha(x, \cdot)$ true in M if α is true in M for some possible domain value x is mapped to. Roughly speaking, can be seen as a disjunction over all possible instantiations of x in α.

One may want to restrict the domain of existential quantification to elements of some kind, so a solution is using a conjunction with restrictive predicate.

Do not use $\Rightarrow$ instead of $\wedge$. $\exists$ distributes with $\vee$ but not with $\wedge$.

Nested quantifiers We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. Consecutive quantifiers of the same type can be written as one quantifier with several variables.

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. The rule is that the variable belongs to the innermost quantifier that mentions it; then it will not be subject to any other quantification.

Connections between $\forall$ and $\exists$: the two quantifiers are connected by negation. Because $\forall$ is really a conjunction over the universe of objects and $\exists$ is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

- $\neg\ \exists$x P $\equiv \forall$x $\neg$ P;
- $\neg$ (P $\vee$Q) $\equiv \neg$ P $\wedge \neg$ Q;
- $\neg\ \forall$x P $\equiv \exists$x $\neg$ P;
- $\forall$x P $\equiv \neg\ \exists$x $\neg$P;

- $\exists$x P $\equiv \neg\, \forall$x $\neg$ P;
- $\neg$ (P$\wedge$Q) $\equiv \neg$ P $\vee \neg$ Q;
- (P$\wedge$Q) $\equiv \neg$ ($\neg$ P $\vee \neg$ Q);
- P $\vee$ Q $\equiv \neg$ ($\neg$ P $\wedge \neg$ Q).

Other properties:

- If x does not occur in ϕ, $\forall x.\phi$ equivalent to $\exists x.\phi$ equivalent to ϕ;
- $\forall xy.P(x,y)$ equivalent to $\forall yx.P(x,y)$;
- $\exists xy.P(x,y)$ equivalent to $\exists yx.P(x,y)$;
- $\exists x \forall y.P(x,y)$ not equivalent to $\forall y \exists x.P(x,y)$;
- Negated restricted quantifiers switch $\Rightarrow$ with $\wedge$:
  - $\forall x.(P(x) \Rightarrow \alpha) \Leftrightarrow \neg\exists x.(P(x) \wedge \neg\alpha)$;
  - $\neg\forall x.(P(x) \Rightarrow \alpha) \Leftrightarrow \exists x.(P(x) \wedge \neg\alpha)$.

Another possible way to build atomic sentences is using the **equality symbol**. It means that two terms refer to the same object. Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object. The equality symbol can be used to state facts about a given function. It can also be used with negation to insist that two terms are not the same object.

Satisfiability, Validity, Entailment There are some important assertions to make:

- A model $M = \langle D, I \rangle$ satisfies $\varphi (M \models \varphi)$ iff $[\varphi]^I$ is true;
- $M(\varphi) = \{M | M models \varphi\}$ (the set of models of φ);
- φ is satisfiable iff $M \models \varphi$ for some M (i.e. $M(\varphi) \neq 0$);
- α entails $\beta (\alpha \models \beta)$ iff, for all $M, M \models \alpha \longrightarrow M \models \beta$;
- φ is valid ($\models \varphi$) iff $M \models \varphi\ \forall Ms$ (i.e., $M \in M(\varphi)\ \forall Ms$)
- α, β are equivalent iff $\alpha \models \beta$ and $\beta \models \alpha$ (i.e. $M(\alpha) = M(\beta)$)

Sets of formulas as conjunctions: Let $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$. Then:

- Γ satisfiable iff $\bigwedge_{i=1}^{n} \varphi_i$ satisfiable;
- $\Gamma \models \phi$ iff $\bigwedge_{i=1}^{n} \varphi_i \models \phi$;
- Γ valid iff $\bigwedge_{i=1}^{n} \varphi_i$ valid.

Properties and results:

- Property: φ is valid iff $\neg\varphi$ is unsatisfiable;
- Deduction Theorem: $\alpha \models \beta$ iff $\alpha \Rightarrow \beta$ is valid;
- Corollary: $\alpha \models \beta$ iff $\alpha \wedge \neg\beta$ is unsatisfiable.

Theorem. Entailment (validity, unsatisfiability) in FOL is only semi-decidable:

- if $\Gamma \models \alpha$, this can be checked in finite time;
- if $\Gamma \not\models \alpha$, no algorithm is guaranteed to check it in finite time.

Database semantics Every constant symbol refer to a distinct object—the unique-names assumption. Second, we assume that atomic sentences not known to be true are in fact false—the closed world assumption. Finally, we invoke domain closure, meaning that each model contains no more domain elements than those named by the constant symbols.

Using First-Order Logic Assertions and queries in first-order logic: Sentences are added to a knowledge base using TELL (Tells the KB of the percept at time step t and Tells the KB that it has in fact taken that action), exactly as in propositional logic. Such sentences are called assertions.

We can ask questions of the knowledge base using **ASK** ($ASKs$ the KB for the best action to do at time step t). Questions asked with ASK are called queries or goals.

Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. If we want to know the value we have to use: **ASKVARS**. The corresponding answer is called a substitution or binding list.

function KB-AGENT(*percept*) **returns** an *action*
  persistent: KB, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(*percept*, t))
  action $\leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(*action*, t))
  $t \leftarrow t + 1$
  return *action*

Figure 3.11: Tell and Ask may require complex logical inference

The kinship domain, or family domain: Kinship relations are represented by binary predicates. We use functions for Mother and Father, because every person has exactly one of each of these, biologically.

Axioms are commonly associated with purely mathematical domains. Our kinship axioms are also definitions; they have the form $\forall x, y.[P(x, y)] \Leftrightarrow$. Not all logical sentences about a domain are axioms. Some are theorems—that is, they are entailed by the axioms. Not all axioms are definitions. Some provide more general information about certain predicates without constituting a definition. Indeed, some predicates have no complete definition because we do not know enough to characterize them fully.

Numbers, sets, and lists: it is important to review the theory of natural numbers. What we need is a predicate $NatNum$ that will be true of natural numbers, a constant symbol, 0, and a function symbol, S, the successor. The **Peano axioms** refer to natural numbers and to the addition. We define natural numbers recursively. It means that 0 is a natural number, and for every object n, if n is a natural number, then $S(n)$ is a natural number. Addition is described in terms of the successor function.
The domain of **sets** is also fundamental. We want to be able to represent individual sets, including the empty set. We need a way to build up sets from elements or from operations on other sets. We will want to know whether an element is a member of a set and we will want to distinguish sets from objects that are not sets. We will use the normal vocabulary of set theory as syntactic sugar. The empty set is a constant written as $\{\}$. There is one unary predicate, Set, which is true of sets. The binary predicates are $x \in s$ and $s_1 \subseteq s_2$. The binary functions are $s_1 \cap s_2$, $s_1 \cup s_2$, and $Add(x,\ s)$. One possible set of axioms is as follows:

- The only sets are the empty set and those made by adding something to a set: $\forall s\ Set(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2\ Set(s_2) \wedge s = Add(x, s_2))$;
- The empty set has no elements added into it. In other words, there is no way to decompose $\{\}$ into a smaller set and an element: $\neg\exists x, sAdd(x, s) = \{\}$;
- Adding an element already in the set has no effect: $\forall x, s\ x \in s \Leftrightarrow s = Add(x, s)$;
- The only members of a set are the elements that were added into it. We express this recursively, saying that x is a member of s if and only if s is equal to some element y added to some set s_2, where either y is the same as x or x is a member of s_2:

$$\forall x, s\ x \in s \Leftrightarrow \exists y, s_2\ (s = Add(y, s_2) \wedge (x = y \vee x \in s_2))$$

- A set is a subset of another set if and only if all of the first set's members are members of the second set: $\forall s_1, s_2 \; s_1 \subseteq s_2 \Leftrightarrow (\forall x \; x \in s_1 \Rightarrow x \in s_2)$.
- Two sets are equal if and only if each is a subset of the other: $\forall s_1, s_2 \; (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1)$;
- An object is in the intersection of two sets if and only if it is a member of both sets: $\forall x, s_1, s_2 \; x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2)$;
- An object is in the union of two sets if and only if it is a member of either set: $\forall x, s_1, s_2 \; x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2)$;

Lists are similar to sets. The differences are that lists are ordered and the same element can List appear more than once.

The wumpus world Recall that the wumpus agent receives a percept vector with five elements: stench, Breeze, Glitter, Bump, Scream plus the time step t which is represented as integer.
The **actions** in the wumpus world can be represented by logical terms: Turn(Right), Turn(Left), Forward, Shoot, Grab, Climb. Percepts imply facts about the current state.

We have represented the agent's inputs and outputs; now it is time to represent the **environment** itself. Let us begin with objects. Obvious candidates are *squares* (*term*, *a pair of integers*), *pits* (*predicate*), *wumpus* (*predicate*), *Adjacency* (*binary predicate Adjacent*) and *Position* (*predicate* $At(Agent, s, t)$. It is better to use a complex term in which the row and column appear as integers; for example, we can simply use the list term [1,2].

We could name each pit, but this would be inappropriate for a different reason: there is no reason to distinguish among pits. It is simpler to use a unary predicate Pit that is true of squares containing pits. Finally, since there is exactly one wumpus, a constant Wumpus is just as good as a unary predicate.

The agent's location changes over **time**, so we write $At(Agent, s, t)$ to mean that the agent is at square s at time t.

We can then say that objects can be at only one location at a time. Given its current location, the agent can infer properties of the square from properties of its current percept.

It is useful to know that a square is breezy because we know that the pits cannot move about. Notice that Breezy has no time argument. Having discovered which places are breezy (or smelly) and, very importantly, not breezy (or not smelly), the agent can deduce where the pits are (and where the wumpus is). We can quantify over time, so we need just one successor-state axiom for each predicate, rather than a different copy for each time step.

Knowledge Engineering in First-Order Logic The preceding section illustrated the use of first-order logic to represent knowledge in three simple domains. Now we describe the general process of knowledge-base construction, the so-called **knowledge engineering**. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain.

The relating **steps** are:

- Identify the questions: determine what knowledge must be represented in order to connect problem instances to answers;
- Assemble the relevant knowledge: understand the scope of the knowledge base and understand how the domain actually works;
- Decide on a vocabulary of predicates, functions, and constants: translate relevant domain-level concepts into logic-level names, define the ontology of the domain;
- Encode general knowledge about the domain: write down the axioms for all the vocabulary terms;

- Encode a description of the problem instance: mostly assertions of ground atomic formulas, for a logical agent problem instances are supplied by the sensors, general knowledge base is supplied with additional sentences;
- Pose queries to the inference procedure and get answers: the final outcome and the control of the queries;
- Debug and evaluate the knowledge base: detect un-answered/wrong queries and identify too-weak or missing axioms by backward-analysis.

The electronic circuits domain We start from the following picture:

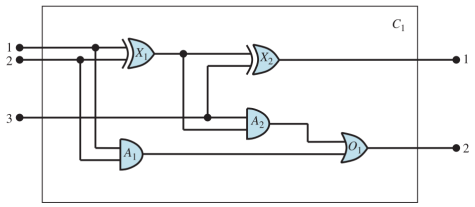


Figure 3.12: The electronic circuits domain

- Identify the questions: There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. Questions about the circuit's structure are also interesting.
- Assemble the relevant knowledge: To reason about functionality and connectivity, all that matters is the connections between terminals.

  Other factors such as the size, shape, color, or cost of the various components are irrelevant to our analysis.

  If our purpose were something other than verifying designs at the gate level, the ontology would be different. For example, if we were interested in debugging faulty circuits, then it would probably be a good idea to include the wires in the ontology, because a faulty wire can corrupt the signal flowing along it.
- Decide on a vocabulary of predicates, functions, and constants: The next step is to choose functions, predicates, and constants to represent them.
- Encode general knowledge of the domain: One sign that we have a good ontology is that we require only a few general rules, which can be stated clearly and concisely. These are all the axioms we will need.
- Encode the specific problem instance: The circuit shown is encoded as circuit C_1 with the following description. First we categorize the circuit and its component gates:

$$
\begin{array}{l}
Circuit(C_1) \wedge Arity(C_{1,3,2}) \\
Gate(X_1) \wedge Type(X_1) = XOR \\
Gate(X_2) \wedge Type(X_2) = XOR \\
Gate(A_1) \wedge Type(A_1) = AND \\
Gate(A_2) \wedge Type(A_2) = AND \\
Gate(O_1) \wedge Type(O_1) = OR
\end{array}
$$

  Then we show the connections between them:

$$
\begin{array}{ll}
Connected(Out(1,X_1), In(1,X_2)) & Connected(In(1,C_1), In(1,X_1)) \\
Connected(Out(1,X_1), In(2,A_2)) & Connected(In(1,C_1), In(1,A_1)) \\
Connected(Out(1,A_2), In(1,O_1)) & Connected(In(2,C_1), In(2,X_1)) \\
Connected(Out(1,A_1), In(2,O_1)) & Connected(In(2,C_1), In(2,A_1)) \\
Connected(Out(1,X_2), Out(1,C_1)) & Connected(In(3,C_1), In(2,X_2)) \\
Connected(Out(1,O_1), Out(2,C_1)) & Connected(In(3,C_1), In(1,A_2))
\end{array}
$$

- Pose queries to the inference procedure: This is a simple example of circuit verification.
- Debug the knowledge base: We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge.

3.3 First Order Logic Inference

Sara Miotto

Propositional vs. First-Order Inference There are two main ideas of dealing with First-Order Inference:

- Converting the KB to propositional logic and use propositional inference;
- A shortcut that manipulates on first-order sentences directly (resolution).

Let's start analyzing the first method: we can obtain first-order inference converting the first-order knowledge base to propositional logic and use propositional inference. A first step is eliminating universal quantifiers, instantiating the universal sentence in all possible ways and convert atomic sentences into propositional symbols. A ground sentence is entailed by the propositionalized Γ if it is entailed by the original Γ and every FOL Γ can be propositionalized to preserve entailment.

Theorem

If a ground sentence α is entailed by an FOL KB Γ, then it is entailed by a finite subset of the propositionalized KB Γ.

The vice-versa does not hold, it works if α is entailed, loops if α is not entailed.

The **rule of Universal Instantiation** says that we can infer any sentence obtained by substituting a ground term, that is a term without variables, for a universally quantified variable:

$$\frac{\forall v\ \alpha}{\textsc{Subst}(\{v/g\}, \alpha)} \tag{3.3}$$

UI can be applied several times to add new sentences and the new sentence is logically equivalent to the old one.

Similarly, the **rule of Existential Instantiation** replaces an existentially quantified variable with a single new constant symbol. The statement is: An existentially quantified-sentence can be substituted by one of its instantation with a fresh constant:

$$\frac{\Gamma \wedge \exists\ x.\alpha}{\Gamma \wedge \alpha\{x/C\}} \tag{3.4}$$

for every variable x and for a constant C, i.e. a constant that does not appear in $\Gamma \wedge \exists\ x.\alpha$. Existential Istantation can be applied once to replace the existential sentence. The new sentence is not equivalent to the old, but is (un)satisfiable if the old Γ is (un)satisfiable and so the new Γ can infer β if the old Γ can infer β.

Remember

Before applying whether the universal or existential instantation, sentences, if negative, must be rewritten in the following way:

- $\neg\forall\ x.\alpha \Rightarrow \exists\ x.\neg\alpha$
- $\neg\exists\ x.\alpha \Rightarrow \forall\ x.\neg\alpha$

Term/Subformula Substitutions: If there is some substitution θ that makes the premise of the implication identical to sentences already in the Knowledge-Base, then we assert the conclusion of the implication, after applying θ.

Definition. A **substitution** is an expression obtained by substituting every occurrence of e_1 with e_2 in e:

- e_1, e_2 either both terms or both subformulas;
- e is either a term or a formula.

Another possibility is the **multiple substitution**: $e\{e_1/e_2, e_3/e_4\} = (e\{e_1/e_2\})\{e_3/e_4\}$.Thus, the general Equal-term substitution rule is:

$$\frac{\Gamma \wedge (t_1 = t_2) \wedge \alpha}{\Gamma \wedge (t_1 = t_2) \wedge \alpha \wedge \alpha\{t_1/t_2\}} \tag{3.5}$$

It is valid also in the following form:

$$\frac{\Gamma \wedge (t_1 = t_2) \wedge \alpha}{\Gamma \wedge (t_1 = t_2) \wedge \alpha\{t_1/t_2\}} \tag{3.6}$$

Talking about formulas, the general rule is:

$$\frac{\Gamma \wedge (\beta_1 \iff \beta_2) \wedge \alpha}{\Gamma \wedge (\beta_1 \iff \beta_2) \wedge \alpha\{\beta_1/\beta_2\}} \tag{3.7}$$

It is valid also in the following form:

$$M\{\Gamma \wedge (\beta_1 = \beta_2) \wedge \alpha(\{\beta_1/\beta_2\})\} \tag{3.8}$$

Problems with Propositionalization First of all, propositionalization generates lots of irrelevant sentences, as with $p \times k$-ary predicates and n constants, $p * n^k$ instantations. Then, due to the nested function applications, we can potentially have infinite instantations. So, for $k = 0$ **to** ∞, use terms of function nesting depth k to create propositionalized Γ by instantiating depth-k terms if $\Gamma \models \alpha$, then will find a contradiction for some finite k, whether if is not we may find a loop forever.

Theorem. Entailment in FOL is semidecidable, there exists algorithms that say yes to every entailed sentence, but no algorithm exists that also says no to every.

Unification and First-Order Inference Let's see now a single inference rule that we call *Generalized Modus Ponens*:

Generalized Modus Ponens For atomic sentences p_i, p_i', and q, where there is a substitution θ such that:

$$\frac{\textsc{Subst}(\theta, \pi') = \textsc{Subst}(\theta, \pi), \forall i,\ p_1', p_2', \ldots, p_n', (p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q)}{\textsc{Subst}(\theta, q)}$$

There are $n+1$ premises to this rule: the n atomic sentences p_i' and the one implication. The conclusion is the result of applying the substitution θ to the consequent q.

It is easy to show that Generalized Modus Ponens is a sound inference rule. First, we observe that, for any sentence p, whose variables are assumed to be universally quantified and for any substitution θ, $(p \models \textsc{Subst}(\theta, p))$ is true by Universal Instantiation. It is true in particular for a θ that satisfies the conditions of the Generalized Modus Ponens rule.

Thus, from $p_1', \cdots, p_n$ we have
$\textsc{Subst}(\theta, p_1') \wedge \cdots \wedge \textsc{Subst}(\theta, p_n')$
and from the implication $p_1 \wedge \cdots \wedge p_n \Rightarrow q$
we can infer $\textsc{Subst}(\theta$, p1)$\wedge$...$\wedge$ $\textsc{Subst}(\theta$, pn) $\Rightarrow$ $\textsc{Subst}(\theta$q).

Now, θ in Generalized Modus Ponens is defined so that $\text{SUBST}(\theta, p'_i) = \text{SUBST}(\theta, p_i)$, for all i.

Therefore the first of these two sentences matches the premise of the second exactly. Hence, $\text{SUBST}(\theta, q)$ follows by Modus Ponens.

Definition. Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a unifier for them if one exists:

$$\text{UNIFY}(p,q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q).$$

If the variable is the same you can standardizing apart one of the two sentences being unified, which means renaming its variables to avoid name clashes.

Here below we report is the algorithm for computing most general unifiers. The process is simple: recursively explore the two expressions simultaneously, building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed.

function UNIFY(x, y, θ=*empty*) **returns** a substitution to make x and y identical, or *failure*
  if θ = *failure* **then return** *failure*
  else if x = y **then return** θ
  else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) **and** COMPOUND?(y) **then**
    return UNIFY(ARGS(x), ARGS(y), UNIFY(OP(x), OP(y), θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(REST(x), REST(y), UNIFY(FIRST(x), FIRST(y), θ))
  else return *failure*

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some *val* **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some *val* **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure*
  else return add $\{var/x\}$ to θ

Figure 3.13: The UNIFY algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution

Unifiers are not unique. Given α, β, the unifier θ_1 is more general than the unifier θ_2 for α, β if exists θ_3 *s.t.* $\theta_2 = \theta_1\theta_3$. An example to clarify this concept: $\{y/John, x/z\}$ more general than $\{y/John, x/John, z/John\} : \{y/John, x/John, z/John\} = \{y/John, x/z\}\{z/John\}$

Theorem

If exists an unifier for α, β, then exists a most general unifier (MGU) θ for α, β.

UNIFY() returns the MGU between two formula.

Storage and retrieval: STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some sentence in the knowledge base.

Forward and Backward Chaining for Definite FOL KBs

Definition. First-Order definite clauses are disjunctions of literals of which exactly one is positive. That means a definite clause is either atomic, or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal.

It is important to say that:

- **Universal quantifiers** are **omitted** as variables are universally quantified by convention.
- **Existential quantifiers** are **removed**.
- **Existentially-quantifiers variables** are **substituted** by fresh constants.

A typical first-order definite clause looks like this: $King(x) \wedge Greedy(x) \Rightarrow Evil(x)$.

The knowledge base happens to be a *Datalog* where:

Definition. A **Datalog** is a language consisting of first-order definite clauses with no function symbols.

Datalog gets its name because it can represent the type of statements typically made in relational databases. The absence of function symbols makes inference much easier.

Forward Chaining Starting from known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts.
When a new fact P is added to the KB for each rule such that P unifies with a premise if the other premises are already known, then add the conclusion to the KB and continue chaining.
Forward chaining is **data-driven** and that means that it infers properties and categories from **percepts**.

function FOL-FC-ASK(KB, α) **returns** a substitution or *false*
  inputs: KB, the knowledge base, a set of first-order definite clauses
    α, the query, an atomic sentence

  while *true* **do**
    $new \leftarrow \{\}$ // *The set of new sentences inferred on each iteration*
    for each *rule* **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(*rule*)
      for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
          for some $p'_1, \ldots, p'_n$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new **then**
          add q' to new
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *failure* **then return** ϕ
    if $new = \{\}$ **then return** *false*
    add new to KB

Figure 3.14: The SIMPLE FORWARD-CHAINING ALGORITHM on each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. There are three sources of inefficiency: the inner loop of the algorithm tries to match every rule against every fact in the knowledge base, the algorithm rechecks every rule on every iteration, the algorithm can generate many facts that are irrelevant to the goal.

Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered or no new facts are added.

Such a knowledge base is called a fixed point of the inference process. First, it is sound, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is complete for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses.

Let's see a **pratical application** of this rule. We try to solve the following problem (which we will later use also to study backward chaining): "The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American".

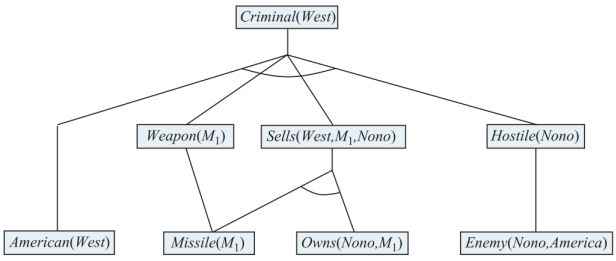


Figure 3.15: The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

With suitable indexing, it is easy to identify all the rules that can be triggered by any given fact, and many real systems operate in an update mode wherein forward chaining occurs in response to every TELL. Inferences cascade through the set of rules until the fixed point is reached, and then the process begins again for the next new fact.

Typically, only a small fraction of the rules in the knowledge base are actually triggered by the addition of a given fact. This means that a great deal of **redundant work** is done in repeatedly constructing partial matches that have some unsatisfied premises. It would be better to retain and gradually complete the partial matches as new facts arrive, rather than discarding them.

The **Rete algorithm** was the first to address this problem.

Irrelevant facts: Another source of inefficiency is that forward chaining makes all allowable inferences based on the known facts, even if they are irrelevant to the goal. In our crime example, there were no rules capable of drawing irrelevant conclusions:

- One way to avoid drawing irrelevant conclusions is to use backward chaining;
- Another way is to restrict forward chaining to a selected subset of rules, as in PL-FC-ENTAILS;
- A third approach has emerged in the field of deductive databases, which are large-scale databases, but which use forward chaining as the standard inference tool. The idea is to rewrite the rule set, using information from the goal, so that only relevant variable bindings are considered during forward inference.

Backward Chaining The second major family of logical inference algorithms uses **backward chaining** over definite clauses. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof.
Backward chaining is the basis for *logic programming* (**Prolog**).

Definition. Logic programming is a technology that comes close to embodying the declarative ideal that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. Prolog is the most widely used logic programming language.

A simple backward-chaining algorithm for first-order knowledge bases is:

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

function FOL-BC-OR(KB, goal, θ) returns a substitution
  for each rule in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs ⇒ rhs) ← STANDARDIZE-VARIABLES(rule)
    for each θ′ in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ′

function FOL-BC-AND(KB, goals, θ) returns a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else
    first,rest ← FIRST(goals), REST(goals)
    for each θ′ in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ″ in FOL-BC-AND(KB, rest, θ′) do
        yield θ″
```

Figure 3.16: A simple backward-chaining algorithm for first-order knowledge bases

FOL-BC-ASK($KB, goal$) will be proved if the knowledge base contains a rule of the form $lhs \Rightarrow goal$, where lhs, left-hand side, is a list of conjuncts.

Backward chaining is a kind of AND/OR search, the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the lhs of a clause must be proved.

FOL-BC-OR works by **fetching** all clauses that might unify with the goal, **standardizing** the variables in the clause to be brand-new variables, and then, if the right-hand-side of the clause does indeed unify with the goal, **proving** every conjunct in the left-hand side, using FOL-BC-AND. That function works by proving each of the conjuncts in turn, keeping track of the accumulated substitution as it goes.

Backward chaining is clearly a **depth-first search algorithm**. This means that its space requirements are linear in the size of the proof. It also means that backward chaining suffers from problems with repeated states and incompleteness.

Here the resolution of the problem previously seen:

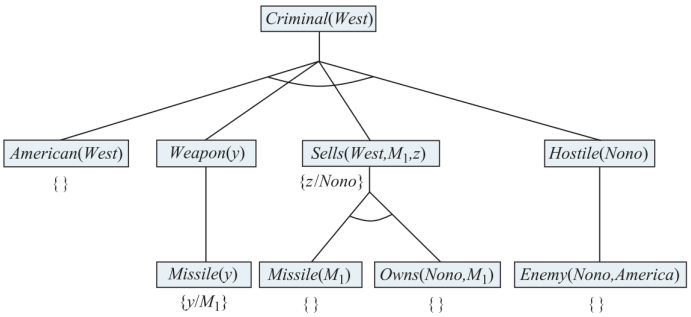


Figure 3.17: Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove Criminal(West), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally Hostile(z), z is already bound to Nono.

Resolution The last of our three families of logical systems, and the only one that works for any knowledge base, not just definite clauses, is resolution.
The **first step** is to convert sentences to conjunctive normal form, **CNF**, a conjunction of clauses, where each clause is a disjunction of literals. In CNF, literals can contain variables, which are assumed to be universally quantified.
The key is that every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. The steps are as follows:

1. Eliminate implications and biconditionals:
   - $\alpha \Rightarrow \beta \Longrightarrow \neg\alpha \vee \beta$;
   - $\alpha \Longleftrightarrow \beta \Longrightarrow (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$;
2. Push inwards negations recursively:
   - $\neg(\alpha \wedge \beta) \longrightarrow \neg\alpha \vee \neg\beta$;
   - $\neg(\alpha \vee \beta) \longrightarrow \neg\alpha \wedge \neg\beta$;
   - $\neg\neg\alpha \longrightarrow \alpha$;
   - $\neg \forall\, x.\alpha \longrightarrow \exists\, x.\neg\alpha$;
   - $\neg \exists\, x.\alpha \longrightarrow \forall\, x.\neg\alpha$.

   Two clarifications:
   - Negation normal form: negations only in front of atomic formulae;
   - Quantified subformulas occur only with positive polarity.
3. Standardize variables: For sentences that use the same variable name twice, change the name of one of the variables: $(\forall x.\exists y.\alpha) \wedge \exists y.\beta \wedge \forall x.\gamma \Rightarrow (\forall x.\exists y.\alpha)\exists y_1.\beta\{y/y_1\} \wedge \forall x_1.\gamma\{x/x_1\}$;
4. Skolemize: Skolemization is the process of removing existential quantifiers by elimination. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable:
   - $\exists y.\alpha \longrightarrow \alpha\{y/c\}$;
   - $\forall x.(\dots \exists y.\alpha \dots) \longrightarrow \forall x.(\dots \alpha\{y/F_1(x)\} \dots)$;
   - $\forall x_1 x_2.(\dots \exists y.\alpha \dots) \longrightarrow \forall x_1 x_2.(\dots \alpha\{y/F_1(x_1, x_2) \dots)\}$;
   - $\exists y_1 \forall x_1 x_2 \exists y_2 \forall x_3 \exists y_3.\alpha \longrightarrow \forall x_1 x_2 x_3.\alpha\{y_1/c, y_2/F_1(x_1, x_2), y_3/F_2(x_1, x_2, x_3)\}$;

   Example. $\forall x \exists y.Father(y, x) \longrightarrow \forall x.Father(s(x), x)$ where ($s(x)$ means "father of x" although $s()$ is a fresh function.
5. Drop universal quantifiers: All remaining variables must be universally quantified:
   $$\forall X_1 \dots X_k.\alpha \Rightarrow \alpha$$
6. Convert to CNF using the distributive laws:
   - Apply recursively the DeMorgan's Rule: $(\alpha \wedge \beta) \vee \gamma \Longrightarrow (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$;
   - Rename subformulas and add definitions: $(\alpha \wedge \beta) \vee \gamma \Longrightarrow (B \vee \gamma) \wedge CNF(B \Longleftrightarrow (\alpha \wedge \beta))$.

   Both satisfiability and entailment are preserved.

Remarks about skolemization

Be aware to convert sentences into NNF and standardize apart variables before applying Skolemization or dropping quantifiers. Moreover, polarity of quantified subformulas affect Skolemization and so NNF-ization may convert $\exists s$ into $\forall s$, and vice versa. Finally, Same-name quantified variable may cause errors.

The resolution inference rule Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain complementary literals. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one unifies with the negation of the other.

$$\frac{l_1 \vee \cdots \vee l_k, \qquad m_1 \vee \cdots \vee m_n}{\textsc{Subst}(\theta,\ l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \cdots \vee l_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)} \tag{3.9}$$

where $\textsc{Unify}\ (l_i, \neg m_j) = \theta$. This rule is called the **binary resolution rule** because it resolves exactly two literals.

To prove that $\Gamma \models \alpha$ in FOL:

- convert $\Gamma \wedge \neg\alpha$ to CNF;
- apply repeatedly resolution rule to CNF $(\Gamma \wedge \neg\alpha)$ until either
  - the empty clause is generate $\Gamma \models \alpha$;
  - no more resolution step is applicable;
  - resource (time, memory) exhausted.

Apply resolution first to unit clauses unit resolution alone complete for definite clauses.

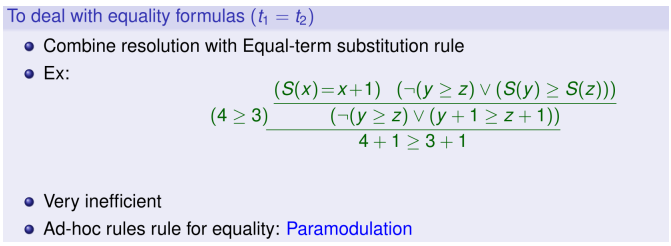
If there is a substitution θ such that $\Gamma \models \theta\alpha$, then it will return θ while if there is no such θ, then the procedure may not terminate.

Saturation Calculus: given N_0, a set of universally quantified clauses, derive $N_0, N_1, N_2, N_3, \ldots\ s.t. N_{i+1} = N_i \cup \{C\}$, where C is the conclusion of a resolution step from premises in N_i. It is refutationally complete, $N_0 \models \bot \Longrightarrow \bot \in N_i$ for some i.

There are some **Resolution Restrictions**:

- Ordered resolution: define stable atom ordering and resolve only maximal literals;
- Hyper-Resolution: clauses are divided into nuclei (they have more than one negative literals) and electrons (they have only positive literals). Resolution can be done only among one nucleus and one electron and globally can produce just electrons.

To **deal with equality formulas** $(t1 = t2)$: Combine resolution with Equal-term substitution rule. For example:



Yet, it is very inefficient.

Paramodulation is an ad-hoc rule to deal with equality:

- Ground case:

$$\frac{D \vee (t = t') \qquad C \vee L}{D \vee C \vee L\{t/t'\}} \tag{3.10}$$

  if t, t' ground, L literal

- General Case:

$$\frac{D \vee (t = t') \quad C \vee L}{(D \vee C \vee L\{u/t'\})\theta} \tag{3.11}$$

  where $\theta = mgu(t, u)$

To conclude this chapter, here is a **complete example** to clarify what we have seen. Consider the following FOL formula set γ:

- $\forall x[\forall y(Child(y) \Rightarrow Loves(x, y))] \Rightarrow [\exists y Loves(y, x)]$
- $\forall x[Child(x) \Rightarrow Loves(Mark, x)]$
- $Beats(Mark, Paul) \vee Beats(John, Paul)$
- $Child(Paul)$
- $\forall x\{[\exists z(Child(z) \wedge Beats(x, z))] \Rightarrow [\forall y \neg Loves(y, x)]\}$

Let's start with the computation of the CNF-ization of γ, Skolemization and standardization of variables:

- 1. $\forall x\{[\forall y(Child(y) \Rightarrow Loves(x, y))] \Rightarrow [\exists y Loves(y, x)]\}$;
  2. $\forall x\{[\neg\forall y(Child(y) \Rightarrow Loves(x, y))] \vee [\exists y Loves(y, x)]\}$;
  3. $\forall x\{[\exists y(Child(y) \wedge \neg Loves(x, y))] \vee [\exists y Loves(y, x)]\}$;
  4. $\{[(Child(F(x)) \wedge \neg Loves(x, F(x)))] \vee [Loves(G(x), x)]\}$
     (a) $Child(F(x)) \vee Loves(G(x), x)$;
     (b) $\neg Loves(y, F(y)) \vee Loves(G(y), y)$
- $\neg Child(z) \vee Loves(Mark, z)$
- $Beats(Mark, Paul) \vee Beats(John, Paul)$
- $Child(Paul)$
- 1. $\forall x\{[\exists z(Child(z) \wedge Beats(x, z))] \Rightarrow [\forall y \neg Loves(y, x)]\}$
  2. $\forall x\{[\neg\exists z(Child(z) \wedge Beats(x, z))] \vee [\forall y \neg Loves(y, x)]\}$
  3. $\forall x\{[\forall z(\neg Child(z) \vee \neg Beats(x, z))] \vee [\forall y \neg Loves(y, x)]\}$
  4. $\neg Child(z2) \vee \neg Beats(x2, z2) \vee \neg Loves(y2, x2)$

Where $F()$, $G()$ are Skolem unary functions. Now we are asked to write a FOL-resolution inference of the query Beats(John, Paul) from the CNF-ized KB:

1. $[1.2, 2.] \longrightarrow \neg Child(F(Mark)) \vee Loves(G(Mark), Mark)$;
2. $[1.1, 6.] \longrightarrow Loves(G(Mark), Mark)$;
3. $[4, 5.] \longrightarrow \neg Beats(x2, Paul) \vee \neg Loves(y2, x2)$;
4. $[7, 8.] \longrightarrow \neg Beats(Mark, Paul)$;
5. $[3, 9.] \longrightarrow Beats(John, Paul)$;

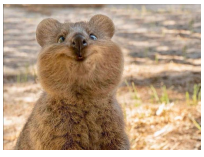


Figure 3.18: Quokka felice.

3.4 Classical Planning

Thomas De Min

Automated Planning (Planning) Synthesize a sequence of actions (plan) to be performed by an agent leading from an initial state of the world to a set of target states (goal). In other words, given an **initial state**, a set of executable **actions** and a **goal**, the objective is to find a plan. A plan is defined as a partially- or totally-ordered set of actions needed to achieve the goal from the initial state. In this chapter we will consider fully observable, deterministic, static environments with a single agent.

Definition. PlanSAT answer the question of whether there exists any plan that solves a planning problem. It is decidable for classical planning, undecidable with function symbols. It is in $\mathbb{PSPACE}$.

Definition. Bounded PlanSAT answer the question of whether there exists any plan of length k or less. It can be used for optimal-length plan and it is decidable in both of the previous cases. For many problems it is in $\mathbb{NP}$ for others it is in $\mathbb{PSPACE}$.

PDDL In order to represent the planning problem we use **PDDL** (Planning Domain Definition Language). Each **state** is represented as a conjunction of fluents that are ground[7], function-less atoms. PDDL use the database semantics which involves the closed-world assumption (any fluent not mentioned is considered false) and the unique names assumption ($Truck_1$ and $Truck_2$ are distinct).

Example. $Poor \wedge Unknown, At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$ are syntactically correct in PDDL. The following fluents are not allowed: $At(x, y)$ (non ground), $\neg Poor$ (negation), $At(Father(Fred), Sydney)$ (not function-less)

Actions are described by a set of action schemas that implicitly define the $\textsc{Actions}(s)$ and $\textsc{Result}(s, a)$ functions needed to do a problem-solving search. Basically actions describe which fluent change. A set of ground actions can be represented by a single action schema. The schema is a **lifted** representation: it lifts the level of reasoning from PL to a restricted subset of FOL and so it consists in action name, the list of variables, the precondition, the effect.

Example. Action schema of a plane flying from a location to another.
$Action(Fly(p, from, to))$,
$\textsc{Precond}$: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
$\textsc{Effect}$: $\neg At(p, from), \wedge At(p, to)$

An instantiation of the same action is:
$Action(Fly(P_1, SFO, JFK))$,
$\textsc{Precond}$: $At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)$
$\textsc{Effect}$: $\neg At(P_1, SFO) \wedge At(P_1, JFK)$

Both the precondition and effect of an action are conjunction of literals. The precondition defines the states in which the action can be executed. The effect defines the result of executing the action. An action a can be executed in state s if s entails the precondition of a. Using the set semantics: $s \models q$ iif every positive literal in q is in s and every negated literal in q is not. We say that action a is **applicable** in state s if the preconditions are satisfied by s. The **result** of executing an action a in state s is defined as a state s' which is represented by the set of fluents formed by starting with s, removing the fluents that appear as negative literals in the action's effects (**delete list** $\textsc{Del}(a)$), and adding the fluents that are positive literals in the action's effects (**add list** $\textsc{Add}(a)$): $\textsc{Result}(s, a) = (\text{s - } \textsc{Del}(a)) \cup \textsc{Add}(a)$.

Example. Take the previous example, after applying $Fly(P_1, SFO, JFK)$ our KB would contain $At(P_1, JFK)$ but no longer $At(P_1, SFO)$.

In order to propositionalize a PDDL problem each action schema must be replaced with a set of ground actions (e.g. $At\_P_1\_SFO \wedge Plane\_P_1 \wedge ...$).

Fluents do not explicitly refer to time. Times and state are implicit in the action schema: precondition refers to time t and effect time $t + 1$.

[7] variable-free.

The initial state is a conjunction of **ground atoms**. The goal is a conjunction of **literals** (positive or negative) that may contains variables, such as $\neg At(p, SFO) \land Plane(p)$:

Example. Here we can see the difference
$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JKF)) \land ...$
$Goal(At(p, SFO) \land Plane(p))$

Planning has all components of search problem. To search in the state space we can use two approaches:

- **Forward Search** which uses actions to search forward, from the initial state, for a goal state.
- **Backward Search** which uses reverse actions (thanks to the declarative representations of action schemas) to search forward, from goal states, for the initial state.

Forward Search Consists in choosing actions whose preconditions are satisfied, then adding positive effects and deleting negative ones. If the current state satisfies the goal test then we found a goal state. Each action cost 1. That implies we can use any of the already seen search algorithms:

- **Breadth-first search**: Sound) If it returns a plan, that plan it is a solution; Complete) If a solution exists, BFS will return one. Requires exponential memory w.r.t. the solution length. Usually it is unpractical.
- **Depth-first search**: Sound, Not complete because may enter in infinite loops (made complete by loop-checking but only in classical planning). Require linear memory w.r.t. the solution length.

Backward Search We start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. It is called **relevant-states** search because we only consider actions that are relevant to the goal (or current state). Given a ground goal description g and a ground action a, the regression from g over a gives us a state description g' defined by

$$g' = (g - \textsc{Add}(a) \cup Precond(a))$$

That is, the effects that were added by the action, need not have been true before and also the preconditions must have held before, or else the action could not have been executed. $\textsc{Del}(a)$ does not appear in the formula because while we know the fluents in $\textsc{Del}(a)$ are no longer true after the action, we do not know whether or not they were true before. We need to deal with partially un-instantiated actions and states, so we would like to avoid unnecessary instantiations because there is no need to produce a goal for every possible instantiation[8]. At the end, we want actions that are relevant (while in Forward search we were looking for applicable ones). **Relevant actions** are those actions that could be the last step in a plan leading up to the current goal state. For an action to be relevant to a goal it obviously must contribute to the goal: at least one of the action's effects (either positive or negative) must unify with an element of the goal[9]. A consistent action must not undo desired literals of the goal (clearly unconsistent actions are non-relevant).

Example. Consider the goal $At(C_1, SFO) \land At(C_2, JFK)$
$Action(Unload(C_1, p', SFO), ...)$ is relevant because C_1 would then be in SFO as the goal requires[10].
$Action(Unload(C_3, p', SFO), ...)$ is not relevant (analogous to the previous case).
$Action(Load(C_2, p', JFK), ...)$ is not consistent because C_2 would no longer be in JFK (requirement for the goal state) thus it is not relevant.

B.S. typically keeps the branching factor lower than F.S. but it reasons with state sets rather than individual states which makes it harder to come up with good heuristics. Most planners work with F.S. plus heuristics.

[8] Slide 24 also refers to the use of most general unifier and to the standardization of the action schemata. The book does not spend too many words on these. In fact everything just mentioned is "between round brackets".

[9] AIMA now gives the formal definition, skipped by rseba.

[10] Note: p' substitutes p due to the standardization of the action schemata as introduced in the footnotes.

Heuristics Due to the high branching factor, planning problems can have huge state spaces, we need a good heuristic to guide the search. A heuristic function $h(a)$ estimates the distance from a state s to the goal and if we can derive an **admissible** heuristic for this one then we can use A* search to find optimal solutions[11]. An admissible heuristic can be derived by defining a **relaxed problem** that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem. There are different forms of problem relaxation: one consists in adding arcs to the search graphs (and so aims at simplifying the research) and the other in clustering nodes (and so reducing the state abstraction). Regarding the first methodology there are two possibilities: ignoring preconditions and ignoring the delete list, while for the second approach just one, that is ignoring less relevant fluents.

The first one is the **Ignore-Preconditions Heuristics**. It drops all preconditions from actions, thus every action becomes applicable in every state and any single goal fluent can be achieved in one step (if there is an applicable action, if not, the problem is impossible). It is a fast but over-optimistic approach. An alternative is to remove all preconditions and effects, except literals in the goal. This produces a more accurate heuristic but, unfortunately, it is an instance of the set-cover problem... which is $\mathbb{NP}$-complete, but efficient greedy algorithms exist[12]. Another alternative is to ignore some selected (less relevant) preconditions.

The second one is the **Ignore Delete-list Heuristics**. Let us assume for a moment that all goals and preconditions contain only positive literals. Then we can remove the delete lists from all actions (i.e. remove all negative literals from effects). That makes it possible to make a monotonic progress towards the goal since no action will ever undo progress made by another action. This is in $\mathbb{NP}$-hard but it can be approximated in polynomial time with hill-climbing.

Finally, **State Abstraction** is a many-to-one mapping from states in the ground/original representation of the problem to a more abstract representation. The easiest form of S.A. is to ignore some fluents.

Example. Consider an air cargo problem with 10 airports, 50 planes, and 200 pieces of cargo. Each plane can be at one of 10 airports and each package can be either in one of the planes or unloaded at one of the airports. There are $10^{50} \cdot (50+10)^{200} \approx 10^{405} states$. Now consider a particular problem in that domain in which it happens that all the packages are at just 5 of the airports, and all packages at a given airport have the same destination. Then we can drop all the At fluents except for the ones involving one plane and one package at each of the 5 airports. Now there are only $10^5 \cdot (5+10)^5 \approx 10^{11}$ states.

Planning Graphs The previous heuristics suffer from inaccuracies. However, there are other strategies that have the objective to define heuristics, that are problem decomposition (divide et impera) and planning graphs, the topic of this chapter. In this paragraph we introduce a data structure that can be used to give better heuristic estimates $h(s)$. It is a polynomial-size approximation to the exponential search tree which therefore can be constructed very quickly. The main drawback of Planning Graphs is that they can't answer definitively whether G is reachable from S_0 (initial state), but they can estimate how many steps it takes to reach G. Anyway the estimate is always correct when it reports the goal it is not reachable, and it never overestimate the number of step, thus it is an admissible heuristic.

Definition. A Planning Graph is a directed graph, built forward and organized into levels:

- level S_0 for the initial state consisting of nodes representing each fluent that holds in S_0;
- level A_0 consisting of nodes for each ground action that might be applicable in S_0;
- ...
- level S_i containing all the literals that could hold at time i, depending on the actions executed at preceding time steps. If it is possible that either P or $\neg P$ could hold, then both will be represented in S_i,
- level A_i contains all the actions that could have their preconditions satisfied at time i.

This goes on until $S_N = S_{N-1}$. In other words, when two consecutive levels are identical. At this point the graph is "leveled off".

Figure 3.19 shows a simple planning problem, and Figure 3.20 shows its planning graph. A literal appears because an action caused it, but we also want to say that a literal can persist if no action negates

[11] See Section 2.1 for further explanation.
[12] Note that greedy algorithms lose the the guarantee of admissibility.

it. This is represented by a **persistence action** (no-op). The gray lines in Figure 3.20 indicate **mutual exclusion** (or **mutex**) links. For example, $Eat(Cake)$ is mutually exclusive with the persistence of either $Have(Cake)$ or $\neg Eaten(Cake)$.

$Init(Have(Cake))$
$Goal(Have(Cake) \ \wedge \ Eaten(Cake))$
$Action(Eat(Cake)$
  PRECOND: $Have(Cake)$
  EFFECT: $\neg \ Have(Cake) \ \wedge \ Eaten(Cake))$
$Action(Bake(Cake)$
  PRECOND: $\neg \ Have(Cake)$
  EFFECT: $Have(Cake))$

Figure 3.19: Have cake and eat cake problem

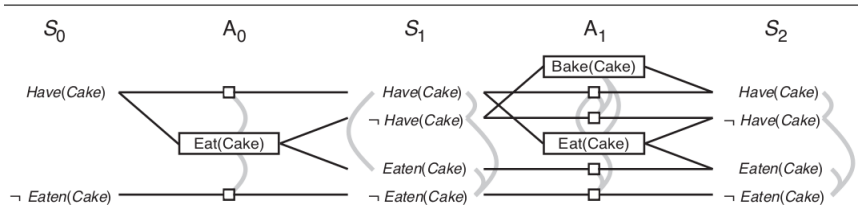


Figure 3.20: Planning graph of Figure 3.19. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutes links are shown, because the graph would be too cluttered. In general, if two literals are mutex at S_i, then the persistence actions for those literals will be mutex at A_i, and we need not draw the mutex link.

We now define mutex links for both actions and literals. A mutex relation holds between two **actions** at a give level if any of the following three conditions holds:

- *Inconsistent effects*: one action negates an effect of the other. For example, $Eat(Cake)$ and the persistence of $Have(Cake)$ have inconsistent effects because they disagree on the effect $Have(Cake)$[13].

- *Interference*: one of the effects of one action is the negation of a precondition of the other. For example $Eat(Cake)$ interferes with the persistence of $Have(Cake)$ by negating its precondition.

- *Competing needs*: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, $Bake(Cake)$ and $Eat(Cake)$ are mutex because they compete on the value of the $Have(Cake)$ precondition.

Otherwise they don't interfere with each other, thus both may appear in a solution plan.

A mutex relation holds between two **literals** at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals are mutually exclusive. This condition is called *Inconsistent support*. All ways of achieving them are pairwise mutex. For example, $Have(Cake)$ and $Eaten(Cake)$ are mutex in S_1 because the only way of achieving $Have(Cake)$, the persistent action, is mutex with the only way of achieving $Eaten(Cake)$, namely $Eat(Cake)$.

[13]This and the next examples refers to Figure 3.20.

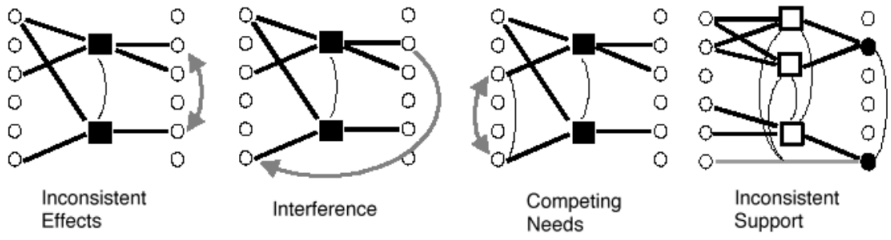


Figure 3.21: Mutex relations

Note

Planning graph deals with ground states and actions only.

Building of the Planning Graph [14]

1. Create initial layer S_0: insert into S_0 all literals in the initial state;
2. **Repeat for increasing values of** $i = 0, 1, 2, ...$**:**
3. Create action layer A_i: for every action schema, for each way to unify its preconditions to non-mutually exclusive literals in S_i, enter an action node into A_i. For every literal in S_i, enter a no-op action node into A_i. Add mutexes between the newly-constructed action nodes;
4. Create state layer S_{i+1}: for each action node a in A_i, add to S_{i+1} the fluents in his Add list linking them to a and the negated fluents in his Del list linking them to a. For every "no-op" action node a in A_i, add the corresponding literal to S_{i+1} and link it to a. Add mutexes between literal nodes in S_{i+1};
5. **... until** $S_{i+1} = S_i$ **or if there is any bound, it is reached**.

A planning graph is polynomial in the size of the planning problem. For a planning problem with l literals and a actions, each S_i has no more than l nodes and l^2 mutex links, and each A_i has no more than $a + l$ nodes (including no-ops), $(a + l)^2$ mutex links, and $2(a \cdot l + l)$ precondition and effect links. Thus, an entire graph with n levels has a size of $O(n(a + l)^2)$. The time to build the graph has the same complexity.

Planning Graphs for Heuristic Estimation With Planning Graphs we can decide if a problem is unsolvable and also we can estimate the cost of achieving any goal literal g_i, from state s, as the level at which g_i first appears in the planning graph, constructed from initial state s. We call the last one **level cost** of g_i. In Figure 3.20, $Have(Cake)$ has level cost 0 and $Eaten(Cake)$ has level cost 1. Each level S_i represents a set of possible belief states, two literals connected by a mutex belong to different belief states. A literal not appearing in the final level of the graph cannot be achieved by any plan, thus if a goal literal is not in the final level, the problem is unsolvable.

Planning graph estimates might not always be accurate, however, because planning graphs allow several actions at each level, whereas the heuristic counts just the level and not the number of actions. For this reason, it is common to use a **serial planning graph** for computing heuristics. A serial graph insists that only one action can actually occur at any given time step. This is done by adding mutex links between every pair of nonpersistence actions. Level costs extracted from serial graphs are often quite reasonable estimates of actual costs.

To estimate the cost of a conjunction of goals there are three simple approaches:

- **Max-level** heuristic: takes the maximum level cost of any of the goals. Admissible but not always accurate;
- **Level-sum** heuristic: returns the sum of the level costs of the goals. This can be inadmissible but works well in practice for problems that are largely decomposable;
- **Set-level** heuristic: finds the level at which all the literals in the conjunctive goal appear in the planning graph without any pair of them being mutually exclusive. It is admissible, and it works extremely well on tasks in which there is a good deal of interaction among subplans.

[14] I think this can be considered as pseudocode.

The Graphplan Algorithm This paragraph shows how to extract a plan directly from the planning graph, rather than just using the graph to provide a heuristic. The GRAPHPLAN Algorithm (Figure 3.22) repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non-mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

function GRAPHPLAN(*problem*) **returns** solution or failure

graph ← INITIAL-PLANNING-GRAPH(*problem*)
goals ← CONJUNCTS(*problem*.GOAL)
nogoods ← an empty hash table
for $t = 0$ **to** ∞ **do**
  if *goals* all non-mutex in S_t of *graph* **then**
    solution ← EXTRACT-SOLUTION(*graph*, *goals*, NUMLEVELS(*graph*), *nogoods*)
    if *solution* $\neq$ *failure* **then return** *solution*
  if *graph* and *nogoods* have both leveled off **then return** *failure*
  graph ← EXPAND-GRAPH(*graph*, *problem*)

Figure 3.22: The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

Graphplan is a "family" of algorithms, depending on approach used in EXTRACT-SOLUTION(...). EXTRACT-SOLUTION can be formulated as an incremental SAT problem with one proposition for each ground action and fluent and clauses that represent precondition, effects, no-ops and mutexes. Alternatively, can be formulated as a backward search problem. The planning problem is therefore restricted to planning graph which is faster than unrestricted planning.

If the planning graph is not serialized may produce partial order plans which can be later serialized into a total-order plan. Total-order plans are strictly linear sequences of actions, disregards the fact that some actions are mutually independent. Partial-order plans, instead, are a set of precedence constraints between action pairs. A partial order plan forms a D.A.G. which longest path to the goal may be much shorter than total-order plan. It can be easily converted into possibly many distinct total-order plans.

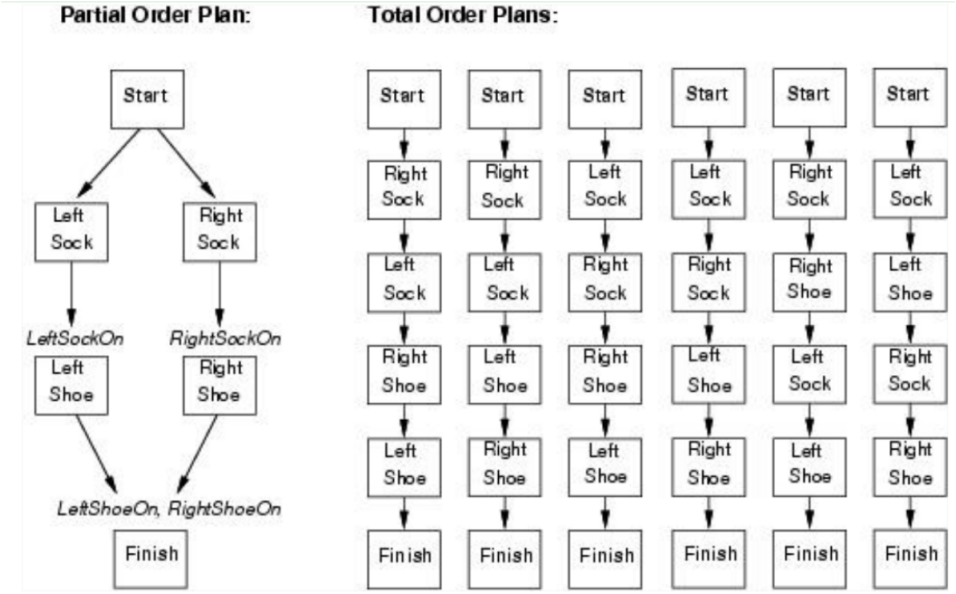


Figure 3.23: Partial Order Plan on the left and Total Order Plans on the right.

Termination of Graphplan

Theorem. If the graph and the no-goods have both leveled off, and no solution is found, we can safely terminate with failure.

Intuition (proof sketch): Literals and actions increase monotonically and are finite because we will eventually reach a level where they stabilize. Mutex and no-goods decrease monotonically (and cannot become less than zero), they too eventually must level off. When we reach this stable state, if one of the goals is missing or is mutex with another goal, then it will remain so, we can stop.

Other Approaches (hints) [15] The first alternative approach is to encode a bounded planning problem into a propositional formula then solve it by (incremental) calls to a SAT solver. It is extremely efficient with many problems of interest.

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*

Figure 3.24: SATPLAN

The second alternative approach involves the formalization of panning into FOL, then use resolution-base inference for planning. On one side it admits qualifications, which can be translated into more expressiveness, on the other hand it is complicate to handle the Frame problem (no-ops)[16]. Moreover, it is not very efficient although it is interesting theoretically[17].

We finally arrived at the end of this troubled chapter. To thank you, here is a kitten 4 u (Figure 3.25).

Figure 3.25: Anakin the cat.

3.5 Real World Planning

Andrea Bonomi, Sara Miotto

[15] I don't think he will ever ask for these in details. For completeness I included the main aspects.
[16] He does not provide further explanations on the Frame problem.
[17] Here I skip the theoretical explanation.

Definition. Classical Planning talks about what to do, in what order, but not how long an action take and when it occurs. This is the subject matter of **scheduling**.

Definition. The real world also imposes **resource constraints**, think for example to an airline company: it has a limited number of staff, and staff who are on one flight cannot be on another at the same time.

Times, Schedule & Resources We understand that real world planning is divided in two phases:

- Planning phase: actions are selected, with some ordering constraint, to meet the goals of the problem;
- Scheduling phase: temporal information is added to the plan to ensure that it meets resource and deadline constraints.

Example. Consider the following planning phase:

$Init(Chassis(C1) \land Chassis(C2) \land Engine(E1, C1, 30) \land Engine\ (E1, C2, 60) \land Wheels(W1, C1, 30) \land Wheels(W2, C2, 15))$

$Goal(Done(C1) \land Done(C2))$

$Action(AddEngine(e, c, d)$
Precond: $Engine(e, c, d) \land Chassis(c) \land \neg EngineIn(c)$
Effect: $EngineIn(c) \land Duration(d)$
Consume: $LugNuts(20)$, Use: $EngineHoists(1))$

$Action(AddWheels(w, c, d)$
Precond: $Wheels(w, c, d) \land Chassis(c)$
Effect: $WheelsOn(c) \land Duration(d)$
Consume: $LugNuts(20)$, Use: $WheelStations(1))$

$Action(Inspect(c,10))$
Precond: $EngineIn(c) \land WheelsOne(c) \land Chassis(c)$
Effect: $Done(c) \land Duration(10)$
Use: $Inspectors(1)$

The solution is a partial plan as following:

$$\begin{Bmatrix} AddEngine(E1, C1, 30) \prec AddWheels(W1, C1, 30) \prec Inspect(C1, 10); \\ AddEngine(E2, C2, 60) \prec AddWheels(W2, C2, 15) \prec Inspect(C2, 10) \end{Bmatrix}$$

Notice that the notation $A \prec B$ means that action A must precede action B.

Now consider a Job-Shop Scheduling problem, it consists of a set of jobs, each of which has a collection of actions with ordering constraints among them. Each action has a duration and a set of resource constraints required by the action. A constraint specifies a type of resource, the number of that resource required, and whether that resource is consumable or reusable. A solution specifies the start time for each action and must satisfy all the temporal ordering constraints and resource constraints. As we have seen before: solutions can be evaluated according to a cost function that could be very complicated.

Definition. In order to be simple, we assume that the cost function is just the total duration of the plan, which is called the **makespan**.

In order to determine the possible start and end times of each action we can apply the **critical path method**(CPM).

Definition. A **path** through a graph representing a partial order plan is a linearly ordered sequence of actions beginning with *Start* and ending with $Finish$.

Definition. The **critical path** is that path whose duration is longest. It is called critical because it determines the duration of the entire plan.

Actions that are off the critical path have a window of time in which they can be executed, the window is specified in terms of an earliest possible start time, ES, and a latest possible start time, LS. The quantity $LS - ES$ is known as the **slack** of an action. Together the ES and LS times for all the actions constitute a **schedule** for the problem with the following formulas:

$$\begin{aligned} ES(Start) &= 0 \\ ES(B) &= \max_{\{A|A \prec B\}} (ES(A) + Duration(A)) \\ LS(Finish) &= ES(Finish) \\ LS(A) &= \min_{\{B|B \succ A\}} (LS(B) - Duration(A)) \end{aligned}$$

The idea is that we start by assigning $ES(Start)$ to be 0. Then, as soon as we get an action B such that all the actions that come immediately before B have ES values assigned, we set $ES(B)$ to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration. This process repeats until every action has been assigned an ES value. The LS values are computed in a similar manner, working backward from the $Finish$ action. The complexity of the critical path algorithm is just $O(Nb)$, where N is the number of actions and b is the maximum branching factor into or out of an action.

Example. Consider as example the following planning phase:

$Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\}, \{AddEngine2 \prec AddWheels2 \prec Inspect2\})$

$Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500))$

$Action(AddEngine1$, Duration:30, Use:$EngineHoists(1))$
$Action(AddEngine2$, Duration:60, Use:$EngineHoists(1))$
$Action(AddWheels1$, Duration:30, Consume: $LugNuts(20)$, Use:$WheelStations(1))$
$Action(AddWheels2$, Duration:15, Consume: $LugNuts(20)$, Use:$WheelStations(1))$
$Action(Inspect_i$, Duration:10, Use:$Inspectors(1))$

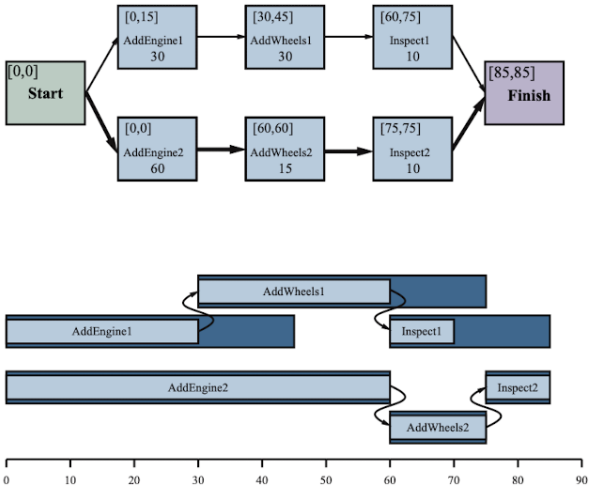


Figure 3.26: Critical Path Method Planning Example

Mathematically (and computationally) speaking, critical-path problems are easy to solve because they are defined as a conjunction of linear inequalities on the start and end times. However, remember that we did not add resources until now. Adding resources makes the problem much harder, this due to the disjunction of linear inequalities (or "*Cannot overlap*" constraint).

Example. Consider the example before and add account resource constraints.

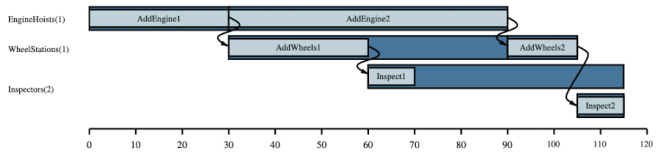


Figure 3.27: Critical Path Method Scheduling Example

Planning and Acting in Non Deterministic Domains Until now we made some assumptions for non deterministic domains:

- The environment is deterministic, fully observable and static;
- The agent knows the effects of each action.

With these assumptions the agent does not need the perception because it can calculate which state results from any sequence of actions and always knows which state it is in. However, considering the real world, the environment may be uncertain, in fact it can be partially observable and/or non deterministic and it could borrow incorrect information.
If one of the assumptions does not hold we use percepts. We have to manage three steps of planning:

- Sensorless planning (or conformant planning): find a plan that achieves the goal in all possible circumstances (if any);
- Conditional planning (or contingency planning): construct a conditional plan with different branches for possible contingencies;
- Executing, monitoring and replanning: while constructing plan, judge whether plan requires revision.

Classical planning is based on **closed-world assumption** (CWA) where the state contain only positive fluents and we assume that every fluent not mentioned in a state is false. Thus, sensorless and partially observable planning are based on **open world assumption** (OWA) where states contain both positive and negative fluents and if a fluent does not appear in the state, its value is unknown. The belief state, represented by a logical formula, corresponds exactly to the set of possible worlds that satisfy the formula representing it. The unknown information can be retrieved via sensing actions added to the plan.

Example. Consider the following problem called "*The table and chair painting problem*":

Given a chair and a table, the goal is to have them of the same color.
In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown.
Only the table is initially in the agent's field of view.

Initial State	$Init(Object(Table) \land Object(Chair) \land Can(C1) \land Can(C2) \land InView(Table))$
Goal	$Goal(Color(Chair, c) \land Color(Table, c))$
Actions	$Action(RemoveLid(can)$, Precond: $Can(can)$, Effect: $Open(can))$ $Action(Paint(x, can)$, Precond: $Object(x) \land Can(can) \land Color(can, c) \land Open(can)$, Effect: $Color(x, c))$
Add action	$Action(LookAt(x)$, Precond: $InView(y) \lor (n \neq y)$, Effect: $InView(x) \land \neg InView(y))$

Table 3.1: Planning phase of the table and chair painting problem

We need to reason about percepts obtained during action because we consider a partial observable problem. To do this we augment PDDL with percept schema for each fluent:

1. $Percept(Color(x, c)$, Precond: $Object(x) \wedge InView(x))$, this means "*if an object is in view, then the true agent will perceive its color*" and thus perception will acquire the truth value of $Color(x, c)$, for every x, c;

2. $Percept(Color(can, c)$, Precond: $Can(can) \wedge InView(can) \wedge Open(can))$, this means "*if an open can is in view, then the agent perceives the color of the paint in the can*" and thus perceptin will acquire the truth value of $Color(can, c)$, for every can, c.

If the problem is fully observable, then there is no preconditions for each fluent: $Percept(Color(x, c))$.

Sensorless Planning The sensorless, or conformant, planning deals with domains in which the state of the world is not fully known. Due to this fact, it comes up with plans that work in all possible cases.

We can tackle a sensorless planning problem as if it were a belief-state planning problem.

The main differences between these two are:

- The underlying physical transition model is represented by a collection of action schemas;

- The belief state can be represented by a logical formula instead of by an explicitly enumerated set of states;

- The planners deal with factored representations rather than atomic.

We assume that our sensorless planning problem is deterministic and that it requires an open-world assumption.

All belief states include unchanging facts while the initial belief state includes facts that are part of the agent's domain knowledge. Let's see now how to progress the belief state through the action sequence to show that the final belief state satisfies the goal. First, in a given belief state b, the agent can consider any action whose preconditions are satisfied by b. To construct the new belief state b', we must consider what happens to each literal l in each physical state s in b when an action a is applied. For literals whose truth value is already known in b, the truth value in b' is computed from the current value and the add list and delete list of the action. That is to say that, for example, if l is in the delete list of the action, then $\neg l$ is added to b'.

What about a literal whose truth value is unknown in b?

- If the action adds l, then l will be true in b' regardless of its initial value;

- If the action deletes l, then l will be false in b' regardless of its initial value;

- If the action does not affect l, then l will retain its initial value (which is unknown) and will not appear in b'.

So, the result is computed:

- Start from b;

- Set to false any atom that appears in $Del(a)$ (after unification);

- Set to true any atom that appears in $Add(a)$ (after unification);

$b' = RESULT(b, a) = (b - DEL(a)) \cup ADD(a)$.

If the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals, so in a world with n fluents, any belief state can be represented by a conjunction of size $O(n)$. So it is much simpler than a belief-state reasoning, that required 2^N states.

Solution of the table and chair painting problem:

- Start from $b_0 : Color(x, C(x))$
- Apply $RemoveLid(Can_1)$ in b_0 and obtain:
  $b_1 : Color(x, C(x)) \wedge Open(Can_1)$
- Apply $Paint(Chair, Can_1)$ in b_1 using $\{x/Can_1, c/C(Can_1)\}$:
  $b_2 : Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1))$
- Apply $Paint(Table, Can_1)$ in b_2 :
  $b_3 : Color(x, C(x)) \wedge Open(Can_1) \wedge Color(Chair, C(Can_1)) \wedge Color(Table, C(Can_1))$
- b_3 Satisfies the goal: $b_3 \models Color(Table, c) \wedge Color(Chair, c)$

8 $\Rightarrow$ $[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)]$
valid conformant plan

Figure 3.28: Here is an example of sensorless planning

Conditional Planning Conditional Planning, or Contingent planning, handles domains where the effects of an action are not deterministic. The approach is of planning ahead for different possible results of each action.

Contingency planning is appropriate for environments with partial observability, nondeterminism, or both of them.

Before going ahead, let's revoke the case of searching with non-deterministic actions. The solution was a contingency plan which contained nested conditions on future percepts.

In a non-deterministic environment, there were branches also on environment's choice of outcome for each action. The agent had to handle all such outcomes.

AND-OR search trees were emploied, where AND nodes corresponded to actions, while leaf nodes to the goal, the dead-end or the loop OR nodes.

However, the main differences between contingent planning and searching in a non-deterministic environment are:

- Planners deal with factored representations rather than atomic;
- Physical transition model is a collection of action schemata;
- The belief state represented by a logical formula instead of an explicitly-enumerated set of states;
- Sets of belief states represented as disjunctions of logical formulas representing belief states.

When executing a contingent plan, the agent maintains its belief state as a logical formula and it evaluates each branch condition:

- If the belief state entails the condition formula, then proceed with the *then* branch;
- If the belief state entails the negation of the condition formula, then proceed with the *else* branch.

To compute the result we have to follow three steps:

- Prediction: same as for sensorless, that is: $\hat{b} = (b - DEL(a)) \cup ADD(a)$;
- Observation Prediction: determines the set of percepts that could be observed in the predicted belief state;
- Update:
  - if p has one percept schema, $Percept(p, Precond : c)$, s.t. $\hat{b} \models c$, then $b_p = p \wedge c$, that means that c, the conjunction of literals, can be thrown into the belief state along with p;

– if p has k percept schemata, $Percept(p, Precond : c_i)$, s.t. $\hat{b} \models c_i, i = 1, ..., k$, then $b_p = \bigvee_{i=1}^{k}(p \wedge c_i)$, to put it in other words, if p has more than one percept schema whose preconditions might hold according to the predicted belief state $\hat{b}$, then we have to add in the disjunction of the preconditions.

Solution of the table and chair painting problem:

- Possible contingent plan for previous problem described below
  - variables in the plan to be considered existentially quantified
  - ex (2^{nd} row): "if there exists some color c that is the color of the table and the chair, then do nothing" (goal reached)
- "Color(Table,c)", "Color(Chair,c)' and "Color(Can,c)" percepts
  $\Longrightarrow$ must be matched against percept schemata

$$
\begin{aligned}
&[LookAt(Table), LookAt(Chair),\\
&\ \textbf{if } Color(Table, c) \wedge Color(Chair, c) \textbf{ then } NoOp\\
&\quad \textbf{else } [RemoveLid(Can_1), LookAt(Can_1), RemoveLid(Can_2), LookAt(Can_2),\\
&\qquad \textbf{if } Color(Table, c) \wedge Color(can, c) \textbf{ then } Paint(Chair, can)\\
&\qquad \textbf{else if } Color(Chair, c) \wedge Color(can, c) \textbf{ then } Paint(Table, can)\\
&\qquad \textbf{else } [Paint(Chair, Can_1), Paint(Table, Can_1)]]]
\end{aligned}
$$

Figure 3.29: Here is an example of contingent planning

3.6 Knowledge Representation

Davide Lobba

In this chapter we address the question of what content to put into an agent's knowledge base and how to represent facts about the world.

Ontological Engineering

Definition. We define **ontological engineering** the representation of abstract concepts such as *Events*, *Time*, *Physical Objects* and *Beliefs*. The activity to build general-purpose ontologies should be applicable in any special-purpose domain.

Definition. upper ontology is the general framework, Figure 3.30.

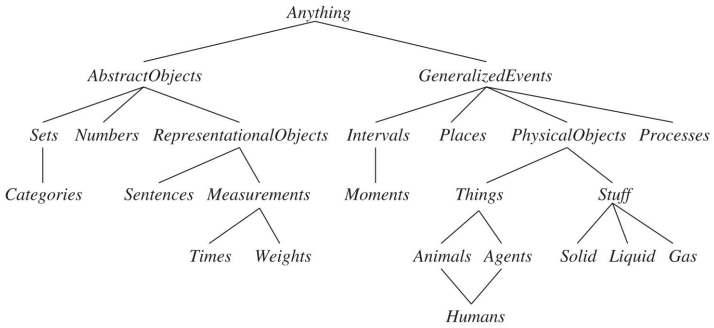


Figure 3.30: The upper ontology of the world. Each link indicates that the lower concept is a specialization of the upper one.

In this section, we present one general-purpose ontology that synthesizes ideas from past centuries. Two major characteristics of general-purpose ontologies distinguish them from collections of special-purpose ontologies:

- A general-purpose ontology should be applicable in more or less any special-purpose domain (with the addition of domain-specific axioms). This means that no representational issue can be finessed or swept under the carpet.
- In any sufficiently demanding domain, different areas of knowledge must be unified, because reasoning and problem solving could involve several areas simultaneously.

None of the top AI applications make use of a general ontology, in fact they all use special-purpose knowledge engineering and machine learning. There have been several attempts to build general-purpose ontologies such as CYC, DBpedia, TextRunner, OpenMind and so on, but they are not very successful so far.

Categories and Objects Knowledge representation requires the organisation of objects into **categories**. In particular, interaction with the world takes place at the level of individual objects and reasoning takes place at the level of categories. For example a shopper would normally have the goal of buying a basketball, rather than a particular basketball.
Categories also serve to make **predictions** about objects once they are classified.

An agent infers the presence of certain objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects.

There are two choices for representing categories in first-order logic: **predicates** and **objects**.
We can use the predicate *Basketball(b)*, or we can **reify**[18] the category as an object, *Basketball*.
To say that b is a member of the category of basketballs we could say *Member(b,Basketballs)*, which we will abbreviate as $b \in Basketballs$.

To say that b is a **subcategory** of *Balls* we say *Subset(Basketballs, Balls)*, which we abbreviate as $Basketballs \subset Balls$.
Notice that subcategory, subclass, and subset are interchangeable.
A subcategory **inherits** the properties of the category.

Example. We take for example:

$$\begin{aligned}&\text{If } \forall x\ (\ x \in Food \rightarrow Edible\ (\ x\ )\ )\\ &Fruit \subset Food\\ &Apples \subset Fruit \text{ then } \forall x\ (x \in Apple \rightarrow Edible\ (\ x\ )\ )\end{aligned}$$

A member **inherits** the properties of the category.

For example:
if $a \in Apples$, then $Edible(\ a\ )$

Subclass relations organize categories into a **taxonomic hierarchy** or **taxonomy**. The largest taxonomy organizes about 10 million living and extinct species into a single hierarchy.
First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members.

- An object is a member of a category.
  $BB9 \in Basketballs$
- A category is a subclass of another category.
  $Basketballs \subset Balls$
- All members of a category have some properties.
  $(\ x \in Basketballs) \Rightarrow Spherical(\ x)$

[18] Turning a proposition into an object is called **reification**.

- Members of a category can be recognized by some properties.
  $Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$
- A category as a whole has some properties.
  $Dogs \in DomesticatedSpecies$

Definition. We say that two or more categories are **disjoint** if they have no members in common.
$Disjoint(s) \iff (\forall\, c1, c2\ (c1 \in s \wedge c2 \in s \wedge c1 \neq c2) \Rightarrow Intersection(c1, c2) = \{\})$

For example:
$Disjoint(\{Animals, Vegetables\}), Disjoint(\{Insects, Birds, Mammals, Reptiles\})$

Definition. A set of categories s is an **exhaustive decomposition** of a category c iff all members of c are covered by categories in s.
$ExhaustiveDecomposition(s, c) \iff (\forall i\ (\ i \in c \iff (\exists c2\ (c2 \in s \wedge i \in c2)))$

For example:
$ExhaustiveDecomposition(\{Americans, Canadians, Mexicans\}, NorthAmericans)$

Definition. A exhaustive decomposition of disjoint sets is known as a **partition**.
$Partition(s, c) \iff (\ Disjoint(s) \wedge ExhaustiveDecomposition(s, c)\ )$

For example:
$Partition(\{Males, Females\}, Animals)$

Physical Composition

Definition. We use the general **PartOf** relation to say that one thing is part of another.
Objects can be grouped into **PartOf hierarchies**, reminiscent of the Subset hierarchy.

For example:
$PartOf(Bucharest, Romania)$
$PartOf(Romania, EasternEurope)$
$PartOf(EasternEurope, Europe)$
$PartOf(Europe, Earth)$

The **PartOf** relation is **transitive** and **reflexive**:
$PartOf(x, y) \wedge PartOf(y, z) \Rightarrow PartOf(x, z)$
So we can conclude that $PartOf(Bucharest, Europe)$

Categories of **composite objects** are often characterized by structural relations among parts. For example a biped is an object with exactly two legs attached to a body:

$$\begin{aligned} Biped(a) \Rightarrow\ & \exists l_1, l_2, b\ \ Leg(l_1) \wedge Leg(l_2) \wedge Body(b) \wedge \\ & PartOf(l_1, a) \wedge PartOf(l_2, a) \wedge PartOf(b, a) \wedge \\ & Attached(l_1, b) \wedge Attached(l_2, b) \wedge \\ & l_1 \neq l_2 \wedge [\forall l_3\ \ Leg(l_3) \wedge PartOf(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)]\,. \end{aligned}$$

Some useful relations are **PartPartition** and **BunchOf**.

Definition. An object is composed of the parts in its **PartPartition** and can be viewed as deriving some properties from those parts.

Definition. It is also useful to define composite objects with definite parts but no particular structure, which we will call a **bunch**.

For example, if the apples are *Apple1*, *Apple2*, and *Apple3*, then $BunchOf(Apple1, Apple2, Apple3)$ denotes the composite object with the three apples as parts (not elements).

Measurements Objects have height, mass, cost, and so on. The values that we assign for these properties are called **measures**, which we can represent by **unit functions**.

For example:
$Length(L1) = Inches(1.5) = Centimeters(3.81)$
Conversion between units is done by equating multiples of one unit to another:
$Centimeters(2.54 \times d) = Inches(d)$.

Measures can be used to **describe objects**:
$Diameter(Basketball12) = Inches(9.5)$
$Weight(BunchOf(\{Apple1, Apple2, Apple3\})) = Pounds(2)$

Other measures present more of a problem, because they have no agreed scale of values. For example beauty, deliciousness, difficulty and so on.
The most important aspect of measures is not the particular numerical values, but the fact that **measures can be ordered**.
Although measures are not numbers, we can still compare them, using an ordering symbol such as $>$.

$$e_1 \in Exercises \wedge e_2 \in Exercises \wedge Wrote(Norvig, e_1) \wedge Wrote(Russell, e_2) \Rightarrow Difficulty(e_1) > Difficulty(e_2).$$
$$e_1 \in Exercises \wedge e_2 \in Exercises \wedge Difficulty(e_1) > Difficulty(e_2) \Rightarrow ExpectedScore(e_1) < ExpectedScore(e_2).$$

These sorts of monotonic relationships among measures form the basis for the field of **qualitative physics**, a subfield of AI that investigates how to reason about physical systems without numerical computations.

Objects: Things and stuff

Definition. There is a significant portion of reality that seems to defy any obvious individuation, that is the division into distinct objects. We give this portion the generic name **stuff**.

The major distinction between *stuff* and *things* is that if I have some butter and an aardvark I can say there is one aardvark, but there is no obvious number of "butter-objects", because any part of a butter-object is also a butter-object. If we cut an aardvark in half, we do not get two aardvarks.
Linguists distinguish between **count nouns**, such as aardvarks, holes, and theorems, and **mass nouns**, such as butter, water, and energy.
We can have "an amount/quantity" of stuff. For example if $b \in butter$, b is an amount of butter.
Any part of a butter-object is also a butter-object: $b \in Butter \wedge PartOf(p, b) \Rightarrow p \in Butter$.
We can define sub-categories, which are stuff: $UnsaltedButter \subset Butter$.
Note that the category *PoundOfButter*, which includes as members all butter-objects weighing one pound, is not a kind of *stuff*. If we cut a pound of butter in half, we do not get two pounds of butter.
Some properties are **intrinsic**: they belong to the very substance of the object, rather than to the object as a whole. For example the color, density and so on.
On the other hand, their **extrinsic** properties, such as weight, length, shape, and so on, are not retained under subdivision.

Reasoning about Knowledge The agents we have constructed so far have beliefs and can deduce new beliefs. Yet none of them has any knowledge about *beliefs* or about *deduction*. Knowledge about one's own knowledge and reasoning processes is useful for controlling inference.
What we need is a **model of the mental objects** that are in someone's head (or something's knowledge base) and of **the mental processes** that manipulate those mental objects, in order to predict what other agents will do.
An agent can have **propositional attitudes** toward mental objects such as: *Believes*, *Knows*, *Wants*, and *Informs*.
The problem is that these attitudes do not behave like "normal" predicates due to the issues of **referential opacity** and **referential transparency**.

For example we try to assert that Lois Knows that Superman can fly:
$Knows(\,Lois, CanFly(\,Superman)\,)\,.$
We normally think of *CanFly(Superman)* as a sentence, but here it appears as a term, so we have to apply reification; making it a fluent.
The major problem is the **Referential Transparency** of FOL.
Since Superman is Clark Kent (but Lois doesn't know it!), FOL allows to conclude "Lois knows that Clark Kent can fly", which is a wrong inference.
$Superman = Clark \wedge Knows(\,Lois, CanFly(\,Superman)\,) \models_{\text{FOL}} Knows(\,Lois, CanFly(\,Clark)\,)$
This is a consequence of the fact that equality reasoning is built into logic.
To avoid this problem: FOL predicates transparent to equality reasoning.
$t = s \wedge P(\,s, ...) \models_{\text{FOL}} P(\,t, ...)$
We need a logic which is **opaque** to equality reasoning. **Modal logic** is designed to address this problem.
The **difference** between *regular logic* and *modal logic* is that *modal logic* includes special **modal operators** that take sentences (rather than terms) as arguments.

For example "A knows P" is represented with the notation $\mathbf{K}_{\text{A}}$P, where $\mathbf{K}$ is the modal operator for knowledge and P is a formula.
Another example: we can write "Lois knows Clark Kent knows if he is Superman or not" as
$\mathbf{K}_{\text{Lois}}(\mathbf{K}_{\text{Clark}} Identity(\,Superman, Clark) \vee \mathbf{K}_{\text{Clark}} \neg Identity(\,Superman, Clark)\,)$

Properties in all modal logics:

- $\mathbf{K}_{\text{A}}(\,P \wedge Q) \iff \mathbf{K}_{\text{A}} P \wedge \mathbf{K}_{\text{A}} Q$
- $\mathbf{K}_{\text{A}} P \vee \mathbf{K}_{\text{A}} Q \models \mathbf{K}_{\text{A}}(\,P \vee Q)$,but $\mathbf{K}_{\text{A}}(\,P \vee Q) \not\models \mathbf{K}_{\text{A}} P \vee \mathbf{K}_{\text{A}} Q$

The following axiom holds in all (normal) modal logics:

- $\mathbf{K} : (\,\mathbf{K}_{\text{A}} \phi \wedge \mathbf{K}_{\text{A}}(\,\phi \Rightarrow \psi)\,) \Rightarrow \mathbf{K}_{\text{A}} \psi$ (**distribution axiom**): "A is able to perform propositional inference".

The following axioms hold in some (normal) modal logics:

- $\mathbf{K}_{\text{A}} \varphi \Rightarrow \varphi$(**knowledge axiom**): "A knows only true facts".
- $\mathbf{K}_{\text{A}} \varphi \Rightarrow \mathbf{K}_{\text{A}} \mathbf{K}_{\text{A}} \varphi$(**positive-introspection axiom**): "If A knows fact φ, then it knows it knows it".
- $\neg \mathbf{K}_{\text{A}} \varphi \Rightarrow \mathbf{K}_{\text{A}} \neg \mathbf{K}_{\text{A}} \varphi$ (**negative-introspection axiom**): "If A doesn't know φ, then it knows it doesn't know it".

Semantics of Modal Logics A model (**Kripke model**) is a collection of possible worlds w_{i} in which possible worlds are connected in a graph by **accessibility relations**, in particular one relation for each distinct modal operator $\mathbf{K}_{\text{A}}$.
w_1 is accessible from w_0 with regard to $\mathbf{K}_{\text{A}}$ if everything which holds in w_1 is consistent with what A knows in w_0 (written as $Acc(\,\mathbf{K}_{\text{A}}, w_0, w_1)\,$)
$\mathbf{K}_{\text{A}} \varphi$ holds in w_0 iff φ holds in every world w_{i} accessible from w_0 and the more is known in w_0, the less worlds are accessible from w_0. It is important to remark that two possible worlds may differ also for what an agent knows there.
Different modal logics differ by different properties of $Acc(\mathbf{K}_{\text{A}}, ...)$:

- $\mathbf{K}_{\text{A}} \varphi \rightarrow \varphi$ holds iff $Acc(\mathbf{K}_{\text{A}}, ...)$ **reflexive**: $w \xmapsto{K_A} w$
- $\mathbf{K}_{\text{A}} \varphi \rightarrow \mathbf{K}_{\text{A}} \mathbf{K}_{\text{A}} \varphi$ holds iff $Acc(\mathbf{K}_{\text{A}}, ...)$ **transitive**: $w_0 \xmapsto{K_A} w_1$ and $w_1 \xmapsto{K_A} w_2 \Rightarrow w_0 \xmapsto{K_A} w_2$
- $\neg \mathbf{K}_{\text{A}} \varphi \rightarrow \mathbf{K}_{\text{A}} \neg \mathbf{K}_{\text{A}} \varphi$ holds iff $Acc(\mathbf{K}_{\text{A}}, ...)$ **euclidean**: $w_0 \xmapsto{K_A} w_1$ and $w_0 \xmapsto{K_A} w_2 \Rightarrow w_1 \xmapsto{K_A} w_2$

Notice that:

- $\mathbf{K}_{\text{A}} \neg P$: agent A knows that P does not hold
- $\neg \mathbf{K}_{\text{A}} P$: agent A does not know that P holds

- $\mathbf{K}_{\mathrm{A}}\neg P \models \neg\mathbf{K}_{\mathrm{A}}P$, but $\neg\mathbf{K}_{\mathrm{A}}P \not\models \mathbf{K}_{\mathrm{A}}\neg P$

Example.
- R: "the weather report says tomorrow will rain"
- I: "Superman's secret identity is Clark Kent."
- Ex: $\mathbf{K}_{\mathrm{Lois}}(\mathbf{K}_{\mathrm{Clark}}I \vee \mathbf{K}_{\mathrm{Clark}}\neg I)$: "Lois Knows that Clark Knows if he is Superman or not."

Superman knows his own identity and neither he nor Lois has seen the weather report, she knows Superman knows if he is Clark.
$(\neg\mathbf{K}_{\mathrm{Lois}}R \wedge \neg\mathbf{K}_{\mathrm{Lois}}\neg R) \wedge (\neg\mathbf{K}_{\mathrm{Superman}}R \wedge \neg\mathbf{K}_{\mathrm{Superman}}\neg R) \wedge \mathbf{K}_{\mathrm{Lois}}(\mathbf{K}_{\mathrm{Superman}}I \vee \mathbf{K}_{\mathrm{Superman}}\neg I)$

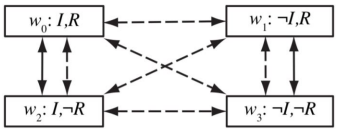


Figure 3.31: $\mathbf{K}_{\mathrm{Superman}}$ (solid arrows) and $\mathbf{K}_{\mathrm{Lois}}$ (dotted arrows)

Example.
- Ex: $\mathbf{K}_{\mathrm{Lois}}(\mathbf{K}_{\mathrm{Clark}}I \vee \mathbf{K}_{\mathrm{Clark}}\neg I)$: "Lois Knows that Clark Knows if he is Superman or not."

Superman knows his own identity and (b) Lois has seen the weather report, Superman has not, but he knows that Lois has seen it.
$(\mathbf{K}_{\mathrm{Lois}}R \vee \mathbf{K}_{\mathrm{Lois}}\neg R) \wedge (\neg\mathbf{K}_{\mathrm{Superman}}R \wedge \neg\mathbf{K}_{\mathrm{Superman}}\neg R)$

$\mathbf{K}_{\mathrm{Lois}}(\mathbf{K}_{\mathrm{Superman}}I \vee \mathbf{K}_{\mathrm{Superman}}\neg I) \wedge \mathbf{K}_{\mathrm{Superman}}(\mathbf{K}_{\mathrm{Lois}}R \vee \mathbf{K}_{\mathrm{Lois}}\neg R)$

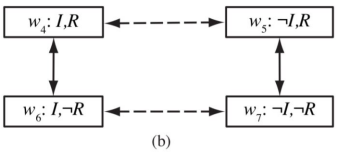


Figure 3.32: $\mathbf{K}_{\mathrm{Superman}}$ (solid arrows) and $\mathbf{K}_{\mathrm{Lois}}$ (dotted arrows)

Example.
- Ex: $\mathbf{K}_{\mathrm{Lois}}(\mathbf{K}_{\mathrm{Clark}}I \vee \mathbf{K}_{\mathrm{Clark}}\neg I)$: "Lois Knows that Clark Knows if he is Superman or not."

Superman knows his own identity and and (c) Lois may or may not have seen the weather report, Superman has not:
$((\neg\mathbf{K}_{\mathrm{Lois}}R \wedge \neg\mathbf{K}_{\mathrm{Lois}}\neg R) \vee (\mathbf{K}_{\mathrm{Lois}}R \vee \mathbf{K}_{\mathrm{Lois}}\neg R)) \wedge (\neg\mathbf{K}_{\mathrm{Superman}}R \wedge \neg\mathbf{K}_{\mathrm{Superman}}\neg R)$
$\mathbf{K}_{\mathrm{Lois}}(\mathbf{K}_{\mathrm{Superman}}I \vee \mathbf{K}_{\mathrm{Superman}}\neg I)$

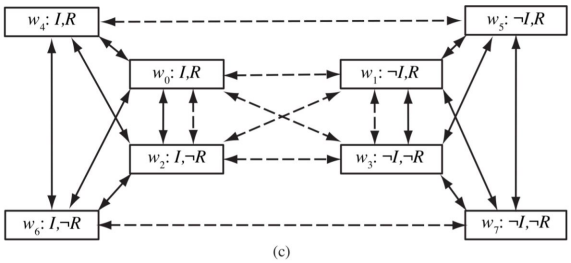


Figure 3.33: $\mathbf{K}_{\mathrm{Superman}}$ (solid arrows) and $\mathbf{K}_{\mathrm{Lois}}$ (dotted arrows)

Reasoning Systems for Categories Categories are the primary building blocks of large-scale knowledge representation schemes.
There are two closely related families of systems: **Semantic Networks** and **Description Logics** (**DLs**).

Semantic Networks **Semantic Networks** allow for representing individual objects, categories of objects, and relations among objects. It is a graph where nodes with a label corresponds to **concepts**, arcs labelled and directed correspond to **binary relations between concepts** (**roles**).
There are **generic concepts**, that correspond to *categories* and **individual concepts** that correspond to *individuals*.
Two relations are always present: **IS-A**, aka **SubsetOf**/**SubclassOf** (*subclass*) and **InstanceOf** aka **MemberOf** (*membership*).
The inference detection is straightforward.
One of the most important aspects of semantic networks is their ability to represent default values for categories.
Semantic Networks can handle exceptions without problems thanks to their ability to represent default values. Semantic Networks have limited expressive power, in fact they cannot represent negation, disjunction, nested function symbols and existential quantification.

For example in Fig 3.34: "HasMother" (double-boxed) is a relation between persons (individuals), and it means that $\forall x.(\, x \in Persons \rightarrow [\, \forall y.(\, HasMother(\, x, y) \,\rightarrow y \in FemalePersons)\,]\,)$
"Legs" (single-boxed) means: $\forall x.(\, x \in Persons \rightarrow Legs(\, x, 2)\,)$

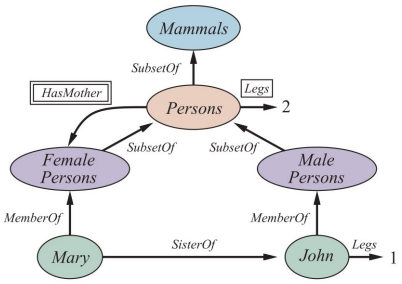


Figure 3.34: A semantic network with four objects (John, Mary, 1, and 2) and four categories.

In Semantic Networks the inheritance is conveniently implemented as link traversal. For example in Fig 3.35, if I want to how many legs has Clyde I can follow the INST-OF/IS-A chain until I find the property NLegs.

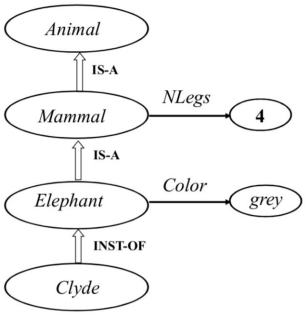


Figure 3.35: A semantic network

Semantic Networks allow only binary relations, but if we want to represent n-ary relations we can reify the proposition as an event belonging to an appropriate event category, as in Fig 3.36 on the facing page.

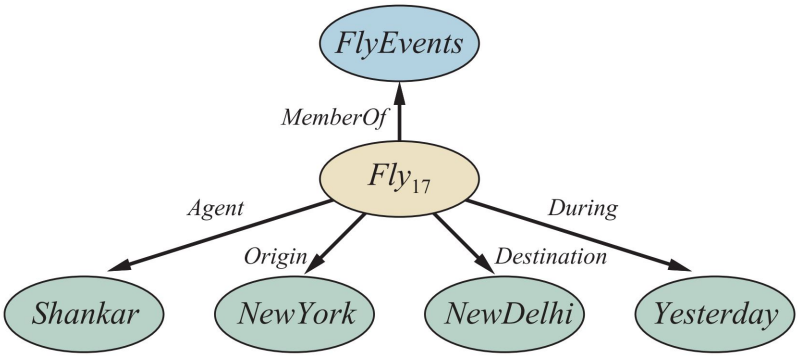


Figure 3.36: Representation of the logical assertion $Fly(Shankar, NewYork, NewDelhi, Yesterday)$

Description Logics Description logics are notations that are designed to make it easier to describe definitions and properties of categories.
The principal inference tasks for description logics are **subsumption**, **classification** and **consistency**.
Subsumption check if one category is the subset of another.
Classification check whether an object belongs to a category.
Consistency check if category membership criteria are satisfiable.
In description logics defaults and exceptions are lost.
Concepts correspond to **unary relations**. Operators s for the construction of complex concepts are and $\sqcap$, or $\sqcup$, not $\neg$, all $\forall$, some $\exists$, atleast $\geq$ n, atmost $\leq$ n, and so on...

For example: mothers (i.e., women who have children) of at least three female children
$Woman \sqcap \exists hasChildren.Person \sqcap \geq 3\ hasChild.Female$

Roles correspond to binary relations and can be combined with operators for constructing complex roles.

For example: $hasChildren \equiv hasSon \sqcup hasDaughter$

T-Boxes and A-Boxes **Terminologies** or **T-boxes** are sets of concept definitions ($C_1 \equiv C_2$) or concept generalizations ($C_1 \sqsubseteq C_2$)
For example: $Father \equiv Man$, $Woman \sqsubseteq Person$.
Assertions or **A-boxes** assert individuals as concept members $i : C$, where i is an individual and C is a concept, and individual pairs as relation members $< i, j >: R$, where i,j are individuals and R is a relation.
For example: $mary : Person$, $< john, mary >: hasChild$.

Woman	≡	Person ⊓ Female
Man	≡	Person ⊓ ¬ Woman
Mother	≡	Woman ⊓ ∃hasChild.Person
Father	≡	Man ⊓ ∃hasChild.Person
Parent	≡	Father ⊔ Mother
Grandmother	≡	Mother ⊓ ∃hasChild. Parent
MotherWithManyChildren	≡	Mother ⊓ ⩾ 3 hasChild .Person
MotherWithoutDaughter	≡	Mother ⊓ ∀hasChild.¬ Woman
Wife	≡	Woman ⊓ ∃hasHusband. Man

Figure 3.37: Example of T-Boxes (Logic $ALCN$)

Reasoning Services for DLs It is important the design and management of ontologies, especially consistency checking of concepts and the creation of hierarchies. Moreover, are necessary the relations between concepts of different ontologies and consistency of integrated hierarchies.
Queries are important because they determine whether facts are consistent with regard to ontologies, if individuals are instances of concepts, they retrieve individuals satisfying a query and they verify if a concept is more general than another.

For example: All the children of John are females. Mary is a child of John. Tim is a friend of professor Blake. Prove that Mary is a female.
$A = \{john : \forall hasChild.female, (john, mary) : hasChild, (blake, tim) : hasFriend, blake : professor\}$
Query: $mary : female$(or: is $A \sqcap mary : \neg female$ unsatisfiable?) $\Rightarrow$ yes.

Chapter 4

Uncertain Knowledge and Reasoning

4.1 Quantifying Uncertainty

Evelyn Turri

Acting under Uncertainty An agent needs to handle uncertainty, because it may never know for sure what state it is in now or where it will end up after a sequence of actions.
Most of the decisions taken by the agent are based on incomplete information, due to the fact that the environment can be partially observable or that the agent acts in a non deterministic way.
In a general problem the agent's knowledge cannot guarantee any outcomes, but it can provide some **degree of belief** that they will be achieved. *The rational decision depends on both the relative importance of various goals and the likelihood, and degree to which, they will be achieved.*
The agent's knowledge can at best provide only a degree of belief in the relevant sentences, and the best way to deal with degrees of belief is **probability theory**.

To make choices between the various actions, an agent must have **preferences** among the different possible **outcomes** of the various plans.

Definition. An **outcome** is a completely specified state.

So we use **utility theory** to represent preferences and reason quantitatively with them. This theory says that every state has a degree of usefulness, or utility, to an agent and that the agent will prefer states with higher utility.

Utilities are combined with probabilities in the **general theory of rational decisions**, so what is called **decision theory**:

$$\text{Decision Theory} = \text{Probability Theory} + \text{Utility Theory}$$

Definition (**Maximum Expected Utility (MEU)**)**.** An agent is rational if and only if it chooses the actions that yields the highest expected utility, averaged over all the possible outcomes of the action.

The agent is identical, at an abstract level, to the agent described in Chapter 2.2 an 3.1, that maintain a belief state reflecting the history of percepts to date. The primary difference is that the decision-theoretic agent's belief state represents not just the possibilities for world states but also their probability.

Basic Probability Notation In this paragraph we take a look at how elementary probability theory meets the needs of AI. Firstly we give some important definitions.

Definition. Probabilistic assertions state how likely possible worlds are.

Definition. The **sample space** is the set of all possible worlds, and it is mathematically defined with Ω. An element $\omega \in \Omega$ is called **sample point** or **event**.

The possible worlds are **mutually exclusive** and **exhaustive**. A factored representation for the possible worlds is defined with sets of *<variable, value>* pairs.
The variables in probability theory are *random variables.*

Definition. A **random variable** is a function that maps from the domain of possible worlds Ω to the set of possible values it can take on.

The variables can have *finite* or *infinite* ranges. They can be *discrete* or *continuous.*

Definition. A **probability model** is a sample space with an assignment $P(\omega)$ $\forall \omega \in \Omega$.

Axiom. The **basic axiom** of probability theory say that:

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

Definition. The **probability distribution** is an assignment of a probability for each possible value of the random variable X:

$$P(X = x_i) = \sum_{\omega \in X(\omega)} P(\omega) \tag{4.1}$$

For continuous variables, it is not possible to write out the entire distribution as a vector, because there are infinitely many values.

Definition. We can define the probability that a random variable takes on some value x as a parameterized function of x, usually called a **probability density function** (**pdf**).

A brief summary of the various probability:

P(X) **Unconditional or Prior probability**
It refers to degrees of belief in propositions *in the absence of any other information.*

P($X|Y$) **Conditional or Posterior probability**
It refers to degrees of belief in proposition *a given some evidence* Y.

P(X, Y) **Joint Probability Distribution**
It is the distribution for mutliple variables, and it denotes the probabilities of all combinations of the value of X and Y.

P(X, Y, Z) **Full Joint Probability Distribution**
It is the joint distribution for all the random variables.
A possible world is defined to be an assignment of values to all of the random variables under consideration.
A probability model is completely determined by the Full Joint Probability Distribution.

Remark

In the following chapters we use different notations:

1. P : for specific probability values
2. **P** : defines a **probability distribution** for the random variable, so in the case of a discrete variable it will be a vectors of probability values
3. X : for the distribution of a single variable
4. x : for a specific value of X
5. **X** : for the list of variables

Definition. Mathematically speaking, **conditional probabilities** are defined in terms of unconditional probabilities as follows, $\forall$ proposition a *and* b, holds (whenever $P(b) > 0$):

$$P(a|b) = \frac{P(a \wedge b)}{P(b)} \tag{4.2}$$

This definition can be rewritten in a form, called **product rule**, which is:

$$P(a \wedge b) = P(a|b)P(b) \tag{4.3}$$

The product rule holds also for whole distribution: $\mathbf{P}(X, Y) = \mathbf{P}(X|Y)\mathbf{P}(Y)$

Definition. An **event** A is any subset of Ω. It holds:

$$\mathbf{P}(A) = \sum_{\omega \in A} \mathbf{P}(w) \tag{4.4}$$

These events can be described by **propositions** in a formal language, and for each proposition the corresponding set contains just those possible worlds in which the proposition holds.
So we think a proposition a as the event A (set of sample points) where the proposition is *true*.

$$a \iff \text{"}A = true\text{"} \tag{4.5}$$

For any proposition ϕ, we have:

$$P(\phi) = \sum_{\omega \in \phi} P(w) \tag{4.6}$$

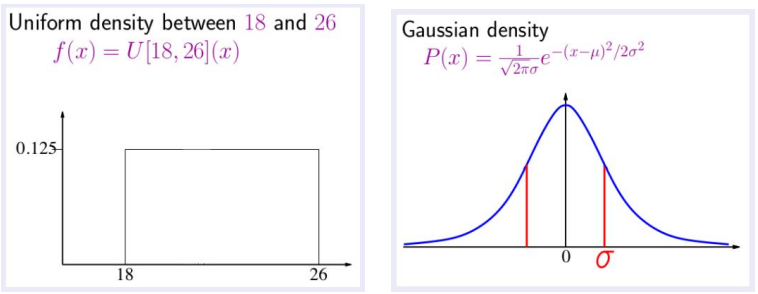
so we are summing the atomic events where ϕ is true.

Boolean Random Variables	
A Boolean random variable has the range *{true, false}*. An alternative range for Boolean Variables is the set $\{0, 1\}$, (the variable is said to have a Bernoulli distribution)	
$\boldsymbol{a}$	set of samples points where $A(\omega) = true$
$\boldsymbol{\neg a}$	set of samples points where $A(\omega) = false$
$\boldsymbol{a \wedge b}$	set of samples points where $A(\omega) = true \wedge B(\omega) = true$

Table 4.1: Boolean Random Variables

Continuous Variables	
$\mathbf{p}(\boldsymbol{x})$	**Probability Density Function** $\mathbf{p}(x) \in [0, 1]$ *s.t.* $\int_{-\infty}^{+\infty} \mathbf{p}(x)dx = 1$
$\mathbf{P}(\boldsymbol{x} \in [\boldsymbol{a}, \boldsymbol{b}])$	$\mathbf{P}(x \in [a, b]) = \int_a^b \mathbf{p}(x)dx$

Table 4.2: Continuous Variables



(a) Example 1 (b) Example 2

Figure 4.1: Examples of continuous variables

The basic axioms of probability imply certain relationships among the degree of belief that can be accorded to logically related propositions.

Probability of $\neg a$ $\quad \mathbf{P}(\neg \boldsymbol{a}) = \mathbf{1} - \mathbf{P}(\boldsymbol{a})$

$$\begin{aligned} P(\neg a) &= \textstyle\sum_{\omega \in \neg a} P(w) \\ &= \textstyle\sum_{\omega \in \neg a} P(w) + \sum_{\omega \in a} P(w) - \sum_{\omega \in a} P(w) \\ &= \textstyle\sum_{\omega \in \Omega} P(w) - \sum_{\omega \in a} P(w) \\ &= 1 - P(a) \end{aligned}$$

Inclusion-exclusion principle $\quad \mathbf{P}(\boldsymbol{a} \vee \boldsymbol{b}) = \mathbf{P}(\boldsymbol{a}) + \mathbf{P}(\boldsymbol{b}) - \mathbf{P}(\boldsymbol{a} \wedge \boldsymbol{b})$

The case where a holds, together with the cases where b holds, certainly cover all the cases where $a \wedge b$ holds, but summing the two sets of cases counts their intersection twice, so we need to subtract $\mathbf{P}(a \wedge b)$.

Chain Rule $\quad \mathbf{P}(\boldsymbol{X_1}, \cdots, \boldsymbol{X_n}) = \prod_{\boldsymbol{i=1}}^{\boldsymbol{n}} \mathbf{P}(\boldsymbol{X_i} | \boldsymbol{X_1}, \cdots, \boldsymbol{X_{i-1}})$

$$\begin{aligned} \mathbf{P}(X_1, \cdots, X_n) &= \mathbf{P}(X_1, \cdots, X_{n-1}) \mathbf{P}(X_n | X_1, \cdots, X_{n-1}) \\ &= \mathbf{P}(X_1, \cdots, X_{n-2}) \mathbf{P}(X_{n-1} | X_1, \cdots, X_{n-2}) \mathbf{P}(X_n | X_1, \cdots, X_{n-1}) \\ &= \cdots \\ &= \textstyle\prod_{i=1}^{n} \mathbf{P}(X_i | X_1, \cdots, X_{i-1}) \end{aligned}$$

We can briefly see the differences between *logic* and *probability*:

Logic	Probability
a	$P(a) = 1$
$\neg a$	$P(\neg a) = 1 - P(a)$
$a \to b$	$P(b\|a) = 1$
$\frac{(a, a \to b)}{b}$	$\frac{P(a)=1, P(b\|a)=1}{P(b)=1}$
$\frac{(a \to b, b \to c)}{a \to c}$	$\frac{P(b\|a)=1, P(c\|b)=1}{P(c\|a)=1}$

Table 4.3: Differences between logic and probability

Inference Using Full Joint Distribution In this paragraph we describe a simple method for *probabilistic inference*.

Definition. Probabilistic inference is the computation of posterior probabilities for query proposition given observe evidence.

To do this process, we use the full joint distribution as **knowledge base**.
There are different techniques to work with, in order to manipulate equations with probabilities.

The first rule is **marginalization**, or **summing out**, where we sum up the probabilities for each possible value of the other variables, thereby taking them out of the equation.
So we can write, for any sets of variables $\mathbf{Y}$ and $\mathbf{Z}$:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}, \mathbf{Z} = \mathbf{z}) \tag{4.7}$$

where $\sum_{\mathbf{z}}$ sums over all the possible combinations of values of the set of variables $\mathbf{Z}$.
Furthermore the notation $\mathbf{P}(\mathbf{Y}, \mathbf{Z} = \mathbf{z})$ can be rewritten as $\mathbf{P}(\mathbf{Y}, \mathbf{z})$.
Thanks to the product rule (Equation (4.3)) variant for marginalization is **conditioning**:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}|\mathbf{z})P(\mathbf{z}) \tag{4.8}$$

The rules above can be very useful for all kind of derivations involving probabilities. Usually we are interesting in computing *conditional probabilities*. These can be found by first using the Equation (4.2) to obtain an expression in terms of unconditional probabilities and then evaluating the expression from the full joint distribution.

Example. Suppose to have this type of table, so we have the full joint distribution for the $Toothache$, $Cavity, Catch$ world:

	$toothache$		$\neg toothache$	
	$catch$	$\neg catch$	$catch$	$\neg catch$
$cavity$	.108	.012	.072	.008
$\neg cavity$	.016	.064	.144	.576

Table 4.4: Example

Now suppose to compute the following probabilities:

- $P(cavity \vee toothache)$: using the Equation (4.6) we can find:

$$P(cavity \vee toothache) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064$$

- $\mathbf{P}(Toothache)$: by *marginalization* we can compute this **marginal probability**:

$$\mathbf{P}(Toothache) = \sum_{\mathbf{z} \in \{Catch, Cavity\}} \mathbf{P}(Toothache, \mathbf{z})$$

Computation
$\mathbf{P}(Toothache) =< P(toothache), P(\neg toothache) >$
$P(toothache) = 0.108 + 0.012 + 0.016 + 0.064 = 0.2$
$P(\neg toothache) = 1 - P(toothache) = 0.8$
$\mathbf{P}(Toothache) =< 0.2, 0.8 >$

- $P(\neg cavity|toothache) = \frac{P(\neg cavity \wedge toothache)}{P(toothache)} = \frac{0.016+0.064}{0.108+0.012+0.016+0.064} = 0.4$
- $P(cavity|toothache) = \frac{P(cavity \wedge toothache)}{P(toothache)} = \frac{0.108+0.012}{0.108+0.012+0.016+0.064} = 0.6$

Another important rule to work with probabilities is the **normalization** rule. It is usually used when we want to compute conditional probabilities.We can derive the general rule as follows:

$$\mathbf{P}(X|\mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) \tag{4.9}$$

where:

- $\alpha = \frac{1}{\mathbf{P}(\mathbf{E}=\mathbf{e})}$ (different α for different values of $\mathbf{e}$)
- X : single variable
- $\mathbf{E}$: list of evidence variables
- $\mathbf{e}$: list of observed values for the evidence variables

- $\mathbf{Y}$: remaining unobserved variables
- the summation is over all possible $\mathbf{y}$s, i.e., all possible combinations of values of the unobserved variables $\mathbf{Y}$
- $\mathbf{P}(X, \mathbf{e}, \mathbf{y})$: is simply a subset of probabilities from the full joint distribution

The full joint distribution in tabular form is a practical tool for building reasoning systems.

Independence and Conditional Independence In this paragraph we want to underline the connection between *independence* and *conditional independence*.

Definition. Two variables X and Y are called **independent** if and only if holds:

$$\mathbf{P}(X, Y) = \mathbf{P}(X)\mathbf{P}(Y) \iff \mathbf{P}(X|Y) = \mathbf{P}(X) \iff \mathbf{P}(X|Y) = \mathbf{P}(Y) \tag{4.10}$$

This type of independence can drastically reduce the number of entries in a table. For example, if we have a 32-element table we can decompose into one 8-element table and 4-element table.
Unfortunately, this independence is quite rare.

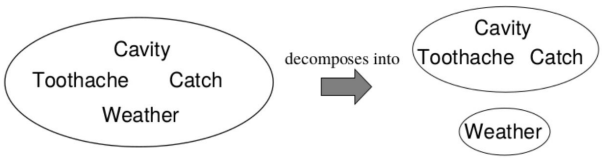


Figure 4.2: Examples of independent variables

Definition. Two variables X and Y are called **conditional independent** given $\mathbf{Z}$ if and only if holds:

$$\mathbf{P}(X, Y|\mathbf{Z}) = \mathbf{P}(X|\mathbf{Z})\mathbf{P}(Y|\mathbf{Z}) \iff \mathbf{P}(X, Y|\mathbf{Z}) = \mathbf{P}(X|\mathbf{Z}) \iff \mathbf{P}(Y|\mathbf{X}, \mathbf{Z}) = \mathbf{P}(Y|\mathbf{Z}) \tag{4.11}$$

Definition. A **naive Bayes model** is a probability model where the *effect* are conditionally independent, given the cause. So it holds:

$$\mathbf{P}(Cause, Effect_1, \cdots, Effect_n) = \mathbf{P}(Cause)\prod_{i=1}^{n}\mathbf{P}(Effect_i|Cause) \tag{4.12}$$

This reduces from $(2^{n+1} - 1)$ to $(2n - 1)$ entries.

Bayes' Rule We know from the product rule (Equation (4.3)) that:

$$P(a \wedge b) = P(a|b)P(b) \text{ and } P(a \wedge b) = P(b|a)P(b)$$

Equating the two right-hand sides and dividing by $P(a)$, we have:

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} \tag{4.13}$$

This equation is known as **Bayes' Rule**. The more general case of Bayes' rule for multivalued variables can be rewritten in the $\mathbf{P}$ notation as follows:

$$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)} \tag{4.14}$$

We can write also a more general version conditionalized on some background evidence $\mathbf{e}$:

$$\mathbf{P}(Y|X, \mathbf{e}) = \frac{\mathbf{P}(X|Y, \mathbf{e})\mathbf{P}(Y|\mathbf{e})}{\mathbf{P}(X|\mathbf{e})} \tag{4.15}$$

If we want to use *normalization*, the general Bayes' rule is as follow:

$$\mathbf{P}(Y|X) = \alpha\mathbf{P}(X|Y)\mathbf{P}(Y) \tag{4.16}$$

where $\alpha = \frac{1}{\mathbf{P}(x)}$ is the normalization constant.

We can underline two cases in which Bayes' rule is used:

Simple Case When we perceive as evidence the *effect* of some unknown *cause* and we would like to determine that *cause*

$$P(cause|effect) = \frac{P(effect|cause)P(cause)}{P(effect)}$$

where:

- $P(cause|effect)$ goes from *effect* to *cause*
- $P(effect|cause)$ goes from *cause* to *effect*

Combining Evidence When we are in the case of a Naive Bayes Model (Equation (4.12)). Conditional independence assertions can allow probabilistic systems to scale up; moreover, they are much more commonly available than absolute independence assertions.

Example : The Wumpus World Revisited We can combine these knowledge to solve probabilistic reasoning problems in the wumpus world. Uncertainty arises in the wumpus world because the agent's sensors give only partial information about the world. Pure logical inference can conclude nothing about which square is most likely to be safe, so a logical agent might have to choose randomly. A probabilistic agent can do much better than the logical agent.

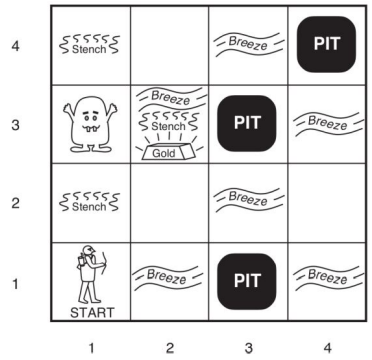


Figure 4.3: Examples of the Wumpus World

Given the wumpus world in Figure 4.3, our aim is to calculate the probability that each of the three squares in [1,3], [2,2], [3,1] contain a pit.

First we define:

- Two groups of variable:
  1. $P_{ij} = true \iff$ [i,j] contains a pit ("*causes*")
  2. $B_{ij} = true \iff$ [i,j] is breezy ("*effects*", consider only B_{11}, B_{12}, B_{21})
- Joint distribution : $\mathbf{P}(P_{11}, \ldots, P_{44}, B_{11}, B_{12}, B_{21})$
- Known facts (evidence) :
  - $b^* = \neg b_{11} \wedge b_{12} \wedge b_{21}$

– $p^* = \neg p_{11} \wedge \neg p_{12} \wedge \neg p_{21}$

Our queries are:

- $\mathbf{P}(P_{13}|p^*, b^*)$
- $\mathbf{P}(P_{22}|p^*, b^*)$
- $\mathbf{P}(P_{13}|p^*, b^*)$: which is symmetric

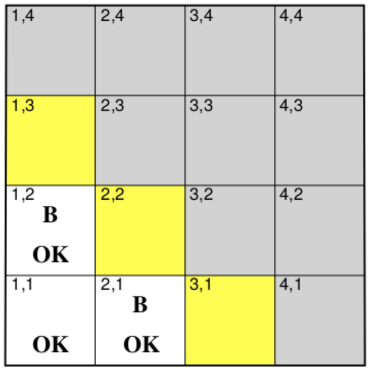


Figure 4.4: Admissible paths of the Wumpus World Example

Apply the product rule to the joint distribution $\mathbf{P}(P_{1,1}, \ldots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1}) = \mathbf{P}(B_{1,1}, B_{1,2}, B_{2,1}|P_{1,1}, \ldots, P_{4,4})\mathbf{P}(P_{1,1}, \ldots, P_{4,4})$. Pay attention that:

$$\mathbf{P}(B_{1,1}, B_{1,2}, B_{2,1}|P_{1,1}, \ldots, P_{4,4})$$

- Is 1 if one pit is adjacent to breeze;
- Is 0 otherwise.

And

$$\mathbf{P}(P_{1,1}, \ldots, P_{4,4})$$

means that pits are placed randomly except in (1,1), in fact:

$$\mathbf{P}(P_{1,1}, \ldots, P_{4,4}) = \prod_{i=1}^{4} \prod_{j=1}^{4} P(P_{i,j})$$

$$P(P_{i,j}) = \begin{cases} 0.2 & if\ (i,j) \neq (1,1) \\ 0 & otherwise \end{cases}$$

For example $\mathbf{P}(P_{1,1}, \ldots, P_{4,4}) = 0.2^3 \cdot 0.8^{15-3} \sim 0.00055$ if 3 pits.
Consider the case of $P_{1,3}$:

- The general form of query is $\mathbf{P}(\mathbf{Y}|\mathbf{E} = e) = \alpha\ P(\mathbf{Y}, \mathbf{E} = e) = \alpha\ \sum_h \mathbf{P}(\mathbf{Y}, \mathbf{E} = e, \mathbf{H} = H)$ where
  - $\mathbf{Y}$ is a query var;
  - $\mathbf{E}$, e are evidence variables/values;
  - $\mathbf{H}$, h are hidden variables/values.
- In our case $\mathbf{P}(P_{1,3}|p^*, b^*)$, such that the evidence is:
  - $b^* = \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1}$;
  - $p^* = \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1}$.
- We sum over the hidden variables:

  $$\mathbf{P}(P_{1,3}|p^*, b^*) = \alpha \sum_{unknown} \mathbf{P}(P_{1,3}|p^*, b^*, unknown)$$

  where *unknown* are all P_{ij}'s such that $(i, j) \notin \{(1, 1), (1, 2), (2, 1), (1, 3)\} \Rightarrow$ 4096 terms of the sum;

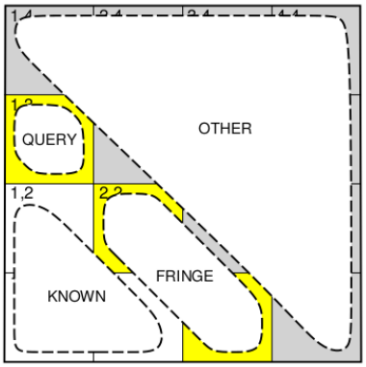


Figure 4.5: Conditional independence usage on the Wumpus World Example

- Grows exponentially in the number of hidden variables **H**! Thus, it is very inefficient.

We use conditional independence, the basic insight is that given the fringe squares (Figure 4.5), b^* is conditionally independent of the other hidden squares. Thus, $Unknown = Fringe \cup Other$.

$$\mathbf{P}(b^*|p^*, P_{1,3}, Unknown) = \mathbf{P}(b^*|p^*, P_{1,3}, Fringe, Others) = \mathbf{P}(b^*|p^*, P_{1,3}, Fringe)$$

And next manipulate the query into a form where this equation can be used. Consider that $\mathbf{P}(p^*, b^*) = P(p^*, b^*)$ is scalar; you can use as a normalization constant.

$$\begin{aligned}
\mathbf{P}(P_{1,3}|p^*, b^*) &= \mathbf{P}(P_{1,3}, p^*, b^*)/\mathbf{P}(p^*, b^*) \\
&= \alpha\ \mathbf{P}(P_{1,3}, p^*, b^*) \\
&= \alpha \sum_{unknown} \mathbf{P}(P_{1,3}, unknown, p^*, b^*) \\
&\quad \textit{Sum over the unknowns} \\
&= \alpha \sum_{unknown} \mathbf{P}(b^*|P_{1,3}, p^*, unknown)\ \mathbf{P}(P_{1,3}, p^*, unknown) \\
&\quad \textit{Use the product rule} \\
&= \alpha \sum_{fringe} \sum_{other} \mathbf{P}(b^*|\ p^*, P_{1,3}, fringe, other)\ \mathbf{P}(P_{1,3}, p^*, fringe, other) \\
&\quad \textit{Separate unknown into fringe and other} \\
&= \alpha \sum_{fringe} \sum_{other} \mathbf{P}(b^*|\ p^*, P_{1,3}, fringe)\ \mathbf{P}(P_{1,3}, p^*, fringe, other) \\
&\quad b^* \textit{ is conditionally independent of other given fringe} \\
&= \alpha \sum_{fringe} \mathbf{P}(b^*|\ p^*, P_{1,3}, fringe) \sum_{other} \mathbf{P}(P_{1,3}, p^*, fringe, other) \\
&\quad \textit{Move } \boldsymbol{P}(b^*|p^*, P_{1,3}, fringe) \textit{ outward} \\
&= \alpha \sum_{fringe} \mathbf{P}(b^*|\ p^*, P_{1,3}, fringe) \sum_{other} \mathbf{P}(P_{1,3})\ P(p^*)\ P(fringe)\ P(other) \\
&\quad \textit{All of the pit locations are independent} \\
&= \alpha\ P(p^*)\ \mathbf{P}(P_{1,3}) \sum_{fringe} \mathbf{P}(b^*\ |\ p^*, P_{1,3}, fringe)\ P(fringe) \sum_{other} P(other) \\
&\quad \textit{Move } P(p^*), \boldsymbol{P}(P_{1,3}) \textit{ and } P(fringe) \textit{ outward} \\
&= \alpha\ P(p^*)\ \mathbf{P}(P_{1,3}) \sum_{fringe} \mathbf{P}(b^*\ |\ p^*, P_{1,3}, fringe)\ P(fringe) \\
&\quad \textit{Remove } \textstyle\sum_{other} P(other) \textit{ because it equals 1} \\
&= \alpha'\ \mathbf{P}(P_{1,3}) \sum_{fringe} \mathbf{P}(b^*\ |\ p^*, P_{1,3}, fringe)\ P(fringe) \\
&\quad P(p^*) \textit{ is scalar, so make it part of the normalization constant}
\end{aligned}$$

We have obtained $\mathbf{P}(P_{1,3}|p^*,\ b^*) = \alpha'\ \mathbf{P}(P_{1,3})\ \sum_{fringe}\ \mathbf{P}(b^*|p^*,\ P_{1,3},\ fringe)\ P(fringe)$. And we know that $\mathbf{P}(P_{1,3}) = \langle 0.2, 0.8\rangle$. We can compute the normalization coefficient α' afterwards, for now we concentrate on $\sum_{fringe}\ \mathbf{P}(b^*|p^*,\ P_{1,3},\ fringe)\ P(fringe)$, it gives only 4 possible fringes:

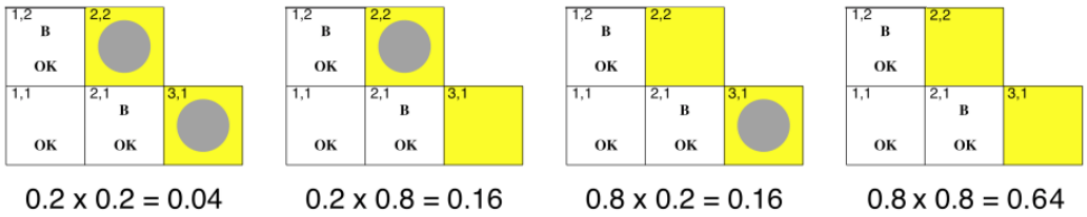


Figure 4.6: The four possible fringes

We can start by rewriting as two separate equations:

1. $$\mathbf{P}(p_{1,3}|p^*, b^*) = \alpha' P(p_{1,3}) \sum_{fringe} \mathbf{P}(b^*|p^*, p_{1,3}, fringe) P(fringe)$$

2. $$\mathbf{P}(\neg p_{1,3}|p^*, b^*) = \alpha' P(\neg p_{1,3}) \sum_{fringe} \mathbf{P}(b^*|p^*, \neg p_{1,3}, fringe) P(fringe)$$

For each of them, $P(b^*|\dots)$ is 1 if the breezes occur, 0 otherwise:

1. $$\sum_{fringe} \mathbf{P}(b^*|p^*, p_{1,3}, fringe) P(fringe) = 1 \cdot 0.01 + 1 \cdot 0.16 + 1 \cdot 0.16 + 0 \cdot 0.64 = 0.36$$

2. $$\sum_{fringe} \mathbf{P}(b^*|p^*, \neg p_{1,3}, fringe) P(fringe) = 1 \cdot 0.01 + 1 \cdot 0.16 + 0 \cdot 0.16 + 0 \cdot 0.64 = 0.2$$

Thus, we can continue our computation:

$$\begin{aligned}
\mathbf{P}(P_{1,3}|p^*, b^*) &= \alpha' \mathbf{P}(P_{1,3}) \sum_{fringe} \mathbf{P}(b^*|p^*, P_{1,3}, fringe) P(fringe) \\
&= \alpha' \langle 0.2, 0.8 \rangle \langle 0.36, 0.2 \rangle \\
&= \alpha' \langle 0.072, 0.16 \rangle \\
&\text{After a normalization such that } \alpha' \approx 4.31 \\
&\approx \langle 0.31, 0.69 \rangle
\end{aligned}$$

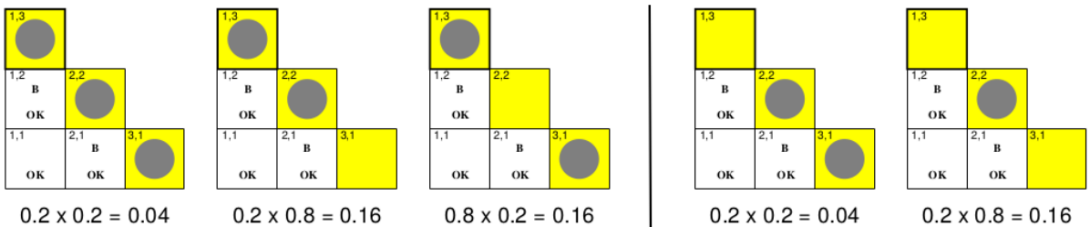


Figure 4.7: The final computations for the Wumpus World Example

4.2 Probabilistic Reasoning

Andrea Bonomi, Evelyn Turri

We introduce a way to represent the relationships explained in the previous chapter, explicitly in the form of *Bayesian Networks*.

Bayesian Networks

Definition. A **Bayesian network** is a directed acyclic graph (DAG), where:

1. Each node represents a random variable (discrete or continuous)
2. Directed arcs connect pairs of nodes: $X \rightarrow Y$ (X is a parent of Y)
3. Each node X_i has associated probability information $\mathbf{P}(X_i|Parents(X_i))$, that quantifies the effect of the parents on the node using a finite number of **parameters**

The **topology** of the network specifies the *conditional independence relationships* that hold in the domain, in a way that will be made shortly.
The intuitive meaning of an arrow is typically that X has a *direct influence* on Y, which suggests that causes should be parents of effects. So if we have no arcs between X and Y, it means that the two variables are independent. An example of these relationships can be:

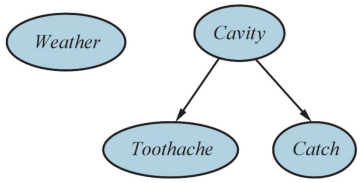


Figure 4.8: A simple Bayesian network in which $Weather$ is independent of the other three variables and $Toothache$ and $Catch$ are conditionally independent, given $Cavity$.

Once the topology of the Bayes net is laid out, we need only to specify the local probability information for each variable, in the form of a conditional distribution given its parents. We can define:

- The *full joint distribution* for all the variables by the topology and the local probability information.
- The *local probability information* (*conditional probability for each node* X_i) by the **conditional probability table** (**CPT**)

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 variables are $Burglary$, $Earthquake$, $Alarm$, $JohnCalls$, $MaryCalls$. A network topology reflects "causal" knowledge:

- A burglar can set the alarm off;
- An earthquake can set the alarm off;
- The alarm can cause Mary to call;
- The alarm can cause John to call.

CPTs:

- Alarm set off if burglar is in: 94% of cases;
- Alarm set off if hearthquake in 29% of cases;
- False alarm set off in 0.1% of cases.

Important to remember for exercises: in CPTs like $\mathbf{P}(A|B)$, only $P(a|B)$ is reported, because

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

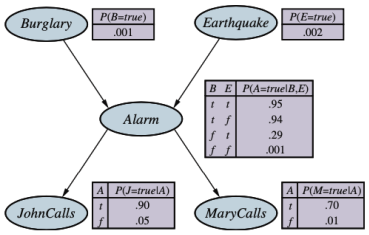


Figure 4.9: Bayes Network Example

In most domains, it is reasonable to suppose that each random variable X_i is directly influenced by only a small number k_i of other variables, called parents of X_i ($parents(X_i)$).

A CPT for Boolean X_i with k_i Boolean parents has:

- 2^{k_i} rows for the combinations of parent values
- each row requires one number p for $P(X_i = true)$ ($P(X_i = false) = 1 - P(X_i = true)$)

If each variable has no more than k parents, the complete network requires $O(n \cdot 2^k)$ numbers, which is linear; while a full joint distribution requires $2^n - 1$ numbers, which is exponential.

Semantics of Bayesian Networks Given a Bayesian net, we have two different ingredients:

Syntax	Consists of a directed acyclic graph with some local probability information attached to each node.
Semantics	Defines how the syntax corresponds to a joint distribution over the variables of the network.

We have two types of semantics:

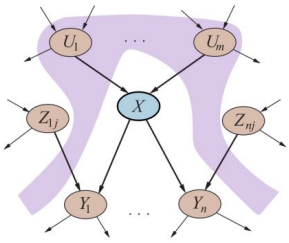
Global Semantic Defines the full joint distribution as the product of the local conditional distributions

$$\mathbf{P}(X_1, \ldots, X_n) = \prod_{i=1}^{n} \mathbf{P}(X_i | parents(X_i)) \tag{4.17}$$

Note that if X_i has no parents, the conditional distributions reduce to prior probability $\mathbf{P}(X_i)$.

Local Semantic
Each node is conditionally independent of its nondescendants $(Z_{1j}, \ldots, Z_{nj})$ given its parents $(U_1, \ldots, U_m)$:

$$\mathbf{P}(X | U_1, \ldots, U_m, Z_{1j}, \ldots, Z_{nj}) = \mathbf{P}(X | U_1, \ldots, U_m) \tag{4.18}$$



Theorem. Local semantics holds $\iff$ **global semantics** holds, so :

$$\mathbf{P}(X_1, \ldots, X_n) = \prod_{i=1}^{n} \mathbf{P}(X_i | parents(X_i))$$

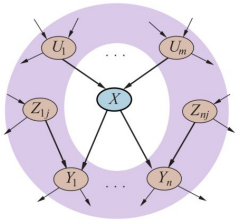
Example. (Global Semantic) Consider Figure 4.9. If we apply global semantics (Equation ((4.17))) we obtain that:

$$\begin{aligned} P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) &= P(j \mid m \wedge a \wedge \neg b \wedge \neg e)\ P(m \mid a \wedge \neg b \wedge \neg e)\ P(a \mid \neg b \wedge \neg e)\ P(b \mid \neg e)\ P(\neg e) \\ &= P(j \mid a)\ P(m \mid a)\ P(a \mid \neg b \wedge \neg e)\ P(\neg b)\ P(\neg e) \\ &= 0.9 \cdot 0.7 \cdot 0.001 \cdot 0.999 \cdot 0.998 \\ &\simeq 0.00063 \end{aligned}$$

Example. (Local Semantic) Consider Figure 4.9. An example of local semantics is: *JohnCalls* is independent of *Burglary*, *Earthquake*, and *MaryCalls* given the value of *Alarm*.
Thus: $\mathbf{P}(JohnCalls \mid Alarm, Burglary, Earthquake, MaryCalls) = \mathbf{P}(JohnCalls \mid Alarm)$.

Furthermore we have another important *independence rule*, which is the **Markov Blanket**.

Markov Blanket
In a Bayesian Network, each node is conditionally independent of all others given its *parents* $(U_1, \ldots, U_m)$ + *children* $(Y_1, \ldots, Y_n)$ + *children's parents* $(Z_{1j}, \ldots, Z_{nj})$ - that is, given its Markov Blanket.



$$\mathbf{P}(X|U_1, \ldots, U_m, Y_1, \ldots, Y_n, Z_{1j}, \ldots, Z_{nj}, W_1, \ldots, W_k) = \mathbf{P}(X|U_1, \ldots, U_m, Y_1, \ldots, Y_n, Z_{1j}, \ldots, Z_{nj})$$
$$\forall X, \text{ with } W_1, \ldots, W_k \text{ all other nodes} \tag{4.19}$$

Example. (Markov Blanket) Consider Figure 4.9. An example of Markov Blanket is: *Burglary* is independent of *JohnCalls* and *MaryCalls*, given *Alarm* and *Earthquake*.
Thus: $\mathbf{P}(Burglary \mid Alarm, Earthquake, JohnCalls, MaryCalls) = \mathbf{P}(Burglary \mid Alarm, Earthquake)$.

Constructing Bayesian Networks Given a set of random variables:

1. Choose an ordering $\{X_1, \ldots, X_n\}$
   - in principle, any ordering will work (but some may cause blowups)
   - general rule: follow causality, $X \prec Y$ if $X \in causes(Y)$
2. For $i = 1$ to n do:
   (a) Add X_i to the network
   (b) As $parents(X_i)$ choose a subset of $\{X_1, \ldots, X_{i-1}\}$ s.t. $\mathbf{P}(X_i|parents(X_i)) = \mathbf{P}(X_i|X_1, \ldots, X_{i-1})$

These two guarantees the global semantics by construction.

Example. Now we want to construct a Bayesian Network. Suppose we choose the ordering {*MaryCalls*, *JohnCalls*, *Alarm*, *Burglary*, *Earthquake*} (non-causal ordering). The process of constructing goes as follows:

1. Adding *MaryCalls*: No parents;
2. Adding *JohnCalls*: If Mary calls, that probably means the alarm has gone off, which makes it more likely that John calls. Therefore, *JohnCalls* needs *MaryCalls* as a parent;
3. Adding *Alarm*: Clearly, if both call, it is more likely that the alarm has gone off than if just one or neither calls, so we need both *MaryCalls* and *JohnCalls* as parents;
4. Adding *Burglary*: If we know the alarm state, then the call from John or Mary might give us information about our phone ringing or Mary's music, but not about burglary: $\mathbf{P}(Burglary \mid Alarm, JohnCalls, MaryCalls) = \mathbf{P}(Burglary \mid Alarm)$. Hence we need just *Alarm* as parent.

5. Adding *Earthquake*: If the alarm is on, it is more likely that there has been an earthquake. (The alarm is an earthquake detector of sorts.) But if we know that there has been a burglary, then that explains the alarm, and the probability of an earthquake would be only slightly above normal. Hence, we need both *Alarm* and *Burglary* as parents.

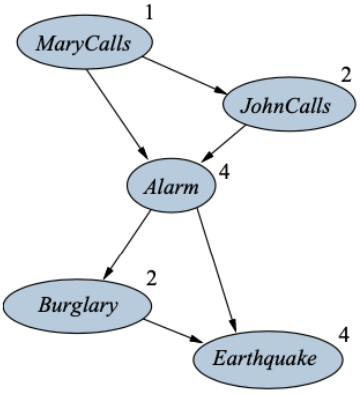


Figure 4.10: Constructed Bayesian Network Example

CPT grow exponentially with the number of parents. But if the causes do not interact we can use a **noisy-OR distribution**, which is the generalization of logical or.
So now assume:

- Parents $U_1, \ldots, U_k$ include all causes
- Independent failure probability $q_i = P(\neg x \,|\, U_i \wedge \bigwedge_{j \neq i} U_j)$ for each cause U_i :
  $P(\neg x \,|\, U_1, \ldots, U_j, \neg U_{j+1}, \ldots, \neg U_k) = \prod_{i=1}^{j} q_i$

  then the number of parameters is linear in the number of parents!

Cold	*Flu*	*Malaria*	$P(fever\|\cdot)$	$P(\neg fever\|\cdot)$
F	F	F	0.0	1.0
F	F	T	0.9	0.1
F	T	F	0.8	0.2
F	T	T	0.98	$0.02 = 0.2 \times 0.1$
T	F	F	0.4	0.6
T	F	T	0.94	$0.06 = 0.6 \times 0.1$
T	T	F	0.88	$0.12 = 0.6 \times 0.2$
T	T	T	0.988	$0.012 = 0.6 \times 0.2 \times 0.1$

Table 4.5: A complete conditional probability table for $\mathbf{P}(\textit{Fever} \mid \textit{Cold}, \textit{Flu}, \textit{Malaria})$

Exact Inference with Bayesian Networks The basic task for any probabilistic inference system is to compute the posterior probability distribution for a set of **query variables**, given some observed **event**. To simplify the presentation, we will consider only one query variable at a time (the algorithm can be easily extended to queries with multiple variables). So given :

- X : the query variable (we assume one for simplicity)
- $\mathbf{E}$: the set of evidence variables $E_1, \ldots, E_m$
- $\mathbf{E}/\mathbf{e}$: the set of evidence values $e_1, \ldots, e_m$ ($\mathbf{e}$ is a particular observed event)
- The set of unknown variables (aka hidden variables) $Y_1, \ldots, Y_l$ and unknown values $y_1, \ldots, y_l$
- $\mathbf{X} = X \cup \mathbf{E} \cup \mathbf{Y}$

A typical query asks for the posterior probability distribution $\mathbf{P}(X|\mathbf{e})$.

There are two types of exact inference, by *enumeration* and by *variable elimination*.

Inference by Enumeration We know from the previous Chapter, that any conditional probability can be computed by summing terms from the full joint distribution. More specifically:

$$\mathbf{P}(X|\mathbf{e}) = \alpha\mathbf{P}(X,\mathbf{e}) = \alpha\sum_{\mathbf{y}}\mathbf{P}(X,\mathbf{e},\mathbf{y})$$

Thanks to the Bayes bet we have a complete representation of the full joint distribution. Indeed terms as $\mathbf{P}(X,\mathbf{e},\mathbf{y})$ can be rewritten as product of prior and conditional probabilities according to the Bayesian Network.

Now we show the ENUMERATION-ASK algorithm with Algorithm (19). It evaluates the expressions trees using depth-first, left-to-right recursion. In space complexity is only linear in the number of variables, because the algorithm sums over the full joint distribution without ever constructing it explicitly. Unfortunately, its time complexity for a network with n Boolean variables is always $O(2^n)$.

Algorithm 19 Enumeration Algorithm for Exact Inference

function *Enumeration-Ask*(X, e, bn) **return** a distribution over X, in other words: $\mathbf{P}(X|e)$
  X is the query variable
  e represents the observed values for variables E
  bn is a Bayes net with variables var

  $Q(X) \leftarrow$ a distribution over X, initially empty
  for each value x_i of X **do**
    $Q(x_i) \leftarrow$ *Enumerate-All*($vars$, e_{x_i}) ▷ where e_{x_i} is e extended with $X = x_i$
  return *Normalize*($Q(X)$)

function *Enumerate-All*($vars$, e) **return** a real number, in other words, computes $\mathbf{P}(x_i, Y, e)$
  if *IsEmpty*($vars$) **then return** 1.0
  $V \leftarrow$ *First*($vars$)
  if V is an evidence variable with value v in e **then**
    return $\mathbf{P}(v \mid parents(V)) \times$ *Enumerate-All*(*Rest*($vars$), e)
  else
    return $\sum_v \mathbf{P}(v \mid parents(V)) \times$ *Enumerate-All*(*Rest*($vars$), e_v)
      ▷ where e_v is e extended with $V = v$

Example. (Inference by Enumeration) Consider the query:
$\mathbf{P}(Burglary \mid JohnCalls = true, MaryCalls = true)$, we compute P for $b = true$ and $b = false$:

$$
\begin{aligned}
&P(b|j,m) = \alpha P(b) \sum_e P(e) \sum_a P(a|b,e) P(j|a) P(m|a) = \alpha \cdot 0.00059224 \\
&P(\neg b|j,m) = \alpha P(\neg b) \sum_e P(e) \sum_a P(a|\neg b,e) P(j|a) P(m|a) = \alpha \cdot 0.0014919 \qquad (4.20) \\
&\text{Thus, } P(B|j,m) = \alpha \cdot \langle 0.00059224, 0.0014919 \rangle \xrightarrow{normalized} \langle 0.284, 0.716 \rangle
\end{aligned}
$$

Leads to:

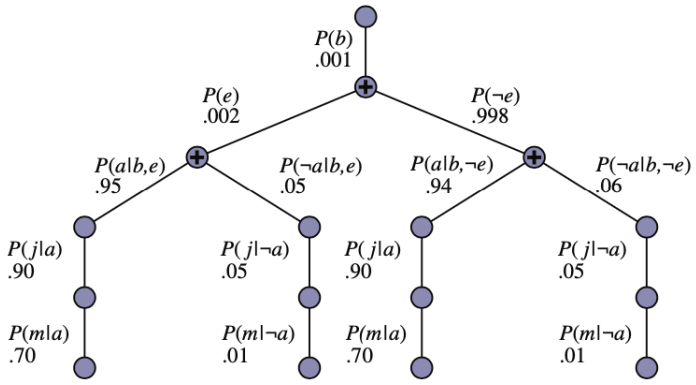


Figure 4.11: Structure of Equation above

A problem is that enumeration can be inefficient when it deals with repeated computations, due to the fact that there are some *repeated subexpressions*, which are computed more than one time.
In the previous Example we can notice that the products $P(j|a)P(m|a)$ and $P(j|\neg a)P(m|\neg a)$ are computed twice, once for each value of E.

Inference by Variable Elimination The Enumeration algorithm can be improve by eliminating repeated calculations underlined in the previous paragraph. The idea behind is to *do the calculation once and save the results for later use*, which is a kind of dynamic programming.
Variable Elimination algorithm works by evaluating such as the first Equation in (4.20) in *right-to-left* order, intermediate results are stored, and summations over each variable are done only for those portions of the expression that depend on the variable.

For this algorithm we require two **basic computational operations**. Suppose to have three **factors**: $\mathbf{f}(X_1, \ldots, X_j, Y_1, \ldots, Y_k), \mathbf{g}(Y_1, \ldots, Y_k, Z_1, \ldots, Z_l), \mathbf{h}(X_1, \ldots, X_j, Y_1, \ldots, Y_k, Z_1, \ldots, Z_l)$, then we show the basic operations with them:

Factor summation The factors must have the same arguments values, so for example to sum out X from $\mathbf{h}(X, Y, Z)$, we write:

$$\sum_x \mathbf{h}(X, Y, Z) = \mathbf{h}(x, Y, Z) + \mathbf{h}(\neg x, Y, Z)$$

Each factor is a matrix indexed by the values of its arguments variables.
So we have to take in consideration the standard matrix summation:

$$\begin{bmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ \dots & \dots & \dots & \dots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{21} & \dots & b_{n1} \\ \dots & \dots & \dots & \dots \\ b_{1n} & b_{2n} & \dots & b_{nn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} + b_{11} & a_{21} + b_{21} & \dots & a_{n1} + b_{n1} \\ \dots & \dots & \dots & \dots \\ a_{1n} + b_{1n} & a_{2n} + b_{2n} & \dots & a_{nn} + b_{nn} \end{bmatrix}$$

Pointwise product The pointwise product of two factors $\mathbf{f}$ and $\mathbf{g}$ yields a new factor $\mathbf{h}$ whose variables are the *union* of the variables in $\mathbf{f}$ and $\mathbf{g}$ and whose elements are given by the product of the corresponding elements in the two factors.

$$\mathbf{f}(X_1, \dots, X_j, Y_1, \dots, Y_k) \times \mathbf{g}(Y_1, \dots, Y_k, Z_1, \dots, Z_l) = \mathbf{h}(X_1, \dots, X_j, Y_1, \dots, Y_k, Z_1, \dots, Z_l)$$

Note that the pointwise product has 2^{j+k+l} entries and the size of a factor is exponential in the number of variables.
An example is given by:

$$f_J(A) \times f_M(A) = \begin{bmatrix} P(j|a) \\ P(j|\neg a) \end{bmatrix} \times \begin{bmatrix} P(m|a) \\ P(m|\neg a) \end{bmatrix} = \begin{bmatrix} P(j|a)\ P(m|a) \\ P(j|\neg a)\ P(m|\neg a) \end{bmatrix}$$

An important property that brings together the two basic operations is that any factor that does *not* depend on the variable to be summed out can be moved outside the summation.

$$\sum_x \mathbf{f}(X, Y) \times \mathbf{g}(Y, Z) = \mathbf{g}(Y, Z) \times \sum_x \mathbf{f}(X, Y)$$

Example. (Basic operations) $f_3(A, B, C) = f_1(A, B) \times f_2(B, C)$. Summing out one variable we obtain:

$$\begin{aligned} f(B, C) &= \textstyle\sum_a f_3(A, B, C) = f_3(a, B, C) + f_3(\neg a, B, C) \\ &= \begin{bmatrix} 0.06 & 0.24 \\ 0.42 & 0.28 \end{bmatrix} + \begin{bmatrix} 0.18 & 0.72 \\ 0.06 & 0.04 \end{bmatrix} \\ &= \begin{bmatrix} 0.24 & 0.96 \\ 0.48 & 0.32 \end{bmatrix} \end{aligned}$$

A	B	$f_1(A, B)$	B	C	$f_2(B, C)$	A	B	C	$f_3(A, B, C)$
T	T	0.3	T	T	0.2	T	T	T	$0.3 \times 0.2 = 0.06$
T	F	0.7	T	F	0.8	T	T	F	$0.3 \times 0.8 = 0.24$
F	T	0.9	F	T	0.6	T	F	T	$0.7 \times 0.6 = 0.42$
F	T	0.1	F	F	0.4	T	F	F	$0.7 \times 0.4 = 0.28$
						F	T	T	$0.9 \times 0.2 = 0.18$
						F	T	F	$0.9 \times 0.8 = 0.72$
						F	F	T	$0.1 \times 0.6 = 0.06$
						F	F	F	$0.1 \times 0.4 = 0.04$

Table 4.6: Variable Elimination: Factor Summation Example

Algorithm 20 Variable Elimination Algorithm

function *Elimination-Ask*(X, e, bn) **return** a distribution over X, in other words: $\mathbf{P}(X|e)$
  X is the query variable
  e represents the observed values for variables E
  bn is a Bayesian network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$

  $factors \leftarrow [\,]$
  for each var **in** $Order(bn.Vars)$ **do**
    $factors \leftarrow [Make\text{-}Factor(var, e) \mid factors]$
    if var is a hidden variable **then**
      $factors \leftarrow Sum\text{-}Out(var, factors)$

  return $Normalize(Pointwise\text{-}Product(factors))$