Artificial Intelligence

Prof. Dr. Jürgen Dix

Computational Intelligence Group
TU Clausthal
Clausthal, SS 2013

Time and place: Tuesday and Wednesday 10–12 in Multimediahörsaal (Tannenhöhe)
Exercises: See schedule (7 exercises in total).

Website

http://www.in.tu-clausthal.de/index.php?id=cig_ki13

Visit regularly!

There you will find important information about the lecture, documents, exercises et cetera.

Organization: Exercise: F. Schlesinger, M. Janßen
Exam: tba

History

This course evolved over the last 16 years. It was first held in Koblenz (WS 95/96, WS 96/97), then in Vienna (Austria, WS 98, WS00) , Bahia Blanca (Argentina, SS 98, SS 01) and Clausthal (SS 04–12).
Chapters 1–4, 9 are based on the seminal book of Russel/Norvig: Artificial Intelligence.

Many thanks to Nils, Tristan, Wojtek and Federico for the time they invested in crafting slides and doing the exercises. Their help over the years improved the course a lot.

Lecture Overview

1. Introduction (2 lectures)
2. Searching (3 lectures)
3. Supervised Learning (3 lectures)
4. Neural Nets (1 lecture)
5. Knowledge Engineering: SL (3 lectures)
6. Hoare Calculus (2 lectures)
7. Knowledge Engineering: FOL (2 lectures)
8. Knowledge Engineering: Provers (2 lectures)
9. Planning (1 lecture: overview)

1. Introduction

Content of this chapter (1):

Defining AI: There are several definitions of *AI*. They lead to several scientific areas ranging from *Cognitive Science* to *Rational Agents*.

History: We discuss some important philosophical ideas in the last 3 millennia and touch several events that play a role in later chapters (*syllogisms of Aristotle*, *Ockhams razor*, *Ars Magna*).

Content of this chapter (2):

AI since 1958: AI came into being in 1956-1958 with John McCarthy. We give a rough overview of its successes and failures.

Rational Agent: The modern approach to AI is based on the notion of a Rational Agent that is situated in an environment. We discuss the PEAS description and give some formal definitions of agency, introducing the notions of run, standard- vs. state- based agent

1.1 What Is AI?

"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense" (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

Table 1.1: Several Definitions of AI

1. Cognitive science

2. "Socrates is a man. All men are mortal. Therefore Socrates is mortal."
   (Famous *syllogisms* by Aristotle.)

   (1) **Informal description** $\rightsquigarrow$
   (2) **Formal description** $\rightsquigarrow$
   (3) Problem solution

   (2) is often problematic due to under-specification

   (3) is deduction (correct inferences): only enumerable, but **not decidable**

3. Turing Test:

`http://cogsci.ucsd.edu/~asaygin/tt/ttest.html`

`http://www.loebner.net/Prizef/loebner-prize.html`

- Standard Turing Test
- Total Turing Test

Turing believed in 1950:

In 2000 a computer with 10^9 memory-units could be programmed such that it can chat with a human for 5 minutes and pass the Turing Test with a probability of 30 %.

4. In item **2.** *correct* inferences were mentioned.

   Often not enough information is available in order to act in a way that makes sense (to act in a provably correct way).
   $\rightsquigarrow$ Non-monotonic logics.

   The world is in general under-specified. It is also impossible to act rationally without *correct* inferences: **reflexes**.

To pass the total Turing Test, the computer will need:

- computer vision to perceive objects,
- robotics to move them about.

The question *Are machines able to think?* leads to 2 theses:

- **Weak AI thesis:** Machines can be built, that **act as if they were intelligent**.
- **Strong AI thesis:** Machines that act in an intelligent way do possess cognitive states, i.e. mind.

The Chinese Chamber

was used in 1980 by Searle to demonstrate that a system can pass the Turing Test but disproves the Strong AI Thesis. The chamber consists of:

- **CPU:** an English-speaking man without experiences with the Chinese language,
- **Program:** a book containing rules formulated in English which describe how to translate Chinese texts,
- **Memory:** sufficient pens and paper

Papers with Chinese texts are passed to the man, which he translates using the book.

Question

Does the chamber understand Chinese?

1.2 From Plato To Zuse

450 BC: **Plato, Socrates, Aristotle**

Sokr.: *"What is characteristic of piety which makes all actions pious?"*
Aris.: *"Which laws govern the rational part of the mind?"*

800 : **Al Chwarizmi** (Arabia): Algorithm

1300 : **Raymundus Lullus**: Ars Magna

1350 : **William van Ockham**: Ockham's Razor
"Entia non sunt multiplicanda
praeter necessitatem."

1596–1650: **R. Descartes:**
Mind = physical system
Free will, dualism

1623–1662: **B. Pascal, W. Schickard:**
Addition-machines

1646–1716: **G. W. Leibniz:**
Materialism, uses ideas of Ars Magna to build a machine for simulating the human mind

1561–1626: **F. Bacon:** Empirism

1632–1704: **J. Locke:** Empirism
”Nihil est in intellectu quod non antefuerat in sensu.”

1711–1776 : **D. Hume:** Induction

1724–1804: **I. Kant:**
”Der Verstand schöpft seine Gesetze nicht aus der Natur, sondern schreibt sie dieser vor.”

1805 : **Jacquard:** Loom

1815–1864: **G. Boole:**
Formal language,
Logic as a **mathematical** discipline

1792–1871: **Ch. Babbage:**
Difference Engine: Logarithm-tables
Analytical Engine: with addressable memory,
stored programs and conditional jumps

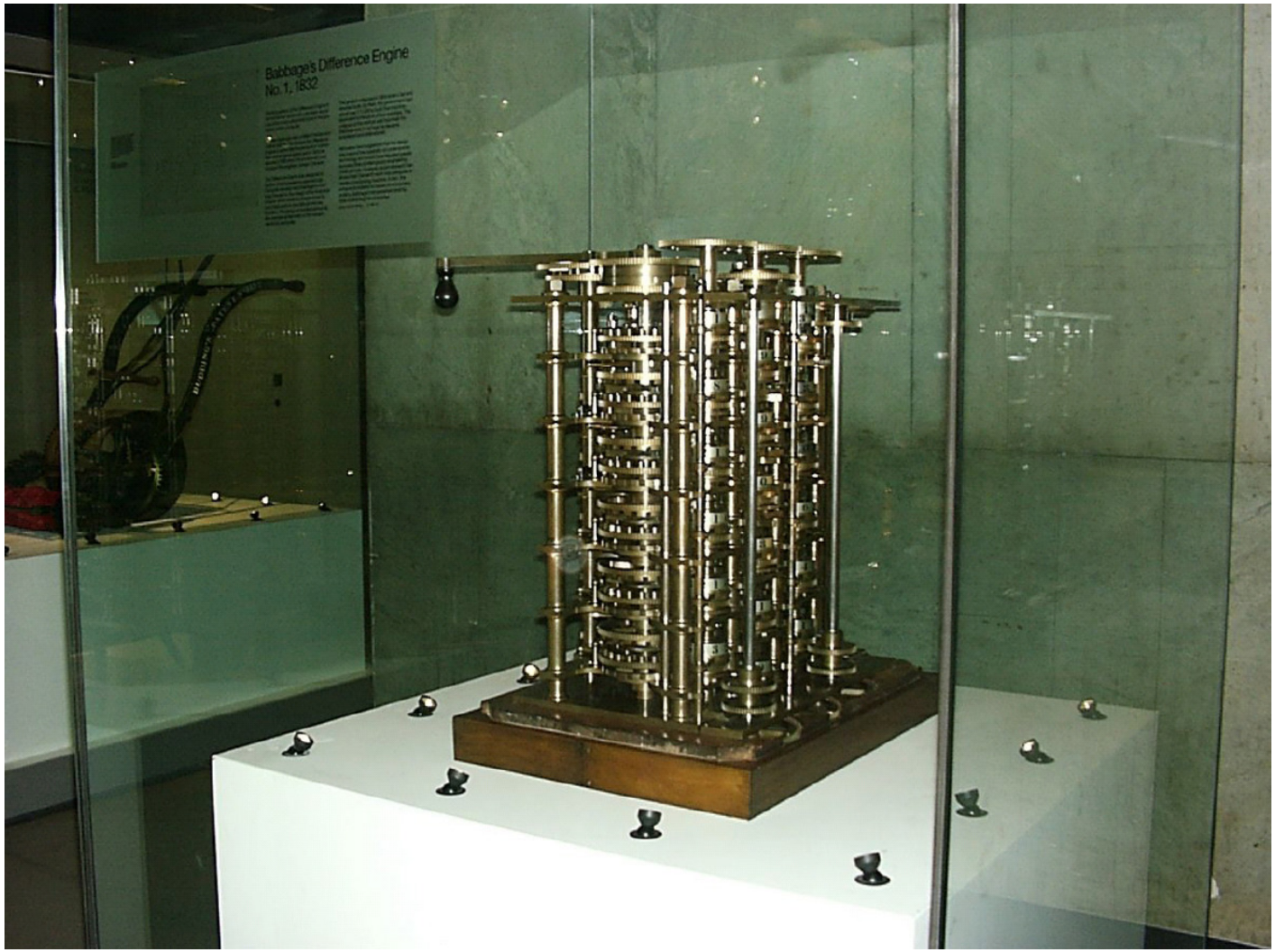


Figure 1.1: Reconstruction of Babbage's difference engine.

1848–1925 : G. Frege: Begriffsschrift

2-dimensional notation for PL1

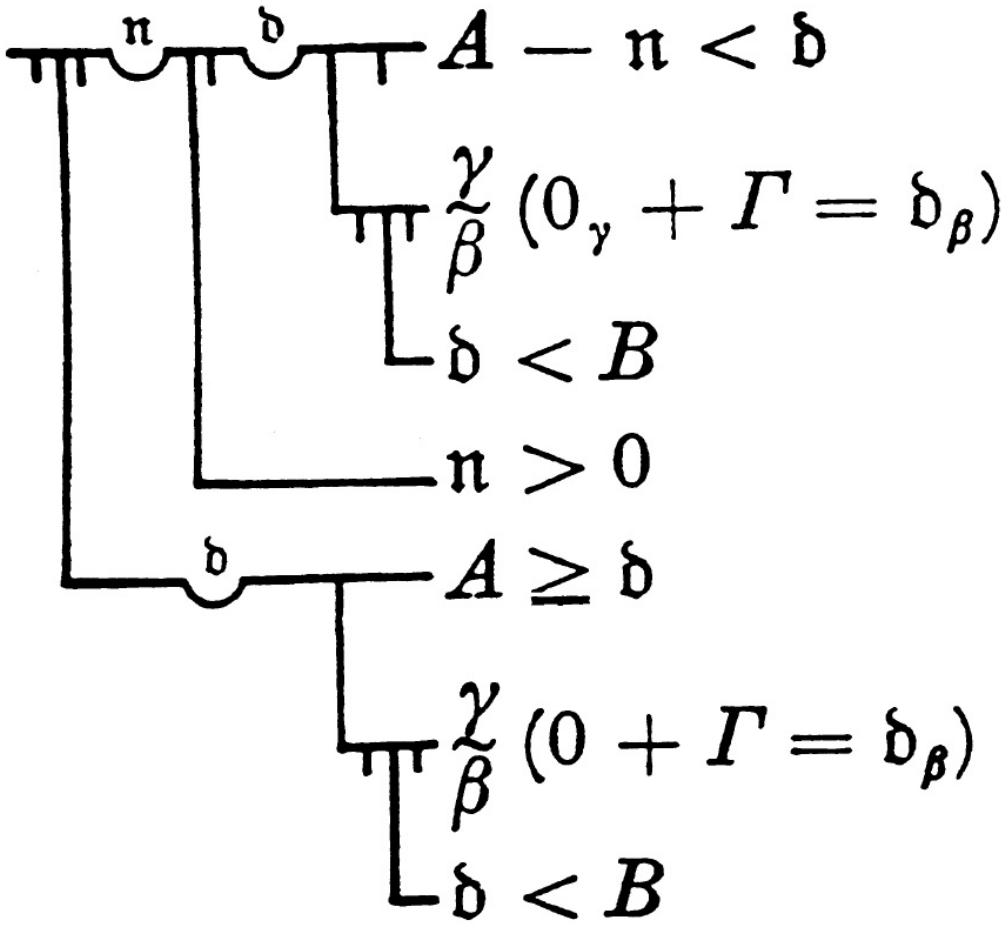


Figure 1.2: A formula from Frege's Begriffsschrift.

1862–1943: **D. Hilbert:**
Famous talk 1900 in Paris: 23 problems
23rd problem: The Entscheidungsproblem

1872–1970: **B. Russell:**
1910: Principia Mathematica
Logical positivism, *Vienna Circle* (1920–40)

1902–1983 : **A. Tarski:** (1936)
Idea of truth in formal languages

1906–1978 : **K. Gödel:**
Completeness theorem (1930)
Incompleteness theorem (1930/31)
Unprovability of theorems (1936)

1912–1954 : **A. Turing:**
Turing-machine (1936)
Computability

1903–1995 : **A. Church:**
λ-*Calculus*, Church-Turing-thesis

1938/42: First operational programmable computer: Z 1
Z 3 by **K. Zuse** (Deutsches Museum)
with floating-point-arithmetic.
Plankalkül: First high-level programming language

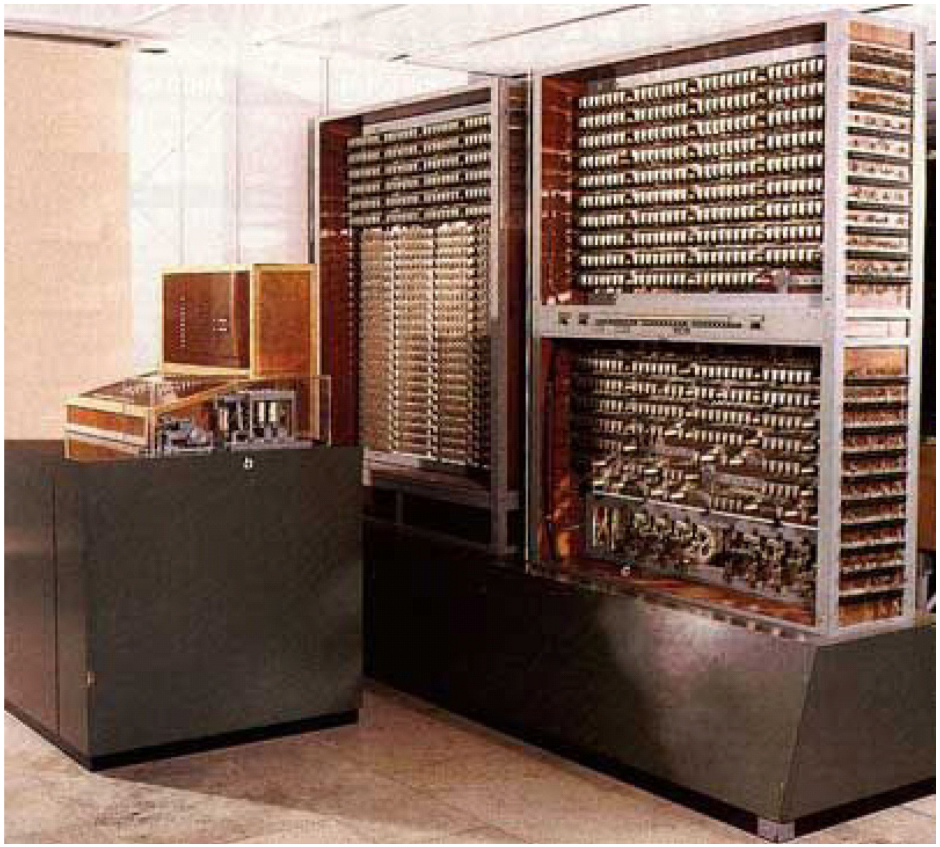


Figure 1.3: Reconstruction of Zuse's Z3.

1940 : First computer *"Heath Robinson"*
built to decipher German messages (Turing)
1943 *"Collossus"* built from vacuum tubes

1940–45: **H. Aiken:** develops *MARK I, II, III*.
ENIAC: First general purpose electronic comp.

1952 : IBM: *IBM 701*, first computer to yield a profit
(Rochester et al.)

1948: First stored program computer (*The Baby*)

Tom Kilburn (Manchester)

Manchester beats Cambridge by 3 months

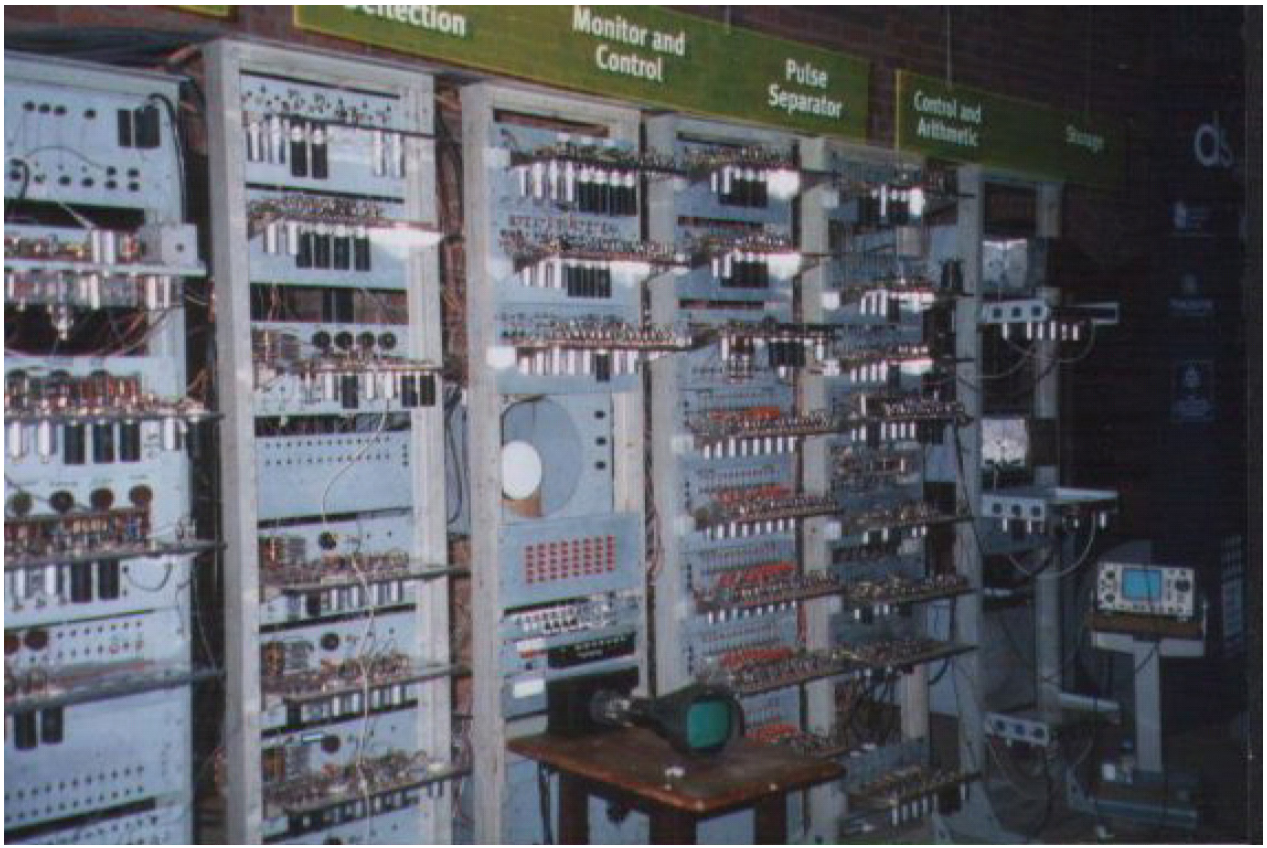


Figure 1.4: Reconstruction of Kilburn's baby.

First program run on The Baby in 1948:

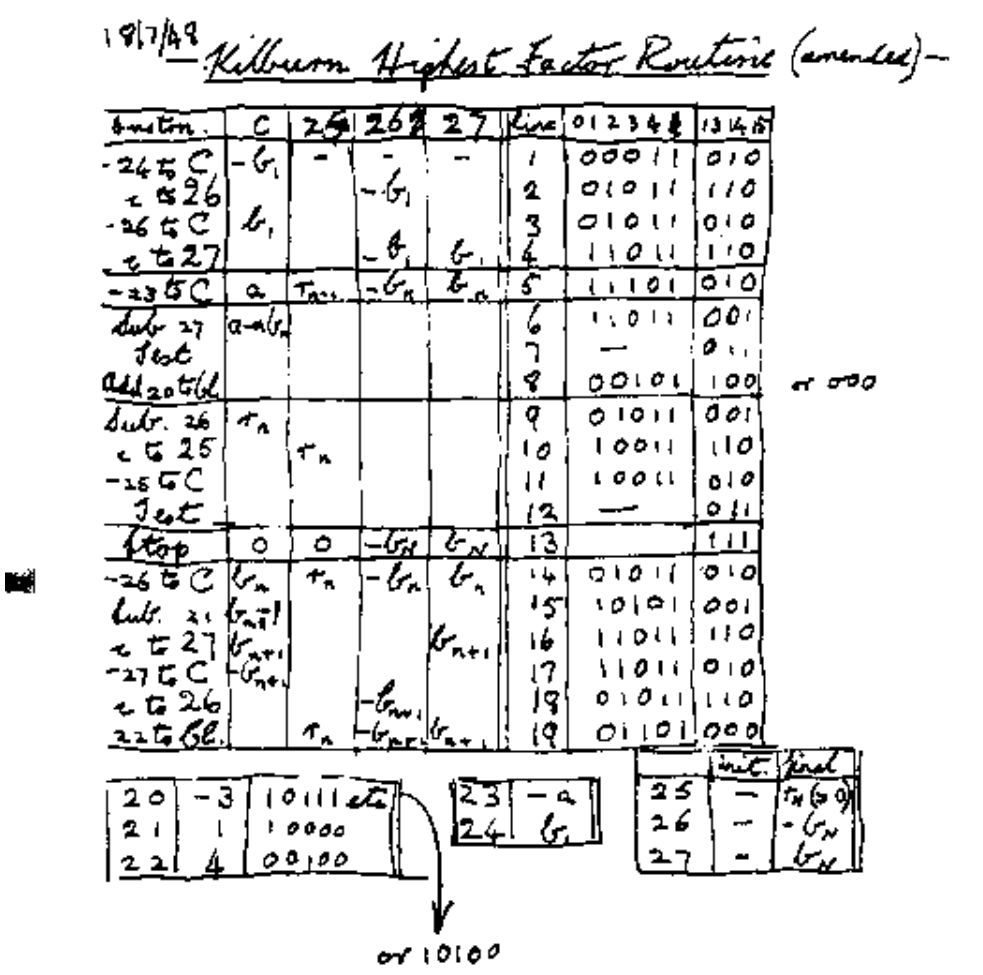


Figure 1.5: Reconstruction of first executed program on The Baby.

1.3 History of AI

The year 1943:

McCulloch and **W. Pitts** drew on three sources:

1. physiology and function of neurons in the brain,
2. propositional logic due to **Russell/Whitehead**,
3. Turing's theory of computation.

Model of artificial, connected neurons:

- Any computable function can be computed by some network of neurons.
- All the logical connectives can be implemented by simple net-structures.

The year 1951:

Minsky and **Edwards** build the first computer based on neuronal networks (Princeton)

The year 1952:

A. Samuel develops programs for checkers that learn to play tournament-level checkers.

The year 1956:

Two-month workshop at Dartmouth organized by **McCarthy**, **Minsky**, **Shannon** and **Rochester**.

Idea:

Combine knowledge about automata theory, neural nets and the studies of intelligence (10 participants)

Newell und **Simon** show a reasoning program, the Logic Theorist, able to prove most of the theorems in Chapter 2 of the *Principia Mathematica* (even one with a shorter proof).

But the *Journal of Symbolic Logic* rejected a paper **authored by *Newell, Simon and Logical Theorist***.

Newell and Simon claim to have solved the venerable mind-body problem.

The term Artificial Intelligence is proposed as the name of the new discipline.

Logic Theorist is followed by the General Problem Solver, which was designed from the start to imitate human problem-solving protocols.

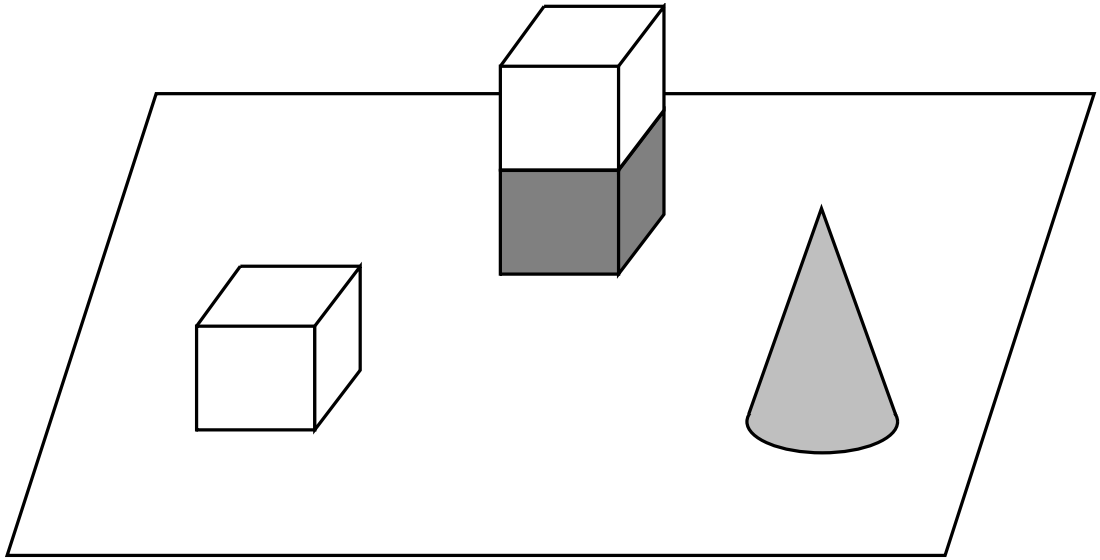
The year 1958: Birthyear of AI

McCarthy joins **MIT** and develops:

1. Lisp, the dominant AI programming language
2. Time-Sharing to optimize the use of computer-time
3. Programs with Common-Sense.
   Advice-Taker: A hypothetical program that can be seen as the first complete AI system. Unlike others it embodies general knowledge of the world.

The year 1959:

H. Gelernter: Geometry Theorem Prover

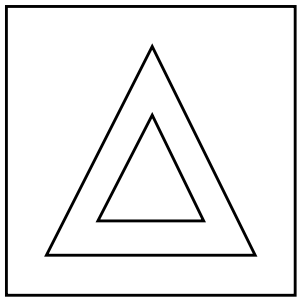
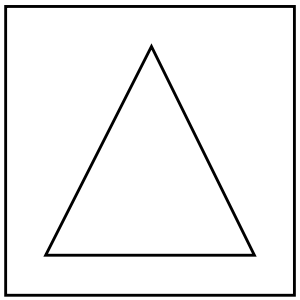
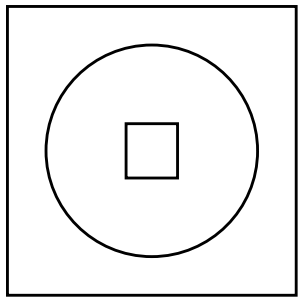


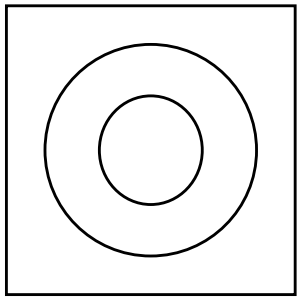
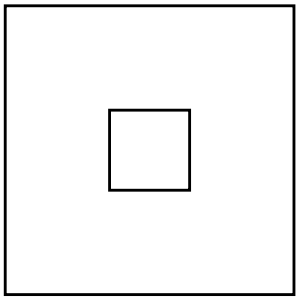
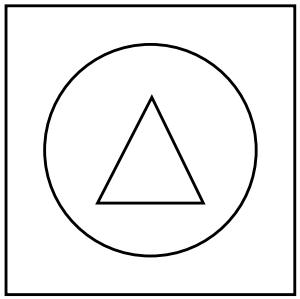
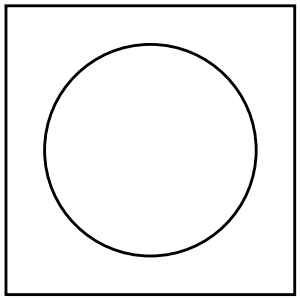
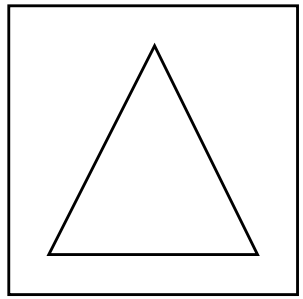
The years 1960-1966:

McCarthy concentrates on knowledge-representation and reasoning in formal logic ($\rightsquigarrow$ Robinson's Resolution, $\rightsquigarrow$ Green's Planner, $\rightsquigarrow$ Shakey).

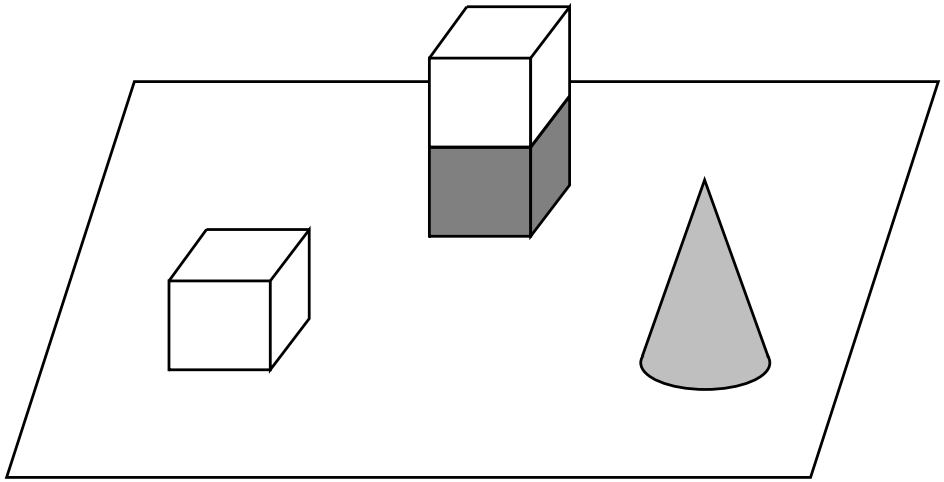
Minsky is more interested in getting programs to work and focusses on special worlds, the Microworlds.

- **SAINT** is able to solve integration problems typical of first-year college calculus courses
- **ANALOGY** is able to solve geometric analogy problems that appear in IQ-tests


is to

as

is to:


1

2

3

4

5

Blocksworld is the most famous microworld.



Work building on the neural networks of **McCulloch** and **Pitts** continued. *Perceptrons* by **Rosenblatt** and *convergence theorem*:

Convergence theorem

An algorithm exists that can adjust the connection strengths of a perceptron to match any input data.

Summary:

Great promises for the future, initial successes but miserable further results.

The year 1966: All US funds for AI research are cancelled.

Inacceptable mistake

The spirit is willing but the flesh is weak.

was translated into

The vodka is good but the meat is rotten.

The years 1966–1974: A dose of reality

Simon 1958: *"In 10 years a computer will be grandmaster of chess."*

Simple problems are solvable due to small search-space. Serious problems remain unsolvable.

Hope:

Faster hardware and more memory will solve everything!

$\rightsquigarrow$ NP-Completeness, **S. Cook/R. Karp** (1971), $P \neq NP$?

The year 1973:

The Lighthill report forms the basis for the decision by the British government to end support for AI research.

Minsky's book Perceptrons proved limitations of some approaches with fatal consequences:

Research funding for neural net research decreased to almost nothing.

The years 1969–79: Knowledge-based systems

General purpose mechanisms are called *weak methods*, because they use weak information about the domain. For many complex problems it turns out that their performance is also weak.

Idea:

Use knowledge suited to making larger reasoning steps and to solving typically occurring cases on narrow areas of expertise.

Example DENDRAL (1969)

Leads to expert systems like **MYCIN** (diagnosis of blood infections).

- ’73: PROLOG
- ’74: Relational databases (**Codd**)
- 81-91: Fifth generation project
- ’91: Dynamic Analysis and Replanning Tool (**DART**) paid back DARPA’s investment in AI during the last 30 years
- ’97: IBM’s Deep Blue
- ’98: NASA’s remote agent program
- ’11: IBM’s Watson winning *Jeopardy!* `http://www-05.ibm.com/de/pov/watson/`

Something to laugh about: In 1902 a German poem was translated into Japanese. The Japanese version war translated into French. At last this version was translated back into German, assuming that it was a Japanese poem.

The result:

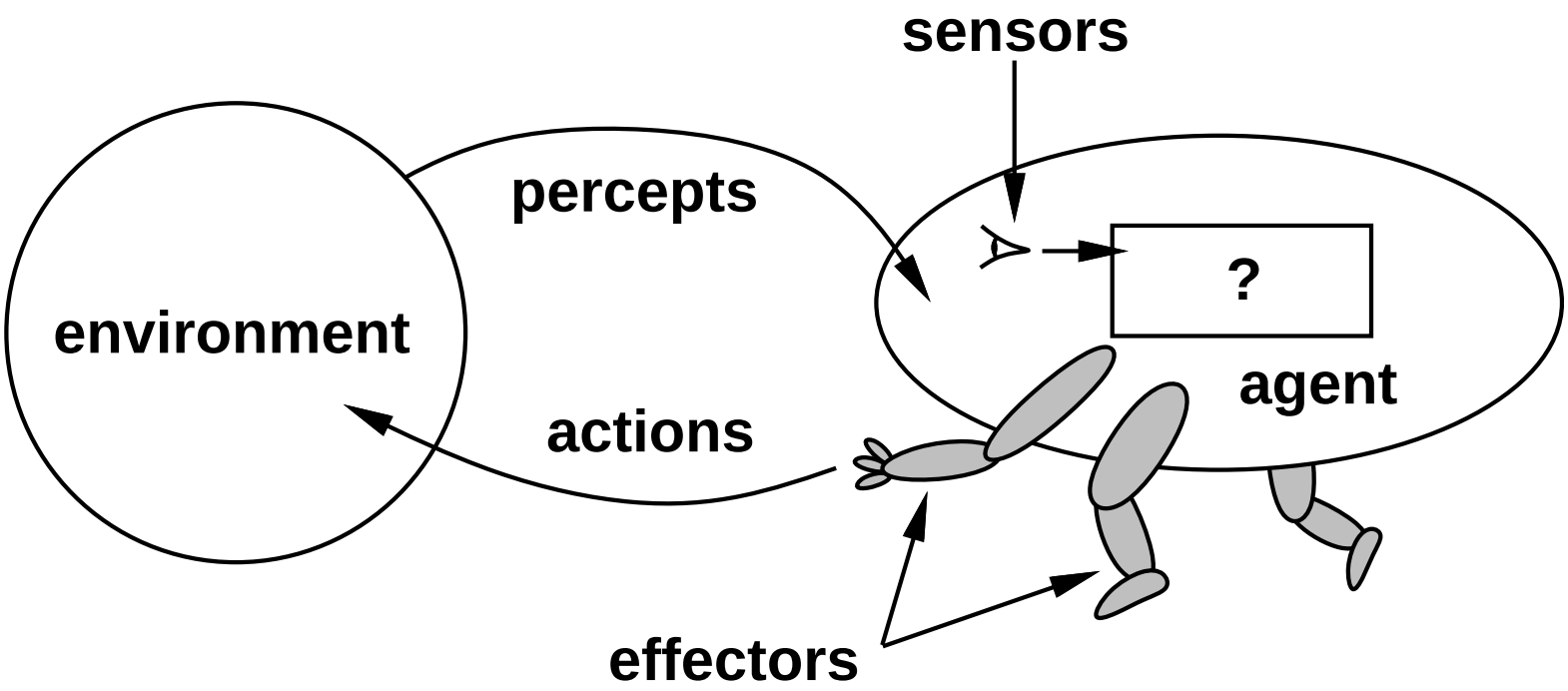
Stille ist im Pavillon aus Jade,
Krähen fliegen stumm
Zu beschneiten Kirschbäumen im Mondlicht.
Ich sitze
und weine.

What is the original poem?

1.4 Intelligent Agents

Definition 1.1 (Agent α)

An agent α is anything that can be viewed as **perceiving** its environment through **sensor** and **acting** upon that environment through **effectors**.



Definition 1.2 (Rational Agent, Omniscient Agent)

A rational agent is one that does the right thing (**Performance measure** determines how successful an agent is).

A omniscient agent knows the actual outcome of his actions and can act accordingly.

Attention:

A rational agent is in general not omniscient!

Question

What is the right thing and what does it depend on?

1. Performance measure (as objective as possible).
2. Percept sequence (everything the agent has received so far).
3. The agent's knowledge about the environment.
4. How the agent can act.

Definition 1.3 (Ideal Rational Agent)

For each possible percept-sequence an ideal rational agent should do whatever action is expected to maximize its performance measure (based on the evidence provided by the percepts and built-in knowledge).

Note the performance measure is something **outside** the agent. It allows to compare agents performances.

Mappings:

$$\text{set of percept sequences} \mapsto \text{set of actions}$$

can be used to describe agents in a mathematical way.

Remark:

Internally an agent is

agent = architecture + program

AI is engaged in designing agent programs

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Table 1.2: Examples of agents types and their PEAS descriptions.

A simple agent program:

function SKELETON-AGENT(*percept*) **returns** action
static: *memory*, the agent's memory of the world

memory ← UPDATE-MEMORY(*memory*, *percept*)
action ← CHOOSE-BEST-ACTION(*memory*)
memory ← UPDATE-MEMORY(*memory*, *action*)
return *action*

In theory everything is trivial:

function TABLE-DRIVEN-AGENT(*percept*) **returns** *action*
  static: *percepts*, a sequence, initially empty
    table, a table, indexed by percept sequences, initially fully specified

  append *percept* to the end of *percepts*
  action ← LOOKUP(*percepts*, *table*)
  return *action*

An agent example – taxi driver:

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Table 1.3: PEAS description of the environment for an automated taxi

Some examples:

1. **Production rules:** If the driver in front hits the breaks, then hit the breaks too.

function SIMPLE-REFLEX-AGENT(*percept*) **returns** *action*
static: *rules*, a set of condition-action rules

state ← INTERPRET-INPUT(*percept*)
rule ← RULE-MATCH(*state*, *rules*)
action ← RULE-ACTION[*rule*]
return *action*

A first mathematical description

At first, we want to keep everything as simple as possible.

Agents and environments

An agent is situated in an environment and can perform actions

$$A := \{a_1, \ldots, a_n\} \qquad \text{(set of actions)}$$

and change the state of the environment

$$S := \{s_1, s_2, \ldots, s_n\} \qquad \text{(set of states).}$$

How does the environment (the state $\mathbf{s}$) develop when an action $\mathbf{a}$ is executed?

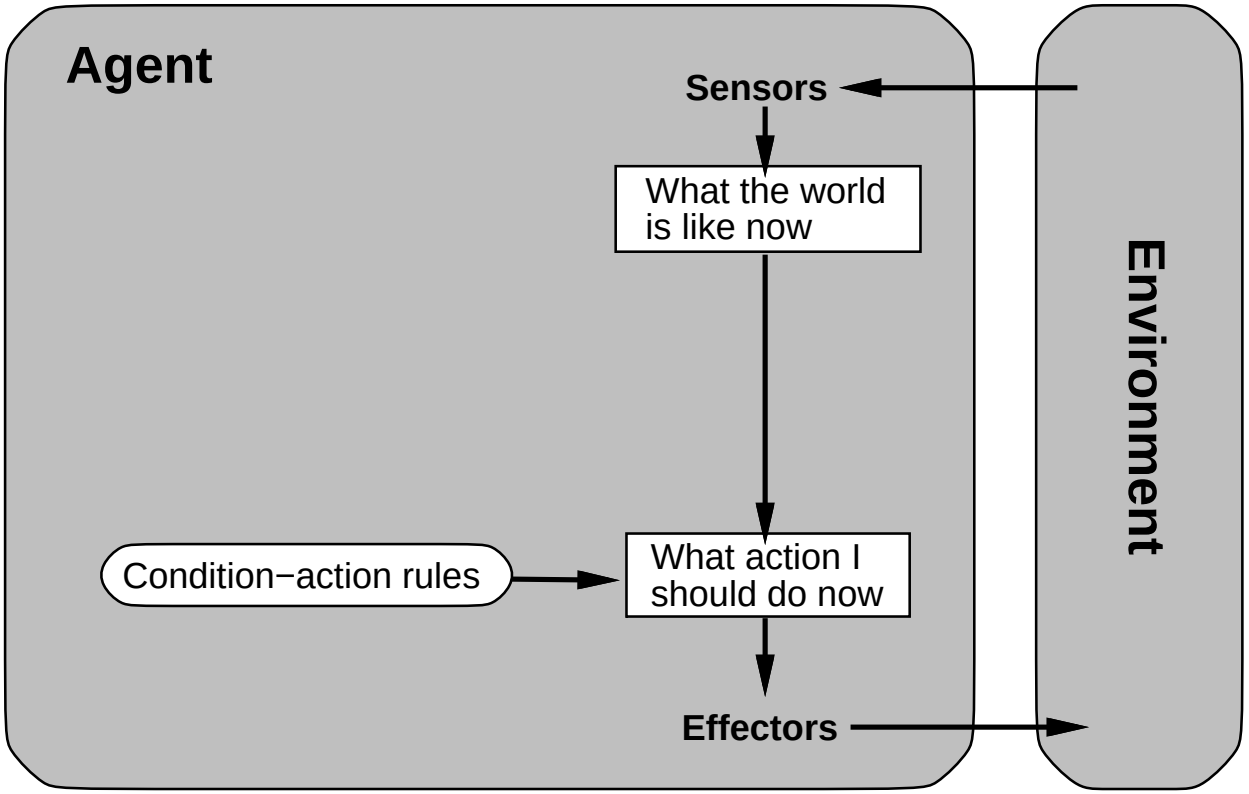
We describe this with a function

$$\mathbf{env} : \mathbf{S} \times \mathbf{A} \longrightarrow \mathbf{2^S}.$$

This includes non-deterministic environments.

How do we describe agents?

We could take a function $\mathbf{action} : \mathbf{S} \longrightarrow \mathbf{A}$.



Question:

How can we describe an agent, now?

Definition 1.4 (Purely Reactive Agent)

An agent is called purely reactive, if its function is given by

$$\mathbf{action} : \mathbf{S} \longrightarrow \mathbf{A}.$$

This is too weak!

Take the whole history (of the environment) into account:

$$\mathbf{s_0} \rightarrow_{\mathbf{a_0}} \mathbf{s_1} \rightarrow_{\mathbf{a_1}} \ldots \mathbf{s_n} \rightarrow_{\mathbf{a_n}} \ldots.$$

The same should be done for $\mathbf{env}$!

This leads to agents that take the **whole sequence of states** into account, i.e.

$$\mathbf{action} : \mathbf{S}^* \longrightarrow \mathbf{A}.$$

We also want to consider the actions **performed by an agent**. This requires the notion of a run (next slide).

We define the run of an agent in an environment as a sequence of interleaved states and actions:

Definition 1.5 (Run r, $\mathrm{R} = \mathrm{R}^{act} \cup \mathrm{R}^{state}$)

A run r over $\mathbf{A}$ and $\mathbf{S}$ is a finite sequence

$$\mathbf{r} : \mathbf{s_0} \rightarrow_{\mathbf{a_0}} \mathbf{s_1} \rightarrow_{\mathbf{a_1}} \ldots \mathbf{s_n} \rightarrow_{\mathbf{a_n}} \ldots$$

Such a sequence may end with a state $\mathbf{s_n}$ or with an action $\mathbf{a_n}$: we denote by R^{act} the set of runs ending with an action and by R^{state} the set of runs ending with a state.

Definition 1.6 (Environment, 2. version)

An environment Env is a triple $\langle \mathrm{S}, \mathrm{s_0}, \tau \rangle$ consisting of

1. the set S of states,
2. the initial state $\mathrm{s_0} \in \mathrm{S}$,
3. a function $\tau : \mathrm{R}^{act} \longrightarrow 2^{\mathrm{S}}$, which describes how the environment changes when an action is performed (given the whole history).

Definition 1.7 (Agent system $\mathfrak{a}$)

An agent $\mathfrak{a}$ is determined by a function

$$\mathbf{action} : \mathrm{R}^{state} \longrightarrow \mathbf{A},$$

describing which action the agent performs, given its current history.
An agent system is then a pair $\mathfrak{a} = \langle \mathbf{action}, Env \rangle$ consisting of an agent and an environment.
We denote by $\mathrm{R}(\mathfrak{a}, Env)$ the set of runs of agent $\mathfrak{a}$ in environment Env.

Definition 1.8 (Characteristic Behaviour)

The characteristic behaviour of an agent $\mathbf{a}$ in an environment Env is the set R of all possible runs $\mathbf{r} : \mathbf{s_0} \rightarrow_{a_0} \mathbf{s_1} \rightarrow_{a_1} \ldots \mathbf{s_n} \rightarrow_{a_n} \ldots$ with:

1. for all n: $\mathbf{a_n} = \mathbf{action}(\langle \mathbf{s_0}, \mathbf{a_0} \ldots, \mathbf{a_{n-1}}, \mathbf{s_n} \rangle)$,
2. for all $n > 0$: $\mathbf{s_n} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{n-1}}, a_{n-1})$.

For deterministic τ, the relation "$\in$" can be replaced by "=".

Important:

The formalization of the characteristic behaviour is dependent on the concrete **agent type**. Later we will introduce further behaviours (and corresponding agent designs).

Definition 1.9 (Equivalence)

Two agents $\mathbf{a}$, $\mathbf{b}$ are called **behaviourally equivalent wrt. environment** Env, if $\mathrm{R}(\mathbf{a}, Env) = \mathrm{R}(\mathbf{b}, Env)$.

Two agents $\mathbf{a}$, $\mathbf{b}$ are called behaviourally equivalent, if they are behaviourally equivalent wrt. all possible environments Env.

So far so good, but...

What is the problem with all these agents and this framework in general?

Problem

All agents have perfect information about the environment!

(Of course, it can also be seen as feature!)

We need more realistic agents!

Note

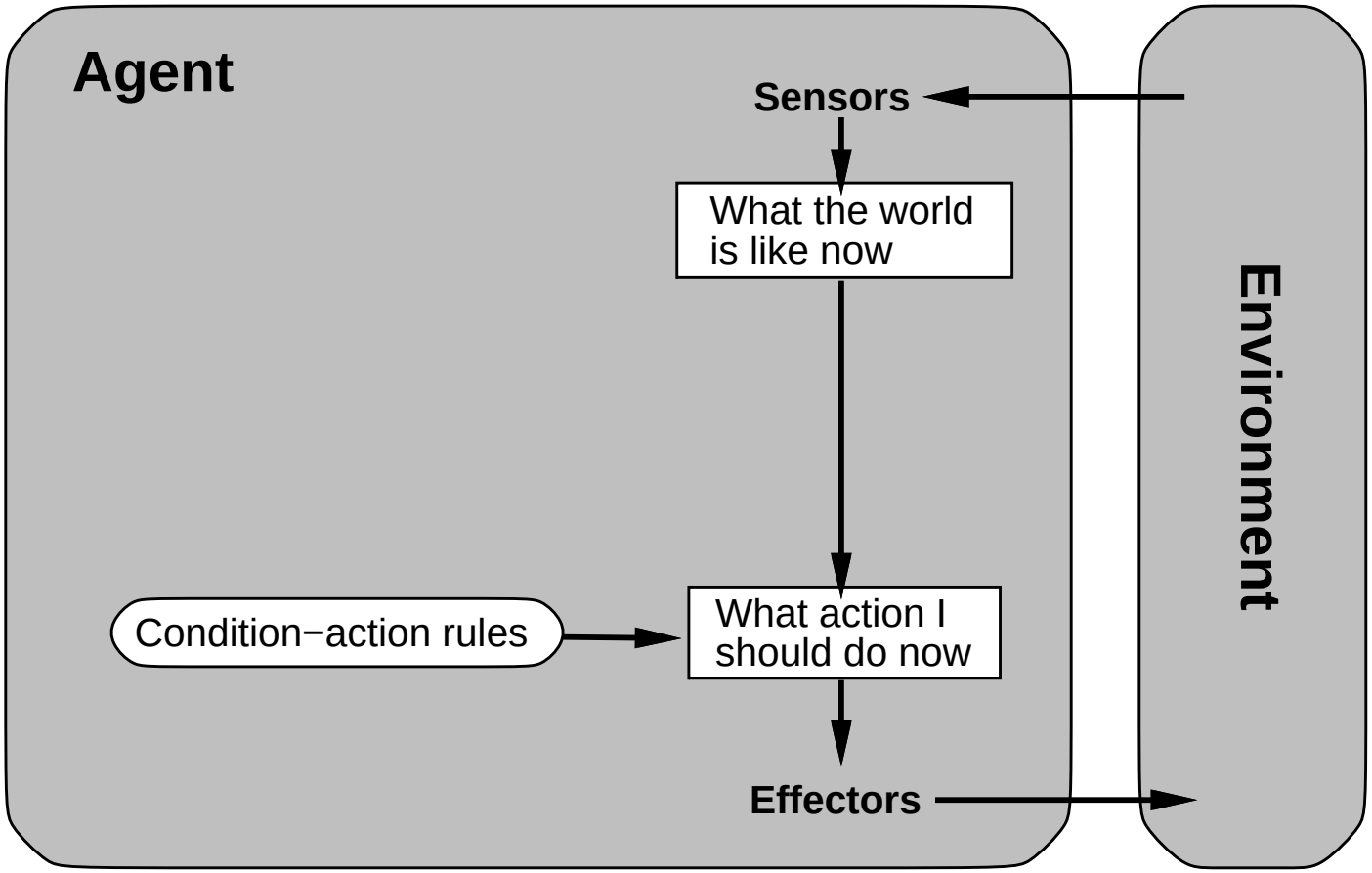
In general, agents only have incomplete/uncertain information about the environment!

We extend our framework by perceptions:

Definition 1.10 (Actions A, Percepts P, States S)

$\mathbf{A} := \{\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_n}\}$ is the set of *actions*.
$\mathbf{P} := \{\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_m}\}$ is the set of percepts.
$\mathbf{S} := \{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_l}\}$ is the set of *states*

Sensors don't need to provide perfect information!



Question:

How can agent programs be designed?

There are four types of agent programs:

- Simple **reflex agents**
- **Agents that keep track of the world**
- Goal-based agents
- Utility-based agents

First try

We consider a purely reactive agent and just replace states by perceptions.

Definition 1.11 (Simple Reflex Agent)

An agent is called simple reflex agent, if its function is given by

$$\mathbf{action} : \mathbf{P} \longrightarrow \mathbf{A}.$$

A very simple reflex agent

function SIMPLE-REFLEX-AGENT(*percept*) **returns** *action*
static: *rules*, a set of condition-action rules

state ← INTERPRET-INPUT(*percept*)
rule ← RULE-MATCH(*state*, *rules*)
action ← RULE-ACTION[*rule*]
return *action*

A simple reflex agent with memory

function REFLEX-AGENT-WITH-STATE(*percept*) **returns** *action*
  static: *state*, a description of the current world state
    rules, a set of condition-action rules

  state ← UPDATE-STATE(*state, percept*)
  rule ← RULE-MATCH(*state, rules*)
  action ← RULE-ACTION[*rule*]
  state ← UPDATE-STATE(*state, action*)
  return *action*

As before, let us now consider sequences of percepts:

Definition 1.12 (Standard Agent $\mathfrak{a}$)

A standard agent $\mathfrak{a}$ is given by a function

$$\mathbf{action} : \mathbf{P}^* \longrightarrow A$$

together with

$$\mathbf{see} : \mathbf{S} \longrightarrow \mathbf{P}.$$

An agent is thus a pair $\langle \mathbf{see}, \mathbf{action} \rangle$.

Definition 1.13 (Indistinguishable)

Two different states s, s' are indistinguishable for an agent $\mathfrak{a}$, if $\mathbf{see}(s) = \mathbf{see}(s')$.

The relation "indistinguishable" on $\mathbf{S} \times \mathbf{S}$ is an **equivalence** relation.

What does $|\sim| = |\mathbf{S}|$ mean?

And what $|\sim| = 1$?

The characteristic behaviour has to match with the agent design!

Definition 1.14 (Characteristic Behaviour)

The characteristic behaviour of a standard agent $\langle \mathbf{see}, \mathbf{action} \rangle$ in an environment Env is the set of all finite sequences $\mathbf{p_0} \rightarrow_{a_0} \mathbf{p_1} \rightarrow_{a_1} \ldots \mathbf{p_n} \rightarrow_{a_n} \ldots$ where

$$\mathbf{p_0} = \mathbf{see}(\mathbf{s_0}),$$
$$\mathbf{a_i} = \mathbf{action}(\langle \mathbf{p_0}, \ldots, \mathbf{p_i} \rangle),$$
$$\mathbf{p_i} = \mathbf{see}(\mathbf{s_i}), \text{ where } \mathbf{s_i} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{i-1}}, a_{i-1}).$$

Such a sequence, even if deterministic from the agent's viewpoint, may cover different environmental behaviours (runs):

$$\mathbf{s_0} \rightarrow_{a_0} \mathbf{s_1} \rightarrow_{a_1} \ldots \mathbf{s_n} \rightarrow_{a_n} \ldots$$

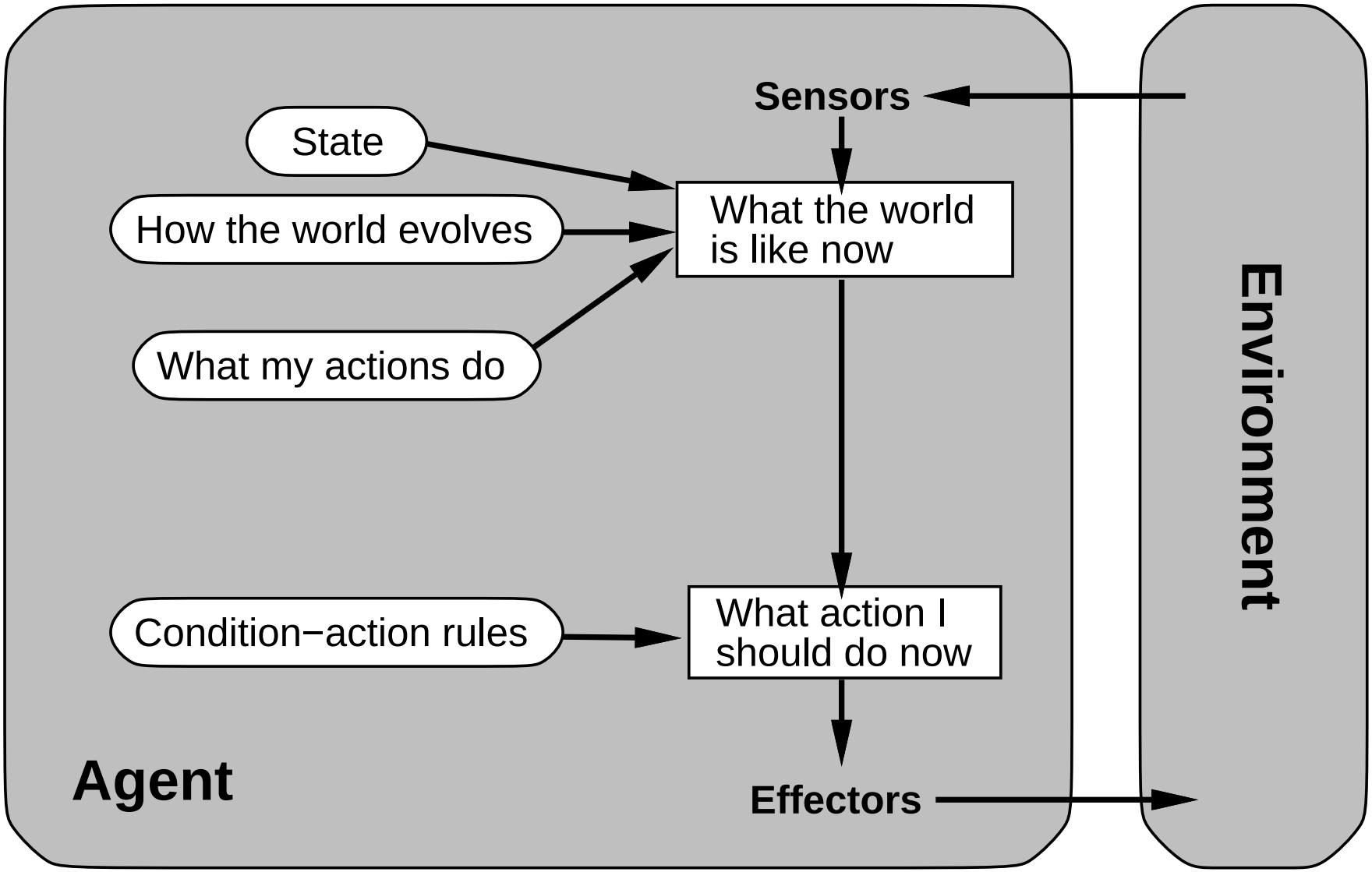
Instead of using the whole history, resp. $\mathbf{P}^*$, one can also use **internal states**
$\mathbf{I} := \{\mathbf{i_1}, \mathbf{i_2}, \ldots, \mathbf{i_n}, \mathbf{i_{n+1}}, \ldots\}$.

Definition 1.15 (State-based Agent $\mathbf{a}_{state}$)

A state-based agent $\mathbf{a}_{state}$ is given by a function $\mathbf{action} : I \longrightarrow A$ together with

$$\mathbf{see} : \mathbf{S} \longrightarrow \mathbf{P},$$
$$\text{and} \quad \mathbf{next} : \mathbf{I} \times \mathbf{P} \longrightarrow \mathbf{I}.$$

Here $\mathbf{next}(\mathbf{i}, \mathbf{p})$ is the successor state of $\mathbf{i}$ if $\mathbf{p}$ is observed.


Sensors
State
How the world evolves
What my actions do
What the world is like now
Condition−action rules
What action I should do now
Effectors
Agent
Environment

Definition 1.16 (Characteristic Behaviour)

The characteristic behaviour of a state-based agent $\mathbf{a}_{state}$ in an environment Env is the set of all finite sequences

$$(\mathbf{i}_0, \mathbf{p_0}) \rightarrow_{a_0} (\mathbf{i}_1, \mathbf{p_1}) \rightarrow_{a_1} \ldots \rightarrow_{a_{n-1}} (\mathbf{i}_n, \mathbf{p_n}), \ldots$$

with

$$\begin{aligned}
&\mathbf{p_0} = \mathbf{see}(\mathbf{s_0}),\\
&\mathbf{p_i} = \mathbf{see}(\mathbf{s_i}), \text{ where } \mathbf{s_i} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{i-1}}, a_{i-1}),\\
&\mathbf{a_n} = \mathbf{action}(\mathbf{i_{n+1}}),\\
&\mathbf{next}(\mathbf{i_n}, \mathbf{p_n}) = \mathbf{i_{n+1}}.
\end{aligned}$$

Sequence covers the runs $\mathbf{r} : \mathbf{s_0} \rightarrow_{\mathbf{a_0}} \mathbf{s_1} \rightarrow_{\mathbf{a_1}} \ldots$ where

$$\begin{aligned}
&\mathbf{a_j} = \mathbf{action}(\mathbf{i_{j+1}}),\\
&\mathbf{s_j} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{j-1}}, a_{j-1}),\\
&\mathbf{p_j} = \mathbf{see}(\mathbf{s_j})
\end{aligned}$$

Are state-based agents more expressive than standard agents? How to measure?

Definition 1.17 (Environmental Behaviour of a_{state})

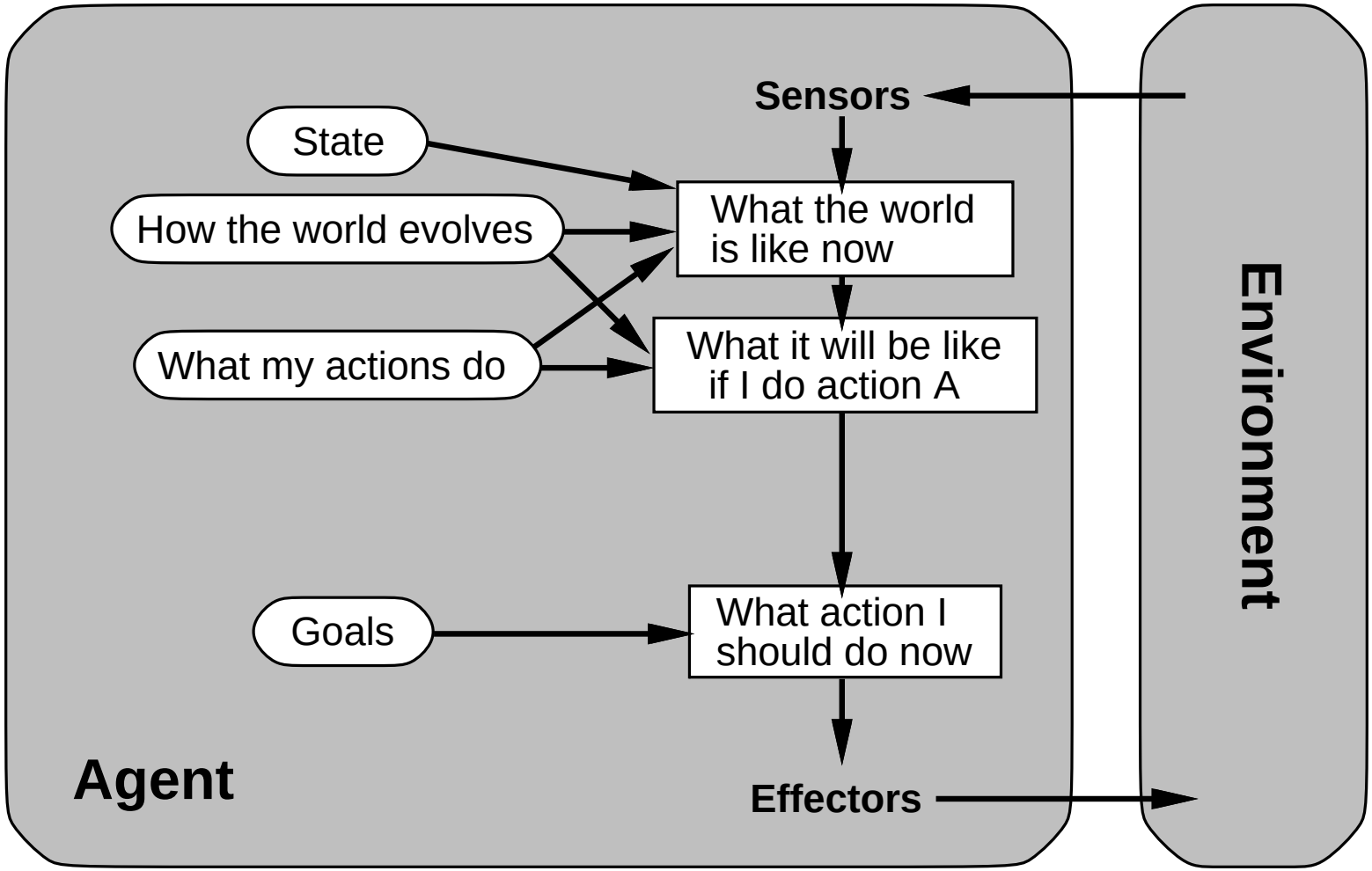
The environmental behaviour of an agent a_{state} is the set of possible runs covered by the characteristic behaviour of the agent.

Theorem 1.18 (Equivalence)

Standard agents and state-based agents are equivalent with respect to their environmental behaviour.

More precisely: For each state-based agent $\mathbf{a}_{state}$ *and* $\mathbf{next}$ *storage function there exists a standard agent* $\mathbf{a}$ *which has the same environmental behaviour, and vice versa.*

3. Goal based agents:



Will be discussed in **Planning** (Chapter 9).

Intention:

We want to tell our agent **what to do**, without telling it **how to do it**!

We need a utility measure.

How can we define a utility function?

Take for example

$$u : \mathbf{S} \mapsto \mathbb{R}$$

Question:

What is the problem with this definition?

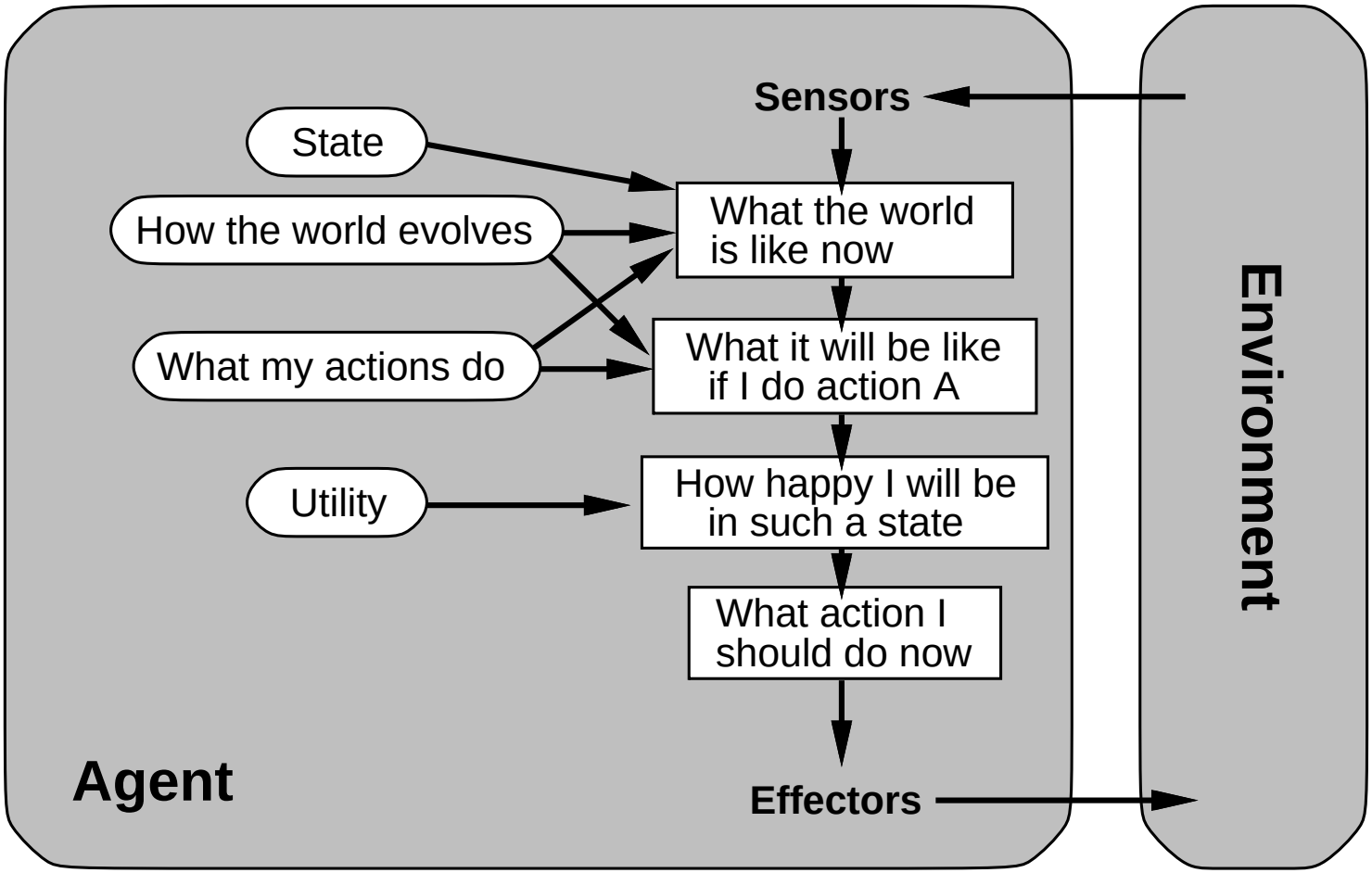
Better idea:

$$u : \mathrm{R} \mapsto \mathbb{R}$$

Take the set of runs R and not the set of states $\mathbf{S}$.

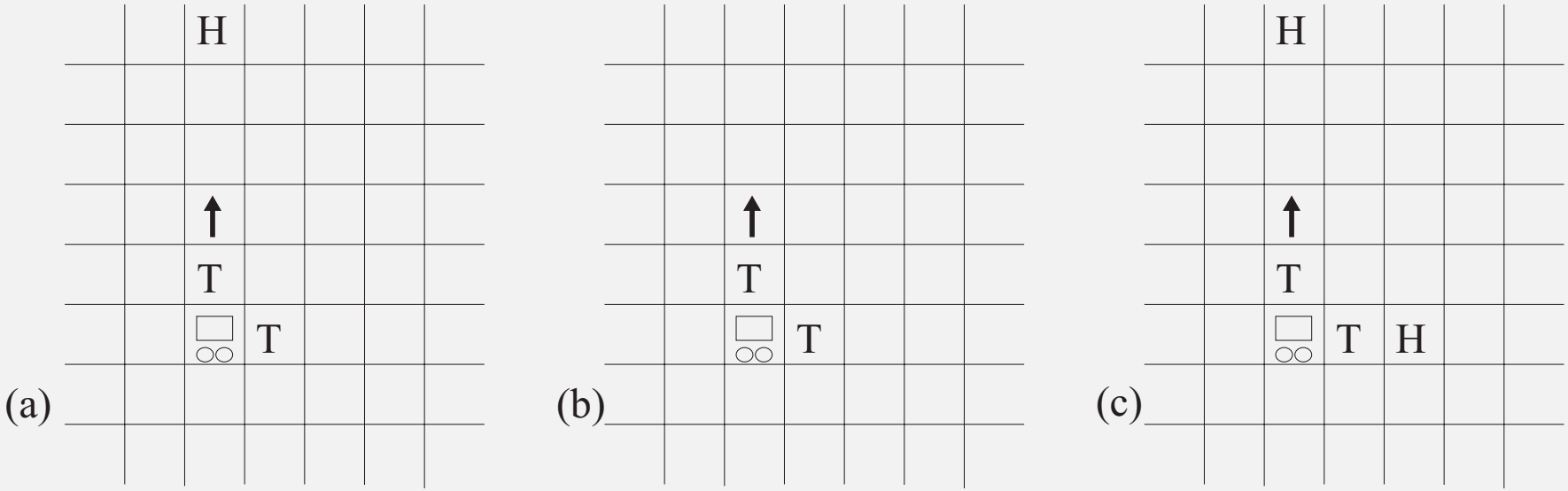
How to calculate the utility if the agent has incomplete information?

4. Agents with a utility measure:



The utility measure can often be simulated through a set of goals. **It would be helpful, if the utility is close to the performance measure.**

Example 1.19 (Tileworld)



Question:

How do **properties of the environment** influence the *design of an agent*?

Some **important properties**:

Fully/partially oberservable: If the environment is not completely observable the agent will need **internal states**.

Deterministic/stochastic: If the environment is only partially observable, then it may **appear stochastic** (while it is deterministic).

Episodic/nonepisodic: Percept-action-sequences are independent. The agent's experience is divided into episodes.

Static/dynamic: The environment can change while an agent is deliberating. An environment is semidynamic if it does not change with the passage of time but the performance measure does.

Discrete/continuous: If there is a limited number of percepts and actions the environment is discrete.

Single/multi agents: Is there just one agent or are there several interacting with each other.

Task Environment	Observ.	In/Determ.	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Determ.	Seq.	Stat.	Discr.	Single
Chess with a clock	Fully	Strategic	Seq.	Semi	Discr.	Multi
Poker	Part.	Strategic	Seq.	Stat.	Discr.	Multi
Backgammon	Fully	Stochastic	Seq.	Stat.	Discr.	Multi
Taxi driving	Part.	Stochastic	Seq.	Dyn.	Cont	Multi
Medical diagnosis	Part.	Stochastic	Seq.	Dyn.	Cont	Single
Image-analysis	Fully	Determ.	Epis.	Semi	Cont	Single
Part-picking robot	Part.	Stochastic	Epis.	Dyn.	Cont	Single
Refinery controller	Part.	Stochastic	Seq.	Dyn.	Cont	Single
Interactive tutor	Part.	Stochastic	Seq.	Dyn.	Discr.	Multi

Table 1.4: Examples of task environments and their characteristics

2. Searching

Content of this chapter (1):

Searching: Search Algorithms are perhaps the most basic notion of AI. Almost any problem can be formulated as a search problem.

(Un-) informed: We distinguish between uninformed and informed search algorithms. In the latter case, there is information available to guide the search. Often, this results in algorithms that are exponentially better than uninformed ones.

Content of this chapter (2):

A^*: The A^* algorithm is one of the fundamental informed search algorithms in AI. We discuss several variants based on Tree-Search or Graph-Search and discuss their correctness and completeness. We also consider Heuristics.

Memory: We discuss several variants which use only limited memory: IDA*, RBFS, and SMA*.

Extensions: Finally, we conclude with a few words about iterative improvements (genetic algorithms or simulated annealing) and online algorithms (where actions and search are interleaved). We present the LRTA* algorithm.

Wanted: Problem-solving agents, which form a subclass of the goal-oriented agents.

Structure: **Formulate**, **Search**, **Execute**.

Formulate:

- **Goal:** a set of states,
- **Problem:** States, actions mapping from states into states, transitions

Search: Which sequence of actions is helpful?

Execute: The resulting sequence of actions is applied to the initial state.

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept
  static: *seq*, an action sequence, initially empty
    state, some description of the current world state
    goal, a goal, initially null
    problem, a problem formulation

state ← UPDATE-STATE(*state*, *percept*)
if *seq* is empty **then do**
  goal ← FORMULATE-GOAL(*state*)
  problem ← FORMULATE-PROBLEM(*state*, *goal*)
  seq ← SEARCH(*problem*)
action ← FIRST(*seq*)
seq ← REST(*seq*)
return *action*

Table 2.5: A simple problem solving agent.

When executing, percepts are ignored.

2.1 Problem Formulation

We distinguish four types:

1. **1-state-problems:** Actions are completely described. Complete information through sensors to determine the actual state.
2. **Multiple-state-problems:** Actions are completely described, but the initial state is not certain.
3. **Contingency-problems:** Sometimes the result is not a fixed sequence, so the complete tree must be considered.
   ($\rightsquigarrow$ Excercise: Murphy's law, $\rightsquigarrow$ Chapter 9. Planning)
4. **Exploration-problems:** Not even the effect of each action is known. You have to search in the world instead of **searching in the abstract model**.

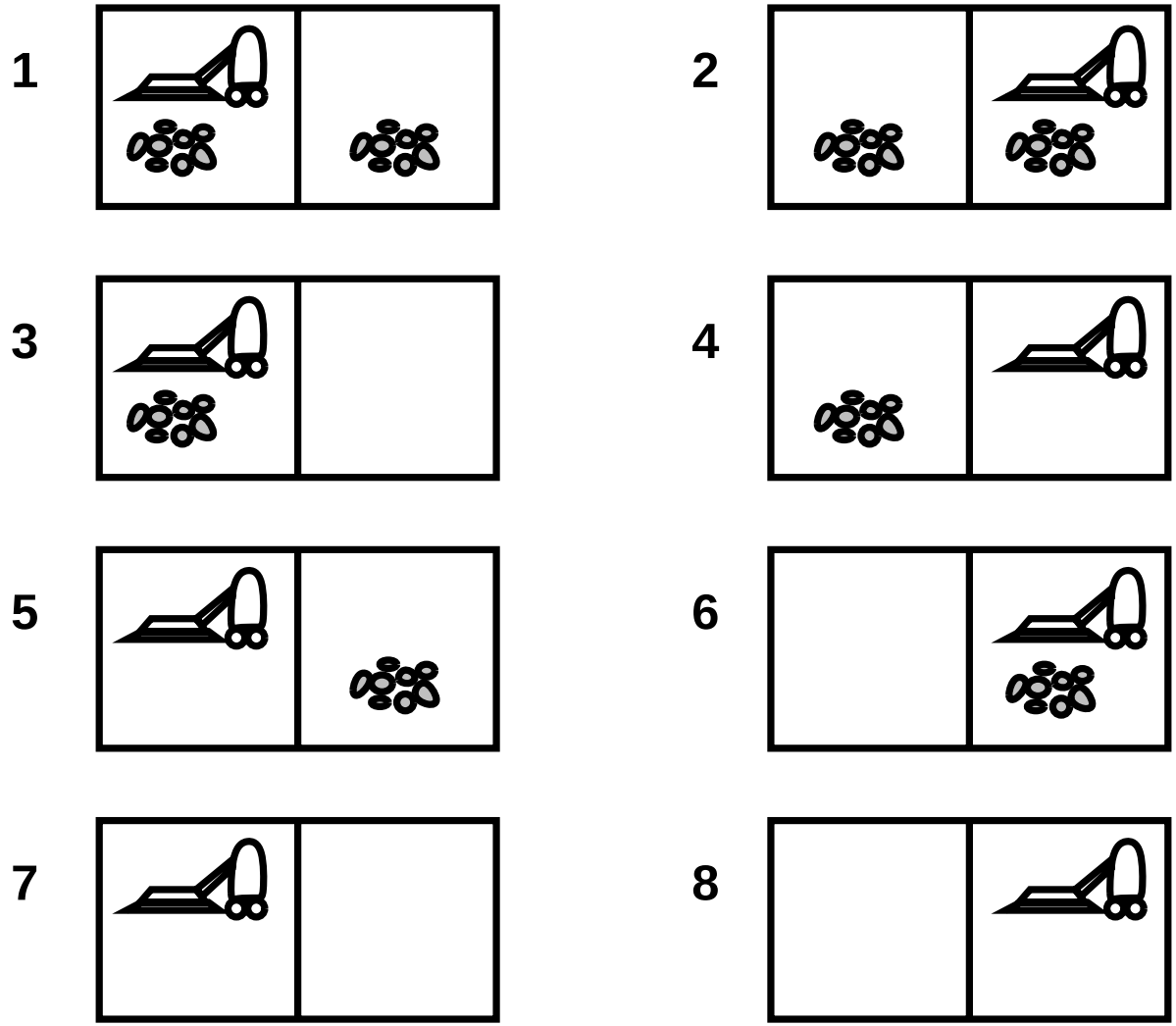


Table 2.6: The vacuum world.

Definition 2.1 (1-state-problem)

A 1-state-problem consists of:

- a set of **states** (incl. the *initial state*)
- a set of n **actions** (*operators*), each of which – applied to a state – leads to another state:

$$\text{Operator}_i\text{: States} \to \text{States}, \quad i = 1, \ldots, n$$

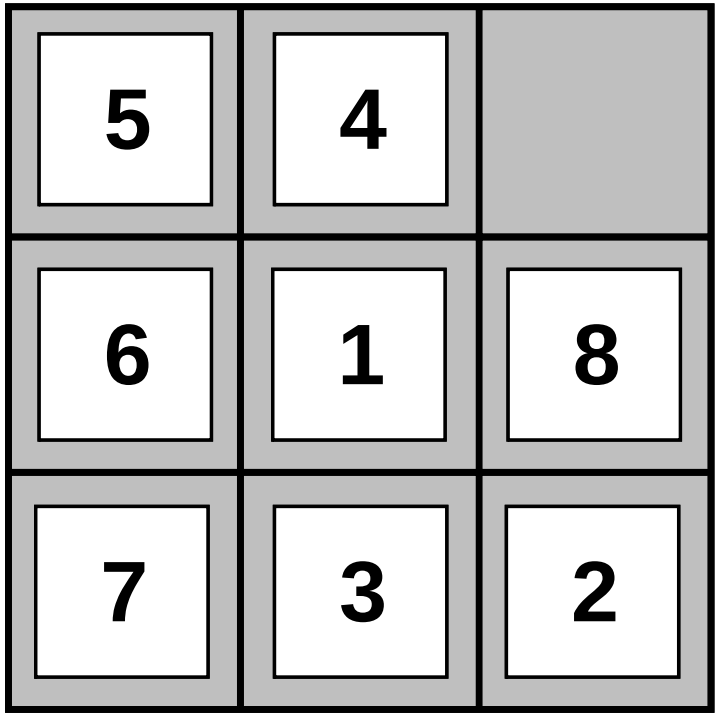
  We use a function **Successor-Fn**: $\mathbf{S} \to 2^{\mathbf{A} \times \mathbf{S}}$. It assigns each state a set of pairs $\langle a, s \rangle$: the set of possible actions and the state it leads to.
- a set of **goal states** or a *goal test*, which – applied on a state – determines if it is a goal-state or not.
- a **cost function** g, which assesses every path in the state space (set of reachable states) and is usually additive.

What about multiple-state-problems?
($\rightsquigarrow$ excercise)

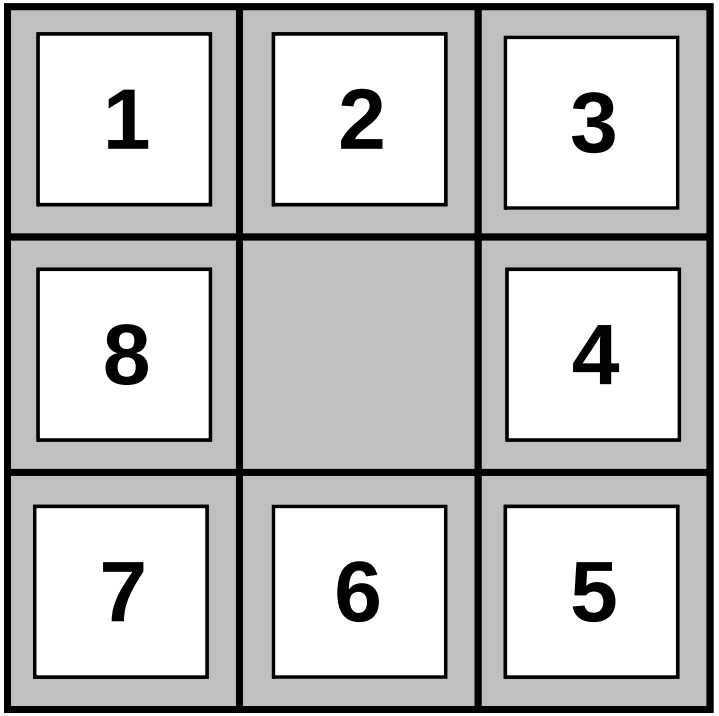
How to choose actions and states?
($\rightsquigarrow$ abstraction)

Definition 2.2 (State Space)

The state space of a problem is the set of all reachable states (from the initial state). It forms a directed graph with the states as nodes and the arcs the actions leading from one state to another.



Start State



Goal State

Table 2.7: The 8-puzzle.

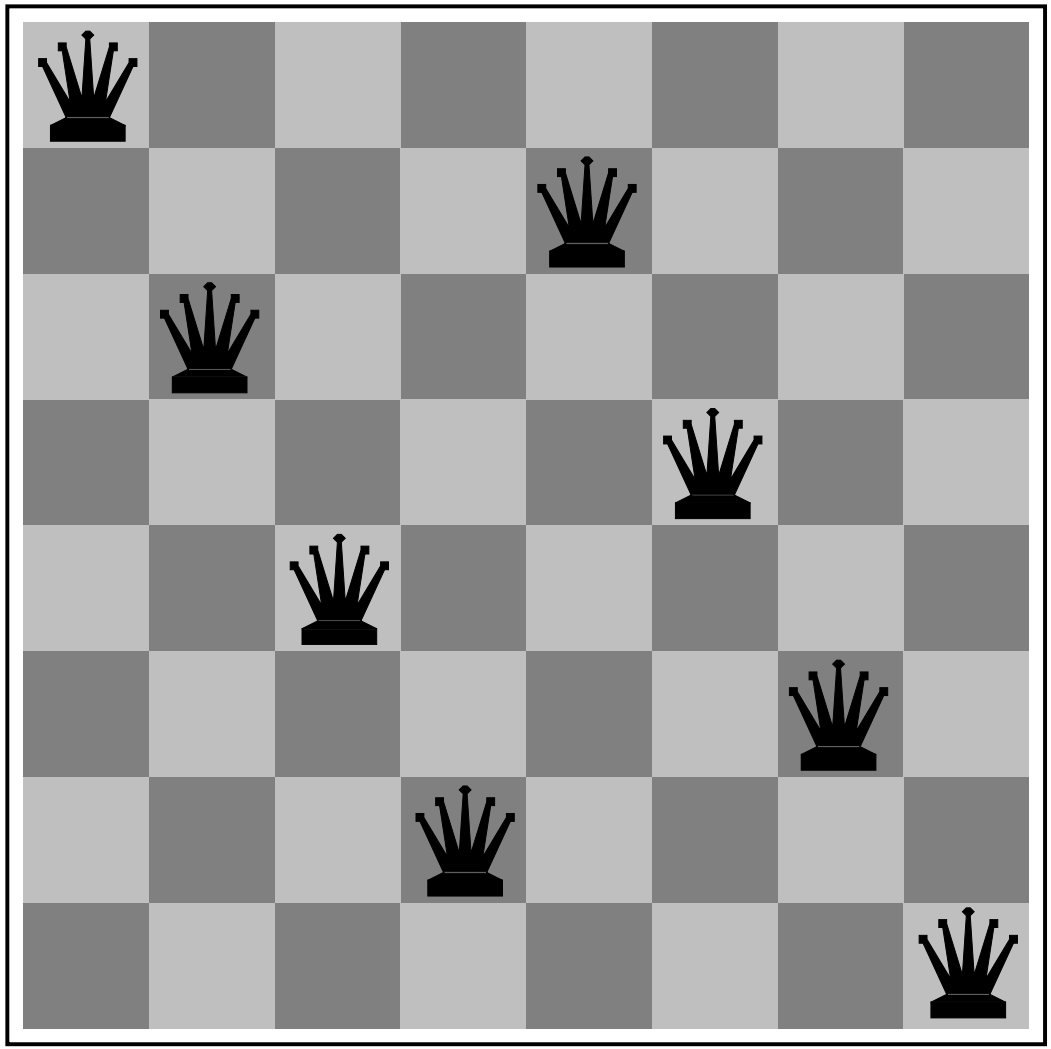


Table 2.8: The 8-queens problem.

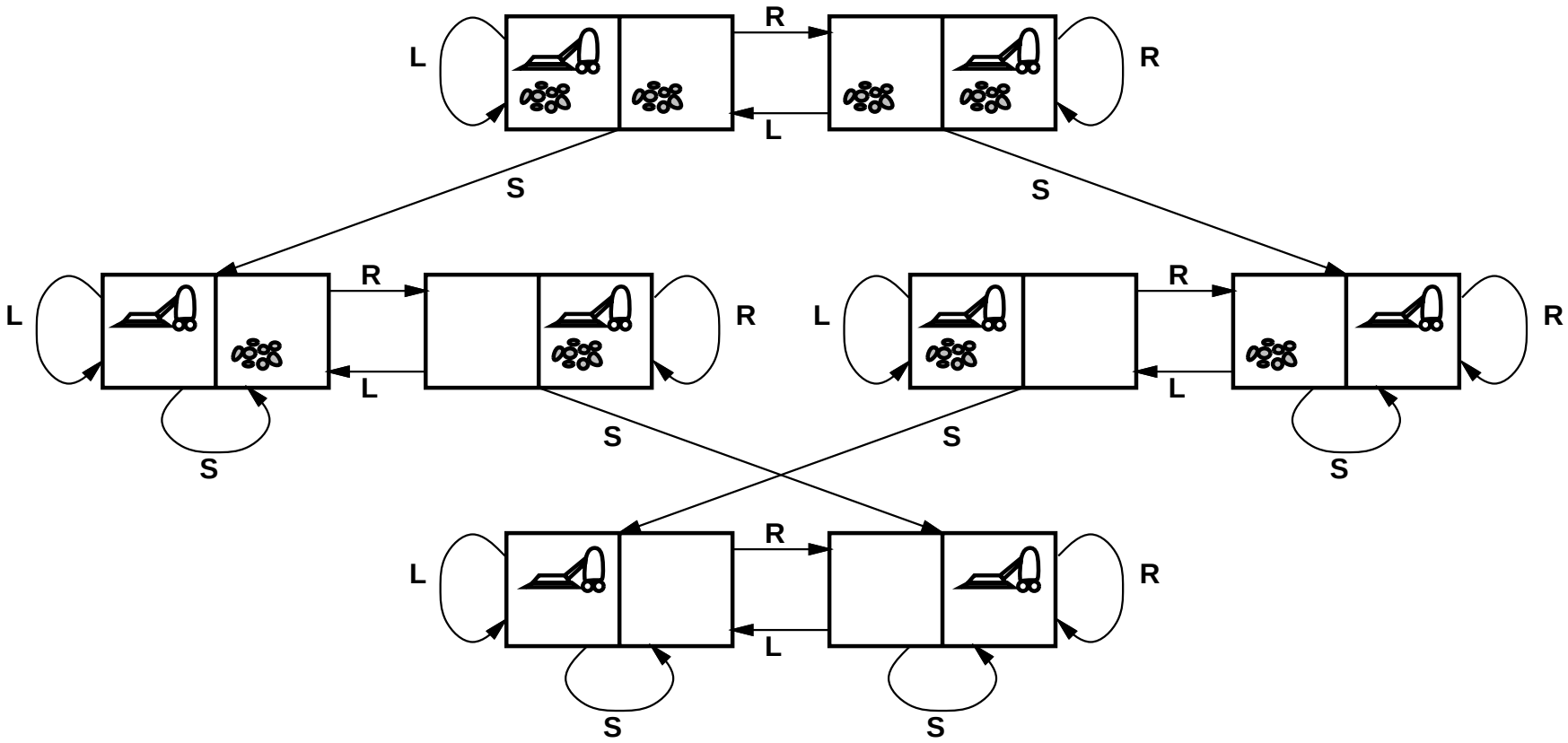


Table 2.9: State Space for Vacuum world.

- Missionaries vs. cannibals

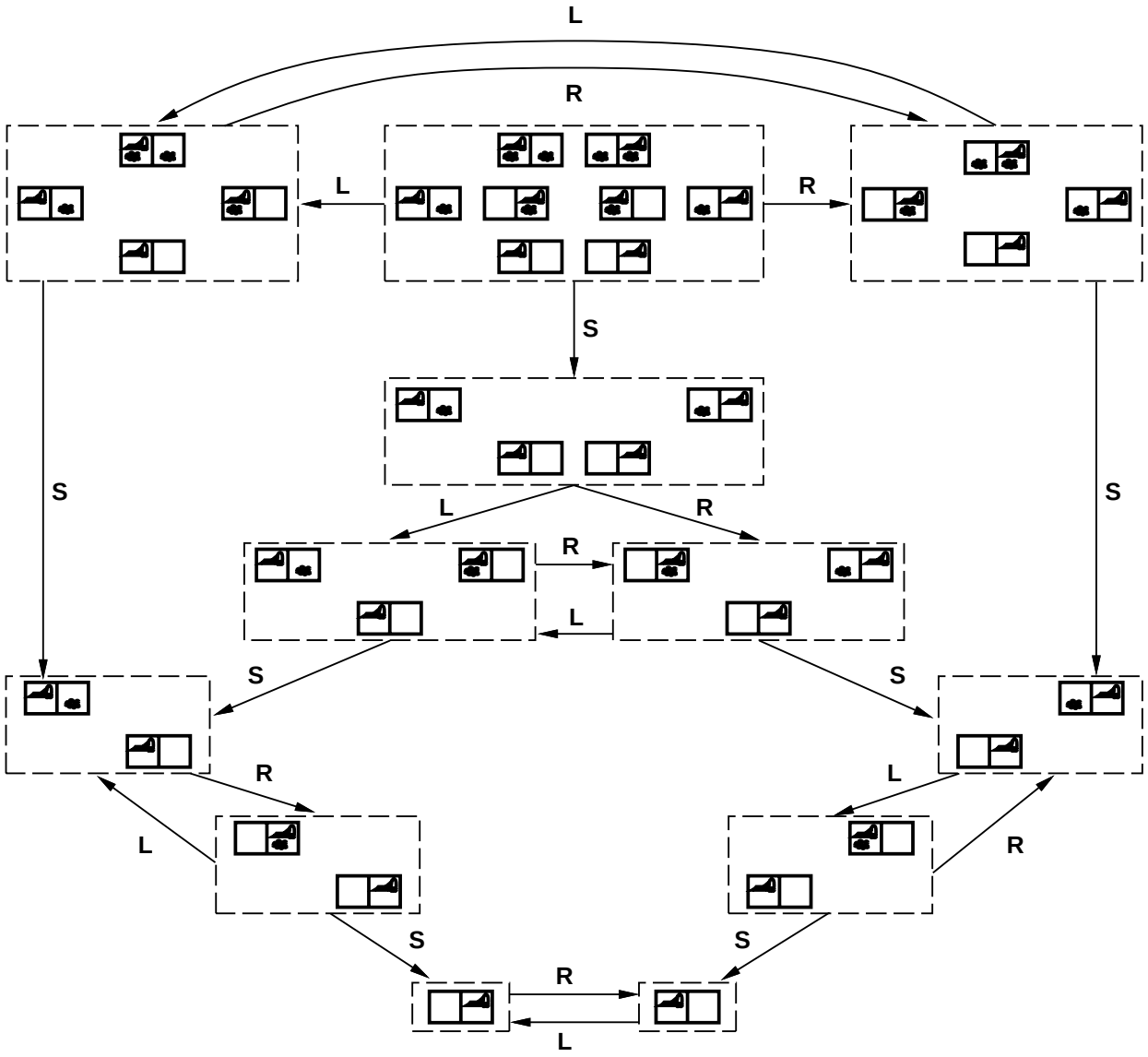
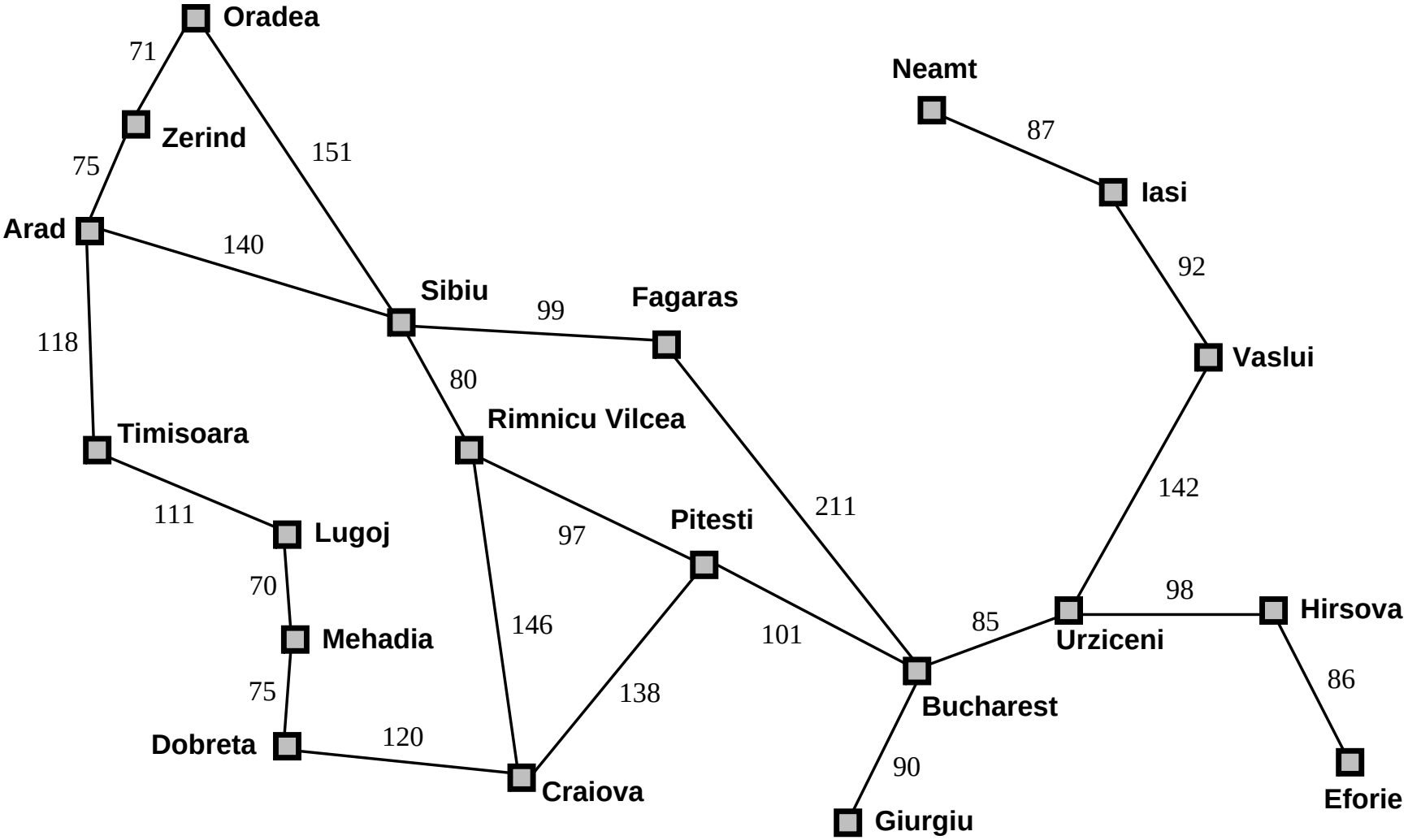


Table 2.10: Belief Space for Vacuum world without sensors.

Real-world-problems:

- travelling-salesman-problem
- VLSI-layout
- labelling maps
- robot-navigation

2.2 Uninformed search

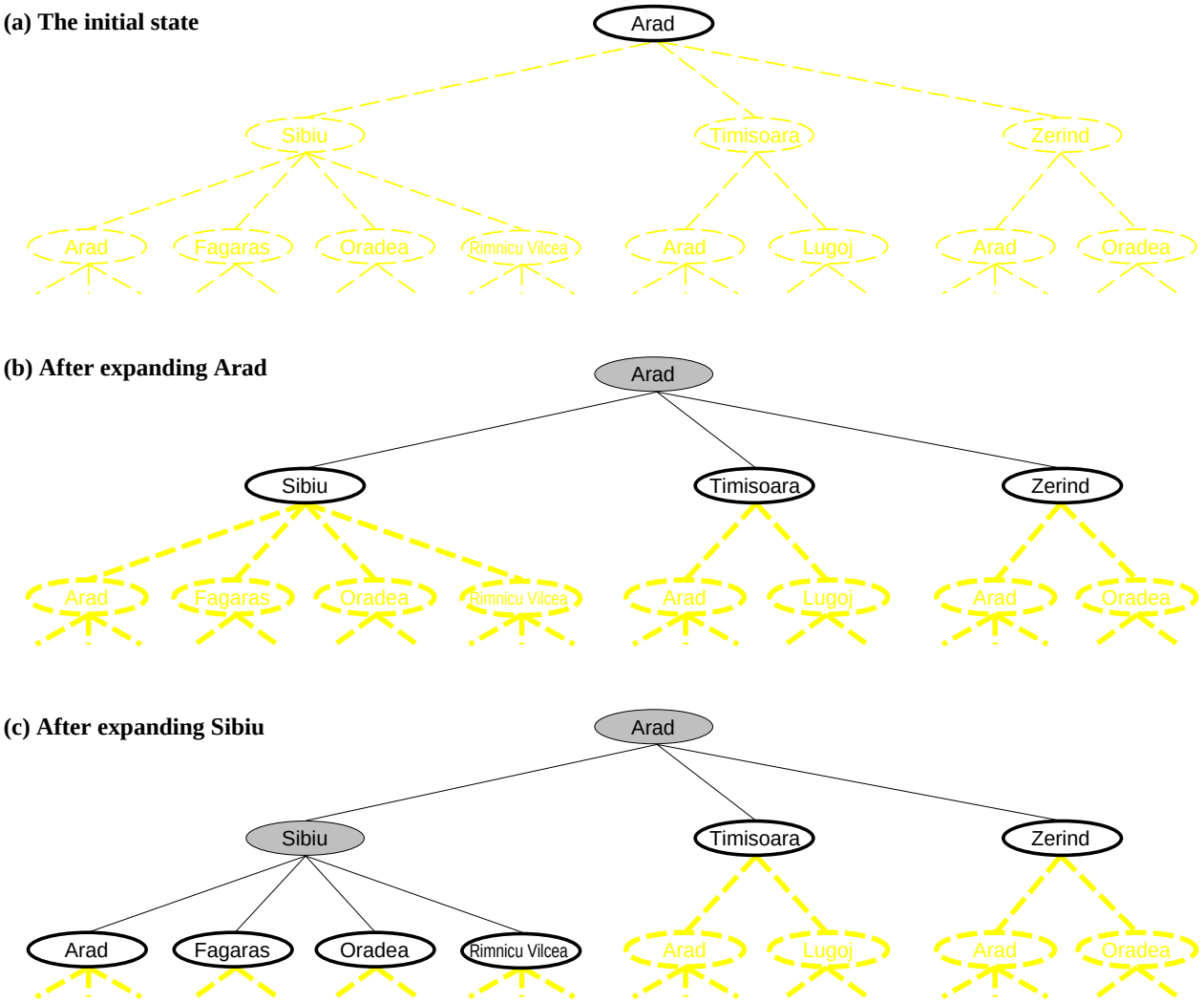


▸ Choose, test, expand ▸ RBFS

Table 2.11: Map of Romania

Principle: Choose, test, expand.

Search-tree



Tree Search

function TREE-SEARCH(*problem*, *strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree

Table 2.12: Tree Search.

Important:

State-space versus **search-tree**:

The search-tree is countably infinite in contrast to the finite state-space.

- a node is **a bookkeeping data structure** with respect to the problem instance and with respect to an algorithm;
- a **state** is a **snapshot** of the world.

Definition 2.3 (Datatype Node)

The datatype **node** is defined by state ($\in$ S), parent (a node), action (also called operator) which **generated** this node, path-costs (the costs to reach the node) and depth (distance from the root).

▸ Tree-Search

Important:

The recursive dependency between node and parent is important. If the depth is left out then a special node $root$ has to be introduced.

Conversely the $root$ can be defined by the depth: $root$ is its own parent with depth 0.

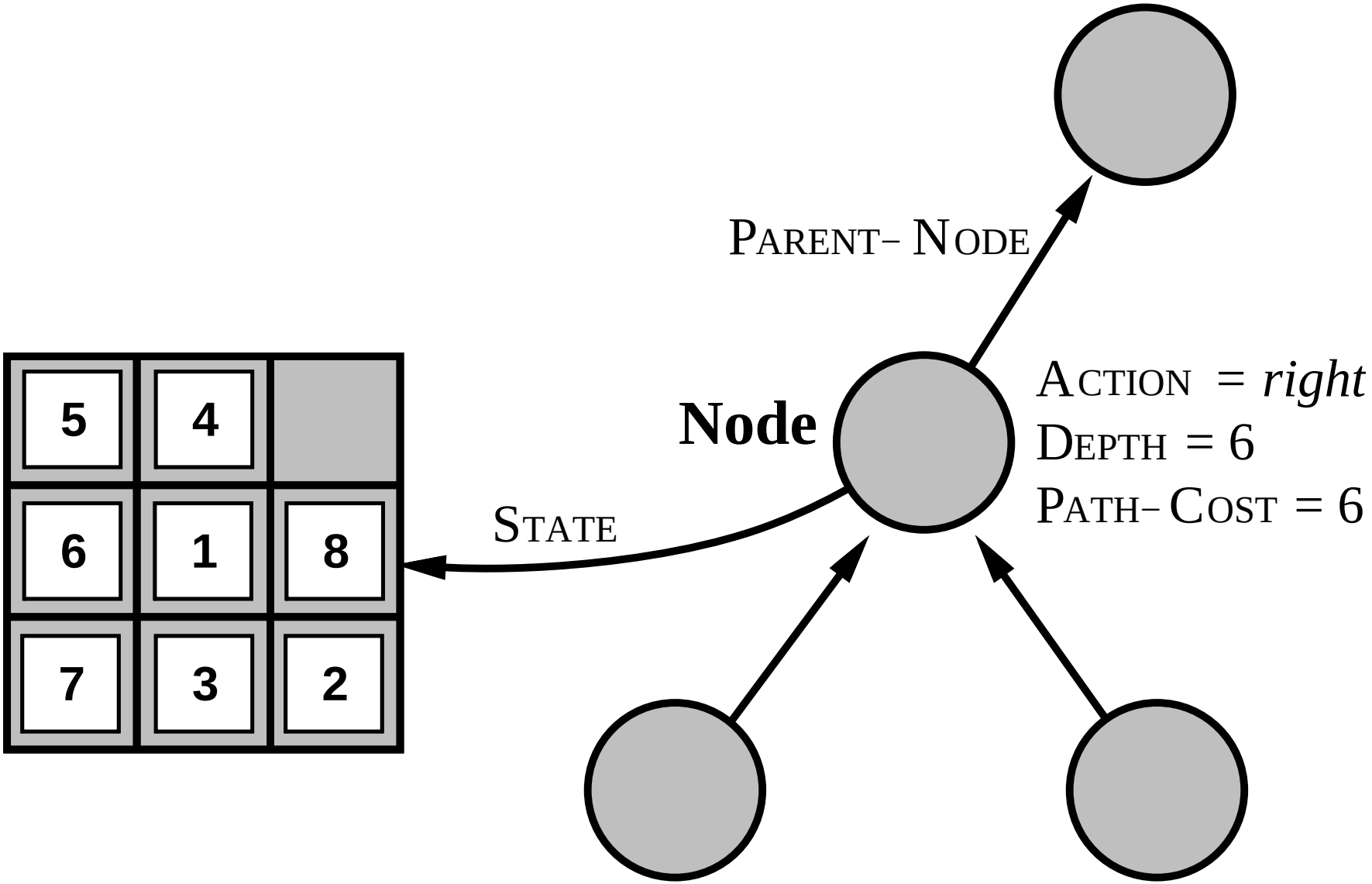


Figure 2.6: Illustration of a node in the 8-puzzle.

Instantiating Tree-SEARCH

Design-decision: Queue

Tree-SEARCH generates nodes. Among them are those that **are-to-be-expanded** later on. Rather than describing them as a set, we use a queue instead.
The fringe is the set of generated nodes that have not yet been expanded.

Here are a few functions operating on queues:

Make-Queue(Elements)
Empty?(Queue)
First(Queue)
Remove-First(Queue)
Insert(Element,Queue)
Insert-All(Elements,Queue)

function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
  if EMPTY?(*fringe*) **then return** failure
  node ← REMOVE-FIRST(*fringe*)
  if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds
    then return SOLUTION(*node*)
  fringe ← INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

---

function EXPAND(*node*, *problem*) **returns** a set of nodes

successors ← the empty set
for each ⟨*action*, *result*⟩ **in** SUCCESSOR-FN[*problem*](STATE[*node*]) **do**
  s ← a new NODE
  STATE[*s*] ← *result*
  PARENT-NODE[*s*] ← *node*
  ACTION[*s*] ← *action*
  PATH-COST[*s*] ← PATH-COST[*node*] + STEP-COST(*node*, *action*, *s*)
  DEPTH[*s*] ← DEPTH[*node*] + 1
  add *s* to *successors*
return *successors*

▸ Datatype Node

▸ Graph-Search

Table 2.13: Tree-Search

Question:

Whiat are interesting requirements of search-strategies?

- **completeness**
- **time-complexity**
- **space complexity**
- **optimality (w.r.t. path-costs)**

We distinguish:

Uninformed vs. informed search.

Definition 2.4 (Completeness, optimality)

A search strategy is called

- complete, if it finds a solution, provided there exists one at all.
- optimal, if whenever it produces an output, this output is an optimal solution, i.e. one with the smallest path costs among all solutions.

Is any optimal strategy also complete?
Vice versa?

Breadth-first search: *"nodes with the smallest depth are expanded first"*,

Make-Queue : add new nodes at the end: FIFO

Complete? Yes.
Optimal? Yes, if all operators are equally expensive.

Constant branching-factor b: for a solution at depth d we have generated[1] (in the worst case)

$$b + b^2 + \ldots + b^d + (b^{d+1} - b)\text{-many nodes.}$$

Space complexity = Time Complexity

[1] Note this is different from "expanded".

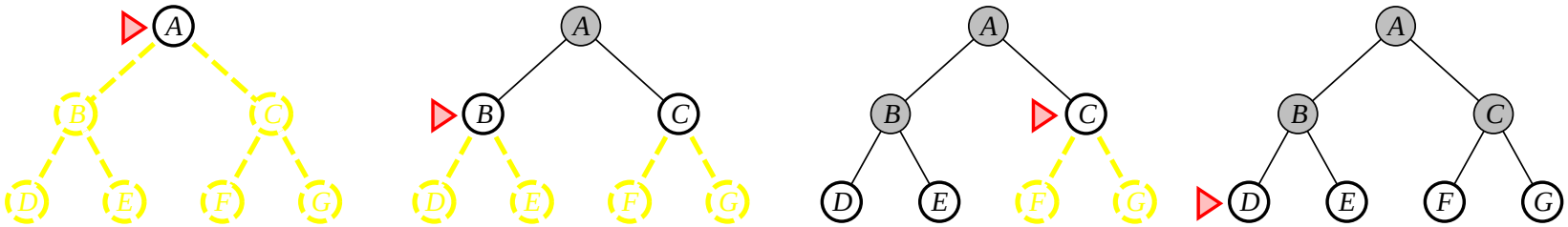


Figure 2.7: Illustration of Breadth-First Search.

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Table 2.14: Time versus Memory.

Uniform-Cost-Search: *"Nodes n with lowest path-costs $g(n)$ are expanded first"*

Make-Queue : new nodes are compared to those in the queue according to their path costs and are inserted accordingly

Complete? Yes, if each operator increases the path-costs by a minimum of $\delta > 0$ (see below).

Worst case space/time complexity: $\mathrm{O}(b^{1+\lfloor \frac{C^*}{\delta} \rfloor})$, where C^* is the cost of the optimal solution and each action costs at least δ

If all operators have the same costs (in particular if $g(n) = \text{depth}(n)$ holds):

Uniform-cost search

Uniform-cost search=**Breadth-first search**.

Theorem 2.5 (Optimality of Uniform-cost search)

If $\exists \delta > 0 : \ g(succ(n)) \geq g(n) + \delta$ *then: Uniform-cost search is* optimal.

Depth-first search: *"Nodes of the greatest depth are expanded first"*,

Make-Queue : LIFO, new nodes are added at the beginning

If branching-factor b is constant and the maximum depth is m then:

- **Space-complexity:** $b \times m$,
- **Time-complexity:** b^m.

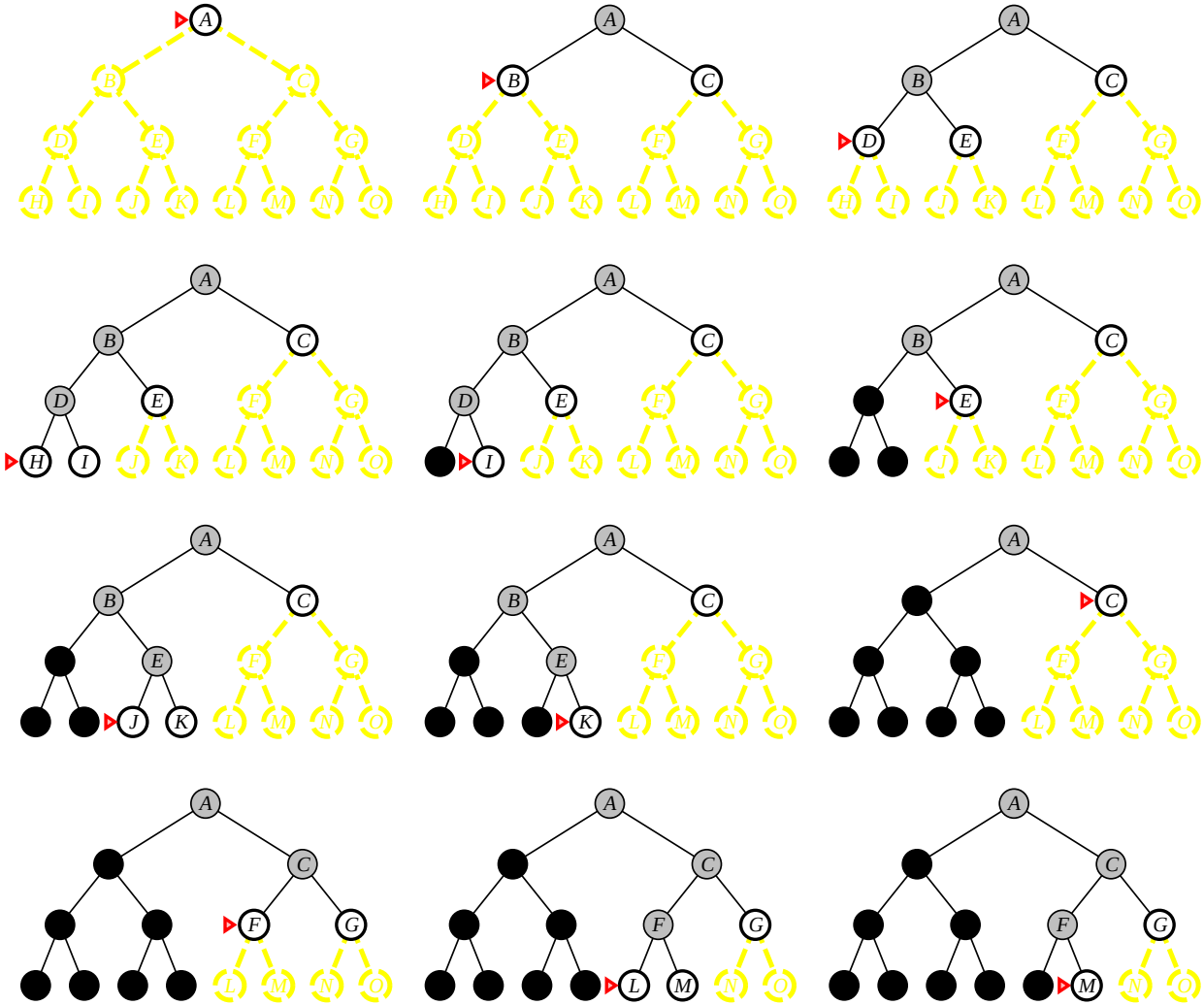


Table 2.15: Illustration of Depth-First-Search.

Depth-limited search: *"search to depth d"*.

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[*problem*]), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff
  cutoff_occurred? $\leftarrow$ false
  if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
  else if DEPTH[*node*] = *limit* **then return** *cutoff*
  else for each *successor* **in** EXPAND(*node*, *problem*) **do**
    result $\leftarrow$ RECURSIVE-DLS(*successor*, *problem*, *limit*)
    if *result* = *cutoff* **then** *cutoff_occurred*? $\leftarrow$ true
    else if *result* $\neq$ *failure* **then return** *result*
  if *cutoff_occurred*? **then return** *cutoff* **else return** *failure*

Table 2.16: Depth-Limited-Search.

In total: $1 + b + b^2 + \ldots + b^{d-1} + b^d + (b^{d+1} - b)$ many nodes.

- **Space-complexity:** $b \times l$,
- **Time-complexity:** b^l.

Two different sorts of failures!

Iterative-deepening search: *"depth increases"*

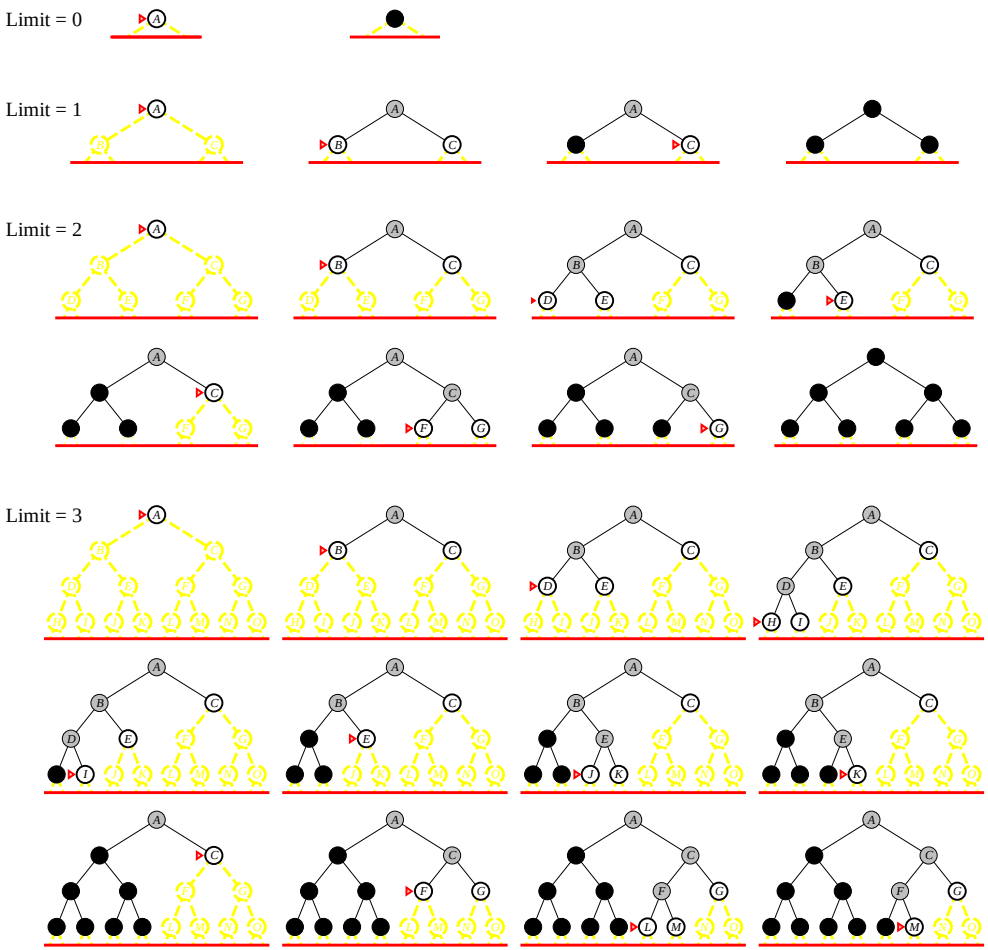


Table 2.17: Illustration of Iterative-Deepening-Search.

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  inputs: *problem*, a problem

  for *depth* $\leftarrow$ 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

Table 2.18: Iterative-Deepening-Search.

How many nodes?

$$d \times b + (d-1) \times b^2 + \ldots + 2 \times b^{d-1} + 1 \times b^d.$$

Compare with Breadth-first search:

$$b + b^2 + \ldots + b^d + (b^{d+1} - b).$$

Attention:

Iterative-deepening search is faster than Breadth-first search because the latter also generates nodes at depth $d+1$ (even if the solution is at depth d).

The amount of revisited nodes is kept within a limit.

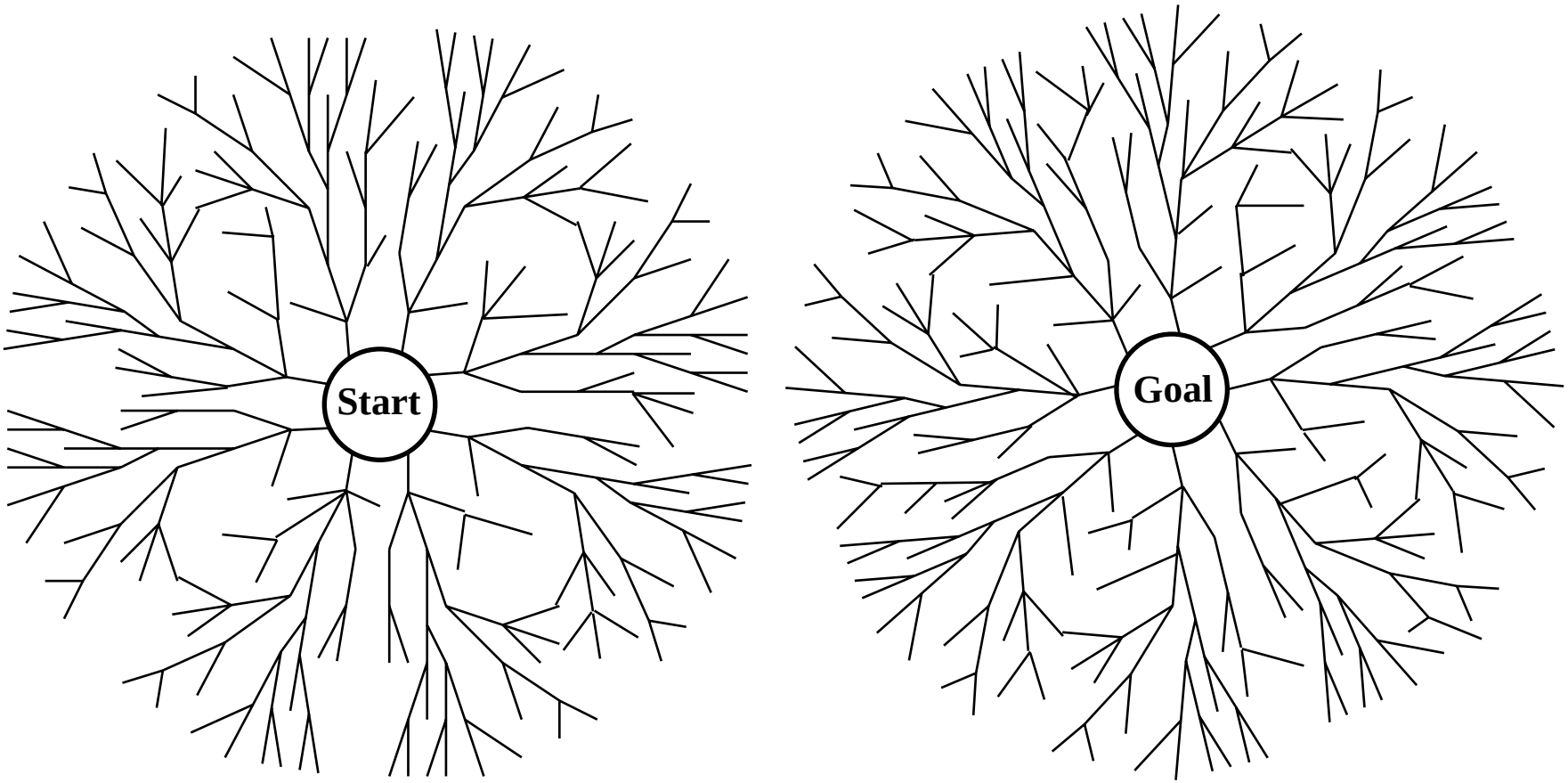


Figure 2.8: Bidirectional search.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deep.	Bi-direct.
Complete	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Time	$\mathcal{O}(b^{d+1})$	$\mathcal{O}\left(b^{\left\lceil \frac{C^*}{\varepsilon} \right\rceil}\right)$	$\mathcal{O}(b^m)$	$\mathcal{O}(b^l)$	$\mathcal{O}(b^d)$	$\mathcal{O}\left(b^{\frac{d}{2}}\right)$
Space	$\mathcal{O}(b^{d+1})$	$\mathcal{O}\left(b^{\left\lceil \frac{C^*}{\varepsilon} \right\rceil}\right)$	$\mathcal{O}(bm)$	$\mathcal{O}(bl)$	$\mathcal{O}(bd)$	$\mathcal{O}\left(b^{\frac{d}{2}}\right)$
Optimal	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]

Table 2.19: Complexities of uninformed search procedures. Cells regarding time and space denote the nodes generated.

[1] if b is finite;
[2] if step costs $\geq \varepsilon$ for positive ε;
[3] if step costs are all identical;
[4] if both directions use Breadth-first search

b branching factor, $b \geq 2$

d depth of the shallowest solution, $d < m$

m maximum depth of the search tree

l depth limit, $l \leq m$

C^* cost of the optimal solution

How to avoid repeated states?

- Can we avoid infinite trees by checking for loops?
- Compare number of states with number of paths in the search tree.

State space vs. Search tree

Rectangular grid: How many different states are reachable within a path of length d?

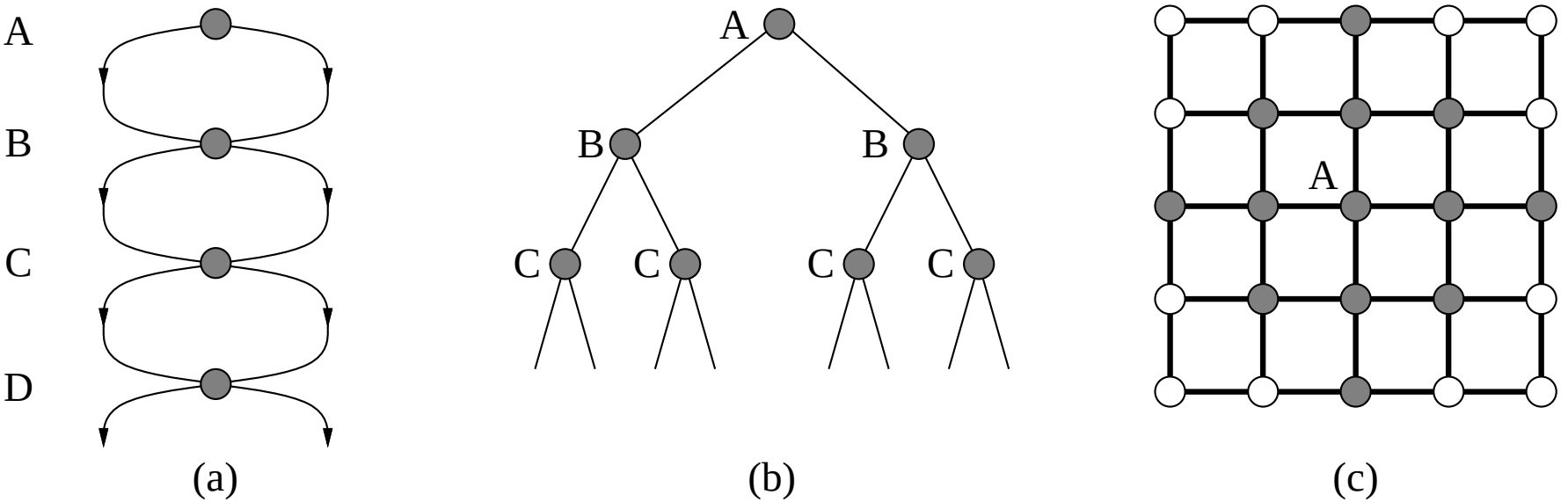


Table 2.20: State space versus Search tree: exponential blow-up.

Graph-Search= Tree-Search+ Loop-checking

▸ Tree-Search

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
  if EMPTY?(*fringe*) **then return** failure
  node ← REMOVE-FIRST(*fringe*)
  if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
  if STATE[*node*] is not in *closed* **then**
    add STATE[*node*] to *closed*
    fringe ← INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

2.3 Best-First Search

Idea:

Use problem-specific knowledge to improve the search.

Tree-search is precisely defined. Only freedom: Make-Queue.

Let's assume we have an evaluation-function f which assigns a value $\mathrm{f}(n)$ to each node n.

We change Make-Queue as follows

the nodes with smallest f are located at the beginning of the queue

- thus the queue is sorted wrt. f.

function BEST-FIRST-SEARCH(*problem*, EVAL-FN) **returns** a solution sequence
  inputs: *problem*, a problem
    Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(*problem*, *Queueing-Fn*)

Table 2.22: Best-First-Search.

What about time and space complexity?

Best-first search: here the evaluation-function is

$f(n)$:= expected costs of an optimal path from the state in n to a goal state.

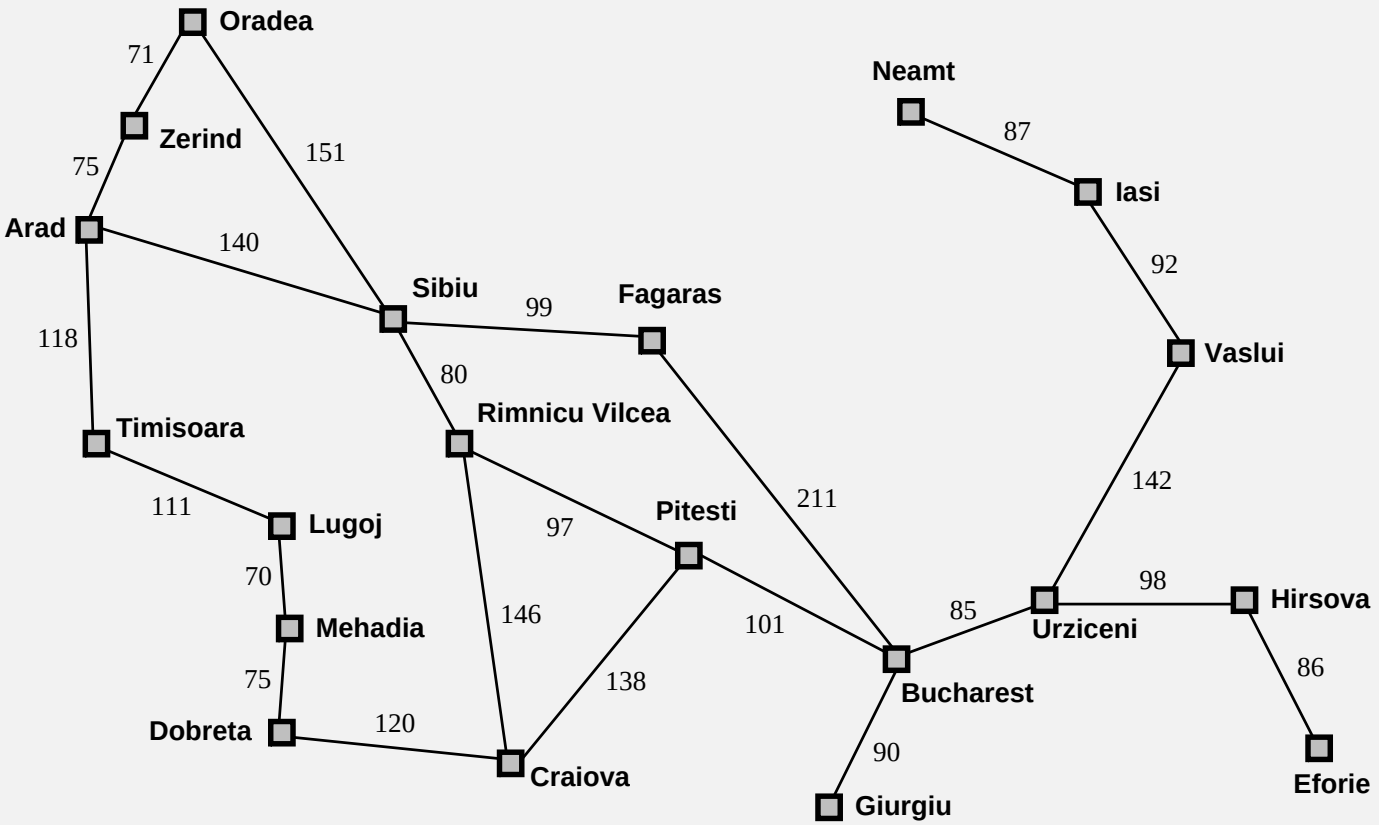
The word optimal is used with respect to the given cost-function $\mathbf{g}$.

This evaluation-function is also called heuristic function, written $\mathbf{h}(n)$.

Heuristic Function

We require from all heuristic functions that they assign the value 0 to goal states.

Example 2.6 (path-finding)



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

$\mathbf{h}(n)$ might be defined as the direct-line distance between Bucharest and the city denoted by n.

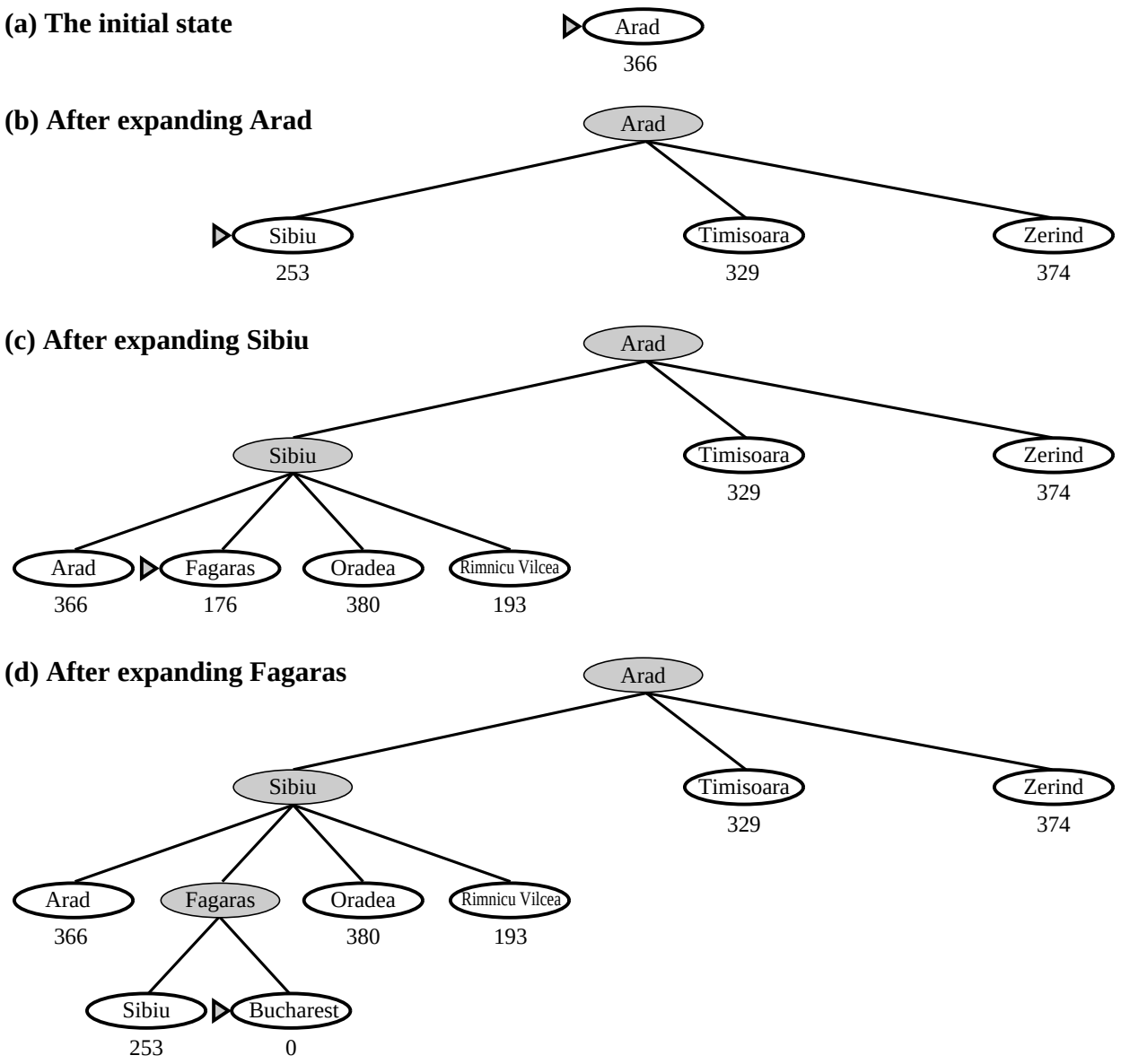


Table 2.23: Illustration of Best-first search.

Questions:

1. Is Best-first search optimal?
2. Is Best-first search complete?

2.4 A* Search

Definition of A*

A* search: Here the evaluation-function is the sum of an heuristic function $\mathbf{h}(n)$ and the real path-costs $\mathbf{g}(n)$:

$$f(n) := \mathbf{h}(n) + \mathbf{g}(n).$$

So A* search is "best-first + uniform-cost", because $\mathbf{h}(n_z) = 0$ holds for final states n_z, as $\mathbf{f}(n_z) = \mathbf{g}(n_z)$.

The notion of admissibility

On Slide 140 we required from any heuristic function that its value is 0 for goal nodes.
An important generalization of this is that it never **overestimates** the cost to reach the goal.

Definition 2.7 (Admissible heuristic function)

The heuristic function $\mathbf{h}$ is called admissible if $\mathbf{h}(n)$ is always smaller or equal than the optimal costs $\mathbf{h}^*$ from n to a goal-node:

$$\mathbf{h}(n) \leqq \mathbf{h}^*(n)$$

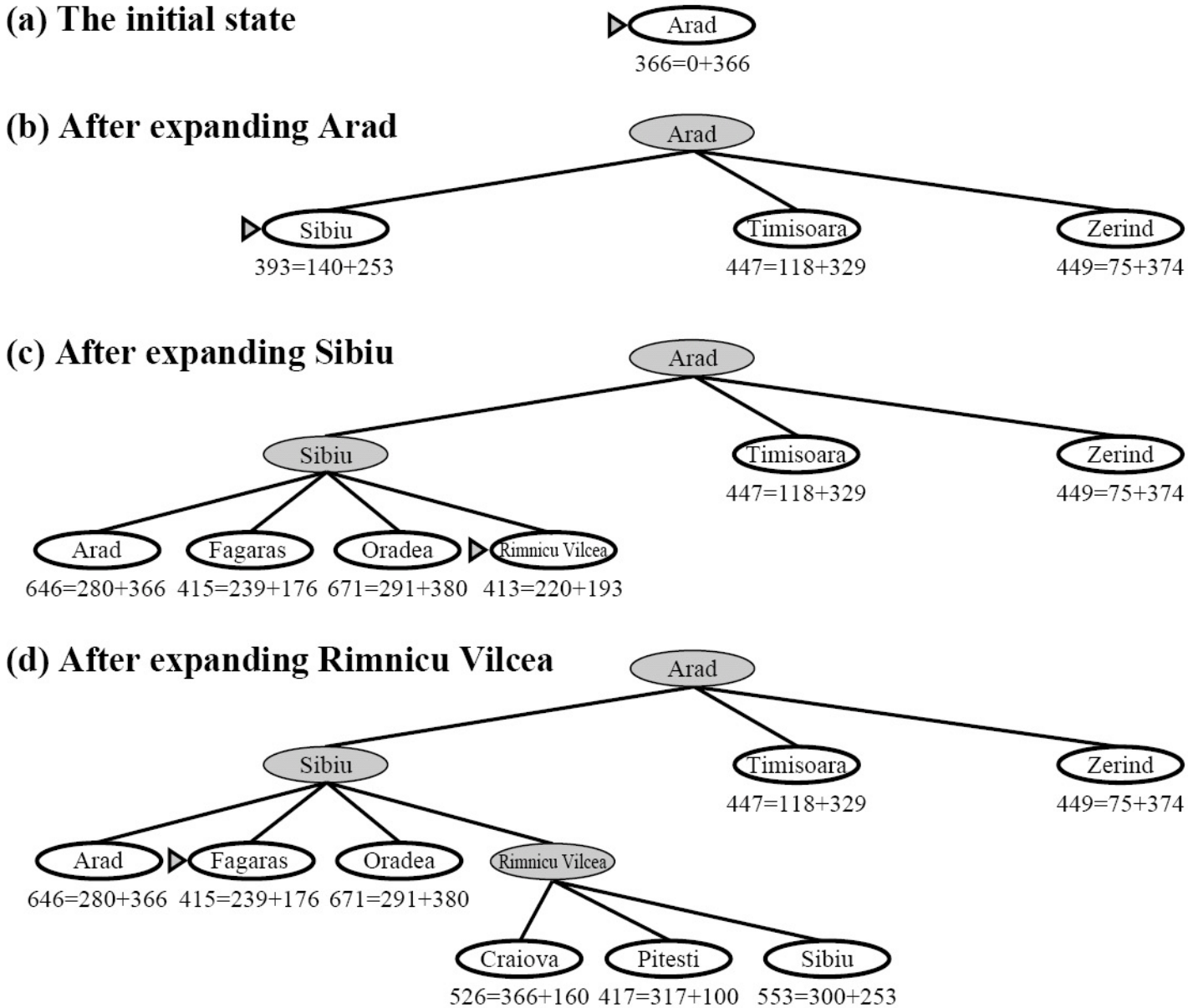


Table 2.24: Illustration of A* (1).

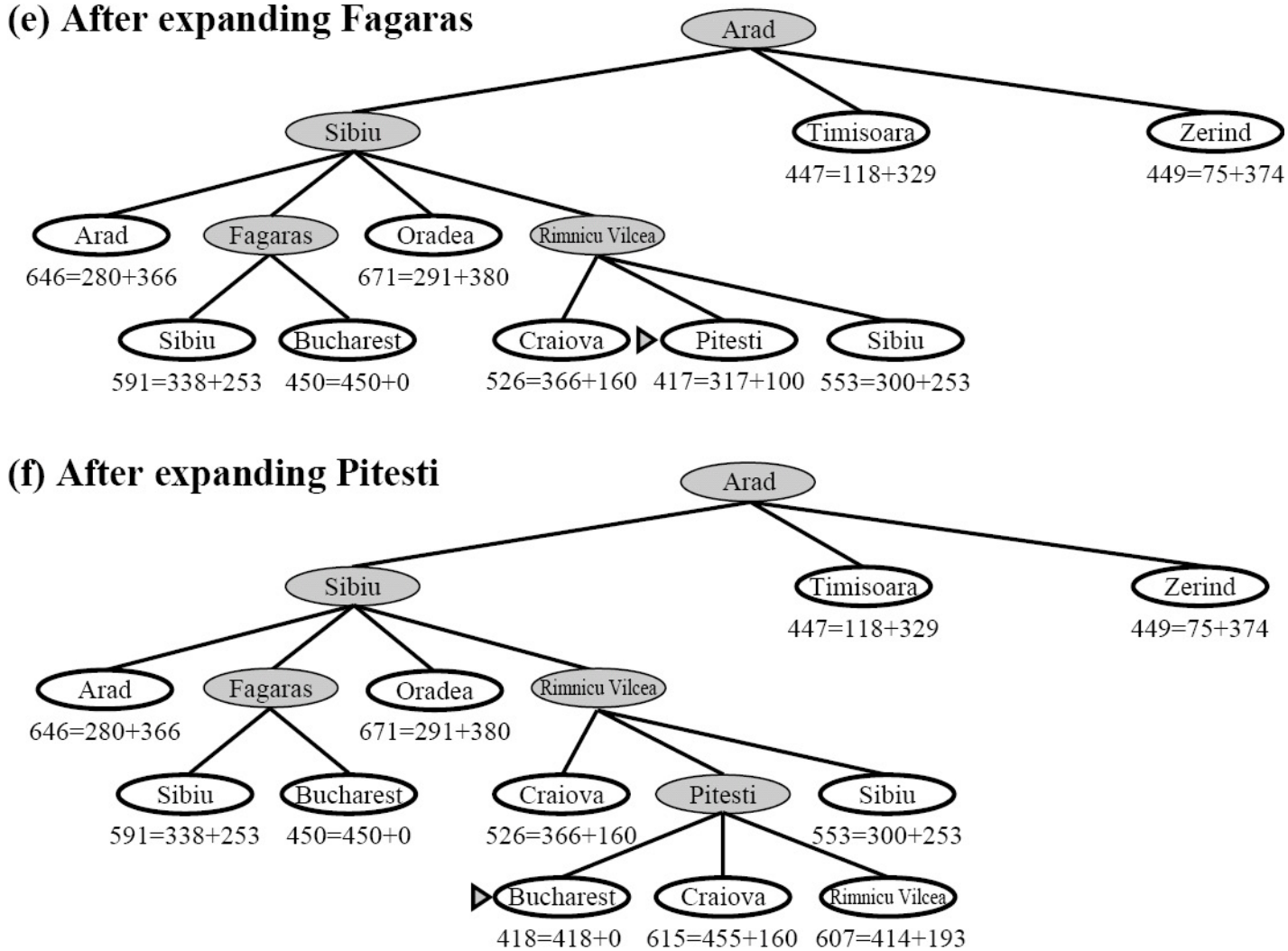


Table 2.25: Illustration of A* (2).

To show completeness of A* we have to ensure:

1. Never it is the case that an infinite number of nodes is generated in one step (locally finite).
2. There is a $\delta > 0$ such that in each step the path costs increase by at least δ.

These conditions must also hold in the following optimality results.

Theorem 2.8 (Completeness of A*)

A is complete (wrt. **Tree Search** or **Graph Search**), if the above two properties are satisfied.*

- f is monotone in our example.
- **This does not hold in general**.
- Monotony of f is **not needed** to ensure optimality.
- But if the heuristic function is admissible, then we can easily modify f to be monotone (how?) and make the search more efficient.

Theorem 2.9 (Optimality of A* wrt Tree Search)

A is optimal using Tree Search, if the heuristic function* $\mathbf{h}$ *is admissible.*

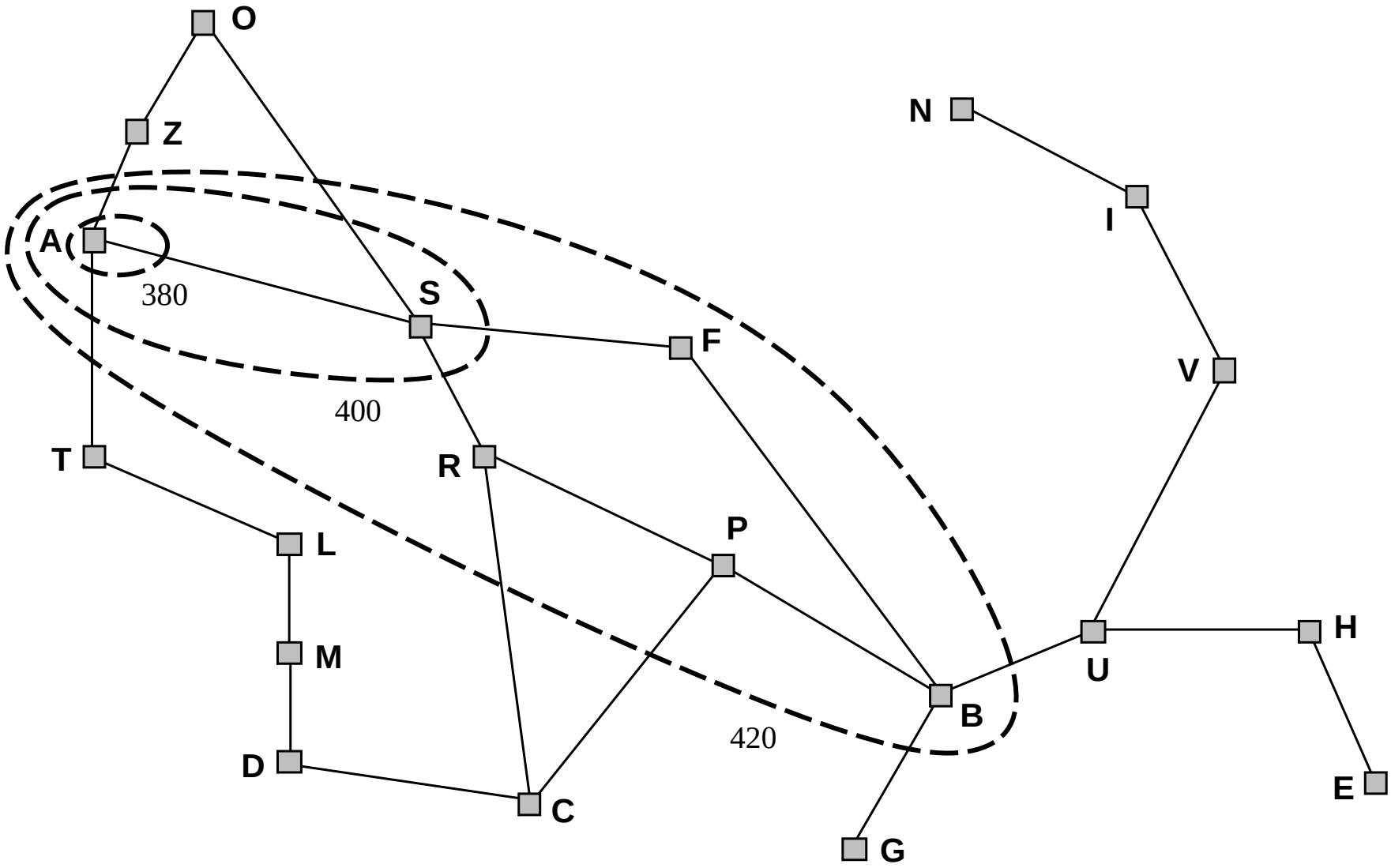


Figure 2.9: Goal-directed contours of A*.

What if we use Graph Search?

The proof breaks down!

The notion of consistency

Definition 2.10 (Consistent heuristic function)

The heuristic function $\mathbf{h}$ is called consistent if the following holds for every node n and successor n' of n:

$$\mathbf{h}(n) \leq cost(n, a, n') + \mathbf{h}(n').$$

Consistency of h implies monotony of f.

Is the converse also true?

Theorem 2.11 (Optimality of A* wrt Graph Search)

A is optimal using Graph Search, if the heuristic function* $\mathbf{h}$ *is consistent.*

Is the last theorem also true if we require monotony of f (instead of consistency of $\mathbf{h}$)?

Question:

How many nodes does A* store in memory?

Answer:

Virtually always exponentially many with respect to the length of the solution.

It can be shown: As long as the heuristic function is not **extremely exact**

$$|\mathbf{h}(n) - \mathbf{h}^*(n)| < \mathbf{O}(\log \mathbf{h}^*(n))$$

the amount of nodes is always exponential with respect to the solution.

For almost every usable heuristic a bad error-estimation holds:

$$|\mathbf{h}(n) - \mathbf{h}^*(n)| \approx \mathbf{O}(\mathbf{h}^*(n))$$

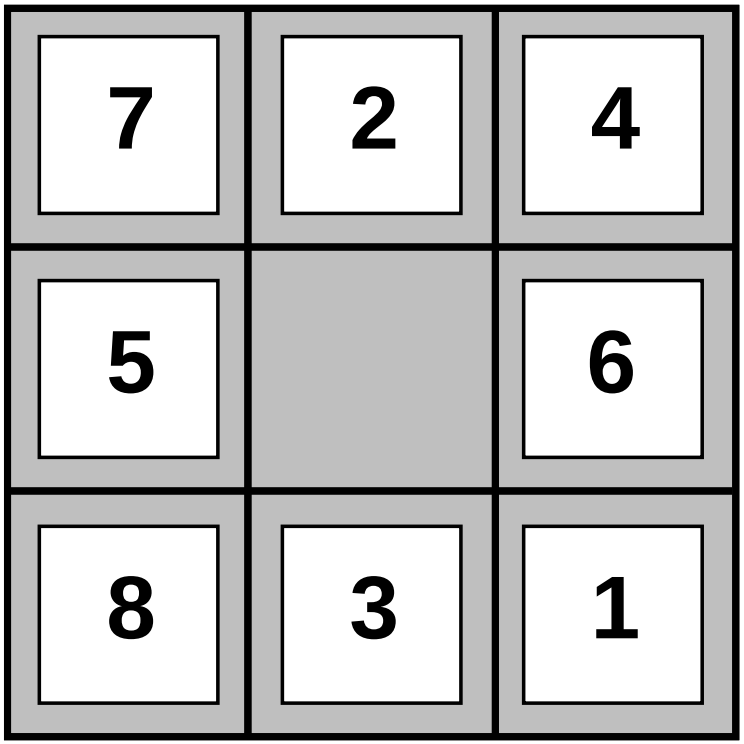
Important:

A*'s problem is space not time.

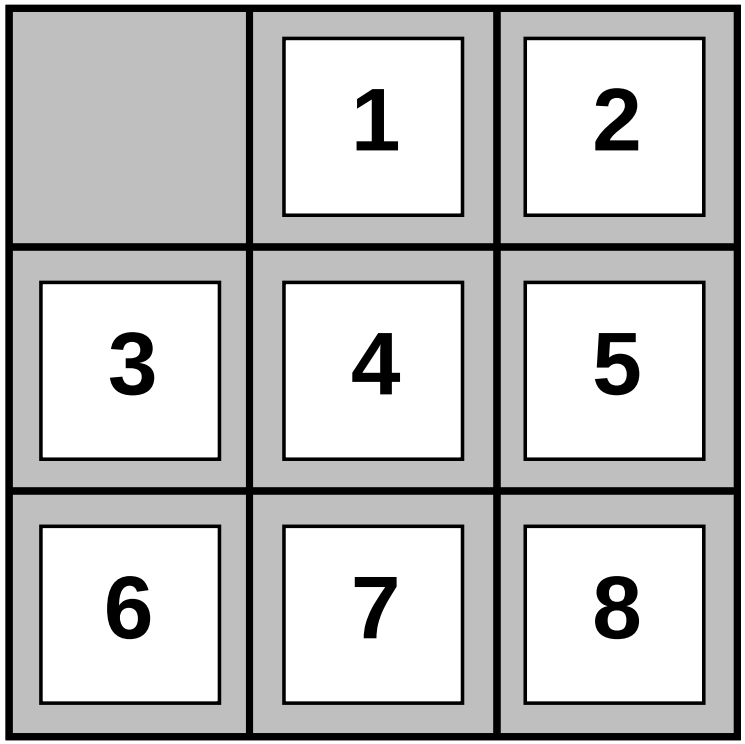
Important:

A* is even optimally efficient: No other **optimal** algorithm (which expands search-paths beginning with an initial node) **expands less nodes** than A*.

2.5 Heuristics



Start State



Goal State

Table 2.26: An instance of the 8-puzzle.

Question:

Which branching-factor?

Answer:

Approx. 3 (more exactly $\frac{8}{3}$).

Question:

How many nodes have to be considered?

Answer:

$$3^g \approx 10^{10}$$

in which g is the amount of moves necessary to get a solution. g is approx. 22.

Checking for loops

But: There are only $9! \approx 10^5$ states!

In other words: Looking at cycles can be very helpful.

Question:

Which heuristic functions come in handy?

Hamming-distance: h_1 is the amount of numbers which are in the wrong position. I.e. $h_1(start) = 8$.

Manhattan-distance: Calculate for every piece the distance to the right position and sum up:

$$h_2 := \sum_{i=1}^{8} (\text{distance of } i \text{ to the right position})$$

$$h_2(start) = 2+3+2+1+2+2+1+2 = 15.$$

Question:

How to determine the quality of a heuristic function? (In a single value)

Definition 2.12 (Effective Branching Factor)

Suppose A* detects an optimal solution for an instance of a problem at depth d with N nodes generated. Then we define $\mathbf{b}^*$ via $N + 1 = 1 + \mathbf{b}^* + (\mathbf{b}^*)^2 + \ldots + (\mathbf{b}^*)^d$: the effective branching-factor of A*.

Attention:

b^* depends on h and on the special problem instance.

But for many classes of problem instances b^* is quite constant.

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

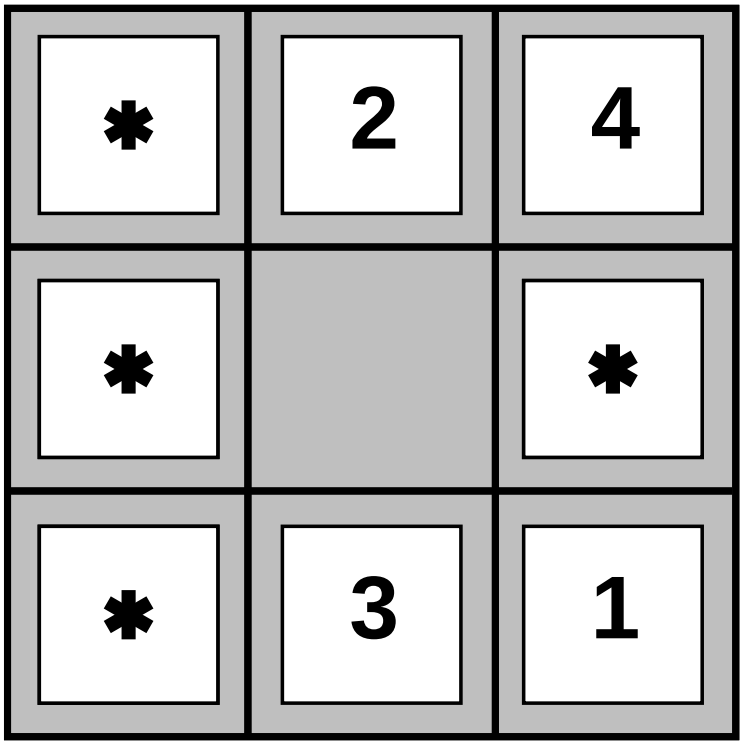
Table 2.27: Comparing A* (Hamming and Manhattan) with IDS.

Question:

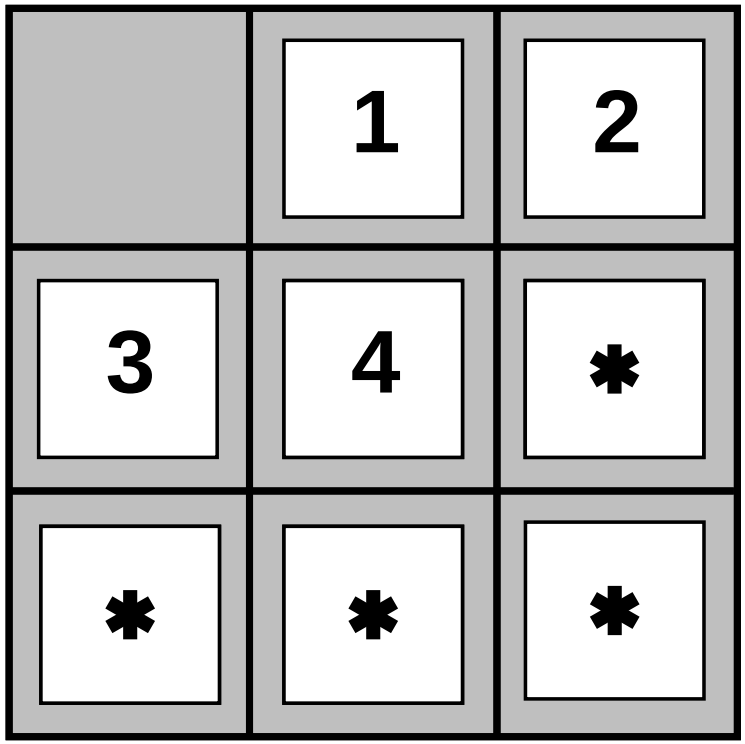
Is Manhattan better than Hamming?

How to determine a (good) heuristics for a given problem?

- **There is no general solution, it always depends on the problem.**
- Often one can consider a **relaxed** problem, and take the precise solution of the relaxed problem as a heuristics for the original one.



Start State



Goal State

Table 2.28: A relaxed version of the 8-puzzle.

Relaxed problem:

Try to bring 1–4 in the right positions, but do not care about all the others.

This heuristics is better than Manhattan distance (for the 8-puzzle).

2.6 Limited Memory

Question:

A* is memory-intensive. What if the memory is limited? What to do if the queue is restricted in its length?

This leads to:

- **IDA***: A* + iterative deepening,
- **RBFS**: Recursive Best-First Search,
- **SMA***: Simplified memory bounded A*.

IDA*: A* combined with iterative-deepening search. We perform Depth-first search (small memory), but use values of f instead of **depth**.

So we consider contures: only nodes within the f-limit. Concerning those beyond the limit we only store the smallest f-value above the actual limit.

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f- COST limit
          root, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f- COST(root)
  loop do
      solution, f-limit ← DFS-CONTOUR(root, f-limit)
      if solution is non-null then return solution
      if f-limit = ∞ then return failure; end
```

```
function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f- COST limit
  inputs: node, a node
          f-limit, the current f- COST limit
  static: next-f, the f- COST limit for the next contour, initially ∞

  if f- COST[node] > f-limit then return null, f- COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
      solution, new-f ← DFS-CONTOUR(s, f-limit)
      if solution is non-null then return solution, f-limit
      next-f ← MIN(next-f, new-f); end
  return null, next-f
```

Figure 2.10: IDA*.

Theorem 2.13 (Properties of IDA*)

IDA* is optimal if enough memory is available to store the longest solution-path with costs $\leq f^*$.

Question:

What about complexity?

space: depends on the smallest operator-costs δ, the branching-factor b and the optimal costs f^*. bf^*/δ-many nodes are generated (worst case)

time: depends on the amount of values of the heuristic function.

If we consider a small amount of values, the last iteration of IDA* will often be like A*.

- Consider a large amount of values. Then only one node per conture will be added: **How many nodes will IDA* visit if A* expands n-many?**
- What does this say about the **time-complexity of A* and IDA*?**

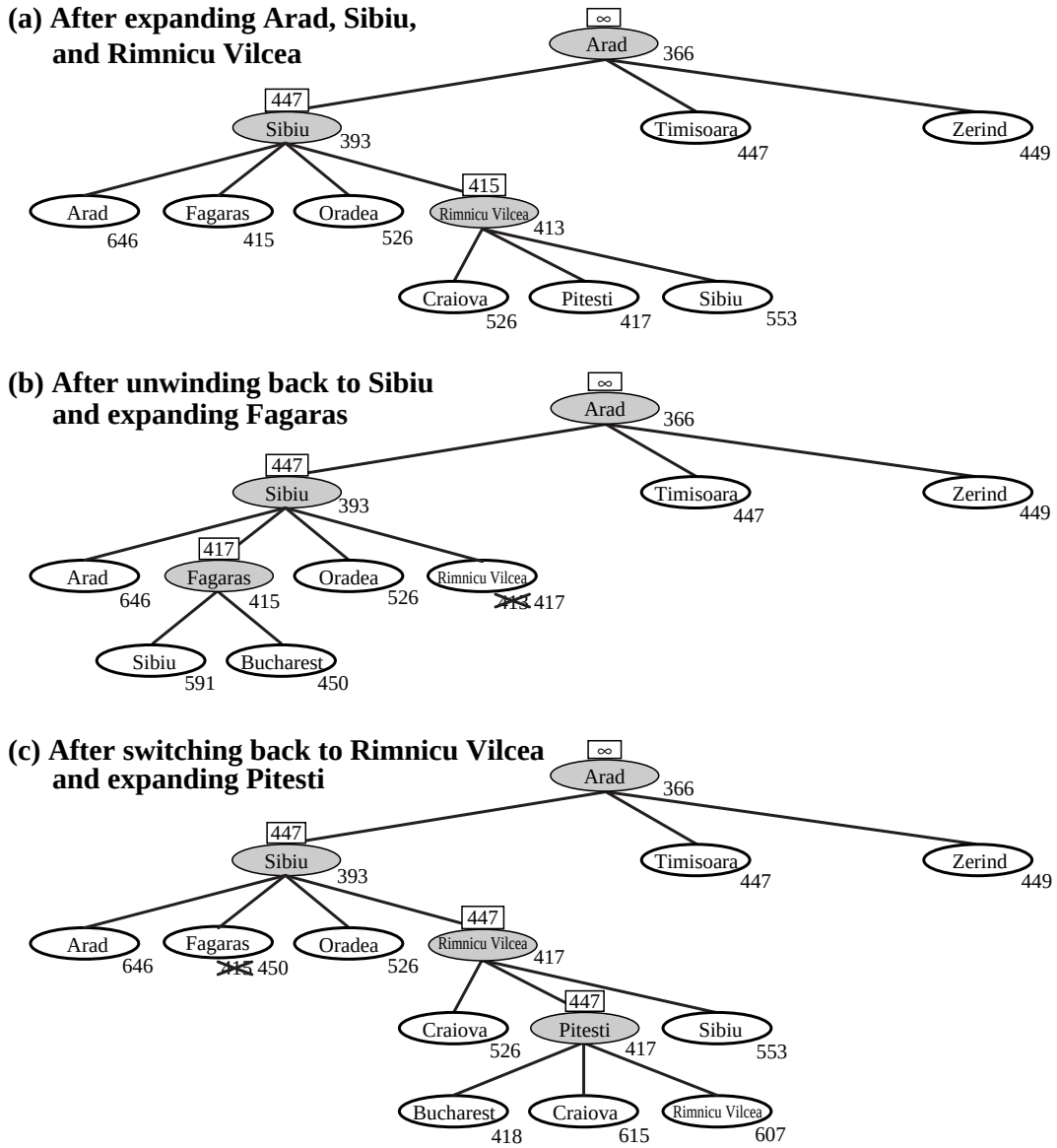
RBFS: recursive Depth-first version of Best-First search using only linear memory. Memorizes f-value of best alternative path.

f-value of each node is replaced by the best (smallest) f-value of its children.

function RECURSIVE-BEST-FIRST-SEARCH($problem$) **returns** a solution, or failure
RBFS($problem$, MAKE-NODE(INITIAL-STATE[$problem$]), ∞)

function RBFS($problem$, $node$, f_limit) **returns** a solution, or failure and a new f-cost limit
if GOAL-TEST[$problem$]($state$) **then return** $node$
$successors \leftarrow$ EXPAND($node$, $problem$)
if $successors$ is empty **then return** $failure$, ∞
for each s **in** $successors$ **do**
f[s] $\leftarrow \max(g(s) + h(s), f[node])$
repeat
$best \leftarrow$ the lowest f-value node in $successors$
if $f[best] > f\_limit$ **then return** $failure$, $f[best]$
$alternative \leftarrow$ the second-lowest f-value among $successors$
$result, f[best] \leftarrow$ RBFS($problem$, $best$, min(f_limit, $alternative$))
if $result \neq failure$ **then return** $result$

Table 2.29: RBFS



▸ Map of Romania

Table 2.30: Illustration of RBFS.

- RBFS is optimal and complete under the same assumptions as A^*.
- IDA* and RBFS suffer from using too little memory.

We would like to use as much memory as possible. This leads to SMA*

SMA*: is an extension of A*, which only needs a limited amount of memory.

If there is no space left but nodes have to be expanded, nodes will be removed from the queue:

- those with possibly great f-value (forgotten nodes). But their f-costs will be stored. Later those nodes will be considered if all other paths lead to higher costs.

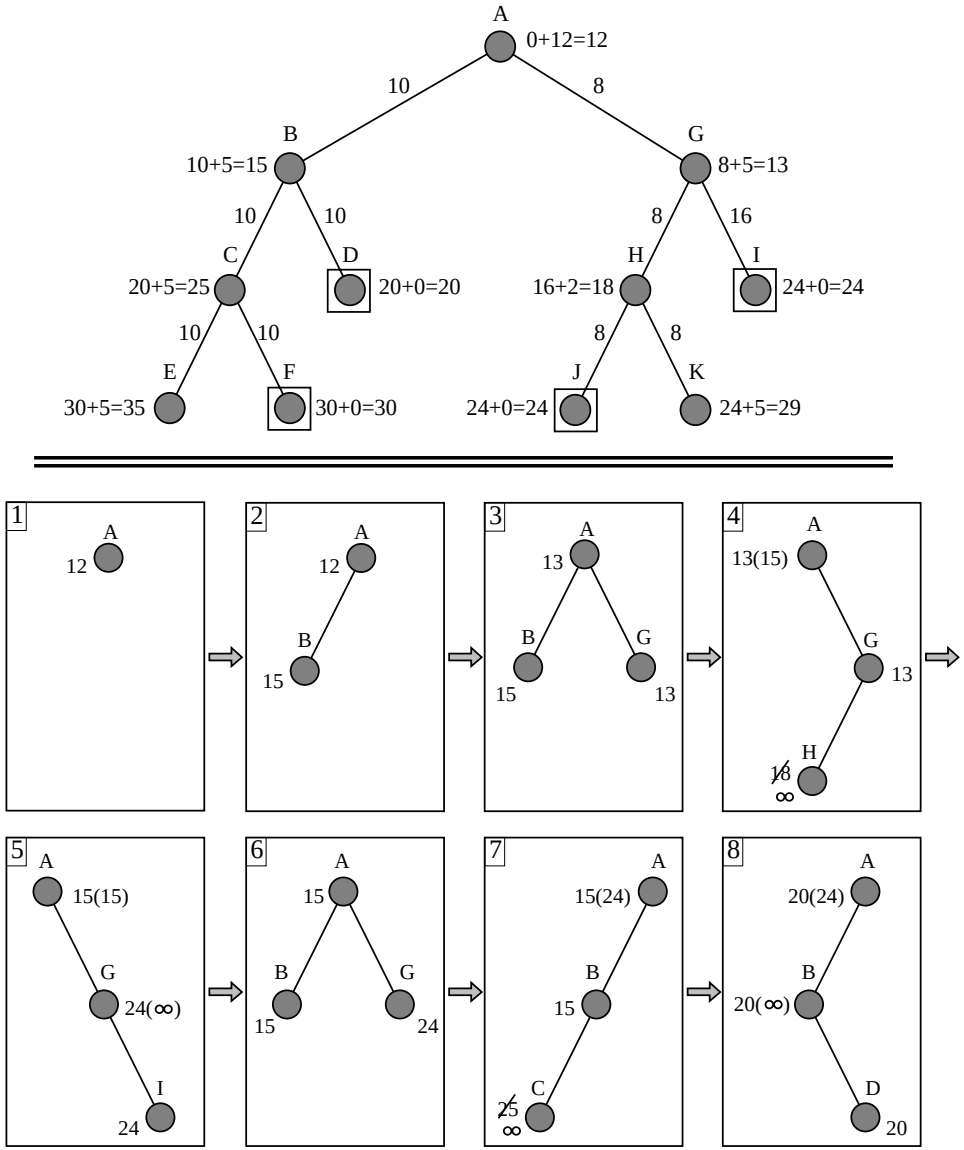


Figure 2.11: Illustration of SMA*.

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
    if Queue is empty then return failure
    n ← deepest least-f-cost node in Queue
    if GOAL-TEST(n) then return success
    s ← NEXT-SUCCESSOR(n)
    if s is not a goal and is at maximum depth then
      f(s) ← ∞
    else
      f(s) ← MAX(f(n), g(s)+h(s))
    if all of n's successors have been generated then
      update n's f-cost and those of its ancestors if necessary
    if SUCCESSORS(n) all in memory then remove n from Queue
    if memory is full then
      delete shallowest, highest-f-cost node in Queue
      remove it from its parent's successor list
      insert its parent on Queue if necessary
    insert s on Queue
  end
```

Table 2.31: SMA*.

Theorem 2.14 (Properties of SMA*)

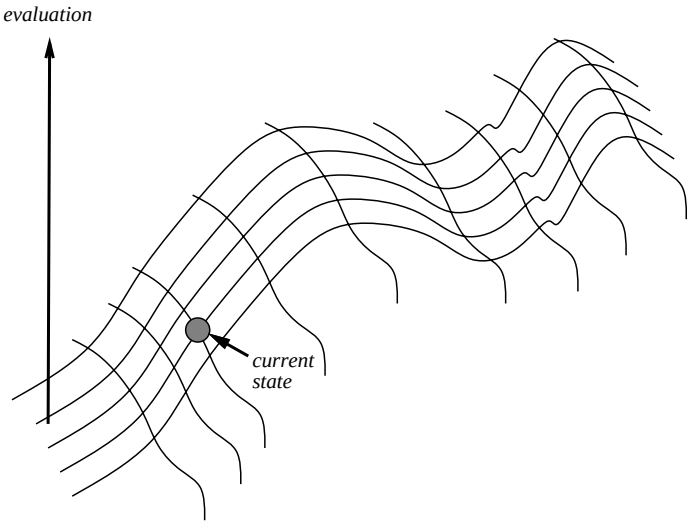
*SMA** *is* complete *if enough memory is available to* store the shortest *solution-path.*

*SMA** *is* optimal *if there is enough memory to* store the optimal *solution-path.*

2.7 Iterative Improvements

Idea:

Considering certain problems only the actual state is important, but not the path leading to it:
Local Search problems



Of course this problem is as difficult as you like!

Hill-climbing: gradient-descent (-ascent). Move in a direction **at random**. Compare the new evaluation with the old one. Move to the new point if the evaluation is better.

Problems:

- **local optima**: (getting lost).
- **plateaux**: (wandering around).
- **ridge**: (detour).

```
function HILL-CLIMBING( problem) returns a solution state
   inputs: problem, a problem
   static: current, a node
           next, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      next ← a highest-valued successor of current
      if VALUE[next] < VALUE[current] then return current
      current ← next
   end
```

Table 2.32: Hill Climbing.

Random Restart Hill Climbing: Start again and again from randomly generated initial states.

- N-queens: heuristic function h is the number of pairs of queens that attack each other. Random restart hill climbing works well even for $N = 10^6$.

Simulated annealing: modified hill-climbing: also bad moves (small evaluation-value) are allowed with the small probability of $e^{-\frac{|\Delta f|}{T}}$. It depends on the "temperature" T ranging from ∞ to 0.

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  static: *current*, a node
    next, a node
    T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if T=0 **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

Table 2.33: Simulated Annealing.

function GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an individual
  inputs: *population*, a set of individuals
    FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    loop for i **from** 1 **to** SIZE(*population*) **do**
      x ← RANDOM-SELECTION(*population*, FITNESS-FN)
      y ← RANDOM-SELECTION(*population*, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) **then** *child* ← 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) **returns** an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, $c+1$, n))

Table 2.34: Genetic Algorithm.

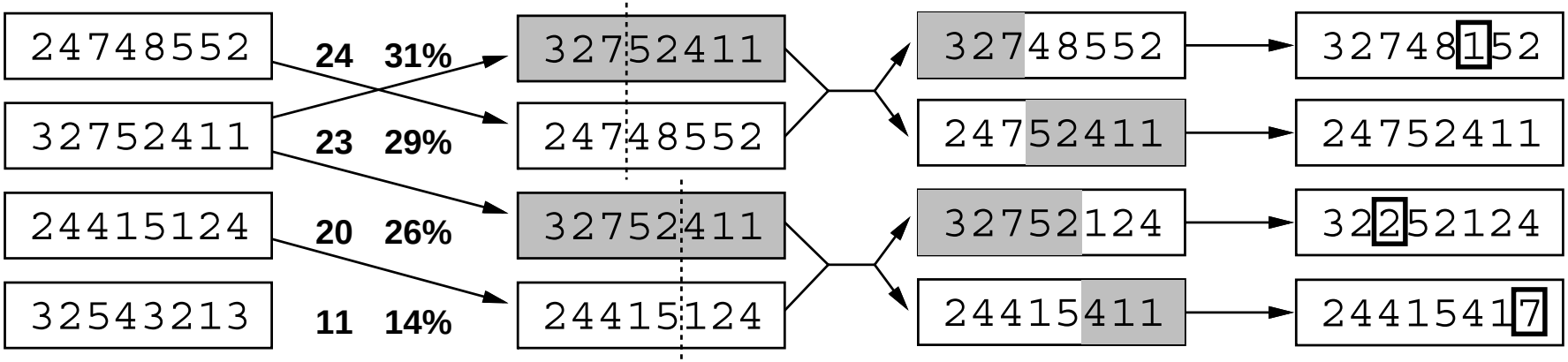


Table 2.35: Illustration of Genetic Algorithm.

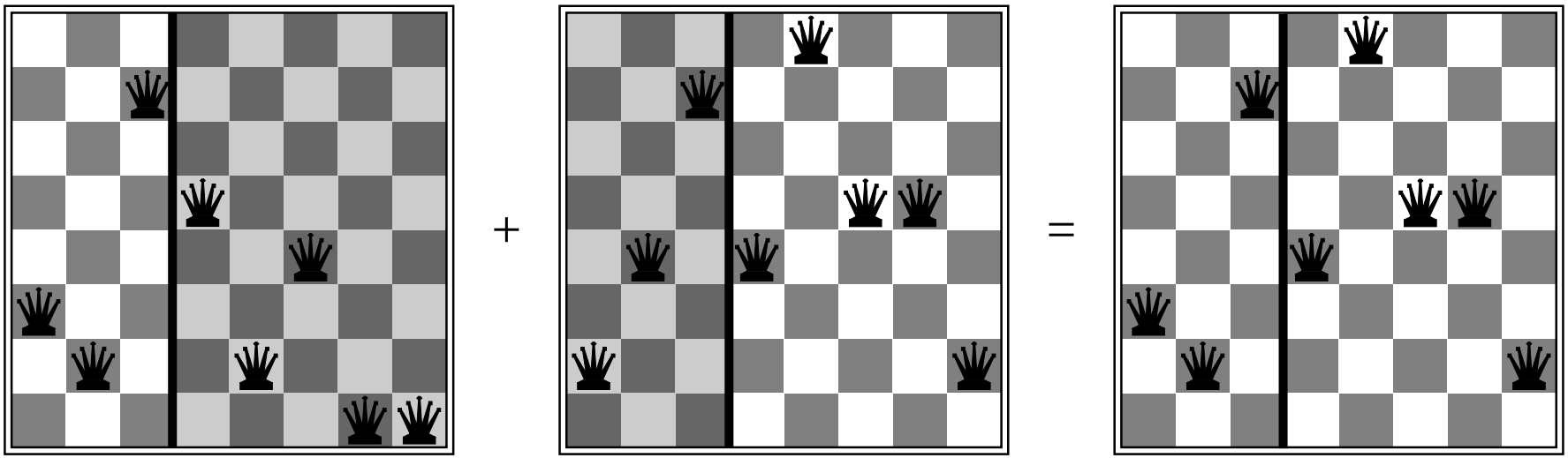


Table 2.36: Crossover in the 8-queens problem.

Fitness function: number of non-attacking pairs of queens.

2.8 Online Search

Up to now: **offline search**. What about interleaving actions and computation?
Remember the **exploration problem**. One does not know the effect of actions, nor the state space. Therefore one has to try out all actions and remember to which states they lead.

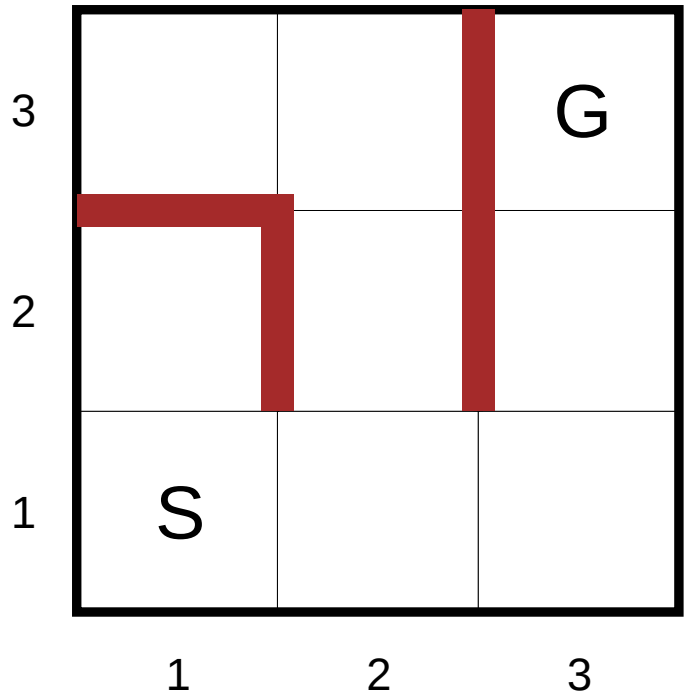


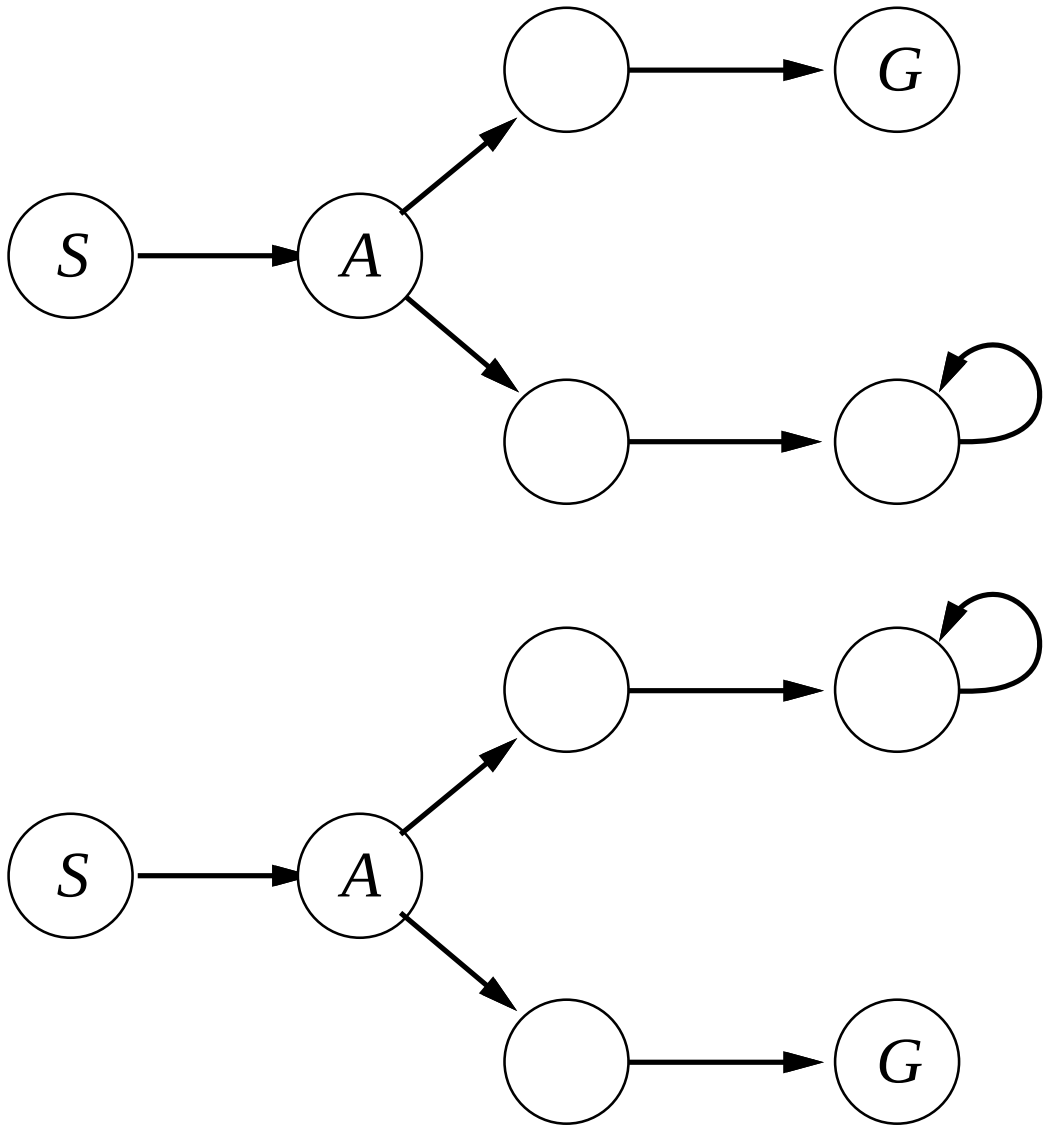
Table 2.37: Online vs Offline search.

How to measure the quality of an online algorithm? Number of steps alone does not make sense. One has to **explore the space and find the optimal path**.

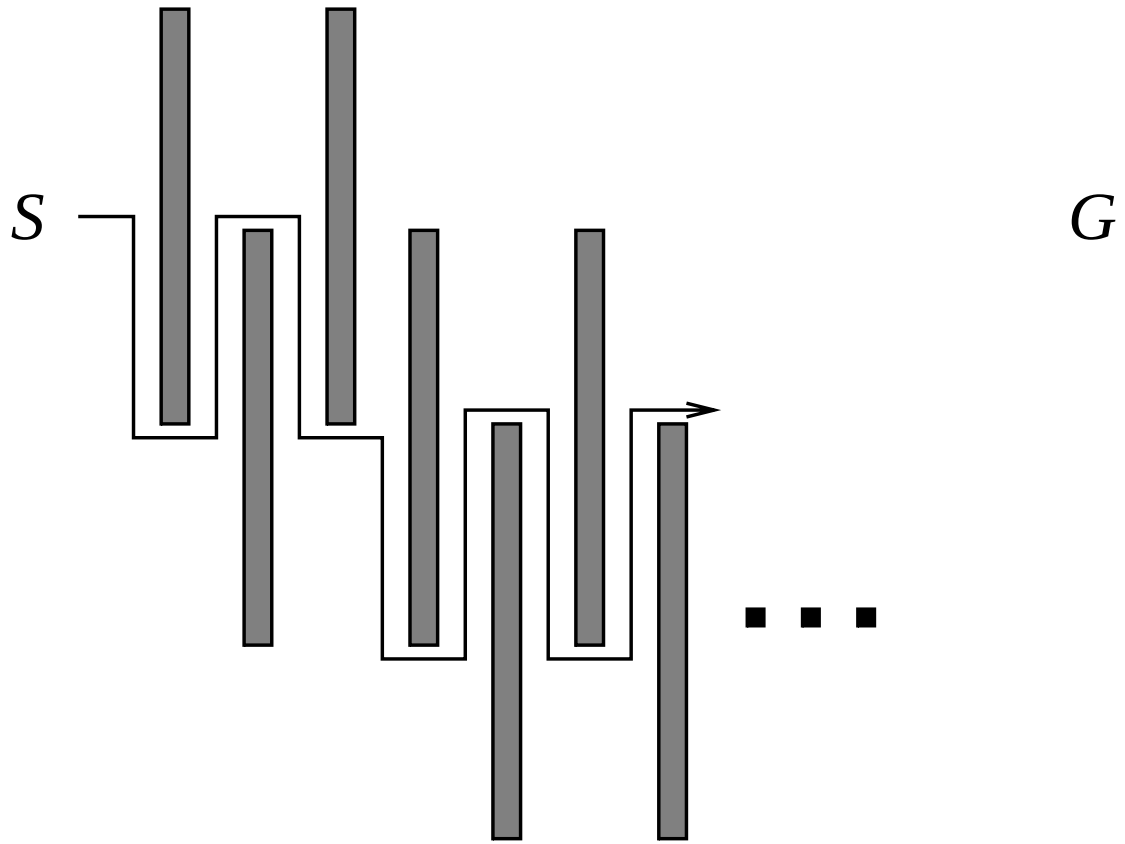
Definition 2.15 (Competitive Ratio)

The competitive ratio of an online search problem is the costs of the path taken by the agent divided by the costs of the optimal path.

Infinite competitive ratio



Unbounded competitive ratio



We assume the following

1. actions are reversible (why?),
2. once a state has been visited, it will be recognized when it is visited again.

- Compare backtracking in offline versus online search. In online search, we have to find an action to backtrack to the previous state. We cannot take it from the queue!!
- Similarly in online search we can only expand a node that we physically occupy.
- We have to keep a table $result(a, s)$ listing the effects of actions a executed in state s.
- We also have to keep the following two tables:
  1. unexplored: for each state the actions not yet tried,
  2. unbacktracked: for each state the backtracks not yet tried.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          unexplored, a table that lists, for each visited state, the actions not yet tried
          unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
          s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state then unexplored[s'] ← ACTIONS(s')
  if s is not null then do
      result[a, s] ← s'
      add s to the front of unbacktracked[s']
  if unexplored[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a ← an action b such that result[b, s'] = POP(unbacktracked[s'])
  else a ← POP(unexplored[s'])
  s ← s'
  return a
```

Table 2.38: Online DFS Agent.

- Hill Climbing is an Online Search algorithm!
- What about using random restarts?
- What about using random walks?
- This certainly works for finite spaces, but not for infinite ones.

Why not simply taking random walks?

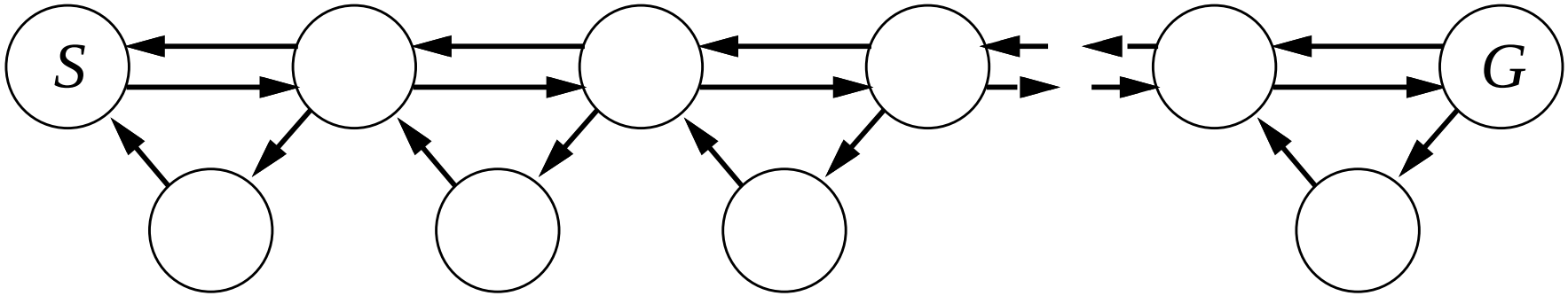


Table 2.39: Random Walk.

Because they can lead to exponentially many steps.

Will a random walk eventually find the goal (completeness)?

$$\begin{cases} Yes, & \text{if state space is finite;} \\ Yes, & \text{for 2-dimensional grids;} \\ \text{with probability } 0.34, & \text{for 3-dimensional grids.} \end{cases}$$

Instead of randomness, let's try to use memory!

- Given a heuristics h, it should be used.
- But it can be misleading (local maxima).
- Therefore we build a better, more realistic estimate H that takes h into account and the path that the hill climbing algorithm takes to explore the states.
- This can help to get out of the local maximum.

Hill-Climbing + Memory=LRTA*

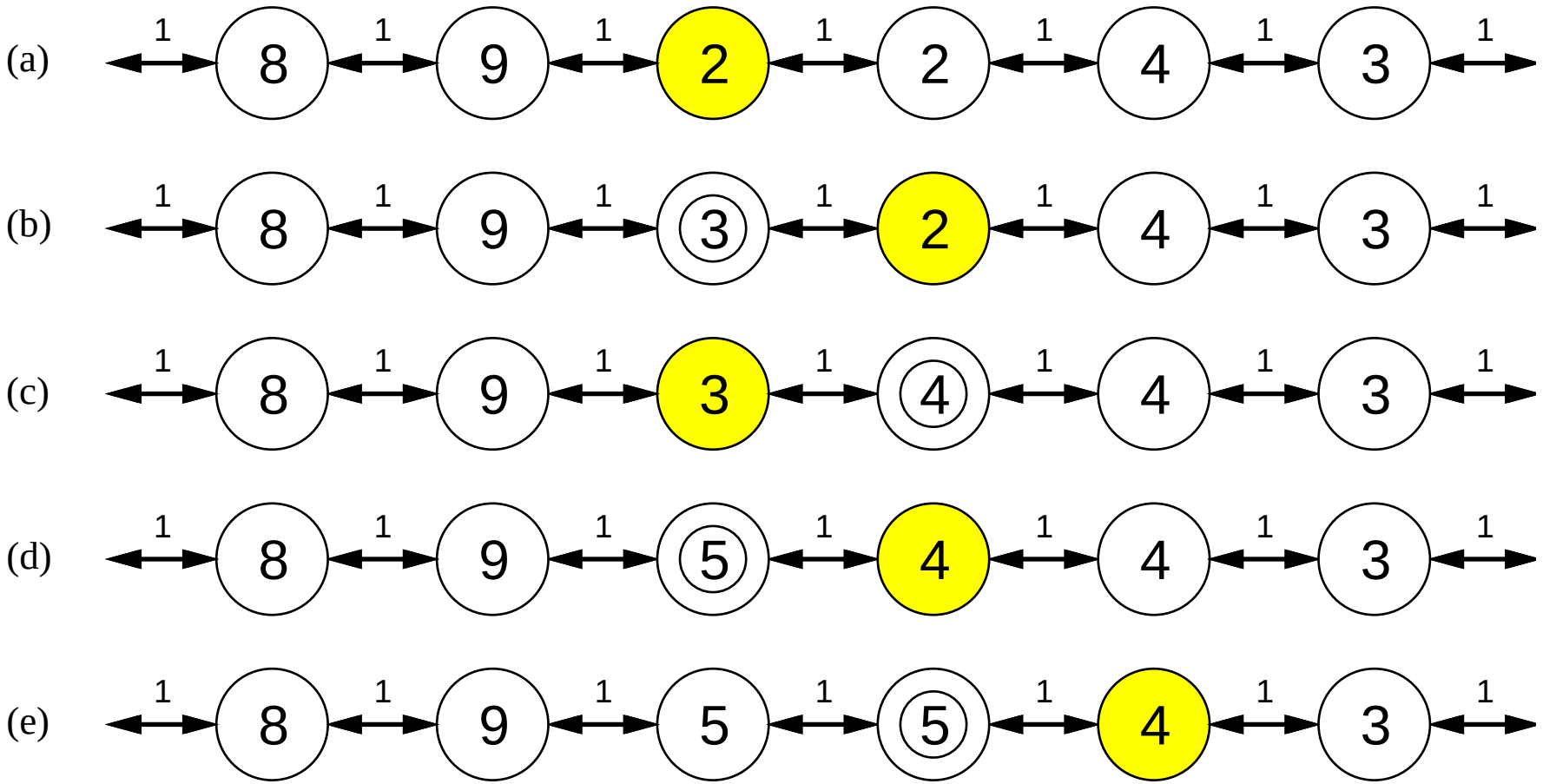


Table 2.40: Illustration of LRTA*

LRTA*

a) The algorithm got stuck in a local minimum. The best neighbour is right from the current (yellow) state.

b) Therefore the expected costs of the previous state have to be updated (3 instead of 2 because the best neighbour has expected costs 2 and the costs to go there is 1). Therefore the algorithm walks back to this node (it is the best one).

c) In the same way the expected costs of the previous node have to be updated (4 instead of 3).

d) Similarly, the expected costs of the current node have to be updated (5 instead of 4).

e) Finally, the best next node is to the right (4 is better than 5).

function LRTA*-AGENT(s') **returns** an action
  inputs: s', a percept that identifies the current state
  static: *result*, a table, indexed by action and state, initially empty
    H, a table of cost estimates indexed by state, initially empty
    s, a, the previous state and action, initially null

  if GOAL-TEST(s') **then return** *stop*
  if s' is a new state (not in H) **then** $H[s'] \leftarrow h(s')$
  unless s is null
    $result[a, s] \leftarrow s'$
    $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)}$ LRTA*-COST($s, b, result[b, s], H$)
  $a \leftarrow$ an action b in ACTIONS(s') that minimizes LRTA*-COST($s', b, result[b, s'], H$)
  $s \leftarrow s'$
  return a

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

Table 2.41: LRTA*

3. Supervised Learning

Content of this chapter (1):

Learning: We describe the general structure of a *Learning Agent*. An agent should be capable of learning new concepts through observing its environment. We distinguish between *supervised-*, *reinforcement-* and *unsupervised* learning.

Decision Trees: We describe a simple algorithm built on some general assumption of *Shannon's information theory*, to construct decision trees given a table of observations. We apply *Ockham's razor* and generate a tree that can be used to make predictions about unseen cases.

Content of this chapter (2):

Ensembles: Here we use not a single hypothesis, but an ensemble of hypotheses to make predictions. We describe the Ada Boost Algorithm, which often improves the hypothesis enormously.

Logic: We discuss another formulation of learning, based on first-order logical formulae: The Version Space Algorithm.

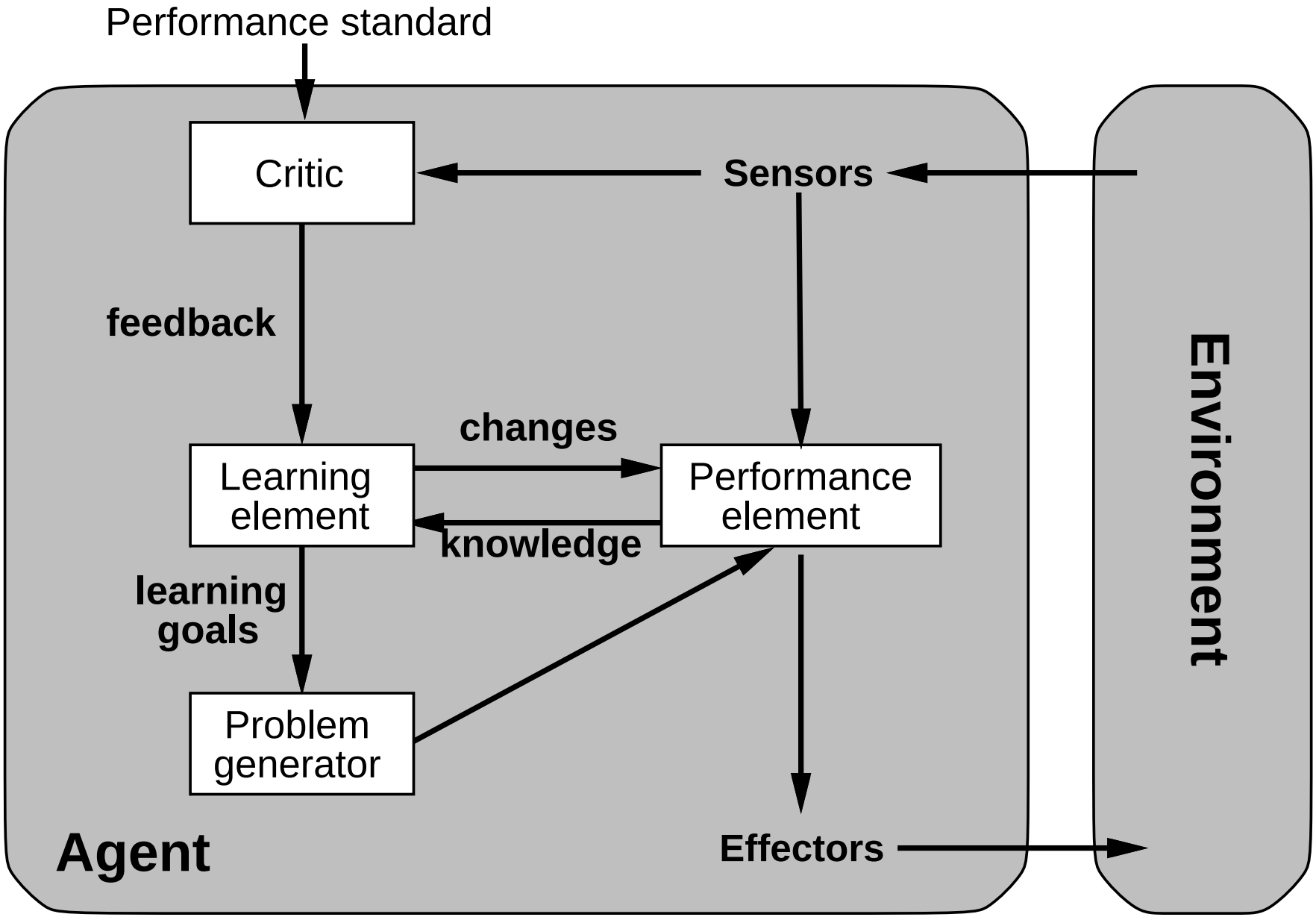
Learning in general

Hitherto: An agent's intelligence is in his program, it is **hard-wired**.

Now: We want a more autonomous agent, which should learn through percepts (experiences) to know its environment.

Important: If the domain in which it acts can't be described completely.

3.1 Basics


Performance standard
Critic
Sensors
feedback
changes
Learning element
Performance element
knowledge
learning goals
Problem generator
Agent
Effectors
Environment

- **Performance element:** Observes and chooses actions. This was the whole agent until now.
- **Critic:** Observes the result of an action and assesses it with respect to an external standard.

Why external?

Otherwise it would set its own standard so low that it always holds!

- **Learning element:** Modifies the performance element by considering the critic (and architecture of the performance element). It also creates new goals (to improve understanding effects of actions).
- **Problem generator:** It proposes the execution of actions to satisfy the goals of the learning element. These do not have to be the "best" actions (wrt. performance element): but they should be **informative** and deliver new knowledge about the world.

Example: Driving a taxi.

Question:

What does learning (the design of the learning element) really depend on?

1. Which components of the performance element should be improved?
2. How are these components represented?

(⇝ Slide 224 (Decision trees), ⇝ Slide 248 (Ensemble Learning), ⇝ Slide 257 (Domains formalised in PL1))

3. Which kind of feedback?
   - **Supervised learning**: The execution of an incorrect action leads to the "right" solution as feedback (e.g. How intensively should the brakes be used?).
     Driving instructor
   - **Reinforcement learning**: Only the result is perceived. Critic tells, if good or bad, but not what would have been right.
   - **Unsupervised learning**: No hints about the right actions.

4. Which a-priori-information is there?
   (Often there is useful background knowledge)

Components of the performance element:

1. Mappings from the conditions of the actual state into the set of actions
2. Deriving relevant properties of the world from the percepts
3. Information about the development of the world
4. Information about the sequence of actions
5. **Assessing-function** of the world-states
6. **Assessing-function** concerning the quality of single actions in one state
7. Description of state-classes, which maximise the utility-function of an agent

Important:

All these components are – from a mathematical point of view – mappings.

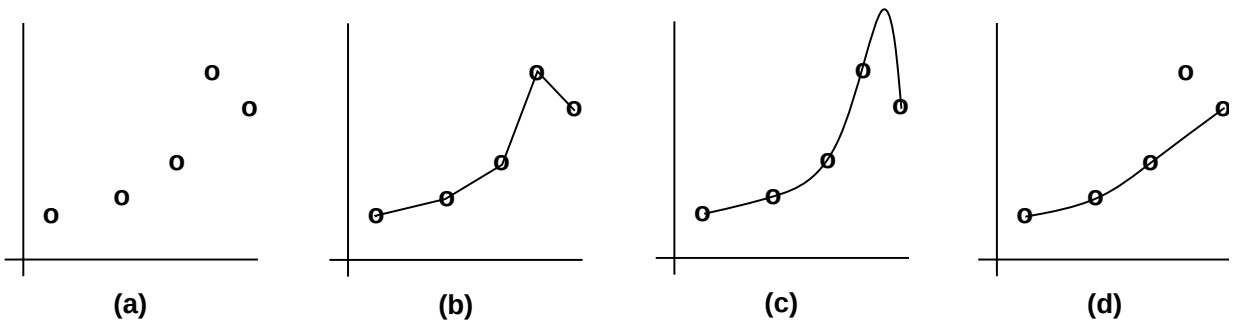
Learning **means** to represent these mappings.

Inductive learning: Given a set of pairs (x, y).
Find f with $f(x) = y$.

Example 3.1 (Continue a series of numbers)

Which number is next?

- 3, 5, 7, ?
- 3, 5, 17, 257, ?



Simple reflex agent:

global *examples* $\leftarrow \{\}$

---

function REFLEX-PERFORMANCE-ELEMENT(*percept*) **returns** an action

if (*percept*, *a*) **in** *examples* **then return** *a*
else
h $\leftarrow$ INDUCE(*examples*)
return *h*(*percept*)

---

procedure REFLEX-LEARNING-ELEMENT(*percept*, *action*)
inputs: *percept*, feedback percept
action, feedback action

examples $\leftarrow$ *examples* $\cup \{($ *percept*, *action*$)\}$

Example 3.2 (Wason's Test; Verify and Falsify)

Consider a set of cards. Each card has a letter printed on one side and a number on the other. Having taken a look at some of these cards you formulate the following hypothesis:

If there is a vowel on one side then there is an even number on the other.

Now there are the following cards on the table:

4 T A 7.

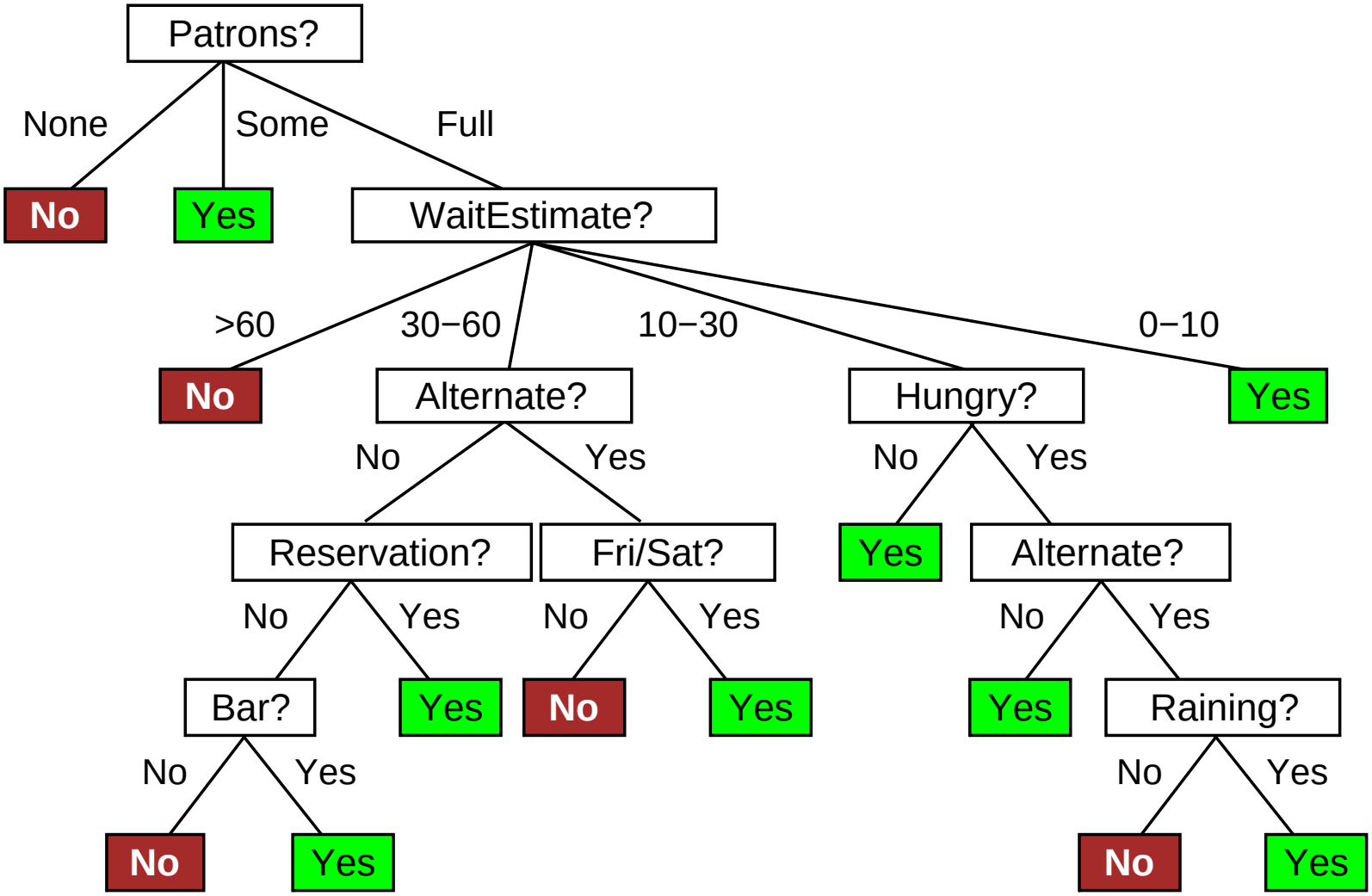
You are allowed to turn around only two cards to check the hypothesis.
Which card(s) do you flip?

3.2 Decision Trees

Decision trees represent boolean functions

Small example:

You plan to go out for dinner and arrive at a restaurant. Should you wait for a free table or should you move on?



Learned Tree

Decision Tree

decision tree = conjunction of implications

(implication = path leading to a leaf)

For all restaurants r:

$$(Patrons(r, Full) \wedge Wait\_estimate(r, 10-30) \wedge \neg Hungry(r)) \longrightarrow Will\_Wait(r)$$

Attention:

This is written in first order logic but a decision tree talks only about a single object (r above). So this is really propositional logic:

$$Patrons_r^{Full} \wedge Wait\_estimate_r^{10-30} \wedge \neg Hungry_r \longrightarrow Will\_Wait_r$$

Question:

boolean functions = decision trees?

Answer:

Yes! Each row of the table describing the function belongs to one path in the tree.

Attention

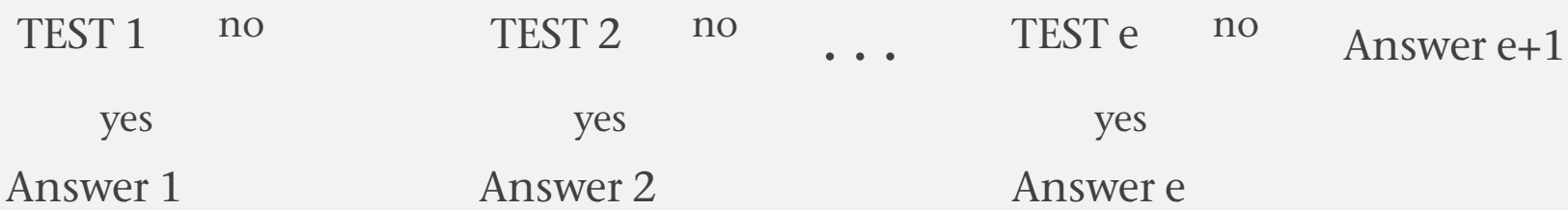
Decision trees can be much smaller! But there are boolean function which can only be represented by trees with an exponential size:

Parity function: $par(x_1, \ldots, x_n) := \begin{cases} 1, & \text{if } \sum_{i=1}^{n} x_i \text{ is even} \\ 0, & \text{else} \end{cases}$

Variant of decision-trees

Example 3.3 (Decision List)

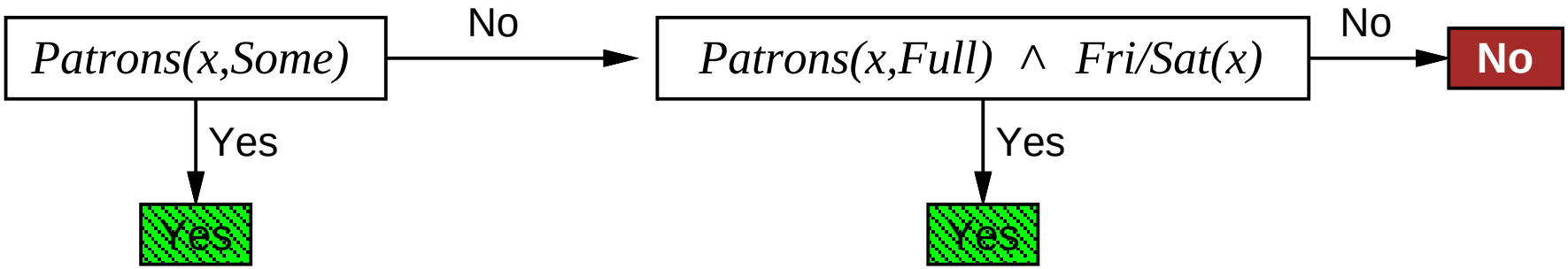
All attributes are boolean. A decision list is a tree of the following form



with $Answer_i \in \{Yes, No\}$ and Test_i a conjunction of (possibly negated) attributes ($\rightsquigarrow$ **Exercise: Compare decision trees and decision lists**).

$k\text{-}DL(n)$ is the *set of boolean functions with n attributes, which can be represented by decision lists with at most k checks in each test.*

PAC-Learning



Obviously:

- n-$DL(n)$ = set of all boolean functions
- $card(n\text{-}DL(n)) = 2^{2^n}$.

Question: Table of examples

How should decision trees be learned?

Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

▸ Decision Tree ▸ Learned Tree

The set of examples-to-be-learned is called training set. Examples can be evaluated positively (attribute holds) or negatively (attribute does not hold).

Trivial solution of learning

The paths in the tree are exactly the examples.

Disadvantage:

New cases can not be considered.

Idea:

Choose the simplest tree (or rather the most general) which is compatible with all examples.

Ockham's razor: Entia non sunt multiplicanda praeter necessitatem.

Example 3.4 (Guess!)

A computer program encodes triples of numbers with respect to a certain rule. **Find out that rule.**

You enter triples (x_1, x_2, x_3) of your choice ($x_i \in \mathbb{N}$) and get as answers "yes" or "no".

Simplification: At the beginning the program tells you that these triples are in the set:

$$(4, 6, 8),\ (6, 8, 12),\ (20, 22, 40)$$

Your task:

Make more enquiries (approx. 10) and try to find out the rule.

The idea behind the learning algorithm

Goal: A tree which is as small as possible. First test the most important attributes (in order to get a quick classification).

This will be formalised later, using information theory.

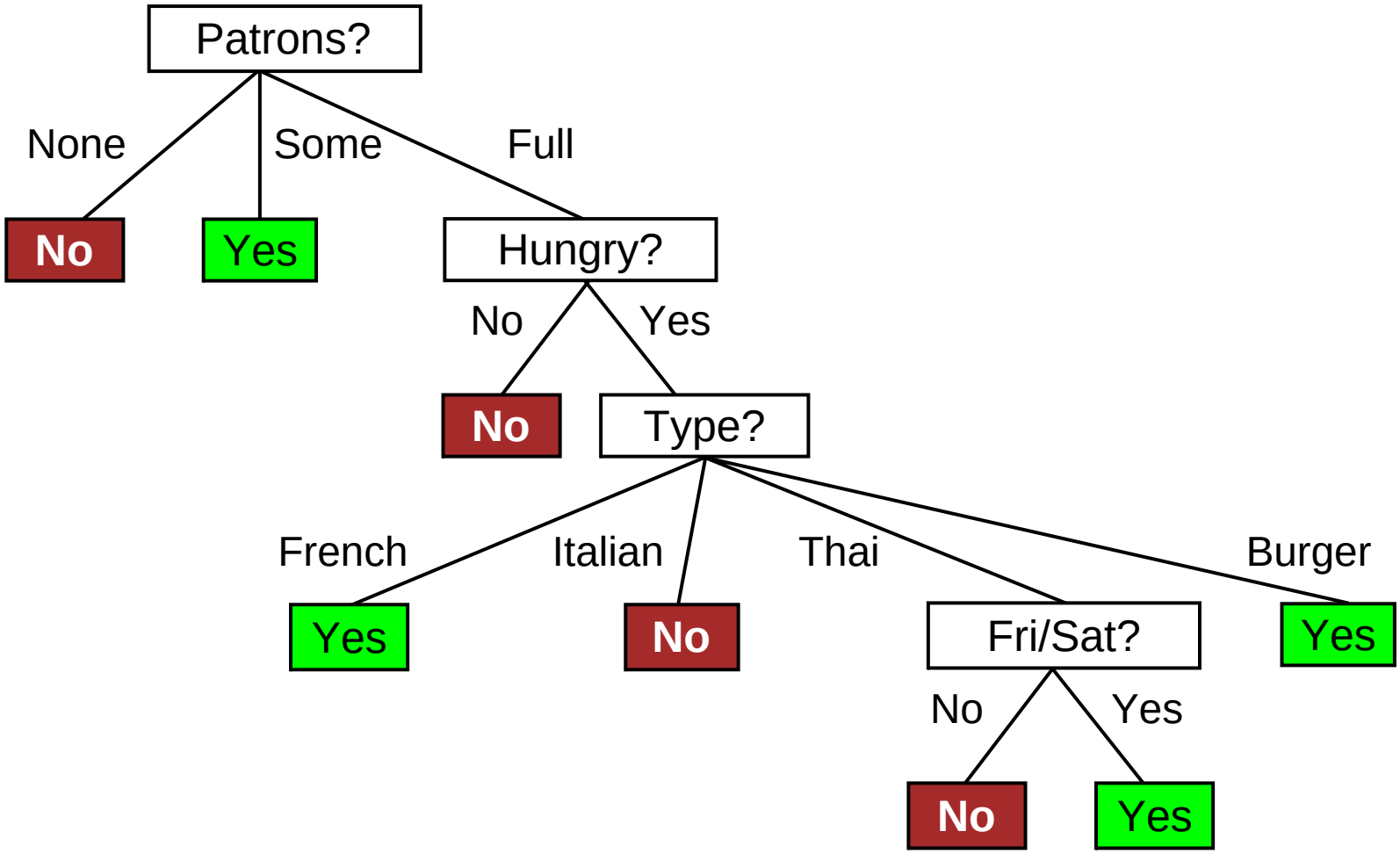
Then proceed recursively, i.e. with decreasing amounts of examples and attributes.

We distinguish the following cases:

1. **There are positive and negative examples.** Choose the best attribute.
2. **There are only positive or only negative examples.** Done – a solution has been found.
3. **There are no more examples.** Then a default value has to be chosen, e.g. the majority of examples of the parent node.
4. **There are positive and negative examples, but no more attributes.** Then the basic set of attributes does not suffice, a decision can not be made. Not enough information is given.

function DECISION-TREE-LEARNING(*examples, attributes, default*) **returns** a decision tree
  inputs: *examples*, set of examples
    attributes, set of attributes
    default, default value for the goal predicate

  if *examples* is empty **then return** *default*
  else if all *examples* have the same classification **then return** the classification
  else if *attributes* is empty **then return** MAJORITY-VALUE(*examples*)
  else
    best $\leftarrow$ CHOOSE-ATTRIBUTE(*attributes, examples*)
    tree $\leftarrow$ a new decision tree with root test *best*
    for each value v_i of *best* **do**
      $examples_i \leftarrow \{$elements of *examples* with *best* $= v_i\}$
      subtree $\leftarrow$ DECISION-TREE-LEARNING($examples_i$, *attributes* $-$ *best*,
                MAJORITY-VALUE(*examples*))
      add a branch to *tree* with label v_i and subtree *subtree*
    end
    return *tree*



Decision Tree | Trainings set

Learned Tree

The algorithm computes a tree which is as small as possible and consistent with the given examples.

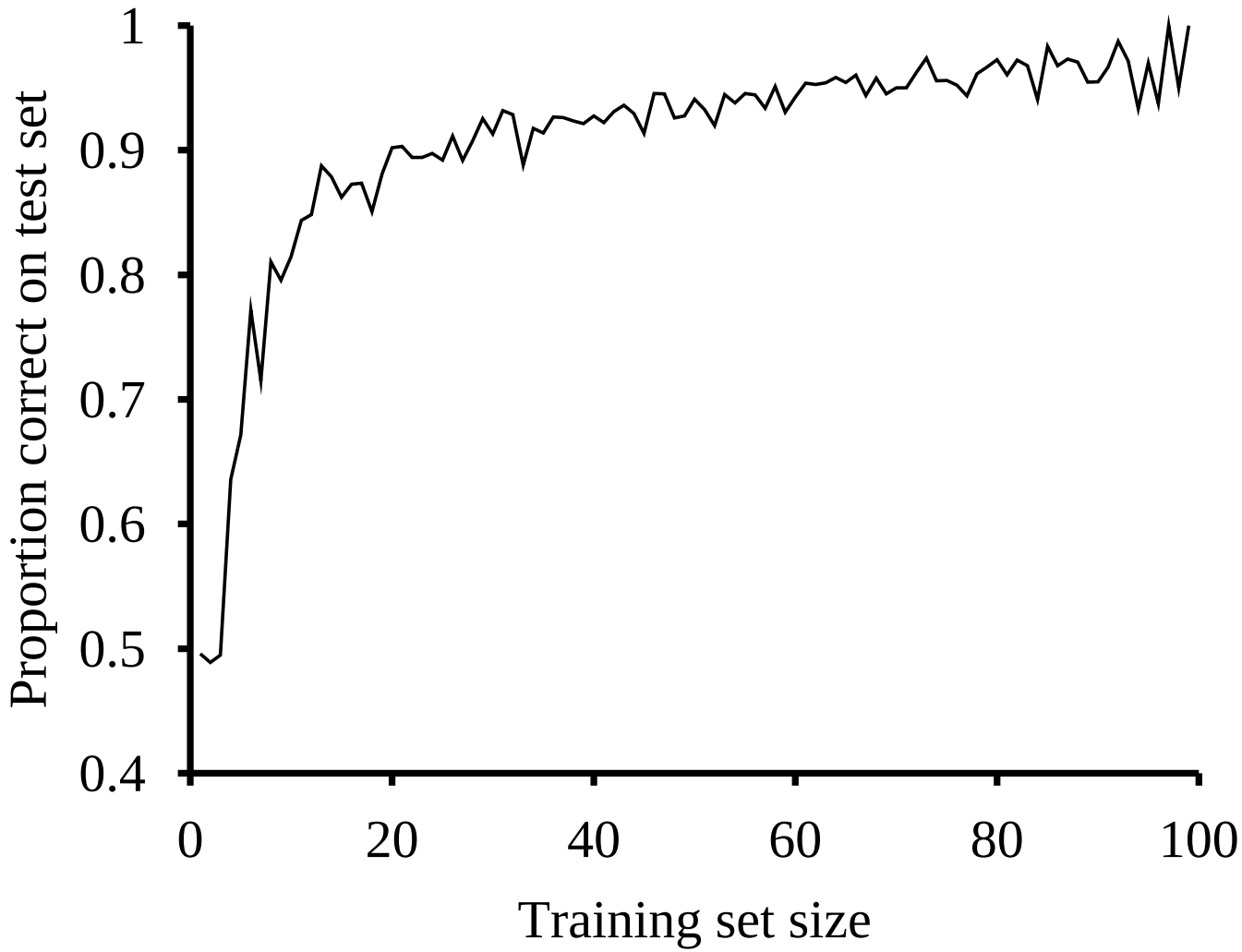
Question:

How good is the generated tree? How different is it from the "actual" tree? Is there an a-priory-estimation? ($\rightsquigarrow$ PAC learning).

Empiric approach:

1. Chose a set of examples M_{Ex}.
2. Divide into two sets: $M_{Ex} = M_{Trai} \cup M_{Test}$.
3. Apply the learning algorithm on M_{Trai} and get a hypothesis H.
4. Calculate the amount of correctly classified elements of M_{Test}.
5. Repeat 1.-4. for many $M_{Trai} \cup M_{Test}$ with randomly generated M_{Trai}.

Attention: Peeking!


Proportion correct on test set
1
0.9
0.8
0.7
0.6
0.5
0.4
0
20
40
60
80
100
Training set size

Information theory

Question:

How to choose the best attribute? The best attribute is the one that delivers the highest amount of information.

Example: Flipping a coin

Shannon's theory

Definition 3.5 (1 bit, information)

1 bit is the information contained in the outcome of flipping a (fair) coin.

More generally: assume there is an experiment with n possible outcomes $v_1, \ldots, v_n$. Each outcome v_i will result with a probability of $P(v_i)$. The information encoded in this result (the outcome of the experiment) is defined as follows:

$$\mathbf{I}(P(v_1), \ldots, P(v_n)) := \sum_{i=1}^{n} -P(v_i) log_2 P(v_i)$$

Assume the coin is manipulated. With a probability of 90% head will come out. Then

$$\mathbf{I}(0.1, 0.9) = \ldots \approx 0.47$$

Question:

For each attribute A: If this attribute is evaluated with respect to the actual training-set, how much information will be gained this way?

The "best" attribute is the one with the highest gain of information!

Definition 3.6 (Gain of Information)

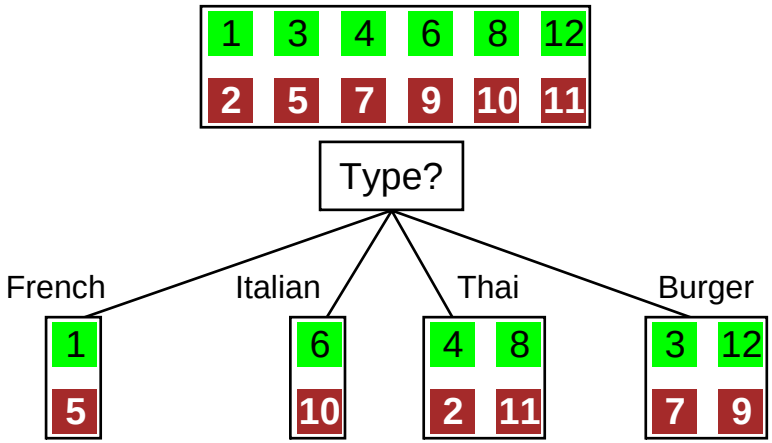
We gain the following information by testing the attribute A:

$$\text{Gain}(A) = \mathbf{I}(\frac{p}{p+n}, \frac{n}{p+n}) - \text{Missing\_Inf}(A)$$

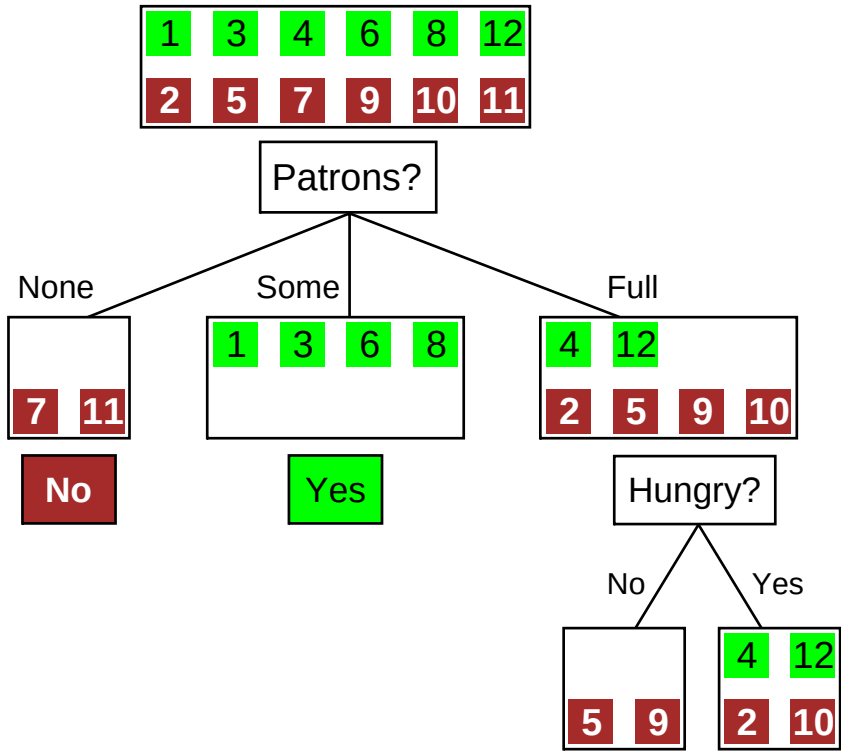
with

$$\text{Missing\_Inf}(A) = \sum_{i=1}^{\nu} \frac{p_i + n_i}{p+n} \mathbf{I}(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$$

Choose_Attribute chooses the A with maximal $Gain(A)$.



(a)



(b)

The figure implies $Gain(Patron) \approx 0.54$. Calculate $Gain(Type)$, $Gain(Hungry)$ ($Hungry$ as the first attribute), $Gain(Hungry)$ (with predecessor $Patron$).

3.3 Ensemble Learning

So far:

A single hypothesis is used to make predictions.

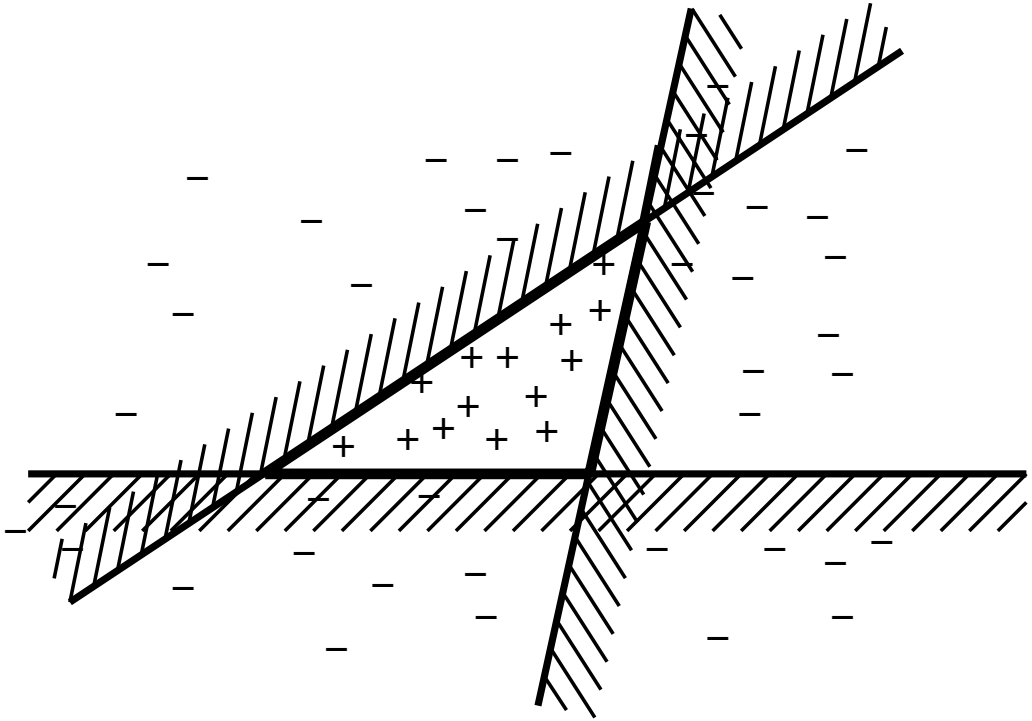
Idea:

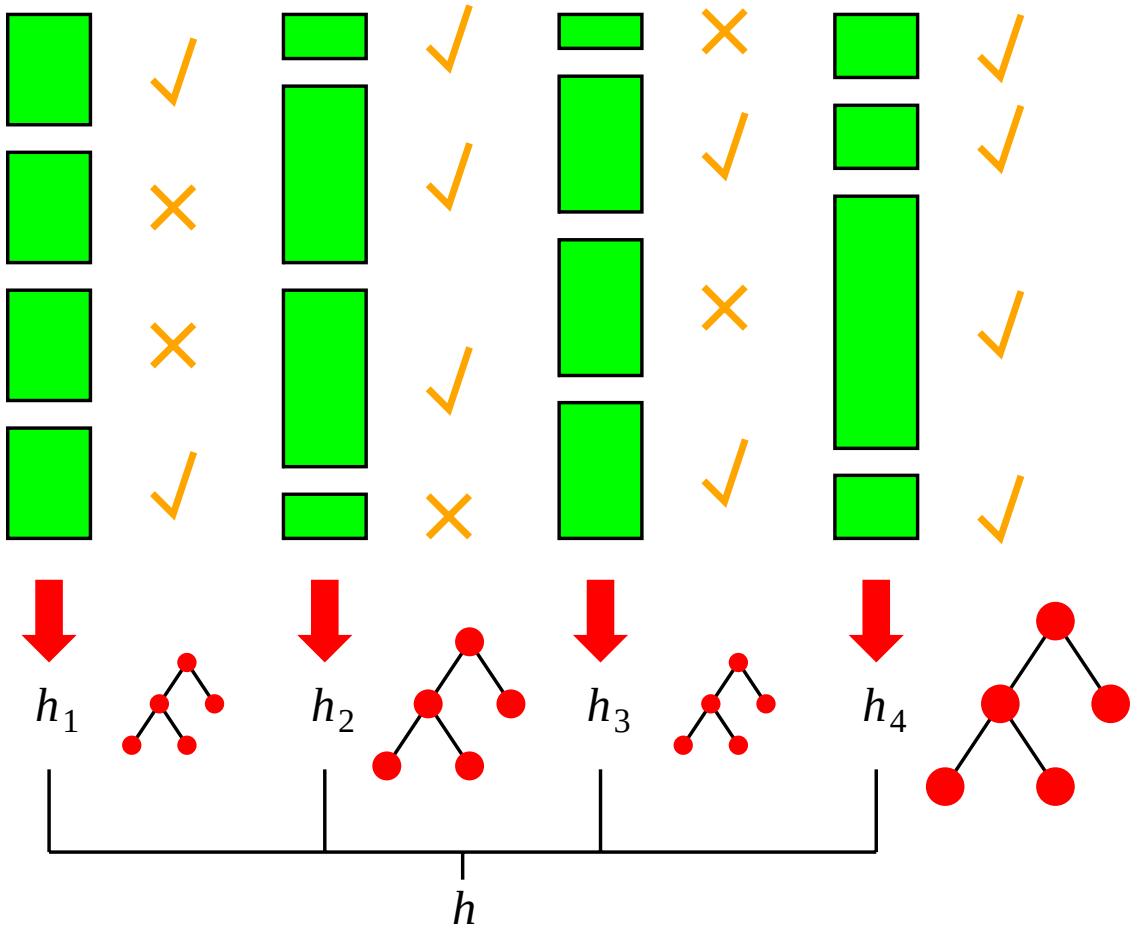
Let's take a whole bunch of them (an ensemble).

Motivation:

Among several hypotheses, use majority voting.
The misclassified ones get higher weights!

Consider three simple hypotheses. No one is perfect. But all taken together, a **new hypothesis** is created (which is not constructible by the original method).




h_1
h_2
h_3
h_4
h

Weighted Training Set: Each example gets a weight $w_j \geq 0$.

Initialisation: All weights are set to $\frac{1}{n}$.

Boosting: Misclassified examples are getting higher weights.

Iterate: We get new hypotheses h_i. After we got a certain number M of them we feed them into the

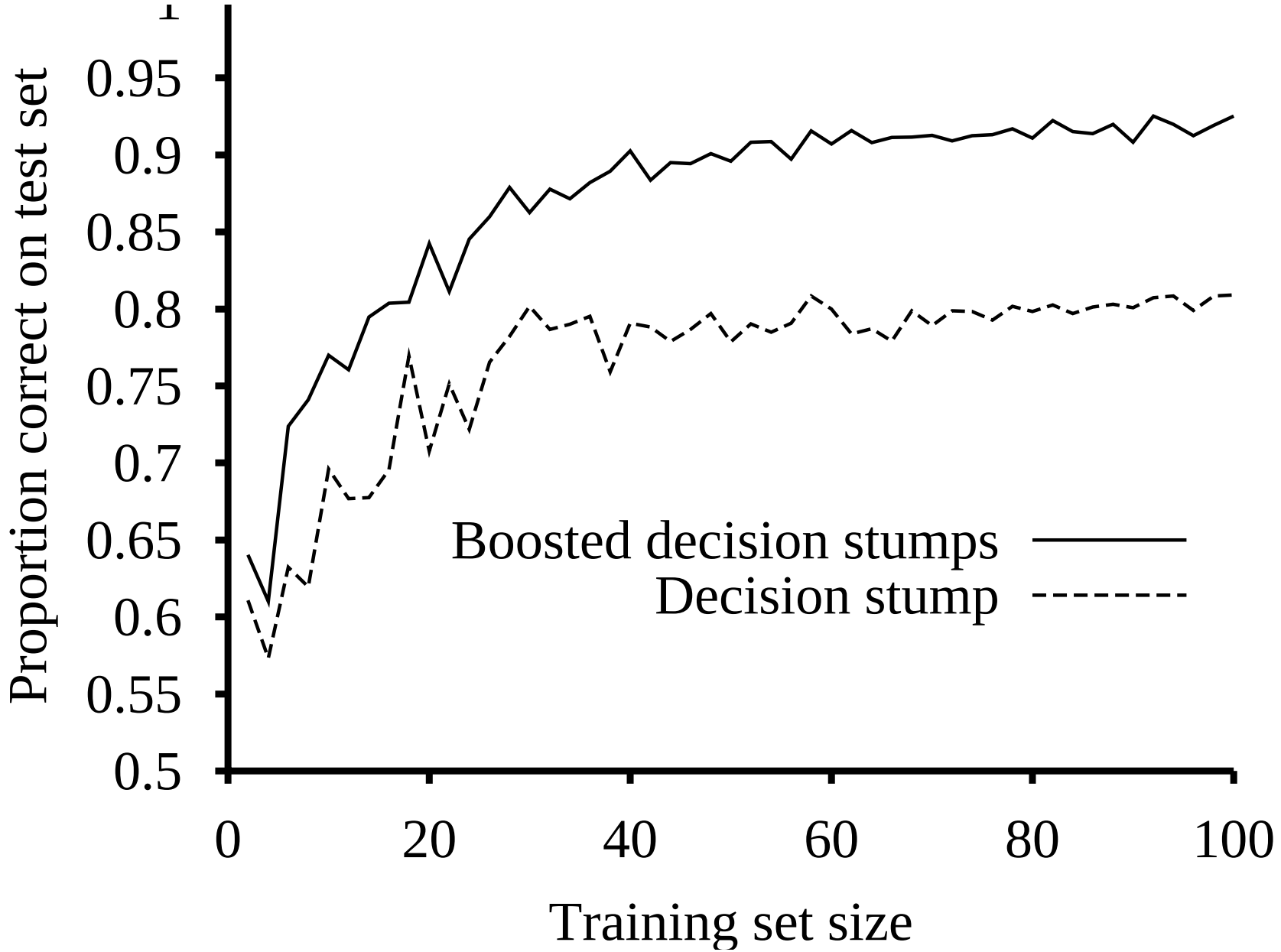
Boosting-Algorithm: It creates a weighted ensemble hypothesis.

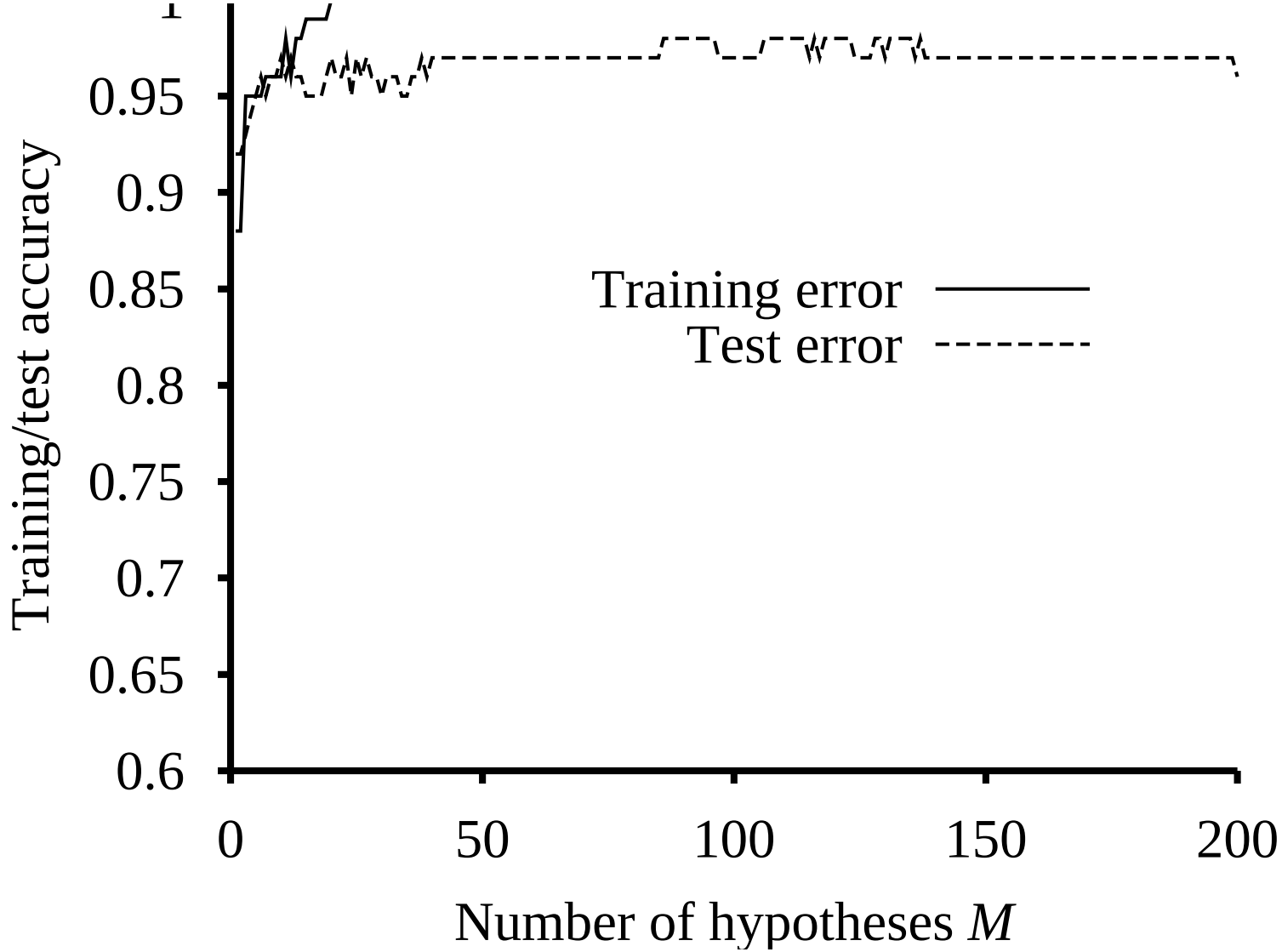
function ADABOOST($examples, L, M$) **returns** a weighted-majority hypothesis
  inputs: $examples$, set of N labelled examples $(x_1, y_1), \ldots, (x_N, y_N)$
    L, a learning algorithm
    M, the number of hypotheses in the ensemble
  local variables: $\mathbf{w}$, a vector of N example weights, initially $1/N$
    $\mathbf{h}$, a vector of M hypotheses
    $\mathbf{z}$, a vector of M hypothesis weights

  for $m = 1$ **to** M **do**
    $\mathbf{h}[m] \leftarrow L(examples, \mathbf{w})$
    $error \leftarrow 0$
    for $j = 1$ **to** N **do**
      if $\mathbf{h}[m](x_j) \neq y_j$ **then** $error \leftarrow error + \mathbf{w}[j]$
    for $j = 1$ **to** N **do**
      if $\mathbf{h}[m](x_j) = y_j$ **then** $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot error/(1 - error)$
    $\mathbf{w} \leftarrow$ NORMALIZE($\mathbf{w}$)
    $\mathbf{z}[m] \leftarrow \log(1 - error)/error$
  return WEIGHTED-MAJORITY($\mathbf{h}, \mathbf{z}$)

Theorem 3.7 (Effect of boosting)

Suppose the Learning algorithm has the following **property**: *it always* **returns a hypothesis with weighted error that is slightly better than random guessing**.
Then AdaBOOST will return a hypothesis classifying *the training data* perfectly for large enough M.


Proportion correct on test set
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0
20
40
60
80
100
Training set size
Boosted decision stumps
Decision stump


Training/test accuracy
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0
50
100
150
200
Number of hypotheses M
Training error
Test error

3.4 PL1 Formalisations

Goal:

A more general framework.

Idea:

To learn means to search in the hypotheses space ($\rightsquigarrow$ planning).

Goal-predicate:

$Q(x)$, one-dimensional (hitherto: $Will\_Wait$)

We seek a definition of $Q(x)$, i.e. a formula $C(x)$ with

$$\forall x \ (Q(x) \leftrightarrow C(x))$$

Each example X_i represents a set of conditions under which $Q(X_i)$ holds or not. We look for an **explanation**: a **formula** $C(x)$ which uses all predicates of the examples.

$$\begin{aligned}\forall r \quad & Will\_Wait(r) \leftrightarrow \\ & Patrons(r, Some) \\ & \vee(Patrons(r, Full) \wedge \neg Hungry(r) \wedge Type(r, French)) \\ & \vee(Patrons(r, Full) \wedge \neg Hungry(r) \wedge Type(r, Thai) \wedge Fri\_Sat(r)) \\ & \vee(Patrons(r, Full) \wedge \neg Hungry(r) \wedge Type(r, Burger))\end{aligned}$$

Note that we require the formulae $C(x)$ to be of a certain form: **disjunctions of conjunctions** of atomic or negated predicates. The negations of such formulae are also called **clauses**. They will be defined more precisely on Slide 401 in Chapter 5.

Definition 3.8 (Hypothesis, Candidate Function)

A formula $C_i(x)$ with $\forall x \ (Q(x) \leftrightarrow C_i(x))$ is called candidate function. The whole formula is called hypothesis:

$$H_i : \quad \forall x \ (Q(x) \leftrightarrow C_i(x))$$

Definition 3.9 (Hypotheses-Space $\mathcal{H}$)

The hypotheses-space $\mathcal{H}$ of a learning-algorithm is the set of hypotheses the algorithm can create.

The extension of a hypothesis H with respect to the goal-predicate Q is the set of examples for which H holds.

Attention:

The combination of hypotheses with different extensions leads to inconsistency.

Hypotheses with the same extensions are logically equivalent.

The situation in general:

- We have a set of examples $\{X_1, \ldots, X_n\}$. We describe each example X through a clause and the declaration $Q(X)$ or $\neg Q(X)$.

Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

X_1 is defined through

$$\ldots lt(X_1) \wedge \neg Bar(X_1) \wedge \neg Fri\_Sat(X_1) \wedge Hungry(X_1) \wedge \ldots$$

und $Will\_Wait(X_1)$.

Note that $\mathcal{H}$ is the set of hypotheses as defined in Definition 3.8. While it corresponds to decision trees, it is not the same.

- The **training set** is the **set of all such conjunctions**.

- We search for a hypothesis which is consistent with the training set.

Question:

Under which conditions is a hypothesis H **inconsistent** with an example X?

false negative: Hypothesis says no ($\neg Q(X)$) but $Q(X)$ does hold.

false positive: Hypothesis says yes ($Q(X)$) but $\neg Q(X)$ does hold.

Attention:

Inductive learning in logic-based domains means to restrict the set of possible hypotheses with every example.

For first order logic: $\mathcal{H}$ is in general infinite, thus automatic theorem proving is much too general.

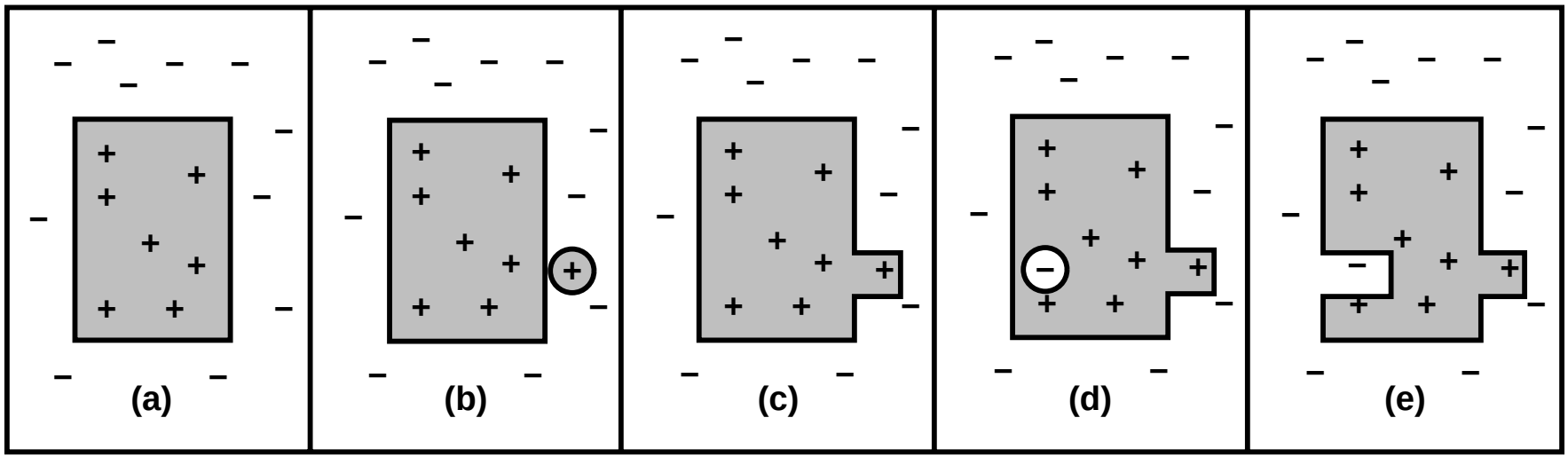
1st approach: We **keep one hypothesis** and **modify it** if the examples are inconsistent with it.

2nd approach: We keep the whole subspace that is still consistent with the examples (version space).
This is effectively represented by two sets (analogical to the representation of a range of real numbers by $[a, b]$).

1st approach: Current-best-hypothesis Search.

Begin with a simple hypothesis H.

- If a new example is consistent with H: okay.
- If it is false negative: **Generalise** H.
- If it is false positive: **Specialise** H.



This leads to an algorithm:

function CURRENT-BEST-LEARNING(*examples*) **returns** a hypothesis

$H \leftarrow$ any hypothesis consistent with the first example in *examples*
for each remaining example in *examples* **do**
  if *e* is false positive for *H* **then**
    $H \leftarrow$ **choose** a specialization of *H* consistent with *examples*
  else if *e* is false negative for *H* **then**
    $H \leftarrow$ **choose** a generalization of *H* consistent with *examples*
  if no consistent specialization/generalization can be found **then fail**
end
return *H*

Question:

How to generalize/specialize?

$$H_1: \quad \forall x\ (Q(x) \leftrightarrow C_1(x))$$
$$H_2: \quad \forall x\ (Q(x) \leftrightarrow C_2(x))$$

- H_1 generalises H_2, if $\forall x\ (C_2(x) \rightarrow C_1(x))$,
- H_1 specialises H_2, if $\forall x\ (C_1(x) \rightarrow C_2(x))$.
- Generalisation means: **leave out $\wedge$-elements in a conjunction, add $\vee$-elements to a disjunction.**
- Specialisation means: **add $\wedge$-elements to a conjunction, leave out $\vee$-elements in a disjunction.**

2nd approach: Version-space.

function VERSION-SPACE-LEARNING(*examples*) **returns** a version space
  local variables: V, the version space: the set of all hypotheses

  $V \leftarrow$ the set of all hypotheses
  for each example e in *examples* **do**
    if V is not empty **then** $V \leftarrow$ VERSION-SPACE-UPDATE(V, e)
  end
  return V

---

function VERSION-SPACE-UPDATE(V, e) **returns** an updated version space

  $V \leftarrow \{h \in V : h \text{ is consistent with } e\}$

Problem:

$\mathcal{H}$ is a big disjunction $H_1 \vee \ldots \vee H_n$. How to represent this?

Reminder:

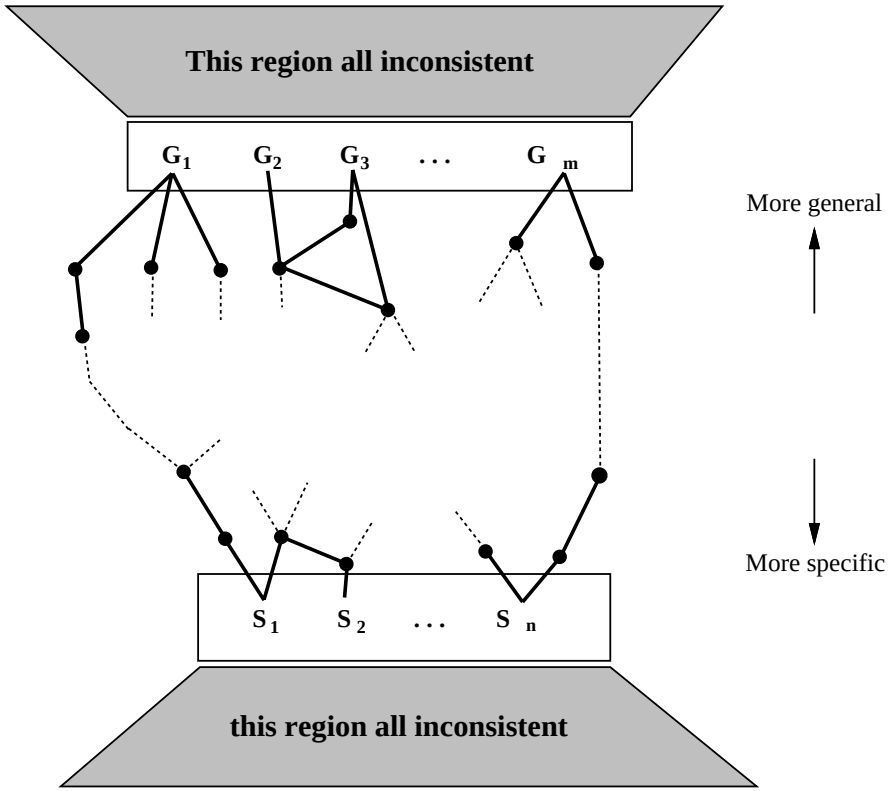
How is the set of real numbers between 0 and 1 represented? Through the range $[0, 1]$.

To solve our problem:

There is a partial order on $\mathcal{H}$ (generalize/specialize). The borders are defined through

- G set: is consistent with all previous examples and there is no more general hypothesis.
- S set: is consistent with all previous examples and there is no more special hypothesis.

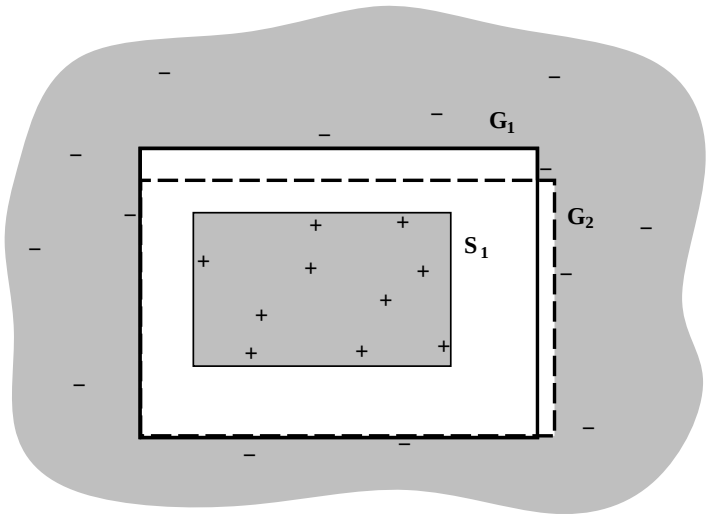
Initially, the G set consists of $\top$ (true), and the S set consists of $\bot$ (false). This is because the initial version space should represent all possible hypotheses.



We have to ensure the following:

1. Each consistent hypothesis (except those in G or S) is more specific than some member of G and more general than some member of S.
2. Each hypothesis that is more specific than some member of G and more general than some member of S is a consistent hypothesis.

When considering **new examples** the G- and S-sets must be appropriately **modified** in VERSION-SPACE-UPDATE (so that the two conditions on the last slide are satisfied).



False positive for S_i: S_i too general, no consistent specializations: **Remove S_i from S.**

False negative for S_i: S_i too specific: **Replace it by all generalizations that are more specific than some element of G.**

False positive for G_i: G_i too general: **Replace it by all specializations that are more general than some element of S.**

False negative for G_i: G_i too specific, no consistent generalizations: **Remove G_i from G.**

Question:

What can happen?

We iterate the algorithm on the previous slide until one of the following happens:

1. Only one hypothesis remains: That is our solution!
2. The space collapses: There is no consistent hypothesis, $G = \emptyset$ or $S = \emptyset$.
3. No examples are left, but there are still several hypotheses: Result is a big disjunction.

3.5 PAC Learning

Question:

What is the distance between the hypothesis H calculated by the learning algorithm and the real function f?

$\rightsquigarrow$ computational learning theory: **PAC**-learning – **P**robably **A**pproximately **C**orrect.

Idea:

If a **hypothesis is consistent with a big training set** then it cannot be completely wrong.

Question:

How are the training set and test set related?

We assume:

The elements of the training and test set are taken from the set of all examples with the same probability.

Definition 3.10 (error(h))

Let $\mathbf{h} \in \mathcal{H}$ be a hypothesis and $\mathbf{f}$ the target (i.e. to-be-learned) function. We are interested in the set

$$Diff(\mathbf{f}, \mathbf{h}) := \{x : \mathbf{h}(x) \neq \mathbf{f}(x)\}.$$

We denote with $error(\mathbf{h})$ the probability of a randomly selected example being in $Diff(\mathbf{f}, h)$. With $\epsilon > 0$ the hypothesis $\mathbf{h}$ is called ϵ approximatively correct, if $error(\mathbf{h}) \leq \epsilon$ holds.

Question:

$\epsilon > 0$ is given. How many examples must the training set contain to make sure that the hypothesis created by a learning algorithm is ϵ approximatively correct?

Question is wrongly stated!

- Different sets of examples lead to different propositions and with it to different values of ϵ. It depends not only on **how many** but also which examples are chosen.
  For this reason we reformulate our question more carefully.

Question: More carefully and precisely stated.

Let $\epsilon > 0$ and $\delta > 0$ be given. How many examples must the training-set contain to make sure that the **hypothesis** computed by a learning-algorithm is ϵ **approximatively correct with a probability of at least** $1 - \delta$?

- We want to abstract from particular learning-algorithms and make a statement about all possible learning algorithms.
  So we assume only that a learning-algorithm **calculates a hypothesis that is consistent with all previous examples**. Our result holds for this class of learning algorithms.

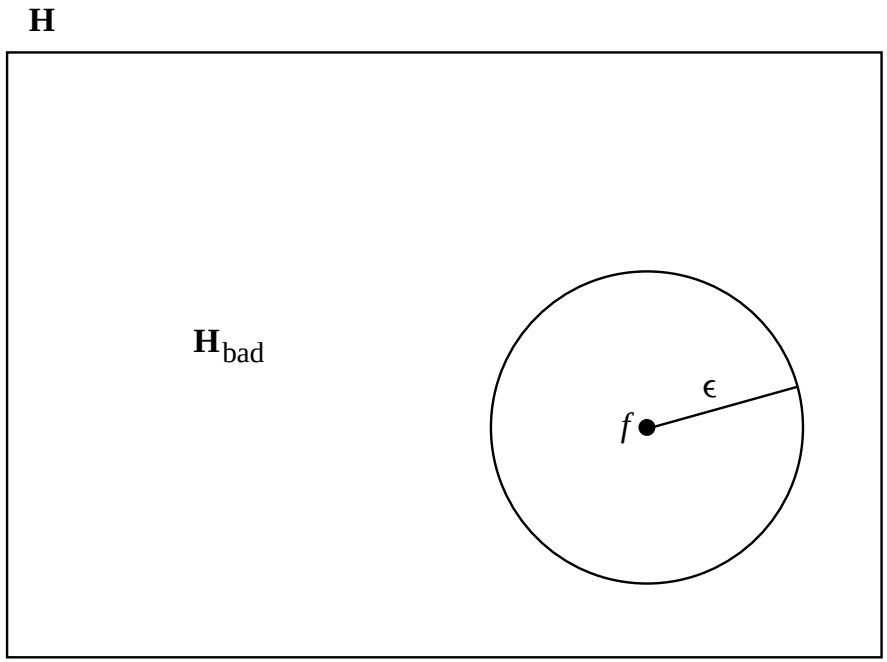
Definition 3.11 (Example Complexity)

Let $\delta > 0$ and $\epsilon > 0$ be given. The example complexity is the number m of examples an arbitrary learning algorithm needs so that the created hypothesis h is ϵ approximatively correct with the probability $1 - \delta$.

Theorem 3.12 (Example Complexity)

The example complexity $\mathbf{m}$ *depends on* ϵ, δ *and the hypotheses-space* $\mathcal{H}$ *as follows:*

$$\mathbf{m} \geq \frac{1}{\epsilon}(ln\frac{1}{\delta} + ln|\mathcal{H}|)$$



Question:

What does the last result mean?

Question:

The complexity depends on $\log(|\mathcal{H}|)$. **What does this mean for boolean functions with n arguments?**

Answer:

We have $\log(|\mathcal{H}|) = 2^n$. Thus one needs exponentially-many examples even if one is satisfied with ϵ approximative correctness under a certain probability!

Proof (of the theorem):

$h_b \in \mathcal{H}$ with $error(h_b) > \epsilon$.

- **What is the probability P_b of h_b being consistent with the chosen examples?** For one single example: it is $\leq (1-\epsilon)$ because of the definition of $error$. Therefore:

$$P_b \leq (1-\epsilon)^m$$

- **What is the probability P' that there is a hypothesis h_b with $error(h_b) > \epsilon$ and consistent with m examples at all?**

$$\begin{aligned} P' &\leq |\mathcal{H}_{bad}|(1-\epsilon)^m \\ &\leq |\{h : error(h) > \epsilon\}|(1-\epsilon)^m \\ &\leq |\mathcal{H}|(1-\epsilon)^m \end{aligned}$$

Proof (continuation):

- We want $P' \leq \delta$:

$$|\mathcal{H}|(1-\epsilon)^m \leq \delta$$

After some transformations:

$$\begin{aligned}
(1-\epsilon)^m &\leq \frac{\delta}{|\mathcal{H}|} \\
m \ln(1-\epsilon) &\leq \ln(\delta) - \ln(|\mathcal{H}|) \\
m &\geq -\frac{1}{\ln(1-\epsilon)}(\ln(\tfrac{1}{\delta}) + \ln(|\mathcal{H}|)) \\
m &\geq \frac{1}{\epsilon}(\ln(\tfrac{1}{\delta}) + \ln(|\mathcal{H}|))
\end{aligned}$$

The last line holds because of $\ln(1-\epsilon) < -\epsilon$. □

Essential result:

Learning is never better than looking up in a table!

1st way out: We ask for a more **specialised hypothesis** instead of one that is just consistent (complexity gets worse).

2nd way out: We give up on learning **arbitrary boolean functions** and concentrate on appropriate subclasses.

We consider Decision lists.

Theorem 3.13 (Decision Lists can be learned)

Learning functions in k-$DL(n)$ (decision lists with a maximum of k tests) has a PAC-complexity of

$$m = \frac{1}{\epsilon}(\ln(\frac{1}{\delta}) + O(n^k log_2(n^k))).$$

▸ Decision Lists

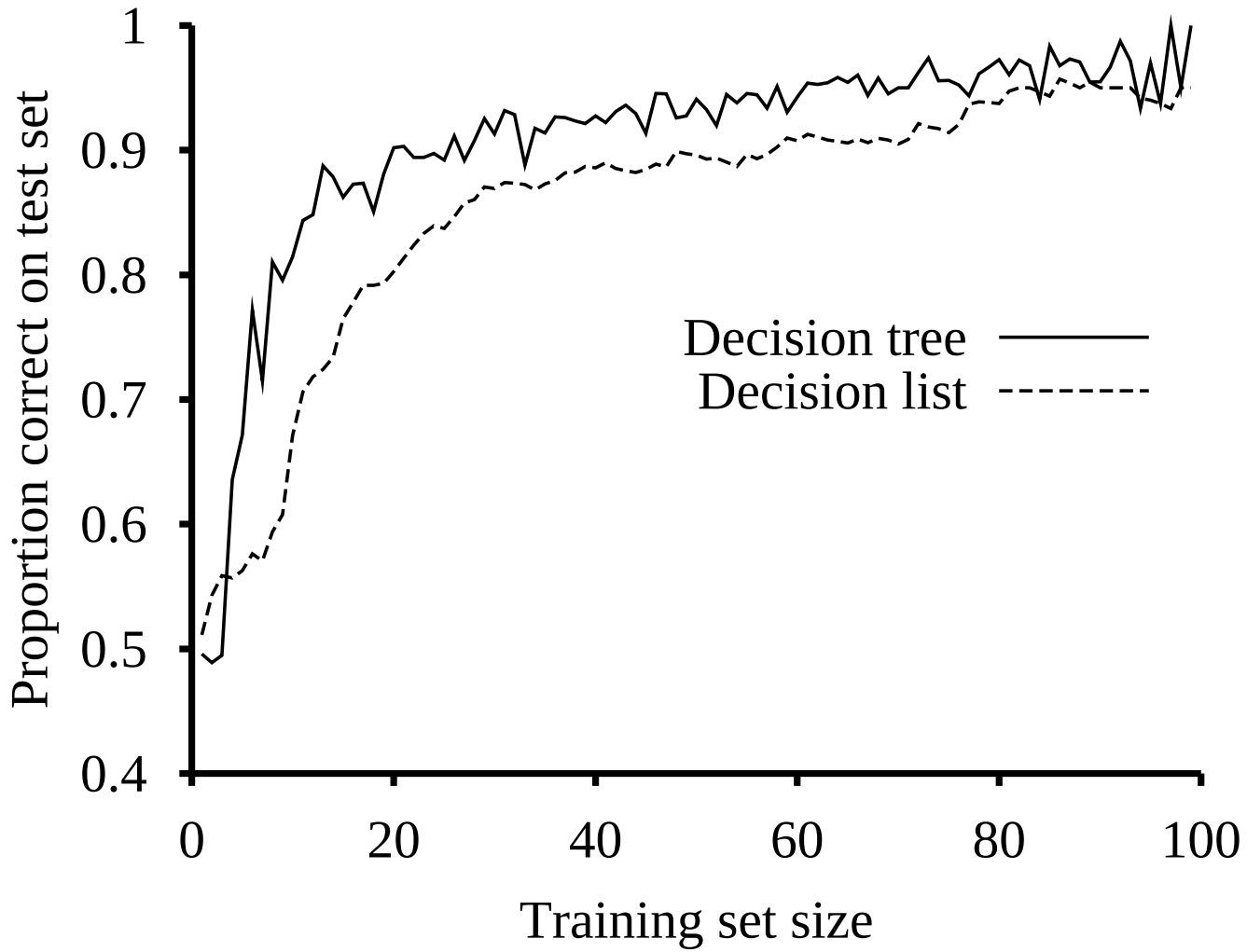
Each algorithm which returns a consistent decision list for a set of examples can be turned into a PAC-learning-algorithm, which learns a k-$DL(n)$ function after a maximum of m examples.

Important estimates (exercises):

- $|Conj(n,k)| = \sum_{i=0}^{k} \binom{2n}{i} = \mathsf{O}(n^k)$,
- $|k\text{-}DL(n)| \leq 3^{|Conj(n,k)|}|Conj(n,k)|!$,
- $|k\text{-}DL(n)| \leq 2^{\mathsf{O}(n^k log_2(n^k))}$.

function DECISION-LIST-LEARNING(*examples*) **returns** a decision list, *No* or failure

if *examples* is empty **then return** the value *No*
t $\leftarrow$ a test that matches a nonempty subset $examples_t$ of *examples*
such that the members of $examples_t$ are all positive or all negative
if there is no such *t* **then return** failure
if the examples in $examples_t$ are positive **then** $o \leftarrow Yes$
else $o \leftarrow No$
return a decision list with initial test *t* and outcome *o*
and remaining elements given by DECISION-LIST-LEARNING($examples - examples_t$)


Proportion correct on test set
1
0.9
0.8
0.7
0.6
0.5
0.4
0
20
40
60
80
100
Training set size
Decision tree
Decision list

3.6 Noise and overfitting

Noise:

- examples are inconsistent ($Q(x)$ together with $\neg Q(x)$),
- no attributes left to classify more examples,
- makes sense if the environment is **stochastic**.

Overfitting: Dual to noise.

- remaining examples can be classified using attributes which establish a *pattern*, which is not existent (irrelevant attributes).

Example 3.14 (Tossing Dice)

Several coloured dice are tossed. Every toss is described via (day, month, time, colour). As long as there is no inconsistency every toss is described by exactly one (totally overfitted) hypothesis.

Other examples:

- the pyramids,
- astrology,
- “Mein magisches Fahrrad”.

4. Learning in networks

Content of this chapter:

Neural Nets: Neural nets are an abstraction from entities operating in our brain.

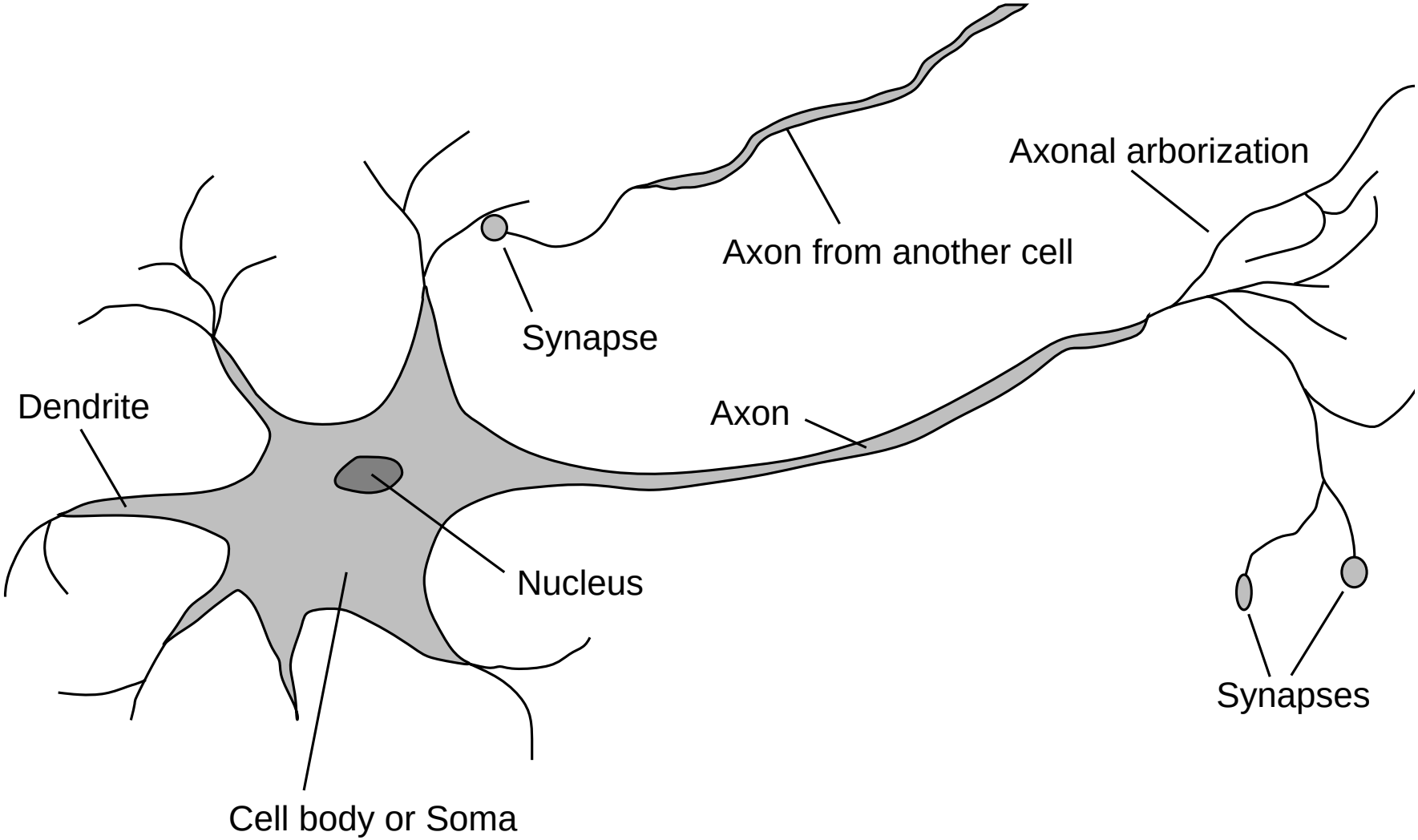
Perceptrons: the perceptron is a particularly simple model. We describe the perceptron learning algorithm, which converges for each representable function: The linear separable functions.

Feed Forward Nets: We illustrate back propagation by simulating methods and techniques from perceptrons.

Learning with networks is a method to

- build **complex functions** from **many very simple but connected units** and to learn this construction from examples,
- improve the **understanding** of the functionality of **the human brain**.

4.1 The human brain


Axonal arborization
Axon from another cell
Synapse
Dendrite
Axon
Nucleus
Synapses
Cell body or Soma

A neuron consists of

- **the soma**: the body of the cell,
- **the nucleus**: the core of the cell,
- **the dendrites**,
- **the axon**: 1 cm - 1 m in length.

The axon branches and connects to the dendrites of other neurons: these locations are called **synapses**. Each neuron shares synapses with

10-100000 others.

Signals are **propagated** from neuron to neuron by a complicated electrochemical reaction.

Chemical transmitter substances are released from the synapses and enter the dendrite, raising or lowering the electrical potential of the cell body.

When the **potential reaches a threshold** an electrical pulse or action potential is sent along the axon.

The pulse spreads out along the branches of the axons and releases transmitters into the bodies of other cells.

Question:

How does the building process of the network of neurons look like?

Answer:

Long term changes in the strength of the connections are in response to the pattern of stimulation.

Biology versus electronics:

	Computer	Human Brain
Computational units	1 CPU, 10^5 gates	10^{11} neurons
Storage units	10^9 bits RAM, 10^{10} bits disk	10^{11} neurons, 10^{14} synapses
Cycle time	10^{-8} sec	10^{-3} sec
Bandwidth	10^9 bits/sec	10^{14} bits/sec
Neuron updates/sec	10^5	10^{14}

Computer: sequential processes, very fast, **“rebooting quite often”**

Brain: works profoundly **concurrently**, quite slow, **error-correcting**, fault-tolerant (neurons die constantly)

4.2 Neural networks

Definition 4.1 (Neural Network)

A neural network consists of:

1. **units**,
2. **links** between units.

The links are **weighted**. There are three kinds of units:

1. **input** units,
2. **hidden** units,
3. **output** units.

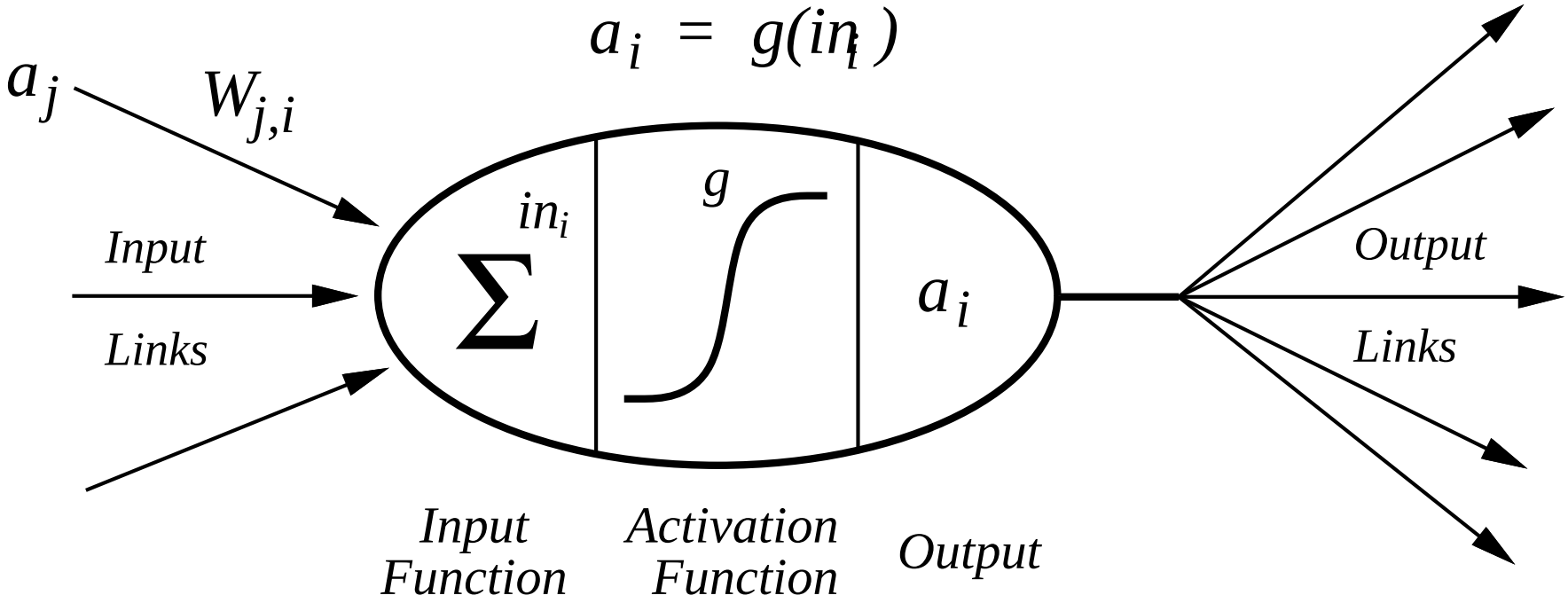
Idea:

A unit i receives an input via links to other units j.
The input function

$$in_i := \sum_j W_{j,i}\, a_j$$

calculates the weighted sum.

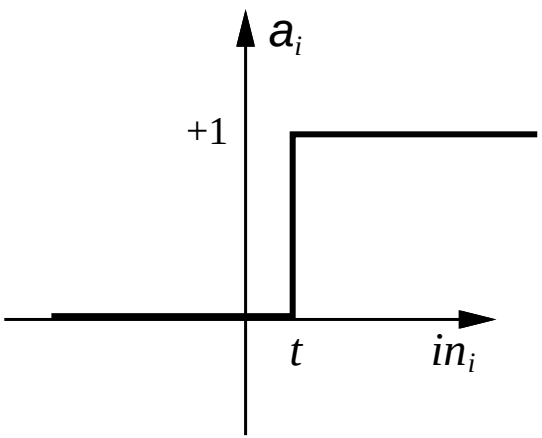
Notation	Meaning
a_i $\mathbf{a}_i$	Activation value of unit i (also the output of the unit) Vector of activation values for the inputs to unit i
g g'	Activation function Derivative of the activation function
Err_i Err^e	Error (difference between output and target) for unit i Error for example e
I_i $\mathbf{I}$ $\mathbf{I}^e$	Activation of a unit i in the input layer Vector of activations of all input units Vector of inputs for example e
in_i	Weighted sum of inputs to unit i
N	Total number of units in the network
O O_i $\mathbf{O}$	Activation of the single output unit of a perceptron Activation of a unit i in the output layer Vector of activations of all units in the output layer
t	Threshold for a step function
T $\mathbf{T}$ $\mathbf{T}^e$	Target (desired) output for a perceptron Target vector when there are several output units Target vector for example e
$W_{j,i}$ W_i $\mathbf{W}_i$ $\mathbf{W}$	Weight on the link from unit j to unit i Weight from unit i to the output in a perceptron Vector of weights leading into unit i Vector of all weights in the network


$a_i = g(in_i)$
a_j
$W_{j,i}$
Input
Links
in_i
Σ
g
a_i
Output
Links
Input Function
Activation Function
Output

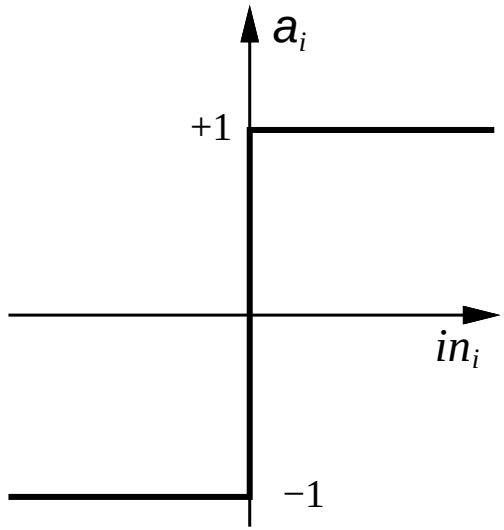
The **activation function** g calculates the output a_i (from the inputs) which will be transferred to other units via output-links:

$$a_i := g(in_i)$$

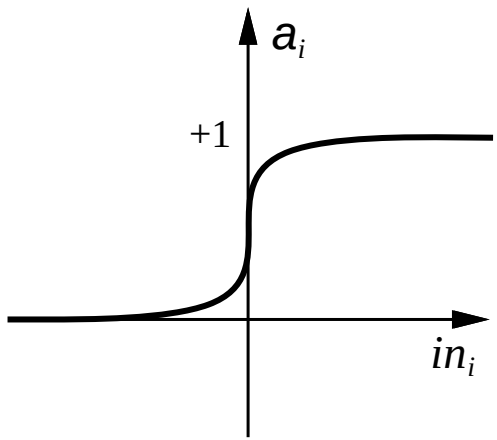
Examples:



(a) Step function



(b) Sign function



(c) Sigmoid function

Standardisation:

We consider $step_0$ instead of $step_t$.
If an unit i uses the activation function $step_t(x)$ then we bring in an additional input link "0" which adds a constant value of $a_0 := -1$. This value is weighted as $W_{0,i} := t$. Now we can use $step_0$ for the activation function:

$$step_t(\sum_{j=1}^{n} W_{j,i}a_j) \;=\; step_0(-t + \sum_{j=1}^{n} W_{j,i}a_j) \;=\; step_0(\sum_{j=0}^{n} W_{j,i}a_j)$$

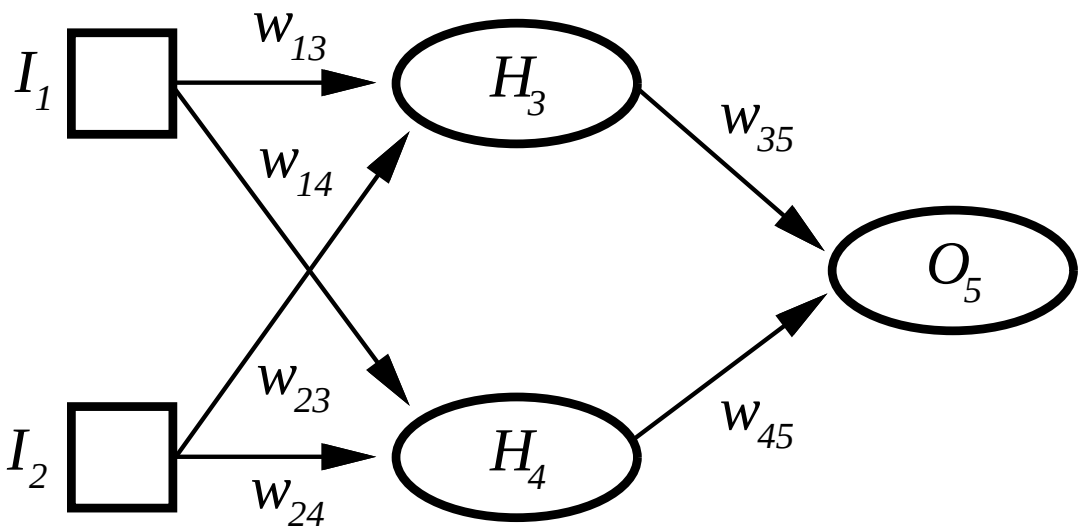
Networks can be

- recurrent, i.e. somehow connected,
- feed-forward, i.e. they form an acyclic graph.

Usually networks are partitioned into **layers**:

- units in one layer have only links to units of the next layer.

E.g. multi-layer feed-forward networks: without internal states (no short-term memory).



Important:

The output of the input-units is determined by the environment.

Question 1:

Which function does the figure describe?

Question 2:

Why can **non-trivial functions** be represented at all?

Question 3:

How many units do we need?

- **a few:** a small number of functions can be represented,
- **many:** the network **learns by heart** ($\rightsquigarrow$ overfitting)

Hopfield networks:

1. bidirectional links with symmetrical weights,
2. activation function: $sign$,
3. units are input- and output-units,
4. can store up to $0.14\,N$ training examples.

Learning with neural networks means the adjustment of the parameters to ensure consistency with the training-data.

Question:

How to find the optimal network structure?

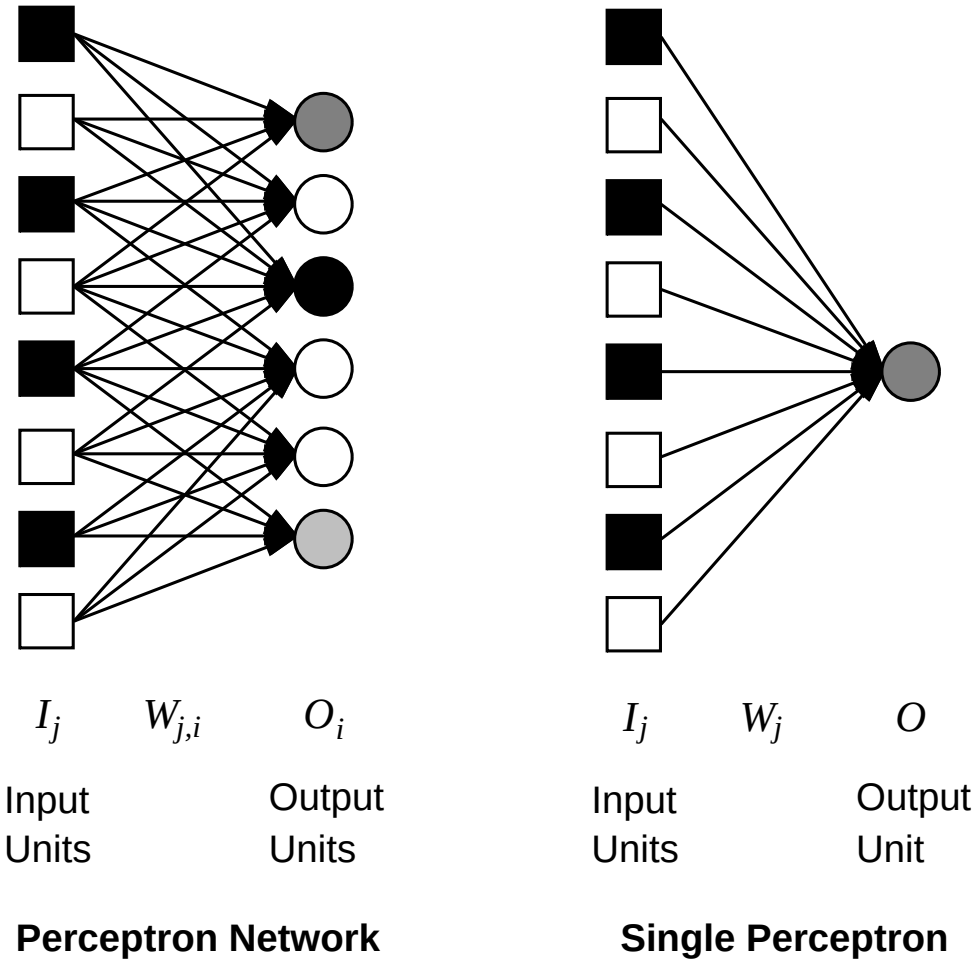
Answer:

Perform a search in the space of network structures (e.g. with genetic algorithms).

4.3 The Perceptron

Definition 4.2 (Perceptron)

A perceptron is a feed-forward network with one layer based on the activation function $step_0$.



Perceptron Network **Single Perceptron**

Question:

Can all boolean functions be represented by a feed-forward network?

Can AND, OR and NOT be represented?

Is it possible to represent every boolean function by simply combining these?

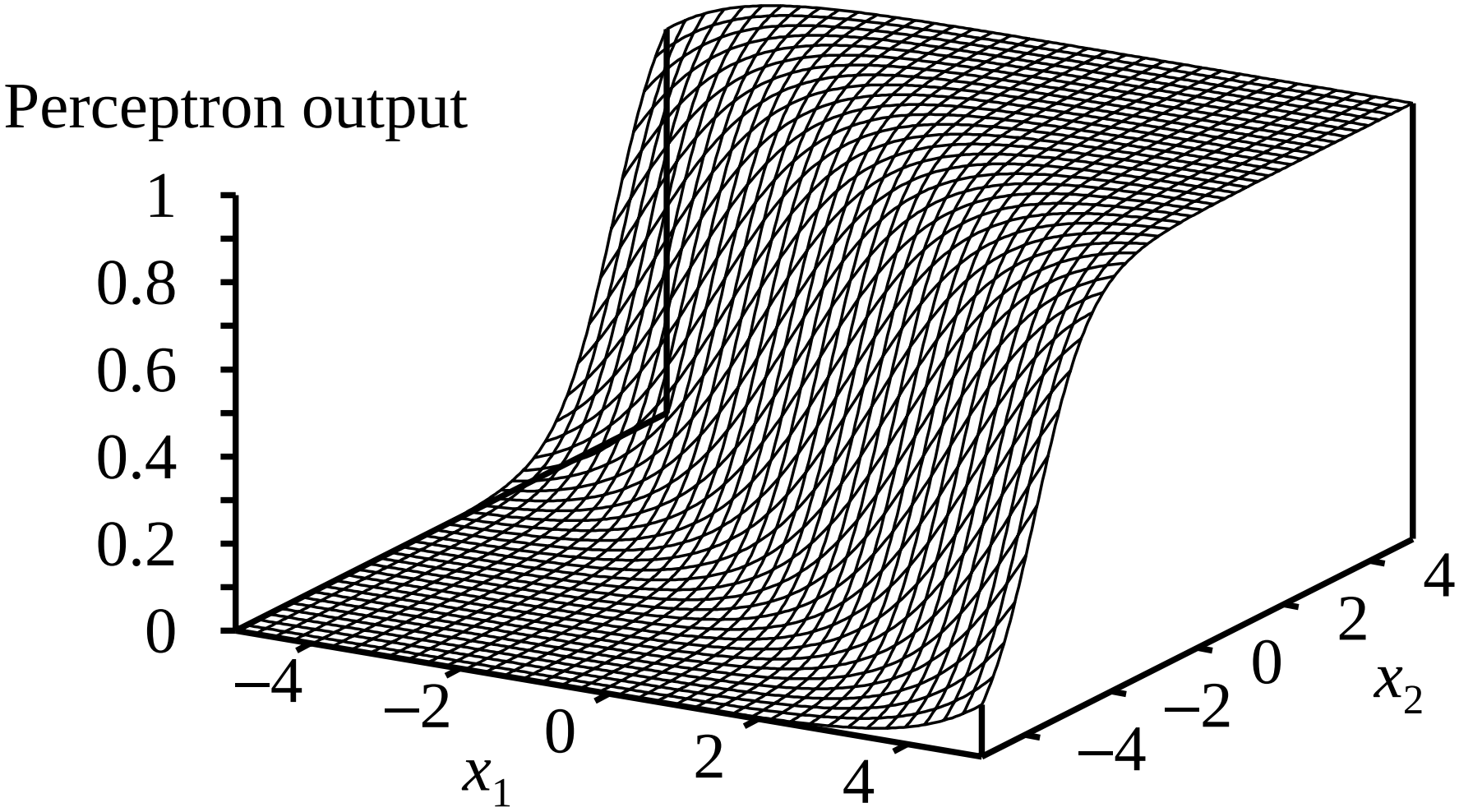
What about

$$f(x_1, \ldots, x_n) := \begin{cases} 1, & \text{if } \sum_{i=1}^{n} x_i > \frac{n}{2} \\ 0, & \text{else.} \end{cases}$$

Solution:

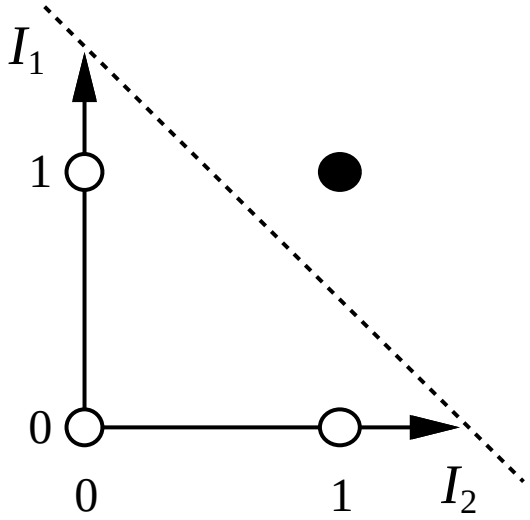
- Every boolean function can be composed using AND, OR and NOT (or even only $NAND$).
- The combination of the respective perceptrons is not a perceptron!

Perceptron with sigmoid activation

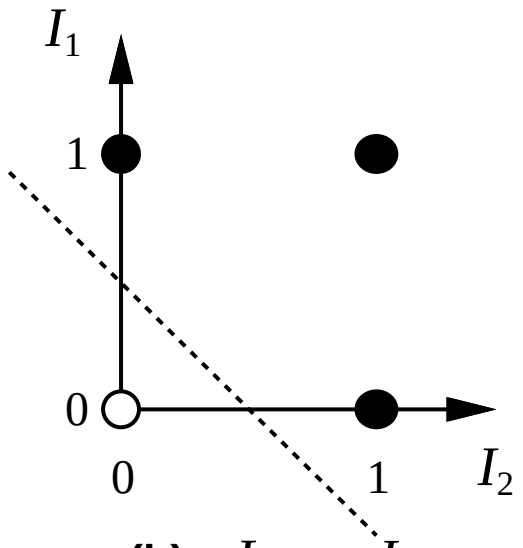


Question:

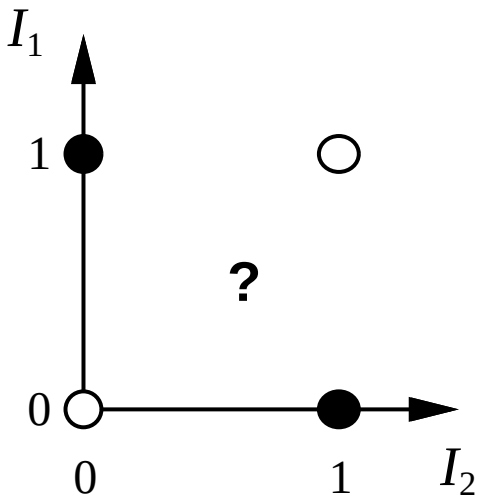
What about XOR?



(a) I_1 **and** I_2



(b) I_1 **or** I_2



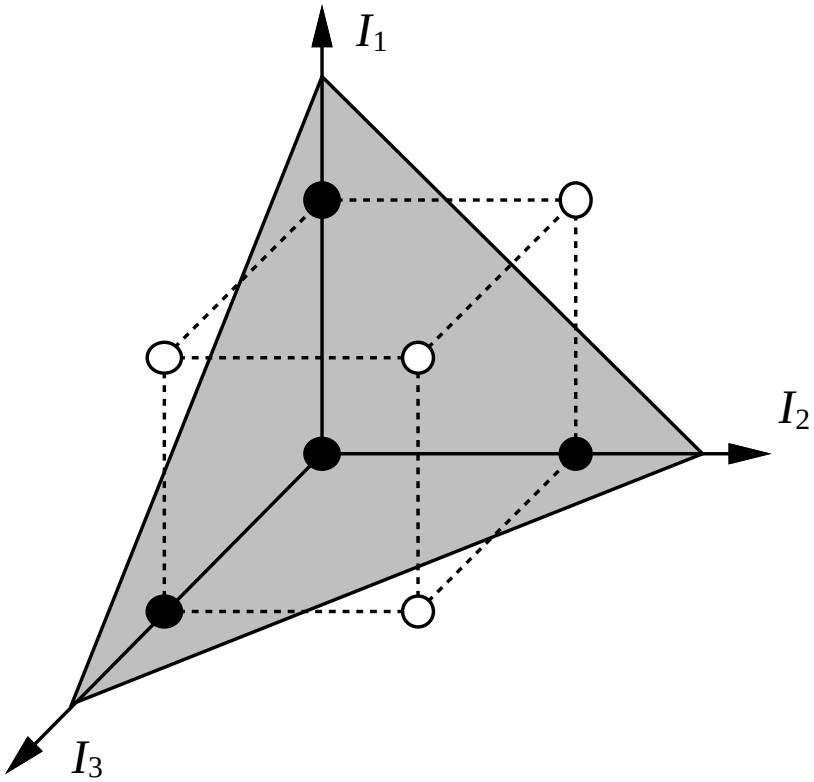
(c) I_1 **xor** I_2

$$\text{Output} = step_0(\sum_{j=1}^{n} W_j I_j)$$

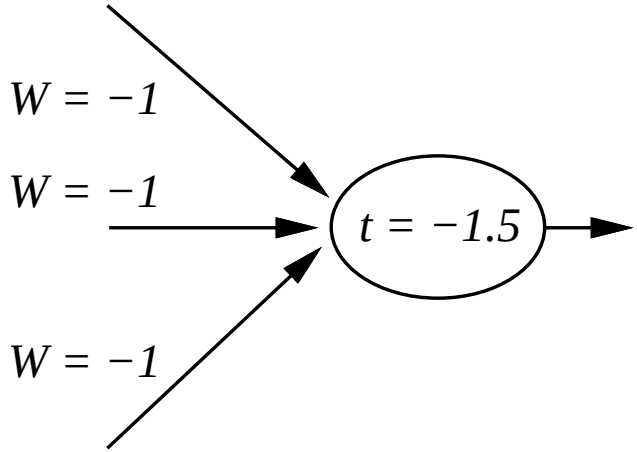
$\sum_{j=1}^{n} W_j I_j = 0$ defines a **n-dimensional hyperplane**.

Definition 4.3 (Linear Separable)

A boolean function with n attributes is called linear separable if there is a hyperplane (**$(n-1)$-dimensional subspace**) which separates the positive domain-values from the negative ones.



(a) Separating plane



(b) Weights and threshold

Learning algorithm:

Similar to **current best hypothesis** ($\rightsquigarrow$ chapter on learning).

- **hypothesis:** network with the current weights (firstly randomly generated)
- **UPDATE:** make it consistent through small changes.
  Important: for each example **UPDATE is called several times**. These calls are called **epochs**.

function NEURAL-NETWORK-LEARNING(*examples*) **returns** *network*

network ← a network with randomly assigned weights
repeat
  for each *e* **in** *examples* **do**
    O ← NEURAL-NETWORK-OUTPUT(*network*, *e*)
    T ← the observed output values from *e*
    update the weights in *network* based on *e*, **O**, and **T**
  end
until all examples correctly predicted or stopping criterion is reached
return *network*

Definition 4.4 (Perceptron Learning ($step_0$)

Perceptron learning modifies the weights W_j with respect to this rule:

$$W_j := W_j + \alpha \times I_j \times \textbf{Error}$$

with **Error**:$= T - O$ (i.e. the difference between the correct and the current output-value). α is the **learning rate**.

Definition 4.5 (Perceptron Learning ($sigmoid$)

Perceptron learning modifies the weights W_j with respect to this rule:

$$W_j := W_j + \alpha \times g'(in)\, I_j \times \textbf{Error}$$

with **Error**:$= T - O$ (i.e. the difference between the correct and the current output-value). α is the **learning rate**.

The following algorithm uses $x_j[e]$ for I_j.

function PERCEPTRON-LEARNING($examples, network$) **returns** a perceptron hypothesis
inputs: $examples$, a set of examples, each with input $\mathbf{x} = x_1, \ldots, x_n$ and output y
$network$, a perceptron with weights $W_j,\ j = 0 \ldots n$, and activation function g

repeat
for each e **in** $examples$ **do**
$in \leftarrow \sum_{j=0}^{n} W_j\ x_j[e]$
$Err \leftarrow y[e] - g(in)$
$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
until some stopping criterion is satisfied
return NEURAL-NET-HYPOTHESIS($network$)

Theorem 4.6 (Rosenblatt's Theorem)

Every function which can be represented by a perceptron is learned through the perceptron learning algorithm (Definition 4.4).

More exactly: The series W_j converges to a function which represents the examples correctly.

Proof:

Let $\hat{w}$ be a solution, wlog. we can assume (why? $\rightsquigarrow$ exercise)

$$\hat{w}\,\overrightarrow{I} > 0 \text{ for all } \overrightarrow{I} \in \mathbf{I_{pos}} \cup -\mathbf{I_{neg}}$$

with $\mathbf{I_{pos}}$ consisting of the positive and $\mathbf{I_{neg}}$ consisting of the negative examples (and $-\mathbf{I_{neg}} = \{-\overrightarrow{I} : \overrightarrow{I} \in \mathbf{I_{neg}}\}$).
Let $\mathbf{I'} := \mathbf{I_{pos}} \cup -\mathbf{I_{neg}}$ and $m := min\{\hat{w}\,\overrightarrow{I} : \overrightarrow{I} \in \mathbf{I'}\}$.
$\overrightarrow{w_1}, \ldots, \overrightarrow{w_j}, \ldots$ be the sequence of weights resulting from the algorithm.
We want to show that this sequence eventually becomes constant.

Proof (continued):

Consider the possibility that not all $\overrightarrow{w_i}$ are different from their predecessor (or successor) (this is the case if the new example is not consistent with the current weights). Let $k_1, k_2, \ldots, k_j$ be the indices of the changed weights (where error is non-zero), i.e.

$$\overrightarrow{w_{k_j}}\overrightarrow{I_{k_j}} \leq 0, \quad \overrightarrow{w_{k_j+1}} = \overrightarrow{w_{k_j}} + \alpha\overrightarrow{I_{k_j}}.$$

With $\overrightarrow{I_{k_j}}$ being the k_j-th tested example in $\mathbf{I}'$ (which is not consistent (wrt the definition of k_j)).
The we have

$$\overrightarrow{w_{k_j+1}} = \overrightarrow{w_{k_1}} + \alpha\overrightarrow{I_{k_1}} + \alpha\overrightarrow{I_{k_2}} + \ldots + \alpha\overrightarrow{I_{k_j}}$$

Proof (continued):

We use this to show that j cannot become arbitrarily big. We compose

$$\cos\omega = \frac{\hat{w}\overrightarrow{w_{k_j+1}}}{\|\hat{w}\|\,\|\overrightarrow{w_{k_j+1}}\|}$$

and estimate as follows (by decreasing the numerator and increasing the denominator):

$$\cos\omega = \frac{\hat{w}\overrightarrow{w_{k_j+1}}}{\|\hat{w}\|\,\|\overrightarrow{w_{k_j+1}}\|} \geq \frac{\hat{w}\overrightarrow{w_{k_1}} + \alpha m j}{\|\hat{w}\|\sqrt{\|\overrightarrow{w_{k_1}}\|^2 + \alpha^2 M j}}$$

The right side converges to infinity (when j increases to infinity) and cosinus will never get greater than 1. This leads to the contradiction.

Proof (continued):

How do we get the estimation above?

- the scalar product $\hat{w}\overrightarrow{w_{k_j+1}}$ leads to $\hat{w}\overrightarrow{w_{k_1}} + \alpha\hat{w}\,(\,\overrightarrow{I_1} + \ldots + \overrightarrow{I_j}\,) \geq \hat{w}\overrightarrow{w_{k_1}} + \alpha m j.$
- $\|\overrightarrow{w_{k_j+1}}\|^2 = \|\overrightarrow{w_{k_j}} + \alpha\overrightarrow{I_{k_j}}\|^2 = \|\overrightarrow{w_{k_j}}\|^2 + 2\alpha\overrightarrow{I_{k_j}}\overrightarrow{w_{k_j}} + \alpha^2\|\overrightarrow{I_{k_j}}\|^2 \leq \|\overrightarrow{w_{k_j}}\|^2 + \alpha^2\|\overrightarrow{I_{k_j}}\|^2$, da $\overrightarrow{I_{k_j}}\overrightarrow{w_{k_j}} \leq 0$, this is how we have chosen k_j.
  Now be $M := max\,\{\|\overrightarrow{I}\|^2 : \overrightarrow{I} \in \mathbf{I}'\}$. Then $\|\overrightarrow{w_{k_j+1}}\|^2 \leq \|\overrightarrow{w_{k_1}}\|^2 + \alpha^2\|\overrightarrow{I_{k_1}}\|^2 + \ldots + \alpha^2\|\overrightarrow{I_{k_j}}\|^2 \leq \overrightarrow{w_{k_1}}\|^2 + \alpha^2 M j$ holds. □

What is the underlying example?

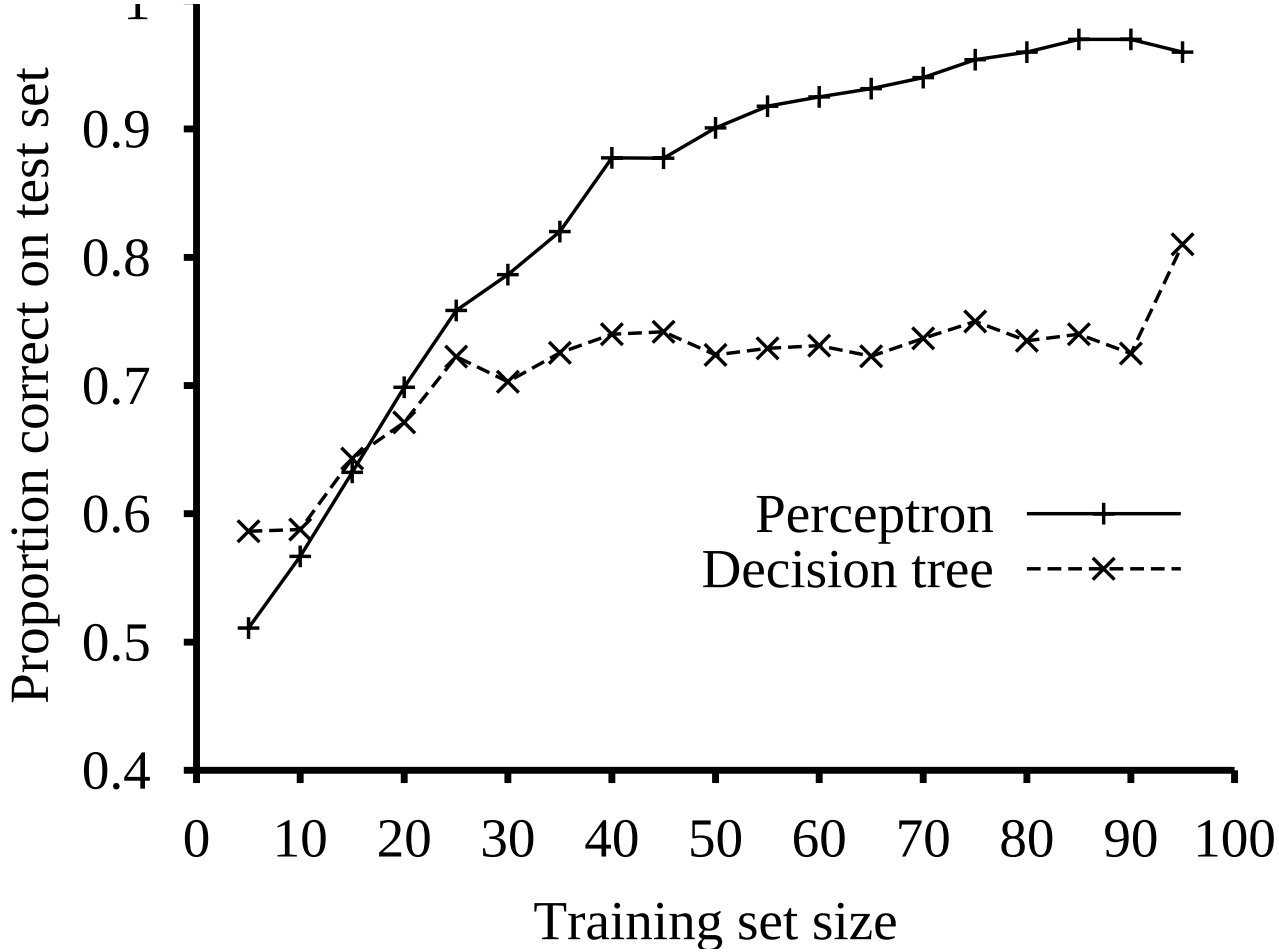


Figure 4.12: Perceptron Better than Decision Tree

What is the underlying example?

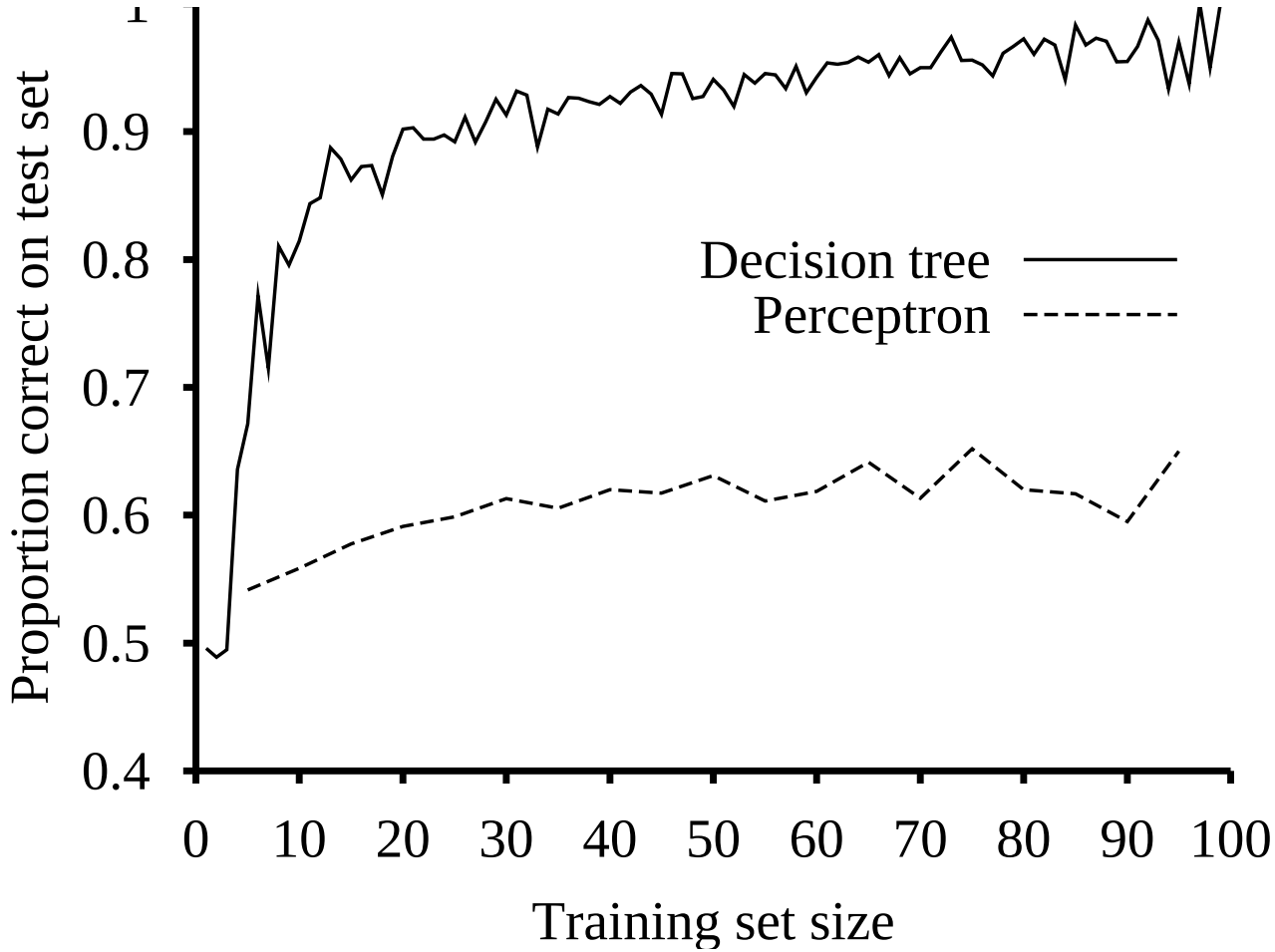


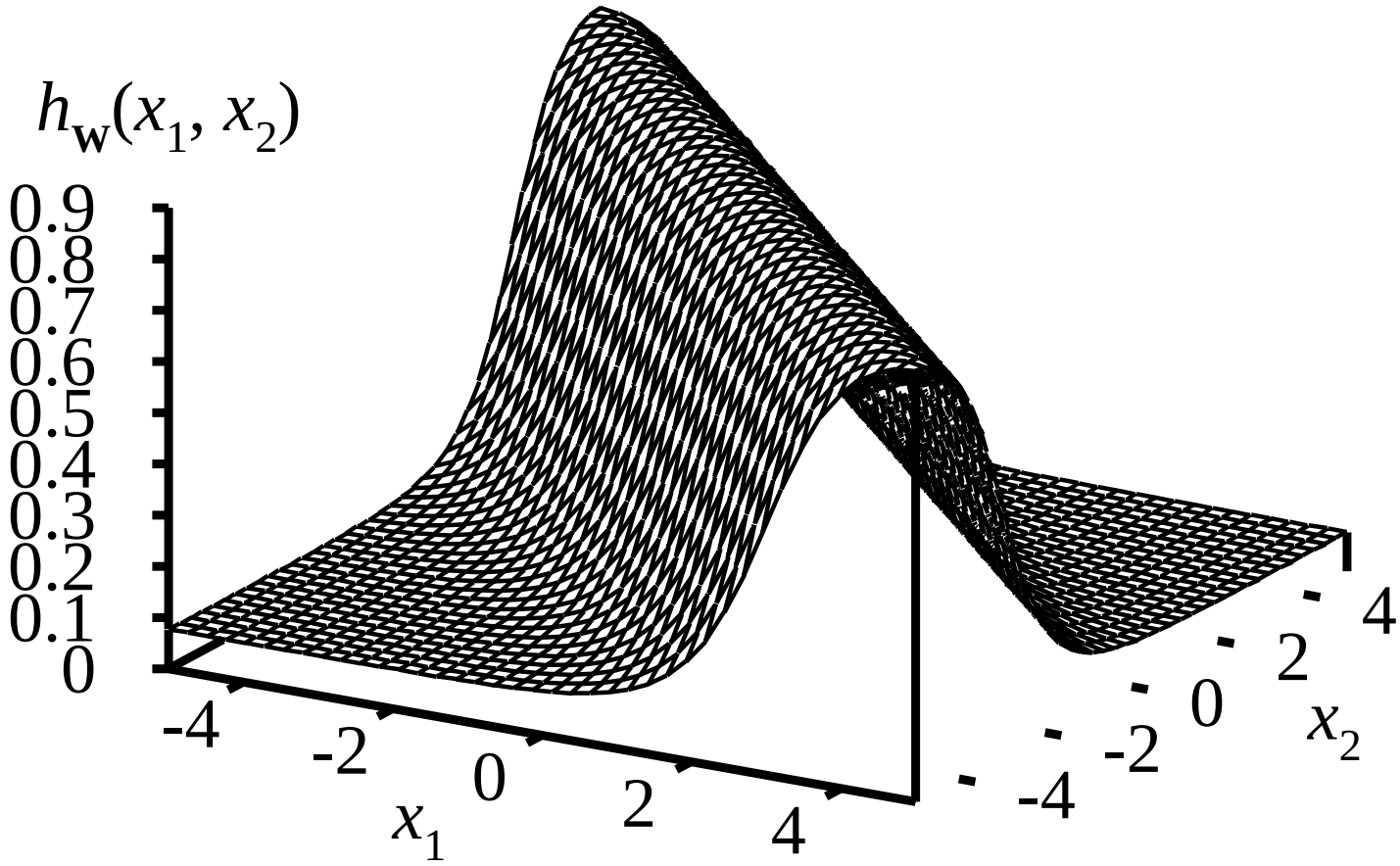
Figure 4.13: Decision Tree Better than Perceptron

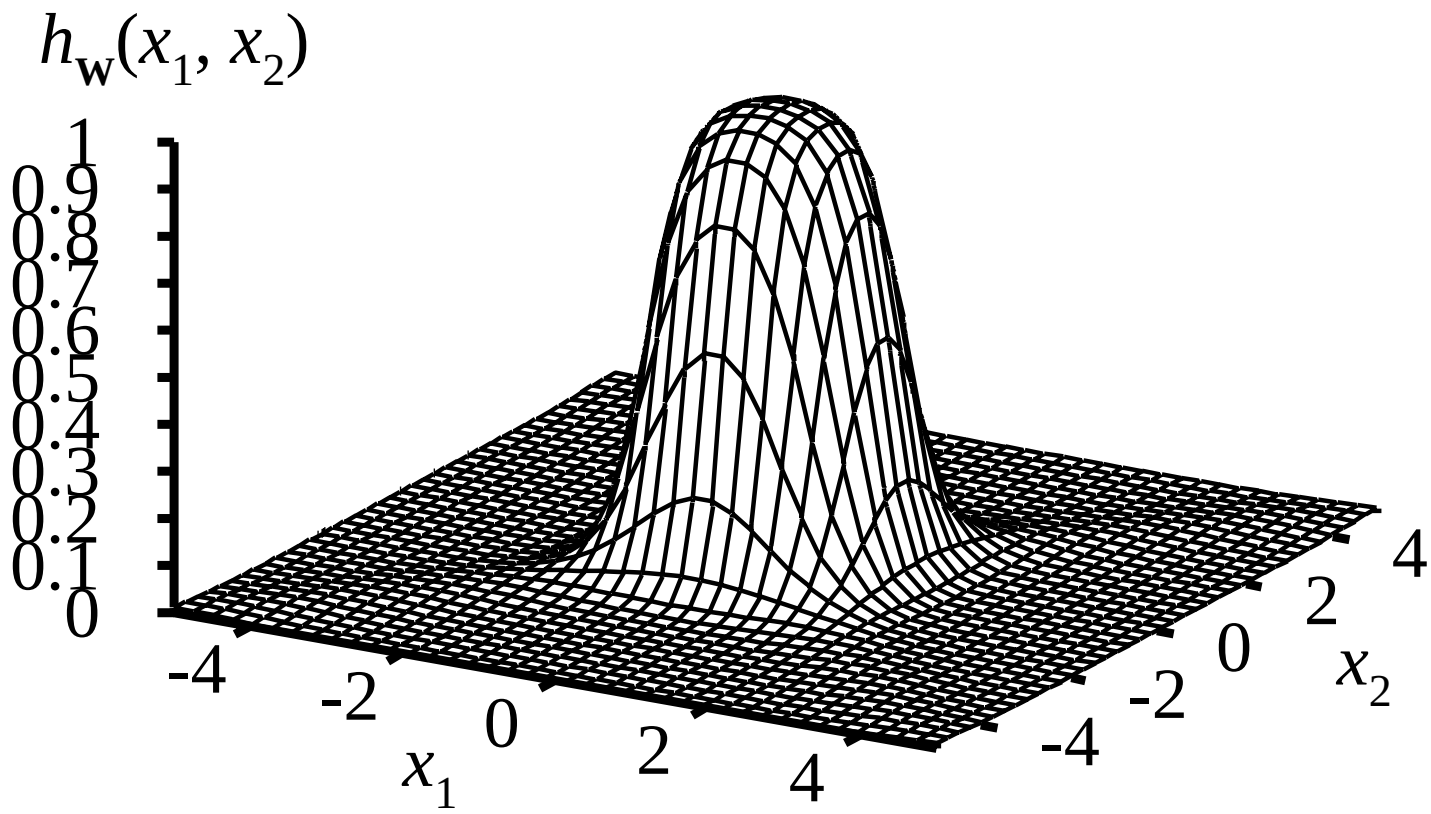
4.4 Multi-layer feed-forward

Problem:

How does the error-function of the hidden units look like?

Learning with multi-layer networks is called back propagation.
Hidden units can be seen as perceptrons (Figure on page 320). The outcome can be a linear combination of such perceptrons (see next two slides).


$h_{\mathbf{w}}(x_1, x_2)$
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-4
-2
0
2
4
x_1
-4
-2
0
2
4
x_2


$h_{\mathbf{w}}(x_1, x_2)$
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-4
-2
0
2
4
x_1
-4
-2
0
2
4
x_2

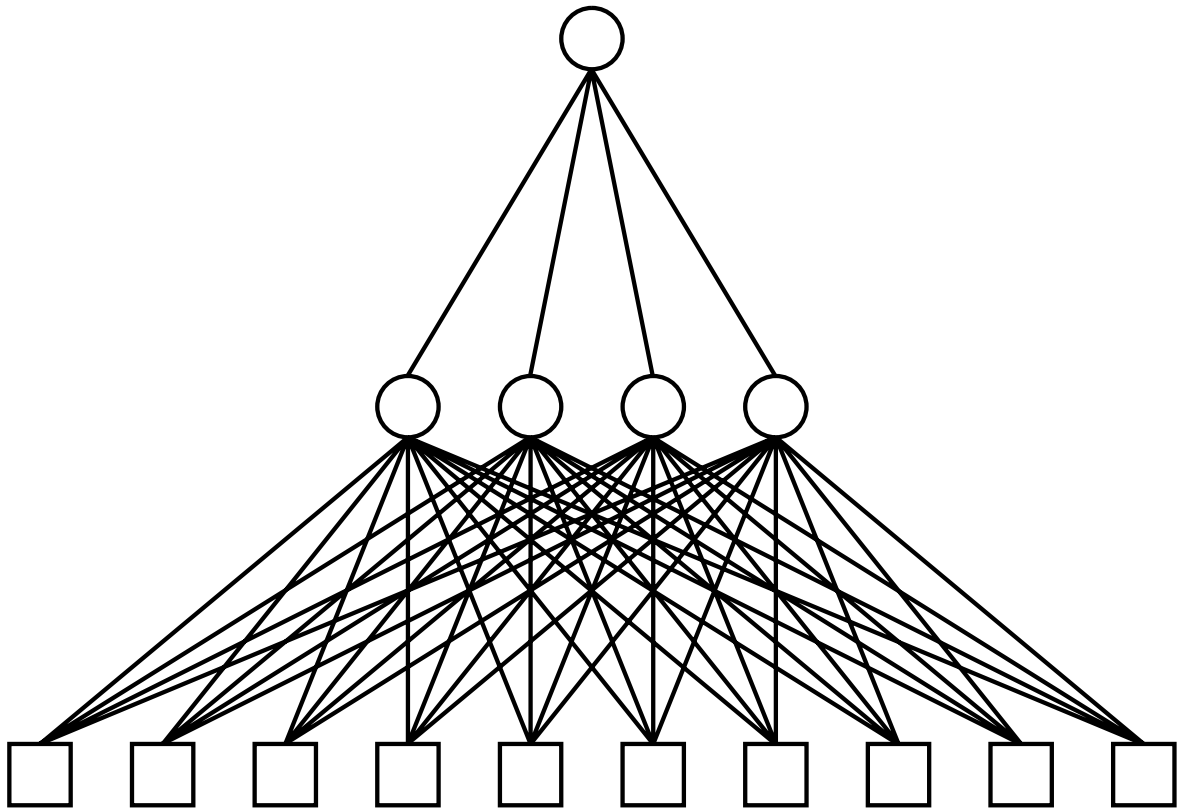
Output units O_i

$W_{j,i}$

Hidden units a_j

$W_{k,j}$

Input units I_k



The perceptron was not powerful enough in our restaurant-example (Figure 335). So we try 2 layers. 10 attributes lead to 10 input-units.

Question

How **many hidden units** are necessary?

Answer:

Four!

Perceptron's error is easily determined because there was only one W_j between input and output. Now we have several.

How should the error be distributed?

Minimising by using derivatives (1)

We minimise $E = \frac{1}{2}\sum_i (T_i - O_i)^2$ and get

$$\begin{aligned} E &= \frac{1}{2}\sum_i (T_i - g(\sum_j W_{j,i} a_j))^2 \\ &= \frac{1}{2}\sum_i (T_i - g(\sum_j W_{j,i} g(\sum_k W_{k,j} I_k)))^2 \end{aligned}$$

Do a gradient descent.

$$\begin{aligned} \frac{\partial E}{\partial W_{j,i}} &= \frac{1}{2}\sum_i (T_i - O_i)^2 \\ &= \frac{1}{2} 2 (T_i - g(\ldots))(-\frac{\partial g}{\partial x}(in_i))a_j \\ &= -a_j (T_i - O_i)\frac{\partial g}{\partial x}(in_i) \\ &= -a_j \Delta_i \end{aligned}$$

Minimising by using derivatives (2)

Now the $W_{k,j}$:

$$\begin{aligned}\frac{\partial E}{\partial W_{k,j}} &= \frac{1}{2}\sum_i \mathbf{(}2(T_i - g(\ldots))(-\frac{\partial g}{\partial x}(in_i))(W_{j,i}\frac{\partial g}{\partial x}(in_j)I_k)\mathbf{)} \\ &= \sum_i \mathbf{(}\Delta_i W_{j,i}\frac{\partial g}{\partial x}(in_j)I_k\mathbf{)} \\ &= \frac{\partial g}{\partial x}(in_j)I_k\sum_i W_{j,i}\Delta_i\end{aligned}$$

Idea:

We perform **two** different updates. One for the weights to the input units and one for the weights to the output units.

output units: similar to the perceptron

$$W_{j,i} := W_{j,i} + \alpha \times a_j \times \text{Error}_i \times g'(in_i)$$

Instead of $\text{Error}_i \times g'(in_i)$ write Δ_i.

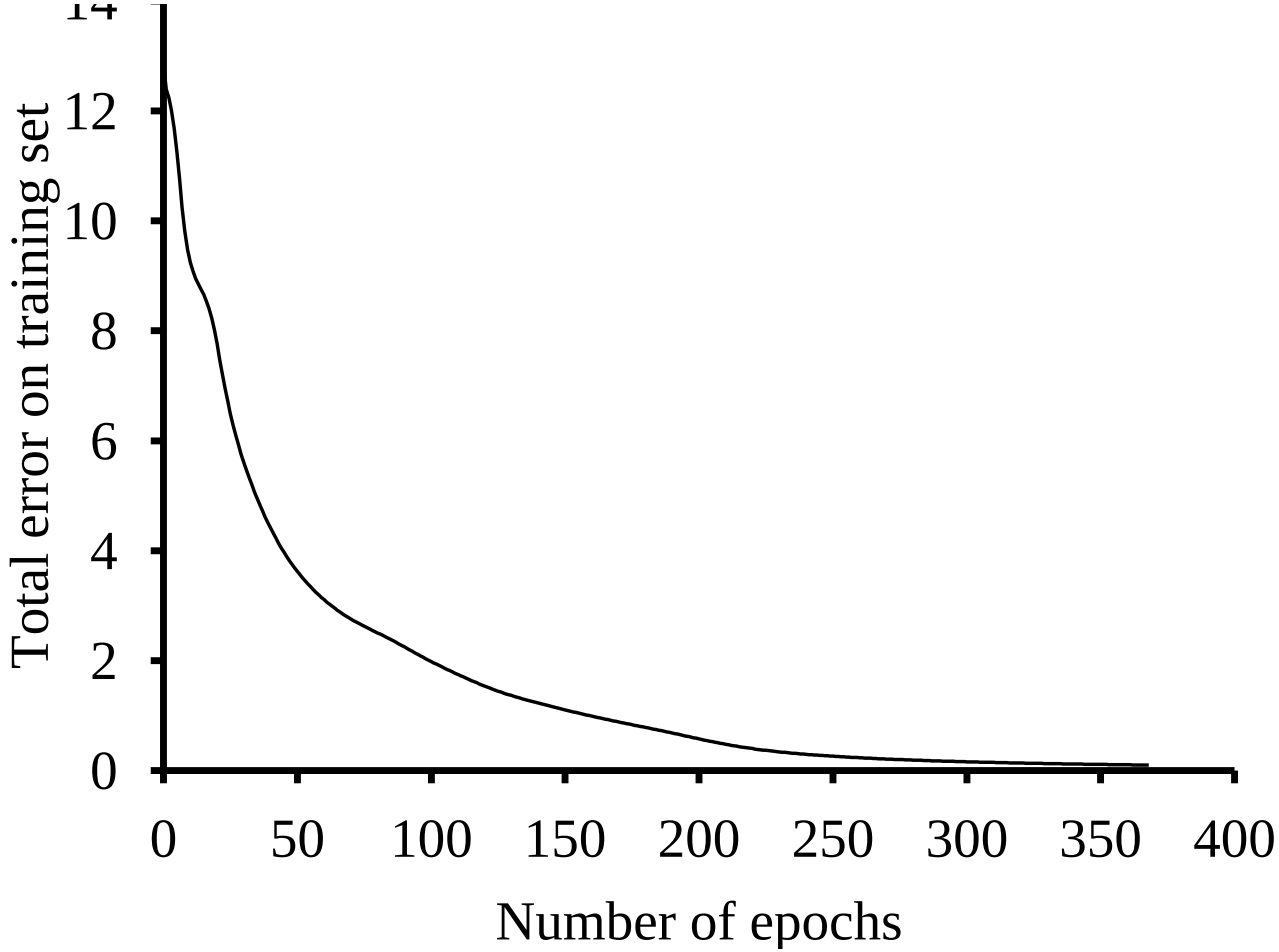
hidden units: each hidden unit j is partly responsible for the error Δ_i (if j is connected with the output unit i).

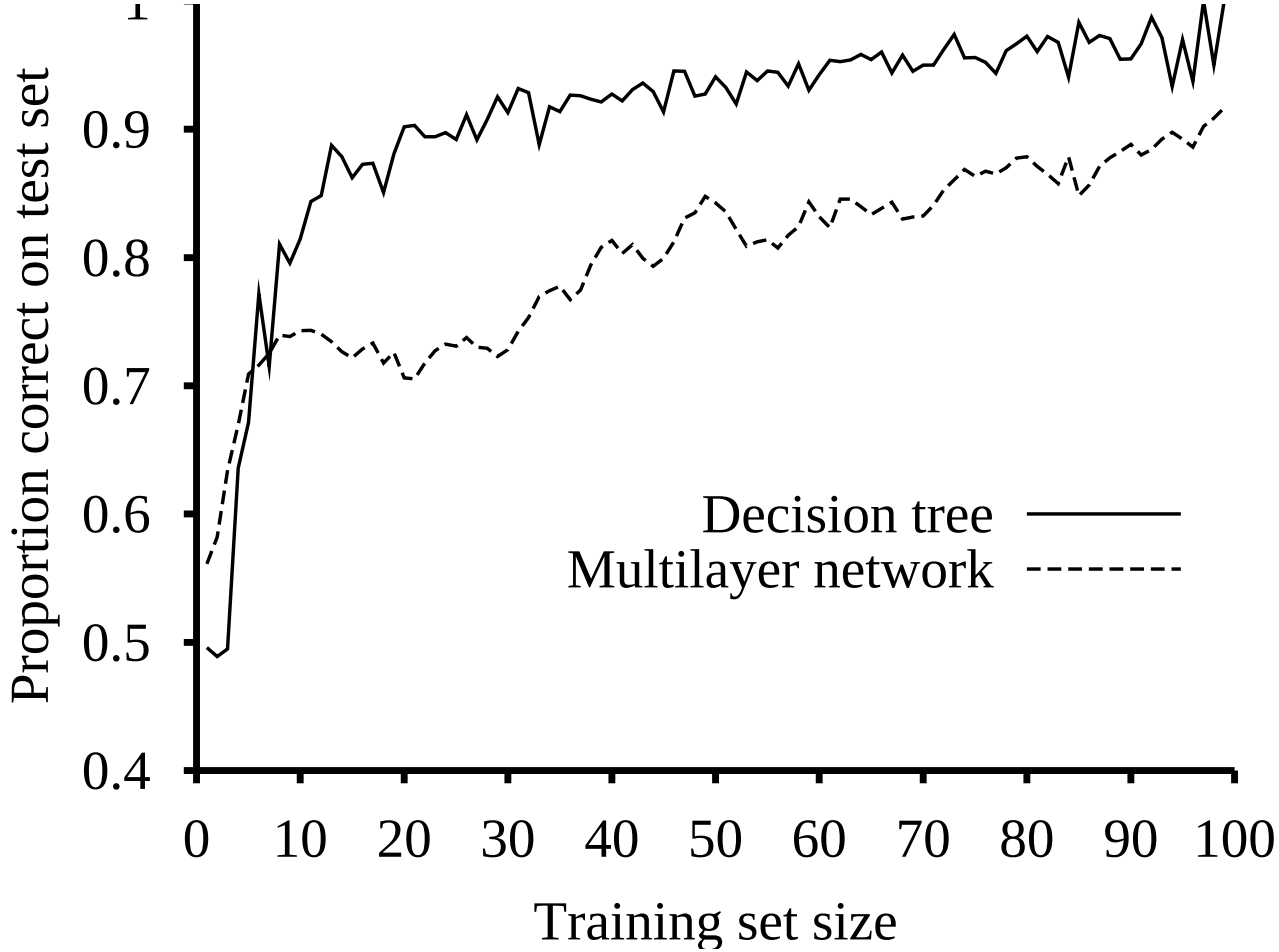
$$W_{k,j} := W_{k,j} + \alpha \times I_k \times \Delta_j$$

with $\Delta_j := g'(in_j) \sum_i W_{j,i} \Delta_i$.

function BACK-PROP-LEARNING($examples$, $network$) **returns** a neural network
inputs: $examples$, a set of examples, each with input vector **x** and output vector **y**
$network$, a multilayer network with L layers, weights $W_{j,i}$, activation function g

repeat
for each e **in** $examples$ **do**
for each node j in the input layer **do** $a_j \leftarrow x_j[e]$
for $\ell = 2$ **to** M **do**
$in_i \leftarrow \sum_j W_{j,i}\ a_j$
$a_i \leftarrow g(in_i)$
for each node i in the output layer **do**
$\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$
for $\ell = M - 1$ **to** 1 **do**
for each node j in layer ℓ **do**
$\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i}\ \Delta_i$
for each node i in layer $\ell + 1$ **do**
$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$
until some stopping criterion is satisfied
return NEURAL-NET-HYPOTHESIS($network$)


Total error on training set
14
12
10
8
6
4
2
0
0
50
100
150
200
250
300
350
400
Number of epochs


Proportion correct on test set
1
0.9
0.8
0.7
0.6
0.5
0.4
0
10
20
30
40
50
60
70
80
90
100
Training set size
Decision tree
Multilayer network

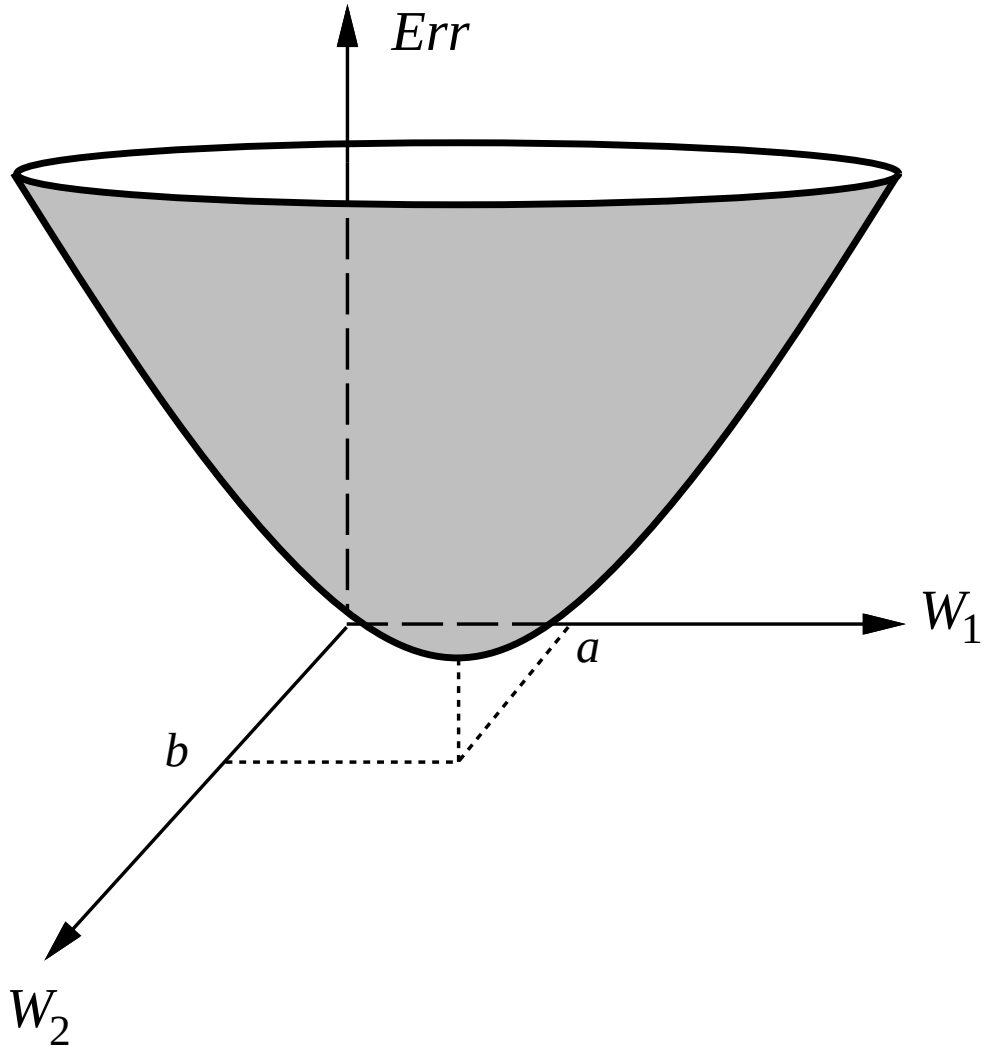
Back propagation algorithm:

1. calculate Δ_i for the output units based on the observed error $error_i$.
2. for each layer proceed recursively (output layer first):
   - back propagate the Δ_i (predecessor layer)
   - modify the weight between the current layers

Important:

Back propagation is gradient search!

error is the function of the network's weights.
This function delivers the error surface.



General remarks:

expressibility: neural networks are suitable for continous input and outputs (noise).

To represent all boolean functions with n attributes $\frac{2^n}{n}$ hidden units suffice.

Often much less suffice: the art of determining the topology of the network.

efficiency: m examples, $|W|$ weights: each epoch needs $\mathbf{O}(m \times |W|)$-time. We know:
Number of epochs is exponential.
In practice the time of convergence is very variable.
Problem: local minima on the error surface.

transparency: **black box**. Trees and lists explain their results!

5. Knowledge Engineering (1)

Content of this chapter (1):

Logic: We introduce sentential logic (also called propositional logic). This logic dates back to Boole and is the basis for many logical frameworks. The essential features of most logics can be illustrated in a puristic way.
We are using logics to describe the world and how the world behaves.

Sudoku: We illustrate how to use SL with the game of Sudoku. The conditions of being a solution can be easily stated in SL.

Content of this chapter (2):

Calculi for SL: While it is nice to describe the world, or the solution of a Sudoku puzzle, we also want to draw conclusions about it or even to solve the puzzle. Therefore we have to derive new information and deduce statements, that are not explicitly given.

Examples: We illustrate the use of SL with two more examples: The Wumpus world and one of the weekly puzzles in the newspaper *Die Zeit*.

5.1 Sentential Logic

Definition 5.1 (Sentential Logic $\mathcal{L}_{SL}$, Lang. $\mathcal{L} \subseteq \mathcal{L}_{SL}$)

The language $\mathcal{L}_{SL}$ of propositional (or sentential) logic consists of

- $\Box$ and $\top$: the constants *falsum* and *verum*,
- $p, q, r, x_1, x_2, \ldots x_n, \ldots$: a countable set $\mathcal{AT}$ of SL-constants,
- $\neg$, $\wedge$, $\vee$, $\rightarrow$: the sentential connectives ($\neg$ is unary, all others are binary operators),
- $($, $)$: the parentheses to help readability.

In most cases we consider only a finite set of SL-constants. They define a language $\mathcal{L} \subseteq \mathcal{L}_{SL}$. The set of $\mathcal{L}$-formulae $Fml_{\mathcal{L}}$ is defined inductively.

Definition 5.2 (Semantics, Valuation, Model)

A valuation v for a language $\mathcal{L} \subseteq \mathcal{L}_{SL}$ is a mapping from the set of SL-constants defined by $\mathcal{L}$ into the set $\{\text{true}, \text{false}\}$ with $v(\square) = \text{false}$, $v(\top) = \text{true}$.
Each valuation v can be uniquely extended to a function $\bar{v} : Fml_{\mathcal{L}} \to \{\text{true}, \text{false}\}$ so that:

- $\bar{v}(\neg p) = \begin{cases} \text{true}, & \text{if } \bar{v}(p) = \text{false}, \\ \text{false}, & \text{if } \bar{v}(p) = \text{true}. \end{cases}$
- $\bar{v}(\varphi \wedge \gamma) = \begin{cases} \text{true}, & \text{if } \bar{v}(\varphi) = \text{true and } \bar{v}(\gamma) = \text{true}, \\ \text{false}, & \text{else} \end{cases}$
- $\bar{v}(\varphi \vee \gamma) = \begin{cases} \text{true}, & \text{if } \bar{v}(\varphi) = \text{true or } \bar{v}(\gamma) = \text{true}, \\ \text{false}, & \text{else} \end{cases}$

Definition (continued)

- $\bar{v}(\varphi \to \gamma) =$
$$\begin{cases} \text{true}, & \text{if } \bar{v}(\varphi) = \text{false or } (\bar{v}(\varphi) = \text{true and } \bar{v}(\gamma) = \text{true}), \\ \text{false}, & else \end{cases}$$

Thus each valuation v uniquely defines a $\bar{v}$. We call $\bar{v}$ $\mathcal{L}$-structure.
A structure determines for each formula if it is true or false. If a formula ϕ is true in structure $\bar{v}$ we also say $\mathcal{A}_v$ is a model of ϕ. From now on we will speak of models, structures and valuations synonymously.

Semantics

The process of mapping a set of $\mathcal{L}$-formulae into $\{\text{true}, \text{false}\}$ is called semantics.

Definition 5.3 (Model, Theory, Tautology (Valid))

1. A formula $\varphi \in Fml_{\mathcal{L}}$ **holds under the valuation** v if $\bar{v}(\varphi) = \text{true}$. We also write $\bar{v} \models \varphi$ or simply $v \models \varphi$. $\bar{v}$ **is a model of** φ.
2. A **theory** is a set of formulae: $T \subseteq Fml_{\mathcal{L}}$. v satisfies T if $\bar{v}(\varphi) = \text{true}$ for all $\varphi \in T$. We write $v \models T$.
3. A $\mathcal{L}$-formula φ is called **$\mathcal{L}$-tautology** (or simply called **valid**) if **for all possible valuations v in $\mathcal{L}$** $v \models \varphi$ **holds**.

From now on we suppress the language $\mathcal{L}$ when obvious from context.

Definition 5.4 (Consequence Set $Cn(T)$)

A formula φ follows from T if for all models v of T (i.e. $v \models T$) also $v \models \varphi$ holds. We write: $T \models \varphi$.
We call

$$Cn_{\mathcal{L}}(T) =_{def} \{\varphi \in Fml_{\mathcal{L}} : T \models \varphi\},$$

or simply $Cn(T)$, the semantic consequence operator.

Lemma 5.5 (Properties of $Cn(T)$)

The semantic consequence operator has the following properties:

1. T-expansion: $T \subseteq Cn(T)$,
2. Monotony: $T \subseteq T' \Rightarrow Cn(T) \subseteq Cn(T')$,
3. Closure: $Cn(Cn(T)) = Cn(T)$.

Lemma 5.6 ($\varphi \notin$ Cn(T))

$\varphi \notin Cn(T)$ *if and only if there is a model* v *with* $v \models T$ *and* $\bar{v}(\varphi) =$ *false.*

Definition 5.7 (MOD(T), $Cn(\mathcal{U})$)

If $T \subseteq Fml_{\mathcal{L}}$ then we denote with $\mathbf{MOD}(T)$ the set of all $\mathcal{L}$-structures $\mathcal{A}$ which are models of T:

$$\mathrm{MOD}(T) =_{def} \{\mathcal{A} : \mathcal{A} \models T\}.$$

If $\mathcal{U}$ is a set of models, we consider all those sentences, which are valid in all models of $\mathcal{U}$. We call this set $Cn(\mathcal{U})$:

$$Cn(\mathcal{U}) =_{def} \{\varphi \in Fml_{\mathcal{L}} : \forall v \in \mathcal{U} : \bar{v}(\varphi) = \mathrm{true}\}.$$

MOD is obviously dual to Cn:

$$Cn(\mathrm{MOD}(T)) = Cn(T), \ \mathrm{MOD}(Cn(T)) = \mathrm{MOD}(T).$$

Definition 5.8 (Completeness of a Theory T)

T is called complete if for each formula $\varphi \in Fml$: $T \models \varphi$ or $T \models \neg\varphi$ holds.

Attention:

Do not mix up this last condition with the property of a valuation (model) v: **each model is complete in the above sense**.

Definition 5.9 (Consistency of a Theory)

T is called consistent if there is a valuation (model) v with $\bar{v}(\varphi) = \text{true}$ for all $\varphi \in T$.

Lemma 5.10 (Ex Falso Quodlibet)

T is consistent if and only if $Cn(T) \neq Fml_{\mathcal{L}}$.

5.2 Sudoku

Since some time, Sudoku puzzles are becoming quite famous.

			1	5		4	6	7
5	2	6		4				
4			9					
		4			5		2	6
6	9				7		1	
					1	8	3	9
	8							
	4	3	5	7	8	6		
			3				4	

Table 5.42: A simple Sudoku (S_1)

Can they be solved with sentential logic?

Idea: Given a Sudoku-Puzzle S, construct a language $\mathcal{L}_{\text{Sudoku}}$ and a theory $T_S \subseteq Fml_{\mathcal{L}_{\text{Sudoku}}}$ such that

$$\text{MOD}(T_S) = \text{ Solutions of the puzzle S}$$

Solution

In fact, we construct a theory T_{Sudoku} and for each (partial) instance of a 9×9 puzzle S a particular theory T_S such that

$$\text{MOD}(T_{\text{Sudoku}} \cup T_S) = \{\mathcal{S} : \mathcal{S} \text{ is a solution of S}\}$$

We introduce the following language $\mathcal{L}_{\text{Sudoku}}$:

1. $\text{eins}_{i,j}$, $1 \leq i, j \leq 9$,
2. $\text{zwei}_{i,j}$, $1 \leq i, j \leq 9$,
3. $\text{drei}_{i,j}$, $1 \leq i, j \leq 9$,
4. $\text{vier}_{i,j}$, $1 \leq i, j \leq 9$,
5. $\text{fuenf}_{i,j}$, $1 \leq i, j \leq 9$,
6. $\text{sechs}_{i,j}$, $1 \leq i, j \leq 9$,
7. $\text{sieben}_{i,j}$, $1 \leq i, j \leq 9$,
8. $\text{acht}_{i,j}$, $1 \leq i, j \leq 9$,
9. $\text{neun}_{i,j}$, $1 \leq i, j \leq 9$.

This completes the language, the **syntax**.

How many symbols are these?

We distinguished between the puzzle S and a solution $\mathcal{S}$ of it.

What is a model (or valuation) in the sense of Definition 5.2?

			1	5		4	6	7
5	2	6		4				
4			9					
		4			5		2	6
6	9				7		1	
					1	8	3	9
	8							
	4	3	5	7	8	6		
			3				4	

Table 5.43: How to construct a model $\mathcal{S}$?

We have to give our symbols a meaning: **the semantics!**

$$\text{eins}_{i,j} \textbf{ means } i,j \text{ contains a } 1$$
$$\text{zwei}_{i,j} \textbf{ means } i,j \text{ contains a } 2$$
$$\vdots$$
$$\text{neun}_{i,j} \textbf{ means } i,j \text{ contains a } 9$$

To be precise: given a 9×9 square that is completely filled out, we define our valuation v as follows (for all $1 \leq i,j \leq 9$).

$$v(\text{eins}_{i,j}) = \begin{cases} \text{true, if } 1 \text{ is at position } (i,j), \\ \text{false, else .} \end{cases}$$

$$v(\text{zwei}_{i,j}) = \begin{cases} \text{true, if } 2 \text{ is at position } (i,j), \\ \text{false, else .} \end{cases}$$

$$v(\text{drei}_{i,j}) = \begin{cases} \text{true, if } 3 \text{ is at position } (i,j), \\ \text{false, else .} \end{cases}$$

$$v(\text{vier}_{i,j}) = \begin{cases} \text{true, if } 4 \text{ is at position } (i,j), \\ \text{false, else .} \end{cases}$$

etc.

$$v(\text{neun}_{i,j}) = \begin{cases} \text{true, if } 9 \text{ is at position } (i,j), \\ \text{false, else .} \end{cases}$$

Therefore any 9×9 square can be seen as a model or valuation with respect to the language $\mathcal{L}_{\text{Sudoku}}$.

How does T_S look like?

$$
\begin{aligned}
T_S = \{ \quad & \mathsf{eins}_{1,4}, \mathsf{eins}_{5,8}, \mathsf{eins}_{6,6}, \\
& \mathsf{zwei}_{2,2}, \mathsf{zwei}_{4,8}, \\
& \mathsf{drei}_{6,8}, \mathsf{drei}_{8,3}, \mathsf{drei}_{9,4}, \\
& \mathsf{vier}_{1,7}, \mathsf{vier}_{2,5}, \mathsf{vier}_{3,1}, \mathsf{vier}_{4,3}, \mathsf{vier}_{8,2}, \mathsf{vier}_{9,8}, \\
& \vdots \\
& \mathsf{neun}_{3,4}, \mathsf{neun}_{5,2}, \mathsf{neun}_{6,9}, \\
\} &
\end{aligned}
$$

How should the theory T_{Sudoku} look like (s.t. models of $T_{\text{Sudoku}} \cup T_S$ correspond to solutions of the puzzle)?

First square: T_1

1. $\text{eins}_{1,1} \vee \ldots \vee \text{eins}_{3,3}$
2. $\text{zwei}_{1,1} \vee \ldots \vee \text{zwei}_{3,3}$
3. $\text{drei}_{1,1} \vee \ldots \vee \text{drei}_{3,3}$
4. $\text{vier}_{1,1} \vee \ldots \vee \text{vier}_{3,3}$
5. $\text{fuenf}_{1,1} \vee \ldots \vee \text{fuenf}_{3,3}$
6. $\text{sechs}_{1,1} \vee \ldots \vee \text{sechs}_{3,3}$
7. $\text{sieben}_{1,1} \vee \ldots \vee \text{sieben}_{3,3}$
8. $\text{acht}_{1,1} \vee \ldots \vee \text{acht}_{3,3}$
9. $\text{neun}_{1,1} \vee \ldots \vee \text{neun}_{3,3}$

The formulae on the last slide are saying, that

1. The number 1 must appear somewhere in the first square.
2. The number 2 must appear somewhere in the first square.
3. The number 3 must appear somewhere in the first square.
4. etc

Does that mean, that each number $1, \ldots, 9$ occurs exactly once in the first square?

No! We have to say, that each number occurs only once:

T_1':

1. $\neg(\text{eins}_{i,j} \wedge \text{zwei}_{i,j})$, $1 \leq i, j \leq 3$,
2. $\neg(\text{eins}_{i,j} \wedge \text{drei}_{i,j})$, $1 \leq i, j \leq 3$,
3. $\neg(\text{eins}_{i,j} \wedge \text{vier}_{i,j})$, $1 \leq i, j \leq 3$,
4. etc
5. $\neg(\text{zwei}_{i,j} \wedge \text{drei}_{i,j})$, $1 \leq i, j \leq 3$,
6. $\neg(\text{zwei}_{i,j} \wedge \text{vier}_{i,j})$, $1 \leq i, j \leq 3$,
7. $\neg(\text{zwei}_{i,j} \wedge \text{fuenf}_{i,j})$, $1 \leq i, j \leq 3$,
8. etc

How many formulae are these?

Second square: T_2

1. $\text{eins}_{1,4} \vee \ldots \vee \text{eins}_{3,6}$
2. $\text{zwei}_{1,4} \vee \ldots \vee \text{zwei}_{3,6}$
3. $\text{drei}_{1,4} \vee \ldots \vee \text{drei}_{3,6}$
4. $\text{vier}_{1,4} \vee \ldots \vee \text{vier}_{3,6}$
5. $\text{fuenf}_{1,4} \vee \ldots \vee \text{fuenf}_{3,6}$
6. $\text{sechs}_{1,4} \vee \ldots \vee \text{sechs}_{3,6}$
7. $\text{sieben}_{1,4} \vee \ldots \vee \text{sieben}_{3,6}$
8. $\text{acht}_{1,4} \vee \ldots \vee \text{acht}_{3,6}$
9. $\text{neun}_{1,4} \vee \ldots \vee \text{neun}_{3,6}$

And all the other formulae from the previous slides (adapted to this case): T_2'

The same has to be done for all 9 squares.

What is still missing:

Rows: Each row should contain exactly the numbers from 1 to 9 (no number twice).

Columns: Each column should contain exactly the numbers from 1 to 9 (no number twice).

First Row: $T_{\text{Row 1}}$

1. $\text{eins}_{1,1} \vee \text{eins}_{1,2} \vee \ldots \vee \text{eins}_{1,9}$
2. $\text{zwei}_{1,1} \vee \text{zwei}_{1,2} \vee \ldots \vee \text{zwei}_{1,9}$
3. $\text{drei}_{1,1} \vee \text{drei}_{1,2} \vee \ldots \vee \text{drei}_{1,9}$
4. $\text{vier}_{1,1} \vee \text{vier}_{1,2} \vee \ldots \vee \text{vier}_{1,9}$
5. $\text{fuenf}_{1,1} \vee \text{fuenf}_{1,2} \vee \ldots \vee \text{fuenf}_{1,9}$
6. $\text{sechs}_{1,1} \vee \text{sechs}_{1,2} \vee \ldots \vee \text{sechs}_{1,9}$
7. $\text{sieben}_{1,1} \vee \text{sieben}_{1,2} \vee \ldots \vee \text{sieben}_{1,9}$
8. $\text{acht}_{1,1} \vee \text{acht}_{1,2} \vee \ldots \vee \text{acht}_{1,9}$
9. $\text{neun}_{1,1} \vee \text{neun}_{1,2} \vee \ldots \vee \text{neun}_{1,9}$

Analogously for all other rows, eg.

Ninth Row: $T_{\text{Row 9}}$

1. $\text{eins}_{9,1} \vee \text{eins}_{9,2} \vee \ldots \vee \text{eins}_{9,9}$
2. $\text{zwei}_{9,1} \vee \text{zwei}_{9,2} \vee \ldots \vee \text{zwei}_{9,9}$
3. $\text{drei}_{9,1} \vee \text{drei}_{9,2} \vee \ldots \vee \text{drei}_{9,9}$
4. $\text{vier}_{9,1} \vee \text{vier}_{9,2} \vee \ldots \vee \text{vier}_{9,9}$
5. $\text{fuenf}_{9,1} \vee \text{fuenf}_{9,2} \vee \ldots \vee \text{fuenf}_{9,9}$
6. $\text{sechs}_{9,1} \vee \text{sechs}_{9,2} \vee \ldots \vee \text{sechs}_{9,9}$
7. $\text{sieben}_{9,1} \vee \text{sieben}_{9,2} \vee \ldots \vee \text{sieben}_{9,9}$
8. $\text{acht}_{9,1} \vee \text{acht}_{9,2} \vee \ldots \vee \text{acht}_{9,9}$
9. $\text{neun}_{9,1} \vee \text{neun}_{9,2} \vee \ldots \vee \text{neun}_{9,9}$

Is that sufficient? What if a row contains several 1's?

First Column: $T_{\text{Column 1}}$

1. $\text{eins}_{1,1} \vee \text{eins}_{2,1} \vee \ldots \vee \text{eins}_{9,1}$
2. $\text{zwei}_{1,1} \vee \text{zwei}_{2,1} \vee \ldots \vee \text{zwei}_{9,1}$
3. $\text{drei}_{1,1} \vee \text{drei}_{2,1} \vee \ldots \vee \text{drei}_{9,1}$
4. $\text{vier}_{1,1} \vee \text{vier}_{2,1} \vee \ldots \vee \text{vier}_{9,1}$
5. $\text{fuenf}_{1,1} \vee \text{fuenf}_{2,1} \vee \ldots \vee \text{fuenf}_{9,1}$
6. $\text{sechs}_{1,1} \vee \text{sechs}_{2,1} \vee \ldots \vee \text{sechs}_{9,1}$
7. $\text{sieben}_{1,1} \vee \text{sieben}_{2,1} \vee \ldots \vee \text{sieben}_{9,1}$
8. $\text{acht}_{1,1} \vee \text{acht}_{2,1} \vee \ldots \vee \text{acht}_{9,1}$
9. $\text{neun}_{1,1} \vee \text{neun}_{2,1} \vee \ldots \vee \text{neun}_{9,1}$

Analogously for all other columns.

Is that sufficient? What if a column contains several 1's?

All put together:

$$\begin{aligned} T_{\text{Sudoku}} = \; & T_1 \cup T_1' \cup \ldots \cup T_9 \cup T_9' \\ & T_{\text{Row 1}} \cup \ldots \cup T_{\text{Row 9}} \\ & T_{\text{Column 1}} \cup \ldots \cup T_{\text{Column 9}} \end{aligned}$$

Here is a more difficult one.

2		3		9				6
	1							4
	9				5			
		7						
6			9				7	
8	4							
9	3		1		7	5		
7		2			9	4		
1			2				6	

Table 5.44: A difficult Sudoku $S_{\text{difficult}}$

- The above formulation is strictly formulated in propositional logic.
- Theorem provers, even if they consider only propositional theories, often use **predicates**, **variables** etc.
- smodels uses a predicate logic formulation, including variables. But as there are no function symbols, such an input can be seen as a compact representation.
- It allows to use a few rules as a shorthand for thousands of rules using propositional constants.

For example smodels uses the following constructs:

1. `row(0..8)` is a shorthand for `row(0), row(1), ..., row(8).`
2. `val(1..9)` is a shorthand for `val(1), val(2), ..., val(9).`
3. The constants `1, ..., 9` will be treated as numbers (so there are operations available to add, subtract or divide them).

1. Statements in smodels are written as

   $$\texttt{head :- body}$$

   This corresponds to an implication $body \rightarrow head$.
2. An important feature in smodels is that all atoms that do not occur in any head, are automatically false.
   For example the theory

   $$\texttt{p(X, Y, 5) :- row(X), col(Y)}$$

   means that the whole grid is filled with 5's and only with 5's: eg. $\neg p(X, Y, 1)$ is true for all X, Y, as well as $\neg p(X, Y, 2)$ etc.

More constructs in smodels

1. ```
   1 { p(X,Y,A) : val(A) } 1
             :- row(X), col(Y)
   ```
   this makes sure that in all entries of the grid, exactly one number (`val()`) is contained.
2. ```
   1 { p(X,Y,A) : row(X) : col(Y)
                : eq(div(X,3),div(R,3))
                : eq(div(Y,3),div(C,3)) } 1
             :- val(A), row(R), col(C)
   ```
   this rule ensures that in each of the 9 squares each number from 1 to 9 occurs only once.
3. More detailed info on the web-page.

5.3 Calculi for SL

A general notion of a certain sort of calculi.

Definition 5.11 (Hilbert-Type Calculi)

A Hilbert-Type calculus over a language $\mathcal{L}$ is a pair $\langle \mathrm{Ax}, \mathrm{Inf} \rangle$ where

Ax: is a subset of $Fml_{\mathcal{L}}$, the set of well-formed formulae in $\mathcal{L}$: they are called axioms,

Inf: is a set of pairs written in the form

$$\frac{\phi_1, \phi_2, \ldots, \phi_n}{\psi}$$

where $\phi_1, \phi_2, \ldots, \phi_n, \psi$ are $\mathcal{L}$-formulae: they are called inference rules.

Intuitively, one can assume all axioms as "true formulae" (**tautologies**) and then use the inference rules to derive even more new formulae.

Definition 5.12 (Calculus for Sentential Logic SL)

We define $\text{Hilbert}_{\mathcal{L}}^{SL} = \langle Ax_{\mathcal{L}}^{SL}, \{MP\}\rangle$, the Hilbert-Type calculus: $\mathcal{L} \subseteq \mathcal{L}_{SL}$ with the wellformed formulae $Fml_{\mathcal{L}}$ as defined in Definition 5.1.
Axioms in SL ($Ax_{\mathcal{L}}^{SL}$) are the following formulae:

1. $\phi \to \top$, $\Box \to \phi$, $\neg\top \to \Box$, $\Box \to \neg\top$,
2. $(\phi \to \psi) \to ((\phi \to (\psi \to \chi)) \to (\phi \to \chi))$,
3. $(\phi \wedge \psi) \to \phi$, $(\phi \wedge \psi) \to \psi$,
4. $\phi \to (\phi \vee \psi)$, $\psi \to (\phi \vee \psi)$,
5. $\neg\neg\phi \to \phi$, $(\phi \to \psi) \to ((\phi \to \neg\psi) \to \neg\phi)$,
6. $\phi \to (\psi \to \phi)$, $\phi \to (\psi \to (\phi \wedge \psi))$.
7. $(\phi \to \chi) \to ((\psi \to \chi) \to (\phi \vee \psi \to \chi))$.

ϕ, ψ, χ stand for arbitrarily complex formulae (not just constants). They represent schemata, rather than formulae in the language.

Definition (continued)

The only inference rule in SL is **modus ponens**:

$$MP: \ Fml \times Fml \to Fml: \ (\varphi, \varphi \to \psi) \mapsto \psi.$$

or short

$$\textbf{(MP)} \ \frac{\varphi, \ \varphi \to \psi}{\psi}.$$

(φ, ψ are arbitrarily complex formulae).

Definition 5.13 (Proof)

A **proof** of a formula φ from a theory $T \subseteq Fml_{\mathcal{L}}$ is a **sequence** $\varphi_1, \ldots, \varphi_n$ of formulae such that $\varphi_n = \varphi$ and for all i with $1 \leq i \leq n$ one of the following conditions holds:

- φ_i is substitution instance of an axiom,
- $\varphi_i \in T$,
- there is $\varphi_l, \varphi_k = (\varphi_l \to \varphi_i)$ with $l, k < i$. Then φ_i is the result of the application of modus ponens on the predecessor-formulae of φ_i.

We write: $T \vdash \varphi$ (φ can be **derived** from T).

We show that:

1. $\vdash \top$ and $\vdash \neg\Box$.
2. $A \vdash A \vee B$ and $\vdash A \vee \neg A$.
3. The rule

$$\textbf{(R)}\ \frac{A \to \varphi,\ \neg A \to \psi}{\varphi \vee \psi}$$

   can be derived.
4. Our version of sentential logic does not contain a connective "$\leftrightarrow$". We define "$\phi \leftrightarrow \psi$" as a macro for "$\phi \to \psi\ \wedge\ \psi \to \phi$". Show the following:

$$\text{If } \vdash \phi \leftrightarrow \psi, \text{ then } \vdash \phi \text{ if and only if } \vdash \psi.$$

We have now introduced two important notions:

Syntactic derivability $\vdash$: the notion that certain formulae can be derived from other formulae using a certain calculus,

Semantic validity $\models$: the notion that certain formulae follow from other formulae based on the semantic notion of a model.

Definition 5.14 (Correct-, Completeness for a calculus)

Given an arbitrary **calculus** (which defines a notion $\vdash$) and a **semantics** based on certain models (which defines a relation $\models$), we say that

Correctness: The calculus is **correct** with respect to the semantics, if the following holds:

$$\Phi \vdash \phi \text{ implies } \Phi \models \phi.$$

Completeness: The calculus is **complete** with respect to the semantics, if the following holds:

$$\Phi \models \phi \text{ implies } \Phi \vdash \phi.$$

Theorem 5.15 (Correct-, Completeness for $\mathrm{Hilbert}_{\mathcal{L}}^{SL}$)

A formula follows semantically from a theory T if and only if it can be derived:

$$T \models \varphi \text{ if and only if } T \vdash \varphi$$

Theorem 5.16 (Compactness for $\mathrm{Hilbert}_{\mathcal{L}}^{SL}$)

A formula follows from a theory T *if and only if it* follows from a finite subset of T*:*

$$Cn(T) = \bigcup \{Cn(T') : T' \subseteq T, T' \textit{finite}\}.$$

Although the axioms from above and modus ponens suffice it is reasonable to consider more general systems. Therefore we introduce the notion of a rule system.

Definition 5.17 (Rule System MP + D)

Let D be a set of general inference rules, e.g. mappings, which assign a formula ψ to a finite set of formulae $\varphi_1, \varphi_2, \ldots, \varphi_n$. We write

$$\frac{\varphi_1, \varphi_2, \ldots, \varphi_n}{\psi}.$$

MP + D is the rule system which emerges from adding the rules in D to modus ponens. For $W \subseteq Fml$ let

$$Cn^D(W)$$

be the set of all formulae φ, which can be derived from W and the inference rules from MP+D.

We bear in mind that $Cn(W)$ is defined semantically but $Cn^{\mathrm{D}}(W)$ is defined syntactically (using the notion of proof). Both sets are equal according to the completeness-theorem in the special case $\mathrm{D} = \emptyset$.

Lemma 5.18 (Properties of Cn^{D})

Let D *be a set of general inference rules and* $W \subseteq Fml$. *Then:*

1. $Cn(W) \subseteq Cn^{\mathrm{D}}(W)$.
2. $Cn^{\mathrm{D}}(Cn^{\mathrm{D}}(W)) = Cn^{\mathrm{D}}(W)$.
3. $Cn^{\mathrm{D}}(W)$ *is the smallest set which is closed in respect to* D *and contains* W.

Question:

What is the difference between an inference rule $\frac{\varphi}{\psi}$ and the implication $\varphi \to \psi$?

Assume we have a set T of formulae and we choose two constants $p, q \in \mathcal{L}$. We can either consider

(1) T together with MP and $\{\frac{p}{q}\}$

or

(2) $T \cup \{p \to q\}$ together with MP

1. Case:

$$\mathbf{Cn}^{\{\frac{p}{q}\}}(T),$$

2. Case:

$$\mathbf{Cn}(T \cup \{p \to q\}).$$

If $T = \{\neg q\}$, then we have in (2):
$\neg p \in \mathbf{Cn}(T \cup \{p \to q\})$, but not in (1).

It is well-known, that any formula ϕ can be written as a **conjunction of disjunctions**

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} \phi_{i,j}$$

The $\phi_{i,j}$ are just constants or negated constants.
The n disjunctions $\bigvee_{j=1}^{m_i} \phi_{i,j}$ are called clauses of ϕ.

Normalform

Instead of working on arbitrary formulae, it is sometimes easier to work on finite sets of clauses.

A resolution calculus for SL

The resolution calculus is defined over the language $\mathcal{L}^{res} \subseteq \mathcal{L}_{SL}$ where the set of well-formed formulae $Fml^{Res}_{\mathcal{L}^{res}}$ consists of all disjunctions of the following form

$$A \vee \neg B \vee C \vee \ldots \vee \neg E,$$

i.e. the disjuncts are only constants or their negations. **No implications or conjunctions are allowed**. These formulae are also called clauses.

- $\square$ is also a clause: the **empty disjunction**.

Set-notation of clauses

A disjunction $A \vee \neg B \vee C \vee \ldots \vee \neg E$ is often written as a set

$$\{A, \neg B, C, \ldots, \neg E\}.$$

Thus the set-theoretic union of such sets corresponds again to a clause: $\{A, \neg B\} \cup \{A, \neg C\}$ represents $A \vee \neg B \vee \neg C$. Note that the empty set $\emptyset$ is identified with $\square$.

We define the following inference rule on $Fml^{Res}_{\mathcal{L}^{res}}$:

Definition 5.19 (SL resolution)

Let C_1, C_2 be clauses (disjunctions). Deduce the clause $C_1 \vee C_2$ from $C_1 \vee A$ and $C_2 \vee \neg A$:

$$\textbf{(Res)}\ \frac{C_1 \vee A,\ C_2 \vee \neg A}{C_1 \vee C_2}$$

If $C_1 = C_2 = \emptyset$, then $C_1 \vee C_2 = \square$.

If we use the set-notation for clauses, we can formulate the inference rule as follows:

Definition 5.20 (SL resolution (Set notation))

Deduce the clause $C_1 \cup C_2$ from $C_1 \cup \{A\}$ and $C_2 \cup \{\neg A\}$:

$$\textbf{(Res)}\ \frac{C_1 \cup \{A\},\ C_2 \cup \{\neg A\}}{C_1 \cup C_2}$$

Again, we identify the empty set $\emptyset$ with $\square$.

Definition 5.21 (Resolution Calculus for SL)

We define the **resolution calculus** $\mathrm{Robinson}^{SL}_{\mathcal{L}^{res}} = \langle \emptyset, \{\mathbf{Res}\} \rangle$ as follows. The underlying language is $\mathcal{L}^{res} \subseteq \mathcal{L}_{SL}$ defined on Slide 402 together with the well-formed formulae $Fml^{Res}_{\mathcal{L}^{res}}$.

Thus there are no axioms and only one inference rule. The well-formed formulae are just clauses.

Question:

Is this calculus correct and complete?

Answer:

It is correct, but not complete!

But every problem of the kind "$T \models \phi$" is equivalent to

"$T \cup \{\neg\phi\}$ is unsatisfiable"

or rather to

$$T \cup \{\neg\phi\} \vdash \square$$

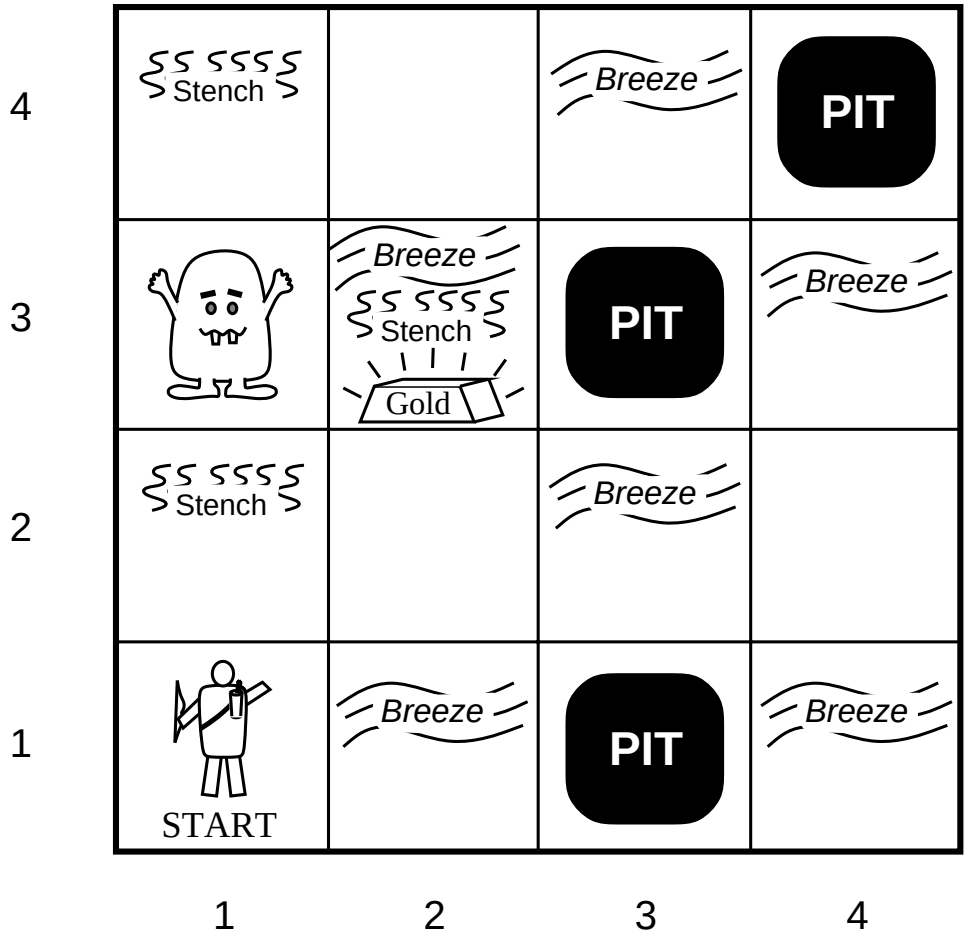
($\vdash$ stands for the calculus introduced above).

Theorem 5.22 (Completeness of Resolution Refutation)

If M is an unsatisfiable set of clauses then the empty clause $\square$ can be derived in $\text{Robinson}^{SL}_{\mathcal{L}^{res}}$.

We also say that resolution is refutation complete.

5.4 Wumpus in SL


4
3
2
1
Stench
Breeze
PIT
Breeze
Stench
Gold
PIT
Breeze
Stench
Breeze
Breeze
START
Breeze
PIT
Breeze
1
2
3
4

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 **OK**	2,2	3,2	4,2
1,1 **A** **OK**	2,1 **OK**	3,1	4,1

(a)

A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 **OK**	2,2 **P?**	3,2	4,2
1,1 **V** **OK**	2,1 **A** **B** **OK**	3,1 **P?**	4,1

(b)

1,4	2,4	3,4	4,4
1,3 **W!**	2,3	3,3	4,3
1,2 **A** **S** **OK**	2,2 **OK**	3,2	4,2
1,1 **V** **OK**	2,1 **B** **V** **OK**	3,1 **P!**	4,1

(a)

A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*

1,4	2,4 **P?**	3,4	4,4
1,3 **W!**	2,3 **A** **S G** **B**	3,3 **P?**	4,3
1,2 **S** **V** **OK**	2,2 **V** **OK**	3,2	4,2
1,1 **V** **OK**	2,1 **B** **V** **OK**	3,1 **P!**	4,1

(b)

Language definition:

$S_{i,j}$	stench
$B_{i,j}$	breeze
$Pit_{i,j}$	is a pit
$Gl_{i,j}$	glitters
$W_{i,j}$	contains Wumpus

General knowledge:

$$\neg S_{1,1} \longrightarrow (\neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1})$$
$$\neg S_{2,1} \longrightarrow (\neg W_{1,1} \wedge \neg W_{2,1} \wedge \neg W_{2,2} \wedge \neg W_{3,1})$$
$$\neg S_{1,2} \longrightarrow (\neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,2} \wedge \neg W_{1,3})$$

$$S_{1,2} \longrightarrow (W_{1,3} \vee W_{1,2} \vee W_{2,2} \vee W_{1,1})$$

Knowledge after the 3rd move:

$$\neg S_{1,1} \wedge \neg S_{2,1} \wedge S_{1,2} \wedge \neg B_{1,1} \wedge B_{2,1} \wedge \neg B_{1,2}$$

Question:

Can we deduce that the wumpus is located at (1,3)?

Answer:

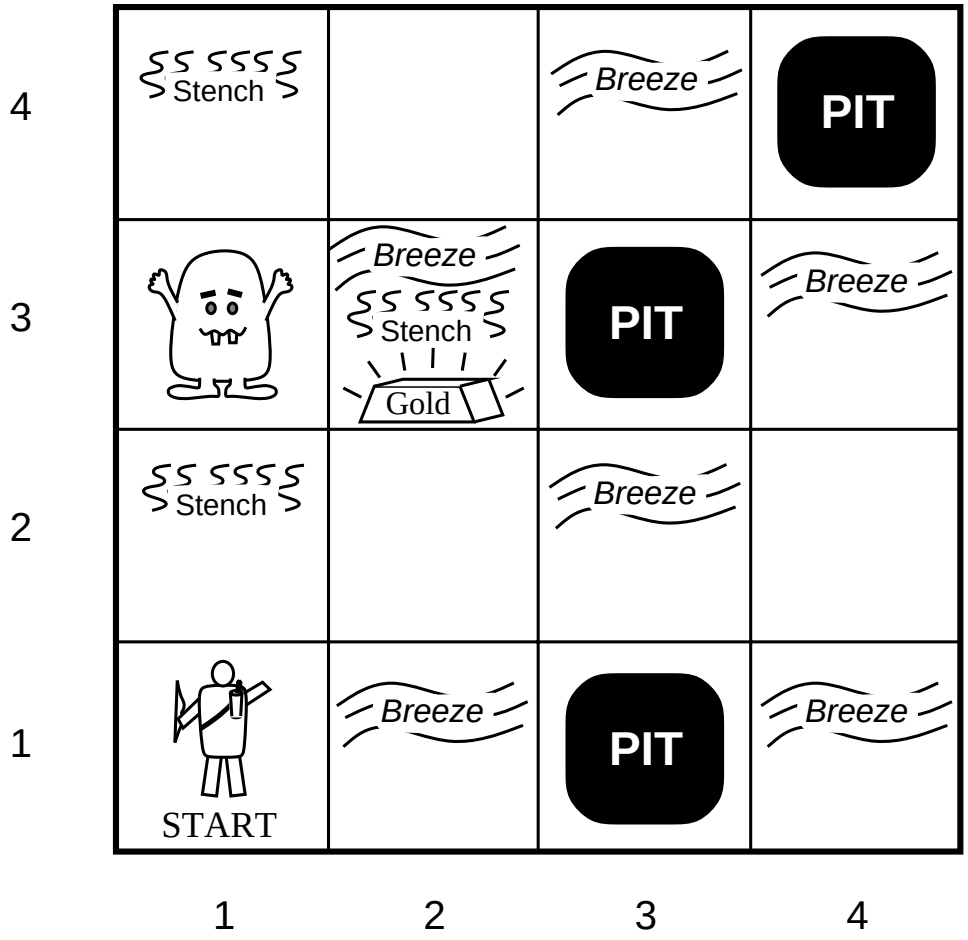
Yes. Either via resolution or using our Hilbert-calculus.

Problem:

We want more: given a certain situation we would like to determine the **best** action, i.e. to ask a query which gives us back such an action. This is impossible in SL: we can only check for each action whether it is good or not and then, by comparison, try to find the best action.

But we can check for each action if it should be done or not. Therefore we need additional axioms:

$$\begin{aligned} A_{1,1} \wedge East \wedge W_{2,1} &\longrightarrow \neg Forward \\ A_{1,1} \wedge East \wedge Pit_{2,1} &\longrightarrow \neg Forward \\ A_{i,j} \wedge Gl_{i,j} &\longrightarrow Take_{Gold} \end{aligned}$$


4
3
2
1
Stench
Breeze
PIT
Breeze
Stench
Gold
PIT
Breeze
Stench
Breeze
Breeze
PIT
Breeze
START
1
2
3
4

Disadvantages

- actions can only be guessed
- database must be changed continuously
- the set of rules becomes very big because there are no variables

Using an appropriate formalisation (additional axioms) we can check if

$$KB \vdash \neg \text{action} \quad \text{or } KB \vdash \text{action}$$

But it can happen that neither one nor the other is deducible.

5.5 A Puzzle

We now want to formalize a ”Logelei“ and solve it with a **theorem prover**.

”Logelei“ from ”Die Zeit“ (1)

Alfred ist als neuer Korrespondent in Wongowongo. Er soll über die Präsidentschaftswahlen berichten, weiß aber noch nichts über die beiden Kandidaten, weswegen er sich unter die Leute begibt, um Infos zu sammeln. Er befragt eine Gruppe von Passanten, von denen drei Anhänger der Entweder-oder-Partei sind und drei Anhänger der Konsequenten.

”Logelei“ from ”Die Zeit“ (2)

Auf seinem Notizzettel notiert er stichwortartig die Antworten.
A: »Nachname Songo: Stadt Rongo«,
B: »Entweder-oder-Partei: älter«,
C: »Vorname Dongo: bei Umfrage hinten«,
A: »Konsequenten: Vorname Mongo«,
B: »Stamm Bongo: Nachname Gongo«,
C: »Vorname Dongo: jünger«,
D: »Stamm Bongo: bei Umfrage vorn«,
E: »Vorname Mongo: bei Umfrage hinten«,
F: »Konsequenten: Stamm Nongo«,
D: »Stadt Longo: jünger«,
E: »Stamm Nongo: jünger«.
F: »Konsequenten: Nachname Gongo«.

”Logelei“ from ”Die Zeit“ (3)

Jetzt grübelt Alfred. Er weiß, dass die Anhänger der Entweder-oder-Partei (A, B und C) immer eine richtige und eine falsche Aussage machen, während die Anhänger der Konsequenten (D, E und F) entweder nur wahre Aussagen oder nur falsche Aussagen machen.
Welche Informationen hat Alfred über die beiden Kandidaten?

(By Zweistein)

Towards a solution

- Selection of the language (Propositional Logic, Predicate Logic,...).
- Analysis and formalization of the problem.
- Transformation to the input format of a prover.
- Output of a solution, i.e. a **model**.

Definition of the constants

$Sur_{x,\text{Songo}} \equiv$	x's surname is Songo
$Sur_{x,\text{Gongo}} \equiv$	x's surname is Gongo
$First_{x,\text{Dongo}} \equiv$	x's first name is Dongo
$First_{x,\text{Mongo}} \equiv$	x's first name is Mongo
$Tribe_{x,\text{Bongo}} \equiv$	x belongs to the Bongos
$Tribe_{x,\text{Nongo}} \equiv$	x belongs to the Nongos
$City_{x,\text{Rongo}} \equiv$	x comes from Rongo
$City_{x,\text{Longo}} \equiv$	x comes from Longo
$A_x \equiv$	x is the senior candidate
$J_x \equiv$	x is the junior candidate
$H_x \equiv$	x's poll is worse
$V_x \equiv$	x's poll is better

Here x is a candidate, i.e. $x \in \{a, b\}$. So we have 24 constants in total.

The correspondent Alfred noted 12 statements about the candidates (each interviewee gave 2 statements, ϕ, ϕ') which we enumerate as follows

$$\phi_A, \phi'_A, \phi_B, \phi'_B, \ldots, \phi_F, \phi'_F,$$

All necessary symbols are now defined, and we can formalize the given statements.

Formalization of the statements

$$
\begin{aligned}
\phi_A &\leftrightarrow (Sur_{a,\text{Songo}} \wedge City_{a,\text{Rongo}}) \vee \\
&\quad\ (Sur_{b,\text{Songo}} \wedge City_{b,\text{Rongo}}) \\
\phi'_A &\leftrightarrow First_{b,\text{Mongo}} \\
\phi_B &\leftrightarrow A_a \\
\phi'_B &\leftrightarrow (Tribe_{a,\text{Bongo}} \wedge Sur_{a,\text{Gongo}}) \vee \\
&\quad\ (Tribe_{b,\text{Bongo}} \wedge Sur_{b,\text{Gongo}}) \\
&\ \ \vdots
\end{aligned}
$$

Furthermore, **explicit** conditions between the statements are given, e.g.

$$(\phi_A \wedge \neg\phi'_A) \vee (\neg\phi_A \wedge \phi'_A)$$

and

$$(\phi_D \wedge \phi'_D) \vee (\neg\phi_D \wedge \neg\phi'_D).$$

Analogously, for the other statements.

Is this enough information to solve the puzzle?

E.g., can the following formula be satisfied?

$$Sur_{a,\text{Songo}} \wedge Sur_{a,\text{Gongo}}$$

We also need **implicit** conditions (**axioms**) which are required to solve this problem.
It is necessary to state that each candidate has only one name, comes from one city, etc.

We need the following background knowledge...

$$Sur_{x,\text{Songo}} \leftrightarrow \neg Sur_{x,\text{Gongo}}$$
$$First_{x,\text{Dongo}} \leftrightarrow \neg First_{x,\text{Mongo}}$$
$$\vdots$$
$$H_x \leftrightarrow \neg V_x$$

Can we abstain from these axioms by changing our representation of the puzzle?

What is still missing?

Can we prove that when a's poll is worse, then s's poll is better?

We need to state the relationships between these attributes:

$$H_x \leftrightarrow V_y$$
$$A_x \leftrightarrow J_y$$

Finally, we have modeled all "sensible" information. Does this yield a unique model?

No! There are **6 models** in total, but this is all right. It just means there is no unique solution.

What if a unique model is desirable?

Often, there are additional assumptions hidden "between the lines". Think, for example, of deductions by Sherlock Holmes (or Miss Marple, Spock etc).

For example, it might be sensible to assume that both candidates come from **different** cities:

$$City_{x,\text{Rongo}} \leftrightarrow City_{y,\text{Longo}}$$

Indeed, with this additional axiom there is an unique model.

But, be careful...

... this additional information may not be justified by the nature of the task!

Tractatus Logico-Philosophicus

1. Die Welt ist alles was der Fall ist.
   - 1.1 Die Welt ist die Gesamtheit der Tatsachen, nicht der Dinge.
2. Was der Fall ist, die Tatsache, ist das Bestehen von Sachverhalten.
3. Das logische Bild der Tatsachen ist der Gedanke.
4. Der Satz ist eine Wahrheitsfunktion der Elementarsätze.
5. Die allgemeine Form der Wahrheitsfunktion ist: $[\overline{p}, \overline{\xi}, N(\overline{\xi})]$.
6. Wovon man nicht sprechen kann, darüber muß man schweigen.

6. Hoare Calculus

Content of this chapter (1):

We introduce a calculus to prove correctness of computer programs.

Verification: We argue why formal methods are useful/necessary for program verification.

Core Programming Language: An abstract but powerful programming language is introduced.

Hoare Logic: We introduce the Hoare Calculus which operates on triples $\{\phi_{pre}\}\ \mathsf{P}\ \{\psi_{post}\}$.

Content of this chapter (2):

Proof Calculi (Partial Correctness): Here we present three calculi (proof rules, annotation calculus, and the weakest precondition calculus) for partial correctness.

Proof Calculi (Total Correctness): Partial correctness is trivially satisfied when a program does not terminate. Therefore, we consider total correctness as well and an extension of the calculi to deal with it. We consider the partial and total correctness of programs.

Sound & Completeness: We briefly discuss the sound and completeness of the Hoare calculus.

6.1 Verification

Why to formally specify/verify code?

- Formal specifications are often important for **unambiguous** documentations.
- Formal methods cut down software development and maintenance costs.
- Formal specification makes software easier to reuse due to a **clear specification**.
- Formal verification can ensure **error-free** software required by safety-critical computer systems.

Verification of Programs

How can we define the state of a program?

The state of a program is given by the contents of all variables.

What about the **size** of the state-space of a program?

The state space is usually infinite! Hence, the technique of model checking is inappropriate.

Software Verification Framework

Main reasons for formal specifications:

- **Informal** descriptions often lead to **ambiguities** which can result in serious (and potentially expensive) design flaws.
- Without formal specifications a rigorous verification is not possible!

Methodology

We assume the following methodology:

1. Build an **informal description** D of the program and the domain.
2. Convert D into an equivalent formula ϕ in a suitable logic.
3. (Try to) build a program P **realizing** ϕ.
4. Prove that P satisfies ϕ.

Methodology (2)

Some points to think about:

(i) Point 2 is a **non-trivial** and **non-formal** problem and thus "cannot be proven": D is an **informal** specification!

(ii) Often there are alternations between 3 and 4.

(iii) Sometimes one might realize that ϕ is not equivalent to D and thus one has to revise ϕ.

(iv) Often, P must have a **specific structure** to prove it against φ.

In this lecture, we will focus on points 3 and 4.

Some Properties of the Procedure

Proof-based: Not every **state** of the system is considered (there are infinitely many anyway), rather a proof for correctness is constructed: this works then for **all** states.

Semi-automatic: Fully automatic systems are desireable but they are not always possible: undecidability, time constraints, efficiency, and “lack of intelligence”.

Some Properties of the Procedure (2)

Property-oriented: Only certain properties of a program are proven and not the "complete" behavior.

Application domain: Only sequential programs are considered.

Pre/post-development: The proof techniques are designed to be used **during the programming process** (development phase).

6.2 Core Programming Language

Core Programming Language

To deal with an up-to-date programming language like Java or C++ is out of scope of this introductory lecture. Instead we identify some core programming constructs and abstract away from other syntactic and language-specific variations.

Our programming language is built over

1. integer expressions,
2. boolean expressions, and
3. commands.

Definition 6.1 (Program Variables)

We use $\mathcal{V}ars$ to denote the set of (program) variables. Typically, variables will be denoted by $u, \ldots, x, y, z$ or $x_1, x_2, \ldots$.

Definition 6.2 (Integer Expression)

Let $x \in \mathcal{V}ars$. The set of integer expressions $\mathcal{I}$ is given by all terms generated according to the following grammar:

$$I ::= 0 \mid 1 \mid x \mid (I + I) \mid (I - I) \mid (I \cdot I)$$

We also use $-I$ to stand for $0 - I$.

Although the term "$1-1$" is different from "0", its **meaning (semantics)** is the same. We identify "$1+1+\ldots+1$" with "n" (if the term consist of n 1's). The precise notion of meaning will be given in Definition 6.10.

Integer expressions are **terms** over a set of function symbols $\mathcal{F}unc$, here

$$\mathcal{F}unc = \mathcal{V}ars \cup \{0, 1, +, -, \cdot\}$$

Note also, that we consider elements from $\mathbb{Z}$ as **constants** with their **canonical denotation**! Thus we write "3" instead of "$1+1+1$".

Example 6.3 (Some integer expressions)

$5,\ x,\ 6+(3\cdot x),\ x\cdot y+z-3, -x\cdot x,\ \ldots$

Note that x^x or $y!$ are not integer expressions. **Do we need them?**

Definition 6.4 (Boolean Expression)

The set of **Boolean expressions** $\mathcal{B}$ is given by all formulae generated according to the following grammar:

$$B ::= \top \mid (\neg B) \mid (B \wedge B) \mid (I < I)$$

We also use $(B \vee B)$ as an abbreviation of $\neg(\neg B \wedge \neg B)$ and $\Box$ as an abbreviation for $\neg\top$. Boolean expressions are **formulae** over the set of relation symbols $\mathcal{P}red = \{\top, <\}$. So, boolean expressions are similar to **sentential logic formulae**.

Note that we use $I = I'$ to stand for $\neg(I < I') \wedge \neg(I' < I)$. We also write $I \neq I'$ for $\neg(I = I')$.

Formulae over $\mathbb{Z}$

When describing the Hoare calculus, we also need formulae to express (1) what should hold **before** a program is executed, and (2) what should be the case **after** the program has been executed. These formulae are built using the boolean expressions just introduced.

Definition 6.5 (Formulae over $\mathbb{Z}$)

The set of **formulae** $\mathcal{F}ml_{\mathcal{V}ars}$ is given by all formulae generated according to the following grammar:

$$\phi ::= \exists x \phi \mid B \mid (\neg \phi) \mid (\phi \wedge \phi)$$

where B is a boolean expression according to Definition 6.4 and $x \in \mathcal{V}ars$.

We note that "$\forall x \phi$" is just an abbreviation for "$\neg \exists x \neg \Phi$"

Formulae over $\mathbb{Z}$ (2)

We have to give a meaning (semantics) to the syntactic constructs! Our model is given by $\mathbb{Z}$ where $\mathbb{Z}$ is the set of integer numbers. Note that in $\mathbb{Z}$ we have a natural meaning of the constructs $0, 1, -, +, \cdot, <$.

Which of the following formulae are true in $\mathbb{Z}$?

1. $\forall x \forall y \forall z\ (x \cdot x + y \cdot y + z \cdot z\ > 0)$,
2. $\forall x \exists y\ (y < x)$,
3. $x \cdot x\ <\ x \cdot x \cdot x$,
4. $x + 4 < x + 5$.

Formulae over $\mathbb{Z}$ (3)

Note that when evaluating a formula with free variables, there are two possibilities:

1. Either we need to assign values to these variables, or
2. we add "$\forall$" quantifiers to bind all variables.

The truth of a formula can only be established if the formula is a sentence, i.e. does not contain free variables. We will be more formal in Chapter 7.

Let $\phi(x, y)$ be the formula $\exists z\ x + z < y$ and let $n, n' \in \mathbb{Z}$. Then we denote

- by $\overline{\forall}\phi(x, y)$ the formula $\forall x \forall y(\exists z\ x + z < y)$,
- by $\phi(x, y)[n/x, n'/y]$ the formula where x is replaced by n and y is replaced by n': $\exists z\ n + z < n'$.

Formulae over $\mathbb{Z}$ (4)

How large is this class of formulae over $\mathbb{Z}$? Is it expressive enough?

1. How can we express the function $x!$ or x^x?
2. Which functions are not expressible? Are there any?
3. Is there an algorithm to decide whether a given sentence ϕ (formulae without free variables) holds in $\mathbb{Z}$, i.e. whether $\mathbb{Z} \models \phi$?
4. Attention: everything radically changes, when we do not allow multiplication! Then the resulting theory is **decidable**.

Formulae over $\mathbb{Z}$ (5)

Let us express some functions $f : \mathbb{Z} \to \mathbb{Z}$ that are not available as integer expressions.

Definition 6.6 (Functions Expressible over $\mathbb{Z}$)

A function $f : \mathbb{Z} \to \mathbb{Z}$ is expressible over $\mathbb{Z}$ if there is a formula $\Phi(x, y)$ with x, y as the only free variables such that the following holds for all $z, z' \in \mathbb{Z}$:

$$\mathbb{Z} \models \Phi(x, y)[z/x, z'/y] \text{ iff } f(z) = z'$$

Formulae over $\mathbb{Z}$ (6)

$x!$: Not obvious.

x^x: Not obvious.

Using Gödelization, it can be shown that all naturally occurring functions can be expressed. In fact, all **recursive functions** are expressible. We shall therefore use functions like $x!$ as macros (knowing that they can be expressed as formulae in the language).

Definition 6.7 (Program)

The set $\mathcal{P}rog$ of (while) programs over $\mathcal{V}ars$, $\mathcal{I}$ and $\mathcal{B}$ is given by all well-formed sequences which can be formed according to the following grammar:

$$\text{C} ::= \text{skip} \mid \text{x} := \text{I} \mid \text{C; C} \mid \text{if B } \{\text{C}\} \text{ else } \{\text{C}\} \mid \text{while B } \{\text{C}\}$$

where $B \in \mathcal{B}$, $I \in \mathcal{I}$, and $x \in \mathcal{V}ars$.

We usually write programs in lines (for better readability).

skip: Do nothing.

x := I: Assign I to x.

$C_1; C_2$: Sequential execution: C_2 is executed after C_1 provided that C_1 terminates.

if B $\{C_1\}$ else $\{C_2\}$: If B is true then C_1 is executed otherwise C_2.

while B $\{C\}$: C is executed as long as B is true.

The statement "B is true" is defined in Definition 6.11.

Example 6.8

What does the following program Fac(x) calculate?

```
y := 1;
z := 0;
while z ≠ x {
    z := z + 1;
    y := y · z
}
```

Is this class of programs large or small?

Can we express everything we want?

1. No, because we need better software engineering constructs.
2. Yes, because we have "while" and therefore we can do anything.
3. No, because we can not simulate (emulate) Java or C++.
4. It is already too large, because we assume that we can store arbitrary numbers. This is clearly not true on real computers.
5. Yes, because this class of programs corresponds to the class of deterministic Turing machines. And we cannot aim for more.

While Programs = Turing Machines

Turing Machines

The class of programs we have introduced corresponds exactly to programs of Turing machines. Thus it is an **idealization** (arbitrary numbers can be stored in one cell) and therefore it is much more expressive than any real programming language.
But playing quake requires a lot of coding . . .

- In particular, there is no algorithm to decide whether a given program terminates or not.
- The set of all terminating programs is recursive enumerable, but not recursive.
- Therefore the set of non-terminating programs is not even recursively enumerable.

Meaning of a Program

Definition 6.9 (State)

A **state** s is a mapping

$$s : \mathcal{V}ars \to \mathbb{Z}$$

A state assigns to each variable an integer. The set of all states is denoted by S.

The **semantics of a program** P is a **partial function**

$$[\![\mathrm{P}]\!] : S \to S$$

that describes how a state s changes after executing the program.

Definition 6.10 (Semantics: Integer Expression)

The semantics of integer expressions is defined:

$$\begin{aligned} [\![0]\!]_s &= 0 \\ [\![1]\!]_s &= 1 \\ [\![x]\!]_s &= s(x) \text{ for } \quad x \in \mathcal{V}ars \\ [\![E * E']\!]_s &= [\![E]\!]_s * [\![E']\!]_s \quad \text{for } * \in \{+, -, \cdot\} \end{aligned}$$

Definition 6.11 (Meaning: Boolean Expression)

The semantics of a Boolean expression is given as for sentential logic; that is, we write $\mathbb{Z}, s \models B$ if B is true in $\mathbb{Z}$ wrt to state (valuation) s.

Definition 6.12 (Meaning: Satisfaction $\mathbb{Z}, s \models \phi$)

The semantics of a formula ϕ is defined inductively. We have already defined it in Definition 6.11 for atomic expressions. Arbitrary formulae can also contain the quantifier $\exists$.

$$\mathbb{Z}, s \models \exists x \phi(x) \quad \text{iff} \quad \text{there is a } n \in \mathbb{Z} \text{ such that } \mathbb{Z}, s \models \phi(x)[n/x]$$

1. $\exists x \phi : \ \exists x\ 3x < 4$,
2. $\exists x \phi : \ \exists x\ 3x < 4y$.

Definition 6.13 (Meaning: Program)

1. $[\![\mathsf{skip}]\!](s) = s$
2. $[\![\mathsf{x} := \mathsf{I}]\!](s) = t$ where $t(y) = \begin{cases} s(y) & \text{if } y \neq x \\ [\![I]\!]_s & \text{else.} \end{cases}$
3. $[\![\mathsf{C}_1; \mathsf{C}_2]\!](s) = [\![\mathsf{C}_2]\!]([\![\mathsf{C}_1]\!](s))$
4. $[\![\text{ if B } \{\mathsf{C}_1\} \text{ else } \{\mathsf{C}_2\}]\!](s) = \begin{cases} [\![\mathsf{C}_1]\!](s) & \text{if } \mathbb{Z}, s \models B, \\ [\![\mathsf{C}_2]\!](s) & \text{else.} \end{cases}$
5. $[\![\text{ while B } \{\mathsf{C}\}]\!](s) =$
$\begin{cases} [\![\text{ while B } \{\mathsf{C}\}]\!]([\![\mathsf{C}]\!](s)) & \text{if } \mathbb{Z}, s \models B, \\ s & \text{else.} \end{cases}$

Note that the recursive definition of the while cases is the reason that $[\![\cdot]\!]$ might be a **partial function**: it is (perhaps) not everywhere defined.

6.3 Hoare Logic

Hoare Triples

How can we prove that a program/method P really does what it is intended to do?

We describe the desired state of the (overall) program before and after the execution of P.

For example, let P(x) be a program that should return a number whose square is strictly less than x.

Is the correctness of the program ensured when we require that P$(x) \cdot$ P$(x) < x$**?**

Not completely! What if $x < 0$? So, the precondition is also very important!

Definition 6.14 (Hoare Triple)

Let ϕ and ψ be **formulae of** $\mathcal{F}ml_{\mathcal{V}ars}$ and P be a **program**. A Hoare triple is given by

$$\{\phi\}\mathsf{P}\{\psi\}$$

where ϕ is said to be the precondition and ψ the postcondition.

A Hoare triple $\{\phi\}\mathsf{P}\{\psi\}$ is read as follows:

- If the overall program is in a state that satisfies ϕ, then,
- after the execution (and termination) of P the resulting state of the program satisfies ψ.

Let P be the program given above and y be the variable returned. Then the following Hoare triple would be a valid specification:

$$\{x > 0\}\mathsf{P}\{y \cdot y < x\}.$$

Partial and Total Correctness

We already introduced the informal reading of Hoare triples. Now we will be more formal. There are two cases one has to distinguish: Programs that terminate and ones which do not. Accordingly, we have two definitions of correctness:

partially correct: ψ holds after execution of P, **provided** that P **terminates**,

totally correct: we require in addition that P terminates.

Definition 6.15 (Partial Correctness)

A triple $\{\phi\}\mathsf{P}\{\psi\}$ is satisfied under partial correctness, written as

$$\models^{\mathsf{p}} \{\phi\}\mathsf{P}\{\psi\},$$

if for each state s

$$\mathbb{Z}, [\![\mathsf{P}]\!](s) \models \psi$$

provided that $\mathbb{Z}, s \models \phi$ and $[\![\mathsf{P}]\!](s)$ are defined.

The following program is always partially correct: while $\top$ $\{\mathsf{x} := 0\}$, **for arbitrary pre and postconditions.**

Definition 6.16 (Total Correctness)

A triple $\{\phi\}\mathsf{P}\{\psi\}$ is satisfied under total correctness, written as

$$\models^{\mathsf{t}} \{\phi\}\mathsf{P}\{\psi\},$$

if for each state s, $[\![\mathsf{P}]\!](s)$ is defined and

$$\mathbb{Z}, [\![\mathsf{P}]\!](s) \models \psi$$

provided that $\mathbb{Z}, s \models \phi$.

The following program is usually **not** totally correct: while $\top$ $\{\mathsf{x} := 0\}$. Why "usually"?

Example 6.17

Consider the following program Succ(x):

```
a := x + 1;
if (a − 1 = 0){
    y := 1
} else{
    y := a
}
```

Under which semantics does the program satisfy $\{\top\}\text{Succ}\{y = x + 1\}$?

Note, that the last program is only one program (and a very silly one) that ensures the condition $\{\top\}\mathsf{Succ}\{y = x + 1\}$. There are many more.

Example 6.18

Recall the program $\mathsf{Fac}(x)$ stated in Example 6.8. Which of the following statements are correct?

- $\models^t \{x \geq 0\}\mathsf{Fac}\{y = x!\}$,
- $\models^t \{\top\}\mathsf{Fac}\{y = x!\}$,
- $\models^p \{x \geq 0\}\mathsf{Fac}\{y = x!\}$ and
- $\models^p \{\top\}\mathsf{Fac}\{y = x!\}$.

Program and Logical Variables

The pre- and postconditions in a Hoare triple may contain two kinds of variables:

- program variables and
- logical variables.

Given a program P the former kind occurs in the program whereas the latter refers to fresh variables.

The following example makes clear why we need logical variables.

Example 6.19

The following program Fac2 works as well.

```
y := 1;
while (x ≠ 0){
    y := y · x;
    x := x − 1
}
```

Why is it not a good idea to use the Hoare triple $\{x \geq 0\}\text{Fac2}\{y = x!\}$ in this case?

What about $\{x = x_0 \wedge x \geq 0\}\text{Fac2}\{y = x_0!\}$? The variable x_0 is a **logical variable**!

6.4 Proof Calculi: Partial Correctness

Proof Rules

We have introduced a semantic notion of (partial and total) correctness. As for resolution we are after syntactic versions ($\vdash^t$ and $\vdash^p$) which can be used on computers.

Sound Calculus: Can we define a calculus, that allows us to derive **only valid Hoare triples**?

Complete Calculus: Can we define a calculus, that generates all valid Hoare triples?

Answer to question 1: Yes, for both versions.

Answer to question 2: No, not even for $\vdash^p$:

$$\vdash^p \{\top\}\, P\, \{\square\} \quad \text{iff} \quad P \textbf{ is not terminating.}$$

And this set is **not recursive enumerable**.

The following rules were proposed by R. Floyd and C.A.R. Hoare. A rule for each basic program construct is presented.

If a program is correct we may be able to show it by only applying the following rules (compare with Modus Ponens).

Composition

Definition 6.20 (Composition Rule (comp))

$$\frac{\{\phi\}\mathsf{C}_1\{\eta\} \qquad \{\eta\}\mathsf{C}_2\{\psi\}}{\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}}$$

In order to prove that $\{\phi\}\mathsf{C}_1;\mathsf{C}_2\{\psi\}$ we have to prove the Hoare triples $\{\phi\}\mathsf{C}_1\{\eta\}$ and $\{\eta\}\mathsf{C}_2\{\psi\}$ for some appropriate η.

An axiom

Definition 6.21 (Assignment Rule (assign))

$$\frac{}{\{\psi[I/x]\}\mathsf{x} := \mathsf{I}\{\psi\}}$$

The rule is self-explanatory. Recall that $\psi[I/x]$ denotes the formula that is equal to ψ but each free occurrence of x is replaced by I.

Another axiom

Definition 6.22 (Skip Rule (skip))

$$\frac{}{\{\phi\}\text{skip}\{\phi\}}$$

If Rule

Definition 6.23 (If Rule (if))

$$\frac{\{\phi \wedge B\}\mathsf{C}_1\{\psi\} \qquad \{\phi \wedge \neg B\}\mathsf{C}_2\{\psi\}}{\{\phi\}\ \mathsf{if\ B}\ \{\mathsf{C}_1\}\ \mathsf{else}\ \{\mathsf{C}_2\}\{\psi\}}$$

In order to prove the conclusion we prove that ψ holds for **both** possible program executions of the if-rule: when B holds and when it does not.

Partial-While Rule

Definition 6.24 (Partial-While Rule (while))

$$\frac{\{\psi \wedge B\}\mathsf{C}\{\psi\}}{\{\psi\}\ \mathsf{while\ B}\ \{\mathsf{C}\}\{\psi \wedge \neg B\}}$$

The while-rule is the most sophisticated piece of code; it may allow infinite looping. The formula ψ in the premise plays a decisive rule: ψ is true **before** and **after** the execution of C, i.e. C does not change the truth value of ψ.

Invariant

Definition 6.25 (Invariant)

An invariant of the while-statement while B {C} is any formula ψ such that $\models^p \{\psi \wedge B\}\mathsf{C}\{\psi\}$.

The conclusion says that ψ does not change, even when C is **executed several times** and if C terminates then B is false.

Example 6.26

What is an invariant of the following program?

```
y := 1;
z := 0;
while z ≠ x {
    z := z + 1;
    y := y · z
}
```

An invariant is $y = z!$.

Implication Rule

Definition 6.27 (Implication Rule (impl))

$$\frac{\{\phi\}\mathsf{C}\{\psi\}}{\{\phi'\}\mathsf{C}\{\psi'\}}$$

whenever $\mathbb{Z} \models \overline{\forall}(\phi' \to \phi)$ and $\mathbb{Z} \models \overline{\forall}(\psi \to \psi')$.

Implication Rule (2)

The implication rule allows us to **strengthen** the precondition ϕ to ϕ' and to **weaken** the postcondition ψ to ψ', provided that the two implications hold.

Note, that this rule links program logic with the truths of formulae in $\mathbb{Z}$, which have to be established. We call them proof obligations.

An Example

We will try to prove the correctness of the program Fac given in Example 6.8. That is, we would like to derive

$$\{\top\}\text{Fac}\{y = x!\}$$

For any input state after the execution of Fac the return value y should have the value $x!$.

We have to start with axioms (by the assignment rule):

$$\underbrace{\dfrac{\dfrac{\{1=1\}\mathsf{y:=1}\{y=1\}}{\{\top\}\mathsf{y:=1}\{y=1\}}(impl) \quad \dfrac{\{y=1\wedge 0=0\}\mathsf{z:=0}\{y=1\wedge z=0\}}{\{y=1\}\mathsf{z:=0}\{y=1\wedge z=0\}}(impl)}{\{\top\}\mathsf{y := 1; z := 0}\{y=1\wedge z=0\}}(comp)}_{\psi_1}$$

Again, on top are axioms:

$$\underbrace{\dfrac{\dfrac{\dfrac{\{y\cdot(z+1)=(z+1)!\}\mathsf{z:=z+1}\{y\cdot z=z!\}}{\{y=z!\wedge z\neq x\}\mathsf{z:=z+1}\{y\cdot z=z!\}}(impl) \quad \{y\cdot z=z!\}\mathsf{y:=y\cdot z}\{y=z!\}}{\{y=z!\wedge z\neq x\}\mathsf{z:=z+1;y:=y\cdot z}\{y=z!\}}(comp)}{\{y=z!\}\ \mathsf{while\ z\neq x\ \{z:=z+1; y:=y\cdot z\}}\{y=z!\wedge z=x\}}(while)}_{\psi_2}$$

Putting both parts together:

$$\dfrac{\psi_1 \quad \dfrac{\psi_2}{\{y=1\wedge z=0\}\ \mathsf{while\ z\neq x\ \{z:=z+1;y:=y\cdot z\}}\{y=x!\}}(impl)}{\{\top\}\mathsf{Fac}\{y=x!\}}(comp)$$

Proof obligations

We have to show that the following formulae hold in $\mathbb{Z}$:

1. $\top \to 1 = 1$,
2. $\forall y(y = 1 \to y = 1 \wedge 0 = 0)$,
3. $\forall x \forall y \forall z(y = z! \wedge z \neq x \to (y(z + 1) = (z + 1)!))$,
4. $\forall y \forall z(y = 1 \wedge z = 0 \to y = z!)$,
5. $\forall x \forall y \forall z(y = z! \wedge z = x \to y = x!)$.

Annotation Calculus

- The proof rules just presented are very similar to Hilbert style calculus rules.
- They inherit some undesirable properties: The proof calculus is not “easy to use”.
- The proof given for the small Fac-program looks already quite complicated.

In this section we present an annotation calculus which is much more convenient for practical purposes.

We consider a program P as a sequence of **basic commands**:

$$C_1;\ C_2;\ \ldots\ ;C_n$$

That is, none of the commands C_i is directly composed of smaller programs by means of composition. Assume we intend to show that

$$\vdash^p \{\phi\}P\{\psi\}$$

In the annotation calculus we try to find **appropriate** ϕ_i, $i = 0, \ldots, n$ such that

$$\textbf{if } \vdash^p \{\phi_i\}C_i\{\phi_{i+1}\} \text{ for all } i = 0, \ldots, n-1$$
$$\textbf{then also } \vdash^p \{\phi\}P\{\psi\}.$$

In other words, we **interleave** the program P with **intermediate conditions** ϕ_i

$$\{\phi_0\}C_1\{\phi_1\}\ C_2\{\phi_2\}\ldots\{\phi_{n-1}\}C_n\{\phi_n\}$$

where each step $\{\phi_i\}C_i\{\phi_{i+1}\}$ is justified by one of the proof rules given above.

That is, an annotation calculus is a way of summarizing the application of the proof rules to a program.

How to find the appropriate ϕ_i?

We determine "something like" the **weakest preconditions** successively such that

$$\vdash^p \{\phi'_0\}\mathsf{C}_1\{\phi'_1\}\ \mathsf{C}_2\{\phi'_2\}\ldots\{\phi'_{n-1}\}\mathsf{C}_n\{\phi'_n\}$$

Under which condition would this guarantee $\vdash^p \{\phi_0\}\mathsf{P}\{\phi_n\}$?

It must be the case that $\mathbb{Z} \models \overline{\forall}\, \phi_0 \to \phi'_0$ and $\mathbb{Z} \models \overline{\forall}\, \phi'_n \to \phi_n$

Why do we say "something like" the weakest precondition? We come back to this point later.

In the following we label the program P with preconditions by means of rewrite rules

$$\frac{X}{Y}$$

Such a rule denotes that X can be rewritten (or replaced) by Y.

Each rewrite rule results from a proof rule.

Definition 6.28 (Assignment Rule)

$$\frac{\mathsf{x := E}\{\psi\}}{\{\psi[E/x]\}\mathsf{x := E}\{\psi\}}$$

Definition 6.29 (If Rule (1))

$$\frac{\mathsf{if\ B}\ \{\mathsf{C}_1\}\ \mathsf{else}\ \{\mathsf{C}_2\}\{\psi\}}{\mathsf{if\ B}\ \{\mathsf{C}_1\{\psi\}\}\ \mathsf{else}\ \{\mathsf{C}_2\{\psi\}\}\{\psi\}}$$

Definition 6.30 (If Rule (2))

$$\frac{\mathsf{if\ B}\ \{\{\phi_1\}\cdots\}\ \mathsf{else}\ \{\{\phi_2\}\cdots\}}{\{(B \to \phi_1) \wedge (\neg B \to \phi_2)\}\ \mathsf{if\ B}\ \{\{\phi_1\}\cdots\}\ \mathsf{else}\ \{\{\phi_2\}\cdots\}}$$

Definition 6.31 (Partial-While Rule)

$$\frac{\text{while B } \{P\}}{\{\phi\}\ \text{while B } \{\{\phi \wedge \text{B}\}P\{\phi\}\}\{\phi \wedge \neg B\}}$$

where ϕ is an invariant of the while-loop.

Definition 6.32 (Skip Rule)

$$\frac{\text{skip}\{\phi\}}{\{\phi\}\text{skip}\{\phi\}}$$

Applying the rules just introduced, we end up with a finite sequence

$$s_1 s_2 \dots s_m$$

where each s_i is of the form $\{\phi\}$ or it is a command of a program (which in the case of if or while commands can also contain such a sequence).

It can also happen that two subsequent elements have the same form: $\dots\{\phi\}\{\psi\}\dots$. Whenever this occurs, we have to show that ϕ implies ψ: a proof obligation (see Slide 483).

Definition 6.33 (Implied Rule)

Whenever applying the rules lead to a situation where two formulae stand next to each other $\dots\{\phi\}\{\psi\}\dots$, then we add a proof obligation of the form

$$\mathbb{Z} \models \overline{\forall}\,(\phi \to \psi).$$

Consider $\{\phi\}$ while B $\{$P$\}\{\psi\}$. Then the while-rule yields the following construct:

$$\{\phi\}\{\eta\} \text{ while B } \{\{\eta \wedge \text{B}\}\text{P}\{\eta\}\}\{\eta \wedge \neg B\}\{\psi\}$$

That is, we have to show that the invariant η satisfies the following properties:

1. $\mathbb{Z} \models \overline{\forall}\,(\phi \to \eta)$,
2. $\mathbb{Z} \models \overline{\forall}\,((\eta \wedge \neg B) \to \psi)$,
3. $\vdash^p \{\eta\}$ while B $\{$P$\}\{\eta \wedge \neg B\}$, and
4. $\vdash^p \{\eta \wedge B\}\text{P}\{\eta\}$ (this is the fact that it is an invariant).

Example 6.34

Prove: $\models^p \{y = 5\}\mathsf{x := y + 1}\{x = 6\}$

1. $\mathsf{x := y + 1}\{x = 6\}$
2. $\{y + 1 = 6\}\mathsf{x := y + 1}\{x = 6\}$ (Assignment)
3. $\{y = 5\}\{y + 1 = 6\}\mathsf{x := y + 1}\{x = 6\}$ (Implied)

Example 6.35

Prove: $\models^p \{y < 3\}\mathsf{y := y + 1}\{y < 4\}$

1. $\mathsf{y := y + 1}\{y < 4\}$
2. $\{y + 1 < 4\}\mathsf{y := y + 1}\{y < 4\}$ (Assignment)
3. $\{y < 3\}\{y + 1 < 4\}\mathsf{y := y + 1}\{y < 4\}$ (Implied)

Example 6.36

$\{y = y_0\}$
z := 0;
while y $\neq$ 0 do{
    z := z + x;
    y := y − 1
}
$\{z = xy_0\}$

Firstly, we have to use the rules of the annotation calculus.

$\{y = y_0\}$
$\{I[0/z]\}$
z := 0;
$\{I\}$
while y $\neq$ 0 do{
    $\{I \wedge y \neq 0\}$
    $\{I[y-1/y][z+x/z]\}$
    z := z + x;
    $\{I[y-1/y]\}$
    y := y − 1
    $\{I\}$
}
$\{I \wedge y = 0\}$
$\{z = xy_0\}$

What is a suitable invariant?

We have to choose an invariant such that the following hold in $\mathbb{Z}$:

1. $y = y_0 \to I[0/z]$,
2. $I \wedge y \neq 0 \to I[y - 1/y][z + x/z]$,
3. $I \wedge y = 0 \to z = xy_0$.

What about $I : \top$? What about $I : (z + xy = xy_0)$?

It is easy to see that the latter invariant satisfies all three conditions. This proves the partial correctness of $\{y = y_0\}\mathsf{P}\{z = xy_0\}$.

How to ensure that $\models^p \{\phi\}P\{\psi\}$?

Definition 6.37 (Valid Annotation)

A valid annotation of $\{\phi\}P\{\psi\}$ is given if

1. only the rules of the **annotation calculus** are used;
2. each command in P is **embraced by a post and a precondition**;
3. the assignment rule is applied **to each** assignment;
4. each proof obligation introduced by the implied rule has to be verified.

Proposition 6.38 ($\vdash^p$ iff Valid Annotation)

Constructing valid annotations *is equivalent to* deriving the Hoare triple $\{\phi\}\mathsf{P}\{\psi\}$ *in the Hoare calculus:*

$\vdash^p \{\phi\}\mathsf{P}\{\psi\}$ *iff there is a* valid annotation *of* $\{\phi\}\mathsf{P}\{\psi\}$.

Note, this does **not** mean that the calculus is complete!

Exercise

Let the program P be:

z := x;

z := z + y;

u := z

Use the annotation calculus to prove that $\vdash^p \{\top\}\mathsf{P}\{u = x + y\}$.

Exercise

Given the program P:

$$\mathsf{a} := \mathsf{x} + 1;$$
$$\textbf{if } \mathsf{a} - 1 = 0 \ \{\mathsf{y} := 1\} \textbf{ else } \{\mathsf{y} := \mathsf{a}\}$$

Use the annotation calculus to prove that
$\vdash^p \{\top\}\mathsf{P}\{y = x + 1\}$.

Exercise

Given the program Sum:

```
z := 0;
while x > 0 {
    z := z + x;
    x := x − 1
}
```

Use the annotation calculus to prove that
$\vdash^p \{x = x_0 \wedge x \geq 0\}\text{Sum}\{z = \frac{x_0(x_0+1)}{2}\}$.

Weakest Liberal Precondition

In principle, the **annotation calculus** determines the weakest precondition of a program and checks whether the given precondition implies the calculated precondition. I.e. when we start with something of the form

$$\mathsf{P}\{\psi\}$$

then the annotation calculus leads to a Hoare triple

$$\{\phi\}\mathsf{P}'\{\psi\}$$

where ϕ is "something like the weakest precondition":

$$\vdash^p \{\phi\}\mathsf{P}\{\psi\}.$$

Without the while-command the annotation calculus does calculate the weakest precondition! But in the rule for the while-command **some** invariant is selected. This does not ensure that the **weakest** precondition is determined!

A formula ψ is said to be weaker than χ if it holds that $\mathbb{Z} \models \overline{\forall}(\chi \to \psi)$.

Theorem 6.39 (Weakest Liberal Precondition)

The weakest liberal precondition of a program P and postcondition ϕ, denoted by $\mathbf{wp}(\mathsf{P}, \phi)$, is the weakest formula ψ such that $\models^p \{\psi\}\mathsf{P}\{\phi\}$. Such a formula exists and can be constructed as a formula in our language.

The reason for the last theorem is that the model $\mathbb{Z}$ with $+, \cdot, <$ is powerful enough to express all notions we need.

6.5 Proof Calculi: Total Correctness

Proof Rules

We extend the proof calculus for partial correctness presented to one that covers **total correctness**.

Partial correctness does not say anything about the termination of programs. It is easy to see that only the **while** construct can cause the nontermination of a program. Hence, in addition to the partial correctness calculus we have to prove that a while loop terminates.

The proof of termination of while statements follows the following schema:

- Given a program P, identify an integer expression I whose value decreases after performing P but which is always non-negative.

Such an expression E is called a variant. Now, it is easy to see that a while loop has to terminate if such a variant exists. The corresponding rule is given below.

Definition 6.40 (Total-While Rule)

$$\frac{\{\phi \wedge B \wedge 0 \leq E = E_0\}\mathsf{C}\{\phi \wedge 0 \leq E < E_0\}}{\{\phi \wedge 0 \leq E\}\ \mathsf{while\ B\ \{C\}}\{\phi \wedge \neg B\}}$$

where

- ϕ is an **invariant** of the while-loop,
- E is a **variant** of the while-loop,
- E_0 represents the **initial value** of E before the loop.

Example 6.41

Let us consider the program Fac once more:

$$\begin{array}{l} \mathsf{y} := 1; \\ \mathsf{z} := 0; \\ \quad \text{while } (\mathsf{z} \neq \mathsf{x}) \ \{\mathsf{z} := \mathsf{z} + 1;\ \mathsf{y} := \mathsf{y} \cdot \mathsf{z}\} \end{array}$$

How does a variant for the proof of $\{x \geq 0\}\mathsf{Fac}\{y = x!\}$ look?

A possible variant is for instance $x - z$. The first time the loop is entered we have that $x - z = x$ and then the expression decreases step by step until $x - z = 0$.

Annotation Calculus

Definition 6.42 (Total-While Rule)

$$\frac{\text{while B } \{\text{P}\}}{\{\phi \wedge 0 \leq E\}\ \text{while B}\ \{\{\phi \wedge \text{B} \wedge 0 \leq \text{E} = \text{E}_0\}\text{P}\{\phi \wedge 0 \leq \text{E} < \text{E}_0\}\}\{\phi \wedge \neg B\}}$$

where

- ϕ is an invariant of the while-loop,
- E is a variant of the while-looping,
- E_0 represents the **initial value** of E before the loop.

Example 6.43

Prove that $\models^t \{x \geq 0\}\mathsf{Fac}\{y = x!\}$. What do we have to do at the beginning?

We have to determine a suitable **variant** and an **invariant**. As variant we may choose $x - z$ and as invariant $y = z!$. Now we can apply the annotation calculus.

$\{x \geq 0\}$
$\{1 = 0! \wedge 0 \leq x - 0\}$
y := 1;
$\{y = 0! \wedge 0 \leq x - 0\}$
z := 0;
$\{y = z! \wedge 0 \leq x - z\}$
while x $\neq$ z {
$\{y = z! \wedge x \neq z \wedge 0 \leq x - z = E_0\}$
$\{y(z+1) = (z+1)! \wedge 0 \leq x - (z+1) < E_0\}$
z := z + 1;
$\{yz = z! \wedge 0 \leq x - z < E_0\}$
y := y · z
$\{y = z! \wedge 0 \leq x - z < E_0\}$ }
$\{y = z! \wedge x = z\}$
$\{y = x!\}$

Weakest Precondition

Proposition 6.44 (wp- Total Correctness)

The weakest precondition for total correctness exists and can be expressed as a formula in our language.

Example 6.45

What is the weakest precondition for

$$\text{while i} < \text{n } \{\text{i} := \text{i} + 1\}\{i = n + 5\}?$$

$i = n + 5$.

6.6 Sound and Completeness

Theoretical Aspects

Finally, we consider some theoretical aspects with respect to the sound and completeness of the introduced calculi.

Recall, that a calculus is sound if everything that can be derived is also semantically true:

$$\text{If } \vdash^x \{\phi\}\mathsf{P}\{\psi\} \text{ then } \models^x \{\phi\}\mathsf{P}\{\psi\}$$

where $x \in \{p, t\}$ stands for partial and total correctness.

The reverse direction is referred to as completeness.

A calculus is complete if everything that can semantically be derived is also derivable by the calculus:

$$\text{If } \models^x \{\phi\}\mathsf{P}\{\psi\} \text{ then } \vdash^x \{\phi\}\mathsf{P}\{\psi\}$$

where $x \in \{p, t\}$ stands for partial and total correctness.

Which direction is more difficult to prove?

For the soundness we just have to make sure that all proof rules introduced make sense. That is, given a rule $\frac{X}{Y}$ it has to be shown that Y holds whenever X holds.

Theorem 6.46

The Hoare calculus is **sound**.

Proof.

Exercise! □

Theorem 6.47

The Hoare calculus is complete.

- The Hoare calculus contains axioms and rules that require to determine whether certain formulae (**proof obligations**) are true in $\mathbb{Z}$.
- The theory of $\mathbb{Z}$ is undecidable.
- Thus any re axiomatization of $\mathbb{Z}$ is incomplete.
- However, most proof obligations occurring in practice can be checked by theorem provers.

7. Knowledge Engineering: FOL

Content of this chapter:

FOL: While sentential logic SL has some nice features, it is quite restricted in expressivity. First order logic (FOL) is an extension of SL which allows us to express much more in a succinct way.

SIT-calculus: The Situation Calculus, introduced by John McCarthy, is a special method for using FOL to express the dynamics of an agent. Applying actions to the world leads to a series of successor worlds that can be represented with special FOL terms.

Content of this chapter (2):

Blocksworld: We consider the blocksworld scenario and discuss how to formalize that with FOL.

HOL: Finally, we give an outlook to higher order logics (HOL), in particular to second order logic. While we can express much more than in 1st order logic, the price we have to pay is that there is no correct and complete calculus.

Declarative: In this chapter we only consider the question How to use FOL to model the world? We are not concerned with **deriving new information** or with implementing FOL. This will be done in the next chapter.

7.1 First Order Logic

Definition 7.1 (First order logic $\mathcal{L}_{FOL}$, $\mathcal{L} \subseteq \mathcal{L}_{FOL}$)

The language $\mathcal{L}_{FOL}$ of first order logic (*Prädikatenlogik erster Stufe*) is:

- $x, y, z, x_1, x_2, \ldots, x_n, \ldots$: a countable set Var of variables
- for each $k \in \mathbb{N}_0$: $P_1^k, P_2^k, \ldots, P_n^k, \ldots$ a countable set Pred^k of k-dimensional predicate symbols (the 0-dimensional predicate symbols are the propositional logic constants from At of $\mathcal{L}_{SL}$, including $\Box$, $\top$).
- for each $k \in \mathbb{N}_0$: $f_1^k, f_2^k, \ldots, f_n^k, \ldots$ a countable set Funct^k of k-dimensional function symbols.
- $\neg$, $\wedge$, $\vee$, $\rightarrow$: the sentential connectives.
- $($, $)$: the parentheses.
- $\forall$, $\exists$: the quantifiers.

Definition (continued)

The 0-dimensional function symbols are called individuum constants – we leave out the parentheses. In general we will need – as in propositional logic – only a certain subset of the predicate or function symbols.

These define a language $\mathcal{L} \subseteq \mathcal{L}_{FOL}$ (analogously to definition 5.1 on page 356). The used set of predicate and function symbols is also called signature Σ.

Definition (continued)

The concept of an $\mathcal{L}$-term t and an $\mathcal{L}$-formula φ are defined inductively:

Term: $\mathcal{L}$-terms t are defined as follows:

1. each variable is a $\mathcal{L}$-term.
2. if f^k is a k-dimensional function symbol from $\mathcal{L}$ and $t_1, \ldots, t_k$ are $\mathcal{L}$-terms, then $f^k(t_1, \ldots, t_k)$ is a $\mathcal{L}$-term.

The set of all $\mathcal{L}$-terms that one can create from the set $X \subseteq \mathrm{Var}$ is called $Term_{\mathcal{L}}(X)$ or $Term_{\Sigma}(X)$. Using $X = \emptyset$ we get the set of basic terms $Term_{\mathcal{L}}(\emptyset)$, short: $Term_{\mathcal{L}}$.

Definition (continued)

Formula: $\mathcal{L}$-formulae φ are also defined inductively:

1. if P^k is a k-dimensional predicate symbol from $\mathcal{L}$ and $t_1, \ldots, t_k$ are $\mathcal{L}$-terms then $P^k(t_1, \ldots, t_k)$ is a $\mathcal{L}$-formula
2. for all $\mathcal{L}$-formulae φ is $(\neg\varphi)$ a $\mathcal{L}$-formula
3. for all $\mathcal{L}$-formulae φ and ψ are $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$ $\mathcal{L}$-formulae.
4. if x is a variable and φ a $\mathcal{L}$-formula then are $(\exists\, x\, \varphi)$ and $(\forall\, x\, \varphi)$ $\mathcal{L}$-formulae.

Definition (continued)

Atomic $\mathcal{L}$-formulae are those which are composed according to 1., we call them $At_{\mathcal{L}}(X)$ ($X \subseteq Var$). The set of all $\mathcal{L}$-formulae in respect to X is called $Fml_{\mathcal{L}}(X)$.

Positive formulae ($Fml^{+}_{\mathcal{L}}(X)$) are those which are composed using only 1, 3. and 4.

If φ is a $\mathcal{L}$-formula and is part of an other $\mathcal{L}$-formula ψ then φ is called **sub-formula** of ψ.

An illustrating example

Example 7.2 (From semigroups to rings)

We consider $\mathcal{L} = \{0, 1, +, \cdot, \leq, \doteq\}$, where $0, 1$ are **constants**, $+, \cdot$ binary **operations** and $\leq, \doteq$ binary **relations**. What can be expressed in this language?

Ax 1:	$\forall x \forall y \forall z$	$x + (y + z) \doteq (x + y) + z$
Ax 2:	$\forall x$	$(x + 0 \doteq 0 + x) \ \wedge \ (0 + x \doteq x)$
Ax 3:	$\forall x \exists y$	$(x + y \doteq 0) \ \wedge \ (y + x \doteq 0)$
Ax 4:	$\forall x \forall y$	$x + y \doteq y + x$
Ax 5:	$\forall x \forall y \forall z$	$x \cdot (y \cdot z) \doteq (x \cdot y) \cdot z$
Ax 6:	$\forall x \forall y \forall z$	$x \cdot (y + z) \doteq x \cdot y + x \cdot z$
Ax 7:	$\forall x \forall y \forall z$	$(y + z) \cdot x \doteq y \cdot x + z \cdot x$

Axiom 1 describes an *semigroup*, the axioms 1-2 describe a *monoid*, the axioms 1-3 a *group*, and the axioms 1-7 a *ring*.

Definition 7.3 ($\mathcal{L}$-structure $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$)

A $\mathcal{L}$-structure or a $\mathcal{L}$-interpretation is a pair $\mathcal{A} =_{def} (U_\mathcal{A}, I_\mathcal{A})$ with $U_\mathcal{A}$ being an arbitrary non-empty set, which is called the **basic set** (the **universe** or the **individuum range**) of $\mathcal{A}$. Further $I_\mathcal{A}$ is a mapping which

- assigns to each k-dimensional predicate symbol P^k in $\mathcal{L}$ a k-dimensional predicate over $U_\mathcal{A}$
- assigns to each k-dimensional function symbol f^k in $\mathcal{L}$ a k-dimensional function on $U_\mathcal{A}$

In other words: the domain of $I_\mathcal{A}$ is exactly the set of predicate and function symbols of $\mathcal{L}$.

Definition (continued)

The range of $I_{\mathcal{A}}$ consists of the predicates and functions on $U_{\mathcal{A}}$. We write:

$$I_{\mathcal{A}}(P) = P^{\mathcal{A}},\ I_{\mathcal{A}}(f) = f^{\mathcal{A}}.$$

φ be a $\mathcal{L}_1$-formula and $\mathcal{A} =_{def} (U_{\mathcal{A}}, I_{\mathcal{A}})$ a $\mathcal{L}$-structure. $\mathcal{A}$ is called **matching with φ** if $I_{\mathcal{A}}$ is defined for all predicate and function symbols which appear in φ, i.e. if $\mathcal{L}_1 \subseteq \mathcal{L}$.

FOL with Equality

Often, one assumes that the predicate symbol $\doteq$ is **built-in** and interpreted by identity in all structures $\mathcal{A}$:
$I_{\mathcal{A}}(\doteq) = \{(x, x) : x \in A\}$. In that case, $\doteq$ is not listed among the predicates in a model $\mathcal{A}$.

Definition 7.4 (Variable assignment ϱ)

A variable assignment ϱ over a $\mathcal{L}$-structure $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ is a function

$$\varrho : \mathrm{Var} \to U_\mathcal{A};\ x \mapsto \varrho(x).$$

Note that this is exactly what we called state in the chapter about the Hoare calculus.

Some structures

The following will be explained in detail on the blackboard.

We consider $\mathcal{L} = \{\leq, \doteq\}$ and the following structures

1. $(\mathbb{Z}, \leq^{\mathbb{Z}})$
2. $(\mathbb{R}, \leq^{\mathbb{R}})$
3. $(\mathbb{Q}, \leq^{\mathbb{Q}})$
4. $(\mathbb{Q}_0^+, \leq^{\mathbb{Q}_0^+})$

State formulae in $\mathcal{L}$ that are true in all structures, just in some, ... Is there a formula ϕ which can distinguish between $(\mathbb{R}, \leq^{\mathbb{R}})$ and $(\mathbb{Q}, \leq^{\mathbb{Q}})$?

Some structures (2)

We have seen some formulae that are true in these structures. Can we come up with a finite axiomatization?

Dense linear order without endpoints

It turns out, that the set of all formulae true in $(\mathbb{R}, \leq^{\mathbb{R}})$ coincides with the set all formulae true in $(\mathbb{Q}, \leq^{\mathbb{Q}})$. An axiomatization is given by the finite set of formulae stating that $<$ is a dense, linear order without endpoints. This theory is also **complete**.

Definition 7.5 (Semantics of first order logic, Model $\mathcal{A}$)

Let φ be a formula, $\mathcal{A}$ a structure matching with φ and ϱ a variable assignment over $\mathcal{A}$. For each term t, which can be built from components of φ, we define the value of t in the structure $\mathcal{A}$, called $\mathcal{A}(t)$.

1. for a variable x is $\mathcal{A}(x) =_{def} \varrho(x)$.
2. if t has the form $t = f^k(t_1, \ldots, t_k)$, with $t_1, \ldots, t_k$ being terms and f^k a k-dimensional function symbol, then $\mathcal{A}(t) =_{def} f^{\mathcal{A}}(\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k))$.

Compare with Definition 6.10.

Definition (continued)

We define inductively the logical value of a formula φ in $\mathcal{A}$:

1. if $\varphi =_{def} P^k(t_1, \ldots, t_k)$ with the terms $t_1, \ldots, t_k$ and the k-dimensional predicate symbol P^k, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \top, & \text{if } (\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k)) \in P^{\mathcal{A}}, \\ \square, & \text{else.} \end{cases}$$

2. if $\varphi =_{def} \neg\psi$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \top, & \text{if } \mathcal{A}(\psi) = \square, \\ \square, & \text{else.} \end{cases}$$

3. if $\varphi =_{def} (\psi \wedge \eta)$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \top, & \text{if } \mathcal{A}(\psi) = \top \text{ and } \mathcal{A}(\eta) = \top, \\ \square, & \text{else.} \end{cases}$$

Definition (continued)

4. if $\varphi =_{def} (\psi \vee \eta)$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \top, & \text{if } \mathcal{A}(\psi) = \top \text{ or } \mathcal{A}(\eta) = \top, \\ \square, & \text{else.} \end{cases}$$

5. if $\varphi =_{def} \forall x\, \psi$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \top, & \text{if } \forall\, d \in U_{\mathcal{A}} : \mathcal{A}_{[x/d]}(\psi) = \top, \\ \square, & \text{else.} \end{cases}$$

6. if $\varphi =_{def} \exists x\, \psi$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \top, & \text{if } \exists d \in U_{\mathcal{A}} : \mathcal{A}_{[x/d]}(\psi) = \top, \\ \square, & \text{else.} \end{cases}$$

For $d \in U_{\mathcal{A}}$ let $\mathcal{A}_{[d/x]}$ be the structure $\mathcal{A}'$, **identical to $\mathcal{A}$** except for the definition of $x^{\mathcal{A}'}$: $x^{\mathcal{A}'} =_{def} d$ (whether $I_{\mathcal{A}}$ is defined for x or not).

Definition (continued)

We write:

- $\mathcal{A} \models \varphi[\varrho]$ for $\mathcal{A}(\varphi) = \top$: $\mathcal{A}$ is a **model** for φ with respect to ϱ.
- If φ does not contain free variables, then $\mathcal{A} \models \varphi[\varrho]$ is independent from ϱ. We simply leave out ϱ.
- If **there is at least one model for** φ, then φ is called satisfiable or consistent.

A **free variable** is a variable which is not in the scope of a quantifier. For instance, z is a free variable of $\forall x P(x, z)$ but not free (or bounded) in $\forall z \exists x P(x, z)$.
A variable can occur free and bound in the same formula. So we have to talk about a particular **occurrence** of a variable: the very position of it in the formula.

Definition 7.6 (Tautology)

1. A theory is a set of formulae without free variables: $T \subseteq Fml_{\mathcal{L}}$. The structure $\mathcal{A}$ **satisfies** T if $\mathcal{A} \models \varphi$ holds for all $\varphi \in T$. We write $\mathcal{A} \models T$ and call $\mathcal{A}$ a model of T.
2. A $\mathcal{L}$-formula φ is called $\mathcal{L}$-tautology, if **for all matching $\mathcal{L}$-structures $\mathcal{A}$ the following holds:** $\mathcal{A} \models \varphi$.

From now on we suppress the language $\mathcal{L}$, because it is obvious from context.

Definition 7.7 (Consequence set $Cn(T)$)

A formula φ follows semantically from T, if for all structures $\mathcal{A}$ with $\mathcal{A} \models T$ also $\mathcal{A} \models \varphi$ holds.
We write: $T \models \varphi$.

In other words: all models of T do also satisfy φ.

We denote by $\mathbf{Cn}_{\mathcal{L}}(T) =_{def} \{\varphi \in Fml_{\mathcal{L}} : T \models \varphi\}$, or simply $Cn(T)$, the semantic consequence operator.

Lemma 7.8 (Properties of $Cn(T)$)

The semantic consequence operator Cn has the following properties

1. *T-extension: $T \subseteq Cn(T)$,*
2. *Monotony: $T \subseteq T' \Rightarrow Cn(T) \subseteq Cn(T')$,*
3. *Closure: $Cn(Cn(T)) = Cn(T)$.*

Lemma 7.9 ($\varphi \notin$ Cn(T))

$\varphi \notin Cn(T)$ if and only if there is a structure $\mathcal{A}$ with $\mathcal{A} \models T$ and $\mathcal{A} \models \neg\varphi$.

Or: $\varphi \notin Cn(T)$ iff there is a counterexample: a model of T in which φ is not true.

Definition 7.10 ($MOD(T)$, $Cn(\mathcal{U})$)

If $T \subseteq Fml_{\mathcal{L}}$, then we denote by $\mathsf{MOD}(T)$ the set of all $\mathcal{L}$-structures $\mathcal{A}$ which are models of T:

$$\mathsf{MOD}(T) =_{def} \{\mathcal{A} : \mathcal{A} \models T\}.$$

If $\mathcal{U}$ is a set of structures then we can consider all sentences, which are true in all structures. We call this set also $Cn(\mathcal{U})$:

$$Cn(\mathcal{U}) =_{def} \{\varphi \in Fml_{\mathcal{L}} : \forall \mathcal{A} \in \mathcal{U} : \mathcal{A} \models \varphi\}.$$

MOD is obviously dual to Cn:

$$Cn(\mathsf{MOD}(T)) = Cn(T), \quad \mathsf{MOD}(Cn(T)) = \mathsf{MOD}(T).$$

Definition 7.11 (Completeness of a theory T)

T is called complete, if for each formula $\varphi \in Fml_{\mathcal{L}}$: $T \models \varphi$ or $T \models \neg\varphi$ holds.

Attention:

Do not mix up this last condition with the property of a structure v (or a model): **each structure is complete in the above sense**.

Lemma 7.12 (Ex Falso Quodlibet)

T is consistent if and only if $Cn(T) \neq Fml_{\mathcal{L}}$.

An illustrating example

Example 7.13 (Natural numbers in different languages)

- $\mathcal{N}_{Pr} = (\mathbb{N}_0, 0^{\mathcal{N}}, +^{\mathcal{N}}, \doteq^{\mathcal{N}})$ („*Presburger Arithmetik*"),
- $\mathcal{N}_{PA} = (\mathbb{N}_0, 0^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, \doteq^{\mathcal{N}})$ („*Peano Arithmetik*"),
- $\mathcal{N}_{PA'} = (\mathbb{N}_0, 0^{\mathcal{N}}, 1^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, \doteq^{\mathcal{N}})$ (variant of $\mathcal{N}_{PA}$).

These sets each define the **same natural numbers**, but in different languages.

Question:

If the language is bigger then we might express more.
Is $\mathcal{L}_{PA'}$ strictly more expressive than $\mathcal{L}_{PA}$?

Answer:

No, because one can replace the $1^{\mathcal{N}}$ by a $\mathcal{L}_{PA}$-formula: there is a $\mathcal{L}_{PA}$-formula $\phi(x)$ so that for each variable assignment ρ the following holds:

$$\mathcal{N}_{PA'} \models_\rho \phi(x) \textit{ if and only if } \rho(x) = 1^{\mathcal{N}}$$

- Thus we can define a macro for 1.
- Each formula of $\mathcal{L}_{PA'}$ can be transformed into an equivalent formula of $\mathcal{L}_{PA}$.

Question:

Is $\mathcal{L}_{PA}$ perhaps more expressive than $\mathcal{L}_{Pr}$, or can the multiplication be defined somehow?

Indeed, $\mathcal{L}_{PA}$ is more expressive:

- the set of sentences valid in $\mathcal{N}_{Pr}$ is decidable, whereas
- the set of sentences valid in $\mathcal{N}_{PA}$ is not even recursively enumerable.

Question:

We have introduced $\mathcal{Z} = (\mathbb{Z}, 0^{\mathbb{Z}}, 1^{\mathbb{Z}}, +^{\mathbb{Z}}, -^{\mathbb{Z}}, \cdot^{\mathbb{Z}}, <^{\mathbb{Z}})$ in the chapter about the Hoare calculus. **How does it compare with $\mathcal{N}_{PA}$?**

- "$\doteq$" can be **defined** with $<$ and vice versa in $\mathcal{Z}$ and $\mathcal{N}_{PA}$ (resp. in $\mathcal{N}_{Pr}$).
- "$-$" can also be **defined** with the other constructs.
- $\mathcal{N}_{PA}$ can be defined in $\mathcal{Z}$ with an appropriate formula $\phi(x)$:

$$\mathcal{Z} \models_\rho \phi(x) \textit{ if and only if } \rho(x) \in \mathbb{N}_0$$

- Can $\mathcal{N}_{PA}$ be defined in $(\mathbb{Z}, +^{\mathbb{Z}}, \cdot^{\mathbb{Z}})$?

To be more precise, for each $\mathcal{L}_{PA}$ formula ϕ, there is a $\mathcal{L}_{\mathcal{Z}}$ formula ϕ' such that: if $\mathcal{N}_{PA'} \models \phi$ then $\mathcal{Z} \models \phi'$.

So $\mathcal{Z}$ is at least as difficult as $\mathcal{N}_{PA}$.

The converse is true as well. Therefore although the theories of $\mathcal{Z}$ and $\mathcal{N}_{PA}$ are **not identical**, the truth of a formula in one of them can be reduced to the truth of a translated formula in the other one.

Question:

What about $\mathcal{R} = (\mathbb{R}, 0^{\mathbb{R}}, 1^{\mathbb{R}}, +^{\mathbb{R}}, -^{\mathbb{R}}, .^{\mathbb{R}}, <^{\mathbb{R}})$ and $\mathcal{Q} = (\mathbb{Q}, 0^{\mathbb{Q}}, 1^{\mathbb{Q}}, +^{\mathbb{Q}}, -^{\mathbb{Q}}, .^{\mathbb{Q}}, <^{\mathbb{Q}})$? **How do they compare with $\mathcal{N}_{PA}$?**

- State a formula that **distinguishes** both structures.
- Can one define $\mathcal{Q}$ within $\mathcal{R}$ (as we did define $\mathcal{N}_{PA}$ in $\mathcal{Z}$)?
- Is there an axiomatization of $\mathcal{R}$?

In general, theories of particular structures are undecidable. But it depends on the underlying language.

As for sentential logic, formulae can be derived from a given theory and they can also (semantically) follow from it.

Syntactic derivability $\vdash$: the notion that certain formulae can be derived from other formulae using a certain calculus,

Semantic validity $\models$: the notion that certain formulae follow from other formulae based on the semantic notion of a model.

Definition 7.14 (Correct-, Completeness for a calculus)

Given an arbitrary calculus (which defines a notion $\vdash$) and a semantics based on certain models (which defines a relation $\models$), we say that

Correctness: The calculus is correct with respect to the semantics, if the following holds:

$$\Phi \vdash \phi \text{ implies } \Phi \models \phi.$$

Completeness: The calculus is complete with respect to the semantics, if the following holds:

$$\Phi \models \phi \text{ implies } \Phi \vdash \phi.$$

We have already defined a complete and correct calculus for sentential logic $\mathcal{L}_{SL}$. Such calculi also exist for first order logic $\mathcal{L}_{FOL}$.

Theorem 7.15 (Correct-, Completeness of FOL)

A formula follows semantically from a theory T if and only if it can be derived:

$$T \vdash \varphi \text{ if and only if } T \models \varphi$$

Theorem 7.16 (Compactness of FOL)

A formula follows follows semantically from a theory T if and only if it follows semantically from a finite subset of T:

$$Cn(T) = \bigcup \{Cn(T') : T' \subseteq T, T' \text{ finite}\}.$$

Predicate- or function- symbols?

How to formalize **each animal has a brain**?

1. Two unary predicate symbols: $animal(x)$, $has\_brain(x)$. The statement becomes

$$\forall x\ (\ animal(x) \rightarrow has\_brain(x)\ )$$

2. Finally, what about a binary predicate $is\_brain\_of(y, x)$ and the statement

$$\forall x\ \ (\ animal(x)\ \rightarrow\ \exists y\, is\_brain\_of(y, x)\ )$$

3. But why not introducing a unary function symbol $brain\_of(x)$ denoting x's brain? Then

$$\forall x\, \exists y\ \ (\ animal(x)\ \rightarrow\ y \doteq brain\_of(x)\ )$$

Unary function or binary predicate?

Given a unary function symbol $f(x)$ and a constant c. Then the only terms that can be built are of the form $f^n(c)$. Assume we have $\doteq$ as the **only predicate**. Then the atomic formulae that we can built have the form $f^n(c) \doteq f^m(c)$. We call this language $\mathcal{L}_1$.

Unary function or binary predicate? (2)

Assume now that instead of f and $\doteq$, we use a binary predicate $p_f(y, x)$ which formalizes $y \doteq f(x)$. We call this language $\mathcal{L}_2$.

- Can we **express** all formulae of $\mathcal{L}_1$ in $\mathcal{L}_2$?
- And vice versa?
- What is the difference between both approaches?

Tuna the cat

We formalize again the example of the killed cat, this time in FOL.

1. Jack owns a dog.
   $\mapsto owns(x,y), dog(x), jack.$
2. Dog owners are animal lovers.
   $\mapsto animal\_lover(x).$
3. Animal lovers do not kill animals.
   $\mapsto killer\_of(x,y), animal(x).$
4. Either Jack or Bill killed Tuna, the cat.
   $\mapsto bill, tuna,\ cat(x), killer\_of(x,y).$

The formalization follows on the blackboard.

7.2 Sit-Calculus

Question:

How do we axiomatize the Wumpus-world in FOL?

function KB-AGENT(*percept*) **returns** an *action*
  static: *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*

Idea:

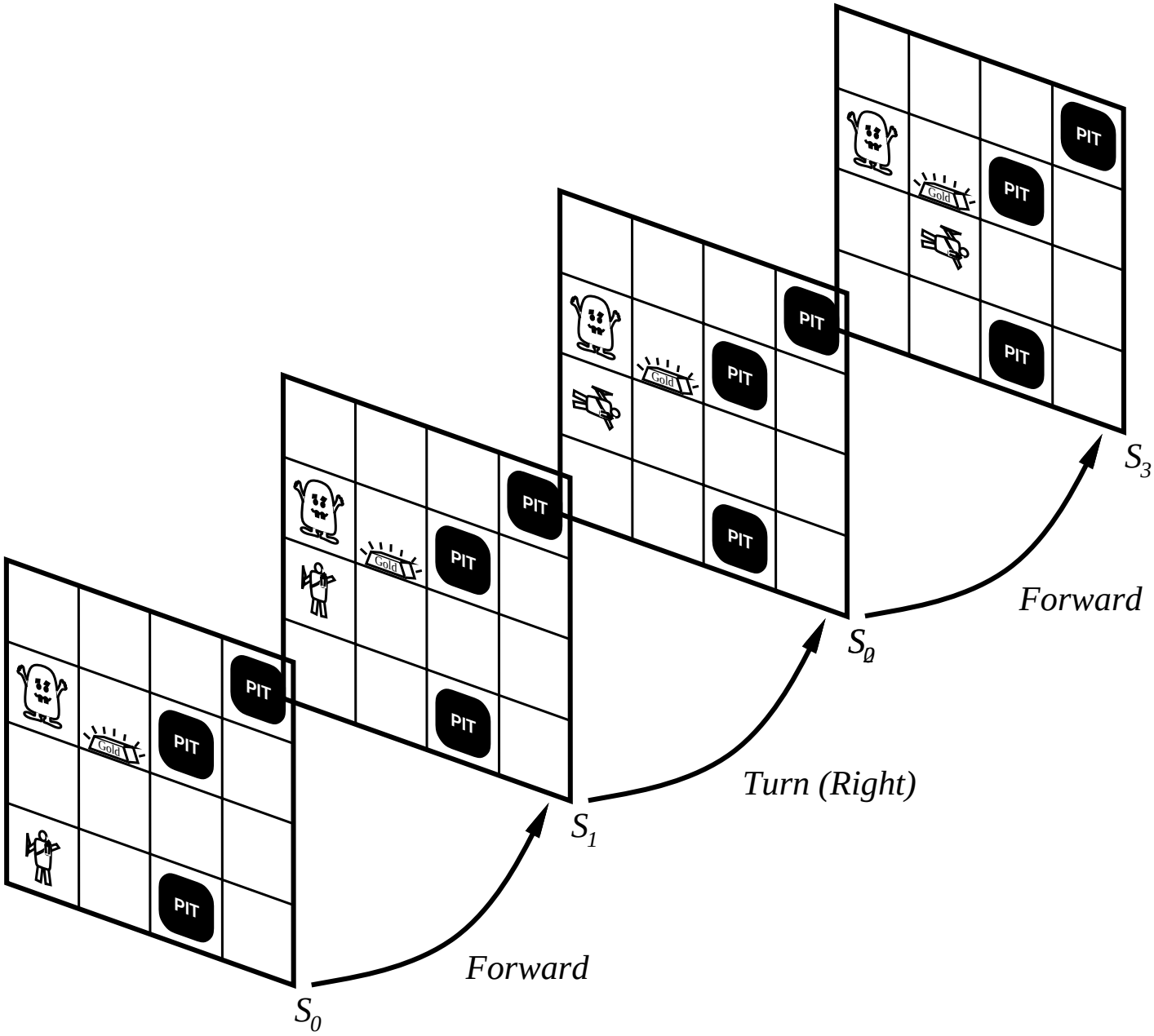
In order to describe actions or their effects consistently we consider the world as a sequence of situations (snapshots of the world). Therefore we have to extend each predicate by an additional argument.

We use the function symbol

$$result(action, situation)$$

as the term for the situation which emerges when the action $action$ is executed in the situation $situation$.

Actions: $Turn\_right$, $Turn\_left$, $Foreward$, $Shoot$, $Grab$, $Release$, $Climb$.


PIT
Gold
S_0
Forward
S_1
Turn (Right)
S_2
Forward
S_3

We also need a **memory**, a predicate

$$\mathbf{At}(person, location, situation)$$

with $person$ being either $Wumpus$ or $Agent$ and $location$ being the actual position (stored as pair [i,j]).

Important axioms are the so called **successor-state axioms**, they describe how actions effect situations. The most general form of these axioms is

true afterwards $\iff$ an action made it true
or it is already true and
no action made it false

Axioms about $At(p, l, s)$:

$$At(p, l, \textbf{result}(Forward, s)) \leftrightarrow ((l \doteq location\_ahead(p, s) \wedge \neg Wall(l))$$

$$At(p, l, s) \rightarrow Location\_ahead(p, s) \doteq Location\_toward(l, Orient.(p, s))$$

$$Wall([x, y]) \leftrightarrow (x \doteq 0 \vee x \doteq 5 \vee y \doteq 0 \vee y \doteq 5)$$

$$Location\_toward([x,y],0) \doteq [x+1,y]$$
$$Location\_toward([x,y],90) \doteq [x,y+1]$$
$$Location\_toward([x,y],180) \doteq [x-1,y]$$
$$Location\_toward([x,y],270) \doteq [x,y-1]$$

$$Orient.(Agent, s_0) \doteq 90$$
$$Orient.(p, \textbf{result}(a,s)) \doteq d \leftrightarrow$$
$$\big((a \doteq turn\_right \wedge d \doteq mod(Orient.(p,s)-90,360))$$
$$\vee(a \doteq turn\_left \wedge d \doteq mod(Orient.(p,s)+90,360))$$
$$\vee(Orient.(p,s) \doteq d \wedge \neg(a \doteq Turn\_right \vee a \doteq Turn\_left))$$

$mod(x,y)$ is the implemented "modulo"-function, assigning a value between 0 and y to each variable x.

Axioms about percepts, useful new notions:

$$
\begin{aligned}
Percept([Stench, b, g, u, c], s) &\rightarrow Stench(s) \\
Percept([a, Breeze, g, u, c], s) &\rightarrow Breeze(s) \\
Percept([a, b, Glitter, u, c], s) &\rightarrow At\_Gold(s) \\
Percept([a, b, g, Bump, c], s) &\rightarrow At\_Wall(s) \\
Percept([a, b, g, u, Scream], s) &\rightarrow Wumpus\_dead(s)
\end{aligned}
$$

$$
\begin{aligned}
At(Agent, l, s) \wedge Breeze(s) &\rightarrow Breezy(l) \\
At(Agent, l, s) \wedge Stench(s) &\rightarrow Smelly(l)
\end{aligned}
$$

$$
\begin{aligned}
Adjacent(l_1, l_2) &\leftrightarrow \exists\, d\; l_1 \doteq Location\_toward(l_2, d) \\
Smelly(l_1) &\rightarrow \exists l_2\; At(Wumpus, l_2, s) \wedge \\
&\qquad (l_2 \doteq l_1 \vee Adjacent(l_1, l_2))
\end{aligned}
$$

$$
\begin{aligned}
&Percept([none, none, g, u, c], s) \wedge \\
&At(Agent, x, s) \wedge Adjacent(x, y) \\
&\qquad \rightarrow OK(y) \\
&(\neg At(Wumpus, x, t) \wedge \neg Pit(x)) \\
&\qquad \rightarrow OK(y) \\
&At(Wumpus, l_1, s) \wedge Adjacent(l_1, l_2) \\
&\qquad \rightarrow Smelly(l_2) \\
&At(Pit, l_1, s) \wedge Adjacent(l_1, l_2) \\
&\qquad \rightarrow Breezy(l_2)
\end{aligned}
$$

Axioms to describe actions:

$$
\begin{array}{lcl}
Holding(Gold, \textbf{result}(Grab, s)) & \leftrightarrow & At\_Gold(s) \vee \\
 & & Holding(Gold, s) \\
Holding(Gold, \textbf{result}(Release, s)) & \leftrightarrow & \square \\
Holding(Gold, \textbf{result}(Turn\_right, s)) & \leftrightarrow & Holding(Gold, s) \\
Holding(Gold, \textbf{result}(Turn\_left, s)) & \leftrightarrow & Holding(Gold, s) \\
Holding(Gold, \textbf{result}(Forward, s)) & \leftrightarrow & Holding(Gold, s) \\
Holding(Gold, \textbf{result}(Climb, s)) & \leftrightarrow & Holding(Gold, s)
\end{array}
$$

Each effect must be described carefully.

Axioms describing preferences among actions:

$$Great(a,s) \rightarrow Action(a,s)$$
$$(Good(a,s) \wedge \neg\exists b\, Great(b,s)) \rightarrow Action(a,s)$$
$$(Medium(a,s) \wedge \neg\exists b\, (Great(b,s) \vee Good(b,s))) \rightarrow Action(a,s)$$
$$(Risky(a,s) \wedge \neg\exists b\, (Great(b,s) \vee Good(b,s) \vee Medium(a,s))) \rightarrow Action(a,s)$$
$$At(Agent,[1,1],s) \wedge Holding(Gold,s) \rightarrow Great(Climb,s)$$
$$At\_Gold(s) \wedge \neg Holding(Gold,s) \rightarrow Great(Grab,s)$$
$$At(Agent,l,s) \wedge \neg Visited(Location\_ahead(Agent,s)) \wedge$$
$$\wedge OK(Location\_ahead(Agent,s)) \rightarrow Good(Forward,s)$$
$$Visited(l) \leftrightarrow \exists s\, At(Agent,l,s)$$

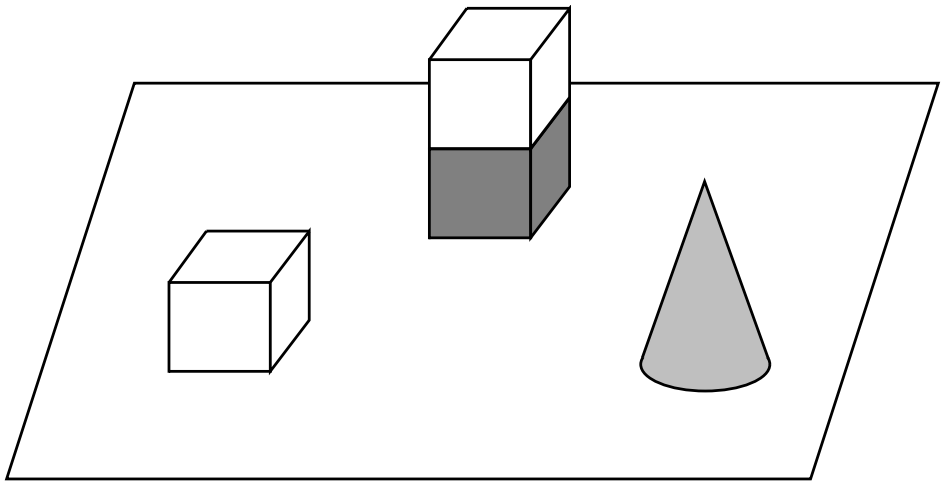
The goal is not only to find the gold but also to return safely.
We need additional axioms like

$$Holding(Gold,s) \rightarrow Go\_back(s).$$

7.3 The Blocksworld

Blocksworld

The Blocks World (BW) is one of the most popular domains in AI (first used in the 1970s). However, the setting is easy:



Domain

Blocks of various shapes, sizes and colors sitting on a table and on each other.

(Here: Quadratic blocks of equal size.)

Actions

Pick up a block and put it to another position (tower of blocks or table). Only the topmost blocks can be used.

How to formalize this? What language to use?

Choosing the predicates: First try.

1. One action: $move$. We add a ternary predicate $move(b, x, y)$: The block b is moved from x to y if both b and y are clear (nothing on top of them).
2. So we need a predicate $clear(x)$.
3. One predicate $on(b, x)$ meaning *block b is on x*.
4. But then $clear(x)$ can be defined by $\forall y \neg on(y, x)$.

Problem: What about the table?

If we view the table as a simple block: $clear(table)$ means that **there is nothing on the table**.

Choosing the predicates: Second try.

Keep the current framework, view the table as a block but add an additional binary predicate

$$move\_to\_table(b, x),$$

meaning that the block b is moved from x to the table.

$clear(x)$ is interpreted as there is free place on x to put a block on x. This interpretation does also work for $x \doteq table$.

Choosing the predicates: Third try.

Definition 7.17 ($\mathcal{L}_{\text{BW}}$)

The BW language $\mathcal{L}_{\text{BW}} \subseteq \mathcal{L}_{FOL}$ is the subset of FOL having a signature consisting of the two **binary** predicate symbols $above$ and $\doteq$.

We define the following macros:

$$\begin{aligned} on(x,y) &: above(x,y) \wedge \neg(\exists z(above(x,z) \wedge above(z,y))) \\ onTable(x) &: \neg(\exists y(above(x,y))) \\ clear(x) &: \neg(\exists y(above(y,x))) \end{aligned}$$

How do the $\mathcal{L}_{BW}$-structures look like?

Do they all make sense?

Our blocksworld has a very specific structure, which should be reflected in the models!

Definition 7.18 (Chain)

Let A be a nonempty set. We say that $(A, <)$ is a chain if $<$ is a binary relation on A which is

1. **irreflexive**,
2. **transitive**, and
3. **connected**, (i.e., for all $a, b \in A$ it holds that either $a < b$, $a = b$, or $a > b$).

(A chain is interpreted as a tower of blocks.)

Definition 7.19 ($\mathcal{L}_{BW}$-BW-structure)

An **BW-structure** is a $\mathcal{L}_{BW}$-structure $\mathcal{A} = (U, I)$ in which $I(above)$ is a finite and disjoint union of chains over U, i.e.

$$I(above) = \bigcup_{(A,<)\in A'} \{(a,b) \in A^2 \mid a > b\}\}$$

where A' is a set of chains over U such that for all $(A_1, <_1), (A_2, <_2) \in A'$ with $(A_1, <_1) \neq (A_2, <_2)$ it holds that $A_1 \cap A_2 = \emptyset$.

(Note: $I(above)$ is transitive!)

Definition 7.20 (BW-theory)

The theory $Th(BW)$ consists of all $\mathcal{L}_{\mathsf{BW}}$-sentences which are true in all BW-structures.

Is the theory complete?

No, consider for example

$$\forall x, y(onTable(x) \wedge onTable(y) \rightarrow x \doteq y)$$

- What about planning in the blocksworld?
- This should be done automatically as in the case of the Sudoku game or the puzzle.
- Thus we need a FOL theorem prover.

$\rightsquigarrow$ An axiomatization is needed!

A complete axiomatization

The following set $\mathcal{AX}_{\mathsf{BW}}$ of axioms was proposed by Cook and Liu (2003):

1. $\forall x \neg above(x, x)$
2. $\forall x \forall y \forall z \ (above(x, y) \wedge above(y, z)) \rightarrow above(x, z)$
3. $\forall x \forall y \forall z \ (above(x, y) \wedge above(x, z)) \rightarrow (y = z \vee above(y, z) \vee above(z, y))$
4. $\forall x \forall y \forall z \ (above(y, x) \wedge above(z, x)) \rightarrow (y = z \vee above(y, z) \vee above(z, y))$
5. $\forall x \ (onTable(x) \vee \exists y(above(x, y) \wedge onTable(y)))$
6. $\forall x \ (clear(x) \vee \exists y(above(y, x) \wedge clear(y)))$
7. $\forall x \forall y \ (above(x, y) \rightarrow (\exists z on(x, z) \wedge \exists z on(z, y))$

The last statement says that if an element is not on top (y) then there is a block above it, and if an element is not at the bottom (x) then there is an element below it.

Is every $\mathcal{L}_{\mathsf{BW}}$-BW-structure also a model for $\mathcal{AX}_{\mathsf{BW}}$?

Lemma 7.21

$Cn(\mathcal{AX}_{BW}) \subseteq Th(BW)$.

Proof: $\rightsquigarrow$ Exercise

Indeed, both sets are identical:

Theorem 7.22 (Cook and Liu)

$Cn(\mathcal{AX}_{BW}) = Th(BW)$.

Thus the axiomatization is sound and complete.

Additionally, the theory is decidable!

$\rightsquigarrow$ **We are ready to use a theorem prover!**

7.4 Higher order logic

Definition 7.23 (Second order logic $\mathcal{L}_{PL\,2}$)

The language $\mathcal{L}_{2ndOL}$ of second order logic consists of the language $\mathcal{L}_{FOL}$ and additionally

- for each $k \in \mathbb{N}_0$: $X_1^k, X_2^k, \ldots, X_n^k, \ldots$ a countable set $\mathcal{RelVar}^k$ of k-ary predicate variables.

Thereby the set of terms gets larger:

- if X^k is a k-ary predicate variable and $t_1, \ldots, t_k$ are terms, then $X^k(t_1, \ldots, t_k)$ is also a term

and also the set of formulae:

- if X is a predicate variable, φ a formula, then $(\exists X \varphi)$ and $(\forall X \varphi)$ are also formulae.

Not only elements of the universe can be quantified but also arbitrary subsets resp. k-ary relations.
The semantics do not change much – except for the new interpretation of formulae like $(\exists X \varphi)$, $(\forall X \varphi)$).

We also require from $I_{\mathcal{A}}$ that the new k-ary predicate variables are mapped onto k-ary relations on $U_{\mathcal{A}}$.

- if $\varphi =_{def} \forall X^k \psi$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \top, & \text{if for all } R^k \subseteq U_{\mathcal{A}} \times \cdots \times U_{\mathcal{A}} : \mathcal{A}_{[X^k/R^k]}(\psi) = \top, \\ \Box, & else. \end{cases}$$

- if $\varphi =_{def} \exists X^k \psi$, then

$$\mathcal{A}(\varphi) =_{def} \begin{cases} \top, & \text{if there is a } R^k \subseteq U_{\mathcal{A}} \times \cdots \times U_{\mathcal{A}} \text{ with } \mathcal{A}_{[X^k/R^k]}(\psi) = \top, \\ \Box, & else. \end{cases}$$

We can quantify over arbitrary n-ary relations, not just over elements (like in first order logic).

Dedekind/Peano Characterization of $\mathbb{N}$

The natural numbers satisfy the following axioms:

- Ax_1: 0 is a natural number.
- Ax_2: For each natural number n, there is exactly one **successor** $S(n)$ of it.
- Ax_3: There is no natural number which has 0 as its successor.
- Ax_4: Each natural number is successor of **at most** one natural number.
- Ax_5: The set of natural numbers is the smallest set N satisfying the following properties:
  1. $0 \in N$,
  2. $n \in N \Rightarrow S(n) \in N$.

The natural numbers are characterized up to isomorphy by these axioms.

How to formalize the last properties?

Language: We choose "0" as a constant and "$S(\cdot)$" as a unary function symbol.

Axiom for 0: $\forall x \neg S(x) \doteq 0$.

Axiom for S: $\forall x \forall y \, (S(x) \doteq S(y) \rightarrow x \doteq y)$.

Axiom 5: $\forall X \, (\, (X(0) \wedge \forall x \, (X(x) \rightarrow X(S(x)))) \rightarrow \forall y X(y))$.

While the first two axioms are **first-order**, the last one is essentially **second-order**.

Different languages for $\mathcal{N}_{PA}$

- $\mathcal{N}_{PA} = (\mathbb{N}_0, 0^{\mathcal{N}}, 1^{\mathcal{N}}, +^{\mathcal{N}}, \doteq^{\mathcal{N}})$
- $\mathcal{N}_{PA} = (\mathbb{N}_0, 0^{\mathcal{N}}, S^{\mathcal{N}}, +^{\mathcal{N}}, \doteq^{\mathcal{N}})$
- $\mathcal{N}_{PA} = (\mathbb{N}_0, 0^{\mathcal{N}}, 1^{\mathcal{N}}, S^{\mathcal{N}}, +^{\mathcal{N}}, \doteq^{\mathcal{N}})$
- $\mathcal{N}_{PA} = (\mathbb{N}_0, 1^{\mathcal{N}}, +^{\mathcal{N}}, \doteq^{\mathcal{N}})$

All these structures, resp. their formulae are interdefinable.

Question

What do the following two 2nd order sentences mean?

- $$\forall x \forall y \, (x \doteq y \iff \forall X \, (X(x) \iff X(y))),$$

- $$\begin{aligned} \forall X \, (\; & \forall x \exists ! y X(x,y) \wedge \\ & \forall x \forall y \forall z \, (\, (X(x,z) \wedge X(y,z)) \rightarrow x \doteq y)) \\ & \rightarrow \forall x \exists y X(y,x) \;) \end{aligned}$$

Answer:

The first sentence shows that equality can be defined in 2nd OL (in contrast to FOL).
The second sentence holds in a structure iff it is finite. Note that this cannot be expressed in FOL.

While the semantics of $\mathcal{L}_{2nd\,OL}$ is a canonical extension of $\mathcal{L}_{FOL}$, this does not hold for the calculus level. It can be shown that the set of valid sentences in $\mathcal{L}_{2nd\,OL}$ is not even recursively enumerable.

Attention:

There is no correct and complete calculus for 2nd Order Logic!

8. Knowledge Engineering: Provers

Content of this chapter:

Provers: We describe several theorem provers that are freely available and that can handle FOL.

Resolution: There are several calculi for implementing FOL. While the classical ones are not really feasible, resolution calculi can be efficiently implemented (and are the basis of most provers).

Variants: Resolution implementations often have huge search spaces. There are several variants that can be more efficiently implemented. SLD resolution leads to PROLOG as a programming language.

8.1 Theorem Proving

Automated Theorem Proving

Development of computer programs that show that some statement (the conjecture) is a logical consequence of a set of statements (axioms and assumptions).

Examples

- First year math students usually prove that groups of order two are commutative.
- Management consultants might formulate axioms that describe how organizations grow and interact, and from those axioms prove that organizational death rates decrease with age.

Examples (cont.)

- Hardware developers might validate the design of a circuit by proving a conjecture that describes a circuit's performance, given axioms that describe the circuit itself.
- Lazy crosswords fans might prepare a database of words and rules of putting a crossword together, and generate the crosswords solution as a proof.

The Idea

- Specification of what we have (the system): Assumptions/axioms $A_1, \ldots, A_n$
- Specification of what we want: Goals $G_1, \ldots, G_k$

Now, we want to prove or disprove that

$$\boxed{A_1 \wedge \cdots \wedge A_n} \models \boxed{G_1 \wedge \cdots \wedge G_k}$$

$\rightsquigarrow$ verification!

constructive proof $\rightsquigarrow$ synthesis!

How can we use a theorem prover?

- **Correctness proving:** We prove $A_1 \wedge \cdots \wedge A_n \vdash G_1 \wedge \cdots \wedge G_k$
- **Testing:** We look for counterexamples (models!)
- **Solution synthesis:** The constructed proof/model is the solution

  $\rightsquigarrow$ *software synthesis*

  $\rightsquigarrow$ *hardware synthesis*

  $\rightsquigarrow$ *plan generation etc.*

The language = logic

Often: Classical First Order Logic

Possibly: A non-classical logic or higher order logic

The system needs a formal description of the problem in some logical form.

- User prepares the description.
- Prover attempts to solve the problem.
- If **successful**: Proof is the output.
- If **unsuccessful**: User can provide guidance, simplify/split the problem, and/or revise the description.

This formality is both the underlying strength and handicap of ATP.

Pros of logical languages

- Allow a **precise formal statement** of the necessary information.
- **No ambiguity** in the statement of the problem (often the case when using natural language).
- Force the user to describe the problem precisely and accurately.
- The process often leads to a **clearer and deeper understanding** of the problem domain.

Cons of logical languages

- Require a precise formal statement of the necessary information
  $\rightsquigarrow$ many problems are not formalizable!
  Examples:
  Creating a **good** user interface.
  Creating a system which defends itself **reasonably well** against *unpredicted* situations.
- Even for domains which are in principle formalizable: Inaccurate formalizations are easy to obtain.

Cons of logical languages (cont.)

- Force the user to describe the problem precisely and accurately.
  $\rightsquigarrow$ considerable skills needed!
- Computational intractability.

Computational complexity of the problem

- Propositional logic: co-NP-complete.
- First-order logic: **recursively enumerable (re)**.
- Higher-order logic: Not even re.

Way out of the complexity

Computer-assisted theorem proving

Fully automatic theorem prover ← · · · · · · → Proof checker

- **Interactive** theorem provers require a **human** to give hints to the system.
- The interaction may be at a very detailed level, where the user guides the inferences made by the system, or at a much higher level where the user determines intermediate lemmas to be proved on the way to the proof of a conjecture.
- Often: the user defines a number of proof strategies $\rightsquigarrow$ proving toolbox (e.g., Isabelle).

Popular ATP techniques

- First-order resolution with unification,
- Higher-order unification,
- Model elimination,
- Superposition and term rewriting,
- Lean theorem proving,
- Method of analytic tableaux,
- Mathematical induction,
- DPLL (Davis-Putnam-Logemann-Loveland algorithm).

All the previously mentioned complexity bounds still apply!

CASC: CADE ATP System Competition, a yearly contest of first-order provers for many important classes of first-order problems.

`http://www.cs.miami.edu/~tptp/CASC/`

Some important first-order provers (all have won at least one CASC competition division):

- E
- Gandalf
- Otter $\rightsquigarrow$ Prover9/Mace4
- SETHEO
- SPASS
- Vampire (won the "world cup for theorem provers" for eight years: 1999, 2001–2007)
- Waldmeister (won the CASC UEQ division for the last ten years: 1997-2006); recently embedded in Mathematica

Other Important Systems

- HOL
- Isabelle
- Mizar

Growing competition: Model checking (especially in logics of time, action and knowledge).

Example: animals, cats, and Garfield

Assumptions:

cat(x) -> animal(x).
cat(Garfield).

Goals:

exists x animal(x).
-(exists x animal(x)).

$\rightsquigarrow$ Ontologies, Semantic web

Prover9 Syntax

Meaning	Connective	Example
negation	–	(–p)
disjunction	|	(p | q | r)
conjunction	&	(p & q & r)
implication	–>	(p –> q)
equivalence	<–>	(p <–> q)
universal quant.	all	(all x all y p(x,y))
existential quant.	exists	(exists x p(x,y))

Prover9 Syntax (cont.)

- Variables start with (lower case) u through z
- Otherwise: Constants
- Free variables in clauses and formulas are assumed to be universally quantified at the outermost level

- Prover9 uses resolution with refutation. Thus, its inference rules operate on clauses.
- If non-clausal formulas are input, Prover9 immediately translates them to clauses by NNF, Skolemization, and CNF conversions.

Schubert's Steamroller

Success Stories

Fields where ATP has been successfully used:

- Logic and mathematics.
- Computer science and engineering (software creation and verification, hardware design and verification – esp. integrated circuit design and verification, and knowledge based systems).
- Social science.

Mathematics

- EQP: Settling of the Robbins problem (open for 63 years!)
- Otter: Several results in quasi-groups; axiomatizations of algebraic structures.
- Mizar: Cross-verification of the Mathematical Library (ongoing).
- Geometry Expert: New results in Euclidean geometry.
- NUPRL helped to confirm Higman's lemma and Gerard's paradox.

Mathematics (cont.)

Results by dedicated provers (no general theorem provers):

- Proof of four color theorem (very controversial: The first mathematical proof which was impossible to verify by humans due to the enormous size of the program's calculation).
- Solving the game of Connect Four.

Software creation

- KIDS: Used to derive scheduling algorithms that have outperformed currently used algorithms.
- Amphion (NASA): Determining subroutines for satellite guidance
- Certifiable Synthesis (NASA).

Software verification

- KIV (Karlsruhe): Verified an implementation of set functions, graph representations and algorithms, and a Prolog to WAM compiler.
- PVS: Diagnosis and scheduling algorithms for fault tolerant architectures; requirements specification for parts of the space shuttle flight control system.

Hardware verification

- Employed e.g. by IBM, Intel, and Motorola (especially since the Pentium FDIV bug).
- ACL2: Correctness proof of the floating point divide code for AMD5K86 microprocessor.
- ANALYTICA: Used to verify a division circuit that implements the floating point standard of IEEE.
- RRL: Verification of adder and multiplier circuits.

Hardware verification (cont.)

- PVS: Used to verify a microprocessor for aircraft flight control.
- Nqthm: Correctness proof of the FM9001 microprocessor.
- HOL: Hardware verification at Bell Laboratories.

Verification of critical section

8.2 Resolution

Definition 8.1 (Most general unifier: mgU)

Given a finite set of equations between terms or equations between literals.
Then there is an algorithm which calculates a **most general solution substitution** (i.e. a substitution of the involved variables so that the left sides of all equations are syntactically identical to the right sides) or which returns *fail*.
In the first case the **most general solution substitution** is defined (up to renaming of variables): it is called

mgU, most general unifier

- $p(x, a) = q(y, b)$,
- $p(g(a), f(x)) = p(g(y), z)$.

Basic substitutions are:

$[a/y, a/x, f(a)/z]$, $[a/y, f(a)/x, f(f(a))/z]$,

and many more.

The mgU is: $[a/y, f(x)/z]$.

Given: $f(x, g(h(y), y)) = f(x, g(z, a))$

The algorithm successively calculates the following sets of equations:

$$\{\ x = x,\ g(h(y), y) = g(z, a)\ \}$$
$$\{\ g(h(y), y) = g(z, a)\ \}$$
$$\{\ h(y) = z,\ y = a\ \}$$
$$\{\ z = h(y),\ y = a\ \}$$
$$\{\ z = h(a),\ y = a\ \}$$

Thus the mgU is: $[x/x, a/y, h(a)/z]$.

The occur-check

Given: $f(x, g(x)) = f(c, c)$

Is there an mgU?

The algorithm gives the following:

$$\{\, x = c,\ g(x) = c \,\}$$

- But setting $x = c$ is not a unifying substitution, because $c \neq g(c)$.
- Therefore there is no mgU. And the algorithm has to do this check, called occur check, to test whether the substitution is really correct.
- However, this check is computationally expensive and many algorithms **do not do it**.

A resolution calculus for FOL

The resolution calculus is defined over the language $\mathcal{L}^{res} \subseteq \mathcal{L}_{FOL}$ where the set of well-formed formulae $Fml^{Res}_{\mathcal{L}^{res}}$ consists of all disjunctions of the following form

$$A \vee \neg B \vee C \vee \ldots \vee \neg E,$$

i.e. the disjuncts are only atoms or their negations. No implications or conjunctions are allowed. These formulae are also called clauses.
Such a clause is also written as the set

$$\{A, \neg B, C, \ldots, \neg E\}.$$

This means that the set-theoretic union of such sets corresponds again to a clause.

Note, that a clause now consists of atoms rather than constants, as it was the case of the resolution calculus for SL.

Definition 8.2 (Robinson's resolution for FOL)

The resolution calculus consists of two rules:

$$\textbf{(Res)}\frac{C_1 \cup \{A_1\} \quad C_2 \cup \{\neg A_2\}}{(C_1 \cup C_2)mgU(A_1, A_2)}$$

where $C_1 \cup \{A_1\}$ and $C_2 \cup \{A_2\}$ are assumed to be disjunct wrt the variables, and the factorization rule

$$\textbf{(Fac)}\ \frac{C_1 \cup \{L_1, L_2\}}{(C_1 \cup \{L_1\})mgU(L_1, L_2)}$$

Illustration of the resolution rule

Example 8.3

Consider the set $M = \{r(x) \vee \neg p(x), p(a), s(a)\}$ and the question $M \models \exists x(s(x) \wedge r(x))$?

Definition 8.4 (Resolution Calculus for FOL)

We define the resolution calculus $\mathrm{Robinson}^{FOL}_{\mathcal{L}^{res}} = \langle \emptyset, \{\mathbf{Res}, \mathbf{Fac}\}\rangle$ as follows. The underlying language is $\mathcal{L}^{res} \subseteq \mathcal{L}_{FOL}$ defined on Slide 627 together with the set of well-formed formulae $Fml^{Res}_{\mathcal{L}^{res}}$.

Thus there are **no axioms** and only **two inference rules**. The well-formed formulae are just clauses.

Question:

Is this calculus correct and complete?

Question:

Why do we need factorization?

Answer:

Consider

$$M = \{\{s(x_1), s(x_2)\}, \{\neg s(y_1), \neg s(y_2)\}\}$$

Resolving both clauses gives

$$\{s(x_1)\} \cup \{\neg s(y_1)\}$$

or variants of it.
Resolving this new clause with one in M only leads to variants of the respective clause in M.

Answer (continued):

$\square$ can not be derived (using resolution only).

Factorization solves the problem, we can deduce both $s(x)$ and $\neg s(y)$, and from there the empty clause.

Theorem 8.5 (Resolution is refutation complete)

Robinsons resolution calculus $\mathrm{Robinson}^{FOL}_{\mathcal{L}^{res}}$ *is* refutation complete*: Given an unsatisfiable set, the empty clause can eventually be derived using resolution and factorization.*

What to do if the formulae are not clauses?

In that case, we have to transform them into clauses of the required form.

Let us consider

Example 8.6

Let
$M = \{\exists x \exists y\, Q(x) \wedge \neg Q(y),\ \forall x \exists y\, P(x,y) \rightarrow \neg Q(y)\}$.
How to deal with the existential quantifiers?

The first formula is easy to transform: we just add two new constants c, c' and instantiate them for the variables. This leads to $Q(c) \wedge \neg Q(c')$, or the two clauses $Q(c)$ and $Q(c')$.

The second formula is more complicated, because there is no single y. We have to take into account, that the y usually depends on the chosen x. Therefore we introduce a function symbol $f(x)$: $\forall x\, P(x, f(x)) \rightarrow \neg Q(f(x))$. Then we have to transform the formula into disjunctive form: $\forall x\, \neg P(x, f(x)) \vee \neg Q(f(x))$.

Transformation into a set of clauses

Applying the technique of the last slide recursively, we can transform each set M in a language $\mathcal{L}$ into a set M' of clauses in an extended language $\mathcal{L}'$. It can be shown that the following holds.

Theorem 8.7

Let M be a set of first-order sentences in a language $\mathcal{L}$ and let M' be its transform. Then

M *is satisfiable if and only if* M' *is satisfiable.*

Because of the refutation completeness of the resolution calculus, this property is all we need.

8.3 Herbrand

The introduced relation $T \models \phi$ states that **each model** of T is also a model of ϕ. But because there are many models with very large universes the following question comes up: **can we restrict to particular models ?**

Theorem 8.8 (Löwenheim-Skolem)

$T \models \phi$ holds if and only if ϕ holds in all countable models of T.

By countable we mean the size of the universe of the model.

Quite often the universes of models (which we are interested in) consist exactly of the basic terms $Term_{\mathcal{L}}(\emptyset)$. This leads to the following notion:

Definition 8.9 (Herbrand model)

A model $\mathcal{A}$ is called Herbrand model with respect to a language if the universe of $\mathcal{A}$ consists exactly of $Term_{\mathcal{L}}(\emptyset)$ and the function symbols f_i^k are interpreted as follows:

$$f_i^{k\mathcal{A}}: \quad Term_{\mathcal{L}}(\emptyset) \times \ldots \times Term_{\mathcal{L}}(\emptyset) \to Term_{\mathcal{L}}(\emptyset);$$
$$(t_1, \ldots, t_k) \mapsto f_i^k(t_1, \ldots, t_k)$$

We write $T \models_{Herb} \phi$ if each Herbrand model of T is also a model of ϕ.

Theorem 8.10 (Reduction to Herbrand models)

If T is universal and ϕ existential, then the following holds:

$$T \models \phi \text{ if and only if } T \models_{Herb} \phi$$

Question:

Is $T \models_{Herb} \phi$ not much easier, because we have to consider only Herbrand models? Is it perhaps decidable?

No, truth in Herbrand models is highly undecidable.

The following theorem is the basic result for applying resolution. In a way it states that **FOL can be somehow reduced to SL**.

Theorem 8.11 (Herbrand)

Let T be universal and ϕ without quantifiers. Then:

$$T \models \exists\phi \quad \textit{if and only if} \quad \text{there are } t_1, \ldots, t_n \in Term_{\mathcal{L}}(\emptyset) \text{ with: } T \models \phi(t_1) \vee \ldots \vee \phi(t_n)$$

Or: Let M be a set of clauses of FOL (formulae in the form $P_1(t_1) \vee \neg P_2(t_2) \vee \ldots \vee P_n(t_n)$ with $t_i \in Term_{\mathcal{L}}(X)$). Then:

M is unsatisfiable
if and only if
there is a finite and unsatisfiable set M_{inst} of basic instances of M.

In automatic theorem proving we are always interested in the question

$$M \models \exists x_1, \ldots x_n \bigwedge_i \phi_i$$

Then

$$M \cup \{\neg \exists x_1, \ldots x_n \bigwedge_i \phi_i\}$$

is a set of clauses, which is unsatisfiable *if and only if* the relation above holds.

8.4 Variants of resolution

Our general goal is to derive an existentially quantified formula from a set of formulae:

$$M \vdash \exists\varphi.$$

To use resolution we must form $M \cup \{\neg\exists\varphi\}$ and **put it into the form of clauses**. This set is called **input**.

Instead of allowing arbitrary resolvents, we try to restrict the search space.

Example 8.12 (Unlimited Resolution)

Let $M := \{r(x) \vee \neg p(x),\ p(a),\ s(a)\}$ and $\square \leftarrow s(x) \wedge r(x)$ the query.

An unlimited resolution might look like this:

$$\dfrac{\dfrac{r(x) \vee \neg p(x) \quad p(a)}{r(a)} \qquad \dfrac{s(a) \quad \neg s(x) \vee \neg r(x)}{\neg r(a)}}{\square}$$

Input resolution: in each resolution step **one of the two parent clauses must be from the input**. In our example:

$$\dfrac{\dfrac{\dfrac{\neg s(x) \vee \neg r(x) \quad s(a)}{\neg r(a)} \qquad r(x) \vee \neg p(x)}{\neg p(a)} \qquad p(a)}{\square}$$

Linear resolution: in each resolution step **one of the two parent clauses must either be from the input or must be a successor of the other parent clause**.

Theorem 8.13 (Completeness of resolution variants)

Linear *resolution is* refutation complete.
Input *resolution is correct but* not refutation complete.

Idea:

Maybe input resolution is complete for a **restricted class of formulae**.

8.5 SLD resolution

Definition 8.14 (Horn clause)

A clause is called Horn clause if it contains at most one positive atom.

A Horn clause is called definite if it contains exactly one positive atom. It has the form

$$A(t) \leftarrow A_1(t_1), \ldots, A_n(t_n).$$

A Horn clause without positive atom is called query:

$$\square \leftarrow A_1(t_1), \ldots, A_n(t_n).$$

Theorem 8.15 (Input resolution for Horn clauses)

Input *resolution for* Horn clauses *is refutation complete.*

Definition 8.16 (SLD resolution wrt P and query Q)

SLD resolution with respect to a program P and the query Q is input resolution beginning with the query $\square \leftarrow A_1, \ldots, A_n$. Then one A_i is chosen and resolved with a clause of the program. A new query emerges, which will be treated as before. If the empty clause $\square \leftarrow$ can be derived then SLD resolution was successful and the instantiation of the variables is called **computed answer**.

Theorem 8.17 (Correctness of SLD resolution)

Let P be a definite program and Q a query. Then each answer calculated for P wrt Q is correct.

Question:

Is SLD completely instantiated?

Definition 8.18 (Computation rule)

A computation rule R is a function which assigns an atom $A_i \in \{A_1, \ldots, A_n\}$ to each query $\square \leftarrow A_1, \ldots, A_n$. This A_i is the chosen atom against which we will resolve in the next step.

Note:

PROLOG always uses the leftmost atom.

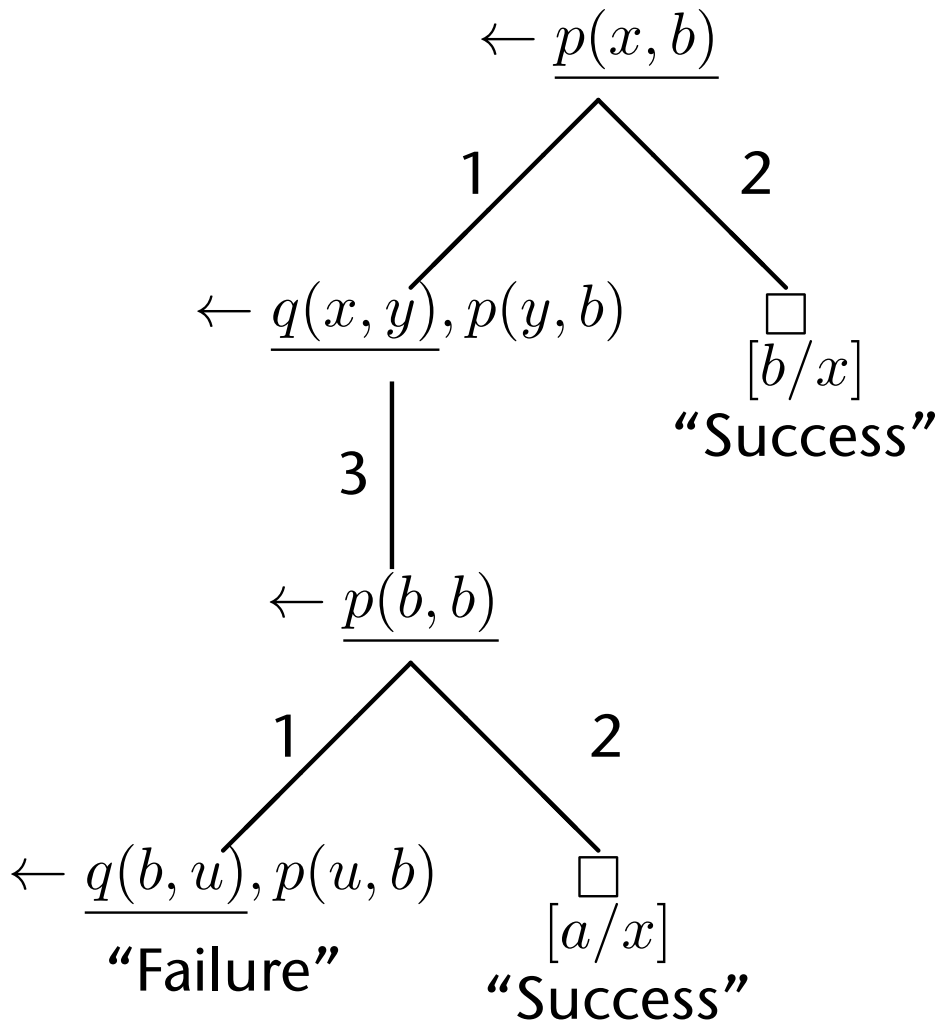
In the following, we are illustrating SLD resolution on the following program:

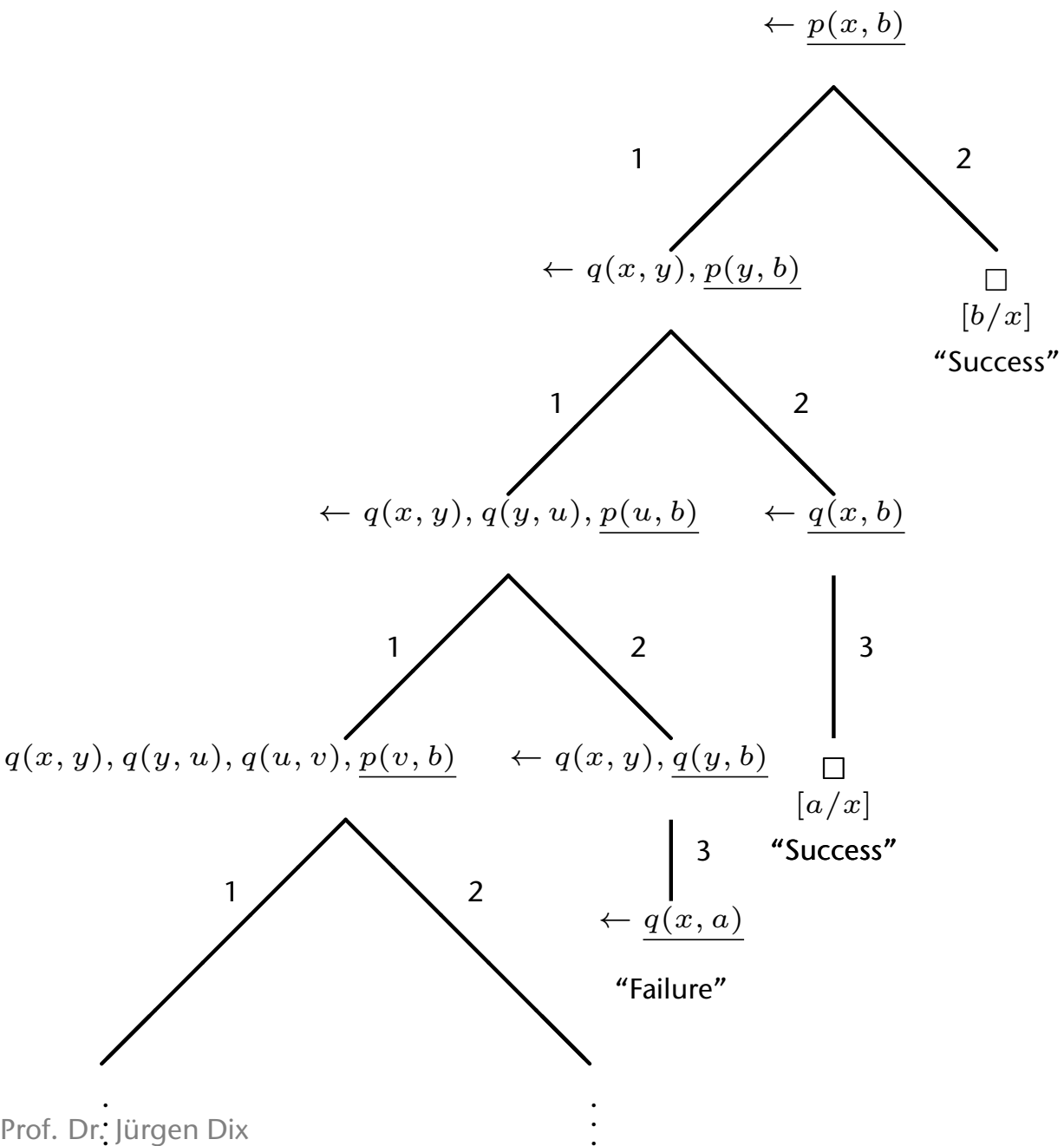
$$\begin{aligned} p(x,z) &\leftarrow q(x,y), p(y,z) \\ p(x,x) & \\ q(a,b) & \end{aligned}$$

We would like to know, for which instances for x, the fact $p(x,b)$ follows from the above theory.

Obviously, there are two solutions: $x = a$ and $x = b$ and these are the only ones.

We are now showing how to derive these solutions using SLD resolution.


$\leftarrow \underline{p(x,b)}$
1
2
$\leftarrow \underline{q(x,y)}, p(y,b)$
$\square$
$[b/x]$
"Success"
3
$\leftarrow \underline{p(b,b)}$
1
2
$\leftarrow \underline{q(b,u)}, p(u,b)$
$\square$
$[a/x]$
"Failure"
"Success"


$\leftarrow \underline{p(x,b)}$
1
2
$\leftarrow q(x,y), \underline{p(y,b)}$
$\square$
$[b/x]$
"Success"
1
2
$\leftarrow q(x,y), q(y,u), \underline{p(u,b)}$
$\leftarrow \underline{q(x,b)}$
1
2
3
$q(x,y), q(y,u), q(u,v), \underline{p(v,b)}$
$\leftarrow q(x,y), \underline{q(y,b)}$
$\square$
$[a/x]$
"Success"
3
1
2
$\leftarrow \underline{q(x,a)}$
"Failure"

A SLD tree may have three different kinds of branches:

1. **infinite ones**,
2. **branches ending with the empty clause** (and leading to an answer) and
3. **failing branches** (dead ends).

Theorem 8.19 (Independence of computation rule)

Let R be a computation rule and σ an answer calculated wrt R (i.e. there is a successful SLD resolution). Then there is also a successful SLD resolution for each other computation rule R' and the answer σ' belonging to R' is a variant of σ.

Theorem 8.20 (Completeness of SLD resolution)

Each correct answer substitution is subsumed through a calculated answer substitution. I.e.:

$$P \models \forall Q\Theta$$

implies

SLD computes an answer τ *with:* $\exists\sigma : Q\tau\sigma = Q\Theta$

Question:

How to find successful branches in a SLD tree?

Definition 8.21 (Search rule)

A search rule is a strategy to search for successful branches in SLD trees.

Note:

PROLOG uses depth-first-search.

A SLD resolution is determined by a **computation rule** and a **search rule**.

SLD trees for $P \cup \{Q\}$ are determined by the computation rule.

PROLOG is incomplete because of two reasons:

- depth-first-search
- incorrect unification (no occur check).

A third reason comes up if we also ask for finite and failed SLD resolutions:

- the computation rule must be **fair**, i.e. there must be a guarantee that each atom on the list of goals is **eventually** chosen.

Programming versus knowledge engineering

programming	knowledge engineering
choose language	choose logic
write program	define knowledge base
write compiler	implement calculus
run program	derive new facts

9. Planning

Content of this chapter (1):

Planning vs Search: While planning can be seen as a purely search problem, available search methods are not feasible. We need to use knowledge about the problem and create a newly planning agent.

STRIPS: The STRIPS approach is based on using logical formulae as a representation language for planning.

POP: We describe a sound and complete partial order planning (POP) algorithm. POP is an action-based planner.

Content of this chapter (2):

Conditional Planning: Often a plan cannot be completely constructed a priori, because the environment is dynamic. The we need sensing actions that have to be checked at run-time.

SHOP: We introduce HTN-planning, which is based on Hierarchical Task networks. HTN planners use domain knowledge and are more expressive than action-based planners. SHOP is one of the most efficient HTN planners.

Extensions: We briefly describe replanning and how to combine it with conditional planning.

9.1 Planning vs. Problem-Solving

Motivation:

problem-solving agent: The effects of a static sequence of actions are determined.

knowledge-based agent: Actions can be chosen.

We try to merge both into a planning agent.

function SIMPLE-PLANNING-AGENT(*percept*) **returns** an *action*
  static: *KB*, a knowledge base (includes action descriptions)
    p, a plan, initially *NoPlan*
    t, a counter, initially 0, indicating time
  local variables: *G*, a goal
    current, a current state description

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  current ← STATE-DESCRIPTION(*KB*, *t*)
  if *p* = *NoPlan* **then**
    G ← ASK(*KB*, MAKE-GOAL-QUERY(*t*))
    p ← IDEAL-PLANNER(*current*, *G*, *KB*)
  if *p* = *NoPlan* **or** *p* is empty **then** *action* ← *NoOp*
  else
    action ← FIRST(*p*)
    p ← REST(*p*)
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  t ← *t* + 1
  return *action*

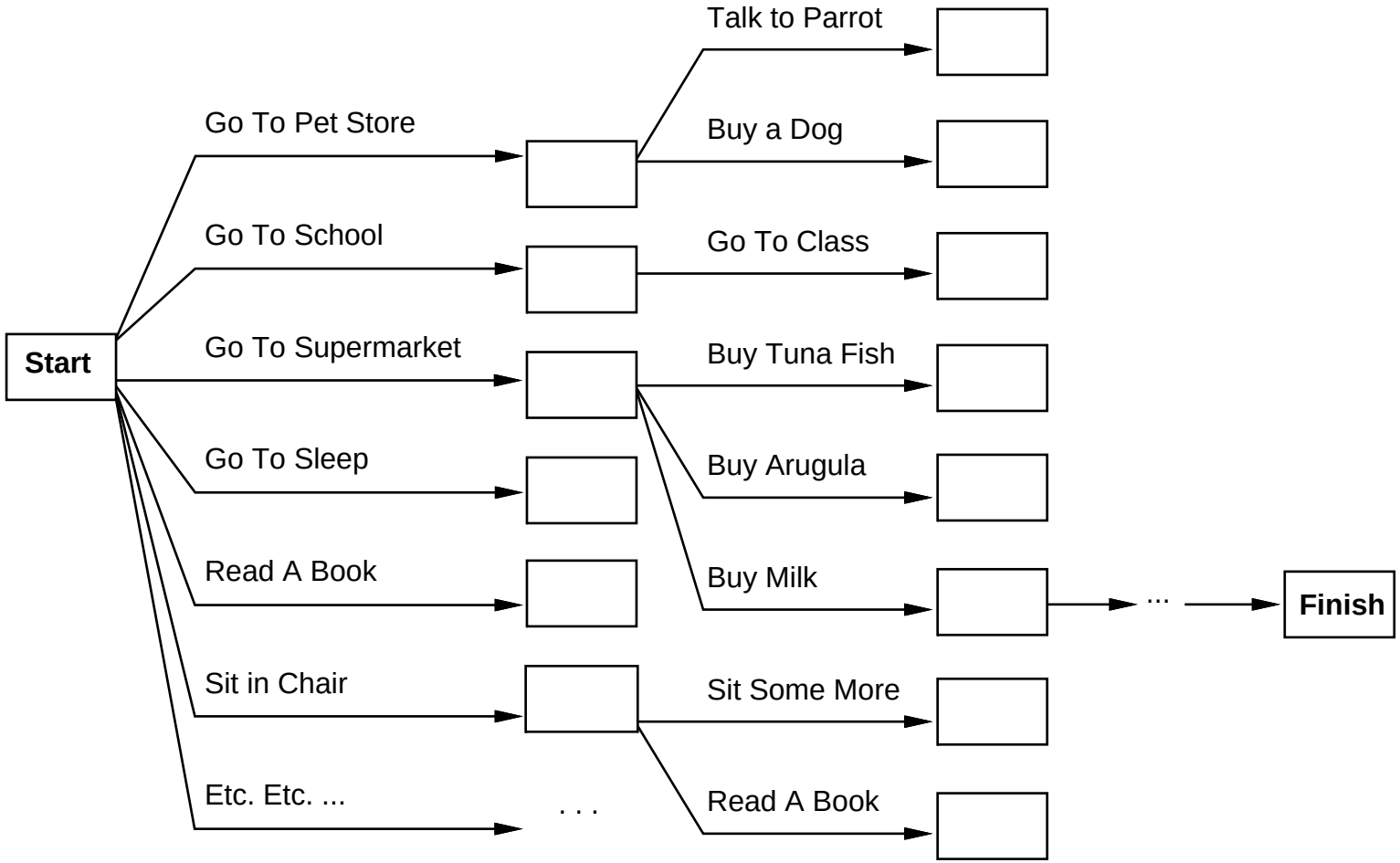
Example 9.1 (Running Example)

We want to drink freshly made banana shake and drill some holes into a wall at home.
Thus an agent needs to solve the following problem:

1. Get a quart of milk,
2. a bunch of bananas, and
3. a variable-speed cordless drill.

Question:

How does a problem-solving agent handle this?



Planning in the situation calculus:

Initial state: $At(Home, S_0)$, $\neg Have(Milk, S_0)$, $\neg Have(Bananas, S_0)$, $\neg Have(Drill, S_0)$.

Goal: $\exists s\ (At(Home, s) \wedge Have(Milk, s) \wedge Have(Bananas, s) \wedge Have(Drill, s))$.

Axioms: e.g. "buy milk":

$$\forall a, s\ Have(Milk, result(a, s)) \leftrightarrow (a = Buy(Milk) \wedge At(Supermarket, s)) \vee (Have(Milk, s) \wedge a \neq Drop(Milk))$$

We also need a term $Result'(l, s)$: the situation that would emerge when the sequence of actions l is executed in s.

$$\begin{array}{lll} \forall s & Result'([], s) & := \quad s \\ \forall a, p, s & Result'([a|p], s) & := \quad Result'(p, result(a, s)) \end{array}$$

The task is now to find a p with

$$At(Home, Result'(p, S_0)) \wedge$$
$$Have(Milk, Result'(p, S_0)) \wedge$$
$$Have(Bananas, Result'(p, S_0)) \wedge$$
$$Have(Drill, Result'(p, S_0)).$$

Problem solved!

This is the solution of our problem! We start a theorem prover and collect the answers!

Problems:

- $T \vdash \phi$ is only **semi-decidable** for FOL.
- If p is a plan then so are $[Empty\_Action|p]$ and $[A, A^{-1}|p]$.

Result:

We should not resort to a **general prover**, but to one which is **specially designed** for our domain. We also should restrict the language.

We should make clear the difference between

- shipping a goal to a planner,
- **asking a query to a theorem prover.**

In the first case we look for a plan so that after the execution of the plan the goal holds.

In the second case we ask if **the query can be made true wrt the KB:** $KB \models \exists x \phi(x)$.

The dynamics is in the terms. The logic itself is static.

9.2 STRIPS

STRIPS stands for **ST**anford **R**esearch **I**nstitute **P**roblem **S**olver.

states: conjunctions of function-free **ground-atoms (positive literals).**

goal: conjunctions of function-free literals

actions: STRIPS-operations consist of three components

1. *description*, name of the action
2. *precondition*, conjunction of atoms
3. *postcondition*, conjunction of literals

At(here), Path(here, there)

Go(there)

At(there), ¬At(here)

Example 9.2 (Air Cargo Transportation)

Three actions: $Load$, $Unload$, Fly. Two predicates

$In(c, p)$: *cargo c is inside plane p,*

$At(x, a)$: *object x is at airport a.*

Is cargo c at airport a when it is loaded in a plane p?

$$
\begin{aligned}
&Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \\
&\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \\
&\quad \land Airport(JFK) \land Airport(SFO)) \\
&Goal(At(C_1, JFK) \land At(C_2, SFO)) \\
&Action(Load(c, p, a), \\
&\quad \text{PRECOND: } At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } \neg At(c, a) \land In(c, p)) \\
&Action(Unload(c, p, a), \\
&\quad \text{PRECOND: } In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \\
&\quad \text{EFFECT: } At(c, a) \land \neg In(c, p)) \\
&Action(Fly(p, from, to), \\
&\quad \text{PRECOND: } At(p, from) \land Plane(p) \land Airport(from) \land Airport(to) \\
&\quad \text{EFFECT: } \neg At(p, from) \land At(p, to))
\end{aligned}
$$

Example 9.3 (Spare Tire)

Four actions:

1. $Remove(spare, trunk)$,
2. $Remove(flat, axle)$,
3. $PutOn$,
4. $LeaveOvernight$.

One predicate $At(x, a)$ meaning *object x is at location a.*

Is the following a STRIPS description?

$$
\begin{aligned}
&Init(At(Flat, Axle) \ \wedge \ At(Spare, Trunk)) \\
&Goal(At(Spare, Axle)) \\
&Action(Remove(Spare, Trunk), \\
&\quad \text{PRECOND: } At(Spare, Trunk) \\
&\quad \text{EFFECT: } \neg\, At(Spare, Trunk) \ \wedge \ At(Spare, Ground)) \\
&Action(Remove(Flat, Axle), \\
&\quad \text{PRECOND: } At(Flat, Axle) \\
&\quad \text{EFFECT: } \neg\, At(Flat, Axle) \ \wedge \ At(Flat, Ground)) \\
&Action(PutOn(Spare, \ Axle), \\
&\quad \text{PRECOND: } At(Spare, Ground) \ \wedge \ \neg\, At(Flat, Axle) \\
&\quad \text{EFFECT: } \neg\, At(Spare, Ground) \ \wedge \ At(Spare, Axle)) \\
&Action(LeaveOvernight, \\
&\quad \text{PRECOND:} \\
&\quad \text{EFFECT: } \neg\, At(Spare, Ground) \ \wedge \ \neg\, At(Spare, Axle) \ \wedge \ \neg\, At(Spare, Trunk) \\
&\qquad\qquad \wedge \neg\, At(Flat, Ground) \ \wedge \ \neg\, At(Flat, Axle))
\end{aligned}
$$

Example 9.4 (Blocks World)

- One action: $Move$. $Move(b, x, y)$ moves the block b from x to y if both b and y are clear.
- One predicate $On(b, x)$ meaning *block b is on x (x can be another block or the table)*.

How to formulate that a block is clear?

"b is clear": $\forall x \neg On(x, b)$.

Not allowed in STRIPS.

Therefore we introduce another predicate $Clear(y)$.

What about $Move(b, x, y)$ defined by
Precondition: $On(b, x) \wedge Clear(b) \wedge Clear(y)$ and
Effect:
$On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)$**?**

$$
\begin{aligned}
&Init(On(A, Table) \wedge On(B, Table) \wedge On(C, Table) \\
&\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \\
&\quad \wedge Clear(A) \wedge Clear(B) \wedge Clear(C)) \\
&Goal(On(A, B) \wedge On(B, C)) \\
&Action(Move(b, x, y), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge \\
&\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y), \\
&\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) \\
&Action(MoveToTable(b, x), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x), \\
&\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))
\end{aligned}
$$

ADL (Action Description Language) and its many derivatives are extensions of STRIPS:

States: both positive and negative literals in states.

OWA: Open World assumption (not CWA)

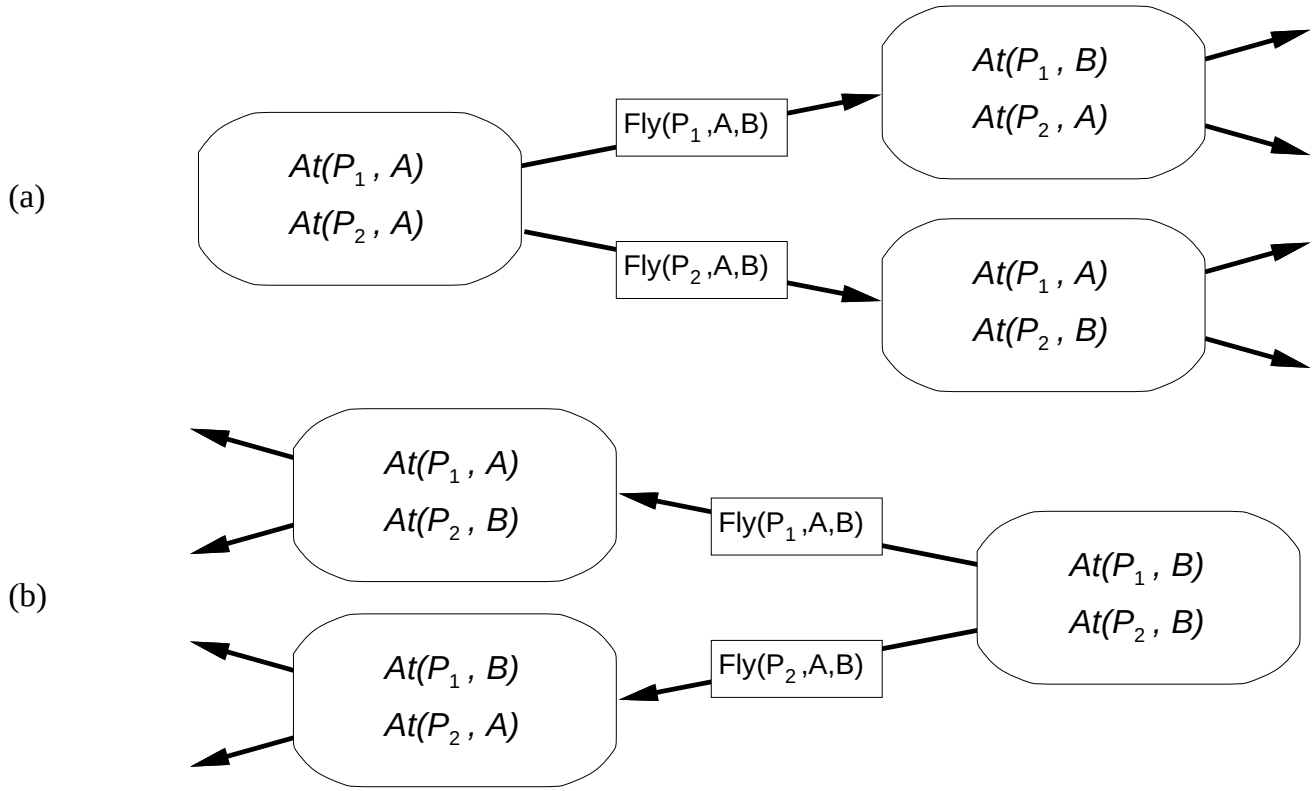
Effects: Add all positive and negative literals, and delete their negations.

Quantification: Quantified variables in goals are allowed.

Goals: disjunction and negation also allowed.

when P: Conditional effects allowed.

(a) Progression (forward) and (b) Regression (backward) state-space search



At(here), Path(here, there)

Go(there)

At(there), ¬At(here)

Definition 9.5 (Applicable Operator)

An operator Op is applicable in a state s if there is some way to instantiate the variables in Op so that every one of the preconditions of Op is true in s: $Precond(Op) \subseteq s$.

In the resulting state, all the positive literals in Effect(Op) hold, as do all the literals that held in s, except for those that are negative literals in Effect(Op).

Frame problem is handled implicitly: literals not mentioned in effects remain unchanged (persistence).
Effect is sometimes split into **add** and **delete** lists.
Up to now we can consider this as being **problem-solving**.
We use STRIPS as an representation-formalism and search a solution-path:

- **nodes in the search-tree $\approx$ situations**
- **solution paths $\approx$ plans.**

Idea:

Perform a search in the space of all plans!

Begin with a partial plan and extend it successively.

Therefore we need operators which operate on plans. We distinguish two types:

refinement-op: constraints are attached to a plan. Then a plan represents the set of all complete plans (analogously $Cn(T)$ for $MOD(T)$).

modification-op: all others.

Question:

How do we represent plans?

Answer:

We have to consider two things:

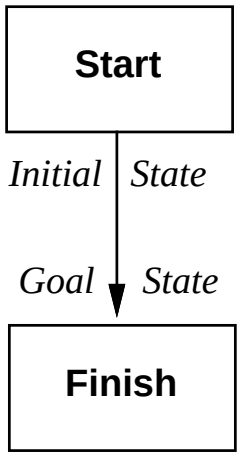
- **instantiation of variables:** instantiate only if necessary, i.e. always choose the **mgU**
- **partial order:** refrain from the exact ordering (reduces the search-space)

Definition 9.6 (Plan)

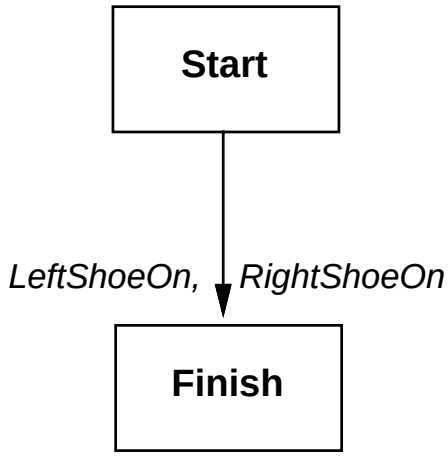
A plan is formally defined as a data structure consisting of the following four components:

- A set of **plan steps**. Each step is one of the operators for the problem.
- A set of **step ordering constraints**. Each one of the form $S_i \prec S_j$: S_i has to be executed before S_j.
- A set of **variable binding constants**. Each one of the form $v = x$: v is a variable in some step, and x is either a constant or another variable.
- A set of **causal links**. Each one of the form $S_i \xrightarrow{c} S_j$: S_i achieves c for S_j.

The initial plan consists of two steps, called *START* and *FINISH*.



(a)

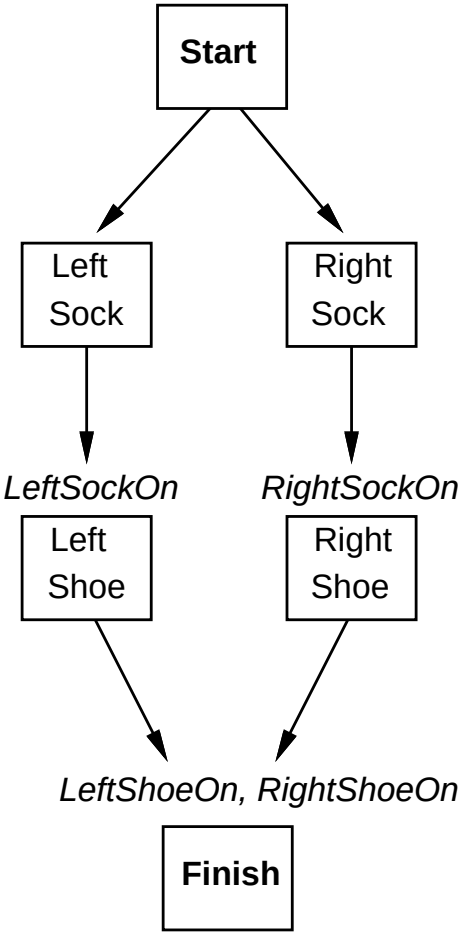


(b)

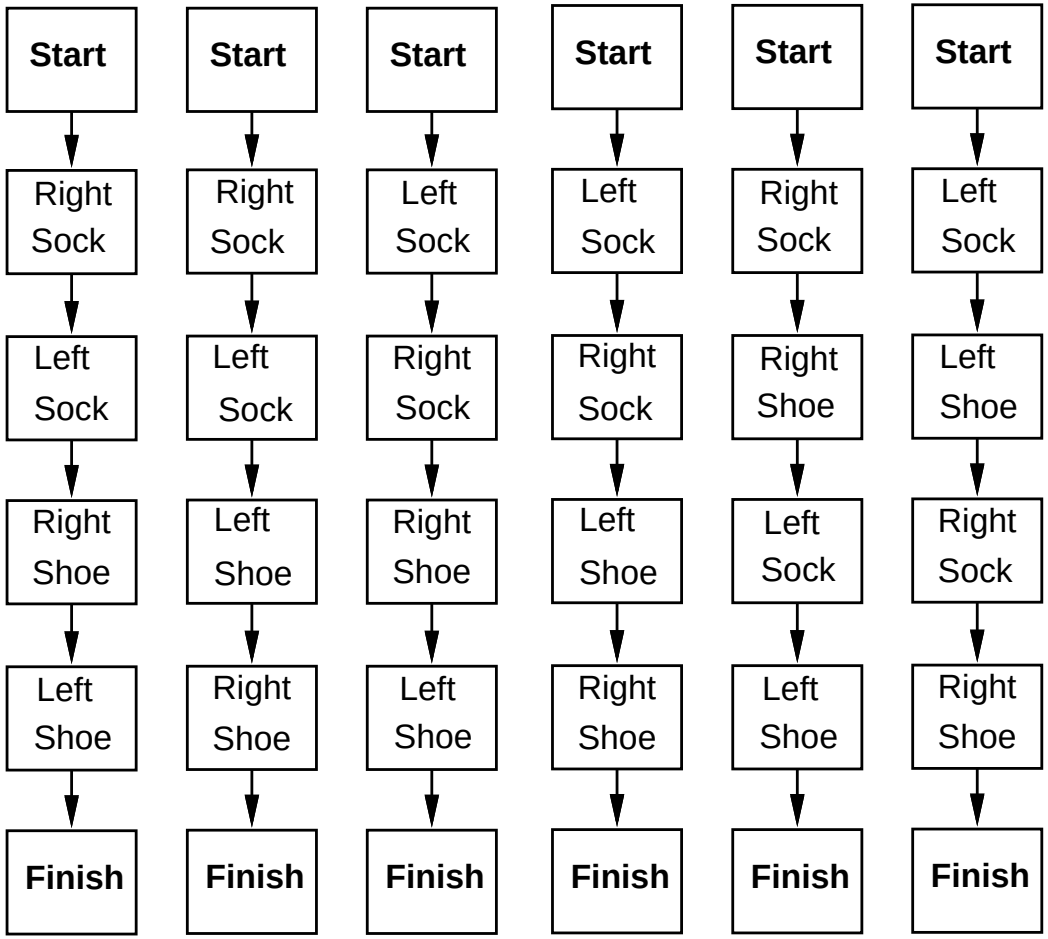
The initial plan of a problem is:

Plan(
$$\begin{aligned} Steps: &\quad \{S_1 : START, S_2 : FINISH\} \\ Orderings: &\quad \{S_1 \prec S_2\} \\ Bindings: &\quad \emptyset \\ Links: &\quad \emptyset \end{aligned}$$
)

Partial Order Plan:



Total Order Plans:



Question:

What is a solution?

Answer:

Considering only **fully instantiated, linearly ordered** plans: checking is easy.

But our case is far more complicated:

Definition 9.7 (Solution of a Plan)

A solution is a complete and consistent plan.

complete: each precondition of each step is achieved by some other step,

consistent: there are no contradictions in the ordering or binding constraints.

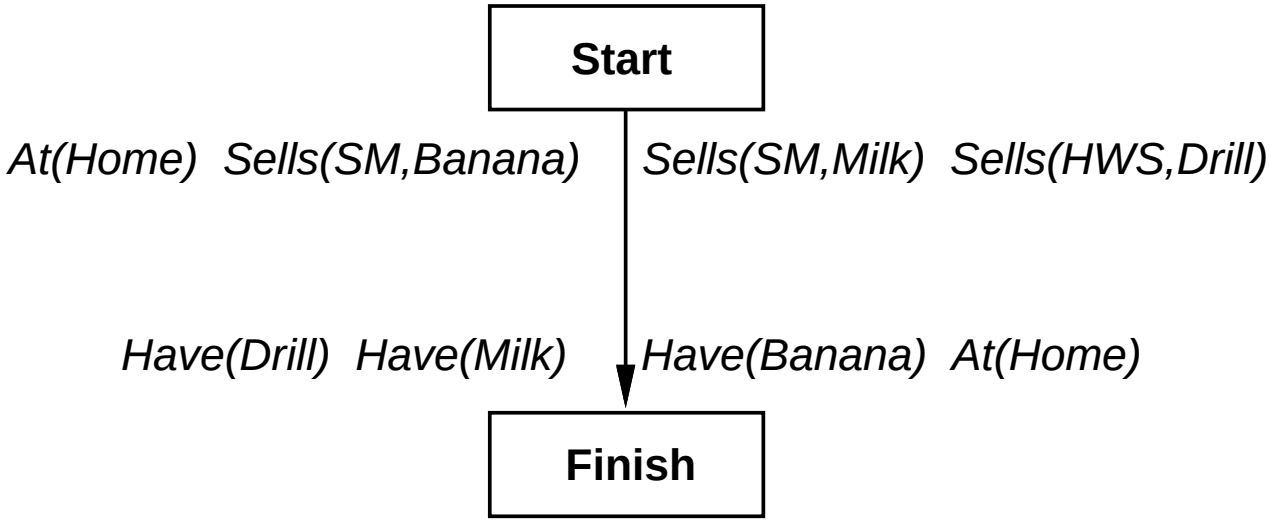
More precisely:

- "S_i **achieves** c **for** S_j" means
  - $c \in \text{Precondition}(S_j)$,
  - $c \in \text{Effect}(S_i)$,
  - $S_i \prec S_j$,
  - $\nexists S_k : \neg c \in \text{Effect}(S_k)$ with $S_i \prec S_k \prec S_j$ in any linearization of the plan
- "**no contradictions**" means
  - neither ($S_i \prec S_j$ and $S_j \prec S_i$) nor ($v = A$ and $v = B$ for different constants A, B).

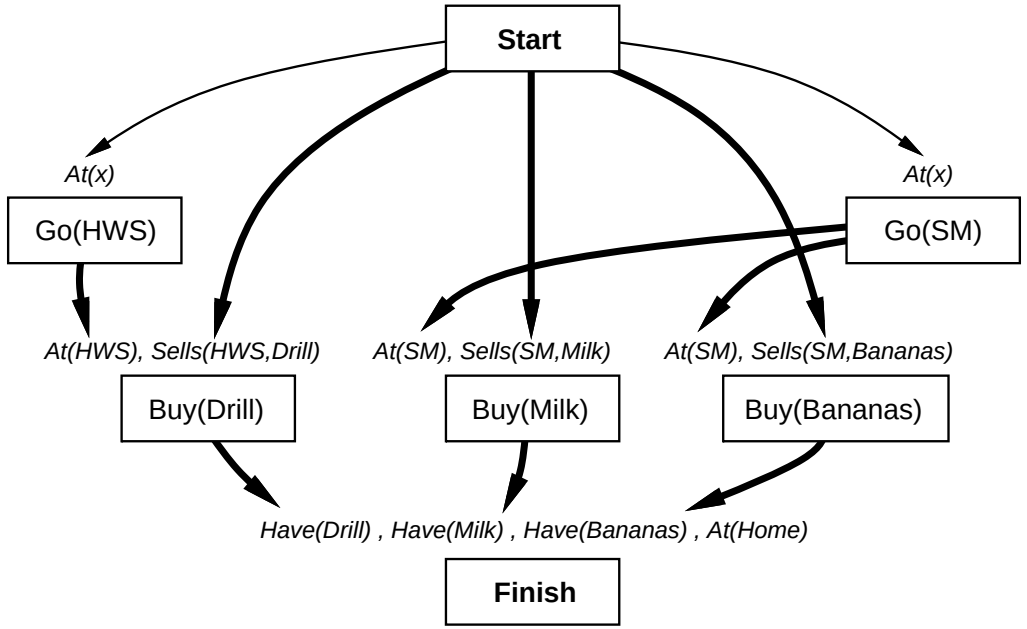
Note: these propositions may be derivable, because $\prec$ and $=$ are transitive.

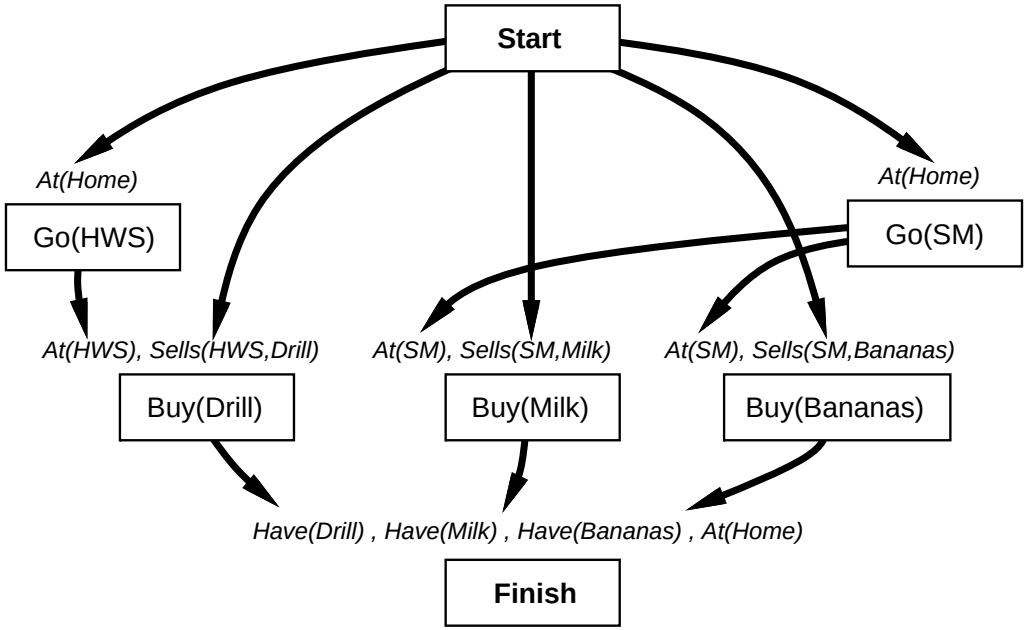
9.3 Partial-Order Planning

We consider the banana-milk example. The operators are Buy and Go.

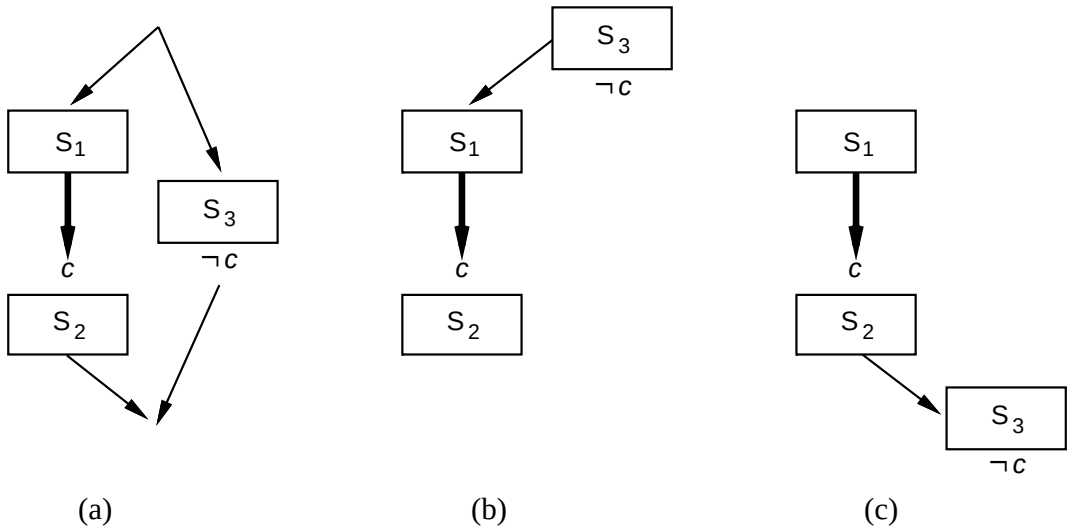


Causal links are protected!


Start
At(x)
At(x)
Go(HWS)
Go(SM)
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill) , Have(Milk) , Have(Bananas) , At(Home)
Finish


Start
At(Home)
At(Home)
Go(HWS)
Go(SM)
At(HWS), Sells(HWS,Drill)
At(SM), Sells(SM,Milk)
At(SM), Sells(SM,Bananas)
Buy(Drill)
Buy(Milk)
Buy(Bananas)
Have(Drill) , Have(Milk) , Have(Bananas) , At(Home)
Finish

The last partial plan **cannot be extended**. How do we determine this? By determining that a **causal link is threatened and the threat cannot be resolved**.

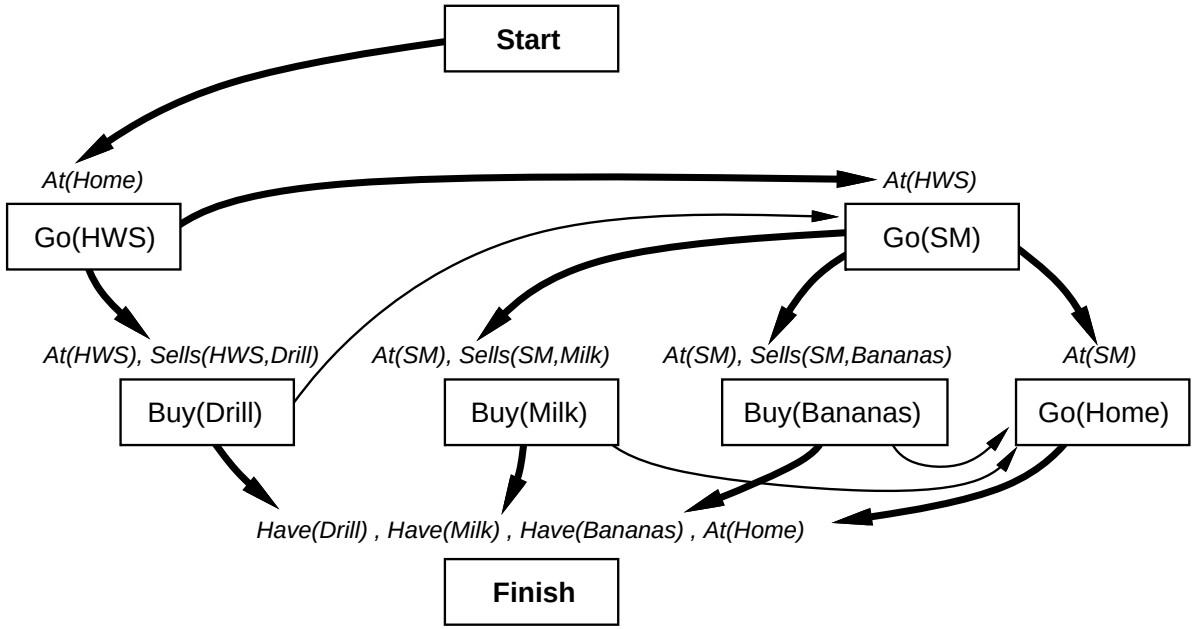


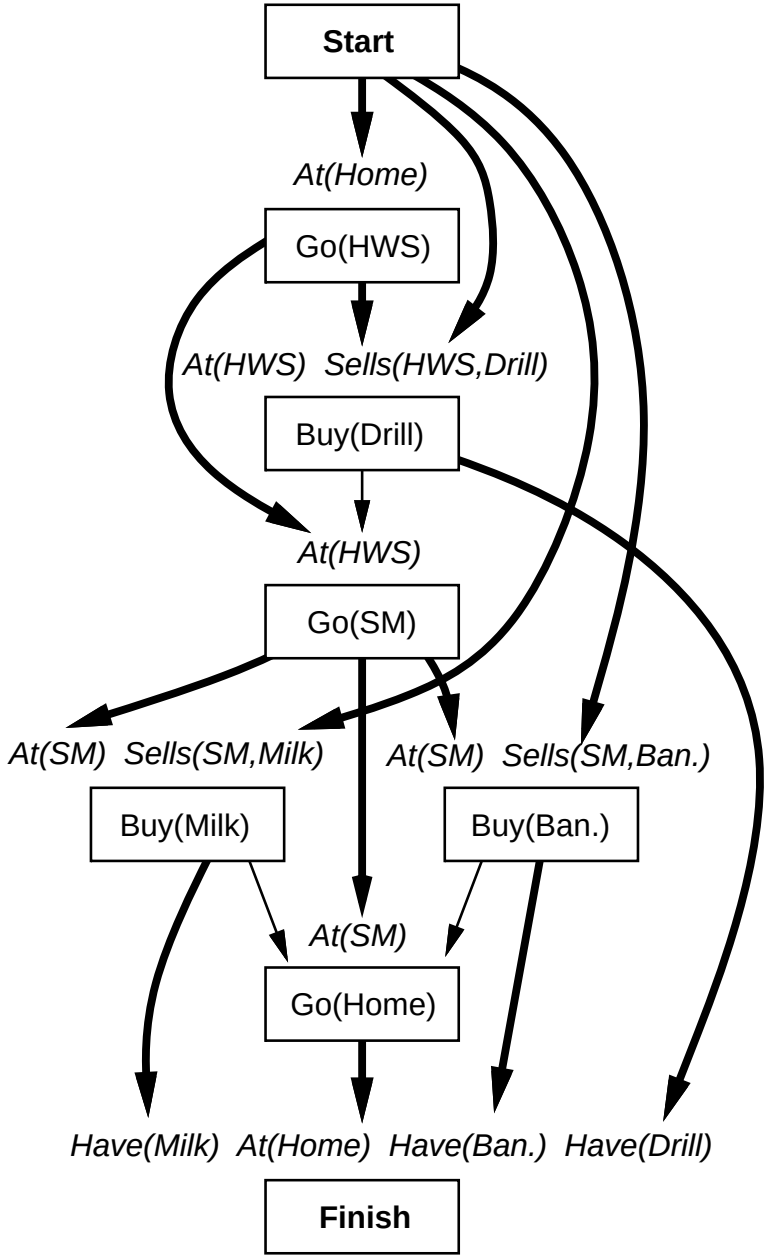
The way to resolve the threat is to add ordering constraints (this will not always work!).

Question:

We have to introduce a Go-step in order to ensure the last precondition. But how can we ensure the precondition of the Go-step?

Now there are a lot of threats and many of them are unresolvable. This leads to




Start
At(Home)
Go(HWS)
At(HWS) Sells(HWS,Drill)
Buy(Drill)
At(HWS)
Go(SM)
At(SM) Sells(SM,Milk)
At(SM) Sells(SM,Ban.)
Buy(Milk)
Buy(Ban.)
At(SM)
Go(Home)
Have(Milk) At(Home) Have(Ban.) Have(Drill)
Finish

This leads to the following algorithm

- In each round the plan is extended in order to **ensure the precondition of a step**. This is done by choosing an appropriate operator.
- The respective **causal link is introduced**. Threats are resolved through ordering constraints (two cases: the new step threatens existing ones or the existing ones threaten the new one).
- If there is **no operator** or the threat **cannot** be resolved then perform **backtracking**.

Theorem 9.8 (POP)

POP is complete and correct.

function POP(*initial*, *goal*, *operators*) **returns** *plan*

plan ← MAKE-MINIMAL-PLAN(*initial*, *goal*)
loop do
  if SOLUTION?(*plan*) **then return** *plan*
  S_{need}, c ← SELECT-SUBGOAL(*plan*)
  CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)
  RESOLVE-THREATS(*plan*)
end

---

function SELECT-SUBGOAL(*plan*) **returns** S_{need}, c

pick a plan step S_{need} from STEPS(*plan*)
  with a precondition c that has not been achieved
return S_{need}, c

---

procedure CHOOSE-OPERATOR(*plan*, *operators*, S_{need}, c)

choose a step S_{add} from *operators* or STEPS(*plan*) that has c as an effect
if there is no such step **then fail**
add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
if S_{add} is a newly added step from *operators* **then**
  add S_{add} to STEPS(*plan*)
  add *Start* $\prec S_{add} \prec$ *Finish* to ORDERINGS(*plan*)

---

procedure RESOLVE-THREATS(*plan*)

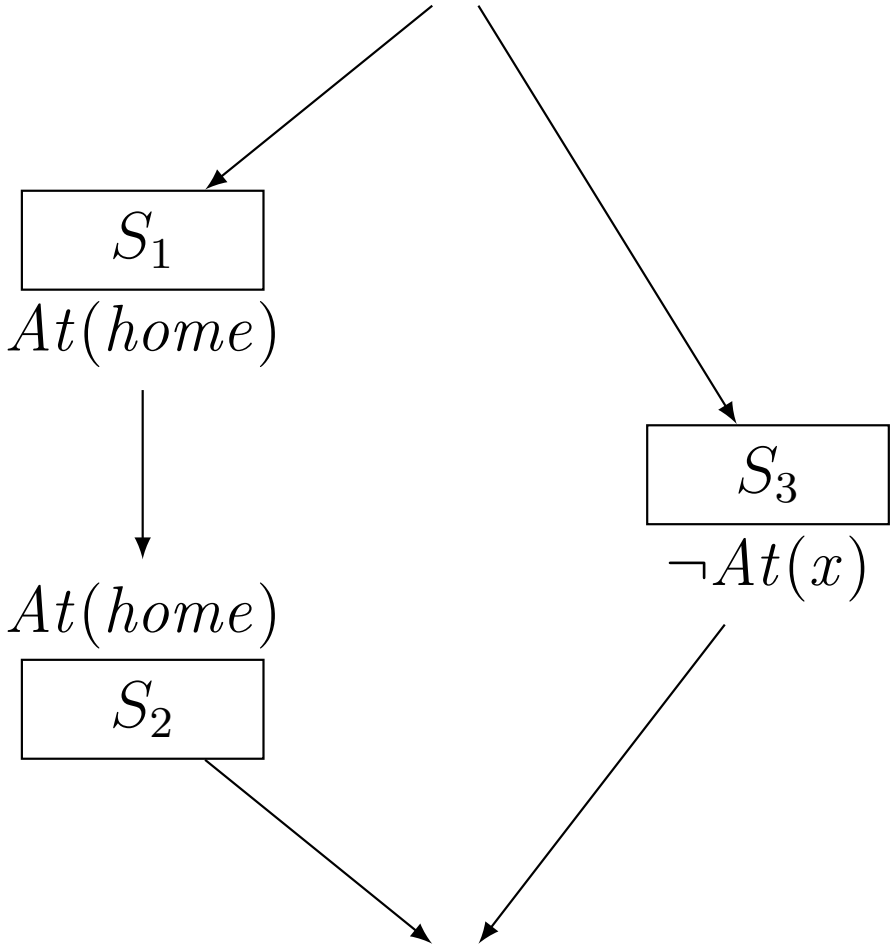
for each S_{threat} that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(*plan*) **do**
  choose either
    Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
    Demotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
  if not CONSISTENT(*plan*) **then fail**
end

So far we did not consider variable-substitutions.

Question:

Suppose S_1 ensures the $At(home)$ precondition of a step S_2 and there is a concurrent step S_3 with the postcondition $\neg At(x)$. Is this a threat?

We call such a threat possible and ignore it for the time being, but keep it in mind. If x is later instantiated with $home$ then a real threat is there which has to be resolved.



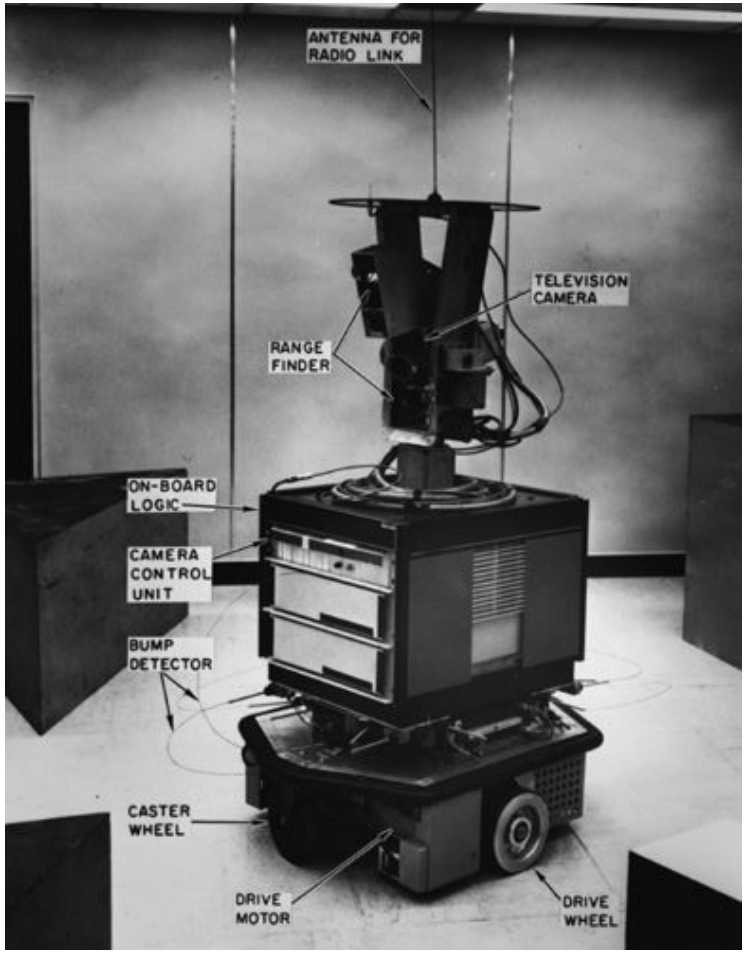
procedure CHOOSE-OPERATOR($plan, operators, S_{need}, c$)

choose a step S_{add} from *operators* or STEPS(*plan*) that has c_{add} as an effect
such that u = UNIFY(c, c_{add}, BINDINGS(*plan*))
if there is no such step
then fail
add u to BINDINGS(*plan*)
add $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
add $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
if S_{add} is a newly added step from *operators* **then**
add S_{add} to STEPS(*plan*)
add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

---

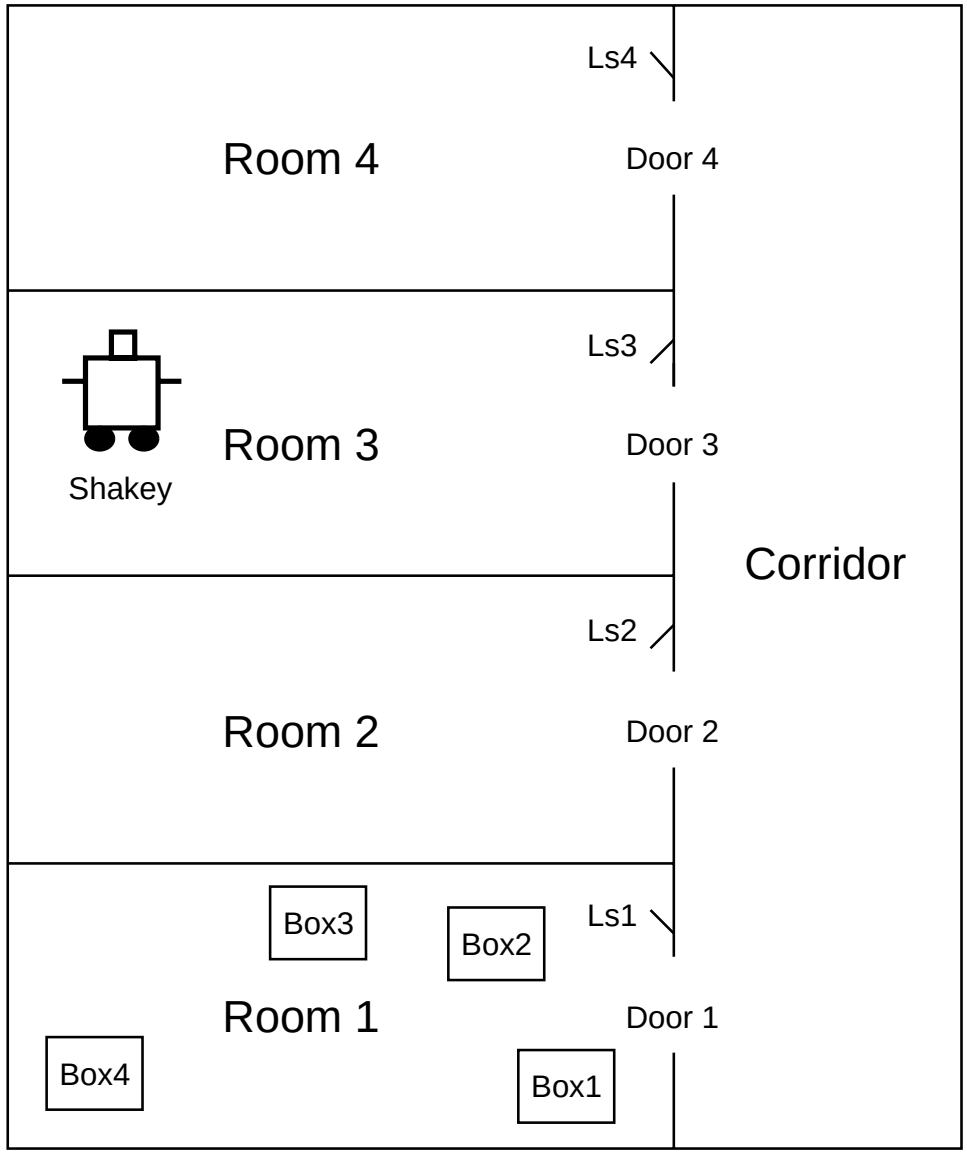
procedure RESOLVE-THREATS(*plan*)

for each $S_i \xrightarrow{c} S_j$ **in** LINKS(*plan*) **do**
for each S_{threat} **in** STEPS(*plan*) **do**
for each c' **in** EFFECT(S_{threat}) **do**
if SUBST(BINDINGS(*plan*), c) = SUBST(BINDINGS(*plan*), $\neg c'$) **then**
choose either
Promotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
Demotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
if not CONSISTENT(*plan*)
then fail
end
end
end

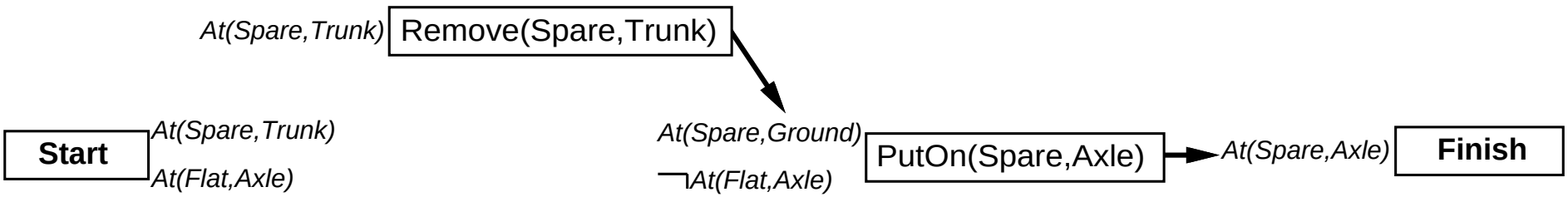


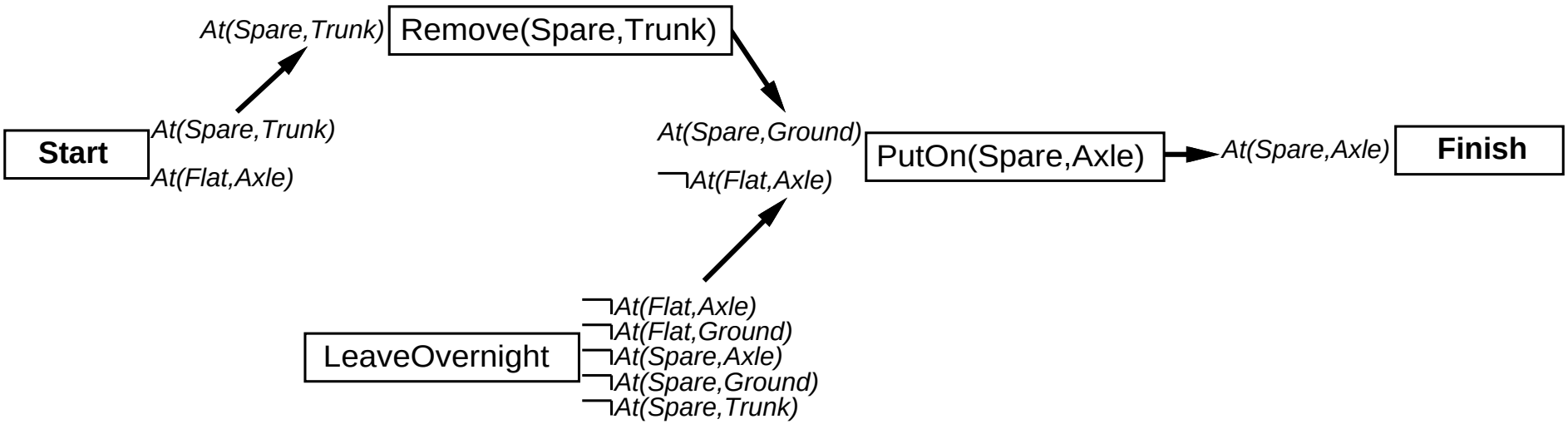
STRIPS was originally designed for SHAKEY, a small and mobile robot. SHAKEY is described through 6 Operators: $Go(x)$, $Push(b, x, y)$, $Climb(b)$, $Down(b)$, $Turn\_On(ls)$, $Turn\_Off(ls)$.

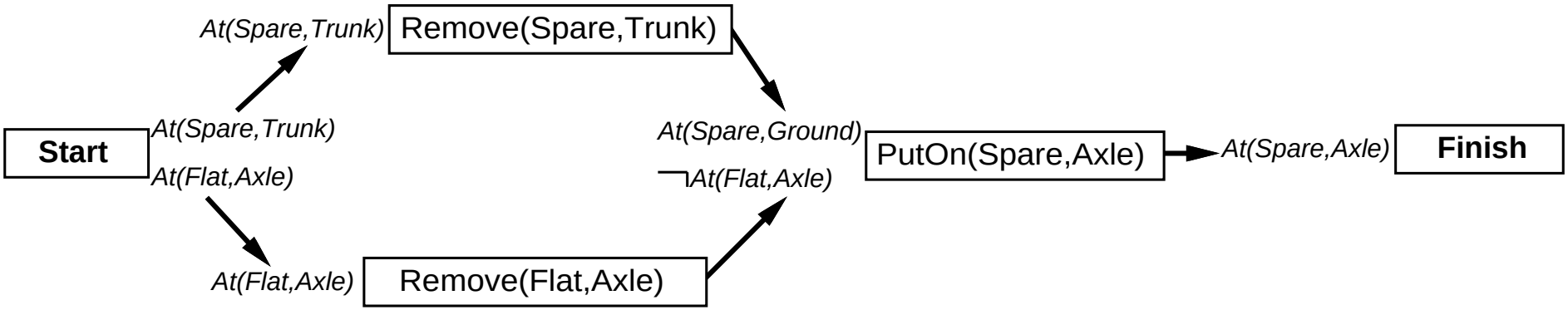
To turn the lights on or off SHAKEY has to stand on a box. $On(Shakey, floor)$ is a precondition of the Go-action so that SHAKEY does not fall off.


Ls4
Room 4
Door 4
Ls3
Shakey
Room 3
Door 3
Corridor
Ls2
Room 2
Door 2
Box3
Box2
Ls1
Room 1
Door 1
Box4
Box1

POP for ADL


At(Spare,Trunk)
Remove(Spare,Trunk)
Start
At(Spare,Trunk)
At(Flat,Axle)
At(Spare,Ground)
¬At(Flat,Axle)
PutOn(Spare,Axle)
At(Spare,Axle)
Finish


At(Spare,Trunk)
Remove(Spare,Trunk)
Start
At(Spare,Trunk)
At(Flat,Axle)
At(Spare,Ground)
¬At(Flat,Axle)
PutOn(Spare,Axle)
At(Spare,Axle)
Finish
¬At(Flat,Axle)
¬At(Flat,Ground)
¬At(Spare,Axle)
¬At(Spare,Ground)
¬At(Spare,Trunk)
LeaveOvernight


At(Spare,Trunk)
Remove(Spare,Trunk)
Start
At(Spare,Trunk)
At(Flat,Axle)
At(Spare,Ground)
¬At(Flat,Axle)
PutOn(Spare,Axle)
At(Spare,Axle)
Finish
At(Flat,Axle)
Remove(Flat,Axle)

9.4 Conditional Planning

Question:

What to do if the world is not fully accessible?
Where to get milk? Milk-price has doubled and we do not have enough money.

Idea:

Introduce new sensing actions to query certain conditions and to react accordingly. (How much does milk cost?)

A flat tire

Op($Action: Remove(x),$
$Prec: On(x),$
Effect : $Off(x) \wedge Clear\_Hub \wedge \neg On(x)$
)

Op($Action: Put\_on(x),$
$Prec: Off(x) \wedge Clear\_Hub,$
Effect : $On(x) \wedge \neg Clear\_Hub \wedge \neg Off(x)$
)

Op($Action: Inflate(x),$
$Prec: Intact(x) \wedge Flat(x),$
Effect : $Inflated(x) \wedge \neg Flat(x)$
)

goal: $On(x) \wedge Inflated(x)$,
initial state: $Inflated(Spare) \wedge Intact(Spare) \wedge Off(Spare) \wedge On(Tire_1) \wedge Flat(Tire_1)$.

Question:

What does POP deliver?

Answer:

Because of $Intact(Tire_1)$ not being present POP delivers the plan

$$[Remove(Tire_1), Put\_on(Spare)].$$

Question:

Would you also do it like that?

Answer:

A better way would be a conditional plan:

If $Intact(Tire_1)$ then: $Inflate(Tire_1)$.

Therefore we have to allow conditional steps in the phase of plan-building:

Definition 9.9 (Conditional Step)

A conditional step in a plan has the form

$$\text{If}(\{Condition\}, \{Then\_Part\}, \{Else\_Part\})$$

The respective planning-agent looks like this:

function CONDITIONAL-PLANNING-AGENT(*percept*) **returns** an *action*
  static: *KB*, a knowledge base (includes action descriptions)
    p, a plan, initially *NoPlan*
    t, a counter, initially 0, indicating time
    G, a goal

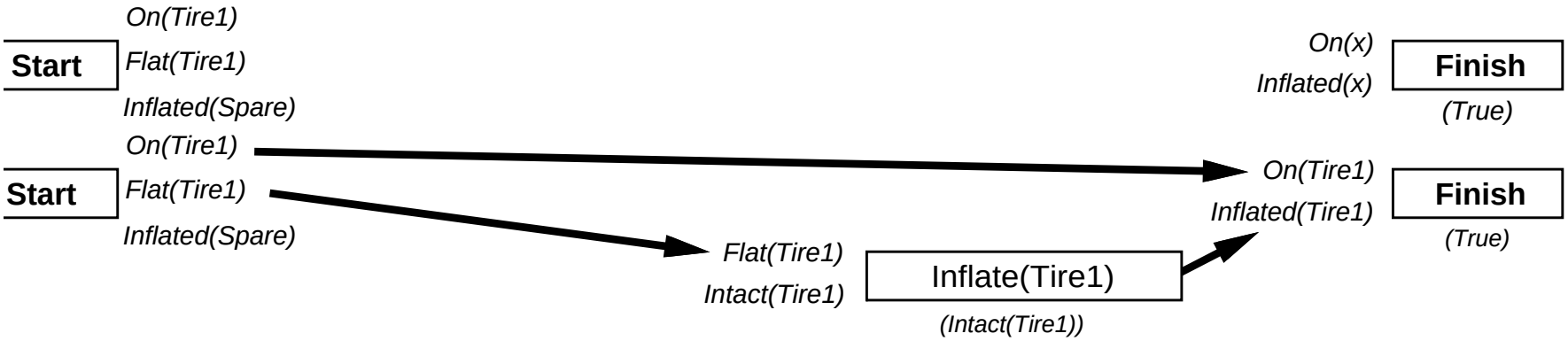
  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  current ← STATE-DESCRIPTION(*KB*, *t*)
  if *p* = *NoPlan* **then** *p* ← CPOP(*current*, *G*, *KB*)
  if *p* = *NoPlan* **or** *p* is empty **then** *action* ← *NoOp*
  else
    action ← FIRST(*p*)
    while CONDITIONAL?(*action*) **do**
      if ASK(*KB*, CONDITION-PART[*action*]) **then** *p* ← APPEND(THEN-PART[*action*], REST(*p*))
      else *p* ← APPEND(ELSE-PART[*action*], REST(*p*))
      action ← FIRST(*p*)
    end
    p ← REST(*p*)
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  t ← *t* + 1
  return *action*

The agent has to know if the respective if-condition holds or not when executing the plan (at runtime). Therefore we introduce new checking actions:

Op($Action:$ $Check\_Tire(x)$,
$Prec:$ $Tire(x)$,
$Effect:$ "We know if $Intact(x)$ holds or not."
)

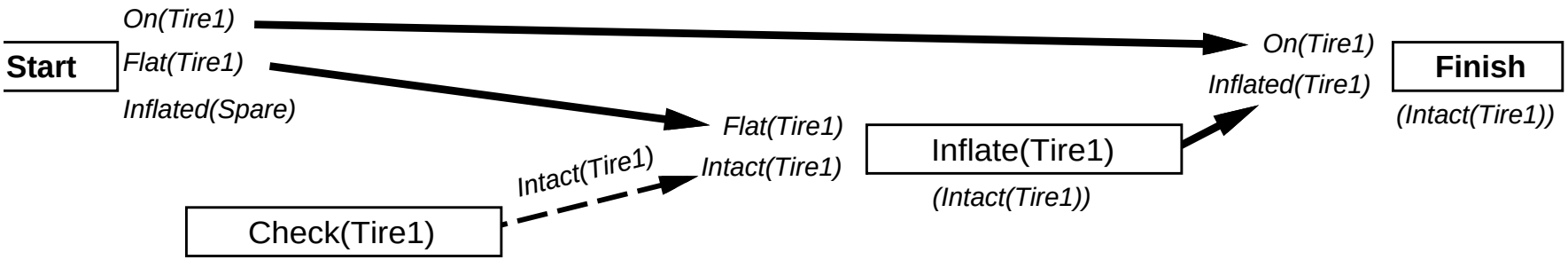
Definition 9.10 (Context)

We associate a context with each step: the set of conditions which have to hold before executing a step.

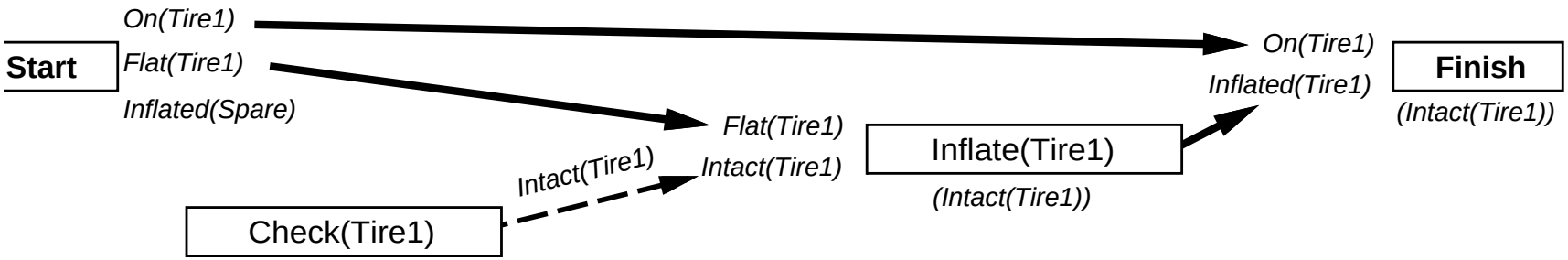

On(Tire1)
Start
Flat(Tire1)
Inflated(Spare)
On(Tire1)
Start
Flat(Tire1)
Inflated(Spare)
Flat(Tire1)
Intact(Tire1)
Inflate(Tire1)
(Intact(Tire1))
On(x)
Inflated(x)
Finish
(True)
On(Tire1)
Inflated(Tire1)
Finish
(True)

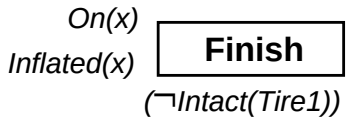
Here POP would backtrack, because $Intact(Tire_1)$ cannot be shown.

Hence we introduce a new type of links: conditional links.


On(Tire1)
Start
Flat(Tire1)
Inflated(Spare)
On(Tire1)
Inflated(Tire1)
Finish
(Intact(Tire1))
Flat(Tire1)
Intact(Tire1)
Inflate(Tire1)
(Intact(Tire1))
Intact(Tire1)
Check(Tire1)

We have to cover each case:





Question:

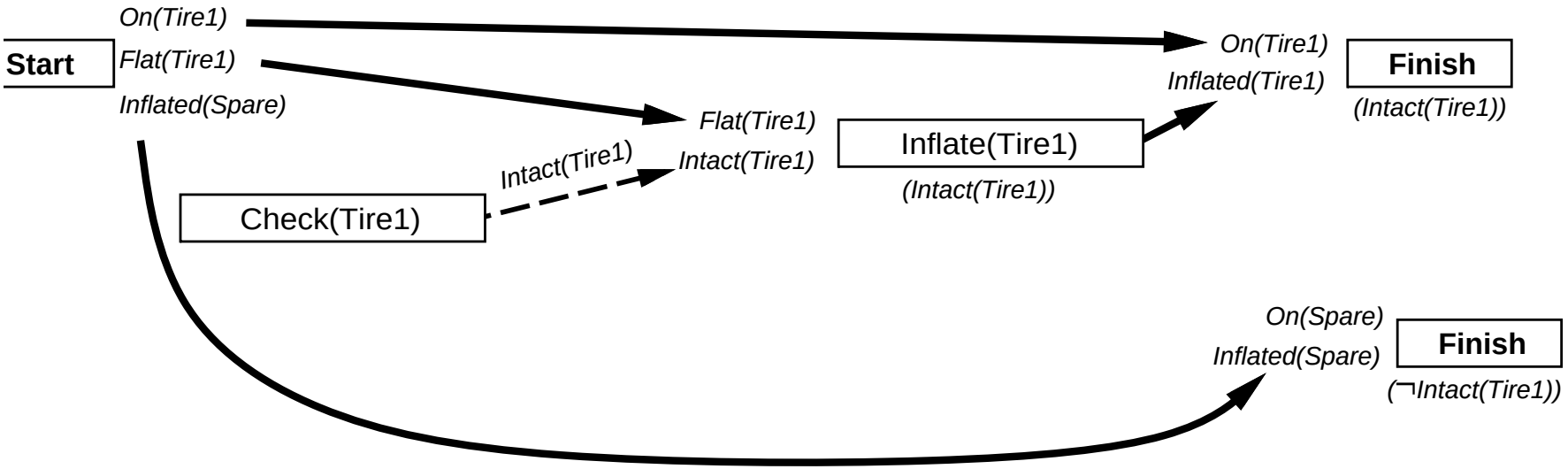
What if new contexts emerge in the second case?

Answer:

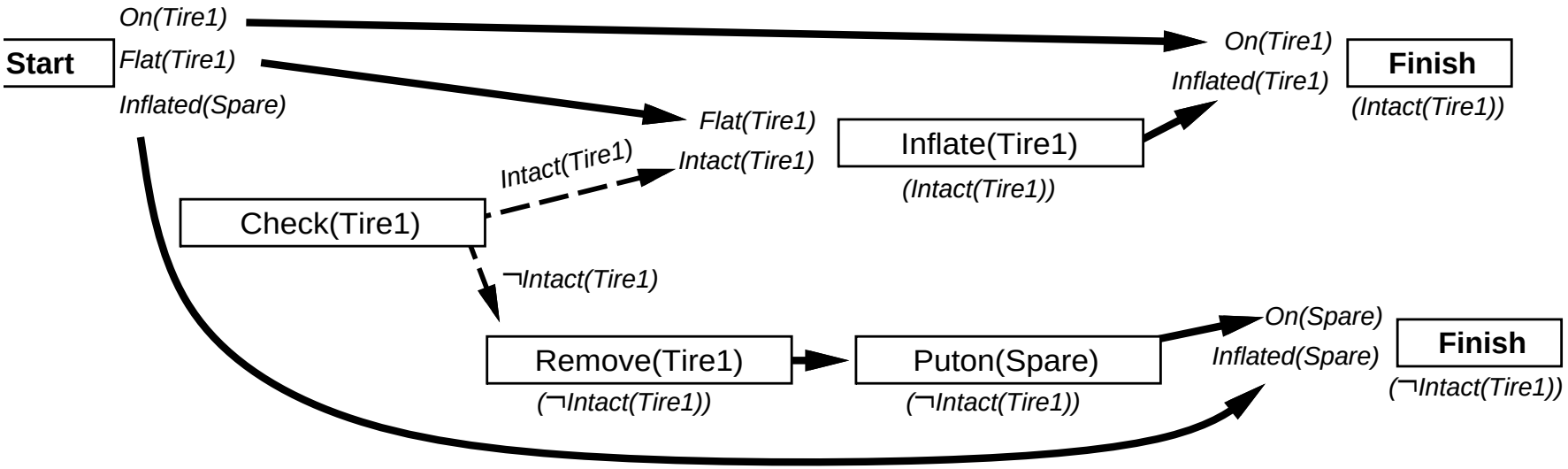
Then we have to introduce new copies of the FINISH step: There must be a complete distinction of cases in the end.

Question:

How can we make $Inflated(x)$ true in the FINISH step? Adding the step $Inflate(Tire_1)$ would not make sense, because the preconditions are inconsistent in combination with the context.



At last $On(Spare)$.



Attention:

At first "True" is the context of $Remove(Tire_1), Put\_on(Spare)$. **But $Remove(Tire_1)$ threatens the $On(Tire_1)$ step (precondition of the first Finish).**

We can resolve this threat by making the contexts incompatible. The respective contexts are **inherited** by the following steps.

More exactly:

Search a conditional step the precondition of which makes the **contexts incompatible** and thereby **resolves threats**.

function CPOP(*initial*, *goals*, *operators*) **returns** *plan*

plan ← MAKE-PLAN(*initial*, *goals*)
loop do

Termination:
if there are no unsatisfied preconditions
and the contexts of the finish steps are exhaustive
then return *plan*

Alternative context generation:
if the plans for existing finish steps are complete and have contexts $C_1 \ldots C_n$ **then**
add a new finish step with a context $\neg (C_1 \vee \ldots \vee C_n)$
this becomes the *current context*

Subgoal selection and addition:
find a plan step S_{need} with an open precondition c

Action selection:
choose a step S_{add} from *operators* or STEPS(*plan*) that adds c or
knowledge of c and has a context compatible with the current context
if there is no such step
then fail
add $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
add $S_{add} < S_{need}$ to ORDERINGS(*plan*)
if S_{add} is a newly added step **then**
add S_{add} to STEPS(*plan*)
add *Start* $< S_{add} <$ *Finish* to ORDERINGS(*plan*)

Threat resolution:
for each step S_{threat} that potentially threatens any causal link $S_i \xrightarrow{c} S_j$
with a compatible context **do**
choose one of
Promotion: Add $S_{threat} < S_i$ to ORDERINGS(*plan*)
Demotion: Add $S_j < S_{threat}$ to ORDERINGS(*plan*)
Conditioning:
find a conditional step S_{cond} possibly before both S_{threat} and S_j, where
1. the context of S_{cond} is compatible with the contexts of S_{threat} and S_j;
2. the step has outcomes consistent with S_{threat} and S_j, respectively
add conditioning links for the outcomes from S_{cond} to S_{threat} and S_j
augment and propagate the contexts of S_{threat} and S_j
if no choice is consistent
then fail
end
end

9.5 SHOP

Up to now: Action-based planning (POP, CPOP)

- Each state of the world is represented by a set of atoms, and each action corresponds to a deterministic state transition.
- Search space is still huge.
- HTN planning has been proved to be more expressive than action-based planning.
- Moreover, HTN planning algorithms have been experimentally proved to be more efficient than their action-based counterparts.

HTN-Planning

- Each state of the world is represented by a set of atoms, and each action corresponds to a deterministic state transition.
- Hierarchical Task Networks: HTN planners differ from classical planners in what they plan for, and how they plan for it.

HTN-Planning (2)

- Classical HTN planning (dating back to mid-70ies) focused on particular application domains: production-line scheduling, crisis management and logistics, planning and scheduling for spacecraft, equipment configuration, manufacturability analysis, evacuation planning, and the game of bridge.
- There are also domain-independent HTN planners: *Nonlin*, *Sipe-2*, *O-Plan*, *UMCP*, *SHOP*, *ASHOP*, and *SHOP2*.
- We focus on SHOP.

Features of HTN planning

- Why is HTN planning superior to classical action-based planning?
- The domain knowledge and the notion of decomposing a task network while satisfying the given constraints enable the planner to focus on a much smaller portion of the search space.

Features of *SHOP*

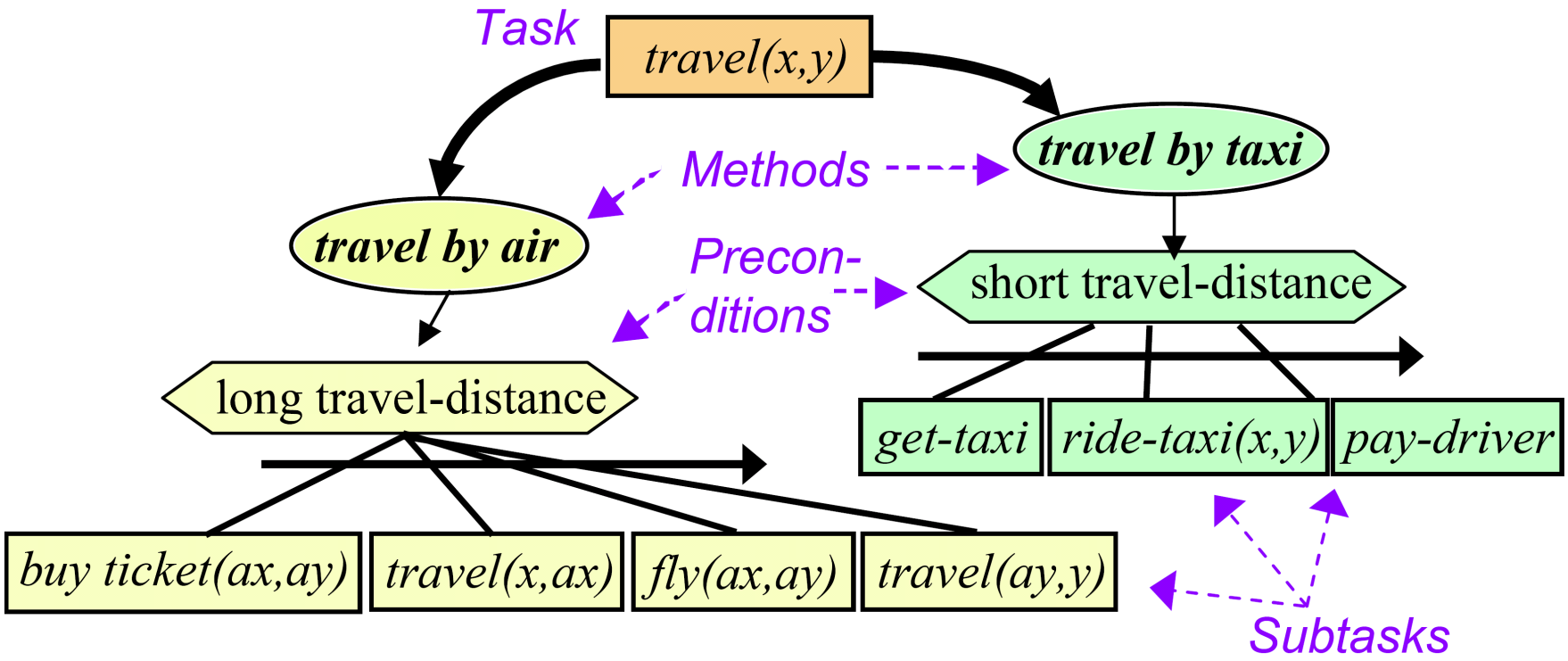
- *SHOP* is based on ordered task decomposition.
- *SHOP* plans for tasks in the same order that they will later be executed.
- Therefore we know the current state of the world at each step in the planning process.
- This eliminates a great deal of uncertainty about the world.
- It helps to incorporate inferencing and reasoning power into the planning system.
- It also enables to call external programs (e.g. to perform numeric computations).

SHOP needs the following

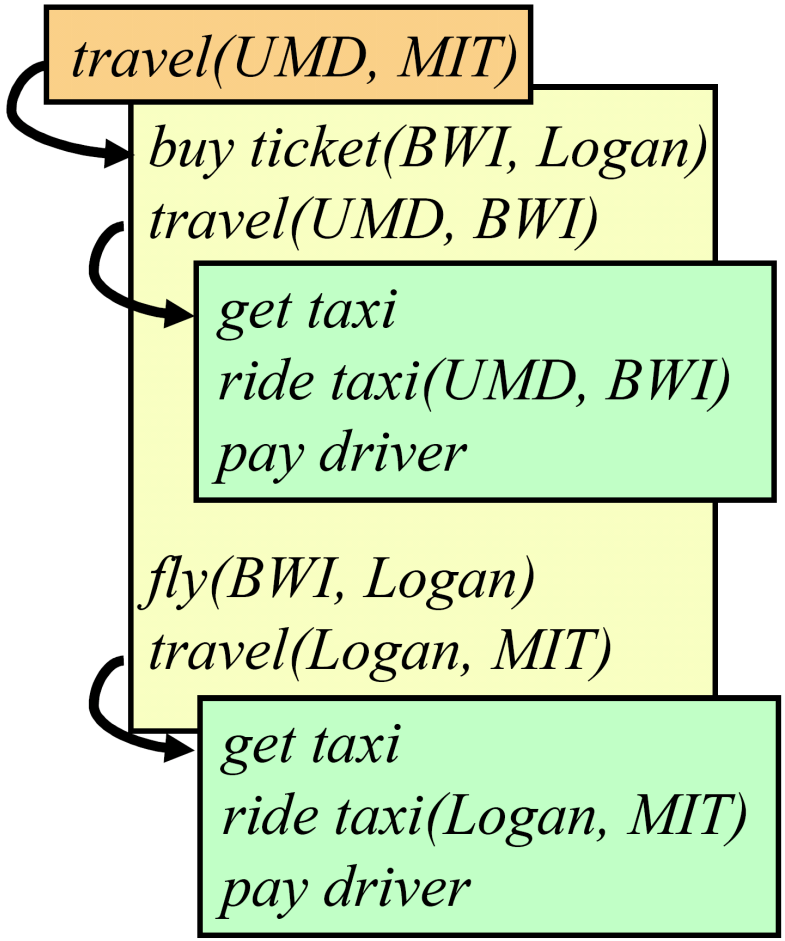
Knowledge: about the domain. Can be given as a set of axioms from which other facts can be deduced.

Operators: they describe what needs to be done to fulfill a primitive task.

Methods: often a task can not be fulfilled in one single step. In that case, the task needs to be reduced to other (new) tasks. Methods are prescriptions for how to decompose some compound (abstract) task into a **totally ordered sequence of subtasks**, along with various restrictions that must be satisfied in order for the method to be applicable.


Task
travel(x,y)
travel by taxi
Methods
travel by air
Precon-
ditions
short travel-distance
long travel-distance
get-taxi
ride-taxi(x,y)
pay-driver
buy ticket(ax,ay)
travel(x,ax)
fly(ax,ay)
travel(ay,y)
Subtasks

- More than one method may be applicable to the same task. Thus several methods can be used.
- The *SHOP* algorithm nondeterministically chooses an applicable method.
- This method is instantiated to decompose the task into (several) subtasks.
- This goes on recursively.
- The deterministic implementation of the *SHOP* algorithm uses **depth-first backtracking**: If the constraints on the subtasks prevent the plan from being feasible, then the implementation will backtrack and try other methods.


travel(UMD, MIT)
buy ticket(BWI, Logan)
travel(UMD, BWI)
get taxi
ride taxi(UMD, BWI)
pay driver
fly(BWI, Logan)
travel(Logan, MIT)
get taxi
ride taxi(Logan, MIT)
pay driver

Beware!

- The planner may need to recognise and resolve interactions among the subtasks. (travel to airport: arrive in time)
- It is not always obvious which method to use.
- If it is not possible to solve the subtasks produced by one method, *SHOP* will backtrack and try another method instead.

SHOP (3 sorts of atoms)

- **Rigid Atoms:** Atoms whose truth values never change during planning. They appear in states, but not in the effects of planning operators nor in the heads of Horn clauses.
- **Primary Atoms:** Atoms that can appear in states and in the effects of planning operators, but cannot appear in the heads of Horn clauses.
- **Secondary Atoms:** These are the ones whose truth values are inferred rather than being stated explicitly. They can appear in the heads of Horn clauses, but cannot appear in states nor in the effects of planning operators.

SHOP (States, axioms)

Definition 9.11 (States ($\mathcal{S}$), Axioms ($\mathcal{AX}$))

A state $\mathcal{S}$ is a set of ground primary atoms. An axiom is an expression of the form

$$a \leftarrow l_1, \ldots, l_n,$$

where a is a **secondary atom** and the $l_1, \ldots, l_n$ are literals that constitute either **primary or secondary** atoms.
Axioms need not be ground.

SHOP (3)

- *SHOP* starts with a state and modifies this state using add/delete lists.
- **Axioms** are used only to check whether the **preconditions** of methods are satisfied.
- A precondition might not be explicitly satisfied (an atom is not contained in $\mathcal{S}$), but might be caused by $\mathcal{S}$ and the axioms.
- The precise definition of this relation "caused by" is given as follows.

SHOP (Caused by)

Definition 9.12 (Literal caused by $(\mathcal{S},\mathcal{AX})$)

A literal l is caused by $(\mathcal{S},\mathcal{AX})$ if l is true in the unique model of $\mathcal{S} \cup \mathcal{AX}$.

SHOP (Task list)

- A *task list* is a list of tasks, like the following:

((!get-taxi ?x) (!ride-taxi ?x ?y) (!pay-driver ?x ?y)))

- A ground task list is a task list that consists of only ground tasks, like the following:

((!get-taxi umd) (!ride-taxi umd mit) (!pay-driver umd mit)))

SHOP (Operator)

Definition 9.13 (Operator: (Op h ϵ_{del} ϵ_{add}))

An operator is an expression of the form **(Op** h ϵ_{del} ϵ_{add}**)**, where h (the *head*) is a primitive task and ϵ_{add} and ϵ_{del} are lists of primary atoms (called the add- and delete-lists, respectively). The set of variables in the atoms in ϵ_{add} and ϵ_{del} must be a subset of the set of variables in h.

- Unlike the operators used in action-based planning, ours have no preconditions.
- Preconditions are not needed for operators in our formulation, because they occur in the methods that invoke the operators.

SHOP

As an example, here is a possible implementation of the `get-taxi` operator:

```
(:Op (!get-taxi ?x)
  ((taxi-called-to ?x))
  ((taxi-standing-at ?x)))
```

SHOP (Decomposing primitive tasks)

Operators are used in decomposition of primitive tasks during planning:

Definition 9.14 (Decomposition of Primitive Tasks)

Let t be a primitive task, and let $\mathrm{Op} = (\mathrm{Op}\ h\ \epsilon_{del}\ \epsilon_{add})$ be an operator. Suppose that θ is a unifier for h and t. Then the ground operator instance $(\mathrm{Op})\theta$ is applicable to t, in which case we define the decomposition of t by Op to be $(\mathrm{Op})\theta$.

SHOP

The decomposition of a primitive task by an operator results in a ground instance of that operator – i.e., it results in an action that can be applied in a state of the world. We now define the result of such an application.

SHOP (Plan)

Definition 9.15 (Plans, result($\mathcal{S}$,π))

A plan is a list of heads of ground operator instances. A plan π is called a simple plan if it consists of the head of just one ground operator instance.
Given a simple plan $\pi = (h)$, we define result$(\mathcal{S}, \pi)$ to be the set $\mathcal{S} \setminus \epsilon_{del} \cup \epsilon_{add}$, obtained by deleting from $\mathcal{S}$ all atoms in ϵ_{del} and by adding all ground instances of atoms in ϵ_{add}.
If $\pi = (h_1, h_2, \ldots, h_n)$ is a plan and $\mathcal{S}$ is a state, then the result of applying π to $\mathcal{S}$ is the state

$$\text{result}(\mathcal{S}, \pi) = \text{result}(\text{result}(\ldots(\text{result}(\mathcal{S}, h_1), h_2), \ldots), h_n).$$

SHOP

- In *SHOP*, a method specifies a possible way to accomplish a compound task.
- The set of methods relevant for a particular compound task can be seen as a recursive definition of that task.

SHOP (Methods)

Definition 9.16 (Method: (Meth h ρ t))

A **method** is an expression of the form **(Meth h ρ t)** where h (the method's **head**) is a compound task, ρ (the method's **preconditions**) is a conjunction of literals and t is a totally-ordered list of subtasks, called the **decomposition list** of the method.

The set of variables that appear in the decomposition list of a method must be a subset of the variables in h (the head of the method) and ρ (the preconditions of the method).

SHOP

Here is a possible implementation of the `travel-by-taxi` method:

```
(:Meth (travel ?x ?y)
   ((smaller-distance ?x ?y))
   ((!get-taxi ?x) (!ride-taxi ?x ?y) (!pay-driver ?x ?y)))
```

SHOP (Decomposing compound tasks)

Definition 9.17 (Decomposition of Compound Tasks)

Let t be a compound task, $\mathcal{S}$ be the current state, $Meth = (\mathbf{Meth}\ h\ \rho\ \mathbf{t})$ be a method, and $\mathcal{AX}$ be an axiom set. Suppose that θ is a unifier for h and t, and that θ' is a unifier such that all literals in $(\rho)\theta\theta'$ are caused wrt. $\mathcal{S}$ and $\mathcal{AX}$ (see Definition 9.12).
Then, the ground method instance $(Meth)\theta\theta'$ is applicable to t in $\mathcal{S}$, and the result of applying it to t is the ground task list $\mathbf{r} = (\mathbf{t})\theta\theta'$. The task list $\mathbf{r}$ is the decomposition of t by *Meth* in $\mathcal{S}$.

SHOP (Planning Problem)

Definition 9.18 (Planning Domain Description)

A planning domain description $\mathcal{D}$ is a triple consisting of (1) a set of axioms, (2) a set of operators such that no two operators have the same head, and (3) a set of methods.

A planning problem is a triple $(\mathcal{S}, \mathrm{t}, \mathcal{D})$, where $\mathcal{S}$ is a state, t= $(t_1, t_2, \ldots, t_k)$ is a ground task list, and $\mathcal{D}$ is a planning domain description.

SHOP (Solutions)

Definition 9.19 (Solutions)

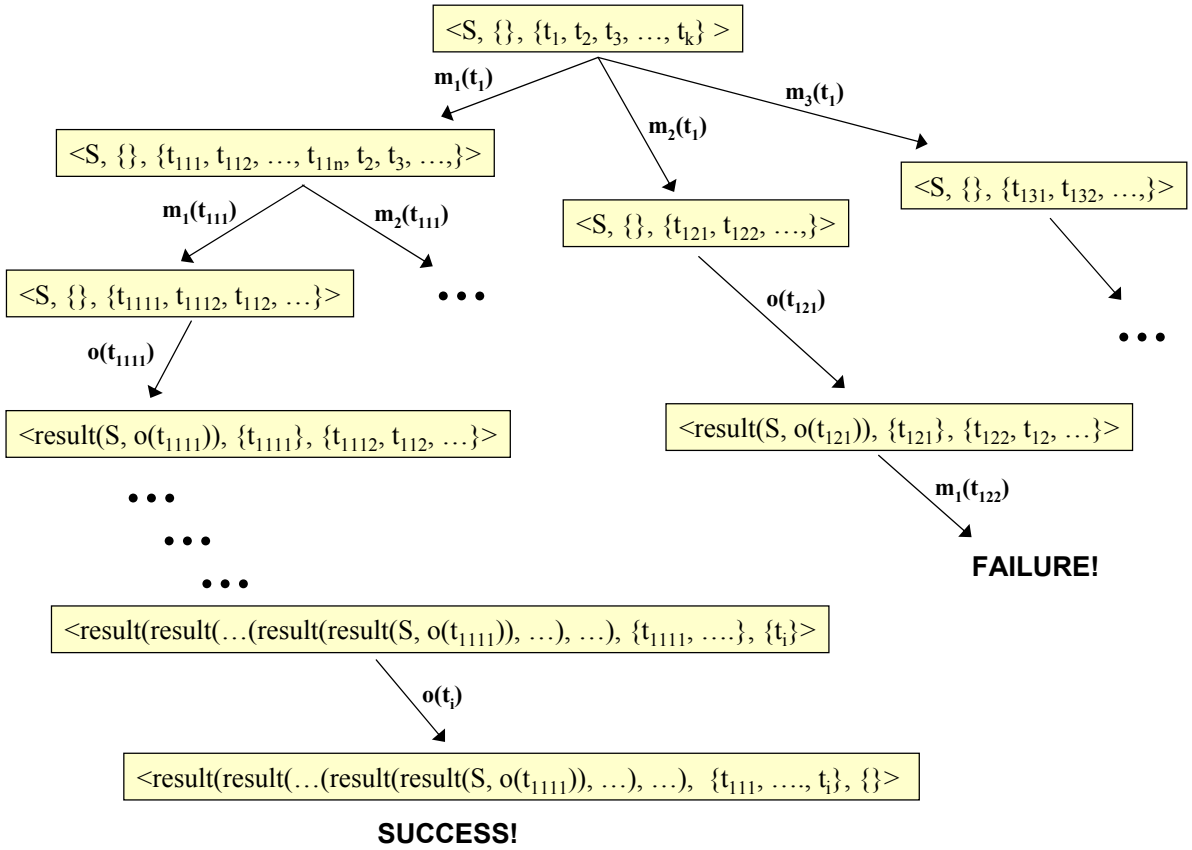
Let $P = (\mathcal{S}, \mathbf{t}, \mathcal{D})$ be a planning problem and $\pi = (h_1, h_2, \ldots, h_n)$ be a plan. Then, π is a **solution** for P, if any of the following is true:

- Case 1: $\mathbf{t}$ and π are both empty, (i.e., $k = 0$ and $n = 0$);
- Case 2: $\mathbf{t} = (t_1, t_2, \ldots, t_k)$, t_1 is a ground primitive task, (h_1) is the decomposition of t_1, and $(h_2 \ldots h_n)$ solves $(\text{result}(\mathcal{S}, (h_1)), (t_2, \ldots, t_k), \mathcal{D})$;
- Case 3: $\mathbf{t} = (t_1, t_2, \ldots, t_k)$, t_1 is a ground compound task, and there is a decomposition $(r_1 \ldots r_j)$ of t_1 in $\mathcal{S}$ such that π solves $(\mathcal{S}, (r_1, \ldots, r_j, t_2, \ldots, t_k), \mathcal{D})$.

The planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is **solvable** if there is a plan that solves it.

SHOP (Search Tree)

Edge labellings $m_i(t)$ (resp. $o(t)$) represent a method (resp. an operator) application to a task t, which is compound (resp. primitive).



9.6 Extensions

1. A plan can fail because of the following reasons:
   - Actions may have unexpected effects, but these can be enumerated (as a disjunction).
   - The **unexpected effects** are known. Then we have to replan.

function REPLANNING-AGENT(*percept*) **returns** an *action*
  static: *KB*, a knowledge base (includes action descriptions)
    p, an annotated plan, initially *NoPlan*
    q, an annotated plan, initially *NoPlan*
    G, a goal

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  current ← STATE-DESCRIPTION(*KB*, *t*)
  if *p* = *NoPlan* **then**
    p ← PLANNER(*current*, *G*, *KB*)
    q ← *p*
    if *p* = *NoPlan* **or** *p* is empty **then return** *NoOp*
  if PRECONDITIONS(*p*) not currently true in *KB* **then**
    p' ← CHOOSE-BEST-CONTINUATION(*current*, *q*)
    p ← APPEND(PLANNER(*current*, PRECONDITIONS(p'), *KB*), p')
    q ← *p*
  action ← FIRST(*p*)
  p ← REST(*p*)
  return *action*

I.e. we perceive and then **plan only if something has changed**.

2. Combine **replanning** and **conditional planning**. Planning and execution are **integrated**.