

Foundations of Artificial Intelligence

Riccardo Salvalaggio

August 30, 2021

Contents

1	Introduction	5
2	Rational Agents	6
2.1	Structure of Rational Agents	6
2.2	Classes of agents	6
2.2.1	Table-Driven (the simplest)	6
2.2.2	Interpretative Reflex	6
2.2.3	Model-Based Reflex	6
2.2.4	Learning agents	7
2.3	Types of environments	7
2.4	Summary	7
3	Solving Problems by Searching	8
3.1	Problem-solving agents	8
3.2	Problem Formulation	8
3.3	Problem Types	8
3.4	Search strategies	9
3.4.1	Uninformed or blind searches	10
3.5	Summary	11
4	Informed Search Methods	12
4.1	Greedy Search	12
4.2	Heuristics	12
4.3	A* and IDA*	12
4.3.1	Optimality of A*	13
4.3.2	Graph- vs. Tree-search	13
4.3.3	Variants of A*	13
4.4	Local Search Methods	13
4.5	Genetic Algorithms	14
4.6	Summary	14
5	Board games	15
5.1	Minimax search	15
5.2	Alpha-Beta Search	15
5.3	Games with an Element of Chance	16
5.4	Summary	17
6	Constraint satisfaction problems	18
6.1	Backtracking search for CSPs	18
6.2	CSP Heuristics	18
6.3	Constraint Propagation	18
6.3.1	Arc Consistency	18
6.4	Problem Structure	19
6.5	Summary	19
7	Propositional logic	20
7.1	Agents that Think Rationally	20
7.1.1	Syntax	21
7.1.2	Normal forms	21
7.1.3	Resolution	22
7.2	Summary	24

8 Satisfiability and Model Construction	25
8.1 Davis-Putnam-Logemann-Loveland (DPLL) Procedure	25
8.1.1 DPLL on Horn Clauses	26
8.2 Average Complexity of the Satisfiability Problem	26
8.3 Local Search	27
8.4 State of the Art	27
8.4.1 Improvements of DPLL Algorithms	27
8.4.2 Algorithm configuration	28
8.4.3 Algorithm selection	28
8.4.4 Automated construction of portfolios from a single algorithm	28
8.5 Summary	29
9 Predicate Logic (First-Order Predicate Logic (PL-1))	30
9.1 Semantics of PL1-Logic	30
9.1.1 Variable assignment	31
9.1.2 Satisfiability	31
9.2 Free and Bound Variables	31
9.3 Reduction to Propositional Theories	31
9.3.1 Derivation in PL1	31
9.3.2 Finite universes	32
9.3.3 Instantiation	32
9.4 Summary	32
10 Action Planning	33
10.1 Planning Formalisms	33
10.1.1 Domain-Independent Action Planning	33
10.1.2 Planning as Logical Inference	33
10.1.3 Basic STRIPS Formalism	33
10.1.4 Plans & Successful Executions	34
10.1.5 PDDL: The Planning Domain Description Language	35
10.2 Basic Planning Algorithms	35
10.3 Computational Complexity	36
10.3.1 The planning problem	36
10.3.2 Restrictions on Plans	37
10.4 Current Algorithmic Approaches	37
10.5 Summary	37
11 Making simple decisions under uncertainty (Probability)	38
11.1 Foundations of Probability Theory	38
11.2 Probabilistic Inference	39
11.2.1 Joint Probability	39
11.2.2 Bayes' Rule	39
11.2.3 Relative Probability	40
11.2.4 Normalization	40
11.2.5 Multiple Evidence	41
11.2.6 Recursive Bayesian Updating	42
11.3 Bayesian Network	42
11.4 Summary	45
12 Acting under Uncertainty Maximizing Expected Utility	46
12.1 Introduction to Utility theory	46
12.2 Choosing Individual Actions	46
12.3 Sequential Decision Problems	47
12.4 Markov Decision Processes	47
12.5 Value Iteration	48
12.5.1 Policy Iteration	49
12.6 Summary	50

13 Machine Learning	51
13.1 The learning agent	51
13.2 Types of learning	51
13.3 Decision trees	51
13.3.1 Boolean Decision Trees	51
13.4 Summary	54
14 Deep Learning	55
14.1 Why?	55
14.2 Representation Learning and Deep Learning	55
14.3 Multilayer Perceptrons	55
14.4 Convolutions Neural Networks	56
14.5 Recurrent Neural Networks	56
14.6 Reinforcement Learning	57
14.7 Limitations	57
14.8 Summary - Why Deep Learning?	58
15 Natural Language Processing	59
15.1 Motivations and Tasks	59
15.2 Learning Representations	59
15.3 Sequence-to-Sequence Deep Learning	59
15.3.1 Association Modules	59
15.3.2 From Representations to the Translation of Texts	60
15.4 Sequence-to-Sequence Neural Machine Translation	60
15.5 Summary	60
16 Ethics	61
16.1 Ethical principles	61
16.2 Algorithmic Fairness	61
16.2.1 Possible reasons for unfairness	61
16.3 Machine Ethics	61
16.4 Self-Driving Cars	62
16.5 Morally Competent Planning Systems	62
16.6 Summary	63

1 Introduction

Artificial Intelligence is the attempt to make computers more "intelligent" to better understand human intelligence.

Thinking Humanly "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)	Thinking Rationally "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)
Acting Humanly "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)	Acting Rationally "Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998) "AI ... is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

2 Rational Agents

An agent is a model that perceive the environment through sensors and act through actuators. In order to evaluate their performance use performance measure (e.g. vacuum → level of cleanliness etc.) even if optimal behaviour is often unattainable because it is quite impossible to reach the goal in every aspect.

Omniscient if it knows the effects of its actions.

Rational agent behaves according to its percepts and knowledge and attempts to maximize the expected performance.

Ideal: for each possible percept sequence, selects an action that is expected to maximize its performance measure.

2.1 Structure of Rational Agents

The mapping is realised through an agent program executed on an Architecture which also provides an interface to the environment(percepts, actions) **Agent = Architecture + Program**

2.2 Classes of agents

2.2.1 Table-Driven (the simplest)

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
            table, a table of actions, indexed by percept sequences, initially fully specified

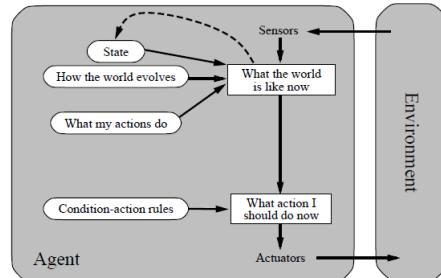
  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Problem: need a huge table to fulfill all the possible perceptions.

2.2.2 Interpretative Reflex

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition-action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

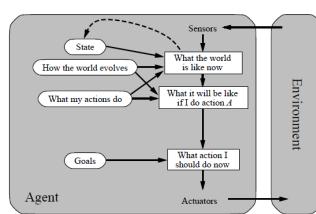


Interpretation of the input, matching to a rule to extract an action.

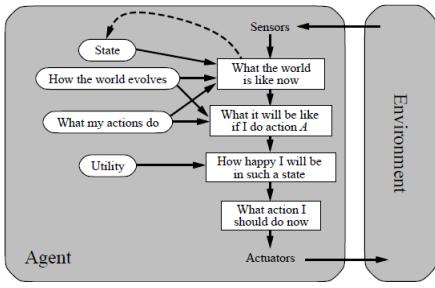
2.2.3 Model-Based Reflex

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
            model, a description of how the next state depends on current state and action
            rules, a set of condition-action rules
            action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```



Introduction of a utility function that maps a state onto a real number in order to compute the best action to do and to weigh the importance of competing goals.



2.2.4 Learning agents

Agents that improve over time starting from an empty knowledge and unknown environments.

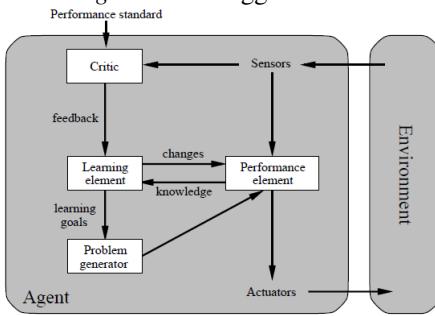
Components:

Learning element: responsible for making improvements.

Performance element: select external actions.

Critic: determines performance of the agent.

Problem generator: suggests actions that lead to informative experiences.



2.3 Types of environments

Accessible vs. inaccessible, Deterministic vs. stochastic, Episodic vs. sequential, static vs. dynamic, discrete vs. continuous, single vs. multi agent.

2.4 Summary

An **agent** is something that perceives and acts. It consists of an architecture and an agent program.

An **ideal rational agent** always takes the action that maximizes its performance.

An **agent program** maps from a percept to an action.

There are a variety of designs:

Reflex agents respond immediately to percepts.

Goal-based agents work towards goals.

Utility-based agents try to maximize their reward.

Learning agents improve their behavior over time.

Some **environments** are more demanding than others. Environments that are partially observable, nondeterministic, strategic, dynamic, and continuous and multi-agent are the most challenging.

3 Solving Problems by Searching

3.1 Problem-solving agents

Formulation: *problem* as a *state-space* and *goal* as a *particular condition on states*

Given: *initial state*

Goal: To reach the specified goal (a state) through the *execution of appropriate actions*

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action

persistent: *seq*, an action sequence, initially empty
state, some description of the current world state
goal, a goal, initially null
problem, a problem formulation

state \leftarrow UPDATE-STATE(*state*, *percept*)
if *seq* is empty **then**
 goal \leftarrow FORMULATE-GOAL(*state*)
 problem \leftarrow FORMULATE-PROBLEM(*state*, *goal*)
 seq \leftarrow SEARCH(*problem*)
 if *seq* = failure **then return** a null action
 action \leftarrow FIRST(*seq*)
 seq \leftarrow REST(*seq*)
return *action*

- Properties: Fully-observable, Deterministic/static env., discrete states, single-agent.

3.2 Problem Formulation

Goal formulation, definition of: State space, actions, problem type, search and execution costs.

3.3 Problem Types

Based on knowledge of States and Actions: Observability, completeness of knowledge about world state and actions.
(e.g. If the environment is completely observable, the vacuum cleaner always knows where it is and where the dirt is.)

Transition Model: Description of the outcome of an action.

Solution: Path from the initial to a goal state.

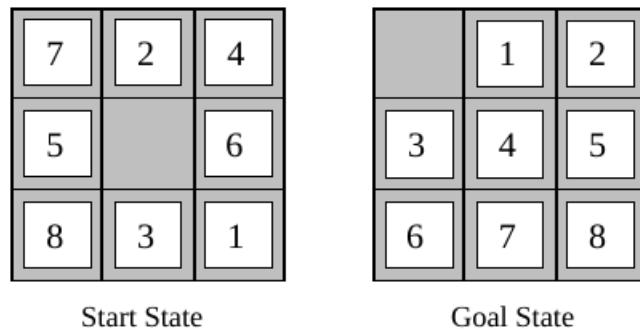
Search Costs: Time and storage requirements to find a solution.

Total Costs: Search costs + path costs.

Alternative formulations can influence a lot number of states, e.g. 8 queens problem: Naive - billions of states, Better - 2057 states.

Examples of Real-World Problems: Route planning, shortest path problem, TSP, VLSI Layout, Robot nav., Assembly sequencing.

Example: The 8-Puzzle



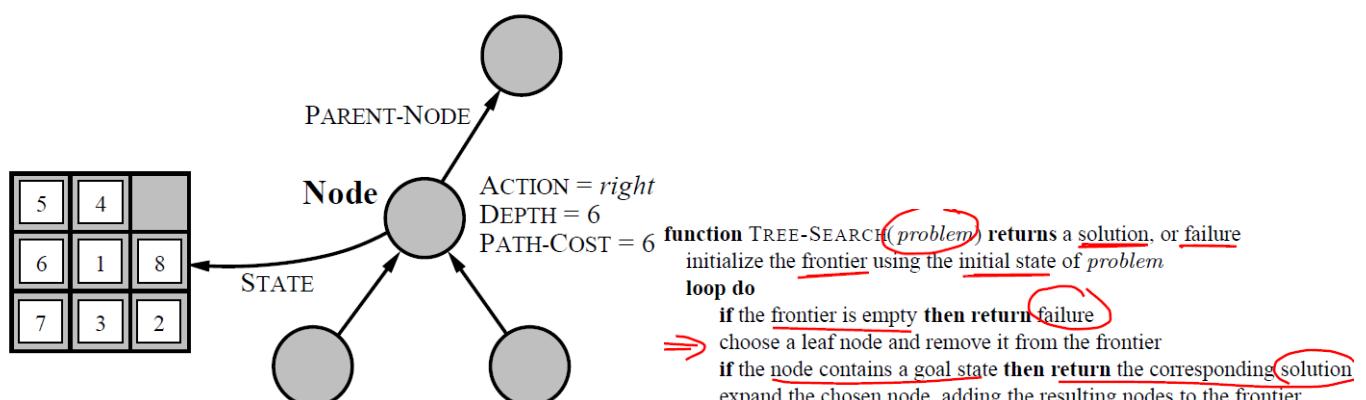
- **States:** Description of the location of each of the eight tiles and (for efficiency) the blank square.
- **Initial State:** Initial configuration of the puzzle.
- **Actions (transition model defined accordingly):** Moving the blank left, right, up, or down.
- **Goal Test:** Does the state match the configuration on the right (or any other configuration)?
- **Path Costs:** Each step costs 1 unit (path costs corresponds to its length).

3.4 Search strategies

E.g.: node expansion, frontier, search strategy, tree-based search, graph-based search.

- Search Tree Data structure:

state, parent, action (that was applied to the parent), path-cost (the cost, traditionally denoted by $g(n)$, of the path from the initial state to the node).



```

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

```

- Criteria for Search Strategies: **Completeness, Time complexity, Space Complexity, Optimality.**

3.4.1 Uninformed or blind searches

- Breadth-First Search:

Nodes are expanded in the order they were produced (first siblings, then children) (frontier = FIFO queue). Completeness is obvious, the solution is optimal. **Time complexity:** Let b be the maximal branching factor and d the depth of a solution path. Then the maximal number of nodes expanded is $= O(b^d)$. **Space Complexity:** $O(b^d)$

```

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier  $\leftarrow$  a FIFO queue with node as the only element
    explored  $\leftarrow$  an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child  $\leftarrow$  CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier  $\leftarrow$  INSERT(child, frontier)

```

- Uniform-Cost Search:

If step costs are different, **uniform cost** is better. It expands node with lowest path costs $g(n)$. It uses a Priority queue. Always finds the cheapest solution, given that $g(successor(n)) \geq g(n)$ for all n .

- Depth-First Search:

Always expands an unexpanded node at the greatest depth (frontier < - a LIFO queue, first children, then siblings). Usually implemented recursively.

Generally, optimal is not guaranteed. Completeness only for graph-based search. **Time complexity:** in graph-based is bounded by the space, so it can be infinite, in tree-based: $O(b^m)$ (m max length of a path). **Space Complexity:** tree-based: $O(b^*m)$, graph-based: worst-case, all states need to be stored. (no better than breadth-first).

Depth-Limited-first search is like depth with an imposed cutoff on the maximum depth of a path. e.g., route planning.

```

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if limit = 0 then return cutoff
    else
        cutoff_occurred?  $\leftarrow$  false
        for each action in problem.ACTIONS(node.STATE) do
            child  $\leftarrow$  CHILD-NODE(problem, node, action)
            result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
            if result = cutoff then cutoff_occurred?  $\leftarrow$  true
            else if result  $\neq$  failure then return result
        if cutoff_occurred? then return cutoff else return failure

```

- Iterative Deepening Search:

Like depth-limited search and in every iteration increase search depth by one. Combines depth and breadth-first. Optimal

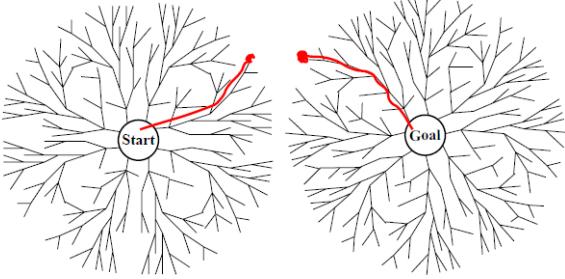
and complete like breadth-first, but requires much less memory: $O(b^d)$.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result
```

Iterative deepening in general is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

For small space it is worse than breadth-first.

- Bidirectional searches:



As long as forward and backward searches are symmetric, search times of $O(2^d/2) = O(d^d)$ can be obtained. The operators are not always reversible, there must be an efficient way to check if a new node already appears in the search tree of the other half of the search.

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

3.5 Summary

Before an agent can start searching for **solutions**, it must formulate a **goal** and then use that goal to **formulate a problem**. A problem consists of five parts: **The state space, initial situation, actions, goal test and path costs**.

A path from an initial state to a goal state is a **solution**.

A **general search** algorithm can be used to solve any problem. Specific **variants** of the algorithm can use different search strategies.

Search algorithms are judged on the basis of completeness, optimality, time complexity and space complexity.

4 Informed Search Methods

They are categorized based on how they determine how to expand to the next node.

- **Uninformed:** rigid procedure with **no knowledge** of the cost of a given node to the goal.
- **Informed:** knowledge of the worth of expanding a node n is given in the form of an **evaluation function $f(n)$** which assigns a real number to each node. Mostly, $f(n)$ includes as a component a heuristic function $h(n)$, which estimates the costs of the cheapest path from n to the goal. $f(n) = g(n) + h(n)$
- **Best-first:** informed that expands with the best f -value.

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

Instance of tree-search algorithm in which frontier is a **priority queue**. When f is always correct, we don't need to search.

4.1 Greedy Search

A possible way is to set: **$h(n) = \text{estimated path-costs from } n \text{ to the goal}$** . A best-first search using $h(n)$ (heuristic function) as evaluation function is called **greedy search** ($f(n)=h(n)$).

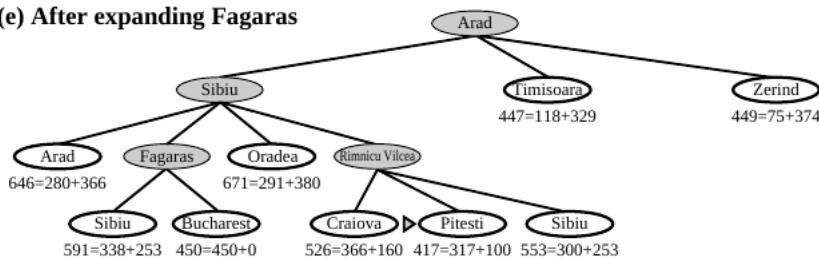
4.2 Heuristics

Heuristics are fast but in **certain situations incomplete** methods for problem-solving, they **improve the search in the average-case and the time complexity**. In general, **not optimal and incomplete**; graph-search version is complete only in finite spaces.

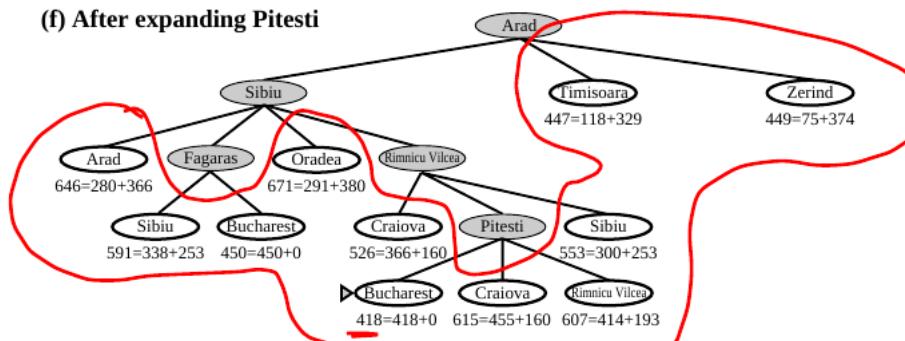
4.3 A* and IDA*

Improvement of greedy, A* **combines greedy search with the uniform-cost search**: always expand node with lowest $f(n) = g(n)$ (**actual cost from start to n**) + $h(n)$ (**estimated cost to goal/optimistic estimate of the costs**). A new h is **admissible** iff: $h(n) \leq h^*(n)$. We require that for A*, h is admissible (example: straight-line distance is admissible).

(e) After expanding Fagaras



(f) After expanding Pitesti



4.3.1 Optimality of A*

- **Claim:** The first solution found has the minimum path cost.
- **Proof:** Suppose there exists a goal node G with optimal path cost f^* , but A* has first found another node G_2 with $g(G_2) > f^*$. Let n be a node on the path from the start to G that has not yet been expanded. Since h is admissible, we have: $f(n) \leftarrow f^* \rightarrow f(G_2) \leftarrow f(n) \rightarrow f(G_2) \leftarrow f^* \rightarrow g(G_2) \leftarrow f^*$ **Contradiction**.
- **Completeness:** If a solution exists, A* will find it provided that (1) every node has a finite number of successor nodes, and (2) there exists a positive constant $\delta > 0$ such that every step has at least cost δ .
- **Complexity:** In general, still exponential in the path length of the solution (space, time), it depends on the choice of Heuristic used.

4.3.2 Graph- vs. Tree-search

For the **graph-based variant**, either needs to consider re-opening nodes from the explored set, when a better estimate becomes known, or needs to require stronger restrictions on the heuristic estimate: it needs to be consistent (**consistent iff for all actions a leading from s to s': $h(s) - h(s') \leftarrow c(a)$** , where **c(a)** denotes the cost of action a). Consistency implies admissibility, A* can still be applied if heuristic is not consistent but optimality is lost.

4.3.3 Variants of A*

A* in general suffers from exponential memory growth.

- **Iterative-deepening A*:** f-costs are used to define the cut-off (IDA*).
- **Recursive Best First Search (RBFS):** introduces a variable *f-limit* to keep track of the best alternative path, if the limit is exceeded opt for the alternative path.
- **Other alternatives:** MA* and SMA*.

4.4 Local Search Methods

In many problems we are interested only on solving it, not how. If there is also a quality measure, local search can be used.

It uses a **Hill Climbing/ Gradient descent mechanisms** (improvements step by step), requires few memory because it operates using just the current node.(no memory)

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor  $\leftarrow$  a highest-valued successor of current
    if neighbor.VALUE  $\leq$  current.VALUE then return current.STATE
    current  $\leftarrow$  neighbor
```

- **Problem:** the algorithm can stop in a sub-optimal solution (local maxima), the algorithm explore at random (plateaus), requires also suboptimal moves (ridges).
- **Solution:** restart if no improvements, inject noise.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
           schedule, a mapping from time to "temperature"
  current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to  $\infty$  do
    T  $\leftarrow$  schedule(t)
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  next.VALUE  $-$  current.VALUE
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E / T}$ 
```



4.5 Genetic Algorithms

Similar to evolution, we search for solutions by three operators: **"mutation"**, **"crossover"**, and **"selection"**. Need of **coding a solution** into a string of symbols or bit-string, fitness function to judge the worth of configurations.

Example: Represent an individual (one 8-queens configuration) as a concatenation of eight x-y coordinates. Its fitness is judged by the number of non-attacks. The population consists of a set of configurations.

4.6 Summary

Heuristics focus the search.

Best-first search expands the node with the highest worth (defined by any measure) first.

With the minimization of the evaluated costs to the goal h we obtain a **greedy search**.

The minimization of $f(n) = g(n) + h(n)$ combines **uniform and greedy searches**. When $h(n)$ is **admissible**, i.e., h^* is never overestimated, we obtain the A^* search, which is complete and optimal.

IDA* is a combination of the iterative-deepening and A^* searches.

Local search methods work on a single state only, attempting to improve it step-wise.

Genetic algorithms imitate evolution by combining good solutions.

5 Board games

They represent an abstract and pure form of competition, perfect for testing "intelligence".

The states of a game are easy to represent. The possible actions of the players are well-defined.

The game can be implemented as a kind of search problem, the individual **states are fully accessible** and is nonetheless a contingency problem, because the **actions of the opponent** are not under the control of the player.

Board games are not only difficult because they are contingency problems, but also because the **state space can become astronomically large**.

5.1 Minimax search

In contrast to regular searches, where a path from beginning to end is a solution, max must come up with a strategy to reach a favorable terminal state regardless of what min does → **all of min moves must be considered and reactions to them must be computed**.

When it is possible to **produce the full game tree**, the **minimax algorithm delivers an optimal strategy for max**.

- Algorithm:

1. Generate the **complete tree with depth-first search**.
2. Apply utility function.
3. Determine the **utility**: if predecessor is **MIN** assign **minimum value**, if predecessor is **MAX** assign **maximum value**.
Predecessor node is a **max-node**: Value is the **maximum of its child nodes**. From the initial state (root of the game tree), max chooses the move that leads to the highest value (minimax decision).

```
function MINIMAX-DECISION(state) returns an action
    return arg maxa ∈ ACTIONS(s) MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← -∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, a)))
    return v

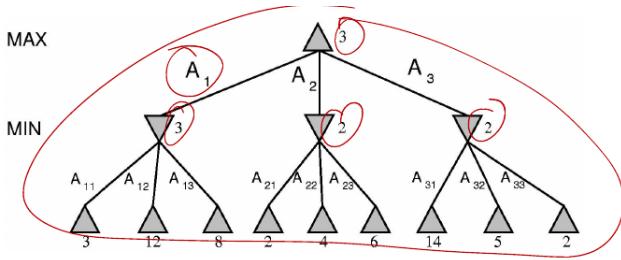
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, a)))
    return v
```

5.2 Alpha-Beta Search

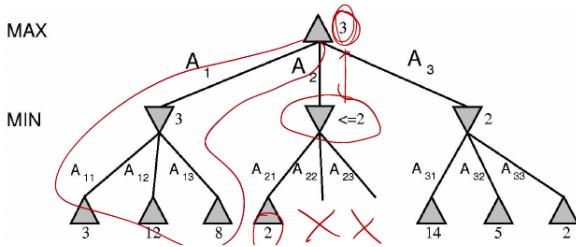
When the tree is becoming too large, we have to evaluate correctly where to expand it. The design of the evaluation function is fundamental!

Standard evaluation functions are weighted linear: $w_1f_1 + w_2f_2 + \dots + w_nf_n$. (w weights, f features).

Contributions of features are independent.



Can we improve this? We do not need to consider all nodes.



- **Alpha-Beta pruning:** α : value of the best choice for MAX, β : value of the best choice for MIN.

(1) Prune below the min node whose β

- bound is less than or equal to the α -bound of its max-predecessor node. (2) Prune below the max node whose α -bound is greater than or equal to the β
- bound of its min-predecessor node.

```

function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v



---


function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$ 
  for each  $a$  in ACTIONS(state) do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$ 
    if  $v \geq \beta$  then return  $v$ 
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
  return  $v$ 



---


function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$ 
  for each  $a$  in ACTIONS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$ 
    if  $v \leq \alpha$  then return  $v$ 
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return  $v$ 

```

The alpha-beta search cuts the largest amount off the tree when we examine the best move first. Best case: $O(b^{d/2})$. Average case: $O(b^{3d/4})$. Practical case: A simple ordering heuristic brings the performance close to the best case.

5.3 Games with an Element of Chance

In addition to min- and max nodes, we need **chance nodes** (e.g. for the dice).

- **Expected value:**

Utility function for chance nodes C over MAX:

d_i : possible dice roll

$P(d_i)$: probability of obtaining that roll

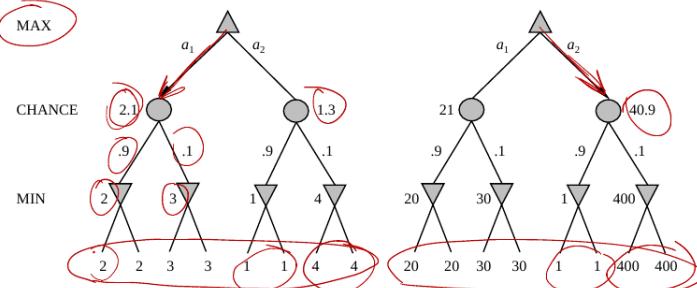
$S(C, d_i)$: attainable positions from C with roll d_i

UTILITY(s): Evaluation of s

$$\text{EXPECTIMAX}(C) = \sum_i P(d_i) \max_{s \in S(C, d_i)} (\text{UTILITY}(s))$$

: EXPECTIMIN likewise

Using expected values: An order-preserving transformation on the evaluation values may change the best move. Example:



Search costs increase: Instead of $O(b^d)$, we get $O((b \cdot n)^d)$, where n is the number of possible dice outcomes.

5.4 Summary

A **game** can be defined by the initial state, the operators (legal moves), a terminal test and a utility function (outcome of the game).

In two-player board games, the **minimax algorithm** can determine the best move by enumerating the entire game tree.

The **alpha-beta algorithm** produces the same result but is more efficient because it prunes away irrelevant branches.

Usually, it is not feasible to construct the complete game tree, so the utility of some states must be determined by an **evaluation function**.

Games of chance can be handled by an **extension of the alpha-beta algorithm**.

The success for different games is based on quite different methodologies.

6 Constraint satisfaction problems

Such problems are defined by:

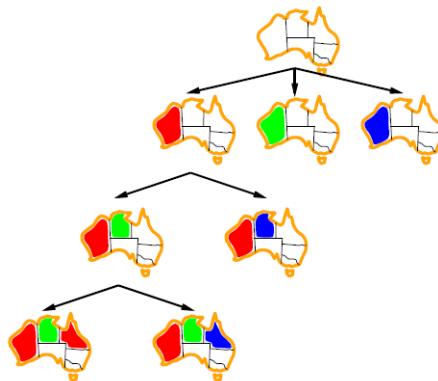
- **variables, a set of value domains, a set of constraints, an assignment.** Main idea is to **exploit the constraints to delete large portions of search space.**

A **constraint graph** can be used to visualize binary constraints, **Nodes = variables, arcs = constraints. State:** a variable assignment.

6.1 Backtracking search for CSPs

It assigns values to variables step by step using DFS single-variable (Backtracking search).

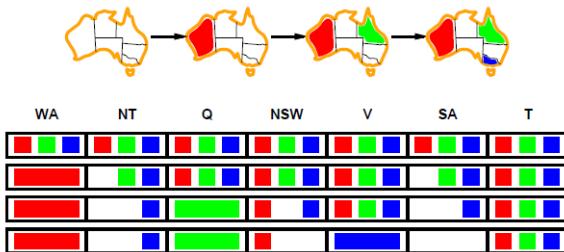
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)
function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var,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



6.2 CSP Heuristics

- **Variable ordering:** e.g.: most constrained first: choose the variable with the fewest remaining legal value.

- **Value ordering:** e.g.: Least constraining value first - **Forward checking:** e.g.: Rule out failures early



6.3 Constraint Propagation

Forward Checking has a problem

It is that constraints of unassigned variables are not propagated.

A directed arc $X \rightarrow Y$ is **consistent** iff for every choice of x there exists y that satisfies constraints, remove values from the domain of X to enforce **arc-consistency**;

6.3.1 Arc Consistency

It detects failures earlier, can be used as preprocessing.

```

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

  while queue is not empty do
    (Xi, Xj)  $\leftarrow$  REMOVE-FIRST(queue)
    if REVISE(csp, Xi, Xj) then
      if size of Di = 0 then return false
      for each Xk in Xi.NEIGHBORS - {Xj} do
        add (Xk, Xi) to queue
  return true
  Check whether I have to remove a value from this domain.

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

```

Time complexity: $O(d^3 \cdot n^2)$). Of course, AC-3 does not detect all inconsistencies (which is an NP-hard problem).

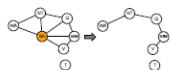
6.4 Problem Structure

If the CSP graph is a tree, then it can be solved in $O(n \cdot d^2)$ (general CSPs need in the worst case $O(d^n)$). Idea: Pick root, order nodes, apply arc consistency bottom-up, assign starting at root. This algorithm is linear in n . Another solution is to reduce the graph structure by fixing values in a reasonably chosen subset. Instantiate a variable and prune values in neighboring variables (conditioning).

- Almost Tree construction:

Algorithm Cutset Conditioning:

- ① Choose a subset *S* of the CSP's variables such that the constraint graph becomes a tree after removal of *S*. The set *S* is called a cycle cutset.
- ② For each possible assignment of variables in *S* that satisfies all constraints on *S*
 - ① remove from the domains of the remaining variables any values that are inconsistent with the assignments for *S*, and
 - ② if the remaining CSP has a solution, return it together with the assignment for *S*



- Tree Decomposition:

Decompose the problem into a set of connected (share a constraint) sub-problems and solve them independently. If a variable appears in two sub-problems, it must appear in all sub-problems on the path between the two sub-problems.

Consider sub-problems as new mega-variables, use tree-structured CSP to find solution.

The tree width *w* of a tree decomposition is the size of largest sub-problem minus 1, If a graph has tree width *w* and we know a tree decomposition with that width, we can solve the problem in $O(n \cdot d^{w+1})$.

6.5 Summary

CSPs are a special kind of search problem: states are value assignments, goal test is defined by constraints.

Backtracking = DFS with one variable assigned per node. Other intelligent backtracking techniques possible.

Variable/value ordering heuristics can help dramatically.

Constraint propagation prunes the search space.

Tree structure of CSP graph simplifies problem significantly.

Cutset conditioning and tree decomposition are two ways to transform part of the problem into a tree.

CSPs can also be solved using **local search**.

7 Propositional logic

Logic is a universal tool with many powerful applications.

7.1 Agents that Think Rationally

Rational action requires rational (logical) thought on the agent's part so they must **know a portion of the world in its near (Knowledge base, KB)**. KB is composed of **sentences in a logic language**.

- **Levels:**

1. **Knowledge level:** most abstract level, concerns the total knowledge.
2. **Logical level:** encoding of knowledge in a formal language.
3. **Implementation level:** the internal representation of the sentences. (as a string or as a value).

function KB-AGENT(*percept*) **returns** an *action*

persistent: *KB*, a knowledge base
 t, a counter, initially 0, indicating time

 TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))

action \leftarrow Ask(*KB*, MAKE-ACTION-QUERY(*t*))

 TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))

t \leftarrow *t* + 1

return *action*

Knowledge bases consist of **sentences**. Sentences are expressed according to the syntax of the **representation language**. A **logic** also defines the semantics or meaning of sentences, defines the **truth** of a sentence with respect to each possible world. **N.B.: If a sentence α is true in a possible world m , we say that m satisfies α or m is a model of α .**

- **Logical entailment:** $\alpha \models \beta$ iff in every model in which α is true, β is also true. α is a stronger assertion than β ; it rules out more possible worlds. (follow from KB) e.g.: $x = 0 \models xy = 0$.

Consider two possible sentences:

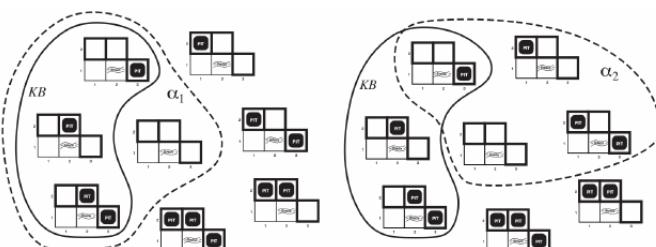
- α_1 =: "There is no pit in [1,2]" (true in models in dashed area below, left)
- α_2 =: "There is no pit in [2,2]" (true in models in dashed area below, right)

$\text{KB} \models \alpha_1$

- By inspection: in every model in which KB is true, α_1 is also true

$\text{KB} \not\models \alpha_2$

- In some models, in which KB is true, α_2 is false



- **Inference:** we can derive α with an inference method i. This is written as: $\text{KB} \vdash \alpha$. (Derivation)

We'd like to have inference algorithms that derive only sentences that are entailed (soundness) and all of them (completeness).

- **Declarative Languages:**

We state what we want to compute, not how. System believes P iff it considers P **true**. We must know symbols, when a sentence is true etc.

The building blocks of propositional logic are indivisible, **atomic statements**.

7.1.1 Syntax

Atom: atomic formula.

Literal: (possibly negated) atomic formula.

Clause: disjunction of literals.

The truth of a formula follows from the truth of its atomic propositions (truth assignment or interpretation) and the connectives.

A **truth assignment** of the atoms in Σ , or **Interpretation I** over Σ , is a function: $I : \Sigma \rightarrow T, F$.

$$I \models \top$$

$$I \not\models \perp$$

$$I \models P \quad \text{iff} \quad P^I = T$$

$$I \not\models \neg\varphi \quad \text{iff} \quad I \models \varphi$$

$$I \models \varphi \wedge \psi \quad \text{iff} \quad I \models \varphi \text{ and } I \models \psi$$

$$I \models \varphi \vee \psi \quad \text{iff} \quad I \models \varphi \text{ or } I \models \psi$$

$$I \models \varphi \Rightarrow \psi \quad \text{iff} \quad \text{if } I \models \varphi, \text{ then } I \models \psi$$

$$I \models \varphi \Leftrightarrow \psi \quad \text{iff} \quad \text{if } I \models \varphi \text{ if and only if } I \models \psi$$

I satisfies ϕ ($I \models \phi$) or ϕ is true under I, when $I(\phi) = T$. I can be seen as a 'possible world'.

An **Interpretation I** is called model of ϕ if $I \models \phi$. A truth assignment of the atoms in Σ , or an interpretation I over Σ , is a function: $I : \Sigma \rightarrow T, F$.

A formula ϕ can be: **satisfiable**, **unsatisfiable**, **falsifiable** (there exists I that doesn't satisfy ω), **valid** (tautology: $I \models \omega \forall I$), **logically equivalent** ($I \models \omega$ iff $I \models \chi \forall I$). A method to decide if a formula is satisfiable is: **Truth Table** (classic table in which we try all the possibilities and check whether it is false or true).

Goal: Find an algorithmic way to derive new knowledge out of a knowledge base.

1. Transform KB into a standardized representation.

2. Define rules that syntactically modify formulae while keeping semantic correctness.

7.1.2 Normal forms

- **conjunctive normal form (CNF):** conjunction of disjunctions:

$$\bigwedge_{i=1}^n \left(\bigvee_{j=1}^{m_i} l_{i,j} \right)$$

- **disjunctive normal form (DNF):** disjunction of conjunctions:

$$\bigvee_{i=1}^n \left(\bigwedge_{j=1}^{m_i} l_{i,j} \right)$$

It is always possible to transform a formula in a CNF/DNF but the conversion can make the sentence growing in an exponential way.

- **Producing CNF:**

- 1. Eliminate \rightarrow and \leftrightarrow : $\alpha \rightarrow \beta$ becomes $\neg\alpha \vee \beta$
- 2. Move \neg inwards: $\neg(\alpha \wedge \beta)$ becomes $\neg\alpha \vee \neg\beta$ (De Morgan's laws).
- 3. Distribute \vee over \wedge : $((\alpha \wedge \beta) \vee \gamma)$ becomes $(\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
- 4. Simplify: $\alpha \vee \alpha$ becomes α

A set of formulae (a KB) usually provides an incomplete description of the world, i.e., it leaves the truth values of certain propositions open.

Example: $KB = (P \vee Q) \wedge (R \vee \neg P) \wedge S$ is definitive with respect to S, but leaves P, Q, R open. **Some properties of logical implication:**

- **Deduction theorem:** $KB \cup \{\varphi\} \models \psi$ iff $KB \models \varphi \Rightarrow \psi$
- **Contraposition theorem:** $KB \cup \{\varphi\} \models \neg\psi$ iff $KB \cup \{\psi\} \models \neg\varphi$
- **Contradiction theorem:** $KB \cup \{\varphi\}$ is unsatisfiable iff $KB \models \neg\varphi$

Deduction theorem: $\text{KB} \cup \{\varphi\} \models \psi$ iff $\text{KB} \models \varphi \Rightarrow \psi$

“ \Rightarrow ” Assumption: $\text{KB} \cup \{\varphi\} \models \psi$, i.e., every model of $\text{KB} \cup \{\varphi\}$ is also a model of ψ .

Let I be any model of KB . If I is also a model of φ , then it follows that I is also a model of ψ .

This means that I is also a model of $\varphi \Rightarrow \psi$, i.e., $\text{KB} \models \varphi \Rightarrow \psi$.

“ \Leftarrow ” Assumption: $\text{KB} \models \varphi \Rightarrow \psi$. Let I be any model of KB that is also a model of φ , i.e., $I \models \text{KB} \cup \{\varphi\}$.

From the assumption, I is also a model of $\varphi \Rightarrow \psi$ and thereby also of ψ , i.e., $\text{KB} \cup \{\varphi\} \models \psi$.

Contraposition theorem: $\text{KB} \cup \{\varphi\} \models \neg\psi$ iff $\text{KB} \cup \{\psi\} \models \neg\varphi$

$$\begin{aligned} & \text{KB} \cup \{\varphi\} \models \neg\psi \\ & \text{iff } \text{KB} \models \varphi \Rightarrow \neg\psi \quad (1) \\ & \text{iff } \text{KB} \models (\neg\varphi \vee \neg\psi) \\ & \text{iff } \text{KB} \models (\neg\psi \vee \neg\varphi) \\ & \text{iff } \text{KB} \models \psi \Rightarrow \neg\varphi \\ & \text{iff } \text{KB} \cup \{\psi\} \models \neg\varphi \quad (2) \end{aligned}$$

Note:

(1) and (2) are applications of the deduction theorem.

We can often derive new formulae from formulae in the KB. These new formulae should follow logically from the syntactical structure of the KB formulae.

In the case where in the calculus C there is a proof for a formula ϕ , we write: $\text{KB} \vdash_C \phi$.

Calculus: Set of inference rules (potentially including so-called logical axioms). A calculus C is **sound (or correct)** if all formulae that are derivable from a KB actually follow logically. A calculus is **complete** if every formula that follows logically from the KB is also derivable with C from the KB.

7.1.3 Resolution

We want a way to derive new formulae that does not depend on testing every interpretation (idea: to prove that $\text{KB} \models \phi$, we can prove that $\text{KB} \cup \neg\phi$ is unsatisfiable (contradiction theorem)).

Assumption: All formulae in the KB are in CNF. Equivalently, we can assume that the KB is a set of clauses. **E.g.:** $\{(P \vee Q) \wedge (R \vee \neg P) \wedge S\}$ by $\{\{P, Q\}, \{R, \neg P\}, \{S\}\}$

Due to commutativity, associativity, and idempotence of \vee , clauses can also be understood as sets of literals. The empty set of literals is denoted by \perp (false).

An interpretation I satisfies C iff there exists $l \in C$ such that $I = l$. I satisfies Δ if for all $C \in \Delta : I = C$, i.e., $I \neq \perp$, $I \models C$, for all C .

- Resolution rule:

$$\frac{C_1 \dot{\cup} \{l\}, C_2 \dot{\cup} \{\bar{l}\}}{C_1 \cup C_2}$$

$C_1 \cup C_2$ are called **resolvents** of the **parent clauses** $C_1 \dot{\cup} \{l\}$ and $C_2 \dot{\cup} \{\bar{l}\}$. l and \bar{l} are the **resolution literals**.

Example: a, b, $\neg c$ resolves with a, d, c to a, b, d.

Note: The resolvent is not equivalent to the parent clauses, but it follows from them!

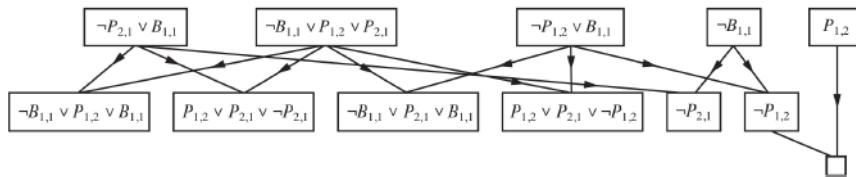
- **Resolution algorithm:**

```

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

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

Example:



Where is the Wumpus? Knowledge of the Situation

$B = \text{Breeze}$, $S = \text{Stench}$, $B_{i,j} = \text{there is a breeze in } (i, j)$

$\neg S_{1,1} \quad \neg B_{1,1}$
 $\neg S_{2,1} \quad B_{2,1}$
 $S_{1,2} \quad \neg B_{1,2}$

Knowledge about the wumpus and smell:

$R_1 : \neg S_{1,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1}$
 $R_2 : \neg S_{2,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{2,1} \wedge \neg W_{2,2} \wedge \neg W_{3,1}$
 $R_3 : \neg S_{1,2} \Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,2} \wedge \neg W_{1,3}$
 $R_4 : S_{1,2} \Rightarrow W_{1,3} \vee W_{1,2} \vee W_{2,2} \vee W_{1,1}$

To show: $\text{KB} \models W_{1,3}$

Lemma (soundness) If $\Delta \vdash D$, then $\Delta \models D$.

Proof idea: Since all $D \in R(\Delta)$ follow logically from Δ , the Lemma results through induction over the length of the derivation.

Is resolution also complete? Not in general.

However, it can be shown that resolution is refutation-complete: Δ is unsatisfiable implies $\Delta \vdash \text{empty}(\perp)$.

Theorem: Δ is unsatisfiable iff $\Delta \vdash \text{empty}(\perp)$.

We can now infer new facts, but how do we translate knowledge into action?

Negative selection: Excludes any provably dangerous actions.

$$A_{1,1} \wedge \text{East}_A \wedge W_{2,1} \rightarrow \neg \text{Forward}$$

Positive selection: Only suggests actions that are provably safe.

$$A_{1,1} \wedge \text{East}_A \wedge \neg W_{2,1} \rightarrow \text{Forward}$$

7.2 Summary

Rational agents require **knowledge of their world** in order to make rational decisions.

With the help of a **declarative** (knowledge-representation) language, this knowledge is represented and stored in a **knowledge base**.

We use **propositional logic** for this (for the time being).

Formulae of propositional logic can be **valid, satisfiable, or unsatisfiable**.

The concept of **logical implication** is important.

Logical implication can be mechanized by using an inference calculus, so: **resolution**.

Propositional logic quickly becomes impractical when the world becomes too large (or infinite).

8 Satisfiability and Model Construction

SAT solving is the best available technology for practical solutions to many NP-hard problems. Differently from Logical deduction, SAT returns a model of the theory (solution) given a logical theory. SAT can be formulated as a Constraint-Satisfaction-Problem (\rightarrow search):

- CSP-variables: alphabet.
- Domain: True, False.
- Constraints given by clauses. E.g.: $a = \text{true}$, $b \vee e \vee c = \text{true}$.

8.1 Davis-Putnam-Logemann-Loveland (DPLL) Procedure

It corresponds to backtracking with inference in CSPs. Inference in DPLL:

Simplify: if variable v is assigned a value d , then all clauses containing v are simplified immediately (corresponds to forward checking), variables in unit clauses are immediately assigned.

Given a set of clauses Δ defined over a set of variables Σ , return "satisfiable" if Δ is satisfiable. Otherwise return "unsatisfiable".

1. If $\Delta = \emptyset$ return "satisfiable"
2. If $\square \in \Delta$ return "unsatisfiable"
3. Unit-propagation Rule: If Δ contains a unit-clause C , assign a truth-value to the variable in C that satisfies C , simplify Δ to Δ' and return $\text{DPLL}(\Delta')$.
4. Splitting Rule: Select from Σ a variable v which has not been assigned a truth-value. Assign one truth value t to it, simplify Δ to Δ' and call $\text{DPLL}(\Delta')$.
 - a. If the call returns "satisfiable", then return "satisfiable".
 - b. Otherwise assign *the other* truth-value to v in Δ , simplify to Δ'' and return $\text{DPLL}(\Delta'')$.

- Examples:

$$\Delta = \{\{a, b, \neg c\}, \{\neg a, \neg b\}, \{c\}, \{a, \neg b\}\}$$

1. Unit-propagation rule: $c \mapsto T$
 $\{\{a, b\}, \{\neg a, \neg b\}, \{a, \neg b\}\}$
 2. Splitting rule:
- | | |
|--|--|
| 2a. $a \mapsto F$
$\{\{b\}, \{\neg b\}\}$ | 2b. $a \mapsto T$
$\{\{\neg b\}\}$ |
| 3a. Unit-propagation rule:
$b \mapsto T$
$\{\square\}$ | 3b. Unit-propagation rule: $b \mapsto F$
$\{\}$ |

$$\Delta = \{\{a, \neg b, \neg c, \neg d\}, \{b, \neg d\}, \{c, \neg d\}, \{d\}\}$$

1. Unit-propagation rule: $d \mapsto T$
 $\{\{a, \neg b, \neg c\}, \{b\}, \{c\}\}$
2. Unit-propagation rule: $b \mapsto T$
 $\{\{a, \neg c\}, \{c\}\}$
3. Unit-propagation rule: $c \mapsto T$
 $\{\{a\}\}$
4. Unit-propagation rule: $a \mapsto T$
 $\{\}$

DPLL is **complete, correct, and guaranteed to terminate**. DPLL constructs a model, if one exists. In general, DPLL requires **exponential time** (splitting rule!), **Heuristics are needed**. DPLL is polynomial on Horn clauses. **Horn Clauses constitute an important special case, since they require only polynomial runtime of DPLL.** A **Horn clause** is a clause with **maximally one positive literal**.

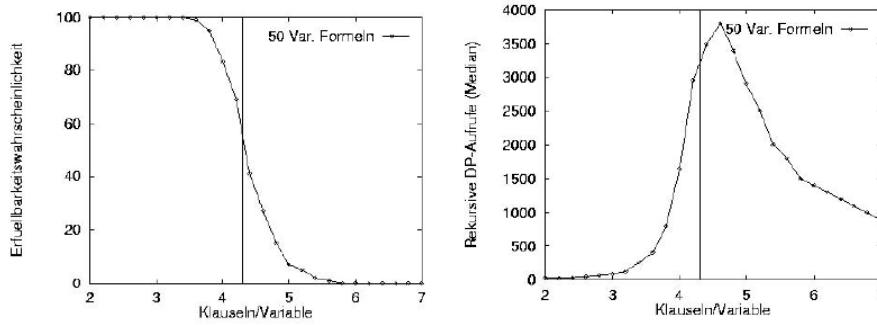
8.1.1 DPLL on Horn Clauses

Horn Clauses constitute an important special case, since they require only polynomial runtime of DPLL.

- 1. The simplifications in DPLL on Horn clauses always generate Horn clauses
- 2. If unit propagation rule in DPLL does not lead to termination, a set of Horn clauses without unit clauses is generated.
- 3. A set of Horn clauses without unit clauses and without the empty clause is satisfiable, since:
All clauses have at least one negative literal.
Assigning false to all variables satisfies formula.
- 4. It follows from 3.:
 - a. every time the splitting rule is applied, the current formula is satisfiable.
 - b. every time, when the wrong decision is made, this will be immediately detected.
- 5. Therefore, the search trees for n variables can only contain a maximum of n nodes, in which the splitting rule is applied (and the tree branches).
- 6. Therefore, the size of the search tree is only polynomial in n and therefore the running time is also polynomial.

8.2 Average Complexity of the Satisfiability Problem

For CNF-formulae, in which the probability for a positive appearance, negative appearance and non-appearance in a clause is 1/3, **DPLL needs on average quadratic time**. All NP-complete problems have at least one order parameter and the hard to solve problems are around a critical value of this order parameter. This critical value (a phase transition) separates one region from another, such as over-constrained and under-constrained regions of the problem space.



When the probability of a solution

is close to 1 (**under-constrained**), there are many solutions. If the probability of a solution is close to 0 (**over-constrained**).

8.3 Local Search

In many cases, we can sacrifice completeness if we can "solve" much larger instances this way. Standard process for optimization problems: **Local Search. Problem: local minima. However: By restarting and/or injecting noise, we can often escape local maxima.** Local search can perform very well for SAT solving.

Procedure GSAT

```

INPUT: a set of clauses  $\alpha$ , MAX-FLIPS, and MAX-TRIES
OUTPUT: a satisfying truth assignment of  $\alpha$ , if found
begin
  for  $i := 1$  to MAX-TRIES
     $T :=$  a randomly-generated truth assignment
    for  $j := 1$  to MAX-FLIPS
      if  $T$  satisfies  $\alpha$  then return  $T$ 
       $v :=$  a propositional variable such that a change in its
            truth assignment gives the largest increase in
            the number of clauses of  $\alpha$  that are satisfied by  $T$ 
       $T := T$  with the truth assignment of  $v$  reversed
    end for
  end for
  return "no satisfying assignment found"
end
```

8.4 State of the Art

8.4.1 Improvements of DPLL Algorithms

Branching on variables can cause stopping, we can "learn" (**here: logically infer**) a new clause (negation of a variable). Leads to conflict-directed clause learning (CDCL).

- Practical improvements:

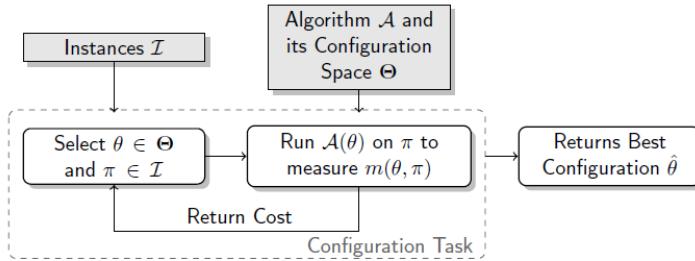
Both for **DPLL/CDCL** algorithms and local search algorithms

- Randomization and restarts
- Efficient data structures and indexing
- Engineering ingenious heuristics

Meta-algorithmic advances:

- Automated parameter tuning and algorithm configuration
- Selection of the best-fitting algorithm based on instance characteristics
- Selection of the best-fitting parameters based on instance characteristics
- Use of machine learning to pinpoint what factors most affects performance.

8.4.2 Algorithm configuration



Definition: algorithm configuration

Given:

- a parameterized algorithm \mathcal{A} with possible parameter settings Θ ;
- a distribution \mathcal{D} over problem instances with domain \mathcal{I} ; and
- a cost metric $m : \Theta \times \mathcal{I} \rightarrow \mathbb{R}$,

Find: $\theta^* \in \arg \min_{\theta \in \Theta} \mathbb{E}_{\pi \sim \mathcal{D}}(m(\theta, \pi))$.

- **Strength:**

find a single configuration with strong performance for a given cost metric.

- **Weakness:** for heterogeneous instance sets, there is often no configuration that performs great for all instances.

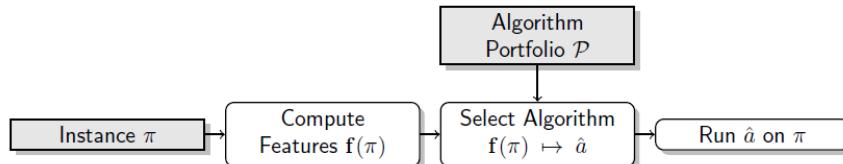
8.4.3 Algorithm selection

Definition: algorithm selection

Given

- a set \mathcal{I} of problem instances,
- a portfolio of algorithms \mathcal{P} ,
- and a cost metric $m : \mathcal{P} \times \mathcal{I} \rightarrow \mathbb{R}$,

the per-instance algorithm selection problem is to find a mapping $s : \mathcal{I} \rightarrow \mathcal{P}$ that optimizes $\sum_{\pi \in \mathcal{I}} m(s(\pi), \pi)$, the sum of cost measures achieved by running the selected algorithm $s(\pi)$ for instance π .



- **Strength:** for heterogeneous instance sets, pick the right algorithm from a set. - **Weakness:** the set to choose from typically only contains a few algorithms.

8.4.4 Automated construction of portfolios from a single algorithm

Putting the two together: Use algorithm configuration to determine useful configurations. Use algorithm selection to select from them based on instance characteristics.

Hydra

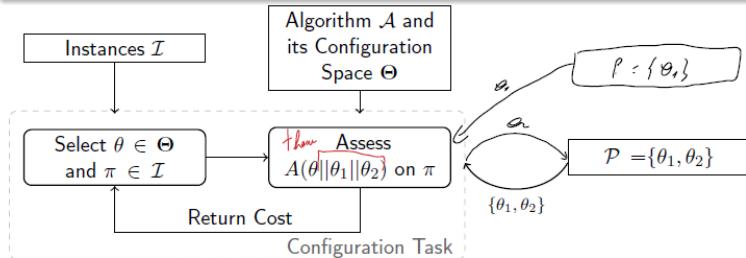
Idea

Iteratively add configurations to a portfolio \mathcal{P} , starting with $\mathcal{P} = \emptyset$

In each iteration, determine configuration that is complementary to \mathcal{P}

Maximize marginal contribution of configuration θ to current portfolio \mathcal{P} :

$$m(\mathcal{P}) - m(\mathcal{P} \cup \{\theta\})$$



8.5 Summary

DPLL: combines simplification, unit-propagation and backtracking: Very efficient implementation techniques, Good branching heuristics, Clause learning.

Incomplete randomized SAT-solvers: Perform best on random satisfiable problem instances.

State of the art: Typically obtained by automatic algorithm configuration & selection.

9 Predicate Logic (First-Order Predicate Logic (PL-1))

Propositional logic has no structure in the atomic propositions.

In addition to Operators, Variables, Brackets we have **Predicate and functions**. They have an arity (number of arguments): 0-ary predicate: propositional logic atoms, 0-ary function = constants.

Terms: Every variable is a term, If t_1, t_2, \dots, t_n are terms and f is an **n-ary** function, then $f(t_1, t_2, \dots, t_n)$ is also a term.
Terms without variables: ground terms.

Atomic Formulae: Represent statements about objects.

- If t_1, t_2, \dots, t_n are terms and P is an **n-ary** predicate, then $P(t_1, t_2, \dots, t_n)$ is an **atomic formula**.
- If t_1 and t_2 are terms, then $t_1 = t_2$ is an **atomic formula**. **Atomic formulae** without variables: **ground atoms** (contain only ground terms).

Propositional logic is part of the PL1 language:

- 1. Atomic formulae: only 0-ary predicates.
- 2. Neither variables nor quantifiers.

9.1 Semantics of PL1-Logic

Interpretation: $I = \langle D, x^I \rangle$ D domain, x^I is a function that:

maps n -ary function symbols to functions over D :

$$f^I \in [D^n \mapsto D]$$

maps individual constants to elements of D :

$$a^I \in D$$

maps n -ary predicate symbols to relations over D :

$$P^I \subseteq D^n$$

Interpretation of ground terms: $(f(t_1, \dots, t_n))^I = f^I(t_1^I, \dots, t_n^I)$.

Satisfaction of ground atoms $P(t_1, \dots, t_n)$: $I \models P(t_1, \dots, t_n)$ iff $t_1^I, \dots, t_n^I \in P^I$.

- Examples:

$$\begin{aligned}
 D &= \{d_1, \dots, d_n \mid n > 1\} \\
 \text{Interpret constant } a &\text{ as the element } d_1 \quad a^I = d_1 \\
 b^I &= d_2 \\
 c^I &= \dots \\
 \text{Block}^I &= \{d_1\} \\
 \text{Red}^I &= D \quad \text{Satisfy} \\
 I \models \text{Red}(b) &\Rightarrow b^I \in \text{Red}^I \\
 I \not\models \text{Block}(b) &\quad \text{d}_1 \notin D \quad \checkmark \\
 \text{d}_1 \text{ not} &\quad \text{d}_1^I \stackrel{?}{\in} \text{Block}^I \\
 \text{d}_1 \in \{d_1\} &\quad \checkmark
 \end{aligned}
 \qquad
 \begin{aligned}
 D &= \{1, 2, 3, \dots\} \\
 1^I &= 1 \\
 2^I &= 2 \\
 \dots & \\
 \text{Even}^I &= \{2, 4, 6, \dots\} \\
 \text{succ}^I &= \{(1 \mapsto 2), (2 \mapsto 3), \dots\} \\
 I \models \text{Even}(2) & \\
 I \not\models \text{Even}(\text{succ}(2)) & \quad \text{Observe}
 \end{aligned}$$

9.1.1 Variable assignment

Set of all variables V . Function $\alpha : V \mapsto D$

Notation: $\alpha[x/d]$ is the same as α apart from point x .

For $x : \alpha[x/d](x) = d$.

Interpretation of terms under I, α :

$$\begin{aligned} x^{I,\alpha} &= \alpha(x) \\ a^{I,\alpha} &= a^I \\ (f(t_1, \dots, t_n))^{I,\alpha} &= f^I(t_1^{I,\alpha}, \dots, t_n^{I,\alpha}) \end{aligned}$$

Satisfaction of atomic formulae:

$$I, \alpha \models P(t_1, \dots, t_n) \text{ iff } \langle t_1^{I,\alpha}, \dots, t_n^{I,\alpha} \rangle \in P^I$$

$$\text{Block}^I = \{d_1\}$$

$$\text{Red}^I = D$$

$$\alpha = \{(x \mapsto d_1), (y \mapsto d_2)\}$$

$$I, \alpha \models \text{Red}(x)$$

$$I, \alpha[y/d_1] \models \text{Block}(y)$$

$$\begin{array}{c} y \models_{I, \alpha[y/d_1]} \text{Block}(y) ? \\ \alpha[y/d_1](y) \notin \text{Block}^I \end{array}$$

$$\begin{array}{l} x \models_{I, \alpha} ? \\ x \in \text{Red}^I \\ \alpha(x) \in \text{Red}^I \\ d_1 \in \text{Red}^I \end{array}$$

9.1.2 Satisfiability

A formula φ is satisfied by an interpretation I and a variable assignment α , i.e., $I, \alpha \models \varphi$:

$$\begin{aligned} I, \alpha &\models \top \\ I, \alpha &\not\models \perp \\ I, \alpha &\models \neg\varphi \text{ iff } I, \alpha \not\models \varphi \\ &\dots \end{aligned}$$

and all other propositional rules as well as

$$\begin{aligned} I, \alpha \models P(t_1, \dots, t_n) &\text{ iff } \langle t_1^{I,\alpha}, \dots, t_n^{I,\alpha} \rangle \in P^I \\ I, \alpha \models \forall x \varphi &\text{ iff for all } d \in D, I, \alpha[x/d] \models \varphi \\ I, \alpha \models \exists x \varphi &\text{ iff there exists a } d \in D \text{ with } I, \alpha[x/d] \models \varphi \end{aligned}$$

$$\begin{aligned} D &= \{d_1, \dots, d_n \mid n > 1\} \\ a^I &= d_1 \\ b^I &= d_2 \\ \text{Block}^I &= \{d_1\} \\ \text{Red}^I &= D \\ \alpha &= \{(x \mapsto d_1), (y \mapsto d_2)\} \end{aligned}$$

Questions:

1. $I, \alpha \models \text{Block}(b) \vee \neg\text{Block}(b)$?
2. $I, \alpha \models \text{Block}(x) \Rightarrow (\text{Block}(x) \vee \neg\text{Block}(y))$?
3. $I, \alpha \models \text{Block}(a) \wedge \text{Block}(b)$?
4. $I, \alpha \models \forall x (\text{Block}(x) \Rightarrow \text{Red}(x))$?

9.2 Free and Bound Variables

$$\forall x [R(\boxed{y}, \boxed{z}) \wedge \exists y ((\neg P(y, x) \vee R(y, \boxed{z}))]$$

When boxed, is called *free*, otherwise **bound**. Formulae with no free variables are called *closed formulae* or *sentences*.

Note: with closed formulae, the concepts logical equivalence, satisfiability, and implication, etc. are not dependent on the variable assignment α (i.e., we can always ignore all variable assignments), and α can be left out on the left side of the model relationship symbol: $I = \omega$. An interpretation I is called a **model** of ω under α if: $I, \alpha \models \omega$.

- **Terminology:**

An interpretation I is called a **model** of ϕ under α if $I, \alpha \models \phi$.

A PL1 formula ϕ can, as in propositional logic, be satisfiable, unsatisfiable, falsifiable, or valid. Analogously, two formulae are logically equivalent ($\phi \equiv \psi$) if for all I, α : $I, \alpha \models \psi$ iff $I, \alpha \models \phi$.

Note: $P(x) \neq P(y)$!

Logical Implication is also analogous to propositional logic.

9.3 Reduction to Propositional Theories

9.3.1 Derivation in PL1

Reduction to propositional logic by instantiation based on the so-called **Herbrand Universe** (all possible terms) \rightarrow infinite propositional theories.

Simple way for special case: If the number of objects is **finite**, instantiate all variables by possible objects.

9.3.2 Finite universes

Domain closure axiom (DCA): $\forall[x = c_1 \vee \dots \vee x = c_n]$

unique name assumption/axiom or UNA: $\wedge_{i \neq j} [c_i \neq c_j]$

Eliminate quantification by instantiating all variables with all possible values.

9.3.3 Instantiation

Notation: if φ is a formula, then $\varphi[x/a]$ is the formula with all free occurrences of x replaced by a .

Universally quantified formulas are replaced by a conjunction of formulas with the variable instantiated to all possible values (from DCA):

$$\forall x \varphi \rightsquigarrow \bigwedge_i \varphi[x/c_i]$$

Existentially quantified variables are replaced by a disjunction of formulas with the variable instantiated to all possible values (from DCA):

$$\exists x \varphi \rightsquigarrow \bigvee_i \varphi[x/c_i]$$

Note: does blow up the formulas exponentially in the **arity** of the predicates!

$$\forall x \ (Block(x) \Rightarrow Red(x))$$

$$\forall x \ (x = a \vee x = b \vee x = c)$$

\rightsquigarrow

$$(Block(a) \Rightarrow Red(a)) \wedge$$

$$(Block(b) \Rightarrow Red(b)) \wedge$$

$$(Block(c) \Rightarrow Red(c))$$

9.4 Summary

PL1 makes it possible to structure statements, thereby giving us considerably **more expressive power than propositional logic**.

Logical implication in PL1 is **undecidable**.

If we only reason over a **finite universe**, **PL1 can be reduced to propositional logic** over finite theories (but the reduction is exponential in the arity of the predicates).

10 Action Planning

Planning is the art and practice of thinking before acting.

Planning is the process of generating (possibly partial) representations of future behavior prior to the use of such plans to constrain or control that behavior.

Planning is not problem solving by search, program synthesis or scheduling. Of course, there is interaction with them.

10.1 Planning Formalisms

10.1.1 Domain-Independent Action Planning

- Start with a declarative specification of the planning problem
- Use a domain-independent planning system to solve the planning problem
- Domain-independent planners are generic problem solvers, good for evolving systems and where performance is not critical.

10.1.2 Planning as Logical Inference

Planning can be elegantly formalized with the help of the situation calculus.

Initial state:

$At(truck1, loc1, s_0) \wedge At(package1, loc3, s_0)$

Operators (successor-state axioms):

$\forall a, s, l, p, t \ At(t, p, Do(a, s)) \Leftrightarrow \{a = Drive(t, l, p) \wedge Poss(Drive(t, l, p), s)$
 $\vee At(t, p, s) \wedge (a \neq \neg Drive(t, p, l, s) \vee \neg Poss(Drive(t, p, l), s))\}$

Goal conditions (query):

$\exists s \ At(package1, loc2, s)$

Inefficient for bigger sets.

10.1.3 Basic STRIPS Formalism

STRIPS: STanford Research Institute Problem Solver

\mathcal{S} is a *first-order vocabulary* (predicate and function symbols) and $\Sigma_{\mathcal{S}}$ denotes the set of *ground atoms* over the signature (also called **facts** or **fluents**).

$\Sigma_{\mathcal{S}, V}$ is the set of atoms over \mathcal{S} using variable symbols from the set of variables V .

A **first-order STRIPS state** S is a subset of $\Sigma_{\mathcal{S}}$ denoting a *complete theory* or *model* (using CWA).

A **planning task** (or **planning instance**) is a 4-tuple

$\Pi = \langle \mathcal{S}, O, I, G \rangle$, where

- O is a set of **operator** (or *action types*)
- $I \subseteq \Sigma_{\mathcal{S}}$ is the **initial state**
- $G \subseteq \Sigma_{\mathcal{S}}$ is the **goal specification**

No domain constraints (although present in original formalism)

- **Operator:** $o = <\text{para}, \text{pre}, \text{eff}>$
- **Operator instance or action:** operator with empty parameter list

$$App(S, o) = \begin{cases} S \cup eff^+(o) - \neg eff^-(o) & \text{if } pre(o) \subseteq S \text{ \&} \\ & \quad eff(o) \text{ is cons.} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- **State change:**

- Example logistics:

Logical atoms: $at(O, L)$, $in(O, V)$, $airconn(L1, L2)$, $street(L1, L2)$,
 $plane(V)$, $truck(V)$

Load into truck: $load$

Parameter list: (O, V, L)

Precondition: $at(O, L), at(V, L), truck(V)$

Effects: $\neg at(O, L), in(O, V)$

Drive operation: $drive$

Parameter list: $(V, L1, L2)$

Precondition: $at(V, L1), truck(V), street(L1, L2)$

Effects: $\neg at(V, L1), at(V, L2)$

...

Some constant symbols: $v1, s, t$ with $truck(v1)$ and $street(s, t)$

Action: $drive(v1, s, t)$

10.1.4 Plans & Successful Executions

A plan Δ is a sequence of actions:

$$Res(S, \langle \rangle) = S$$

$$Res(S, (o; \Delta)) = \begin{cases} Res(App(S, o), \Delta) & \text{if } App(S, o) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

A plan is successful or solves a planning task if has a result that respects goal specifications.

- Example logistics:

Initial state: $S = \{ at(p1, c), at(p2, s), at(t1, c), at(t2, c), street(c, s), street(s, c) \}$

Goal: $G = \{ at(p1, s), at(p2, c) \}$

Successful plan: $\Delta = \langle \overbrace{load(p1, t1, c)}, \overbrace{drive(t1, c, s)}, \overbrace{\cancel{unload(p1, t1, s)}}, \overbrace{\cancel{load(p2, t1, s)}}, \overbrace{\cancel{drive(t1, s, c)}}, \overbrace{\cancel{unload(p2, t1, c)}} \rangle$

STRIPS as described above allows for unrestricted first-order terms.

Simplifications:

- **1. Infinite state space:** No function terms (only 0-ary = constants).
- **2. DATALOG-STRIPS:** No variables in operators (= actions).
- **3. Propositional STRIPS:** used in planning algorithms nowadays.

10.1.5 PDDL: The Planning Domain Description Language

- Example logistics:

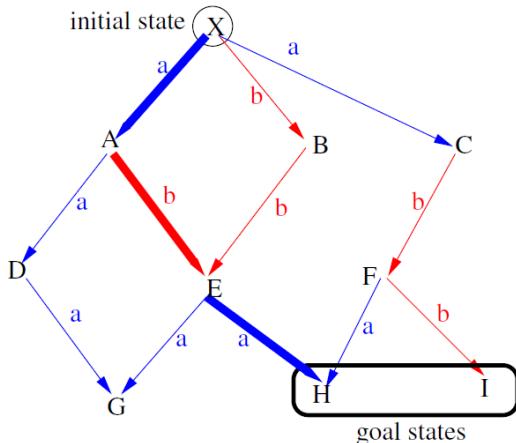
```
(define (domain logistics)
  (:types truck airplane - vehicle
         package vehicle - physobj
         airport location - place
         city place physobj - object)

  (:predicates (in-city ?loc - place ?city - city)
               (at ?obj - physobj ?loc - place)
               (in ?pkg - package ?veh - vehicle))

  (:action LOAD-TRUCK
    :parameters (?pkg - package ?truck - truck ?loc - place)
    :precondition (and (at ?truck ?loc) (at ?pkg ?loc))
    :effect      (and (not (at ?pkg ?loc)) (in ?pkg ?truck)))
    ...))
```

10.2 Basic Planning Algorithms

We can view planning problems as searching for goal nodes in a large **labeled graph** (transition system). **Nodes** are defined by the value assignment to the **fluents** (states). **Labeled edges** are defined by **actions**. Create the transition system on the fly and visit only the parts that are necessary.

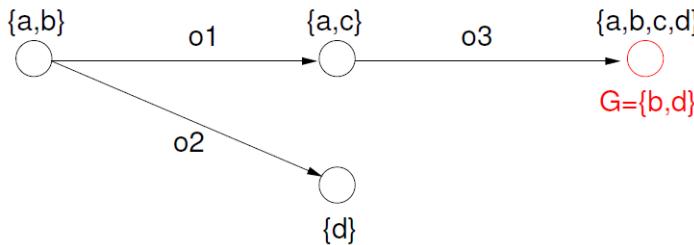


- **Progression Planning: Forward Search** Start at initial state:

1. Initialize: $\Delta = \langle \rangle$, add the initial state and make it S .
2. Test whether we have reached goal: $G \in S$? Return Δ .
3. Select one applicable action a_i non deterministically, compute successor $S = App(S, o_i)$, extend plan adding it and come back to step 2.

This algorithm can be easily extended to more expressive planning languages (not only boolean: goal or not).

$$\begin{aligned}
S &= \{a, b, c, d\}, \\
O &= \{ o_1 = \langle \emptyset, \{a, b\}, \{\neg b, c\} \rangle, \\
&\quad o_2 = \langle \emptyset, \{a, b\}, \{\neg a, \neg b, d\} \rangle, \\
&\quad o_3 = \langle \emptyset, \{c\}, \{b, d\} \rangle, \\
I &= \{a, b\} \\
G &= \{b, d\}
\end{aligned}$$



- **Regression Planning: Backward Search** Start from the goal:

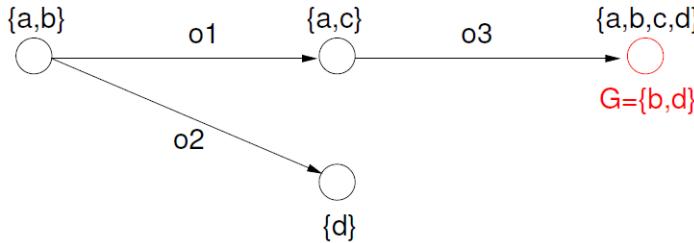
1. Initialize: $\Delta = \langle \rangle$, add the goal state and make it S .
2. Test whether we have reached initial: I in S ? Return.
3. Select one applicable action o_i non deterministically which does not make sub-goals false, $S \cap \neg eff^-(o_i) = \emptyset$ and compute the regression of the description S through o_i :

$$S = S - eff^+(o_i) \cup pre(o_i)$$

then extend plan $\Delta = \langle o_i, \Delta \rangle$, back to step 2.

(Instead of non-deterministic we can use some search strategy).

$$\begin{aligned}
S &= \{a, b, c, d\}, \\
O &= \{ o_1 = \langle \emptyset, \{a, b\}, \{\neg b, c\} \rangle, \\
&\quad o_2 = \langle \emptyset, \{a, b\}, \{\neg a, \neg b, d\} \rangle, \\
&\quad o_3 = \langle \emptyset, \{c\}, \{b, d\} \rangle, \\
I &= \{a, b\} \\
G &= \{b, d\}
\end{aligned}$$



10.3 Computational Complexity

10.3.1 The planning problem

- **Definition (Plan existence problem (PLANEX)):** Does exist a plan that solve it?
- **Definition (Bounded plan existence problem (PLANLEN)):** Doest there exist a plan of length n or less that solves it?

The state space for STRIPS with general first-order terms is infinite. The existence of a plan is then equivalent to the existence of a successful computation on the Turing machine.

Theorem: PLANEX for STRIPS with first-order terms is undecidable.

Theorem: PLANEX is PSPACE-complete for propositional STRIPS.

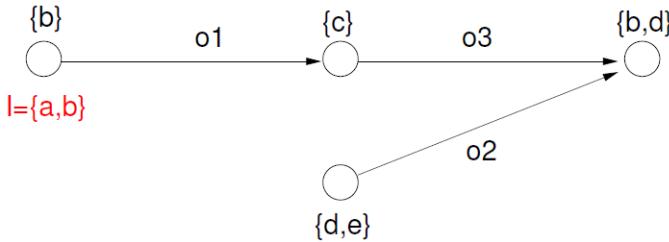
10.3.2 Restrictions on Plans

If we restrict the length of the plans to be short, i.e., only polynomial in the size of the planning task, PLANEX becomes NP-complete. If we use a unary representation of the natural number k , then PLANLEN becomes NP-complete. We can use methods for NP-complete problems if we are only looking for “short” plans.

10.4 Current Algorithmic Approaches

- **Heuristic Search Planning:** use an automatically generated **heuristic estimator** in order to select the next action or state. It is often easier to go for sub-optimal solutions (remember Logistics).
- **Deriving Heuristics: Relaxations** Define a **simplification** (relaxation) of the problem and take the difficulty of a solution for the simplified problem as an heuristic estimator. (**E.g.: Straight line distance on a map to estimate the travel distance**).
- **Ignoring Negative Effects: Example**

$$\begin{aligned} \mathcal{S} &= \{a, b, c, d, e\}, \\ \mathbf{O} &= \{ o_1 = \langle \emptyset, \{b\}, \{\neg b, c\} \rangle, \\ &\quad o_2 = \langle \emptyset, \{e\}, \{b\} \rangle, \\ &\quad o_3 = \langle \emptyset, \{c\}, \{b, d, \neg e\} \rangle, \\ \mathbf{I} &= \{a, b\} \\ \mathbf{G} &= \{b, d\} \end{aligned}$$



10.5 Summary

Rational agents need to plan their course of action.

In order to describe planning tasks in a domain-independent, declarative way, one needs **planning formalisms**.

Basic STRIPS is a simple planning formalism, where actions are described by their preconditions in form of a conjunction of atoms and the effects are described by a list of literals that become true and false.

PDDL is the current “standard language” that has been developed in connection with the international planning competition Basic planning algorithms search through the space created by the transition system or through the **plan space**.

Planning with **STRIPS** using first-order terms is undecidable.

Planning with propositional STRIPS is **PSPACE-complete**.

Since 1992, we have reasonably efficient planning method for **propositional, classical STRIPS planning**.

You can learn more about it in our **planning class** next term.

11 Making simple decisions under uncertainty (Probability)

In many cases, our knowledge of the world is incomplete (not enough information) or uncertain (sensors are unreliable). **Without perfect knowledge, logical rules do not help much!**

One possibility for expressing the degree of belief is to use probabilities. **Probabilities quantify the uncertainty that stems from lack of knowledge.**

We have a choice of actions (or plans).

These can lead to different results (worlds) with different probabilities.

The actions have different (subjective) costs.

The results have different (subjective) utilities.

It would be rational to choose the action with the maximum expected total utility!

$$\text{Decision Theory} = \text{Utility Theory} + \text{Probability Theory}$$

$$\operatorname{argmax}_a \sum_{\omega} p(\omega|a)[U(\omega) - c(a)]$$

- Decision-Theoretic Agent

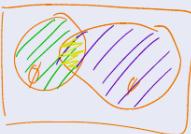
```
function DT-AGENT(percept) returns an action
  persistent: belief_state, probabilistic beliefs about the current state of the world
  action, the agent's action
  update belief_state based on action and percept
  calculate outcome probabilities for actions,
    given action descriptions and current belief_state
  select action with highest expected utility
    given probabilities of outcomes and utility information
  return action
```

11.1 Foundations of Probability Theory

Axioms of Probability Theory

A function P mapping from formulae in propositional logic to the set $[0, 1]$ is a **probability measure** if for all propositions ϕ, ψ (whereby propositions are the equivalence classes formed by logically equivalent formulae):

- ① $0 \leq P(\phi) \leq 1$
- ② $P(\text{true}) = 1$
- ③ $P(\text{false}) = 0$
- ④ $P(\phi \vee \psi) = P(\phi) + P(\psi) - P(\phi \wedge \psi)$



All other properties can be derived from these axioms, for example:

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

since $1 = P(\phi \vee \neg\phi) = P(\phi) + P(\neg\phi) - P(\phi \wedge \neg\phi) = P(\phi) + P(\neg\phi)$.

We use random variables such as Weather (capitalized word), which has a domain of ordered values. In our case that could be sunny, rain, cloudy, snow (lower case words).

- **Unconditional Probabilities:** $P(a)$ denotes the unconditional probability that it will turn out that $A = \text{true}$ in the absence of any other information.
- **Conditional Probabilities:** $P(a | b) = \frac{P(a \wedge b)}{P(b)}$ is the conditional or posterior probability of a given that all we know is b : $P(\text{cavity}|\text{toothache}) = 0.8$
- **- Product rule:** $P(a \wedge b) = P(a|b)P(b)$
- a and b are **independent** iff $P(a|b) = P(a)$
- $P(X, Y) = P(X|Y)P(Y)$ corresponds to a system of equations.

11.2 Probabilistic Inference

11.2.1 Joint Probability

The joint probability distribution $P(X_1, \dots, X_n)$ assigns a probability to every atomic event. Example of such a complete instantiation:

	<i>toothache</i>	$\neg\text{toothache}$
<i>cavity</i>	0.04	0.06
$\neg\text{cavity}$	0.01	0.89

Observe: The sum of all fields is 1 (disjunction of events). Since all atomic events are disjoint, the conjunction of any two atomic events is necessarily *false*.

All relevant probabilities can be computed using the joint probability by expressing them as a disjunction of atomic events. Examples:

$$P(\text{cavity} \vee \text{toothache}) = P(\text{cavity} \wedge \text{toothache}) + P(\neg\text{cavity} \wedge \text{toothache}) + P(\text{cavity} \wedge \neg\text{toothache})$$

as this *transition.*

only this

We obtain **marginal probabilities** by adding across a row or column:

$$P(\text{cavity}) = P(\text{cavity} \wedge \text{toothache}) + P(\text{cavity} \wedge \neg\text{toothache})$$

We obtain **conditional probabilities** by using a marginal probability:

$$P(\text{cavity} | \text{toothache}) = \frac{P(\text{cavity} \wedge \text{toothache})}{P(\text{toothache})} = \frac{0.04}{0.04 + 0.01} = 0.80$$

- **Total probability:** $P(Y) = \sum_z P(Y, z) = \sum_z P(Y|z)P(z)$

We can easily obtain all probabilities from the joint probability. The joint probability, however, involves kn values, if there are n random variables with k values.

$$\begin{aligned} P(x_1, \dots, x_n) &= P(x_n, \dots, x_1) = P(x_n | x_{n-1}, \dots, x_1) P(x_{n-1}, \dots, x_1) \\ &= P(x_n | x_{n-1}, \dots, x_1) P(x_{n-1} | x_{n-2}, \dots, x_1) P(x_{n-2}, \dots, x_1) \\ &= P(x_n | x_{n-1}, \dots, x_1) P(x_{n-1} | x_{n-2}, \dots, x_1) P(x_{n-2} | x_{n-3}, \dots, x_1) \\ &\quad P(x_{n-3}, \dots, x_1) \\ &= \dots \\ &= P(x_n | x_{n-1}, \dots, x_1) P(x_{n-1} | x_{n-2}, \dots, x_1) \dots P(x_2 | x_1) P(x_1) \\ &= \prod_{i=1}^n P(x_i | x_{i-1}, \dots, x_1) \end{aligned}$$

In order to save memory and time, Modern systems work directly with conditional probabilities and make assumptions on the independence of variables (!conditional independence) to simplify calculations.

11.2.2 Bayes' Rule

Deriving from Product rule:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \quad (1)$$

Generalizing:

$$P(Y|X, e) = \frac{P(X|Y, e)P(Y|e)}{P(X|e)} \quad (2)$$

- Example:

$$\begin{aligned} P(\text{toothache} \mid \text{cavity}) &= 0.4 \\ P(\text{cavity}) &= 0.1 \\ P(\text{toothache}) &= 0.05 \\ \rightarrow P(\text{cavity} \mid \text{toothache}) &= \frac{0.4 \times 0.1}{0.05} = 0.8 \end{aligned}$$

$P(\text{toothache} \mid \text{cavity})$ (causal) is more robust than $P(\text{cavity} \mid \text{toothache})$ (diagnostic).

11.2.3 Relative Probability

Let's say we would also like to consider the probability that our patient has gum disease.

$$\begin{aligned} P(\text{toothache} \mid \text{gumdisease}) &= 0.7 \\ P(\text{gumdisease}) &= 0.02 \end{aligned}$$

Which diagnosis is more probable? Cavity or gum disease?

$$P(c \mid t) = \frac{P(t \mid c)P(c)}{P(t)} \quad \text{or} \quad P(g \mid t) = \frac{P(t \mid g)P(g)}{P(t)}$$

If we are only interested in the relative probability, we need not assess $P(t)$:

$$\begin{aligned} \frac{P(c \mid t)}{P(g \mid t)} &= \frac{P(t \mid c)P(c)}{P(t)} \times \frac{P(t)}{P(t \mid g)P(g)} = \frac{P(t \mid c)P(c)}{P(t \mid g)P(g)} \\ &= \frac{0.4 \times 0.1}{0.7 \times 0.02} = 2.857 \end{aligned}$$

→ We elegantly excluded other possible diagnoses for toothache.

11.2.4 Normalization

To compute absolute probability of $P(c|t)$ without knowing $P(t)$ we can do a complete case analysis using the property that $P(c|t) + P(\neg c|t) = 1$.

$$\begin{aligned} P(c \mid t) &= \frac{P(t \mid c)P(c)}{P(t)} \\ P(\neg c \mid t) &= \frac{P(t \mid \neg c)P(\neg c)}{P(t)} \\ P(c \mid t) + P(\neg c \mid t) &= \frac{P(t \mid c)P(c)}{P(t)} + \frac{P(t \mid \neg c)P(\neg c)}{P(t)} \\ P(t) &= P(t \mid c)P(c) + P(t \mid \neg c)P(\neg c) \end{aligned}$$

In general: $P(Y|X) = \alpha P(X|Y)P(Y)$

α is the normalization constant needed to make the entries in $P(X|Y)$ sum to 1 for each X. (e.g.: $\alpha(0.1, 0.1, 0.3) = (0.2, 0.2, 0.6)$)

Your doctor tells you that you have tested positive for a serious but rare (1/10000) disease. This test (t) is correct to 99% (1% false positive & 1% false negative results).

What does this mean for you?

$$P(d | t) = \frac{P(t | d)P(d)}{P(t)}$$

$$P(d) = 0.0001 \quad P(t | d) = 0.99 \quad P(t | \neg d) = 0.01$$

$$\begin{aligned} P(d | t) &= \frac{0.99 \times 0.0001}{0.99 \times 0.0001 + 0.01 \times 0.9999} = \frac{0.000099}{0.000099 + 0.009999} \\ &= \frac{0.000099}{0.010088} \approx 0.01 \end{aligned}$$

11.2.5 Multiple Evidence

A probe by the dentist catches (Catch = true) in the aching tooth (Toothache = true) of a patient. We already know that $P(\text{cavity} \rightarrow \text{toothache}) = 0.8$. Furthermore, using Bayes' rule, we can calculate: $\mathbf{P}(\text{cavity} \rightarrow \text{catch}) = 0.95$

$$\begin{aligned} P(\text{cav} | \text{tooth} \wedge \text{catch}) &= \frac{P(\text{tooth} \wedge \text{catch} | \text{cav}) \times P(\text{cav})}{P(\text{tooth} \wedge \text{catch})} \\ &= \alpha P(\text{tooth} \wedge \text{catch} | \text{cav}) \times P(\text{cav}) \end{aligned}$$

The dentist needs $P(\text{tooth} \wedge \text{catch} | \text{cav})$, i.e., diagnostic knowledge of all combinations of symptoms in the general case. They are conditionally independent given that we know whether the tooth has a cavity:

$$P(\text{tooth} | \text{catch}, \text{cav}) = P(\text{tooth} | \text{cav})$$

If one already knows that there is a cavity, then the additional knowledge of the probe catching does not change the probability.

$$P(\text{tooth} \wedge \text{catch} | \text{cav}) = P(\text{tooth} | \text{catch}, \text{cav})P(\text{catch} | \text{cav}) =$$

$$P(\text{tooth} | \text{cav})P(\text{catch} | \text{cav})$$

The general definition of **conditional independence** of two variables X and Y given a third variable Z (a common cause) is:

$$P(X, Y | Z) = P(X | Z)P(Y | Z) \quad (3)$$

Conditional Independence - Further Example

Eating icecream and observing sunshine is not independent

$$P(\text{ice} | \text{sun}) \neq P(\text{ice})$$

The variables *Ice* and *Sun* are not independent.

But if the reason for eating icecream is simply that it is hot outside, then the additional observation of sunshine does not make a difference:

$$P(\text{ice} | \text{sun}, \text{hot}) = P(\text{ice} | \text{hot})$$

The variables *Ice* and *Sun* are conditionally independent given that *Hot = true* is observed.

The knowledge about independence often comes from insight of the domain and is part of the modelling of the problem. Conditional independence can often be exploited to make things simpler (see later).

11.2.6 Recursive Bayesian Updating

Problem: we would like to avoid calculating the full joint probability table.

Assuming conditional independence, multiple evidence can be reduced to prior probabilities and conditional probabilities. The general combination rule, if Z_1 and Z_2 are independent given X is $P(X|Z_1, Z_2) = \alpha P(X)P(Z_1|X)P(Z_2|X)$ where α is the normalization constant.

Generalization: $P(X|Z_1, \dots, Z_n) = \alpha P(X)/PIP(Z_i|X)$ (**Recursive Bayesian Updating**).

11.3 Bayesian Network

Example of Burglary/Earthquake.

- Domain knowledge/ assumptions:

Events *Burglary* and *Earthquake* are independent. *Alarm* might be activated by burglary or earthquake. *John* calls if and only if he heard the alarm. His call probability is not influenced by the fact, that there is an earthquake at the same time. Same for *Mary*.

The *random variables* are the *nodes*. Directed edges between nodes represent **direct influence**. A **table of conditional probabilities (CPT)** is associated with every node, in which the effect of the parent nodes is quantified. The graph is acyclic (a **DAG**).

$$\begin{aligned} P(\text{maryCalls}|\text{alarm}, \text{burglary}) &= \\ P(\text{maryCalls}|\text{alarm}) \text{and } P(\text{maryCalls}|\text{alarm}, \text{burglary}, \text{johnCalls}, \text{earthquake}) \\ &= P(\text{maryCalls}|\text{alarm}) \end{aligned}$$

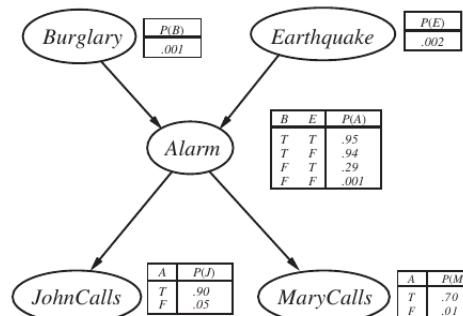
-> Bayesian Networks can be considered as sets of (conditional) independence assumptions.

- General formula:

$$P(x_1, \dots, x_n) = P(x_n|x_{n-1}, \dots, x_1) \cdot \dots \cdot P(x_2|x_1)P(x_1) = \\ \prod_{i=1}^n P(x_i|x_{i-1}, \dots, x_1)$$

This is equivalent to:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i|\text{parents}(x_i))$$



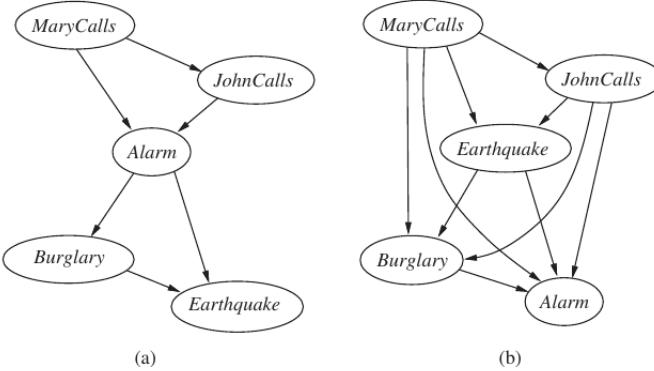
$$\begin{aligned} P(j, m, a, b, \neq) &= P(j|m, a, b, \neq)P(m|a, b, \neq)P(a|b, \neq)P(b|\neq)P(\neq) \\ &= P(j|a)P(m|a)P(a|b, \neq)P(b)P(\neq) \\ &= 0.90 \cdot 0.70 \cdot 0.001 \cdot 0.999 \cdot 0.998 = 0.00062 \end{aligned}$$

In general, we need a table of size 2^n where n is the number of variables. The size depends on the application domain (local vs. global interaction) and the skill of the designer.

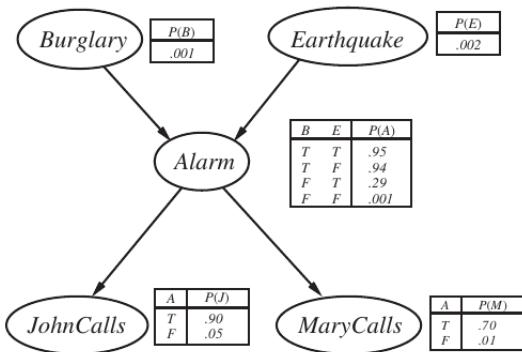
- Naive Networks method :

- Order all variables
- Take the first from those that remain
- Assign all direct influences from nodes already in the network to the new node (Edges + CPT).
- If there are still variables in the list, repeat from step 2.

left = M, J, A, B, E , right = M, J, E, B, A



- Inference in Bayesian Network



$$P(\text{burglary} \mid \text{johncalls})$$

$$P(\text{burglary} \mid \text{johnCalls}, \text{maryCalls})?$$

A node is conditionally independent of its non-descendants given its parents.

Exact Inference in Bayesian Networks

- Compute the posterior probability distribution for a set of query variables X given an observation, i.e., the values of a set of evidence variables E .
- Complete set of variables is $X \cup E \cup Y$
- Y are called the hidden variables
- Typical query $P(X \mid e)$ where e are the observed values of E .
- In the remainder: X is a singleton
- Example:
 $P(\text{Burglary} \mid \text{JohnCalls} = \text{true}, \text{MaryCalls} = \text{true}) = (0.284, 0.716)$

- Inference by Enumeration:

- $P(X|e) = \alpha P(X, e) = \sum_y \alpha P(Xe, y)$
- The network gives a complete representation of the full joint distribution.

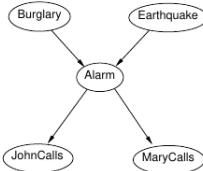
- A query can be answered using a Bayesian network by computing sums of products of conditional probabilities from the network.
- We sum over the hidden variables.

- Example:

Consider $\mathbf{P}(Burglary \mid JohnCalls = true, MaryCalls = true)$

The evidence variables are *JohnCalls* and *MaryCalls*.

The hidden variables are *Earthquake* and *Alarm*.



We have: $\mathbf{P}(B \mid j, m) = \alpha \mathbf{P}(B, j, m)$

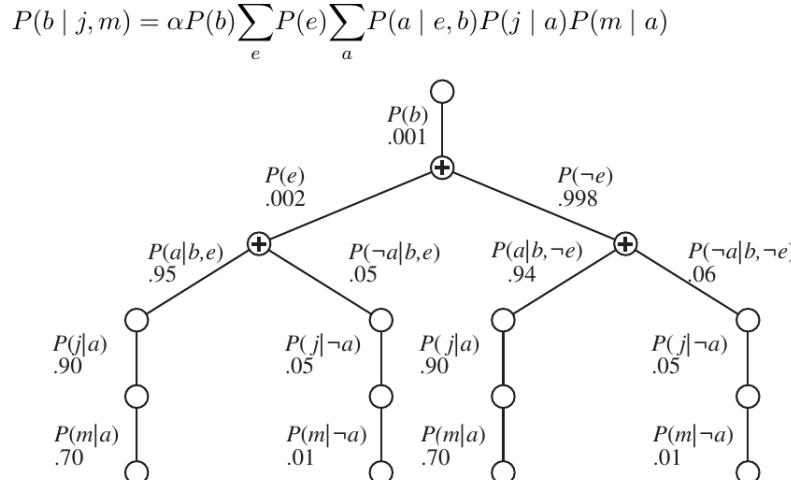
$$= \alpha \sum_e \sum_a \mathbf{P}(B, j, m, e, a)$$

If we consider the independence of variables, we obtain for $B = true$

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

Reorganization of the terms yields:

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



$$\mathbf{P}(B \mid j, m) = \alpha(0.0006, 0.0015) = (0.284, 0.716)$$

```

function ENUMERATION-ASK( $X$ ,  $\mathbf{e}$ ,  $bn$ ) returns a distribution over  $X$ 
  inputs:  $X$ , the query variable
     $\mathbf{e}$ , observed values for variables  $\mathbf{E}$ 
     $bn$ , a Bayes net with variables  $\{X\} \cup \mathbf{E} \cup \mathbf{Y}$  /*  $\mathbf{Y}$  = hidden variables */
   $\mathbf{Q}(X) \leftarrow$  a distribution over  $X$ , initially empty
  for each value  $x_i$  of  $X$  do
     $\mathbf{Q}(x_i) \leftarrow$  ENUMERATE-ALL( $bn.VARS$ ,  $\mathbf{e}_{x_i}$ )
      where  $\mathbf{e}_{x_i}$  is  $\mathbf{e}$  extended with  $X = x_i$ 
  return NORMALIZE( $\mathbf{Q}(X)$ )


---


function ENUMERATE-ALL( $vars$ ,  $\mathbf{e}$ ) returns a real number
  if EMPTY?( $vars$ ) then return 1.0
   $Y \leftarrow$  FIRST( $vars$ )
  if  $Y$  has value  $y$  in  $\mathbf{e}$ 
    then return  $P(y | parents(Y)) \times$  ENUMERATE-ALL( $REST(vars)$ ,  $\mathbf{e}$ )
    else return  $\sum_y P(y | parents(Y)) \times$  ENUMERATE-ALL( $REST(vars)$ ,  $\mathbf{e}_y$ )
      where  $\mathbf{e}_y$  is  $\mathbf{e}$  extended with  $Y = y$ 

```

Depth-first algorithm, linear space complexity, $O(2^n)$ time complexity.

- Variable elimination:

The algorithm can be improved by **eliminating repeating** or **unnecessary** calculations. The key idea is to evaluate expressions from right to left (bottom-up) and to save results for later use.

Let us consider the query $P(JohnCalls | Burglary = true)$.

The nested sum is

$$P(j, b) = \alpha P(b) \sum_e P(e) \sum_a P(a | b, e) P(j, a) \sum_m P(m | a)$$

general observation: variables, that are not query or evidence variables and not ancestor nodes of query or evidence variables can be removed. Variable elimination repeatedly removes these variables and this way speeds up computation.

within example: Alarm and Earthquake are ancestor nodes of query variable JohnCalls and cannot be removed. MaryCalls is neither a query nor an evidence variable and no ancestor node. Therefore it can be removed.

- Complexity of exact inference:

If the network is singly connected or a polytree, the time and space complexity of is linear.

For multiply connected networks inference in Bayesian Networks is NP-hard.

11.4 Summary

Uncertainty is **unavoidable in complex, dynamic worlds** in which agents are ignorant.

Probabilities express the agent's inability to reach a definite decision.

Conditional and unconditional probabilities can be formulated over propositions.

If an agent **disrespects the theoretical probability axioms**, it is likely to demonstrate **irrational behaviour**.

Bayes' rule allows us to calculate **known probabilities from unknown probabilities**.

Multiple evidence (assuming independence) can be effectively incorporated using **recursive Bayesian updating**.

Bayesian Networks allow a compact representation of **joint probability distribution**.

Bayesian Networks provide a concise way to represent **conditional independence in a domain**.

Inference in Bayesian networks means **computing the probability distribution of a set of query variables, given a set of evidence variables**.

Exact inference algorithms such as variable elimination are efficient for poly-trees.

In complexity of belief network inference depends on the network structure.

In general, Bayesian network inference is NP-hard.

12 Acting under Uncertainty Maximizing Expected Utility

12.1 Introduction to Utility theory

The utility function rates states and thus formalizes the desirability of a state by the agent. A non-deterministic action A can lead to the outcome states $Result_i(A)$. How high is the probability that the outcome state $Result_i(A)$ is reached, if A is executed in the current state with evidence E?

$$\beta P(Result_i(A)|Do(A), E) \quad v \rightarrow \text{Utility}$$

- Expected utility:

$$EU(A|E) = \sum_i P(Result_i(A)|Do(A), E) U(Result_i(A))$$

The principle of maximum expected utility (MEU) says that a rational agent should choose an action that maximizes $EU(A|E)$.

- Problem: Utility function requires search or planning, because an agent needs to know the possible future states in order to assess the worth of the current state ("effect of the state on the future").

12.2 Choosing Individual Actions

Preferences:

$L_1 > L_2$ Agent prefers 1 over 2

$L_1 \sim L_2$ Agent is indifferent between 1 or 2

$L_1 \geq L_2$ Agent prefers 1 or is indifferent.

Orderability

$$(A > B) \vee (B > A) \vee (A \sim B)$$

Transitivity

$$(A > B) \vee (B > C) \Rightarrow (A > C)$$

Continuity

$$A > B > C \Rightarrow \exists p [p, A; 1 - p, C] \sim B$$

Substitutability

$$A \sim B \rightarrow [p, A; 1 - p, C] \sim [p, B; 1 - p, C]$$

Monotonicity

$$A > B \Rightarrow (p > q \Leftrightarrow [p, A; 1 - p, B] > [q, A; 1 - q, B])$$

Decomposability

$$[p, A; 1 - p, [q, B; 1 - q, C]] \sim [p, A; (1 - p)q, B; (1 - p)(1 - q), C]$$

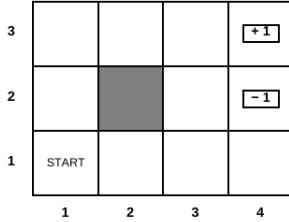
Expected Utility of a Lottery:

$$U[p_1, S_1, \dots, p_n, S_n] = \sum_i p_i U(S_i)$$

Since the outcome of a nondeterministic action is a lottery, an agent can act rationally only by following the **Maximum Expected Utility (MEU) principle**.

Given a utility scale between u_{min} and u_{max} we can assess the utility of any particular outcome S by asking the agent to choose between S and a standard lottery $[p, u_{max}; 1 - p, u_{min}]$. We adjust p until they are equally preferred.

12.3 Sequential Decision Problems



Beginning in the start state the agent must choose an action at each time step.

The interaction with the environment terminates if the agent reaches one of the goal states (4,3) (reward of +1) or (4,2) (reward -1). Each other location has a reward of ~~-0.04~~.

In each location the available actions are Up, Down, Left, Right.

- **Deterministic version:**

All actions always lead to the next square in the selected direction, except that moving into a wall results in no change in position.

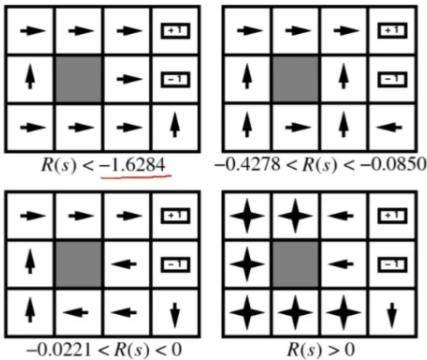
- **Stochastic version:**

Each action achieves the intended effect with probability 0.8, but the rest of the time, the agent moves at right angles to the intended direction.

12.4 Markov Decision Processes

Stochastic environment defined by: **States, Actions, Transition model $P(s'|s, a)$ (probability), Reward function $R(s)$, Policy:** mapping π that specifies for each state which action to take. We, obviously, want the optimal policy π^* that maximizes the future expected reward.

Given the optimal policy, it executes the action $\pi^*(s)$. In this way we obtain a simple reflex agent.



Performance is measured by the sum of rewards. To determine the **optimal policy**, first calculate the utility of each state, it also depends on the horizon, if it is finite or infinite. For **finite horizons** the optimal policy is called **nonstationary**, instead for **infinite** it is **stationary**.

- **- Stationary:**

Two ways to reward:

Additive rewards: $U_h([s_0, s_1, \dots, s_n]) = R(s_0) + R(s_1) \dots$

Discounted rewards: $U_h([s_0, s_1, \dots, s_n]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) \dots$

$\gamma \in [0, 1]$ is the discount factor. With discount, the utility is always finite. The discount factor expresses that future rewards have less than current rewards.

12.5 Value Iteration

$U^\pi(s)$ is the utility of a state under policy π . s_t be the state of the agent after executing π for t steps. The utility of s under π is:

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s\right] \quad (4)$$

$R(s)$ is the short-term reward for being in s and $U(s)$ is the long-term total reward from s onwards.

The utilities of the states in our 4×3 world with $\gamma = 1$ and $R(s) = -0.04$ for non-terminal states:

	3	0.812	0.868	0.918	+1
	2	0.762		0.660	-1
	1	0.705	0.655	0.611	0.388
	1	2	3	4	

The agent simply chooses the action that maximizes the expected utility of the subsequent state:

$$\pi(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s, a) U(s') \quad (5)$$

The utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, - **Bellman-Equation**:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s, a) U(s') \quad (6)$$

- **Example:**

In our 4×3 world the equation for the state $(1,1)$ is

$$\begin{aligned} U(1,1) &= -0.04 + \gamma \max \{ 0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), && (\text{Up}) \\ &\quad 0.9U(1,1) + 0.1U(1,2), && (\text{Left}) \\ &\quad 0.9U(1,1) + 0.1U(2,1), && (\text{Down}) \\ &\quad 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1) \} && (\text{Right}) \\ &= -0.04 + \gamma \max \{ 0.8 \cdot 0.762 + 0.1 \cdot 0.655 + 0.1 \cdot 0.705, && (\text{Up}) \\ &\quad 0.9 \cdot 0.705 + 0.1 \cdot 0.762, && (\text{Left}) \\ &\quad 0.9 \cdot 0.705 + 0.1 \cdot 0.655, && (\text{Down}) \\ &\quad 0.8 \cdot 0.655 + 0.1 \cdot 0.762 + 0.1 \cdot 0.705 \} && (\text{Right}) \\ &= -0.04 + 1.0 (0.6096 + 0.0655 + 0.0705), && (\text{Up}) = -0.04 + 0.7456 = 0.7056 \end{aligned}$$

Up is the optimal action in $(1,1)$.

	3	0.812	0.868	0.918	+1
	2	0.762		0.660	-1
	1	0.705	0.655	0.611	0.388
	1	2	3	4	

- **Value Iteration algorithm**

An algorithm to calculate an optimal strategy. **Basic Idea:** Calculate the utility of each state. A sequence of actions generates a branch in the tree of possible states (histories). A utility function on histories U_h is separable iff there exists a function f such that:

$$U_h([s_0, s_1, \dots, s_n]) = f(s_0, U_h([s_1, \dots, s_n])) \quad (7)$$

The simplest form is an additive reward function R:

$$U_h([s_0, s_1, \dots, s_n]) = R(s_0) + U_h([s_1, \dots, s_n]) \quad (8)$$

Typical problems contain cycles, which means the length of the histories is potentially infinite. Solution: Use

$$U_{t+1}(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s, a) U_t(s') \quad (9)$$

The Bellman equation is the basis of value iteration, thanks to the max-operator the n equations for the n states are nonlinear. We can apply an iterative approach in which we replace the equality by an assignment:

$$U(s') \leftarrow R(s) + \gamma \max_a \sum_{s'} P(s'|s, a) U(s') \quad (10)$$

- **Algorithm**

```

function VALUE-ITERATION(mdp,  $\epsilon$ ) returns a utility function
  inputs: mdp, an MDP with states  $S$ , actions  $A(s)$ , transition model  $P(s' | s, a)$ ,
            rewards  $R(s)$ , discount  $\gamma$ 
             $\epsilon$ , the maximum error allowed in the utility of any state
  local variables:  $U$ ,  $U'$ , vectors of utilities for states in  $S$ , initially zero
             $\delta$ , the maximum change in the utility of any state in an iteration

  repeat
     $U \leftarrow U'$ ;  $\delta \leftarrow 0$ 
    for each state  $s$  in  $S$  do
       $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s']$ 
      if  $|U'[s] - U[s]| > \delta$  then  $\delta \leftarrow |U'[s] - U[s]|$ 
    until  $\delta < \epsilon(1 - \gamma)/\gamma$ 
  return  $U$ 

```

- **Convergence**

```

  if  $\|U_{t+1} - U_t\| < \epsilon(1 - \gamma)/\gamma$  then  $\|U_{t+1} - U\| < \epsilon$ 
  if  $\|U_t - U\| < \epsilon$  then  $\|U^{\pi_t} - U\| < 2\epsilon\gamma/(1 - \gamma)$ 

```

12.5.1 Policy Iteration

Policy iteration alternates the following two steps beginning with an initial policy π_0 :

- **Policy evaluation:** given π_t , calculate $U_t = U^{\pi_t}$ the utility if π_t was executed.
- **Policy improvement:** calculate a new maximum expected utility policy π_{t+1} according to:

$$\pi_{t+1}(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s, a) U_t(s') \quad (11)$$

```

function POLICY-ITERATION(mdp) returns a policy
  inputs: mdp, an MDP with states  $S$ , actions  $A(s)$ , transition model  $P(s' | s, a)$ 
  local variables:  $U$ , a vector of utilities for states in  $S$ , initially zero
             $\pi$ , a policy vector indexed by state, initially random

  repeat
     $U \leftarrow \text{POLICY-EVALUATION}(\pi, U, mdp)$ 
     $\text{unchanged?} \leftarrow \text{true}$ 
    for each state  $s$  in  $S$  do
      if  $\max_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s'] > \sum_{s'} P(s' | s, \pi[s]) U[s']$  then do
         $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' | s, a) U[s']$ 
         $\text{unchanged?} \leftarrow \text{false}$ 
    until  $\text{unchanged?}$ 
  return  $\pi$ 

```

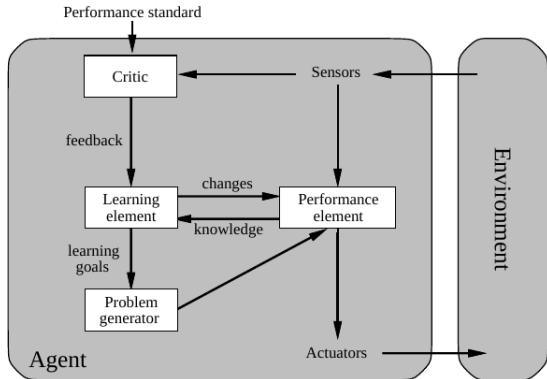
12.6 Summary

Rational agents can be developed on the basis of a **probability theory and a utility theory**. Agents that make decisions according to the axioms of utility theory possess a **utility function**. Sequential problems in **uncertain environments (MDPs)** can be solved by calculating a policy. **Value iteration** is a process for calculating optimal policies.

13 Machine Learning

An agent learns when it improves its performance w.r.t. a specific task with experience.

13.1 The learning agent



- **Performance element:** Processes percepts and chooses actions. (agent model before)
- **Learning element:** Carries out improvements. It requires self knowledge and feedback on how the agent is doing in the environment.
- **Critic:** Evaluation of the agent's behavior based on a given external behavioral measure. (feedback)
- **Problem generator:** Suggests explorative actions that lead the agent to new experiences.

13.2 Types of learning

- **Supervised learning:** Involves learning a function from examples of its inputs and outputs. An example is a pair $(x, f(x))$. The complete set of examples is called the **training set**. Pure inductive inference: for a collection of examples for f , return a function h (hypothesis) that approximates f . A good hypothesis should predict unseen examples correctly. A **hypothesis is consistent** with the data set if it agrees with all the data. **Ockham's razor:** prefer the simplest hypothesis consistent with the data.
- **Unsupervised learning:** The agent has to learn patterns in the input when no specific output values are given.
- **Reinforcement learning:** The most general form of learning in which the agent is not told what to do by a teacher. Rather it must learn from a reinforcement or reward. It typically involves learning how the environment works.

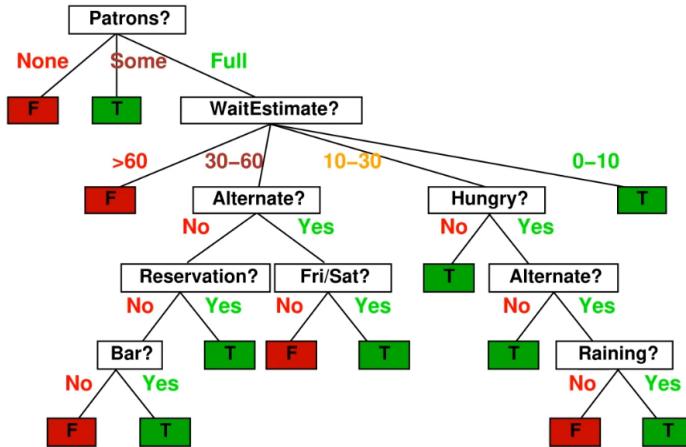
13.3 Decision trees

- **Input:** set of attributes.
- **Output:** a decision.
- **Continuous (regression) or discrete (classification),** binary or not.

13.3.1 Boolean Decision Trees

- **Input:** set of vectors of input attributes.
- **Output:** Yes/No decision based on a goal predicate.
- **Properties:** an internal node represents a test of a property, branches are labeled with the possible values, each leaf specifies the returning value.

- Example:



Each decision tree hypothesis can be seen as an assertion of the form:

$$\forall s \quad \text{WillWait}(s) \leftrightarrow (P_1(s) \vee P_2(s) \vee \dots \vee P_n(s)) \quad (12)$$

$P_i(s)$ are the tests along the path. **Limitation:** trees always involve just one variable.

- Compact representations:

We can construct a decision tree by translating every row of a truth table to a path in the tree. This can lead to exponential trees (**parity**: $p(x) = 1$ if even number of inputs are 1, 0 otherwise, **majority**: $m(x) = 1$ if half of the inputs are 1, 0 otherwise).

- Training Set example:

Classification of an example = Value of the goal predicate

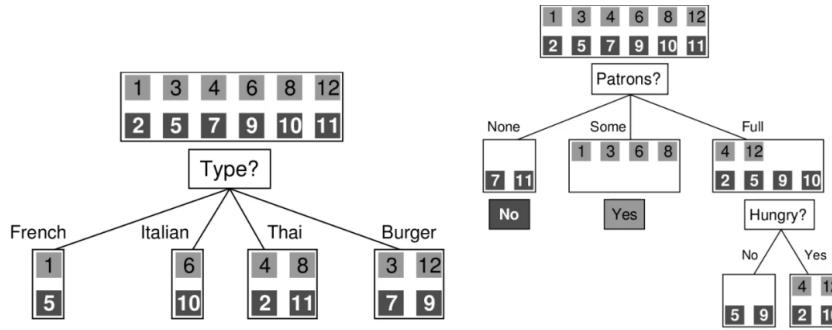
$T \rightarrow$ positive example

$F \rightarrow$ negative example

Example	Attributes										Target
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
X_1	T	F	F	T	Some	$\$\$\$$	F	T	French	$0-10$	T
X_2	T	F	F	T	Full	$\$$	F	F	Thai	$30-60$	F
X_3	F	T	F	F	Some	$\$$	F	F	Burger	$0-10$	T
X_4	T	F	T	T	Full	$\$$	F	F	Thai	$10-30$	T
X_5	T	F	T	F	Full	$\$\$\$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$\$\$$	T	T	Italian	$0-10$	T
X_7	F	T	F	F	None	$\$$	T	F	Burger	$0-10$	F
X_8	F	F	F	T	Some	$\$\$$	T	T	Thai	$0-10$	T
X_9	F	T	T	F	Full	$\$$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$\$\$\$$	F	T	Italian	$10-30$	F
X_{11}	F	F	F	F	None	$\$$	F	F	Thai	$0-10$	F
X_{12}	T	T	T	T	Full	$\$$	F	F	Burger	$30-60$	T

- Creation possibilities:

- **Naive:** simply construct with one path to a leaf for each example. Test all attributes. Even if it will classify the examples, It will not say anything about other classes. It just memorizes the observation and doesn't generalize.
- **Smallest solution:** Applying Ockham's razor we find the smallest decision tree. unfortunately, the smallest definition is intractable.
- **Divide and Conquer:** Choose an attribute, Split the training set such that each corresponds to a particular value of that attribute, recursive apply this process to the smaller sets.



Type is a poor attribute since it leaves us with state full of true and false.

Patrons is a better choice.

In each recursive step there are 4 cases:

- Positive and negative: choose a new attribute.
- Only positive: finished.
- No examples: Answer the majority
- No attributes left: Too noisy.

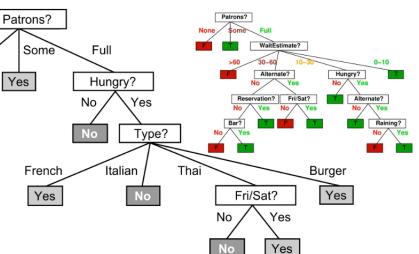
function DTL(examples, attributes, default) returns a decision tree

```

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 MODE(examples)
else
    best ← CHOOSE-ATTRIBUTE(attributes, examples)
    tree ← a new decision tree with root test best
    for each value  $v_i$  of best do
        examples $_i$  ← {elements of examples with best =  $v_i$ }
        subtree ← DTL(examples $_i$ , attributes – best, MODE(examples))
        add a branch to tree with label  $v_i$  and subtree subtree
    return tree

```

Original tree:



- Properties of the Resulting tree:

The learning algorithm outputs a tree that is consistent with all examples it has seen.

The tree does not necessarily agree with the correct function. Some tests (Raining, Reservation) are not included (are useless) since the algorithm can classify the examples without them.

- Choosing Attribute Tests

One goal of decision tree learning is to select attributes that minimize the depth of the final tree. The **perfect attribute** divides the examples into sets that are all positive or all negative. Patrons is not perfect but fairly good. Type is useless since the resulting proportion of positive and negative examples in the resulting sets are the same as in the original set.

- Information provided by an Attribute

One suitable measure is the expected amount of information provided by the attribute.

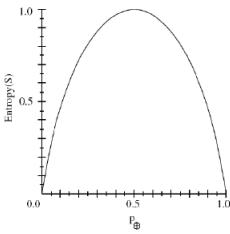
In general, if the possible answers v_i have probabilities $P(v_i)$, the information content is given as the entropy:

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

$$I\left(\frac{1}{2}, \frac{1}{2}\right) = 1 \text{ Max}$$

$$I(1, 0) = -1 \log_2(1) - 0 \log_2(0) = 0$$

$\cancel{\lim_{x \rightarrow 0} x \log_2(x) = 0}$



$$I(0, 1) = 0$$

- Attribute Selection

Suppose the training set E consists of p positive and n negative examples:

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = \frac{p}{p+n} \log_2\left(\frac{p+n}{p}\right) + \frac{n}{p+n} \log_2\left(\frac{p+n}{n}\right)$$

The value of an attribute A depends on the additional information that we still need to collect after we selected it. A random example has value i with probability: $\frac{p_i+n_i}{p+n}$.

The average information content after choosing A is: $R(A) = \sum_{i=1}^v \frac{p_i+n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$

The information gain from choosing A is: $Gain(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - R(A)$.

Heuristic in Choose-Attribute is to select the *attribute with the largest gain*.

Examples:

$$\begin{cases} Gain(Patrons) = 1 - [\frac{2}{12} I(0, 1) + \frac{4}{12} I(1, 0) + \frac{6}{12} I(\frac{2}{6}, \frac{4}{6})] \approx 0.541 & \checkmark \\ Gain(Type) = 1 - [\frac{2}{12} I(\frac{1}{2}, \frac{1}{2}) + \frac{2}{12} I(\frac{1}{2}, \frac{1}{2}) + \frac{4}{12} I(\frac{2}{4}, \frac{2}{4}) + \frac{4}{12} I(\frac{2}{4}, \frac{2}{4})] = 0 & \times \end{cases}$$

The training and test sets must be kept separate.

- Methodology for assessing the power of prediction:

- Collect a large number of examples.
- Divide it into two disjoint sets: the training set and the test set.
- Use the training set to generate h.
- Measure the percentage of examples of the test set that are correctly classified by h.
- Repeat the process for randomly-selected training sets of different sizes.

13.4 Summary

Decision tree is one possibility for representing (Boolean) functions.

Decision trees can be **exponential** in the number of attributes.

It is often too difficult to find the **minimal DT**.

One method for generating DTs that are as flat as possible is based on **ranking the attributes**.

The ranks are computed based on the **information gain**.

14 Deep Learning

14.1 Why?

Excellent empirical results in a lot of Learning fields. Learning as an Alternative to Traditional Programming, we don't understand how the human brain solves certain problems, we can nevertheless learn these tasks from data/experience. If the task changes, we simply re-train. We can construct computer systems that are too complex for us to understand anymore ourselves. **Learning from data / experience may be more human-like.**

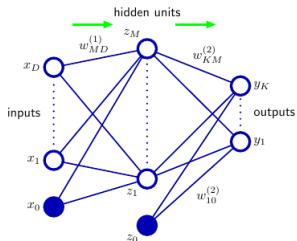
14.2 Representation Learning and Deep Learning

- **Representation Learning:** a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification.
- **Deep Learning:** representation learning methods with multiple levels of representation, composing nonlinear modules.
- **ML Pipeline:** high-level attributes or features of the data E.g., the binary attributes we used for decisions trees (feature engineering).
- **RL Pipeline:** Jointly learn features and classifier, directly from raw data This is also referred to as end-to-end learning. Learning a hierarchy of representations that build on each other, from simple to complex.

ANNs are inspired by biology: Dendrites input information to the cell, Neuron fires (has action potential) if a certain threshold for the voltage is exceeded, Output of information by axon, The axon is connected to dendrites of other cells via synapses.

The advance doesn't depend only on the computer power, but also on the methodological improvements.

14.3 Multilayer Perceptrons



Linear function of previous layer's outputs: $z_{k-1} : a_k = W_k \cdot z_{k-1} + b_k$

Nonlinear transformation: $z_k = h_k(a_k)$

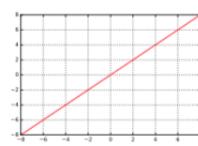
Logistic sigmoid activation function:

$$h_{\text{logistic}}(a) = \frac{1}{1 + \exp(-a)}$$

Non linear.

Linear activation function:

$$h_{\text{linear}}(a) = a$$



Logistic hyperbolic tangent activation function:

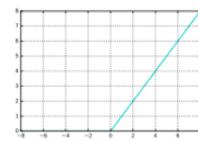
$$\begin{aligned} h_{\tanh}(a) &= \tanh(a) \\ &= \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} \end{aligned}$$



Rectified Linear (ReLU) activation function:

$$h_{\text{relu}}(a) = \max(0, a)$$

*Non linear.
Not smooth.*



- **For regression:** single output with linear activation, loss function is squared error:

$$L(w) = \frac{1}{2} \sum_{n=1}^N y_n h(x_n, w) - y_n^2$$

x_n is the raw input, y_n is the true value of data point.

- **For classification:** single output with logistic activation and loss function is **negative log likelihood (cross-entropy)**:

$$L(w) = - \sum_{n=1}^N y_n \ln y^c + (1 - y_n) \ln(1 - y^c)$$

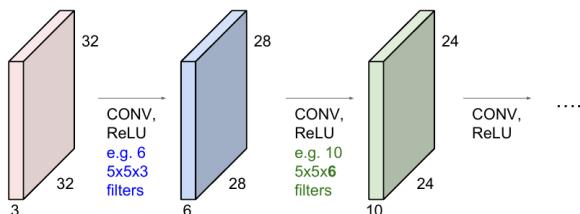
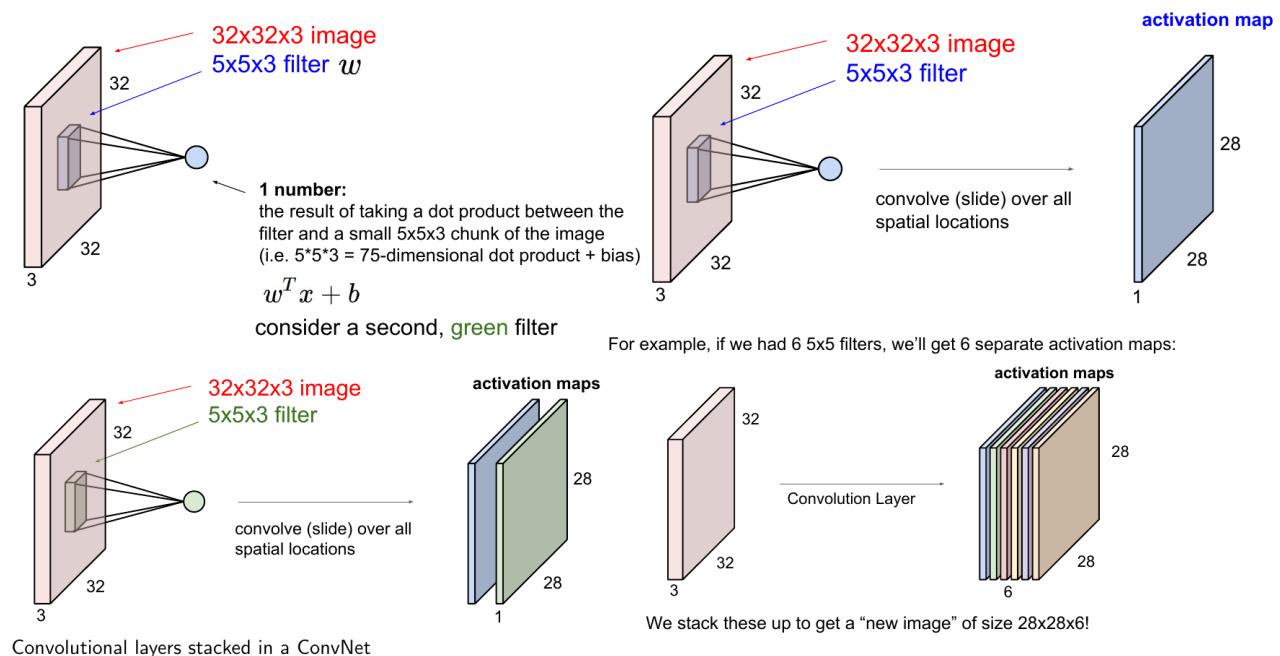
To **optimize a loss function** usually the weights are adapted to minimize the error, one technique can be using gradient-based optimization: gradients are computed and use of backpropagation using small batches of the data → **Stochastic gradient descent (SGD)**:

We can use small batches of the data, i.e., $L(w; D_{batch})$. This yields approximate gradients quickly.

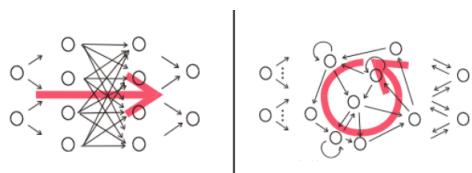
14.4 Convolutions Neural Networks

Start from a 3D image dot product for a smaller 3D filter: $w^T x + b$

Based on the filter, the convolution layer is divided in activation maps.

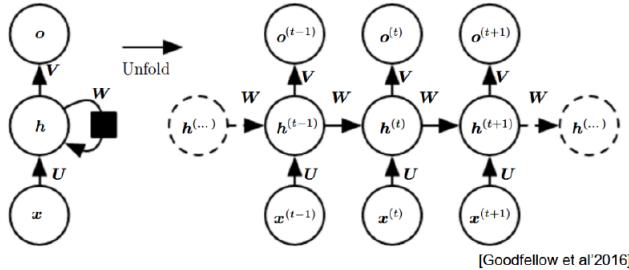


14.5 Recurrent Neural Networks

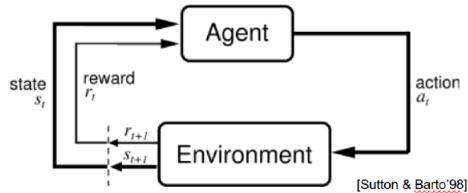


The second one is a **Recurrent Neural Networks**: cycles let information persist in the network for some time (state), and provide a time-context. Implement dynamical systems. Turing complete.

With fully connected hidden layer:



14.6 Reinforcement Learning



Finding **optimal policies for MDPs** composed of: states, actions, transition model and rewards. **Policy**: complete mapping π that specifies for every state s which action to take.

- **Policy-based deep RL**
 - Represent policy $\pi : S \rightarrow A$ as a deep neural network with weights w
 - Evaluate w by “rolling out” the policy defined by w
 - Optimize weights to obtain higher rewards (using approx. gradients)
 - Examples: AlphaGo & modern Atari agents
 - **Value-based deep RL**
 - Basically value iteration, but using a deep neural network (= function approximator) to generalize across many states and actions
 - Approximate optimal state-value function $U(s)$ or state-action value function $Q(s, a)$
 - **Model-based deep RL**
 - If transition model T is not known
 - Approximate T with a deep neural network (learned from data)
 - Plan using this approximate transition model
- Use deep neural networks to represent policy / value function / model

14.7 Limitations

All we've seen till now is bottom-up, there is no logic and no reasoning. It's very far from human performance.

- ▶ Even for very strong networks we can find adversarial examples
 - By following the gradient of the cost function **w.r.t the input**

14.8 Summary - Why Deep Learning?

Excellent empirical results in many domains: very scalable to big data, but beware: not a silver bullet.

Analogy to the ways humans process information: mostly tangential.

Allows end-to-end learning: no more need for many complicated subsystems, e.g., dramatically simplified Google's translation pipeline.

Very versatile/flexible: easy to combine building blocks, allows supervised, unsupervised, and reinforcement learning.

15 Natural Language Processing

15.1 Motivations and Tasks

NLP creates interfaces between human language and computers.

Goal: automatic processing of large amounts of human language data.

Traditional rule-based approaches and (to a smaller degree) probabilistic NLP models faced limitations as human errors, rules and labels.

15.2 Learning Representations

- **Word Embedding**

It is a mapping function from words to R^n . It realizes a look-up table such that related words are near and unrelated words have great distance.

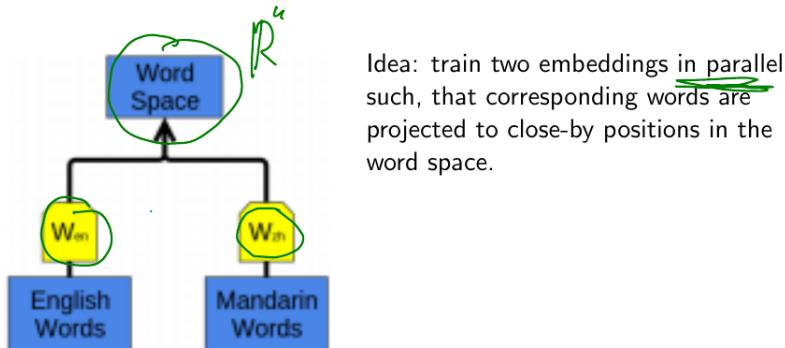
A **word embedding function W** can be trained using different tasks. **EX.:** predict, if a 5-gram (sequence of five words) is valid or not.

$$\begin{aligned} & R(W("cat"), W("sat"), W("on"), W("the"), W("mat")) = 1 \\ & R(W("cat"), W("sat"), W("song"), W("the"), W("mat")) = 0 \end{aligned}$$

Training data contains valid and slightly modified, invalid 5-grams. Train the combination of embedding function W and classification module R.

Embedding allows to work not only with synonyms, **but also with other words of the same category**. In the embedding space, systematic shifts can be observed for analogies.

- **Bilingual Word Embedding**

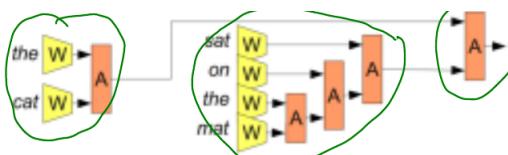


Based on training two embeddings in parallel such, that corresponding words are projected to close-by positions in the word space.

15.3 Sequence-to-Sequence Deep Learning

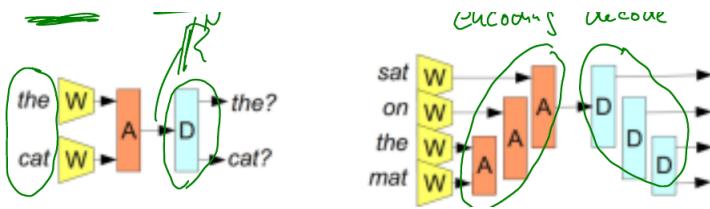
15.3.1 Association Modules

Limitation on number of input words can be **overcome** by adding association modules, which can combine two word and phrase representations and merge them.



15.3.2 From Representations to the Translation of Texts

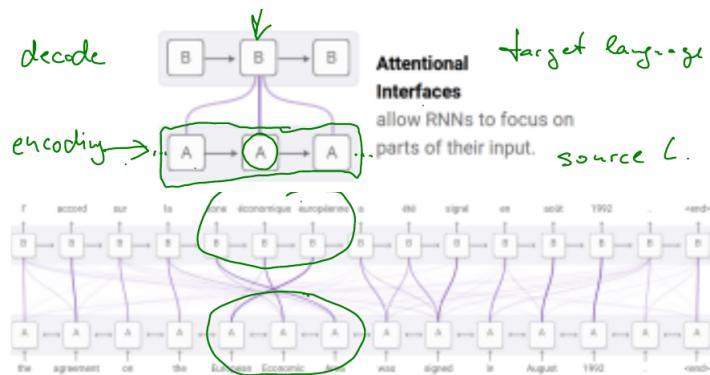
For translations, we also need disassociation modules (encoder - decoder principle)



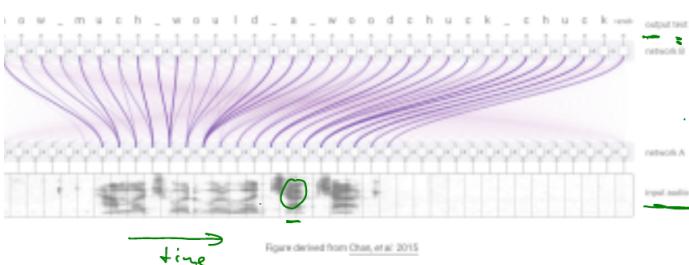
Take context into account by learning the importance of neighboring words → attention mechanism. The same principle is used for voice recognition but here we take into account Frequency and time.

15.4 Sequence-to-Sequence Neural Machine Translation

Shift through the input word sequence, learn to encode and to decode using recurrent neural networks (RNN), learn to align input and output word sequences, take context into account by learning the importance of neighboring words → attention mechanism.



Similar principle, but voice/speech input



15.5 Summary

Natural language processing spans a wide range of problems and applications.

NLP is a rapidly growing field due to availability of **huge data sets**.

Field is moving more and more to **neural networks**, which provide NLP building blocks like end-to-end learning, representation learning, sequence-to-sequence, etc.

16 Ethics

AI is not (only) basic research (anymore) if your research/system can result in something unethical (harm people). Practical ethics in form of guidelines/principles for AI systems/research.

16.1 Ethical principles

There are many different lists of principles, but it seems that they all can be synthesized into five key principles:

autonomy (people should be able to make their own decisions, e.g. human-in-the-loop, privacy protection).

beneficence (society at large should benefit).

non-maleficence (harmful consequences should be avoided, e.g. systems should be robust).

justice (diversity, non-discrimination and fairness).

explicability (transparency and explainability).

Principles can create problem, since they are very high level and can be interpreted in different ways, they can conflict with each other in concrete case, one concrete principle: **No military applications**.

One particular horrifying application are fully autonomous weapons, aka **killer robots**. We are on the verge of building them, and the big players (US, Russia, China) definitely have projects on it. There are campaigns for banning these weapons (similar to banning chemical weapons).

16.2 Algorithmic Fairness

The topic of enforcing fairness has become important, in particular in machine learning. If these system act unfair, groups and individuals may suffer.

16.2.1 Possible reasons for unfairness

Skewed sample: If some initial bias happens, such bias may compound over time.

Tainted examples: E.g. word embeddings may lead to gender stereotypes.

Limited features: Some features may be less informative for a minority group.

Sample size disparity: Training data from minority group is sparse.

16.3 Machine Ethics

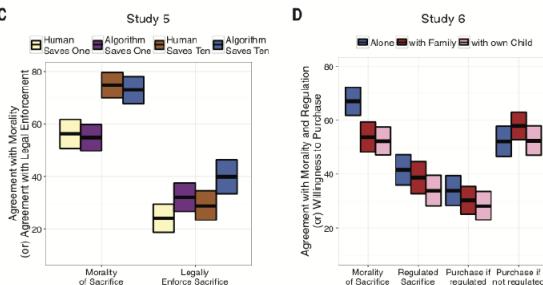
One can try to find properties such that machines could act morally. Machines need to have [Misselhorn] at least: **beliefs about the world, pro-attitudes (intentions), moral knowledge, the possibility to compute what consequences ones own action can have.**

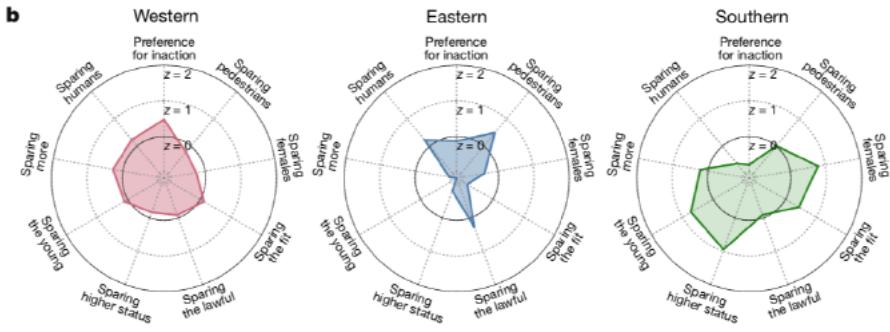
In this cases they can be considered as moral agents.

16.4 Self-Driving Cars

Self-driving cars will come into situations where they have to choose between bad alternatives.

Do you think it is moral to sacrifice yourself? Would you buy such a car?





16.5 Morally Competent Planning Systems

Imagine an household robot that to make children be silent, kill them (**unmoral**).

Can we build morally competent planers?

1 How to judge action plans?

2 How to evaluate goal choices?

3 How to generate morally permissible action plans?

- Ethical principles:

Deontology: Actions have an inherent ethical value (Kantianism).

Utilitarianism: Actions are only judged by their consequences.

Do-no-harm: Don't do anything that leads to (some) negative consequences.

Asimovian: Avoid harm if possible.

Do-no-instrumental-harm: Don't do anything that leads to (some) negative consequences, except it is a non-indented side-effect.

Principle of double effect (DDE): An action is permissible if

- 1. The act itself must be morally good or neutral.
- 2. A positive consequence must be intended.
- 3. No negative consequence may be intended.
- 4. No negative consequence may be a means to the goal.
- 5. There must be proportionally grave reasons to prefer.

A plan is **deontological** permissible if all of its actions are not morally impermissible.

Theorem: The deontological plan validation problem can be decided in time linear in plan size.

The plan is only permissible if the reached state has a maximum utility value over all reachable states. In so far, the validation problem is very similar to over-subscription planning.

The utilitarian plan validation problem is PSPACE-complete.

Proof Sketch: Membership: Impermissibility could be shown by guessing a higher-valued state and then non-deterministically verifying that there exists a plan to it.

Hardness: Reduce (propositional) plan non-existence to permissibility.

- **Do-no-harm plan validation:**

We could ask whether no harmful fact is true in the end. Only then we do no harm.

Harm could already be true in the initial state.

Harmful fact could be removed and added again during execution.

You can avoid harm by doing more or by doing less. We will only consider the latter option.

Could harm be avoided by doing nothing?

Can harm be avoided by deleting a single action?

Could harmful consequences be avoided by leaving out a subset of actions?

Theorem: The do-no-harm plan validation problem is co-NP-complete.

Proof sketch: *Membership:* Impermissibility can be checked by a non-deterministic algorithm using only polynomial time. *Hardness:* 3SAT can be reduced to impermissibility.

Means to an end: An effect in a plan is a means to an intended end effect, if this end effect were not true in the final state if some subset of the particular means effect is deleted in the plan.

Do-no-instrumental-harm plan validation: The means to an end definition implies that we have the same combinatorial problem as for the simpler do-no-harm principle.

Theorem: The do-no-instrumental-harm plan validation problem is co-NP-complete.

Double-effect plan validation: All criteria except for the no negative consequence may be a means to the goal condition can be checked easily.

Theorem: The double-effect plan validation problem is co-NP-complete.

16.6 Summary

Thinking about **ethics in AI** is unavoidable these days!

There exist a number of ethical principles/guidelines from different institutions, which are very similar, though.

In particular, **fairness, privacy, and explainability** have sparked new research directions in AI.

Machine ethics is the field of covering ethics from a computational point of view.

Self-driving cars have to cope with dilemma situations!

There is no theory about ethics in **action planning**.

Generalization of **action-based to plan-based ethical** judgments is possible.

Surprising complexity results, based on the fact that the same effect can be made true arbitrarily often and can interact with each other.