Foundations of Artificial Intelligence

Riccardo Salvalaggio

19th of April, 2021

Contents

1 Introduction

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

2 Rational Agents

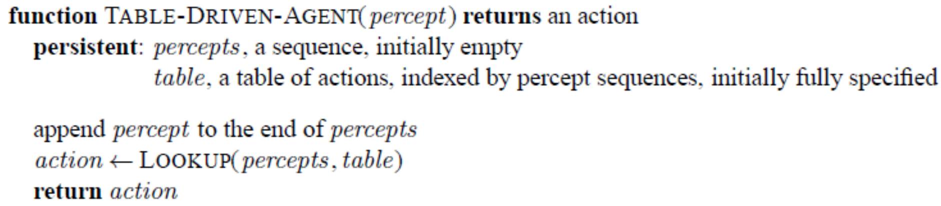
It is a model that perceive the environment through sensors and act through actuators. In order to evaluate their performance use performance measure (e.g. vacuum -¿ level of cleanliness etc.) even if optimal behaviour is often unattainable because it is quite impossible to reach the goal in every aspect.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

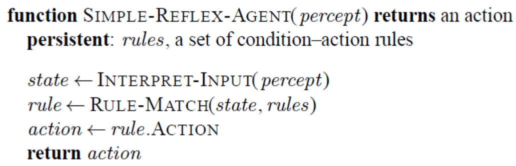
2.2.1 Table-Driven (the simplest)



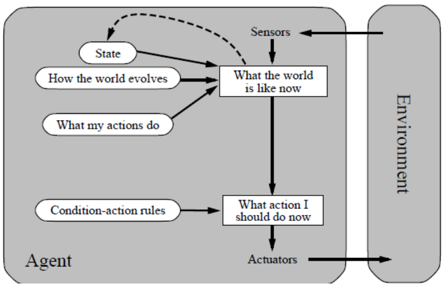
```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  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
```



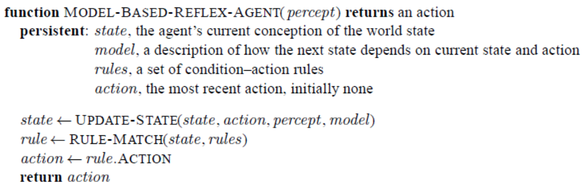
Interpretation of the input, matching to a rule to extract an action.

2.2.3 Model-Based Reflex

Introduction of a utlity 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.



```
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
```

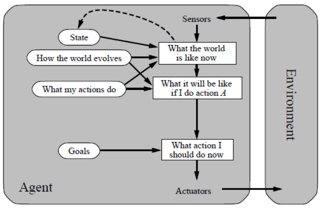

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Figure 1: Goal-based

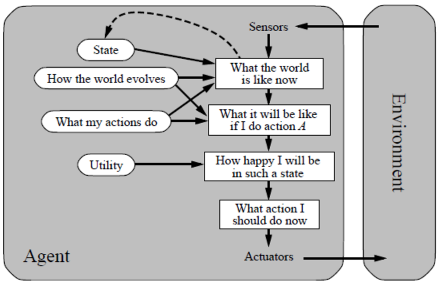
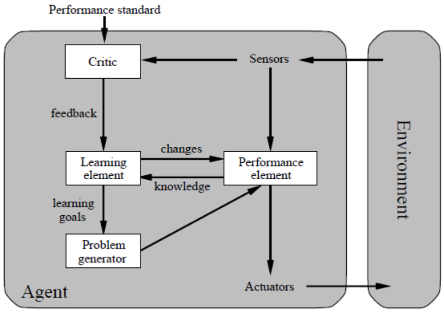

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Figure 2: Utility-based

Performance element: select external actions.
Critic: determines performance of the agent.
Problem generator: suggests actions that lead to informative experiences.

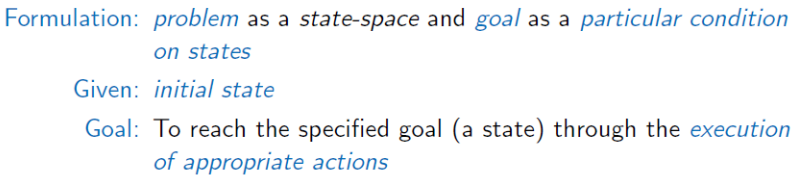


2.3 Types of environments

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

3 Solving Poblems by Searching

3.1 Problem-solving agents

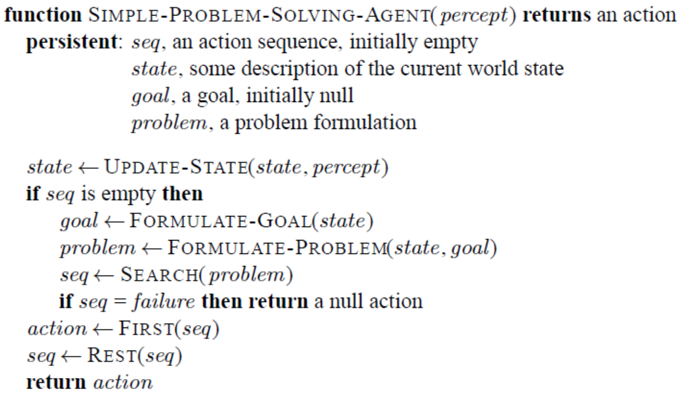


Formulation: *problem* as a *state-space* and *goal* as a *particular condition on states*

Given: *initial state*

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

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



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

Figure 3: Simple Problem-solving Agent

3.2 Problem Formulation

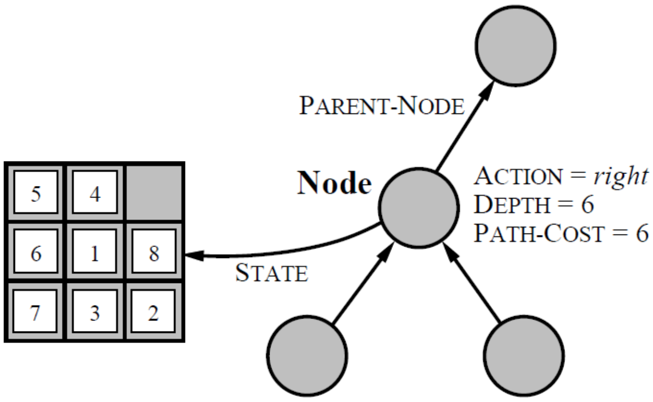
Goal formulation, definition of: State space, actions, problem type, search and execution costs.

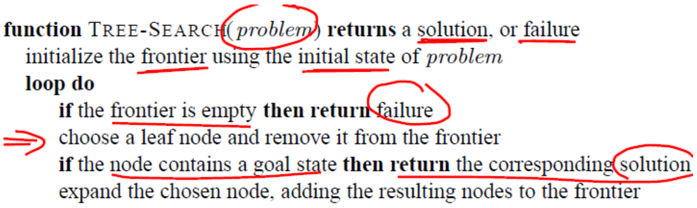
3.3 Problem Types

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 and 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 state, Better - 2057 states.
Examples of Real-World Problems: Route planning, shortest path problem, TSP, VLSI Layout, Robot nav., Assembly sequencing.

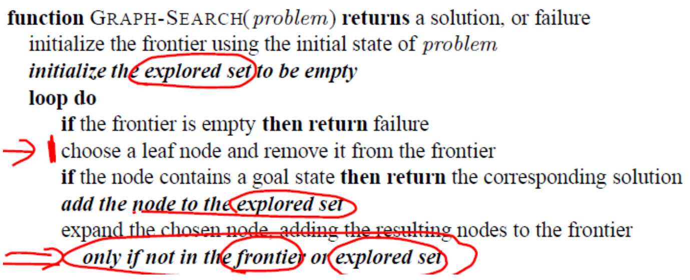
3.4 Search strategies

E.g.: node expansion, frontier, search strategy, tree-based search, graph-based search.
- Search Tree Data structure:
state, parent, action, path-cost.





```
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
```



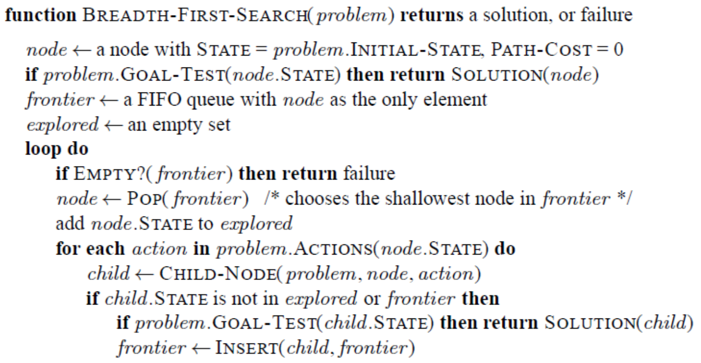
```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node 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 at d). **Space Complexity: O(b at d)**



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

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

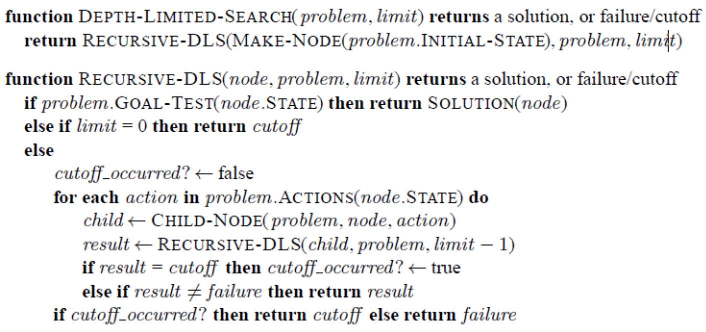
- Uniform-Cost Search:

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)) >= 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.

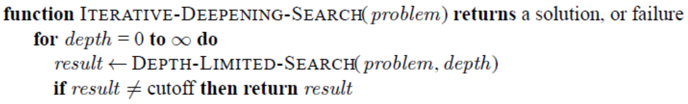
Generally, optimal is not guaranteed. Completeness only for graph-based search. **Time complexity:** in graph-based is bounded by the space, so it can be infinite, in tree-based: O(b at m) (m max length of a path). **Space Complexity:** tree-based: O(b*m), graph-based: worst-case, all states need to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



```
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? ← false
      for each action in problem.ACTIONS(node.STATE) do
         child ← CHILD-NODE(problem, node, action)
         result ← RECURSIVE-DLS(child, problem, limit − 1)
         if result = cutoff then cutoff_occurred? ← true
         else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

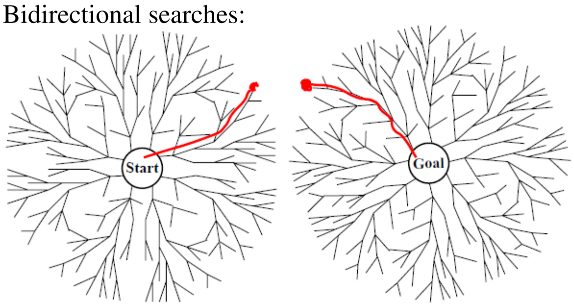
- Iterative Deepening Search:

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).

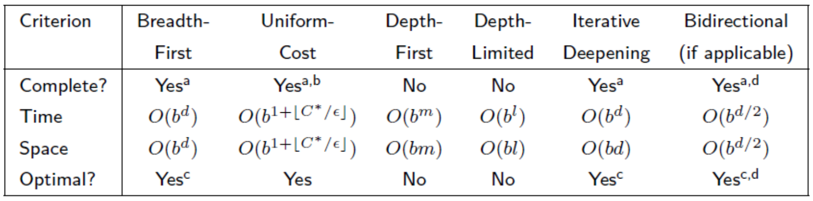


```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

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-fist.
- Bidirectional searches:



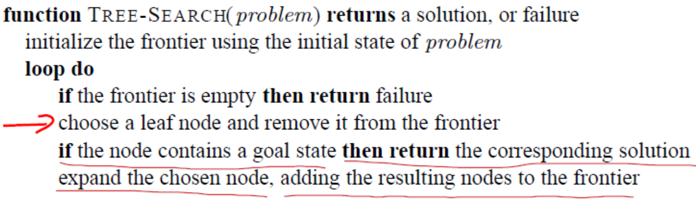
As long as forward and backward searches are symmetric, search times of O(2*b at d/2) = O(b at d/2) can be obtained. The operators are not always reversible, there must be an efficient way to check if a new node already appears in the search tree of the other half of the search.



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]

4 Informed Search Methods

- 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.
- 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

h(n) = estimated path-costs from n to the goal. A best-first search using h(n) (heuristic function) as evaluation function is called greedy search.

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*

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) <= h*(n).

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 G2 with g(G2) ¿ 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)<=f* −− > f(G2)<=f(n) −− > f(G2)<=f* ==> g(G2)<=f* Contraddiciton.
- 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 δ ¿ 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

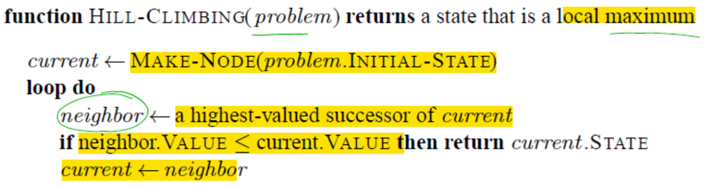
For the graph-based variant, either needs to consider re-opening nodes from the explored set, when a better estimate becomes known, or needs to require stronger restrictions on the heuristic estimate: it needs to be consistent (iff for all actions a leading from s to s': h(s) - h(s') <= c(a), where c(a) 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*

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.
- MA* and SMA*.

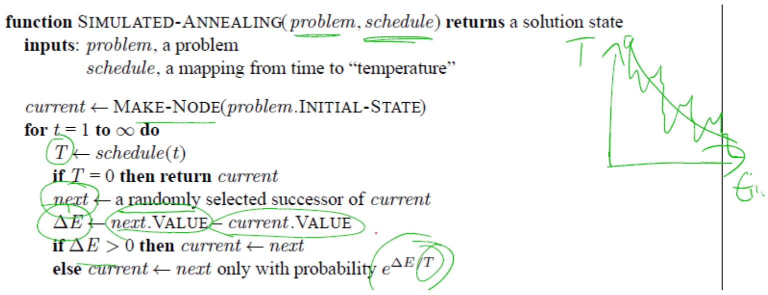
4.4 Local Search Methods

In many problems we are interested only to 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.



```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← 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 ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
     T ← schedule(t)
     if T = 0 then return current
     next ← a randomly selected successor of current
     ΔE ← next.VALUE – current.VALUE
     if ΔE > 0 then current ← next
     else current ← next only with probability $e^{\Delta E/T}$
```

4.5 Genetic Algorithms

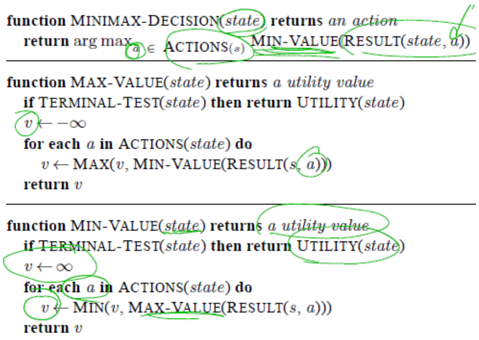
Similar to evolution, we search for solutions by three operators: ”mutation”,”crossover”, and ”selection”. Need of coding a solution into a string of symbols or bit-string, fitness function to judge the worth of configurations.
Example: Represent an individual (one 8-queens configuration) as a concatenation of eight x-y coordinates. Its fitness is judged by the number of non-attacks. The population consists of a set of configurations.

5 Board games

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

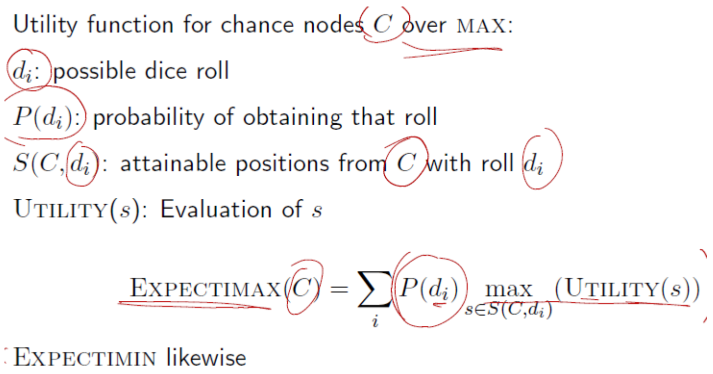
In contrast to regular searches, where a path from beginning to end is a solution, max must come up with a strategy to reach a favorable terminal state regardless of what min does $->$ all of min moves must be considered and reactions to them must be computed.
When it is possible to produce the full 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 max_{a ∈ 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!



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

$\textsc{Utility}(s)$: Evaluation of s

$$\textsc{Expectimax}(C) = \sum_i P(d_i) \max_{s \in S(C,d_i)} (\textsc{Utility}(s))$$

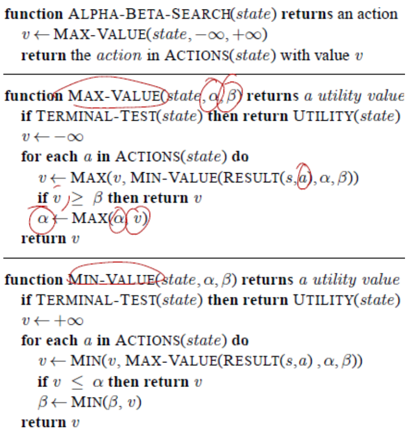
$\textsc{Expectimin}$ likewise

Figure 4: Expected value

Standard evaluation functions are weighted linear - 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, α, β) 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), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) 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) , α, β))
    if v ≤ α then return v
    β ← MIN(β, 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 at d/2). Average case: O(b at 3d/4).Practical case: A simple ordering heuristic brings the performance close to the best case.

5.3 Games with an Element of Chance

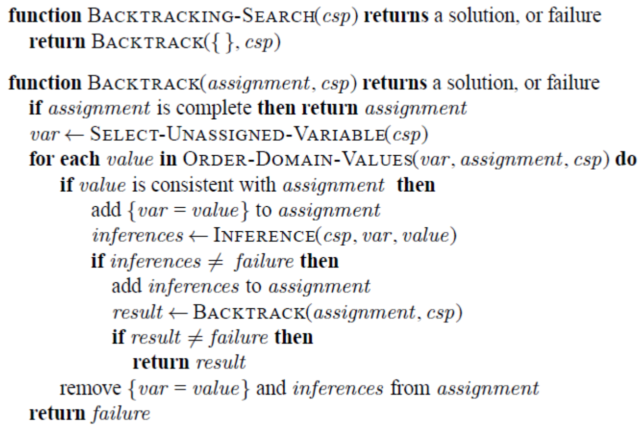
In addition to min- and max nodes, we need chance nodes (e.g. for the dice).
Search costs increase: Instead of O(b at d), we get O((b x n)at d), where n is the number of possible dice outcomes.

6 Constraint satisfaction problems

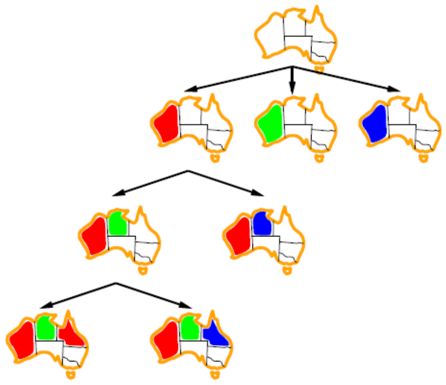
Such problem is defined by:
- variables, a set of value domains, a set of constraints, an assignment.
Main idea is to exploit the constraints to delete large portions of search space.
A constraint graph can be used to visualize binary constraints, Nodes = variables, arcs = constraints.
State: a variable assignment.

6.1 Backtracking search for CSPs

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

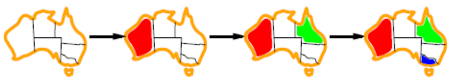


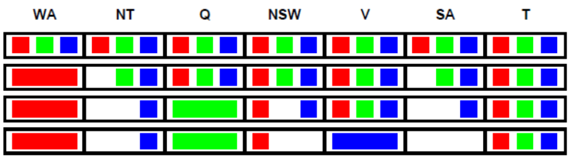
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

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 ← INFERENCE(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 - Rule out failures early: e.g. Forward checking:



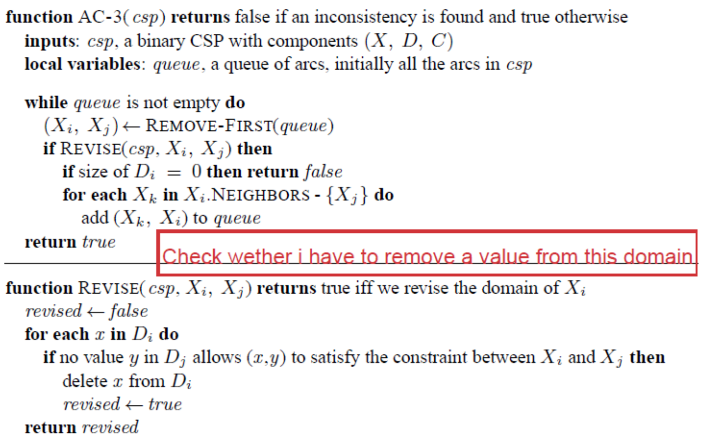


WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥 🟦	🟥 🟩 🟦	🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥	🟦		🟥 🟩 🟦

6.3 Constraint Propagation

A problem of Forward Checking is that constraints of unassigned variables are not propagated.

A directed arc X − > Y is consistent iff for every choice of x there exists y that satisfies constraints; 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
    ($X_i$, $X_j$) ← REMOVE-FIRST(queue)
    if REVISE(csp, $X_i$, $X_j$) then
      if size of $D_i$ = 0 then return false
      for each $X_k$ in $X_i$.NEIGHBORS - {$X_j$} do
        add ($X_k$, $X_i$) to queue
  return true
```

Check wether i have to remove a value from this domain

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

Time complexity: O(

$$d^3 * n^2)$$

). Of course, AC-3 does not detect all inconsistencies (which is an NP-hard problem).

6.4 Problem Structure

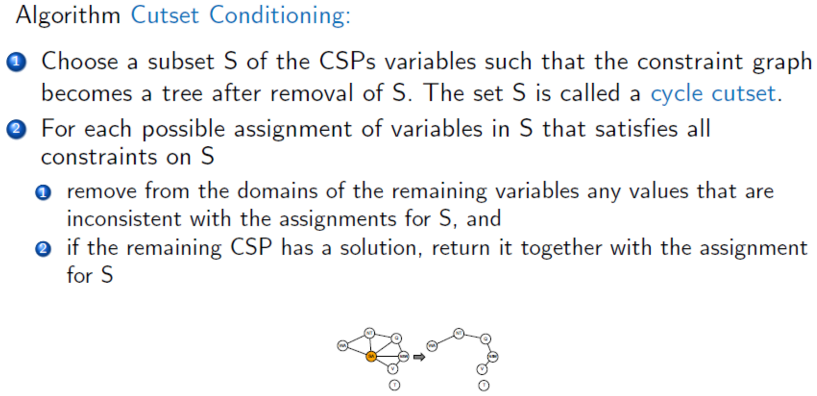
If the CSP graph is a tree, then it can be solved in O(

$$nd^2$$

) (general CSPs need in the worst case O(

$$d^n$$

)). 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 CSPs 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(

$$nd^{(}w + 1)$$

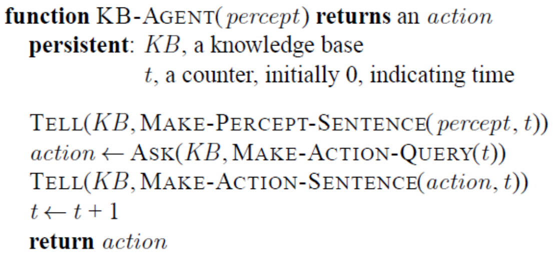
).

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 compose 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 ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

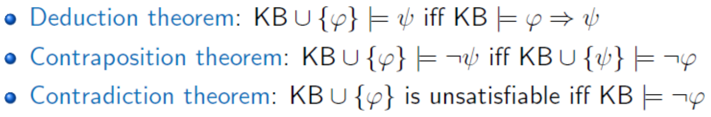
A logic also defines the semantics or meaning of sentences, defines the truth of a sentence with respect to each possible world. If a sentence α is true in a possible world m, we say that m satisfies α or m is a model of α.
- Logical entailment: α —= β iff in every model in which α is true, β is also true. (follow from KB) e.g.: x = 0 | = xy = 0.
- Inference: we can derive α with an inference method i. This is written as: KB —- α. (Derivation)
- Declarative Languages: we state what we want to compute, not how. System believes P iff it consider P true. We must know symbols, when a sentence is true etc.
We'd like to have inference algorithms that derive only sentences that are entailed (*soundness*) and **all** of them (*completeness*). The building blocks of propositional logic are indivisible, atomic statements.

7.1.1 Syntax

An Interpretation I is called model of ω if I | = ω. A truth assignment of the atoms in Σ , or an interpretation I over Σ , is a function: I:Σ − > T,F.
A formula ϕ can be: satisfiable, unsatisfiable, falsifiable (there exists I that doesn't satisfy ω), valid (tautology: I | = ω ∀ I), logically equivalent (I| = ω iff I| = χ ∀I). A method to decide if a formula is satisfiable is: Truth Table.

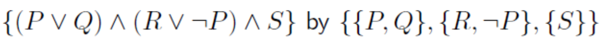
7.1.2 Normal forms

- 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.
Some properties of logical implication:



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

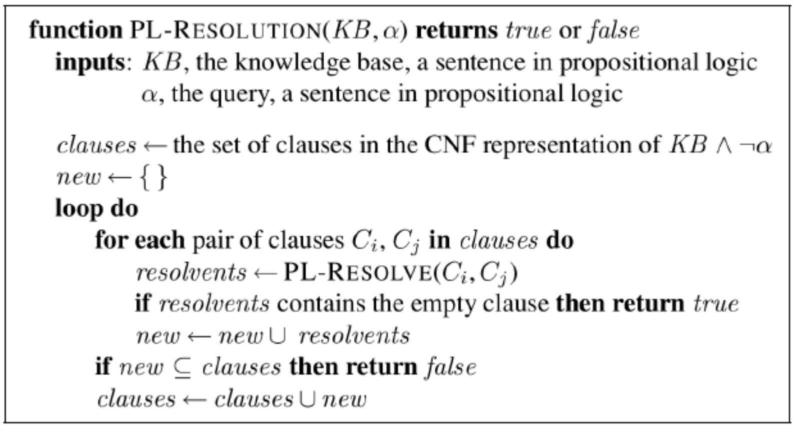
We can often derive new formulae from formulae in the KB. These new formulae should follow logically from the syntactical structure of the KB formulae.
Calculus: Set of inference rules (potentially including so-called logical axioms). A calculus C is sound (or correct) if all formulae that are derivable 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.
Resolution
We want a way to derive new formulae that does not depend on testing every interpretation (idea: to prove that KB | = ϕ, we can prove that KB $\bigcup \neg\phi$ is unsatisfiable (contradiction theorem)).



$\{(P \vee Q) \wedge (R \vee \neg P) \wedge S\}$ by $\{\{P, Q\}, \{R, \neg P\}, \{S\}\}$

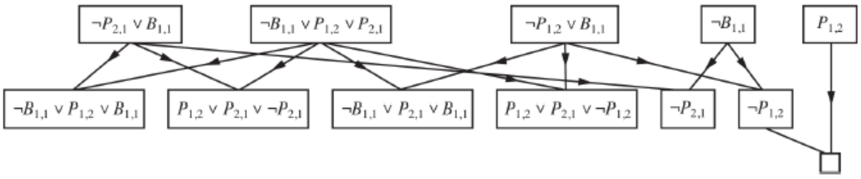
$$\frac{C_1\dot{\cup}\{l\}, C_2\dot{\cup}\{\bar{l}\}}{C_1 \cup C_2}$$

$C_1 \cup C_2$ 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.



```
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:


$\neg P_{2,1} \vee B_{1,1}$
$\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$
$\neg P_{1,2} \vee B_{1,1}$
$\neg B_{1,1}$
$P_{1,2}$
$\neg B_{1,1} \vee P_{1,2} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{2,1}$
$\neg B_{1,1} \vee P_{2,1} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{1,2}$
$\neg P_{2,1}$
$\neg P_{1,2}$
□

Lemma (soundness) If Δ|– D, then Δ| = D.
Proof idea: Since all D ∊ R(Δ) follow logically from Δ, the Lemma results through induction over the length of the derivation.
Is resolution also complete? So the opposite implications are also valid. Not in general.
However, it can be shown that resolution is refutation-complete: Δ is unsatisfiable implies Δ|– empty.
Theorem: Δ is unsatisfiable iff Δ|– empty.

We can now infer new facts, but how do we translate knowledge into action?
Negative selection: Excludes any provably dangerous actions.
A1,1 ∧ EastA ∧ W2,1 => ¬Forward
Positive selection: Only suggests actions that are provably safe.
A1,1 ∧ EastA ∧ ¬W2,1 =>Forward

8 Satisfiability and Model Construction

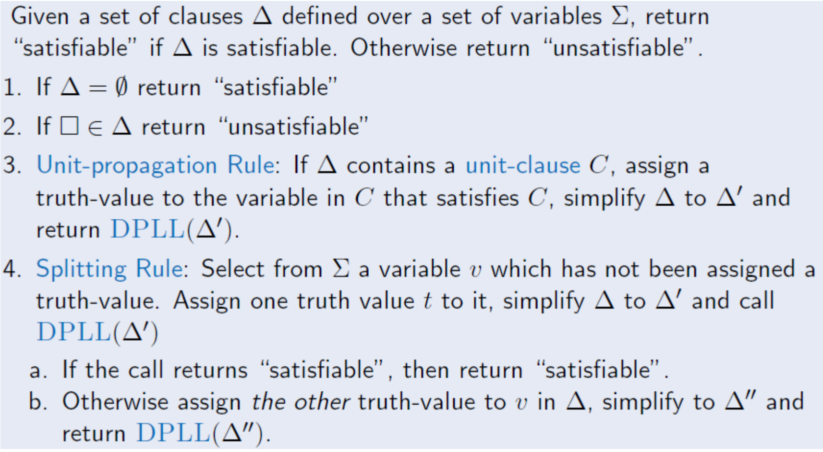
SAT solving is the best available technology for practical solutions to many NP-hard problems. Differently from Logical deduction, SAT returns a model of the theory (solution) given a logical theory. SAT can be formulated as a Constraint-Satisfaction-Problem (– > search):

- CSP-variables: alphabet
- Domain: True, False

- Constraints given by clauses

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 $\Box \in \Delta$ 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 $\mathrm{DPLL}(\Delta')$.
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 $\mathrm{DPLL}(\Delta')$
   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 $\mathrm{DPLL}(\Delta'')$.

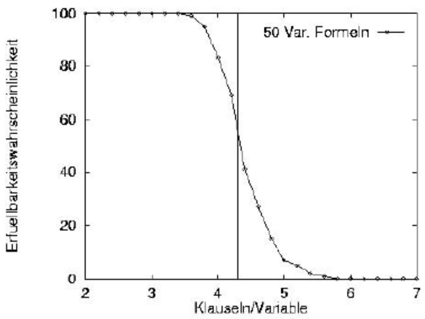
DPLL is complete, correct, and guaranteed to terminate. DPLL constructs a model, if one exists. In general, DPLL requires exponential time (splitting rule!), Heuristics are needed. DPLL is polynomial on Horn clauses. Horn Clauses constitute an important special case, since they require only polynomial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

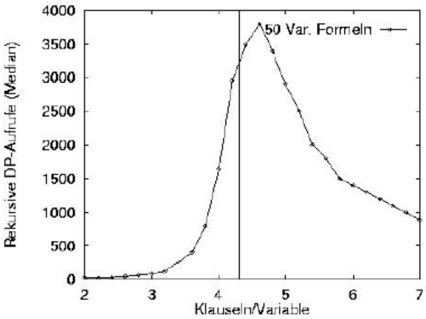
8.1.1 DPLL on Horn Clauses

- **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.

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.

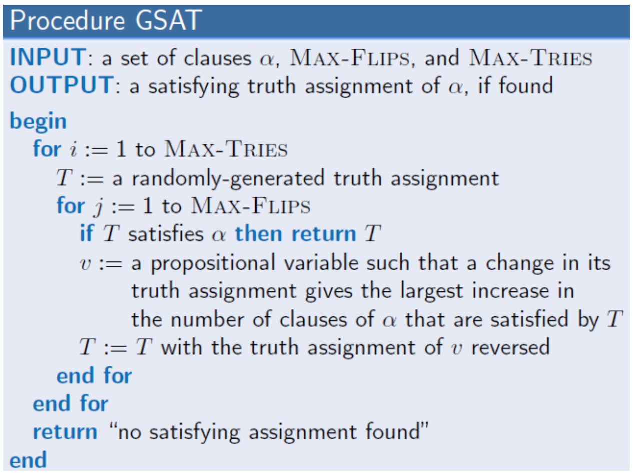
When the probability of a solution is close to 1 **(under-constrained)**, there are many solutions. If the probability of a solution is close to 0 **(over-constrained)**.





8.2 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
```

8.3 State of the Art

8.3.1 Improvements of DPLL Algorithms

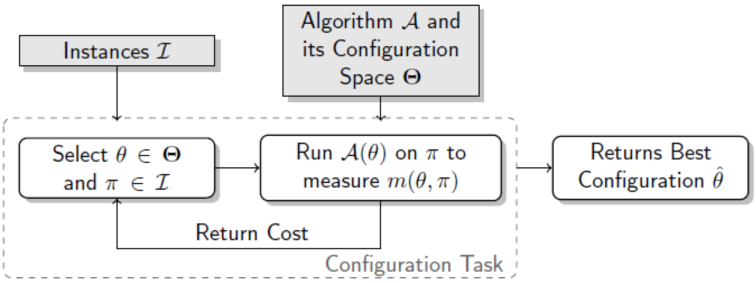
Branching on variables can cause stopping, we can "learn" **(here: logically infer)** a new clause (negation of a variable). Leads to conflict-directed clause learning (CDCL).
Both for DPLL/CDCL algorithms and local search algorithms

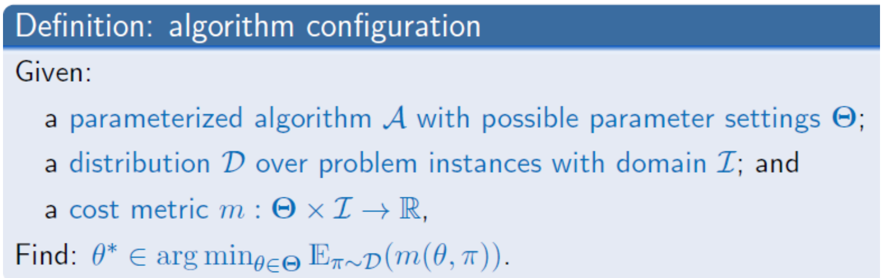
- Randomization and restarts
- Efficient data structures and indexing
- Engineering ingenious heuristics

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.

8.3.2 Algorithm configuration


Instances $\mathcal{I}$
Algorithm $\mathcal{A}$ and its Configuration Space $\mathbf{\Theta}$
Select $\theta \in \mathbf{\Theta}$ and $\pi \in \mathcal{I}$
Run $\mathcal{A}(\theta)$ on π to measure $m(\theta, \pi)$
Returns Best Configuration $\hat{\theta}$
Return Cost
Configuration Task



Definition: algorithm configuration

Given:

a parameterized algorithm $\mathcal{A}$ with possible parameter settings $\mathbf{\Theta}$;

a distribution $\mathcal{D}$ over problem instances with domain $\mathcal{I}$; and

a cost metric $m : \mathbf{\Theta} \times \mathcal{I} \to \mathbb{R}$,

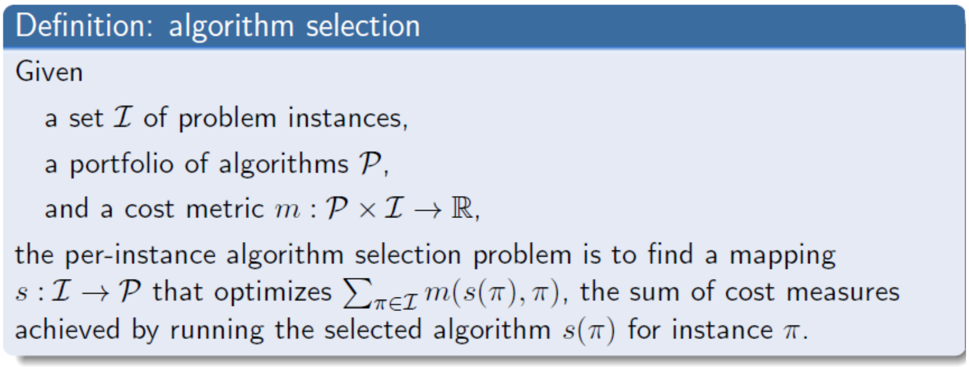
Find: $\theta^* \in \arg\min_{\theta \in \mathbf{\Theta}} \mathbb{E}_{\pi \sim \mathcal{D}}(m(\theta, \pi))$.

- 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.

8.3.3 Algorithm selection



Definition: algorithm selection

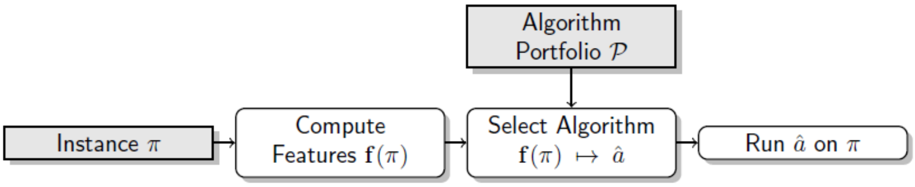
Given

a set $\mathcal{I}$ of problem instances,

a portfolio of algorithms $\mathcal{P}$,

and a cost metric $m : \mathcal{P} \times \mathcal{I} \to \mathbb{R}$,

the per-instance algorithm selection problem is to find a mapping $s : \mathcal{I} \to \mathcal{P}$ 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.

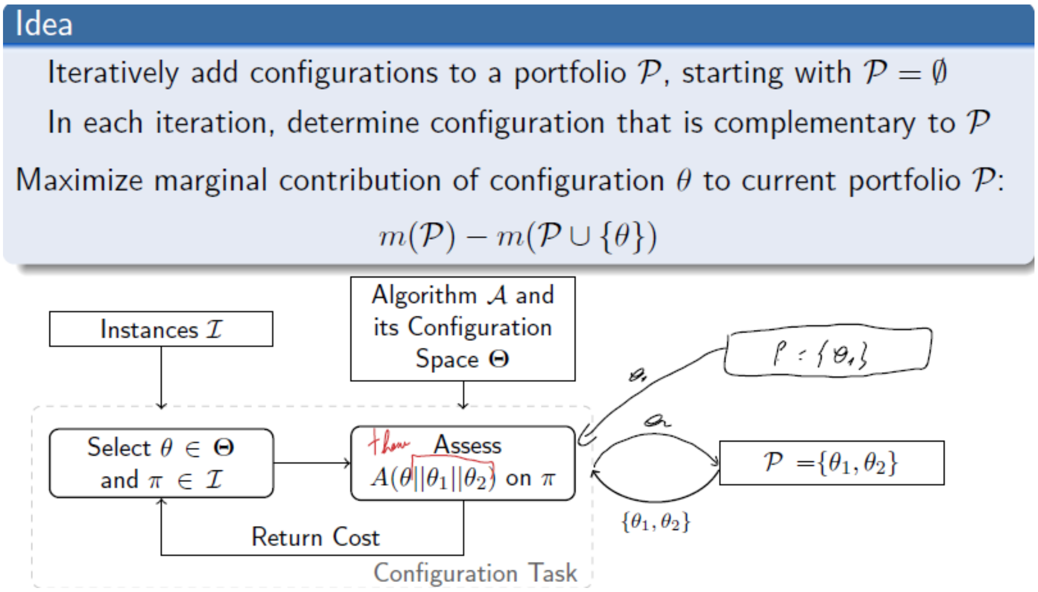

Algorithm Portfolio $\mathcal{P}$
Instance π
Compute Features $\mathbf{f}(\pi)$
Select Algorithm $\mathbf{f}(\pi) \mapsto \hat{a}$
Run $\hat{a}$ on π

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

8.3.4 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 $\mathcal{P}$:

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

Instances $\mathcal{I}$
Algorithm $\mathcal{A}$ and its Configuration Space Θ
Select $\theta \in \Theta$ and $\pi \in \mathcal{I}$
then
Assess $A(\theta||\theta_1||\theta_2)$ on π
Return Cost
Configuration Task
$\rho : \{\theta_1\}$
θ_1
θ_2
$\mathcal{P} = \{\theta_1, \theta_2\}$
$\{\theta_1, \theta_2\}$

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 function. They have an arity (number of arguments): 0-ary predicate: propositional logic atoms, 0-ary function = constants.
Terms: Every variable is a term, If t1, t2,..., tn are terms and f is an n-ary function, then f(t1, t2,..., tn) is also a term. Terms without variables: ground terms.
Atomic Formulae: Represent statements about objects.

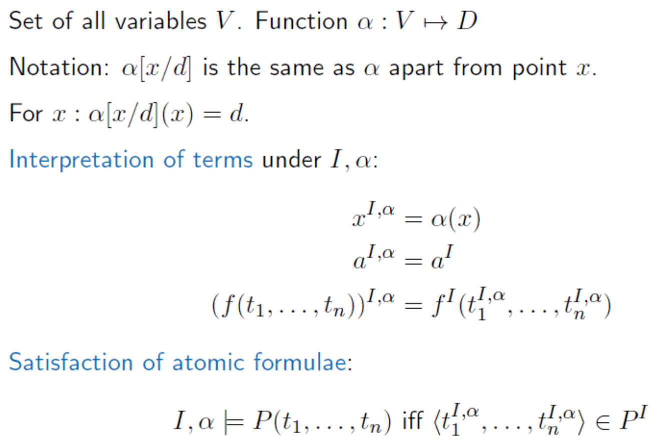
- If t1, t2,...,tn are terms and P is an n-ary predicate, then P(t1, t2,...,tn) is an atomic formula.
- If t1 and t2 are terms, then t1 = t2 is an atomic formula. Atomic formulae without variables: ground atoms (contain only ground terms).

9.1 Semantics of PL1-Logic

Interpretation: I = ¡D,x^I¿ 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: $(f(t1, ..., tn))^I = f^I(t_1^I, ..., t_n^I)$.
Satisfaction of ground atoms P(t1,...,tn): I | = P(t1,...,tn) iff ¡$t_1^I, ..., t_n^I \in P^I$.

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

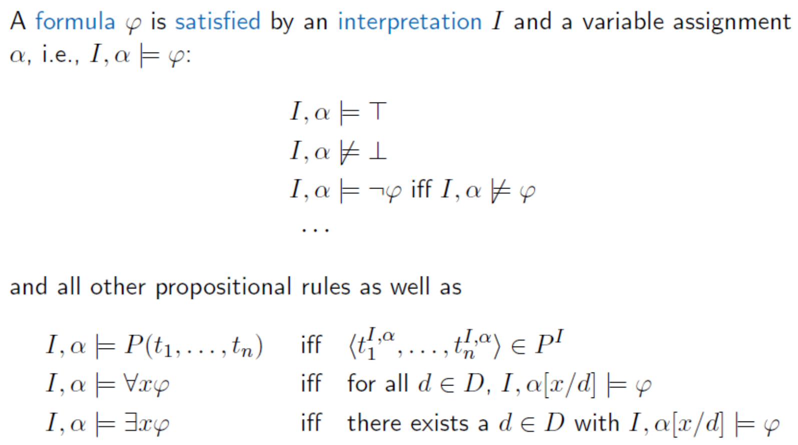
Interpretation of terms under I, α:

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

Satisfaction of atomic formulae:

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

9.1.2 Satisfiability



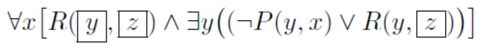
A formula φ is satisfied by an interpretation I and a variable assignment α, i.e., $I, \alpha \models \varphi$:

$$I, \alpha \models \top$$
$$I, \alpha \not\models \bot$$
$$I, \alpha \models \neg\varphi \text{ iff } I, \alpha \not\models \varphi$$
$$\dots$$

and all other propositional rules as well as

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

9.2 Free and Bound Variables



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

When boxed, is called *free*, otherwis **bound**. Formulae with no free variables are called *closed formulae* or *sentences*. With closed formulae, 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: $I,\alpha| = \omega$

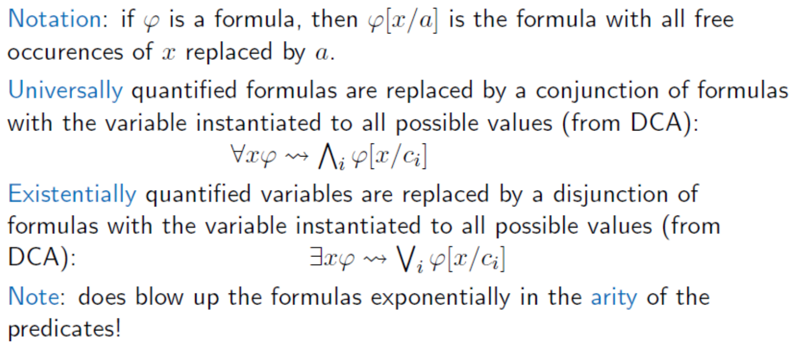
9.3 Derivation in PL1

Reduction to propostional logic by instantiation based on the so-called **Herbrand Universe** (all possible terms) $->$ infinite propositional theories.
Simple way for special case: If the number of objects is **finite**, instantiate all variables by possible objects.

9.3.1 Finite universes

Domain closure axiom (DCA): $\forall[x = c_1 or....orx = c_n]$
unique name assumption/axiom or UNA: $and_{i!=j}[c_i! = c_j]$
Eliminate quantification by instantiating all variables with all possible values.

9.3.2 Instantiation



Notation: if φ is a formula, then $\varphi[x/a]$ is the formula with all free occurences 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!

10 Action Planning

Planning is the art and practive 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.

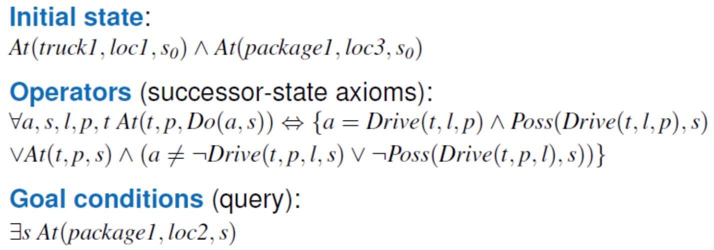
Planning is not searching, synthesis or scheduling.

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

10.1.2 Planning as Logical Inference



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: **ST**anford **R**esearch **I**nstitute **P**roblem **S**olver

$\mathcal{S}$ 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**).

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

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

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

No domain constraints (although present in original formalism)

- **Operator:** o = ¡para,pre,eff¿
- **Operator instance or action:** operator with empty parameter list
- **State change:**

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

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: $drive(v1,s,t)$

A plan Δ is a sequence of actions:

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

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

Initial state: $$S \quad = \quad \left\{ \begin{array}{l} at(p1,c), at(p2,s), at(t1,c), \\ at(t2,c), street(c,s), street(s,c) \end{array} \right\}$$

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

Successful plan: $$\Delta \quad = \quad \begin{array}{l} \langle load(\overset{\text{in}}{p1, t1}, c), drive(\underline{t1, \overset{\text{to}}{c, s}}), \\ \underline{unload}(p1,t1,s), \underline{load}(p2,t1,s), \\ drive(t1,s,c), unload(p2,t1,c)\rangle \end{array}$$

```
(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)))
                    . . . )
```

Figure 5: Logistics Example

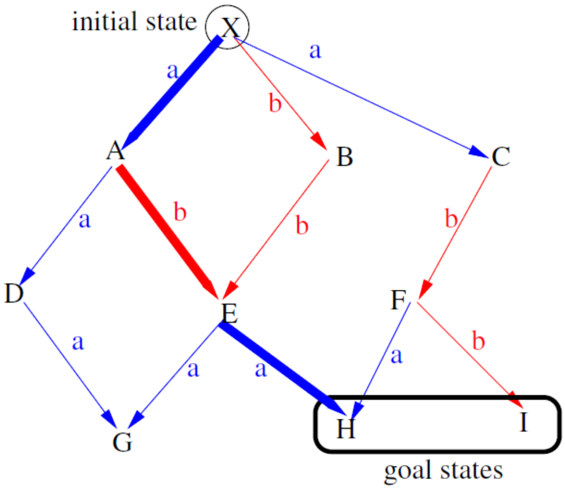
STRIPS as described above allows for unrestricted first-order terms.
Simplifications:

- **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.

10.1.4 PDDL: The Planning Domain Description Language

10.2 Basic Planning Algorithms

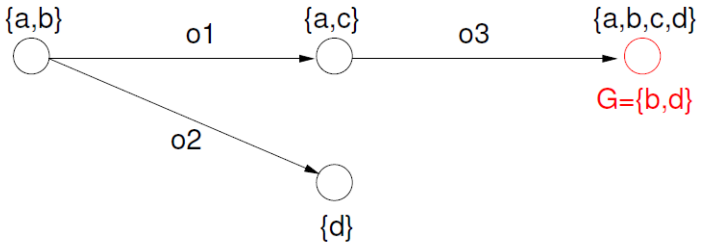
We can view planning problems as searching for goal nodes in a large labeled graph (transition system). Nodes are defined by the value assignment to the fluents (states). Labeled edges are defined by actions. Create the transition system on the fly and visit only the parts that are necessary.



- **Progression Planning: Forward Search** Start at initial state:
  1. Initialize: Δ = <>, add the initial state and make it S.
  2. Test whether we have reached goal: G in S? Return.
  3. Select one applicable action o_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).

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

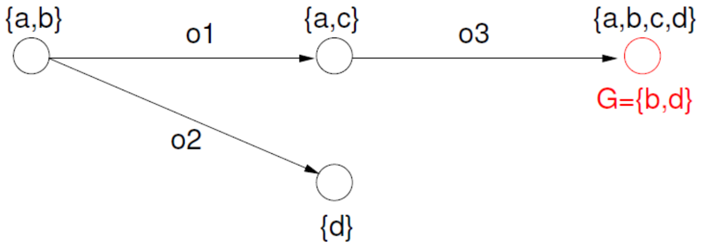


- **Regression Planning: Backward Search** Start from the goal:
  1. Initialize: Δ = <>, add the goal state and make it S.
  2. Test whether we have reached initial: I in S? Return.
  3. Select one applicable action o_i non deterministically which does not make sub-goals false, $S \cap \neg eff^-(o_i)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

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

  then extend plan $\Delta =< o_i, \Delta >$, back to step 2.
  (Instead of non-deterministic we can use some search strategy).

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



10.3 Computational Complexity

- **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.

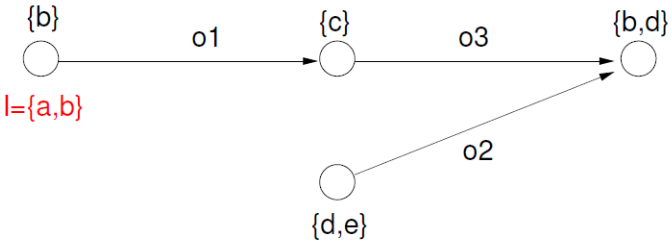
Theorem: *PLANEX for STRIPS with first-order terms is undecidable.*
Theorem: *PLANEX is PSPACE-complete for propositional STRIPS.*

- 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**

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

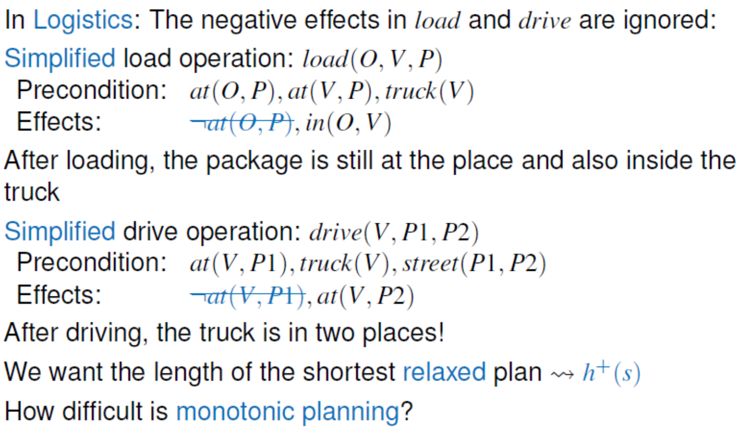


11 Making simple decisions under uncertainty (Probability)

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.

$$\text{Decision Theory} = \text{Utility Theory} + \text{Probability Theory}$$
$$argmax_a \sum_\omega p(\omega|a)[U(\omega) - c(a)]$$

- Decision-Theoretic Agent



In Logistics: The negative effects in *load* and *drive* are ignored:

Simplified load operation: $load(O,V,P)$

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

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

After loading, the package is still at the place and also inside the truck

Simplified drive operation: $drive(V,P1,P2)$

Precondition: $at(V,P1), truck(V), street(P1,P2)$

Effects: ~~$\neg at(V,P1)$~~, $at(V,P2)$

After driving, the truck is in two places!

We want the length of the shortest relaxed plan ⇝ $h^+(s)$

How difficult is monotonic planning?

11.1 Foundations of Probability Theory

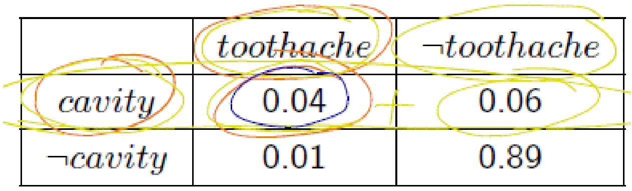
We use random variables such as Weather (capitalized word), which has a domain of ordered values. In our case that could be sunny, rain, cloudy, snow (lower case words).

- **Unconditional Probabilities**: P(a) denotes the unconditional probability that it will turn out that A = *true* in the absence of any other information.
- 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: $P(cavity|toothace) = 0.8$
  - Product rule: P($a \wedge b$)=P(a|b)P(b)

11.2 Probabilistic Inference

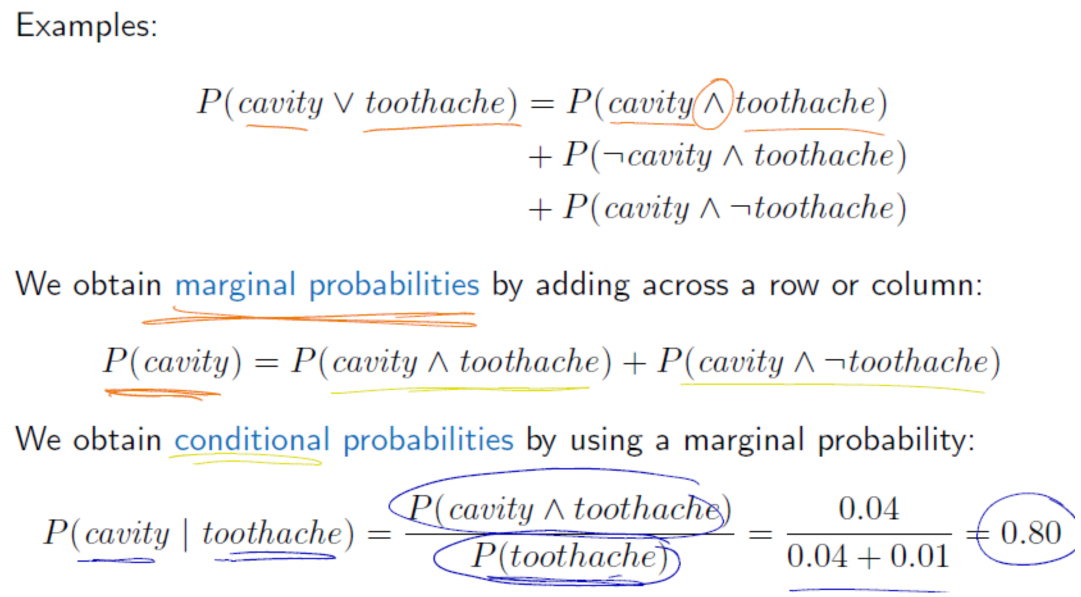
11.2.1 Joint Probability

The joint probability distribution P(X1,...,Xn) assigns a probability to every atomic event. Example of such a complete instantiation:



	toothache	¬*toothache*
cavity	0.04	0.06
¬*cavity*	0.01	0.89

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



Examples:

$$P(cavity \lor toothache) = P(cavity \land toothache) + P(\neg cavity \land toothache) + P(cavity \land \neg toothache)$$

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

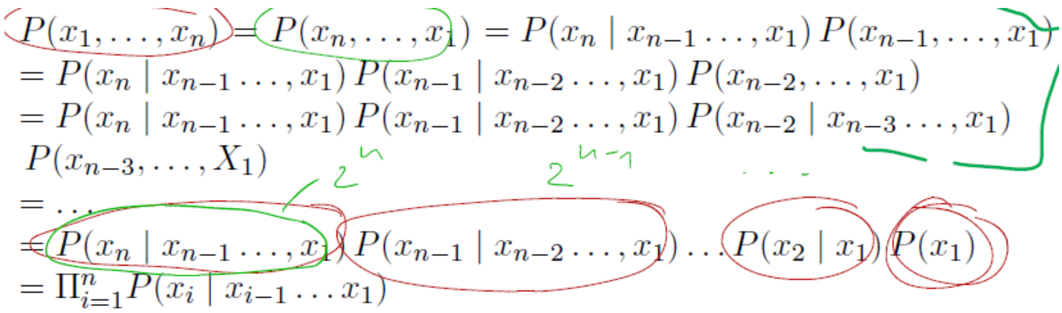
$$P(cavity) = P(cavity \land toothache) + P(cavity \land \neg toothache)$$

We obtain conditional probabilities by using a marginal probability:

$$P(cavity \mid toothache) = \frac{P(cavity \land toothache)}{P(toothache)} = \frac{0.04}{0.04 + 0.01} = 0.80$$

- 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.



$$P(x_1, \ldots, x_n) = P(x_n, \ldots, x_1) = P(x_n \mid x_{n-1} \ldots, x_1)\, P(x_{n-1}, \ldots, x_1)$$
$$= P(x_n \mid x_{n-1} \ldots, x_1)\, P(x_{n-1} \mid x_{n-2} \ldots, x_1)\, P(x_{n-2}, \ldots, x_1)$$
$$= P(x_n \mid x_{n-1} \ldots, x_1)\, P(x_{n-1} \mid x_{n-2} \ldots, x_1)\, P(x_{n-2} \mid x_{n-3} \ldots, x_1)$$
$$P(x_{n-3}, \ldots, X_1)$$
$$= \ldots$$
$$= P(x_n \mid x_{n-1} \ldots, x_1)\, P(x_{n-1} \mid x_{n-2} \ldots, x_1) \ldots P(x_2 \mid x_1)\, P(x_1)$$
$$= \Pi_{i=1}^{n} P(x_i \mid x_{i-1} \ldots x_1)$$

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:

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

Generalizing:

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

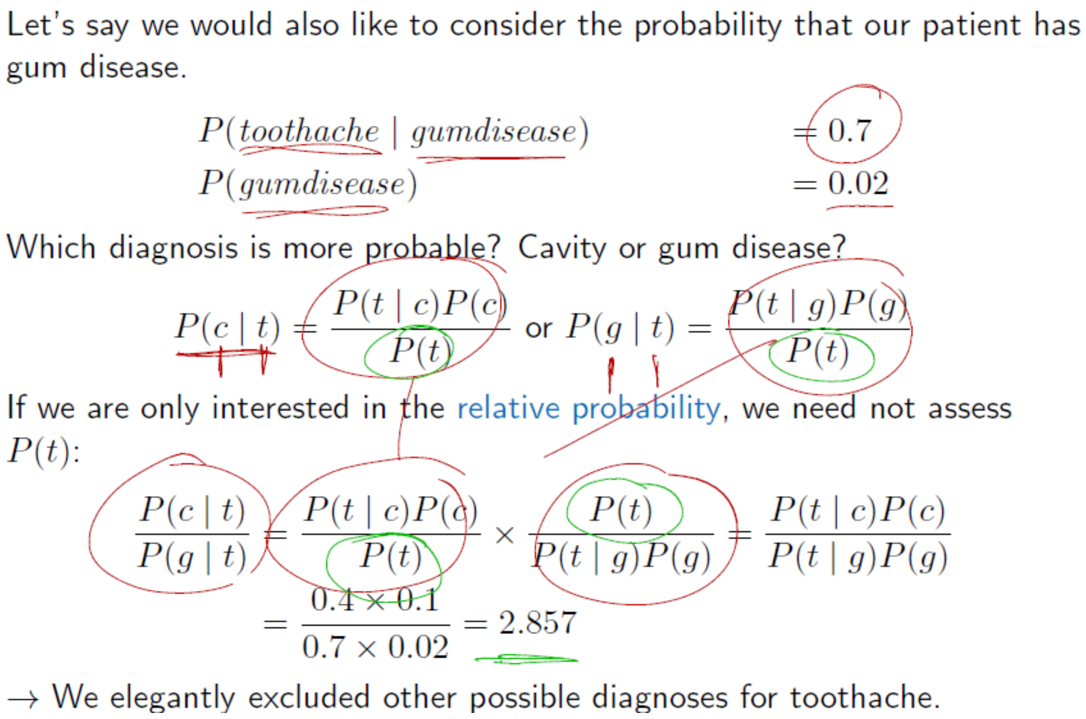
- Example:

$$\begin{aligned} P(toothache \mid cavity) &= 0.4 \\ P(cavity) &= 0.1 \\ P(toothache) &= 0.05 \end{aligned}$$

$$P(cavity \mid toothache) = \frac{0.4 \times 0.1}{0.05} = 0.8$$

P(toothache | cavity) (causal) is more robust than P(cavity | toothache) (diagnostic).

11.2.3 Relative Probability



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

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

Which diagnosis is more probable? Cavity or gum disease?

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

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

$$\frac{P(c \mid t)}{P(g \mid t)} = \frac{P(t \mid c)P(c)}{P(t)} \times \frac{P(t)}{P(t \mid g)P(g)} = \frac{P(t \mid c)P(c)}{P(t \mid g)P(g)}$$

$$= \frac{0.4 \times 0.1}{0.7 \times 0.02} = 2.857$$

→ We elegantly excluded other possible diagnoses for toothache.

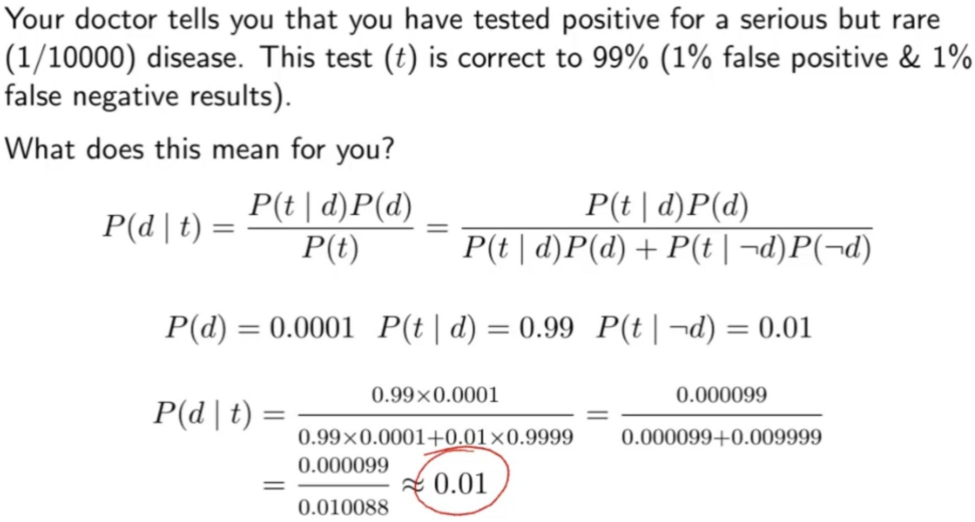
11.2.4 Normalization

To compute absolut 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.: $\alpha(.1, .1, .3) = (.2, .2, .6)$)



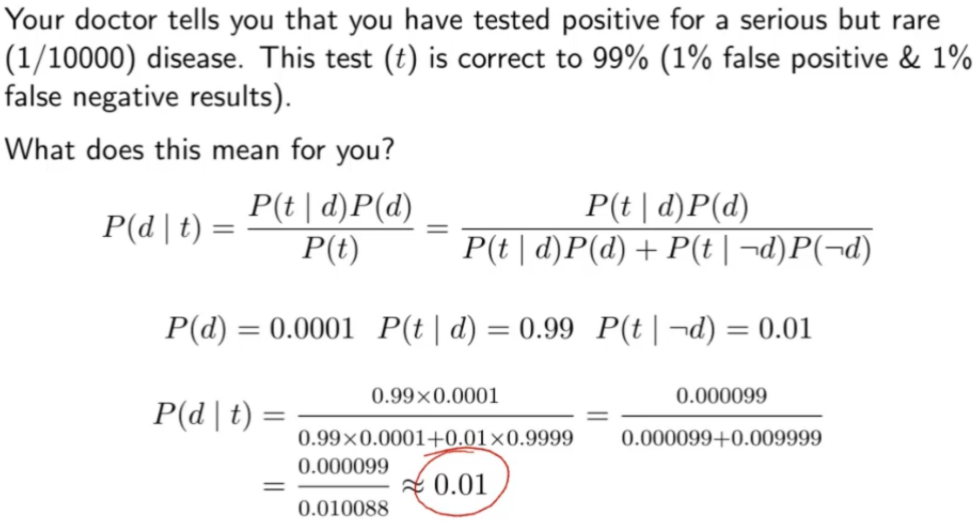
Your doctor tells you that you have tested positive for a serious but rare (1/10000) disease. This test (t) is correct to 99% (1% false positive & 1% false negative results).

What does this mean for you?

$$P(d \mid t) = \frac{P(t \mid d)P(d)}{P(t)} = \frac{P(t \mid d)P(d)}{P(t \mid d)P(d) + P(t \mid \neg d)P(\neg d)}$$

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

$$P(d \mid t) = \frac{0.99 \times 0.0001}{0.99 \times 0.0001 + 0.01 \times 0.9999} = \frac{0.000099}{0.000099 + 0.009999}$$
$$= \frac{0.000099}{0.010088} \approx 0.01$$



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?

$$P(d \mid t) = \frac{P(t \mid d)P(d)}{P(t)} = \frac{P(t \mid d)P(d)}{P(t \mid d)P(d) + P(t \mid \neg d)P(\neg d)}$$

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

$$P(d \mid t) = \frac{0.99 \times 0.0001}{0.99 \times 0.0001 + 0.01 \times 0.9999} = \frac{0.000099}{0.000099 + 0.009999}$$
$$= \frac{0.000099}{0.010088} \approx 0.01$$

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 (cavity — toothache) = 0.8. Furthermore, using Bayes' rule, we can calculate: **P (cavity — catch) = 0.95**

$$P(cav \mid tooth \wedge catch) = \frac{P(tooth \wedge catch \mid cav) \times P(cav)}{P(tooth \wedge catch)}$$
$$= \alpha P(tooth \wedge catch \mid cav) \times P(cav)$$

The dentist needs P $(tooth \wedge catch|cav)$, 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:
$P(tooth|catch, cav) = P(tooth|cav)$
If one already knows that there is a cavity, then the additional knowledge of the probe catching does not change the probability.
$P(tooth \wedge catch|cav) = P(tooth|catch, cav)P(catch|cav) =$
$P(tooth|cav)P(catch|cav)$

The general definition of conditional independence of two variables X and Y given a third variable Z (a common cause) is:
$P(X, Y|Z) = P(X|Z)P(Y|Z)$

11.2.6 Recursive Bayesian Updating

Assuming conditional independence, multiple evidence can be reduced to prior probabilities and conditional probabilities. The general combination rule, if Z1 and Z2 are independent given X is $P(X|Z1, Z2) = \alpha P(X)P(Z1|X)P(Z2|X)$where α is the normalization constant.
Generalization: $P(X|Z_1, ..., Z_n) = \alpha P(X)/PIP(Z_i|X)$

11.3 Bayesian Network

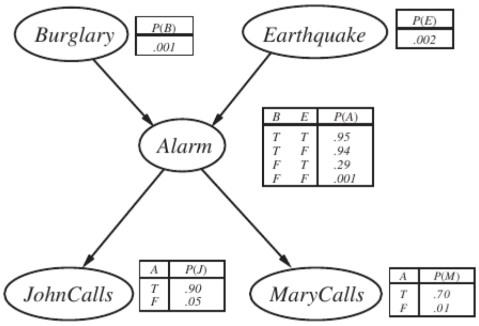
Example of Burglary/Earthquake.

- 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**).
$P(maryCalls|alarm, burglary) =$
$P(maryCalls|alarm) and P(maryCalls|alarm, burglary, johnCalls, earthquake)$
$= P(maryCalls|alarm)$

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

$P(x_1, ..., x_n) = P(x_n|x_{n-1}, ..., x_1) \cdot ... \cdot P(x_2|x_1)P(x_1) =$
$\Pi_{i=1}^{n} P(x_i|x_{i-1}, ..., x_1)$
This is equivalent to:

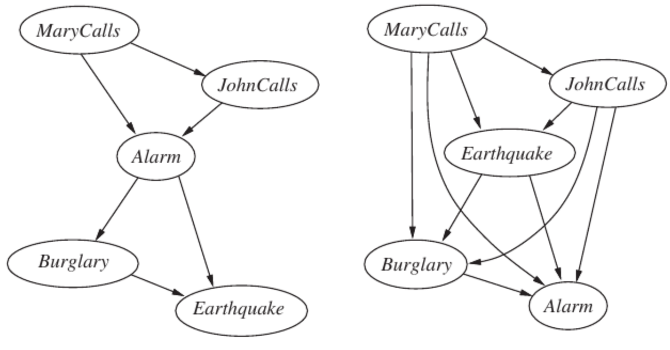
$P(x_1, ..., x_n) = \Pi_{i=1}^{n} P(x_i|parents(x_i))$



$P(j, m, a, b, e) = P(j|m, a, b, e)P(m|a, b, e)P(a|b, e)P(b|e)P(e)$
$= P(j|a)P(m|a)P(a|b, e)P(b)P(e)$
$= 0.90.70.0010.9990.998 = 0.00062$

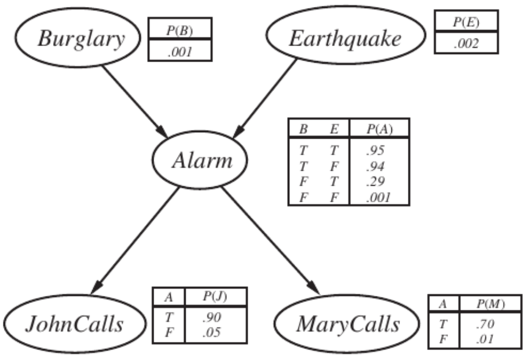
In general, we need a table of size 2^n where n is the number of variables. The size depends on the application domain (local vs. global interaction) and the skill of the designer.
- Naive Networks method :
- Order all variables
- Take the first from those that remain
- Assign all direct influences from 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 $= M, J, E, B, A$



(a) (b)

- Inference in Bayesian Network



P(B)
.001

P(E)
.002

B	E	P(A)
T	T	.95
T	F	.94
F	T	.29
F	F	.001

A	P(J)
T	.90
F	.05

A	P(M)
T	.70
F	.01

$P(burglary \mid johncalls)$
$P(burglary \mid johnCalls, maryCalls)$?

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

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

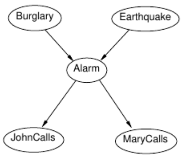
$= \alpha \sum_e \sum_a \mathbf{P}(B, j, m, e, a)$

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

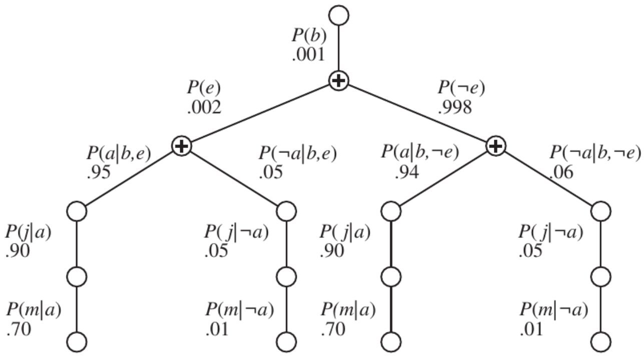
$P(b \mid j, m) = \alpha \sum_e \sum_a P(j \mid a) P(m \mid a) P(a \mid e, b) P(e) P(b)$

Reorganization of the terms yields:

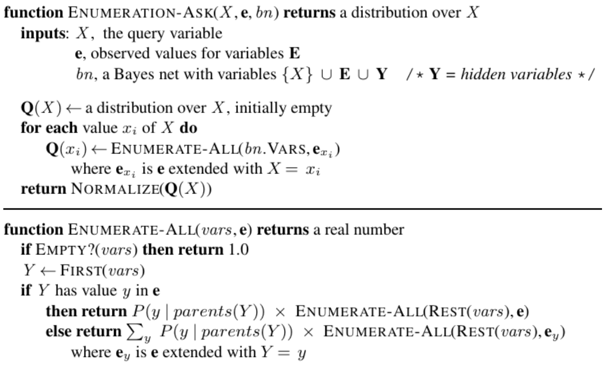
$P(b \mid j, m) = \alpha P(b) \sum_e P(e) \sum_a P(a \mid e, b) P(j \mid a) P(m \mid a)$



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



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

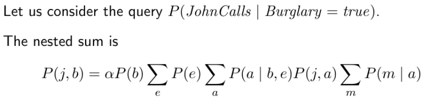


```
function ENUMERATION-ASK($X$, $\mathbf{e}$, $bn$) returns a distribution over $X$
   inputs: $X$, the query variable
           $\mathbf{e}$, observed values for variables $\mathbf{E}$
           $bn$, a Bayes net with variables $\{X\} \cup \mathbf{E} \cup \mathbf{Y}$   /* $\mathbf{Y}$ = hidden variables */

   $\mathbf{Q}(X) \leftarrow$ 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 $P(y \mid parents(Y)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}$)
       else return $\sum_y P(y \mid parents(Y)) \times$ 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 $P(JohnCalls \mid Burglary = true)$.

The nested sum is

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

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. M aryCalls 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.

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?
ß$P(Resulti(A)|Do(A), E)$
- Expected utility:
$EU(A|E) = \Sigma_i P(Result_i(A)|Do(A), E)U(Result_i(A))$
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:
$L_1 \succ L_2$ Agent prefers 1 over 2
$L_1 \sim L_2$ Agent is indifferent between 1 or 2
$L_1 \gtrsim L_2$ Agent prefers 1 or is indifferent.
Orderability
$(A \succ B) \vee (B \succ A) \vee (A \sim B)$

Transitivity
$(A \succ B) \vee (B \succ C) => (A \succ C)$

Continuity
$A \succ B \succ C => \exists p[p, A; 1-p, C] \sim B$

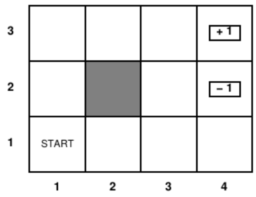
Substitutability
$A \sim B \rightarrow [p, A; 1-p, C] \sim [p, B; 1-p, C]$

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

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:
$U[p_1, S_1, ..., p_n, S_n]) = \Sigma_i p_i U(S_i)$
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 $u_m in$ and $u_m ax$ we can asses the utility of any particular outcome S by asking the agent to choose between S and a standard lottery $[p, u_m ax; 1-p, u_m in]$. We adjust p until they are equally preferred.

12.3 Sequential Decision Problems

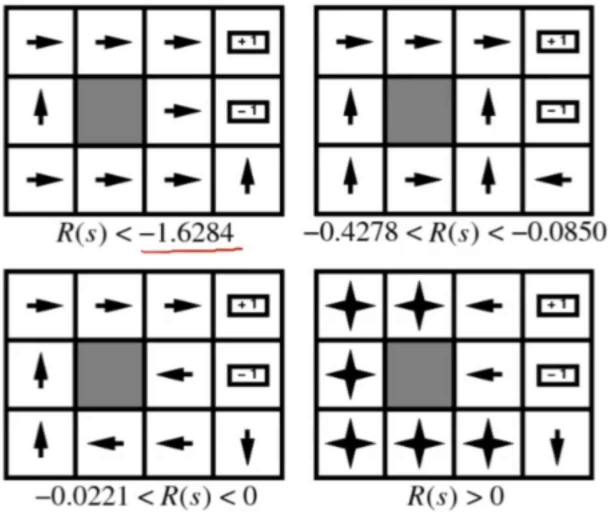


- **Deterministic version:**
  All actions always lead to the next square in the selected direction, except that moving into a wall results in no change in position.

- **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 $\pi*$ 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.



$R(s) < -1.6284$ $\quad$ $-0.4278 < R(s) < -0.0850$

$-0.0221 < R(s) < 0$ $\quad$ $R(s) > 0$

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

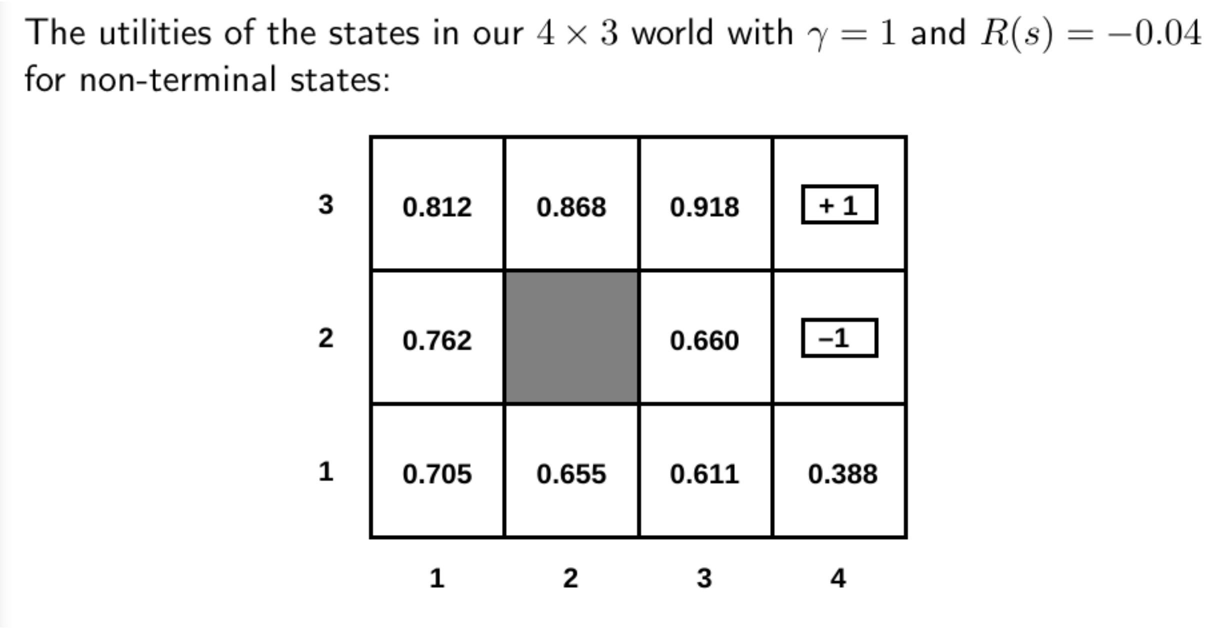
$\gamma \in [0, 1]$ is the discount factor. With discount, the utility is always finite. The discount factor expresses that future rewards have less than current rewards.

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:

$$U^\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi, s_0 = s] \tag{3}$$

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

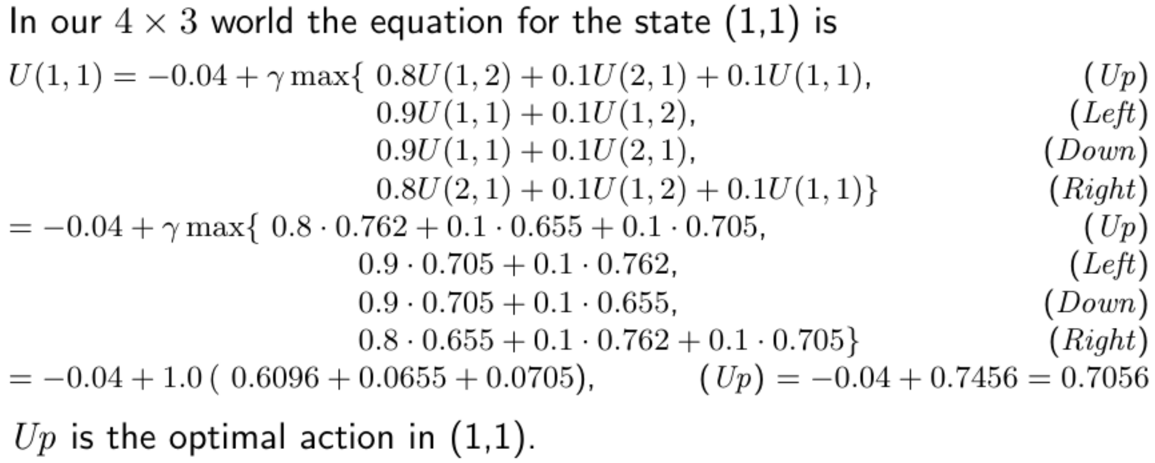
The agent simply chooses the action that maximizes the expected utility of the subsequent state:

$$\pi(s) = argmax_a \sum_{s'} P(s'|s,a)U(s') \tag{4}$$

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

$$U(s) = R(s) + \gamma max_a \sum_{s'} P(s'|s,a)U(s') \tag{5}$$

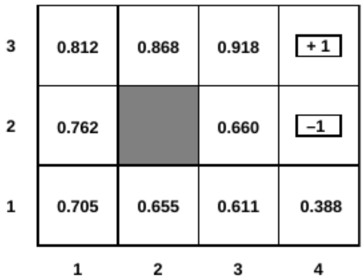
- Example:



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

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

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

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:

$$U_h([s_0, s_1, ..., s_n]) = f(s_0, U_h([s_1, ..., s_n])) \tag{6}$$

The simplest form is an additive reward function R:

$$U_h([s_0, s_1, ..., s_n]) = R(s_0) + U_h([s_1, ..., s_n]) \tag{7}$$

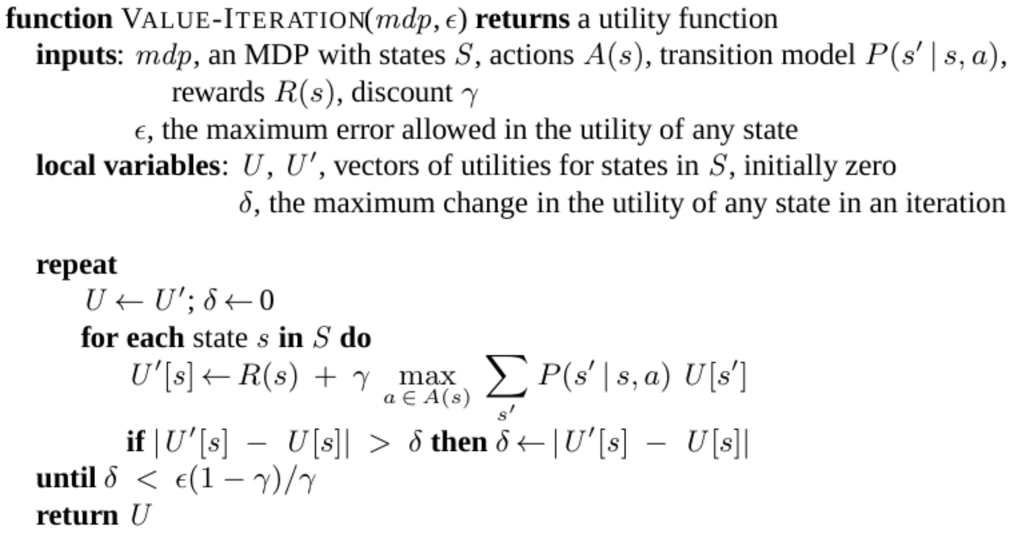
Typical problems contain cycles, which means the length of the histories is potentially infinite.Solution: Use

$$U_{t+1}(s) = R(s) + \gamma max_a \sum_{s'} P(s'|s, a)U_t(s') \tag{8}$$

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:

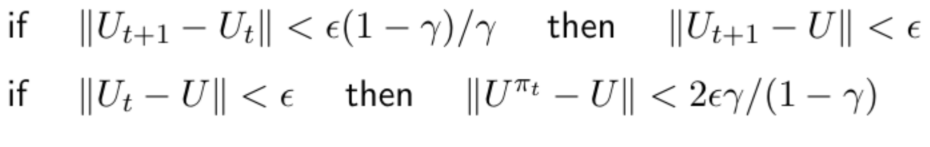
$$U(s') \leftarrow R(s) + \gamma max_a \sum_{s'} P(s'|s, a)U(s') \tag{9}$$

- Algorithm



```
function VALUE-ITERATION($mdp, \epsilon$) returns a utility function
  inputs: $mdp$, an MDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$,
              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
          $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
          if $|U'[s] - U[s]| > \delta$ then $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1 - \gamma)/\gamma$
  return $U$
```

- Convergence:



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

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

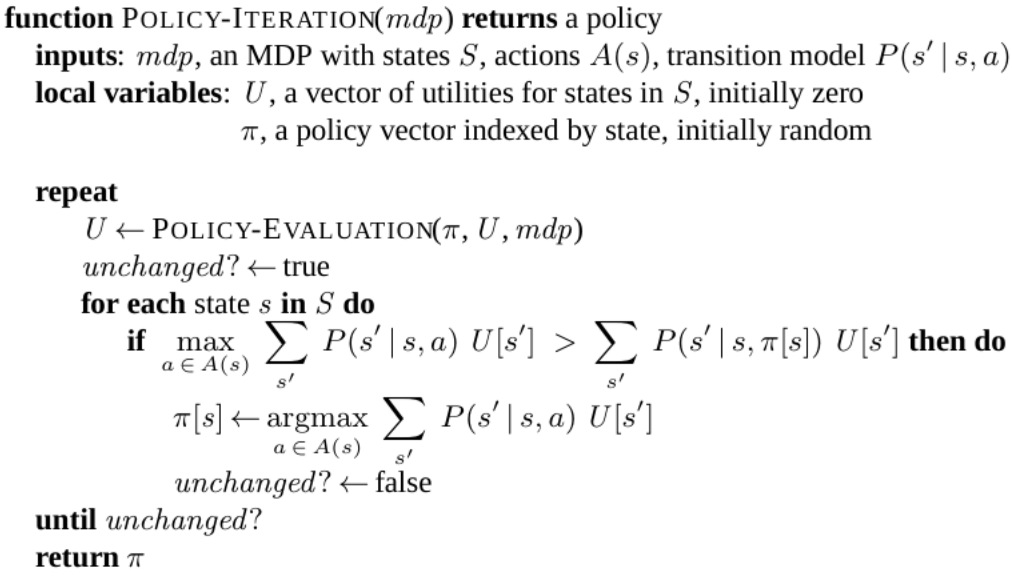
12.5.1 Policy Iteration

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:

$$\pi_{t+1}(s) = argmax_a \sum_{s'} P(s'|s, a)U_t(s') \tag{10}$$



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

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

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.
Learning element: Carries out improvements.
Critic: Evaluation of the agent's behavior based on a given external behavioral measure.
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.

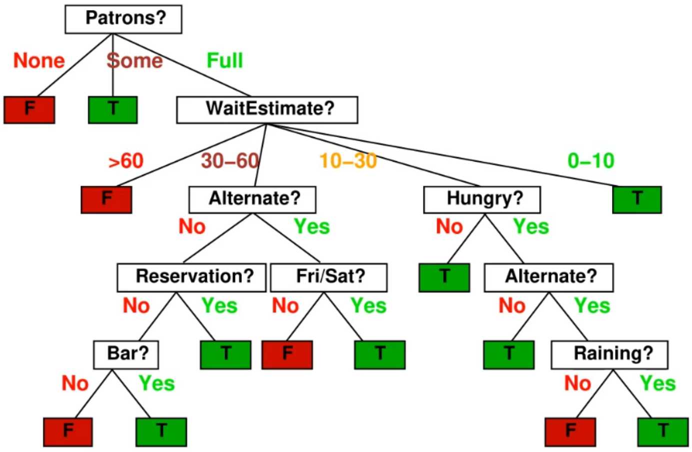
13.3 Decision trees

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:

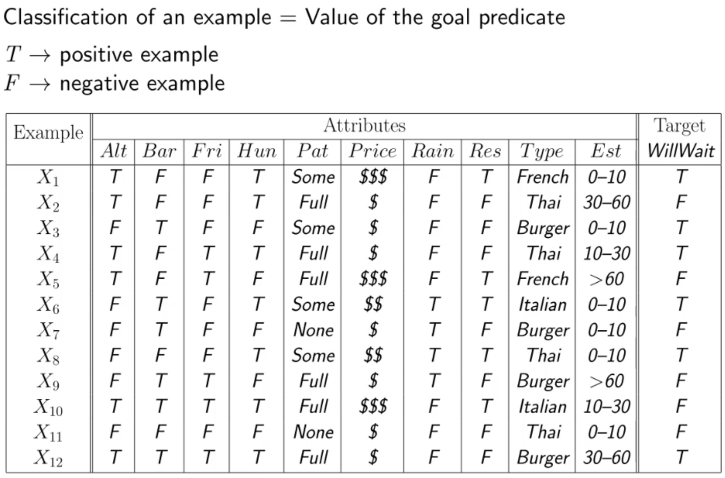
$$\forall s WillWait(s) \leftrightarrow (P_1(s) \vee P_2(s) \vee ...P_n(s)) \tag{11}$$

$P_i(s)$ 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, majority).

- Training Set example:



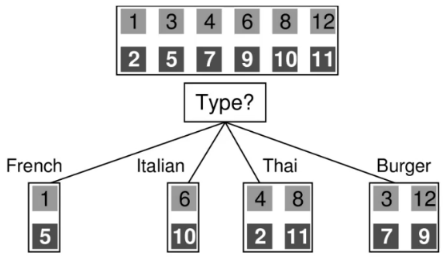
Classification of an example = Value of the goal predicate

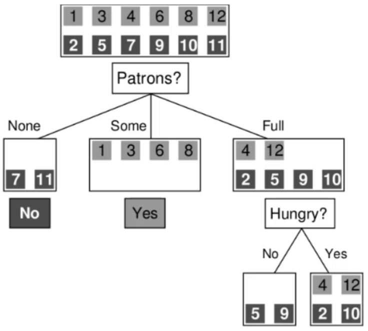
$T \rightarrow$ positive example
$F \rightarrow$ negative example

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
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.



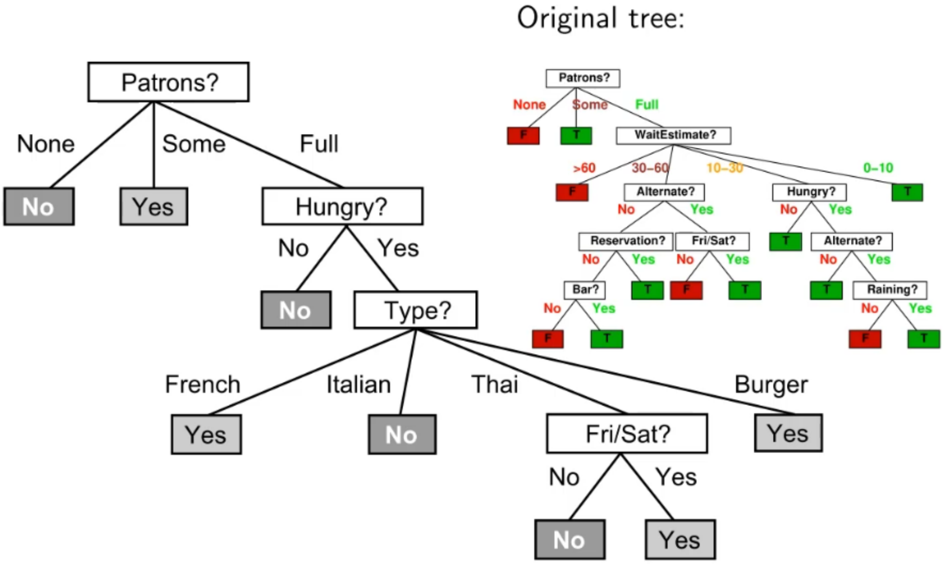


Type is a poor attribute since it leaves us with state full of true and false.
Patrons is a better choice.

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:
Patrons?
None
Some
Full
No
Yes
Hungry?
No
Yes
No
Type?
French
Italian
Thai
Burger
Yes
No
Fri/Sat?
Yes
No
Yes
No
Yes
Patrons?
None
Some
Full
F
T
WaitEstimate?
>60
30–60
10–30
0–10
F
Alternate?
Hungry?
T
No
Yes
No
Yes
Reservation?
Fri/Sat?
T
Alternate?
No
Yes
No
Yes
No
Yes
Bar?
T
F
T
T
Raining?
No
Yes
No
Yes
F
T
F
T