#university #studying #subject-3104

Teoria

Artificial Intelligence

libro: Artificial Intelligence - S. Russell, P. Norvig

Artificial Intelligence

2. Intelligent Agents

- Agente intelligente
    - razionali (-> "fare la cosa giusta" -> "ottenere il massimo grado di successo")
    - human-like
- Agente:
    - può essere definito coma una funzione $f : P^* \to A$
        - P^*: storia delle percezioni
        - A: azione da compiere
    - percepisce l'ambiente tramite sensori
    - agisce sull'ambiente tramite attuatori

Task Environment

Task Environment = PEAS = descrizione problema che risolve un agente:

- definito da:
    - Performance Measure: the notion of desiderability
        - es. safe/fast/legal
    - Environment
    - Actuators
    - Sensors
- dimensions:
    - unobservable/partially observable/fully observable
        - i sensori vedono l'intero ambiente? è necessario avere uno stato interno per tenere traccia dell'ambiente esterno? O è sufficiente utilizzare i sensori?
    - single agent/multi agent
        - multi agent = esistono altri agent (per agent si intende un oggetto il cui comportamento influenza la nostra performance)
        - multi agent può essere competitivo e cooperativo
    - deterministic/stochastic
        - se lo stato successivo dipende solo da quello attuale + azione
        - n.b. se partially observable potrebbe sembrare stocastico ma non è detto
        - stochastic = probabilità associata a stato
        - uncertain = più outcome (non c'è probabilità numerica) = causato da non deterministic oppure non fully observable

- episodic/sequential
  - Per scegliere la prossima azione è importante la storia delle percezioni?
- static/dynamic
  - Mentre l'agent pensa, l'ambiente cambia?
  - semi-dynamic: se il tempo influisce il performance score
- discrete/continuous
  - dipende dal numero di stati e da cosa percepiscono i sensori
- known/unknown
  - conosco gli stati causati dalle mie azioni?

Tipi di agenti

- Simple reflex
  - sulla base di condizioni (if su sensori) scelgo l'azione
- Model based reflex
  - memoria storico percezioni
  - predizione ambiente in base ad azione
- Goal based
  - ha un obiettivo
  - si pensa: la prossima azione come influenzerà il mio obiettivo?
- Utility based
  - si usa una funzione che associa allo stato un valore di "happiness".

Learning agent

- diviso in componenti:
  - performance element: l'agent vero e proprio (simple, model based, ecc)
  - critic: tramite i percettori restituisce la performance dell'agente
  - learning element: prende le critics e modifica il performance element per tentare di aumentare la performance
  - problem generator: individua le azioni per ottenere nuove esperienze

Agent components

- atomic
  - each state is a black box
- factored
  - each state is represented by variables or attributes
- structured
  - each state contains relationships - representation underlies a relational database

Problem-solving

3. Solving Problems by Searching

Problem components

- Initial state

- Actions: given a state, returns all the actions possibile
- Transition Model: given a state and an action, returns the result state (a.k.a. successor)
  - State Space: the set of all reachable states from the initial state
  - Search Tree: a graph with actions as branches and states as nodes
- Goal Test: determines if the a given state is a goal state.
- Path Cost: assigns a numeric cost to each path

Search tree

- frontier: set of nodes available for graph exploration
- Tree-Search: contains redundant paths
- Graph-Search: uses an "explored set" to avoid loops.

Measuring Problem-solving performance

- Types:
  - Completeness: is the algorithm guaranteed to find a solution when there is one?
  - Optimality: does the strategy find the optimal (lowest cost) solution?
  - Time complexity
  - Space complexity
- Variables
  - b "branching factor": maximum number of successors of a node
  - d "depth": shallowest/closest goal node
  - m "maximum length": maximum length of any path in the graph-search tree

Uninformed Search: we have only access to problem definition

Breath-first search

Expands the shallowest nodes first.

- Complete
- Optimal for unit step costs
- Exponential Space Complexity of $O(b^d)$ ("dominated by frontier size")

Uniform-cost search

Expands the lowest path cost node (the frontier is implemented with a priority queue).

- Optimal for any cost
- Is not necessary better than BFS:
  - it has to explore all the low-cost edges before reaching the goal.
  - it requires all the generated costs (to get the minimum)

Depth-first search

Expands the deepest node.

- Complete in the graph-search version (no cycles)
- Not Optimal

- Linear Space Complexity $O(bm)$ (where m is the deepest path)
    - we can reduce this to $O(m)$ by expanding only the successor

Depth-limited search

Expands the deepest node up to l.

- same as DFS, but stops when we reach depth l
- Not Complete
- Not Optimal
- Linear Space Complexity

Iterative deepening depth-first search

Uses Depth-limited search with l starting from 0. It increases the limit until a goal is found. It's useful when we have limited memory. We could use BFS until memory finishes. Then switch to iterative deepening depth-first search.

- Complete
- Optimal for unit step costs (as BFS)
- Linear Space Complexity
- Time complexity comparable to BFS (in terms of O).

Bidirectional Search

Starts two searches in parallel, one from the start and the other from the goal, until they meet (the frontiers touch - frontiers intersection is not null anymore). It is efficient since $b^{\frac{d}{2}} + b^{\frac{d}{2}}$ is much smaller than b^d.

- Complete
- Not Optimal (even if using BFS from both sides - can be fixed using a constant time check)
- Exponential complexity of $O(b^{\frac{d}{2}})$

Informed Search

We have access to a heuristic function that estimates the path cost of a solution.

Intuition (general Best-First Search)

- until now, we used the path cost function g. In informed search, we introduce a heuristics function h.
- $h(n)$: estimated cost of the cheapest path from the state at node n to the goal state. Compared to g, h depends only by the state of the node n.
- we start from the general Tree-Search or Graph-Search algorithm and we use a new function f instead.
- f is a function that *takes in count* (differently depending on the search strategy) the heuristics function.
- The general implementation is the same as the Uniform-Cost Search changing the path cost function from g to f.

Greedy Best-First Search

- $f(n) = h(n)$
- Complete in the graph-search version with *finite* space
- Not Optimal

- Efficient

A* Search

- $f(n) = g(n) + h(n)$
- Complete
- Optimal if some condition match:
  - h must be admissible: h must never overestimate the cost to reach the goal. h must be *optimistic*.
  - h must be consistent: the triangle inequality $h(n) \leq cost(n, a, n') + h(n')$ must be respected.
  - $h(n) \text{ is consistent } \implies h(n) \text{ is admissible}$
  - A* Tree-Search is optimal if h is admissible. Proof:
    - be G_2 an expandable sub-optimal path, and G_1 an optimal path with the node n in the frontier to reach it.
    - $f(G_2) = g(G_2) > g(G_1) \geq f(n)$
    - G_2 will not be chosen.
  - A* Graph-Search is optimal if h is consistent.
- Being a BFS, it uses a lot of space, making it unsuitable for large state space problems.

Iterative Deepening A* Search:

It's essentially iterative deepening depth-first search with the newly defined f function as depth.

Recursive Best-First Search

Similar to IDA*, but it uses less memory. IDA* stores every node to select the node with lowest f. In RBFS instead, we just store the best alternative from any ancestor to the current node.
It uses linear space, is complete and optimal.
It can *change his mind* multiple times, so time complexity is hard to define.

Simplified Memory-bounded A* Search:

Aims to improve RBFS using more space if available.
It's an hybrid between IDA*and RBFS. Uses IDA* until no more space is available to allocate a new node. Then, it drops the worst node and just saves the f value, "switching to RBFS". *Details not explained*
This is a variant of Memory-bounded A* Search, which is not explained.

4. Beyond Classical Search

Hill-Climbing Search

- sometimes called greedy local search. It takes as successor the node nearest to the goal. It doesn't track nodes using a tree, it just moves between nodes.
- Local Maxima is the problem of this algorithm: it's not the global maximum and it gets stuck

Simulated Annealing

- Aims to reduce the Local Maxima problem of Hill-Climbing Search.
- It uses a temperature variable that decreases over time (over iterations).
- The algorithm chooses randomly the successor node: if it has a better cost compared to the current, it gets chosen, otherwise it gets accepted only by a probability that depends on the temperature, allowing to

choose worse nodes at the beginning.

5. Adversarial Search

source

Games are well-defined problems that are generally interpreted as requiring intelligence to play well. Introduces uncertainty since opponents moves can not be determined in advance.

Search spaces can be very large.
For chess:

- Branching factor: 35
- Depth: 50 moves each player
- Search tree: 35100 nodes (~1040 legal positions)

Games are an instance of the general search problem.

- States where the game has ended are called terminal states.
- A utility (payoff) function determines the value of terminal states, e.g. win=+1, draw=0, lose=-1.

Types of game:

	Deterministic	**Chance**
Perfect Information	Chess, Checkers, Go, Othello	Backgammon, Monopoly
Imperfect Information	Battleships, Blind Tic-Tac-Toe	Bridge, Poker, Scrabble, Nuclear War

Minimax Algorithm

- The Minimax algorithm gives perfect play for deterministic, perfect-information games.
- In two-player games, assume one is called MAX (tries to maximize utility) and one is called MIN (tries to minimize utility).
- In the search tree, first layer is move by MAX, next layer by MIN, and alternate to terminal states.
- Each layer in the search is called a ply.

Implementation:

- Generate complete game tree down to terminal states.
- Compute utility of each node bottom up from leaves toward root.
- At each MAX node, pick the move with maximum utility.
- At each MIN node, pick the move with minimum utility (assumes opponent always acts correctly to minimize utility).
- When reach the root, optimal move is determined.

It's basically a BFS in space complexity.

```
function Minimax(state):
    if TerminalTest(state) then return Utility(state)
    A := Actions(state)
    if state is a MAX node then return Minimax(Result(state, a))
    if state is a MIN node then return Minimax(Result(state, a))
```

Alpha-Beta Pruning

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.

Look at the intuition:

```
Minimax(root)   = max(min(3, 12, 8), min(2, x, y), min(14, 5, 2))
                = max(3, min(2, x, y), 2)
                = max(3, z, 2) where z = min(2, x, y) ≤ 2
                = 3

                i.e., we don’t need to know the values of x and y!
```

The algorithm:

```
function AlphaBetaSearch(state):
    v := MaxValue(state, −∞, +∞))
    return the action in Actions(state) that has value v

function MaxValue(state, α, β):
    if TerminalTest(state) then return Utility(state)
    v := −∞
    for each action in Actions(state):
        v := max(v, MinValue(Result(state, action), α, β))
        if v ≥ β then return v
        α := max(α, v)
    return v

function MinValue(state, α, β):
    same as MaxValue but reverse the roles of α/β and min/max and −∞/+∞
```

With perfect ordering, the time complexity reduces from $O(b^m)$ to $O(b^{\frac{m}{2}})$, allowing to double the search depth.

These two algorithms are complete and optimal (if opponent makes optimal decisions), but aren't suitable for real games where states are too many.

Imperfect Algorithms

Previous algorithm still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical. In this chapter we apply a heuristic evaluation function to states in the search, effectively turning nonterminal nodes into terminal leaves.

H-Minimax Algorithm

It's like Minimax, but there are two replacements:

- `TerminalTest` function with `CutOffTest` function
  - the newly added `CutOffTest` function has access to the current node's height.
- `Utility` function with `Eval` function

Evaluation and Utility Functions

Weighted Linear Evaluation Functions

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

This assumes all the features are independent of each other, which is usually not true.
We want more sophisticated cutoff tests.

Quiescient positions

- We only cut off search in quiescient positions.
- i.e. in position that are "stable", unlikely to exhibit wild swings in value
- non-quiescient positions should be expanded further

Horizon effect

another problem - the horizon effect:

- if a bad position is unavoidable (e.g., loss of a piece in chess), but the system can delay it from happening, it might push the bad position “over the horizon”
- in the end, the resulting delayed position might be even worse

Monte Carlo Tree Search

#todo

Algorithms in non-deterministic games

#todo

6. Constraint Satisfaction Problem

Until now we assumed all the states were black boxes.
Now states are factored, so if we're able to formulate a problem as CSP we can get more efficient solutions.
A constraint satisfaction problem consists of:

- X: a set of variables, $\{X_1, \dots, X_n\}$.
- D: a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.
  - D_i is a set of allowable values $\{v_1, \dots, v_k\}$ for X_i.
- C: a set of constraints that specify allowable combinations of values.
  - C_i is a pair $\langle \text{scope}, \text{relation} \rangle$
    - scope: tuple of variables (X_i) that partecipate in the constraint
    - relation: set of values (each value is a tuple of D_i elements) that variables in $scope$ can take on.

In a given state, we can set some values to one of more variables. Those are called **assignments**.

- An assignment is **consistent** if it doesn't violate any constraint.
- An assignment is **complete** if it's consistent and every variable is assigned.
- An assignment is **partial** if it's consistent and at least one variable is assigned.

We can represent a CSP with a **Constraint Chart**:

- Nodes are the variables of the problem X.
- An edge between two nodes exists if there's at least a constraint C that includes both of the nodes as scope.

N-Queens example

$$\begin{aligned} &X = \{Q_1, Q_2, \dots, Q_n\} \\ &D = \{1, 2, \dots, n\} \quad \text{technically not respecting formal definition} \\ &C = \{Q_i \neq Q_k, \quad |Q_i - Q_k| \neq |i - k|\} \quad \text{same} \end{aligned}$$

Implementation with Search

We can solve any CSP using Search. The entities are defined as follows:

- Initial state: all the variables are unassigned.
- Actions: assign a value to a variable
- Goal function: the variables assignment is **complete**.

In each state, we store:

- The UNASSIGNED list: all the variables not assigned.
- The ASSIGNED list: all the variables assigned.
- Additionally, for every variable:
  - NAME: *not really useful for the algorithm itself*
  - DOMAIN: D_i
  - VALUE: the current value, if present.

Constraints could be represented as sets or as functions that check for validity.

We can use DFS as search algorithm.

- $m = n$ (number of variables)
- $d = n$ (number of variables)
- $b = \sum_i |D_i|$

Constraints could be redundant. We can apply a series of reductions to *simplify* them.

Consistency

These can be used to reduce the amount of legal values and could potentially find the solution directly.

- Node Consistency:
  - we consider the unary constraints.

    - this is straightforward: we iterate all the variables and exclude the values in the domain that don't match the unary constraints.
- Arc Consistency:
    - we consider the binary constraints
    - we say X_i is arc-consistent to X_j if for every value D_i there is some value in D_j that satisfies the binary constraint on the arc (X_i, X_j)
    - A real implementation algorithm is called AC-3
- Path Consistency
    - we consider the ternary constraints
    - "A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$"
    - A real implementation algorithm is called PC-2
- K-Consistency
    - it's the generalisation of the previous reductions
        - $K = 1$ -> Node
        - $K = 2$ -> Arc
        - $K = 3$ -> Path

If we impose K-Consistency up to $K = |X|$, we just get the solution, but this is exponentially expensive.

Backtracking Search

We use DFS with fixed assignment order: variable assignment is a commutative operation, sorting ensures getting a faster algorithm.
The algorithm is recursive and it does the following: iterates all values of all variables in X and searches using the following criteria:

- It returns failure if the state is inconsistent (to let the caller iterate the other values)
- It continues until the values are finished (there's no solution) or the assignment is complete (the problem has been solved).
- For each variable it imposes a certain level of consistency, depending on the problem
    - note that we do this **before** performing the search.

If we need to just find a solution, we can apply some ordering tweaks to improve performance.
Of course, if we need all the solutions, ordering doesn't matter: we have to explore the whole graph.

Variables (X) Order

How do we order the iterated variables?

- Minimum Remaining Values (MRV) Heuristics:
    - The variables chosen first are the ones with fewer legal values
    - a.k.a. "fail-first" heuristic -> if we find a variable fails, we can avoid checking other permutations on the other variables.
    - This makes no sense if the variables have the same amount of legal values (e.g. N-Queens example).
- Degree Heuristics:
  - The variables chosen first are the ones involved in the largest number of constraints.

We could combine both of the heuristics: default to MRV and switch to Degree Heuristics in case of legal values amount equality.

Domain D_i values Order

We then need an heuristic to iterate the values in the domain of the selected variable in a more efficient manner.

- Least Constraining Value:
    - we apply the same (but opposite) logic of MRV. We choose the values that rule out the fewest amount of choices for the neighboring variables in the constraint graph.

Forwarding Checking

Inference: imposing a level of consistency to a variable.
Whenever a variable is assigned, we can perform a simple form of *inference* called Forward Checking: applying arc-consistency for it. Only useful if we don't already do it in the preprocessing step
#todo non lo faccio già? qual è il preprocessing? forse qui possiamo applicare un livello di consistenza più alto e quindi più costoso

Local Search

Instead of iterating all the values in all the variables, we select a random variable that violates some constraint. This algorithm can use an heuristics called min-conflicts which selects the variable that violates the minimum amount of constraint conflicts.

This works very well for big problems, because it doesn't depend by the problem size.

Knowledge, reasoning, and planning

7. Logical Agents

Instead of working with states and having `code` to define the actions, now we use logic.
We use a Knowledge Base (KB), which is a set of sentences.
The knowledge base contains sentences that can either be axioms (taken for granted) or derived (from other sentences).
It's possibile to interact with the knowledge base with two operations:

- `TELL` : a way to add a sentence
- `ASK` : a way to query what is known.
  The *derivation* is a way of *inference*, and is hidden behind the `TELL` and `ASK` operations.

An agent maintains a knowledge base, which at start may contain some background knowledge.
Each time an agent is called, it performs the following operations:

1. `MAKE-PERCEPT-SENTENCE` : it uses `TELL` and it constructs a sentence asserting that the agent perceived the given percept in a given time
2. `MAKE-ACTION-QUERY` : it `ASK` s what action should be done at the current time
3. `MAKE-ACTION-SENTENCE` : constructs a sentence asserting that the chosen action was executed

```
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
```

This approach is *declarative* (not *procedural* as we used before). We use the **knowledge level** just by specifying the desired goal, when querying we don't know how the **implementation level** works, since it's independent.

*details about logic and propositional logic already covered in **Strutture Discrete** and **Fondamenti di Informatica** subjects*.

How can we prove a sentence α is true in some KB?
We can either:

1. Iterate the truth table
2. Generate the derivation sequence

Language	Ontological Commitment (what exists in the world)	Epistemological Commitment (what an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief ∈ [0, 1]
Fuzzy logic	facts with degree of truth ∈ [0, 1]	known interval value

8. First-Order Logic

This kind of logic allows us to represent the world in an *easier way*. Propositional Logic isn't suitable to represent knowledge of complex environments. First-Order Logic is sufficiently expressive to represent a good deal of our commonsense knowledge.

Base Elements

- Constant symbols: objects (e.g. `Richard` or `John`)
- Predicate symbols: relations (e.g. `Person`, `King` or `Crown`)
- Function symbols: functions (e.g. `Brother`, `AgeOf`, `SquareRoot`)
- Variables: $a, x, s, \dots$
- Connectors: $\vee, \wedge, \neg, \implies, \iff$
- Quantifier: $\forall \text{ (Universal Quantification)}, \exists \text{ (Existential Quantification)}$
- Equality: $=$

Difference between predicate and function symbols:

- functions *return* a value, predicates are relations among values in a tuple of objects.
- $Red(x)$ is a predicate since it just asserts x is red.
- $FatherOf(x)$ is a function since it returns the father of x.
    - $IsFatherOf(x, y)$ could be a predicate if it meant "x is the father of y"

General Rules for Exercises

- $\forall x \forall y = \forall y \forall x$
- $\exists x \exists y = \exists y \exists x$
- $\exists x \forall y \neq \forall y \exists x$
- $(\forall x) f(x) = (\neg \exists x) \neg f(x)$
- $(\exists x) f(x) = (\neg \forall x) \neg f(x)$

Universal Elimination

We remember the **Modus Ponens** inference rule: $\frac{\alpha, \alpha \implies \beta}{\beta}$
The **Universal Elimination** is useful when we're dealing with quantifiers: $\frac{(\forall x)\alpha}{\alpha\{x/\tau\}}$
Of course, τ must be a sentence (and not a variable)
We call $\{x/t\}$ **unifier** and is often referred to as σ.
This uses the concept of **substitution**.

We need something to understand that $Knows(John, x) = Knows(John, Jane)$.
We need a process that finds $\{x/Jane\}$.
We call this process **unification** - it's a function that returns a unifier, if such exists:

$$\text{UNIFY}(Knows(John, x),\ Knows(John, Jane)) = \{x/Jane\}$$

Generalized Modus Ponens

Basically it's modus ponens with the unification.
#todo

9. Inference in First-Order Logic

We have our KB. In practice, how do we *apply* inference?

Forward Chaining Algorithm

It's called "Forward" since it gets executed whenever a new fact is added to the KB.
It iterates all the rules in the Knowledge Base and for each of them it checks if there are new inferences that can be done. If so, the algorithm continues, until there are no more.
This is like brute force.
Let's analyze the algorithm:

- Every inference is an application of Generalized Modus Ponens
- It is complete if the knowledge base has definite clauses.
- The naive implementation is highly inefficient but it can be improved considerably.

Backward Chaining Algorithm

It's called "Backward" since it gets executed whenever a new query is performed on the KB.
It starts from the goal and reaches the axioms *backwards*.

We perform basically a DFS.

- We look for rules such $lhs \implies \text{goal}$
- We generate the unifiers and proceed recursively

Resolution

Conversion to CNF

> **Theorem**
> Every sentence of first-order logic can be converted into an **inferentially** equivalent CNF sentence
> *notice the "inferentially"*

These are the steps:

1. Eliminate the implications
   - Remember that $x \implies y$ is the same as $\neg x \lor y$
2. Move $\neg$ inwards
   - For example $\neg \forall x\ p$ becomes $\exists x\ \neg p$
   - We can use De Morgan rules as well
3. Standardize variables
   - If we have multiple quantifiers, we make sure that each variable has a unique name (useful for following steps)
4. Skolemize
   - With this step we are able to maintain only universal quantifiers ($\forall$), deleting the existential ones ($\exists$).
   - We first move the quantifiers all to the left, respecting their original order.
   - Then, we have to remove the $\exists$ symbols.
     - Let's take this example: $\forall x \exists y P(x, y)$.
     - This is valid for a generic value that depends on y
     - So we can write this as $\forall x P(x, f(x))$
     - In general: what to substitute depends on the amount of quantifiers the $\exists$ symbol has on its left. If it has none, we can replace it with an arbitrary constant a (called Skolem Constant). Otherwise we replace it with a function that has as many parameters as many quantifiers it has on its left.
5. Drop universal quantifiers
6. Distribute $\land$ over $\lor$
   - We use the distribution property

Proof

Now, we have to prove a sentence α.
Resolution proves that $\text{KB} \vDash \alpha$ by proving $\text{KB} \land \neg\alpha$ unsatisfiable, that is, by deriving the empty clause.
We assume the knowledge is already in CNF. If not, we use the previous algorithm.

1. We negate α.
2. We transform it to CNF
3. We add it to the knowledge base
4. We apply inference and look for a contradiction.

Uncertain knowledge and reasoning

13. Quantifying Uncertainty

*probability already covered in **Strutture Discrete** subject*.

Syntax

We can have random variables. We could write something like:

```
P (Weather = sunny) = 0.6
P (Weather = rain) = 0.1
P (Weather = cloudy) = 0.29
P (Weather = snow) = 0.01
```

or as an abbreviation: `P(Weather) = <0.6, 0.1, 0.29, 0.01>` (assuming a predefined ordering: `<sunny, rain, cloudy, snow>`)
Values must me normalised (the sum must be equal to one).

It's also possible to define the probability of multiple variables together.
It's a table called **Joint Probability Distribution**.
In the following example `W` is weather and `C` is cavity.

```
P (W = sunny ∧ C = true) = P (W = sunny|C = true) P (C = true)
P (W = rain ∧ C = true) = P (W = rain|C = true) P (C = true)
P (W = cloudy ∧ C = true) = P (W = cloudy|C = true) P (C = true)
P (W = snow ∧ C = true) = P (W = snow |C = true) P (C = true)
P (W = sunny ∧ C = false) = P (W = sunny|C = false) P (C = false)
P (W = rain ∧ C = false) = P (W = rain|C = false) P (C = false)
P (W = cloudy ∧ C = false) = P (W = cloudy|C = false) P (C = false)
P (W = snow ∧ C = false) = P (W = snow |C = false) P (C = false)
```

Independence

Some variables are independent on others. for example
`P (cloudy | toothache, catch, cavity)` can be written as `P (cloudy)`

The property used is called independence.
Independence between propositions can be written as follows:
`P(a | b) = P (a)`, `P(b | a) = P(b)` or `P(a ∧ b) = P(a)P(b)`.

This is useful because it can reduce the number of independent cases.

Conditional Independence

#todo

16. Making Simple Decisions

A knowledge base is **not-monotonous** if logic deductions could change due to new facts.

Utility theory: used to represent preferences
Decisions theory: utility theory + probability theory

Games theory: decisions theory against players whose decisions minimize our utility theory

#todo

Learning

18. Learning From Examples

f the function to predict.
h the function we build to predict f

training set: a set of examples in the form $(x, f(x))$
h is consistent if $f(x) = h(x)$ for every example point we have in the training set.

Curve fitting: take the *simplest* consistent function (Ockham's razor)

Examples are:

- **described** by attributes (boolean)
- **classified** by an outcome

Decision Tree Learning

Each node is an attribute and each branch is a possible value.
In case of booleans, the tree is a binary tree.
Goal: find the smallest consistent tree.

The idea is to choose recursively the most significant attribute (the one that *splits better* the examples by outcome - in the case of booleans).
Let's define formally the definition of *split better*

Entropy of a variable V of v_k values:

$$H(V) = \sum_k -P(v_k) \log_2 P(v_k)$$

If a variable is boolean the entropy becomes as follows:

$$B(q) = \sum_k -(q - \log_2 q + (1-q) \log_2 (1-q))$$

To calculate the entropy of a training set T:

$$H(T) = B\left(\frac{p}{p+n}\right)$$

axx: If want to know how much a single attributes takes part in that $H(T)$. We can do it by excluding it and calculating the entropy on the remaining.

$$Remainder(A) = \sum_{k=1}^{v} \frac{p_k + n_k}{p+n} B\left(\frac{p_i}{p_i + n_i}\right)$$

Information Gain:

$$Gain(A) = B\left(\frac{p}{p+n}\right) - Remainder(A)$$

We can choose the attribute with highest $Gain$.

Performance Measure

How do we know $h \approx f$?

A bigger training set implies a better prediction.
The learning curve is the percentage of correctness over the training set size.

A function h is said to be **Probably Approximately Correct** if it's *consistent with a sufficiently large set of training examples, thus being unlikely to be seriously wrong*.
This doesn't not apply for each algorithm. If it does, the algorithm is called PAC Learning algorithm.

A PAC Learning Algorithm takes as input a set of examples and two variables ϵ and δ and it produces a function that with a probability of $1-\delta$ satisfies $P[h(x) \neq f(x)] < \epsilon$.

in other words

- A function h is probably approximately correct with accuracy $\epsilon \iff P[h(x) \text{ is correct}] > 1-\epsilon$.
- A function h is probably approximately correct with confidence $1-\delta \iff P[h \text{ is not AC}] < \epsilon$.

#todo m