Agents usually commit errors, which are caused by either laziness (failure to enumerate exceptions, qualifications, etc.) or ignorance (lack of relevant facts, initial conditions, etc.). We thus need to quantify the level of **uncertainty** they produce in their decision. To do it, we use **probability theory**, as it offers a clean way to quantify likelihood. Probability is derived from statistical data and innate knowledge, it is thus not an absolute truth, but it's *strictly related to the state of knowledge*.
The final decision is then taken considering **utility theory**, which basically finds a compromise between potential benefit and potential loss of each option. *An agent is considered rational if and only if it chooses the action that yields the maximum expected utility, averaged over all the possible outcomes of the action.*

In probability theory and statistics, **normalization** refers to adjusting values or distributions so that the total sums to a specific value, typically 1. This is crucial when working with probability distributions, as it ensures the probabilities are valid.
The process can be computationally expensive as it involves summing over all hidden variables, which might grow exponentially with the number of propositions. Therefore, when dealing with large problems, *it is common to work with non-normalized distributions until the end, where normalization is applied in a single step*.
When computing the unnormalized probabilities, the sum of probabilities might not equal 1. This happens because intermediate calculations (e.g., marginalizing over hidden variables) can yield values that are proportional to the true probabilities but not properly scaled.
To adjust these unnormalized values into valid probabilities, we introduce α, which rescales the results so that the sum equals 1. α is given by 1 divided by the total probability accumulated by the sums, which is usually not equal to 1. After obtaining α it has to be multiplied by the probabilities accumulated to normalize them.

Conditional independence is a key concept in probability theory and statistics. It describes a situation where two events (or variables) are independent of each other given the occurrence of a third event (or variable). Formally, for three variables X, Y, and Z, we say that *X and Y are conditionally independent given Z* if knowing Z makes X and Y independent:

- Without Z: X and Y may be dependent, meaning knowing X gives us information about Y, and vice versa
- With Z: Once Z is known, X and Y become independent, meaning that knowing X no longer gives us information about Y, because Z explains all the dependence between X and Y

For example, without any other information, rain and traffic are dependent: if it rains, traffic might be worse. But once we know the weather forecast (Z), the rain and traffic become conditionally independent. This is because the weather forecast explains the likelihood of both rain and traffic.
Conditional independence is foundational to probabilistic graphical models like Bayesian networks, where dependencies between variables are represented as directed edges in a graph. *It simplifies how we model and compute joint distributions in such networks*.

Bayes' rule states that *the probability of the cause given the effect is equal to the probability of the effect given the cause multiplied by the probability of the cause and divided by the probability of the effect:*

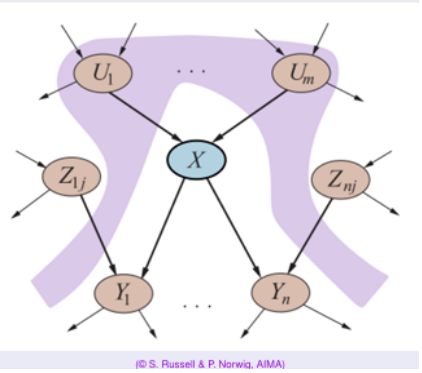
$$P(cause|effect) = \frac{P(effect|cause)P(cause)}{P(effect)}$$

This allows the computation of the inverse dependency between cause and effect and it's particularly useful considering that in most real world situations the probability of the effect given the cause is known, while vice-versa is unknown.
Given this rule, a **naive Bayes model** is a probability model that assumes that effects are conditionally independent once the cause is given.

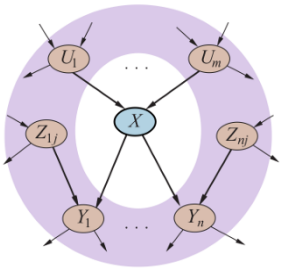
A **Bayesian network** is a **D**irected **A**cyclic **G**raph where each node represents a random variable and where pairs of nodes are connected via directed arcs that start from the parent and go to the children. *The connection represents a conditional distribution of the probability of children given parents' values combinations*. Bayesian networks allow for compact specification of full joint distributions, with all the dependencies and non-dependencies explicitly represented.

A **CPT** for a Boolean X with k Boolean parents has 2^k rows for the combinations of parent values, where each row requires one number. If each variable has no more than k parents, the complete network requires $O(n \cdot 2^k)$ numbers, while a full joint distribution requires $2^n - 1$ numbers.

Global semantics defines the full joint distribution as the product of the local conditional distributions: $P(X_1,...,X_N) = \Pi_{i=1} P(X_i|parents(X_i))$ so if X_i has no parents, then the conditional distributions reduce to prior probability $P(X_i)$.



Local semantics states that each node is conditionally independent of its nondescendants (including ancestors) given its parents:
$P(X|U_1,...,U_m, Z_1,...,Z_{nj}) = P(X|U_1,...,U_m)$ for each X.
This property holds if and only if Global semantics holds.



In a Bayesian Network, each node is conditionally independent of all the other nodes, given its **Markov blanket**, which is the union of parents, children and the other parents of its children:
$P(X|U_1,...,U_m, Z_1,...,Z_{nj}, W_1,...,W_k) = P(X|U_1,...,U_m, Y_1,...,Y_n, Z_{1j},...,Z_{nj})$
This is because in the mathematical sense sometimes it's possible to extract from the effect some information about the cause.

Given a set of random variables, to build a Bayesian Network, the first step is to choose an ordering following causality to identify parents, children and connection. Then, for each element, add it to the network and choose a subset as parents. This will guarantee the global semantics by construction. Next, the **C**onditional **P**robability **T**able (**CPT**) must be defined for each node in the Bayesian Network. *The CPT expresses the conditional distribution of the node given its parents*.
The challenge is that these tables can grow exponentially with the number of parent nodes. In such cases, simplifications can be applied, like using a **Noisy-OR distribution**, which assumes that *causes are independent unless stated otherwise*. In a Noisy-OR distribution, we assume that each parent node represents a potential cause of the effect, and the absence of all causes results in a 100% chance that the effect will not occur. The probability of failure is modeled for each parent independently, simplifying the number of parameters that need to be calculated. This linearizes the complexity of the network, making it computationally more efficient to represent certain relationships.

Once the Bayesian Network is constructed and the conditional distributions are specified, we can perform inference using methods like Enumeration. **Inference by Enumeration** computes the posterior probability $P(X|e)$, which involves summing over all possible values of the hidden variables. The process uses the structure of the network to *factorize joint distributions and simplify algebraic expressions*, though it can still be computationally expensive, especially when many variables are involved. **Recursive depth-first enumeration** is commonly used,

requiring $O(n)$ space and $O(2^n)$ time, leading to potential *inefficiency due to repeated computations*. This can be mitigated by *removing irrelevant variables*, which are additional summations whose probability is equal to 1 and can thus be neglected. This brings us to the concept of Variable Elimination, which is an optimization technique for inference in Bayesian Networks. Variable elimination aims to improve efficiency by carrying out summations in a specific order and storing intermediate results to avoid redundant computations.

The key idea behind variable elimination is to perform summations from right to left (or bottom-up if visualized as a tree) and store the intermediate results, called factors. This approach allows us to reuse these factors instead of recalculating them multiple times.

Let's consider an example to illustrate this process. Suppose we want to calculate ***P(B | j, m)***, where ***B*** represents Burglary, ***j*** represents JohnCalls, and ***m*** represents MaryCalls. The calculation can be broken down into a series of factor operations:

1. We start with the joint probability distribution and apply Bayes' rule.
2. We then factor this expression into smaller components, each representing a conditional probability from our Bayesian Network.
3. These factors are combined using pointwise product operations and summed over variables we want to eliminate.

The process involves creating intermediate factors (f1, f2, etc.) that represent portions of our calculation. By systematically eliminating variables (summing them out), we can efficiently compute the final probability.

It's important to note that the efficiency of variable elimination depends significantly on the order in which we choose to eliminate variables. A good ordering can dramatically reduce the computational complexity of the inference task.

There are two key operations in variable elimination:

1. Factor summation: This is essentially standard matrix addition, where we sum corresponding elements of factors with the same variables.
2. Pointwise product: This operation multiplies the elements of factors for the same variable values. It's crucial to understand that the resulting factor will have as its variables the union of the variables from the input factors.

To further optimize the variable elimination process, we can employ two additional techniques:

1. Factoring out constant terms: If certain factors don't depend on the variable being summed over, we can move them outside of the summation. This can significantly reduce the number of operations needed.
2. Removing irrelevant variables: We can eliminate variables that don't contribute to the final probability we're calculating. This is related to the concept of d-separation in Bayesian Networks.

While variable elimination is more efficient than simple enumeration, it's important to understand that exact inference in Bayesian Networks is still computationally challenging. In fact, it has been proven to be NP-Hard, which means that for large, complex networks, approximate inference methods may be necessary.

To illustrate the complexity, we can consider how a **Boolean Satisfiability** (**SAT**) problem can be reduced to an inference problem in a Bayesian Network. This reduction shows that

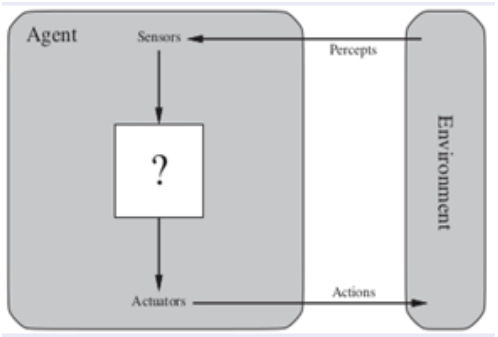
determining whether a Boolean formula is satisfiable is equivalent to calculating a certain probability in a corresponding Bayesian Network.

In conclusion, while techniques like variable elimination can significantly improve the efficiency of inference in Bayesian Networks, the fundamental complexity of the problem remains. Understanding these methods and their limitations is crucial for effectively working with probabilistic models in artificial intelligence.

function ELIMINATION-ASK(X, **e**, bn) **returns** a distribution over X
  inputs: X, the query variable
    e, observed values for variables **E**
    bn, a Bayesian network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$

  $factors \leftarrow [\,]$
  for each var **in** ORDER(bn.VARS) **do**
    $factors \leftarrow [\text{MAKE-FACTOR}(var, \mathbf{e}) | factors]$
    if var is a hidden variable **then** $factors \leftarrow \text{SUM-OUT}(var, factors)$
  return NORMALIZE(POINTWISE-PRODUCT($factors$))

—----------



An **agent** is an entity that perceives the environment through sensors and acts accordingly by using actuators. We define a **percept** as the *collection of agent's perceptual inputs at any given instant* and the **percept sequence** as the *complete history of everything the agent has ever perceived*, including, sometimes, his own actions.
An agent behavior is described by an **agent function** $f: P^* \rightarrow A$ which *maps any given percept sequence into action* (it's an abstract mathematical description).
Internally, the agent function for an AI agent is implemented by an **agent program**, which is the concrete implementation that can physically run on the hardware.
A **rational agent** is an agent that is able to elaborate the most "successful" actions given a defined percept sequence. The sequence of actions causes the environment and the agent to go through a sequence of states. It's thus necessary to find a way to evaluate those states.
To decide which sequence of actions to take, a rational agent bases its response on the performance measure, which defines the criterion of success, the agent's prior knowledge of the environment, the feasible actions that the agents can actually perform and the agent's percept sequence up to date. To sum up: *for each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.*
Other important features are the **information gathering & exploration** abilities to grasp all essential data needed before elaborating an action, the **learning** ability to recognize patterns and perfect the action elaboration, and the **autonomy** capability to be able to rely on itself where prior knowledge is insufficient.
The **task environment** of a rational agent can be described in terms of **P**erformance measure (safety, destination, profits, comfort, …), **E**nvironment (streets, pedestrians, obstacles, buildings, …), **A**ctuators (steering, accelerator, brake, speaker, display, …), **S**ensors (video, sonar, lidar, GPS, gyroscope, …) (**PEAS**).
Task environments can be categorized along six dimensions (underlined conditions are easier to manage, but most real life scenarios present all non-underlined features):

1) Single-agent vs. multi-agent (other agents depend on the current agent's actions. It can be *competitive* or *cooperative*)
2) Fully observable vs. partially observable / unobservable
3) Deterministic vs. stochastic (if the next state is not completely determined by the current state and by the action of the agent) vs nondeterministic (possible outcomes are not attached to probabilities)
4) Episodic vs. sequential (current decisions can affect future decisions)
5) Static vs. dynamic (environment can change while the agent is choosing the action) vs semi dynamic (if the environment doesn't change, but the agent's performance score changes with time)
6) Discrete vs. continuous (time is treated as a continuous flow)

Another important factor is the *state of knowledge about the rules of the environment*.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

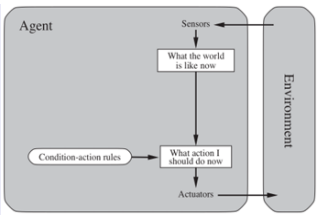
An agent is defined by a **Program**, which runs on some **Architecture**, which is a *computing device with physical sensors and actuators*. The agent takes inputs, and elaborates them into output by using the program. While the agent function takes the *entire percept history* as input, the agent program only takes the *current percept*.

```
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
```

Agent programs can be divided into 4 different basic kinds:

1) Simple Reflex Agents: *implemented through condition-action rules, it selects the next action only on the basis of the current percept, making it implementable in Boolean circuits, but lacking precision due to its memoryless.*
   It's very fast in reactions, but it may work only if the environment is fully observable and may produce errors.



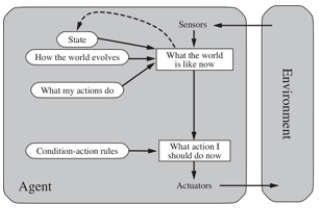

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

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

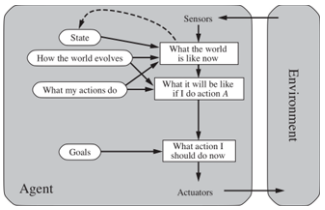
```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              model, a description of how the next state depends on current state and action
              rules, a set of condition–action rules
              action,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

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

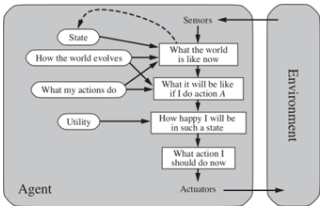
2) Model-based Reflex Agents: *it keeps track of the part of the world which is not observable at the moment, maintaining an internal state which depends on the percept history, in a human-like behavior. To update its internal state the agent needs a model of the world, describing how the world evolves and how the agent can affect it through its actions.*



3) Goal-based Agents: *It combines the goal information, which describes the desirable situation that should be achieved, with the model of the world to choose its actions. It takes the future into account, thus making the system difficult to manage if long sequences of actions are needed to reach the goal. This solution is often used in search/planning.*

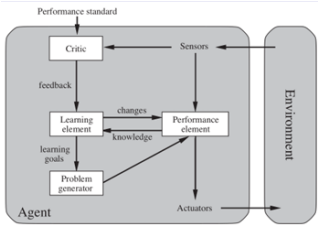


4) Utility-based Agents: *It adds one or more utility functions that internalize a performance measure to perfect the path to the final goal. Generally to maximize efficiency or minimize risks. It can manage conflicting goals, but it's difficult to implement and to run.*



Since programming by hand is extremely inefficient and ineffective for these kinds of programs, a useful solution is to build **learning** machines and then train them.

This approach ensures stiffness of the agent program toward *initially unknown environments*. The **learning element** elaborates new rules based on the experience. The **performance element** selects actions based on percepts and tells the agent how it is doing with respect to a standard via the **critic**, thus elaborating *improvements*. The **problem generator** can *force exploration by suggesting actions that will lead to new and informative experiences*.



At this point it's necessary to define the notion of state and how to transition from one to another. Three main ways are used to represent states and transitions, with increasing expressive power, computational complexity, and at three decreasing levels of abstraction:

1) Atomic: *a state is a black box with no internal structure, where each state of the world is indivisible and it's one among a discrete collection of values* (used in search & games, hidden Markov models, Markov decision processes)
2) Factored: *a state consists of a vector of attribute values, where each state is a combination of values and can represent uncertainty* (used in constraint satisfaction & propositional logic, planning, Bayesian networks, machine learning)
3) Structured: *a state includes objects, each of which may have attributes of its own as well as relationships to other objects and can represent reality in details* (used in relational databases, first-order logic, first-order probability models, knowledge based learning, natural language understanding)

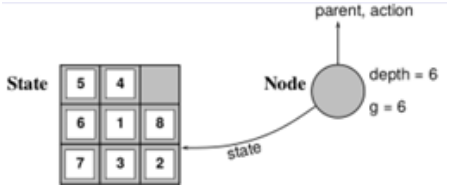
To perform problem solving, AI usually formulates the problem as a **search problem** among a state space. The successful state is set as the goal, middle states are represented as elements of the problem and the transition between states is performed after a sequence of applicable legal actions is elaborated by eventually considering the path cost. Search happens inside the agent, which only uses the description of what to achieve without having an

algorithm to solve it. This could be very demanding, and can significantly benefit from prior heuristic knowledge.

```
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
```

In the easiest search problems states are represented as atomic elements, the environment is fully observable, discrete, known and deterministic, so the agent can ignore its percept while performing an action. At each step, the agent chooses which state to **expand**, meaning the application of operators that will generate new states. The final representation of all states starting from the initial state is a **search tree/DAG**, where the leaves are states that are still to be expanded or dead ends. The search strategy defines the criteria to decide which leaf to expand, choosing among the set of current leaves, called frontier/fringe. The search strategy can be either uninformed if it doesn't use any domain knowledge or informed, if it applies rules following heuristics driven by domain knowledge It's necessary to detect already traveled paths in order to avoid infinite loops or exponentially large paths. To do so, usually a hash table is used as data structure to remember all previously expanded nodes.
Since real world elements and actions are very complex, information can be abstracted to remove irrelevant details and synthesize useful details.
In practice, states are representations of physical configurations and nodes are data structures that constitute parts of a search tree, including state, parent, action, path cost function.



Common queue implementation of the fringe include:

- First In First Out (FIFO) -> O(1)
- Last In First Out (LIFO) -> O(1)
- Best First Out -> O(log(n))

and they are based on the primitives ISEMPTY(), POP(), INSERT()
Implementation of the Explored set is done via hash table -> O(1) and is based on primitives ISTHERE(), INSERT().

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE(node)) then return node
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
      s ← a new NODE
      PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
  return successors
```

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE[node]) then return node
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERTALL(EXPAND(node, problem), fringe)
  end
```

In practice, since the amount of data a graph requires to store is unmanageable, **cache search** is used as an intermediate solution.

To define the goodness of the search, the main parameters that are evaluated are **completeness** (does it find a solution when it exists?), **time complexity** (how many steps are needed to find a solution?), **space complexity** (how much memory is needed?) and **optimality** (does it find a least-cost solution?), where time and space complexity are evaluated considering the maximum branching factor of the search tree (b), the depth of the least cost function (d), and the maximum depth of the state space (m).
Most common **uninformed search strategies** include:

- **Breadth-first search:** expand the oldest unexpanded node, treating the frontier as a FIFO queue. The goal test is applied to each node as soon as it's "discovered", not when it's selected, thus detecting it one layer early. This strategy is characterized by $O(b^d)$ time complexity and $O(b^d)$ memory size requirement (which is a lot); it's also complete if the solution exists, and the state space is finite and if all the costs are equal it is also optimal.

```
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:** expand the node with lowest path cost, treating the frontier as a priority queue. The goal test is applied to a node when it's selected for expansion, replacing a node in the frontier with the same state from another node if it's obtained with a better path cost. This strategy is characterized by $O(b^{1+[C^*/\epsilon]})$ with $1 + [C * /\epsilon]$ being the "effective depth", that is the *Cost of the cheapest solution divided by the minimum arc cost*; time complexity and $O(b^{1+[C^*/\epsilon]})$ memory size requirement (which is a lot); it's also complete if the solution exists, and the state space is finite and it is also optimal.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

- **Depth-first search:** expand the deepest unexpanded node, treating the frontier as a LIFO queue. The goal test is applied to each node as soon as it's "discovered", not when it's selected, thus detecting it one layer early. This strategy is characterized by

$O(b^m)$ time complexity and $O(bm)$ memory size requirement (with m being the maximum depth); it's also complete if the space is finite, the solution exists and if loops are prevented for tree version, and the state space is finite, but it is not optimal. A variant approach is **Backtracking search**, to "discover" only one successor at a time, keeping track of the eventual next successor. This approach requires $O(m)$ memory.

```
function ~~BREADTH~~-FIRST-SEARCH(problem) returns a solution, or failure
                  Depth
  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
                LIFO
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the ~~shallowest~~ node in frontier */
                                  deepest
    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)
```

- **Depth-limited search:** It behaves like DFS, but with a lower limit 'l', once that limit is reached, search restarts from the top, guaranteeing $O(b^l)$ at the cost of incompleteness. This approach can be expanded as **Iterative-deepening search** by iteratively increasing the limit, combining the benefits of BFS and DFS (backtracking version). This strategy is characterized by $\sim O(b^d)$ time complexity and $O(bd)$ memory size requirement; it's also complete, and if all the costs are equal it is also optimal.

```
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

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
```

- **Bidirectional search** employs two simultaneous searches (one forward from the start node and one backward from the goal node) to optimize some aspects, with a time and memory complexity of $\sim O(2b^{d/2})$, so you find the solution one step ahead, by keeping two smaller frontiers instead of a very big one. This is a common approach although backward search is a bit tricky.

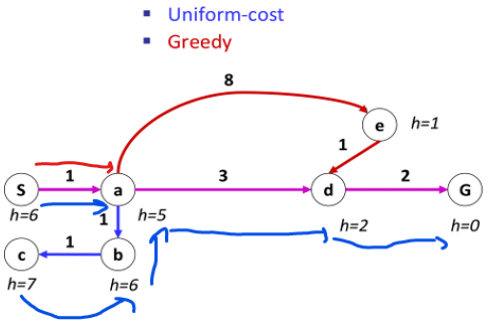
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon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

graph search | tree search

[a]: complete if b is finite
[b]: complete if step costs $\geq \epsilon$ for some positive ϵ
[c]: optimal if step costs are all identical
[d]: if both directions use breadth-first search

An intelligent system can exploit knowledge about the problem to reduce the combinatorial explosion. It uses its control over the most probable situation to direct the search.
We define **best-first search** as a search strategy where the node to expand is selected based on an evaluation function $f(n)$, which often includes a **heuristic function**, that estimates the cost of the cheapest path from the state at a node n to a goal state. Some properties of heuristic functions are that $h(n) \geq 0 \quad \forall n, h(G) = 0$ with G being a goal, it exploits extra domain knowledge and it depends only on the state and not on the specific node.
Most common **Informed Search Strategies include**:

- **Greedy Best-First Search:** expand the node with lowest estimated cost to the closest goal, not considering the cost between the actual state and the node to expand. This strategy is characterized by $O(b^d)$ time complexity and $O(b^d)$ memory size requirement (which is a lot); it's also not complete in its tree-based version if it does not implement loop control, and it is not optimal.
- **A* search:** a variation of the best-first strategy that avoids expanding paths that are expensive. It combines *Uniform-Cost* and *Greedy search* into a function: $f(n) = g(n) + h(n)$ which estimates which one is the node with related paths that have the lowest estimated cost to the goal through the actual node n.



This strategy's heuristics $h(n)$ is <u>admissible/ optimistic</u> if it never overestimates the cost to reach the goal, while it's <u>consistent/ monotonic</u> if each action, that generates a successor n' of n with a step cost $c(n, a, n')$, it verifies triangular inequality: $h(n) \leq c(n, a, n') + h(n')$. *A consistent heuristic is also admissible and the function $f(n)$ is non-decreasing along any path*. In the graph case, A* selects node n from the frontier only if the optimal path to n has been found, and it "*prunes*" non-optimal nodes, which won't be expanded. This strategy is characterized by $O((b^\epsilon)^d)$ time complexity, where the relative error is $\epsilon = (h^* - h)/h^*$ and $O(b^d)$ and b^ϵ is the effective branching factor, $O((b^\epsilon)^d)$ memory size requirement (which is a lot); it's also complete, and it is optimal.

function ~~UNIFORM-COST-SEARCH~~(*problem*) **returns** a solution, or failure
Astar-Search
  node ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with *node* as the only element
  explored ← an empty set
+Heuristic
  loop do
    if EMPTY?(*frontier*) **then return** failure
    node ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
    if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
    add *node*.STATE to *explored*
    for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
      child ← CHILD-NODE(*problem*, *node*, *action*)
      if *child*.STATE is not in *explored* or *frontier* **then**
        frontier ← INSERT(*child*, *frontier*)
      else if *child*.STATE is in *frontier* with higher PATH-COST **then**
        replace that *frontier* node with *child*
+Heuristic

Variations of the A* algorithm are Iterative-deepening A*, Recursive best-first search & Memory-bounded A*, which maintain completeness and optimality and try to limit the exponential memory usage. Note that the memory usage can be substantially mitigated by the quality of the heuristics.

- **Memory-bounded Heuristic Search**
- **Heuristic Functions**

Given two heuristics $h_1(n)$ and $h_2(n)$, $h_2(n)$ dominates $h_1(n)$ if $h_2(n) \geq h_1(n) \quad \forall n$ and $h_2(n)$ is better for search.
Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem, where the relaxed problem adds edges to the state space, as any optimal solution in the original problem is also a solution in the relaxed problem. The only constraint is that the derived heuristics for the relaxed problem must obey the triangular inequality and be consistent. If a problem is written down in a formal language, it's possible to construct relaxed problems automatically (ABSolver tool).
Another form of heuristics discovery is inductive learning through experience via **Machine Learning**.

Given a task environment where we do not care about the path, but only about the goal, we can implement **local search** as an iterative-improvement algorithm, where the current state is continuously improved, without remembering the path. This approach uses significantly less memory and is able to perform even in very large state spaces. A common implementation of this algorithm is ***Hill-Climbing Search***, which implements a sort of gradient-ascent to continuously improve the state, with the drawback of getting stuck on local minima. To avoid this it's enough to complement this technique with random restarts or with other techniques.

```
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
```

In **simulated annealing**, the algorithm starts similarly to Hill-Climbing by trying to iteratively improve the current state. However, instead of only accepting improvements, it occasionally accepts worse states, which allows it to "escape" from local minima and explore a larger portion of the state space. The likelihood of accepting a worse state depends on a probability that decreases over time, governed by a parameter called *temperature*.

```
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^(ΔE/T)
```

Another approach is **Local Beam Search**, which expands the search to k states instead of one, then replacing those k with other k elements among all the current generation's successors (like natural selection with asexual reproduction).
With similar premises, **Genetic algorithms** resemble natural selection with sexual reproduction by combining two parent states for each successor state. Parent states are rated

according to a fitness function, that is related to the probability of being selected. For each pair the crossover point is chosen randomly and a random mutation is introduced.

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

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

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

In **non-deterministic environments**, where outcomes are uncertain, a solution is not a simple sequence of actions but a contingency plan or strategy. This plan must account for various possible future perceptions and make decisions *based on the values of state variables*.
A common way to represent such strategies is through **And-Or Trees**. In these trees:

- OR nodes *represent different possible states the system could be in. The goal is to find a plan that works for at least one of these states*.
- AND nodes *represent actions that the agent can take. To satisfy an AND node, the plan must include solutions for all possible outcomes of that action (since actions in non-deterministic environments may have multiple potential results)*.

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])
```

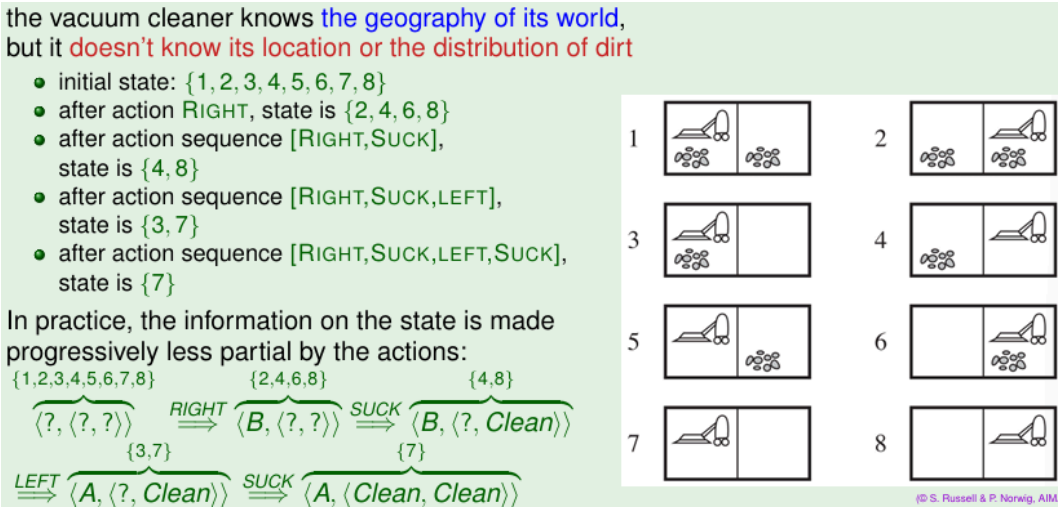
```
function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure   // CYCLE DETECTION
  for each action in problem.ACTIONS(state) do
    plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
    if plan ≠ failure then return [action | plan]
  return failure
```

```
function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
    plan_i ← OR-SEARCH(s_i, problem, path)
    if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

A key feature of this approach is cycle detection. If a state reappears in the same path, it doesn't necessarily imply failure. Instead, it indicates that if a solution exists, it should be accessible from an earlier instance of that state, allowing the new instance to be discarded to prevent looping. This mechanism is crucial in ensuring that the algorithm *avoids infinite loops*

while still exploring all possible paths to a solution. For finite state spaces, the search is considered complete, as every path will eventually lead to a goal, a dead-end, or a loop. The algorithm can also be enhanced with an "explored" set to *avoid redundant searches*, making it similar to graph-based searches. Though typically depth-first, the search can be adapted to breadth-first or best-first strategies, with even an A* variant available for AND-OR search.
In some cases, solutions may involve cycles. *A cyclic solution is valid as long as every leaf in the cycle is a goal state and can be reached from any point within the plan*. This concept can be modeled using programming structures like loops, ensuring that even with non-deterministic actions, the search will eventually reach a goal.
The search assumes nondeterministic yet observable environments. This contrasts with deterministic but partially observable environments, where hidden information, like a broken device we are unaware of, would make such an approach less reliable.

In **partially observable** environments, *agents often cannot determine the precise current state, but instead, they develop a theoretical belief state*, which represents the possible physical states the agent could be in.
When there are no sensors available, thus in **non observable** environments, the agent's **belief state** encompasses all potential physical states. *As the agent performs actions, it transitions from one belief state to another, depending on the effects of those actions in the possible states*.



In the belief-state formulation of a problem, belief states are subsets of physical states. For a physical system P with N states, the sensorless problem can include up to 2^N possible belief states. *Initially, the belief state might represent the set of all physical states*. Actions in this context are defined as the union of actions across all possible physical states within the belief state. The **transition model** describes how an action leads to a new belief state. *For deterministic actions, this new belief state is composed of physical states resulting from the action. For non-deterministic actions, all possible outcomes are considered, forming a broader belief state. For illegal actions the outcome is null as they don't produce changes*.
An action sequence that solves a belief state problem must lead the agent to a state where all possible physical states meet the goal condition. If a solution is found for a belief state, any subset of that belief state is also solvable. This principle allows for search algorithms to be efficiently applied to belief-state spaces, as any path leading to a goal in a larger belief state can also solve any of its subsets, greatly improving computational efficiency.
Since in **partially observable** environments, many states can produce the same percept, meaning that a percept does not fully determine the state, actions and goals are handled similarly to sensorless problems, but observations introduce additional complexity.

The agent thus solves the problem using a three-step process:

1. **Prediction**: This step predicts the next belief state after an action a. The belief state $\hat{b}$ is a set of states that could result from the action, similar to the sensorless case.
2. **Observation Prediction**: The agent determines the set of possible percepts it might receive after the action. This involves identifying all percepts that could be observed from the predicted belief state.
3. **Update**: For each possible percept o, the agent updates its belief state to reflect only the states that could have produced that percept. This partitions the belief state into smaller subsets, one for each percept, reducing uncertainty when observations are deterministic.

Partially observable problems can be framed as non-deterministic belief-state search problems. Non-determinism arises from the uncertainty of what percepts will be received after an action. The solution in this case is a **conditional plan**, which lays out *actions based on future observations*. AND-OR search algorithms can be applied to solve these problems, creating plans that branch depending on the percepts encountered.
The agent in this scenario is similar to a problem-solving agent, but with two key differences. First, the solution is a conditional plan instead of a simple sequence of actions. Second, the agent must maintain its belief state as it performs actions and receives percepts, continuously updating its state estimate. This process mirrors the prediction-observation-update mechanism, except it is simplified since the percept o is provided directly by the environment, removing the need for prediction. Additionally, this process must occur quickly, especially in complex applications where the agent might need to compute approximate belief states in real-time.

In **online search**, the agent operates under the assumption of a deterministic, fully observable environment, but it only knows which actions are available from the current state. *It cannot know the outcome of an action until it performs it*. The agent uses a heuristic function $h(s)$ to estimate the cost from the current state to the goal. The objective is to reach the goal with minimal cost, though the agent must deal with uncertainties like deadends, which can result in an inefficient search.
Deadends are an inevitable challenge in online search, especially in environments with irreversible actions. No algorithm can fully avoid deadends in all possible state spaces. The adversary argument highlights that for any online search algorithm, an adversary can construct a state space that forces the agent into a deadend by dynamically adjusting the space based on the agent's choices. A key assumption in this case is that the state space is "*safely explorable*," meaning that from every reachable state, a goal is attainable, particularly if actions are reversible.
Online search agents operate by *building and continuously updating a map* of the environment based on percepts they receive after every action. Unlike offline algorithms (like A* or BFS), which can expand any node, *online search agents can only expand nodes where they are physically present*. As a result, depth-first search (**DFS**) is a natural choice, as *the agent must physically backtrack to the previous state when necessary*. This requires the agent to maintain a table of predecessor states to facilitate backtracking to unvisited areas. This means that online search agents function well only if actions are reversible. In the worst case, each action-state pair is visited twice: once during exploration and once during backtracking. This means the agent may take a long, indirect path to the goal even if it is close. An online iterative deepening approach can help mitigate this problem by progressively exploring deeper states while maintaining efficiency.

```
function ONLINE-DFS-AGENT(problem, s') returns an action
                s, a, the previous state and action, initially null
                result, a table mapping (s, a) to s', initially empty
                untried, a table mapping s to a list of untried actions
                unbacktracked, a table mapping s to a list of states never backtracked to

  if problem.IS-GOAL(s') then return stop
  if s' is a new state (not in untried) then untried[s'] ← problem.ACTIONS(s')
  if s is not null then  // if neither initial nor result of backtracking
      result[s, a] ← s'
      add s to the front of unbacktracked[s']
  if untried[s'] is empty then //backtrack
      if unbacktracked[s'] is empty then return stop
       a ← an action b such that result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])  // all actions in actions(s') have been tried
  s ← s'
  return a
```

// results[s',b] exists because untried[s'] is empty

// added in 4th ed. AIMA

s' ← null

Hill climbing is a natural candidate for online local search due to its simplicity—it only stores one state at a time. However, it can get stuck in local minima, where no progress can be made toward the goal. Random restarts aren't feasible in an online setting, but **random walks** can be used to explore the environment. In a random walk, the agent *selects a random action at each step, favoring untried actions*. This approach guarantees eventual exploration of the space but is inefficient, as it can take exponentially many steps to find the goal, especially when backward progress is more likely than forward progress.

Learning Real-Time A* (**LRTA***) improves hill climbing by adding *memory*. The agent maintains a "*best estimate*" $H(s)$ of the cost to reach the goal from each state it has visited. This estimate is initially based on the heuristic $h(s)$ but is updated as the agent gains experience in the state space. After each action, the agent updates $H(s)$ using the cost of the transition and the estimate for the target state s'. This method, known as "optimism under uncertainty," *encourages the agent to explore promising new paths by assuming untried actions lead directly to the goal with minimal cost*. An LRTA* agent is guaranteed to find a goal in any safely explorable, finite environment, although it does not guarantee optimality.

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

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

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

Games represent an important class of multi-agent environments in AI. They provide a framework for studying competitive scenarios where agents' actions directly impact each other's success. Unlike cooperative environments, games give rise to adversarial problems that require specialized approaches.
The study of games in AI is valuable for several reasons:

1. Games offer a well-defined, rule-based environment that's easy to represent computationally.
2. Despite their simple representation, many games are computationally complex, presenting significant challenges.
3. Game theory serves as a metaphor for various real-world domains, including competitive markets, biology, sports, politics, and warfare.

Traditional search aims to find a path to a goal state, often using heuristics to estimate the cost from the current state to the goal. In contrast, adversarial search must determine a **strategy** that *specifies moves for every possible opponent action*. The evaluation function in games assesses the "goodness" of a game position rather than estimating a path cost.
Games can be classified based on several characteristics:

- Deterministic vs. stochastic (involving chance)
- Number of players (one, two, or more)
- Zero-sum vs. general payoff structures
- Perfect vs. imperfect information

The most commonly studied games in AI are deterministic, turn-taking, two-player, zero-sum games with perfect information. The goal is to develop algorithms that calculate a policy (or strategy) mapping states to actions.
Key concepts in game theory include:

- *Players*: Usually denoted as MAX and MIN in two-player games
- *Initial state*: The starting configuration of the game
- *Actions*: Legal moves available in each state
- *Result*: The transition model defining the outcome of each action
- *Terminal test*: Determines if the game has ended
- *Utility function*: Assigns numeric values to game outcomes for each player

In **zero-sum games**, *players have directly opposing goals*, and the total payoff is constant across all game instances. This allows us to represent the game using *a single utility value that one player tries to maximize while the other tries to minimize*.
The **Minimax algorithm** is fundamental to *adversarial search*. *It assumes both players act optimally and try to maximize the worst-case outcome for the MAX player*. The algorithm recursively evaluates game states by considering all possible moves and counter-moves, assigning values based on terminal states and propagating them up the game tree.

```
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
```

The Minimax value of a state is defined as:

- The utility value for terminal states
- The maximum of children's Minimax values for MAX nodes
- The minimum of children's Minimax values for MIN nodes

This approach effectively creates a contingent strategy that specifies the best move for every possible game state, ensuring optimal play against a perfect opponent.
While Minimax guarantees optimal play, its computational complexity can be prohibitive for games with large state spaces. This limitation has led to the development of various optimization techniques and approximate algorithms in advanced game AI research.
In a **multi-player** version of the game each node value has to be replaced with a vector, where values represent scores from each player point of view. In this scenario some players' goals might be close, opening to the opportunity of establishing alliances that are based on *trust*.
Since *behavior is preserved under any monotonic transformation*, only the "proportion" of the quantities is important, not the exact values. For example the system cares that *node1* is bigger than *node2*, but it doesn't care if *node1* is *2* and *node2* is *1* or *node1* is *4* and *node2* is *0*.
The MinMax algorithm is *complete* if the state space is finite, it is *optimal* against optimal opponents and is characterized by a $O(b^m)$ *time complexity* and $O(bm)$ *space complexity*.
Since the time requirement is not feasible, a smart choice is **pruning** the tree to avoid analyzing irrelevant scenarios that lead to extreme advantage of the opponents.
Alpha-Beta pruning technique cuts a branch as soon as it discovers that the relative path leads the opponent to better opportunities than the other branches at the same level. The technique starts with a lower bound of -∞ and a higher bound of +∞ which are then updated. This technique does not affect the correctness of the final result and with a *perfect ordering* the algorithm time complexity reduces to $O(b^{m/2})$, while with *random ordering* it still approximately to reduces to $O(b^{3m/4})$. Note that this complexity is for each level. This means that complete bottom up search of the whole tree is impossible for most real case scenarios.
For realistic games it's thus necessary to employ depth limited search in combination with MinMax and replace the utility function with a heuristic evaluation function $Eval(s)$, which rely on a $CutOffTest(s, d)$ function.
The **heuristic evaluation** function returns an estimate of the expected utility from a given position and should thus be correlated to the chances of winning. It is often given by a weighted sum of features and it depends on depth.
The **cut off test** can be either straightforward with a *fixed depth limit*, it can apply *iterative deepening* or it can apply the *evaluation function only to quiescent states*, which are states that are *unlikely to exhibit big variations in value in the near future*, and *expand non-quiescent states* until *quiescence* is reached.
In **Stochastic Games** the minamax value is defined by chance and not by an adversary, making it necessary to rely also on the *average possible scenario* by including chance nodes. This scenario is modeled with the *addition of a third player*, which computes the weighted average of expected minmax outcomes. The ExpMinmax becomes then:

$$ExpectMinimax(s) \overset{\text{def}}{=} \begin{cases} Utility(s) & \text{if } TerminalTest(s) \\ \max_{a \in Actions(s)} ExpectMinimax(Result(s,a)) & \text{if } Player(s) = MAX \\ \min_{a \in Actions(s)} ExpectMinimax(Result(s,a)) & \text{if } Player(s) = MIN \\ \sum_r P(r) \cdot ExpectMinimax(Result(s,r)) & \text{if } Player(s) = Chance \end{cases}$$

In this case *exact values are crucial*, as *behavior is NOT preserved under monotonic transformations*, as it's only preserved under positive linear transformations. The interference of chance makes alpha-beta pruning less effective, but depth-limitations are still necessary.

States and transitions between them can be represented in three different levels of abstraction. In the **atomic** representation the state is a black-box with no internal structure; in the **factored** representation a state consists of a vector of attribute values; in the **structured** representation a state includes objects, each of which may have attributes of its own and relationships with other objects. By passing from atomic to factored representation, problem specific heuristics can be replaced with *general purpose* ones, eliminating large portions of the search space all at once. This is valid because with this type of representation the *goal test* is now a set of constraints given by combinations of values.
This new kind of problem is called **Constraint Satisfaction Problem**, and it's a tuple X-D-C composed by a set of variables, a set of non-empty domains (one for variable) and a set of constraints. ***Relations*** *are explicit lists of tuples of values that satisfy constraints*. ***States*** *are assignments of values to some (incomplete/partial assignment) or all (complete/total assignment) variables*. *A* ***solution*** *to a CSP is a consistent and complete assignment*. Obviously CSP can require a solution that maximizes/minimizes some objective function; in this case we talk about **Constraint Optimization Problems** (**COPs**).
CSP can be graphically represented as *constraint graphs/networks*, where nodes correspond to variables and edges connect variables that participate in a constraint. Variables can be *discrete* (either finite or infinite) or *continuous*, and some problems can have more than one distinct formulation as CSP. Constraints can be *Unary*, *Binary*, *Higher-Order*, *Global* (arbitrary higher number, but cannot be represented by constraint hypergraphs), or *Preference* (*soft constraints*, solved by cost-optimization search techniques).
K-ary constraints can be transformed into sets of binary constraints to reduce complexity. Two variables that share a binary constraint are called neighbors.

CSP search works by interleaving ***search*** (pick a new variable assignment, building complete states by progressively extending partial ones) with ***constraint propagation*** (use the constraints to reduce the set of legal candidate values for the neighboring variables). This means that backtracking is forced when candidate values for next variables are reduced to zero. This propagation enforces local consistency, which can be of different types:

- ***Node consistency*** *is reached when all the values in the variable's domain satisfy its unary constraints*. A CSP is node consistent if every variable is node consistent. Node consistency propagation removes a priori all the values that violate unary constraints from a node's domain.
- ***Arc consistency*** *is reached when for every value of a certain variable, a value for its neighbor exists so that the binary constraint between the two variables is satisfied*. A CSP is arc consistent if every variable is arc consistent. A simple way to implement it is through **forward checking**, which removes "illegal" values from unassigned variables. It first picks a "novel" variable assignment and then it updates the remaining legal values for unassigned variables. This means that it doesn't provide early detection for failures. **Arc consistency propagation** removes from the domains of every variable all values which are not arc consistent with those of some other variables. Contrary to forward checking, it rechecks neighbors of X after domain restriction, implementing an early failure detection.
  AC-3 algorithm is an algorithm used for arc consistency in CSP. It works with an average complexity of $O(|C| \cdot |D|^3)$ and it can be either interleaved with search or used as a preprocessing step.

```
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   // makes Xi arc-consistent wrt. Xj
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE( csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← 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
```

- ***Path consistency*** *is reached in a two variable set if, for every assignment of the two variables which is consistent with the constraints, there exists an assignment of a third variable that also satisfies the constraints*.
- ***K-consistency*** *is reached for a CSP if for any set of k-1 variables and for any consistent assignment of those variables, a consistent value can always be assigned to any other k-th variable*. Obviously time and memory complexity grow exponentially with k.

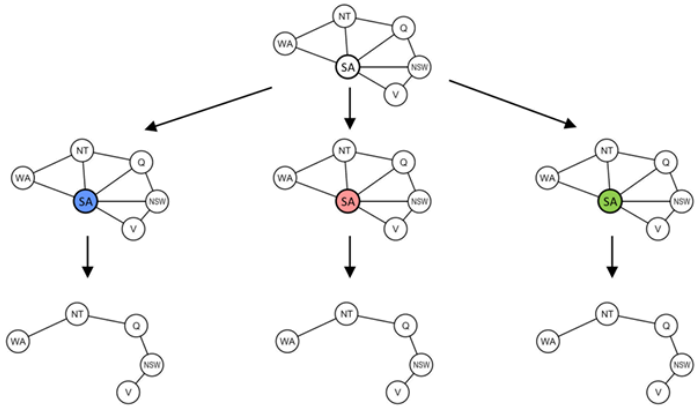
Backtracking search is a basic uninformed algorithm, based on *DFS*, for solving CSPs. It works by *picking one variable at a time*, reasoning on partial assignments and considering assignments to a single variable at each step. It can also check constraints as long as it proceeds, picking only values which do not conflict with previous assignments. This is done via an incremental goal test, which implements *early detection of inconsistencies and pruning*. With backtracking search, there is no need to provide a domain-specific initial state, action function, transition model, or goal test. It can be enhanced by providing custom functions for choosing the next variable to assign, to assign priorities and thus order the domain variables, and what kind of inference the algorithm should perform at each step.
The **variable selection heuristics** is usually implemented as Minimum Remaining Values(MVR) / most constrained variable / fail-first, where *the chosen variable to be assigned is the one with the fewest remaining legal values*. This approach early detects failures and performs the deterministic choices first. Another approach is the Degree heuristic, where *the picked variable is the one involved in the largest number of constraints on other unassigned variables*, so the one whose assignment will lead to the most restrictions in other variables. These two approaches can be used in combination to get the best out of both and solve ties.
The **value selection heuristics** is mainly done by Least Constraining Value (LCS) approach, where *the picked value is the one that reduces the less the number of remaining legal values for the other neighboring variables*.

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

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

Standard chronological backtracking goes back to the preceding variable which still has some untried variables as soon as it detects a branch failure. When *Nogoods* (sub assignments which cannot be part of any solution) happen, a possible solution is Conflict-Driven backjumping, where the no-good which identified the failure is identified via forward checking. Given that a *Conflict Set* of an empty domain variable is indeed a no-good, the failure point can be detected by removing from the set all the decisions that were forced by previous ones. This allows to pop the stack higher, to the latest non-deterministic decision that was taken, saving a lot of search.

Local Search can be extended to CSP by using complete state representation and allowing states with unsatisfied constraints. This process relies on creating neighboring states that differ for one variable value and iteratively reassign some values to try to satisfy all the constraints. A common approach is a sort of *hill climbing*, where a variable is chosen randomly among the conflicting ones and then the value that conflicts the least with the other variables is assigned as new value. Other approaches include *random walk, simulated annealing, GAs, taboo search* etc…
To increase the performance it's possible to partition the problem into a set of smaller problems by identifying strongly connected components with **Tarjan's linear algorithm** and dividing the space separating it at "weak links".
CSPs constraint graphs with no loops can be solved in $O(nd^2)$ *time, while general graphs have a complexity of* $O(d^n)$.
CSPs tree-structures are solved by identifying a small cycle cutset S and then, for each consistent assignment to the variables in S the inconsistent values are removed from the domains of the remaining variables and the tree-structure CSP algorithm is applied. This bring time complexity down to $O(d^c \cdot (n - c)d^2)$, *where* c *is* $|S|$.



If values are equivalent and the only difference is the name, a problem with domain size of n will have $n!$ solutions given by the permutations of every solution. This means that there is more than one possible solution and also that there are similar no-goods. It's thus crucial to assign *value ordering constraints* to avoid problems related to **symmetry**. In this way a certain value is given priority over the others, avoiding useless computations and reducing the possible solutions to just 1.

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

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

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

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

- $\wedge, \vee, \leftrightarrow$ and $\oplus$ are commutative:
  $(\alpha \wedge \beta) \iff (\beta \wedge \alpha)$
  $(\alpha \vee \beta) \iff (\beta \vee \alpha)$
  $(\alpha \leftrightarrow \beta) \iff (\beta \leftrightarrow \alpha)$
  $(\alpha \oplus \beta) \iff (\beta \oplus \alpha)$
- $\wedge, \vee, \leftrightarrow$ and $\oplus$ are associative:
  $((\alpha \wedge \beta) \wedge \gamma) \iff (\alpha \wedge (\beta \wedge \gamma)) \iff (\alpha \wedge \beta \wedge \gamma)$
  $((\alpha \vee \beta) \vee \gamma) \iff (\alpha \vee (\beta \vee \gamma)) \iff (\alpha \vee \beta \vee \gamma)$
  $((\alpha \leftrightarrow \beta) \leftrightarrow \gamma) \iff (\alpha \leftrightarrow (\beta \leftrightarrow \gamma)) \iff (\alpha \leftrightarrow \beta \leftrightarrow \gamma)$
  $((\alpha \oplus \beta) \oplus \gamma) \iff (\alpha \oplus (\beta \oplus \gamma)) \iff (\alpha \oplus \beta \oplus \gamma)$
- $\rightarrow, \leftarrow$ are neither commutative nor associative:
  $(\alpha \rightarrow \beta) \not\iff (\beta \rightarrow \alpha)$
  $((\alpha \rightarrow \beta) \rightarrow \gamma) \not\iff (\alpha \rightarrow (\beta \rightarrow \gamma))$

$\neg\neg\alpha \iff \alpha$
$(\alpha \vee \beta) \iff \neg(\neg\alpha \wedge \neg\beta)$
$\neg(\alpha \vee \beta) \iff (\neg\alpha \wedge \neg\beta)$
$(\alpha \wedge \beta) \iff \neg(\neg\alpha \vee \neg\beta)$
$\neg(\alpha \wedge \beta) \iff (\neg\alpha \vee \neg\beta)$
$(\alpha \rightarrow \beta) \iff (\neg\alpha \vee \beta)$
$\neg(\alpha \rightarrow \beta) \iff (\alpha \wedge \neg\beta)$
$(\alpha \leftarrow \beta) \iff (\alpha \vee \neg\beta)$
$\neg(\alpha \leftarrow \beta) \iff (\neg\alpha \wedge \beta)$
$(\alpha \leftrightarrow \beta) \iff ((\alpha \rightarrow \beta) \wedge (\alpha \leftarrow \beta))$
$\iff ((\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta))$
$\neg(\alpha \leftrightarrow \beta) \iff (\neg\alpha \leftrightarrow \beta)$
$\iff (\alpha \leftrightarrow \neg\beta)$
$\iff ((\alpha \vee \beta) \wedge (\neg\alpha \vee \neg\beta))$
$(\alpha \oplus \beta) \iff \neg(\alpha \leftrightarrow \beta)$

Boolean formulas can be represented either as trees or as Directed Acyclic Graphs. The *DAG* representation is usually exponentially smaller.
Given a logical sequence φ (phi), we define truth assignment μ a mapping that assigns truth values to the atomic variables in φ.
Truth assignment $\{\mu(A_1) := \top, \mu(A_2) := \bot\}$ can be either total or partial (some atomic variables are still unassigned) and can be represented as set of literals ⇒ $\{A_1, \neg A_2\}$ or as a formula (*cube*) ⇒ $\{A_1 \wedge \neg A_2\}$.
We say that alpha **entails** beta (α |= β) when if alpha is true, the beta must also be true. This generates the deduction theorem, which states that α |= β only if $\alpha \rightarrow \beta$ is valid. With the corollary that α |= β only if $\alpha \wedge \neg\beta$ is unsatisfiable.
The logical sequence φ is **valid** (|= φ) if every truth assignment entails phi. Contrary, it's true the property that φ is valid if ¬φ is unsatisfiable.
Given $M(\varphi) = \{\mu | \mu \models \varphi\}$ the set of models of φ, we have that a total truth assignment **satisfies** φ, but a partial truth assignment satisfies φ only if all its total extensions satisfy φ.
This means that validity, inference, equivalence and entailment checking can be reduced to un-satisfiability checking.
Since for N variables there are up to 2^N truth assignments to be checked, the problem of deciding the satisfiability of a propositional formula/logical sequence is NP-complete.
We say that α and β are **equivalent** if for every truth assignment, if the truth assignment entails α then the same truth assignment also entails β. This means that no matter how you set up the truth values for each variable in the formulas, alpha and beta will always both be true or both be false together. If α and β are equivalent, they are also equi-satisfiable.
Two formulas α and β are **equi-satisfiable** if both can be satisfied (made true) by at least one truth assignment, but they do not need to be true in exactly the same situations. These conditions are used when beta is the result of some transformation of alpha. They are useful to check the consistency of the transformation.
A formula φ is in Conjunctive Normal Form (CNF) if it is written as a *conjunction* (AND) of *clauses*, where each clause is a *disjunction* (OR) of literals. $\bigvee_{i=1}^{L} \bigwedge_{j_i=1}^{K_i}$

CNF is useful because it's easier to handle as it's just a list of lists of literals and many logical reasoning techniques, such as resolution and SAT solvers, work more effectively or exclusively with formulas in this form. Breaking down logic problems into CNF allows for standardized processing methods.
To convert any logical formula into CNF, we follow a few systematic steps:
Step-by-Step CNF Conversion:

1. Eliminate Implications and Biconditionals:
   - Replace implications (A→B) with ¬A ∨ B.
   - Replace biconditionals (A↔B) with two implications: (A→B) ∧ (B→A).

2. Move Negations Inward:
    - Use **De Morgan's laws** to push negations inward, turning ¬(A⋀B) into ¬A⋁¬B and ¬(A⋁B) into ¬A⋀¬B.
    - Simplify double negations (¬¬A=A).
3. Distribute OR over AND:
    - Apply distribution to ensure the formula consists of clauses joined by ANDs, where each clause only contains ORs.
    - For example, distribute A⋁(B⋀C) to get (A⋁B)⋀(A⋁C).

It has to be noted that the resulting formula is exponential in the worst case and that Atoms in CNF are the same as they were previously. Also, CNF is equivalent to the previous one.

Labeling CNF, also known as **Tseitin Transformation** or **Tseitin CNF Conversion**, is a method for converting a complex formula into an equivalent CNF form in a more efficient way. This approach avoids the exponential growth in formula size that sometimes occurs with classic CNF conversion, producing a linear worst case result. Labeling CNF solves the exponential problem by introducing new "label" variables to represent subexpressions, converting each subexpression separately to CNF, and connecting them with the new variables. This allows the resulting formula to be **equi-satisfiable** with the original formula without being equivalent.
It's based on the following steps:

Step 1: Introduce Labels for Subformulas

- For each subformula (like A⋀B or C⋁D), introduce a new label variable (e.g., B1, B2, etc.).
- Assign each new variable to represent the truth of each subformula.

Step 2: Express Equivalences with CNF Clauses

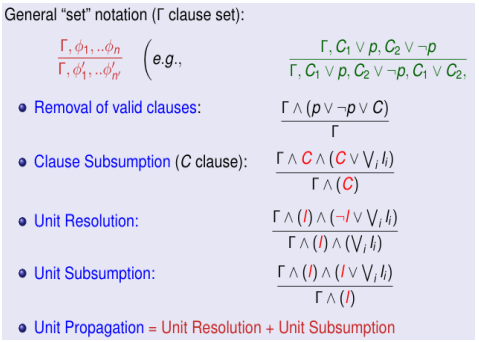
- Write CNF clauses to enforce the equivalence between each label and its subformula. For example, if B1 represents A⋀B, you add CNF clauses that ensure B1 is true if and only if A⋀B is true.

Step 3: Combine All Clauses

- The final CNF formula is the conjunction of all these smaller clauses, along with a clause that enforces the truth of the overall formula's main label.

CNF formulas are used to feed SAT (satisfiability) solvers. This kind of system is used as backend of engines in Automated reasoning, AI and many other fields.

The ***Resolution rule*** states that if you have two clauses $A \vee B$ and $\neg A \vee C$, then you can combine them into a new clause $B \vee C$. This rule is used iteratively in ***Basic Propositional Inference*** until either it's no more applicable or a false clause is generated. This approach is correct and complete, but time and memory inefficient. Some general improvements can be made by subsumptions and unit propagation, with deterministic rules applied before non-deterministic ones.



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

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

The ***Davis-Putnam-Logemann-Loveland Procedure*** tries to build an assignment μ that satisfies φ by assigning a truth variable to an atom (all its instances) at each step. It still performs deterministic choices before non deterministic ones, but it's more efficient than basic PL-resolution and it requires polynomial space, still being complete and correct.

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

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

---

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

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

Modern SAT solvers are non-recursive, stack-based implementations based on ***Conflict Driven Clause Learning*** schemas. The main idea behind ***CDCL*** is to learn from encountered conflicts/nogoods (situations where no satisfying assignment can be made) to avoid repeating the same mistakes and to prune the search space. It starts by making a decision about variables assignments; then it propagates these assignments, deducing other variable values and detects eventual conflicts. In this case it adds new clauses that prevent the same conflict from rising again. After a conflict is detected, the algorithm performs backjumping/chronological backtracking to the earliest point in the search where the conflict could have been avoided. Additionally, CDCL solvers periodically perform random restarts based on various mathematical-based heuristics to escape unproductive parts of the search space. CDCL solvers perform preprocessing/in-processing techniques to simplify formulas and smart indexing to do and undo assignments easily, allowing incremental calls (reusing previous search on "similar" problems).

Horn clauses contain at most one positive literal, they are called definite if they contain exactly one literal and goal if they have no positive literal at all. Similarly, Horn formulas are a conjunction/set of Horn clauses. This means that checking the satisfiability of Horn formulas requires polynomial time, as it starts by propagating unit clauses by propagating their value, then immediately checks if an empty clause is generated, it can directly return that the formula is unsatisfiable, otherwise, since every clause contains at least one negative literal, it can just set all variables to false and return the assignment. An alternative is to run CDCL with the heuristic of selecting the negative literals first.

```
function Horn_SAT(formula φ, assignment & μ) {
    Unit_Propagate(φ, μ);
    if (φ == ⊥)
        then return UNSAT;
    else {
        μ := μ ∪ ⋃_{A_i ∉ μ} {¬A_i};
        return SAT;
}   }

function Unit_Propagate(formula & φ, assignment & μ)
    while (φ ≠ ⊤ and φ ≠ ⊥ and {a unit clause (l) occurs in φ}) do {
        φ = assign(φ, l);
        μ := μ ∪ {l};
}   }
```

Local Search can be implemented for SAT problems by giving as input a set of clauses and using the total truth assignments forcing neighboring states to differ for one variable truth value. At each step variable truth values are reassigned and the total cost is given by the number of unsatisfied clauses. Similarly, stochastic local search applies to SAT problems. An exemplary approach is the ***Walk SAT***, where an unsatisfied clause is randomly selected and for that clause flip one variable at random with probability *p* and flip the one variable causing a minimum number of unsatisfiable clauses with probability *p-1*. This algorithm has many variants, but in general can detect only satisfiability and not unsatisfiability.

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

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

Agents can be improved by allowing them to maintain a ***Knowledge Base*** of data that represent the agent's world as a *collection of domain-specific facts*. These data are expressed in *formal language* (meaning non-ambiguous) like propositional logic, and are thus queried via *logical entailment*, and powered by an inference engine. The engine is powered by *domain-independent algorithms* that are combined with the Knowledge Base. This system allows the definition of "**reasoning**" as "*formal manipulation of the symbols representing a collection of beliefs to produce representations of new ones*".

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

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

Logical Agents combine knowledge with current percepts to infer hidden aspects of current state prior to selecting actions. KB agents must be able to represent states and actions, incorporate new percepts, update internal representation of the world, deduce hidden properties of the world and select appropriate actions.
Logical agents use first-order/propositional logic representing the KB as a set of propositional formulas and percepts and actions as collections of propositional atoms (CNF representation). They then perform inference by *model checking* and *entailment*, implementing incremental calls to a SAT solver.

Logic, which is composed of *Language* (a class of sentences described by a formal grammar), *Semantics* (a formal specification of how to assign meaning in the "real world" to the elements of L), and *Inference System* (a set of formal derivation rules over L). Some of the most famous logics include: Propositional, First Order, Model, Description, Temporal, Fuzzy, Probabilistic, …

Propositional logic is compositional, and its sentences are context independent, making it very solid, but not very expressive, as it's based on binary facts. IN PL, knowledge is separated from inference, and it allows for *partial/disjunctive/negated information*.

First order logic is used to represent a structured environment, where world/states include quantifiable objects, which may have attributes and relationships/functions.
FOL is based on:

- *Constant symbols* (0-ary function symbols) objects
- *Predicate symbols* relations
- *Function symbols* functions
- *Variable symbols* variables
- *Propositional Connectives* (0-ary predicates) logical operators
- *Equality & Inequality*
- *Quantifiers* for-all, exists, etc…
- *Punctuational symbols*
- *Signature*: set of predicate, function & constant symbols

FOL syntax terms are constants or variables, which represent objects in the real world. Its atomic/non-atomic sentences denote facts/complex-facts via propositions and predicates.
A term/formula is "***ground***" only if no variable occurs in it; it is "***closed***" only if all variables occurring in it are quantified. This means that ground formulas are closed, but not vice-versa.

Polarity in formulas tells if the satisfaction of the subformulas contribute positively or negatively to the satisfaction of the general formula. Polarity rules are modulo 2, which means that 2 negatives cancel out, so we just care about the parity/disparity of occurrences:

- φ occurs positively in φ;
- if $\neg\varphi_1$ occurs positively [negatively] in φ,
  then φ_1 occurs negatively [positively] in φ
- if $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$ occur positively [negatively] in φ,
  then φ_1 and φ_2 occur positively [negatively] in φ;
- if $\varphi_1 \rightarrow \varphi_2$ occurs positively [negatively] in φ,
  then φ_1 occurs negatively [positively] in φ and φ_2 occurs positively [negatively] in φ;
- if $\varphi_1 \leftrightarrow \varphi_2$ or $\varphi_1 \oplus \varphi_2$ occurs in φ,
  then φ_1 and φ_2 occur positively and negatively in φ;
- if $\forall x.\varphi_1$ or $\exists x.\varphi_1$ occurs positively [negatively] in φ,
  then φ_1 occurs positively [negatively] in φ

Given a domain, which contains objects/elements, relations and functions, an interpretation is used to specify referents for variables (domain elements), constant symbols (domain elements), predicate symbols (domain relations), and functional symbols (domain functions).
The interpretation assigns domain values to variables and terms(set of variables); and it assigns truth values to formulas.This means that a sentence is true with respect to a model (which is a pair domain-interpretation) if the objects referred to by the sentence are valid in the interpretation. This holds because the interpretation $[f]^I$ must be total, it thus must return an output for every input. A trick to make it total is to introduce a general "Null" element.

The "**for-all**" quantifier is usually associated to implications, in the form $\forall x.\,(P(x) \rightarrow a(x,\ldots))$. It has to be noted that it propagates with AND but not with OR. It can be seen as a conjunction over all possible instantiations of x in a.
The "**exists**" quantifier, on the contrary, distributes over OR but not over AND. It can be seen as a disjunction over all possible instantiations of x in a.
Entailment in First Order Logic is *semi-decidable*, meaning that $\Gamma \models a$ can be checked in finite time, but $\Gamma \not\models a$ cannot be checked in finite time.

First Order Logic reasoning capabilities enables agents to answer queries by entailing information based on predefined assertions or axioms.

Inference in First-Order Logic (FOL) involves techniques that allow reasoning about predicates and quantifiers. One central concept in FOL is **substitution**, where terms or subformulas within an expression are replaced systematically. A substitution can occur at the level of terms, where, for example, substituting y with S(x) in y + 1 = 1 + y results in S(x) + 1 = 1 + S(x). Similarly, subformula substitutions allow entire logical subformulas to be replaced, as in transforming (Even(x) or Odd(x)) into (Odd(S(x)) or Odd(x)) by substituting Even(x) with Odd(S(x)). Multiple substitutions can occur simultaneously, ensuring changes are consistent across the entire formula. Substitution is particularly powerful when combined with rules like the ***Equal-term substitution rule***. This rule lets us replace one term with another equivalent term without altering the truth of the formula. For instance, starting with (S(x) = x + 1 and (0 ≠ S(x))), adding (0 ≠ x + 1) via substitution maintains the formula's validity. A similar principle applies to ***Equivalent-subformula substitution***, where equivalent logical subformulas can replace each other, preserving the truth value. For example, given (Even(x) ↔ Odd(S(x)) and (Even(x) or Odd(x)), substituting Even(x) with Odd(S(x)) , extends the formula to (Odd(S(x) or Odd(x)).

Quantifiers introduce unique inference rules like Existential Instantiation (EI). Here, an existentially quantified sentence, such as ∃x .α, can be replaced by a specific instance α with a fresh constant C (a Skolem constant). For example, the statement "There exists a crown on John's head" ∃ x .(Crown(x) and OnHead(x, John)) can be instantiated as Crown(C) and OnHead(C, John), where C represents a newly named crown. This replacement preserves satisfiability, meaning the truth of the original statement is retained in the instantiation.

It's essential to understand the distinction between Universal Instantiation (UI) and EI. While UI can repeatedly add equivalent sentences without changing the logical equivalence of the knowledge base, EI can only apply once for each existential quantifier. EI, however, ensures inferential equivalence, meaning the inferences drawn from the new knowledge base are consistent with the original one. Before applying either rule, formulas often need rewriting to move negations inside quantifiers. For example, negating a universal quantifier $\neg\forall x.\ \alpha$ converts it into an existential quantifier $\exists x.\ \neg\alpha$, and vice versa. This transformation ensures the formulas are ready for logical operations while maintaining their intended meaning.These tools—substitution, quantifier rules, and careful formula manipulation—form the backbone of inference in FOL, enabling structured reasoning in logical systems.

The ***Herbrand theorem*** states that if a ground sentence α is entailed by a FOL KB Γ, then it's entailed by a finite subset of the propositionalized KB Γ. The main problem is that propositionalization generates lots of irrelevant sentences and may create infinite instantiations where loops occur. It's possible to mitigate this problem by using terms of function nesting depth, but in general propositionalization is not very efficient and is not used.

A solution is the combination between Propositional Logic and **Unification** by generalizing the Modus Ponens. Traditional Modus Ponens states that if "All men are mortal" and "Socrates is a man," we can infer "Socrates is mortal." In its generalized form, **GMP** combines propositional logic with universal and existential instantiations (UI and EI) to allow reasoning over variables and predicates. The key idea is that if a substitution exists (denoted as θ) that matches all premises with their universally quantified variables replaced by terms, we can apply this substitution to infer new conclusions. This principle is foundational in systems like Prolog, where reasoning over knowledge bases of definite clauses (those with exactly one positive literal) leverages GMP for efficient inference.
To make GMP practically usable, the process of unification is employed. Unification is the method of finding a substitution θ that makes two formulas identical by aligning their terms and variables. For instance, unifying Knows(John,x) with Knows(John,Jane) results in {x/Jane}, and unifying Knows(John,x) with Knows(y,OJ) produces {x/OJ,y/John}. However,

unification can fail when contradictions arise, such as attempting to unify Knows(John,x) with Knows(x,OJ), where no consistent substitution exists for x.
Unification also involves a concept called **standardizing apart**, which ensures that different quantified formulas use distinct variable names to avoid conflicts. For example, ∀x.α and ∀x.β should be rewritten as ∀x1.α and ∀x2.β, where x1 and x2 are "fresh" variables.
Among the possible unifiers for two formulas, the **Most-General Unifier (MGU)** is the simplest and most flexible substitution. It ensures minimal commitments, allowing for further refinement if needed. For instance, given Knows(John,x) and Knows(y,z), two potential unifiers are {y/John,x/z} and {y/John,x/John,z/John}. The former is more general since the latter can be derived by applying an additional substitution {z/John} to it. The existence of an MGU is guaranteed when unification is possible, and it is unique up to variable renaming. This makes MGUs crucial in automated reasoning systems, which often compute them using optimized algorithms.

function UNIFY(x, y, θ) **returns** a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
    y, a variable, constant, list, or compound expression
    θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure **then return** failure
  else if $x = y$ **then return** θ
  else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) **and** COMPOUND?(y) **then**
    return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure

---

function UNIFY-VAR(var, x, θ) **returns** a substitution

  if $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** failure
  else return add $\{var/x\}$ to θ

First-order ***definite clauses*** are a specific type of logical expression in First-Order Logic (FOL) that play a significant role in knowledge representation and reasoning. They consist of clauses with *exactly one positive literal*, which ensures a simplified structure for inference. Universal quantifiers, which declare that a statement applies to all members of a domain, are implicitly assumed in these clauses, allowing for the omission of explicit ∀ symbols to streamline notation. Moreover, existential quantifiers (∃) are removed through a process called *existential instantiation (EI)*, where existentially quantified variables are substituted with fresh constants. This results in clearer and more manageable formulas for computational purposes. For example, a formula like `∀x ((King(x) ∧ Greedy(x)) → Evil(x))` can be restructured as a definite clause: `(¬King(x) ∨ ¬Greedy(x) ∨ Evil(x))`, using standard logical equivalences.

A crucial application of first-order definite clauses is found in Datalog, a system commonly used for relational database queries and knowledge bases. Datalog consists of *a set of definite clauses without function symbols*, which eliminates unnecessary complexity and makes inference processes more efficient. The restrictions imposed on function symbols and quantifiers in these clauses not only reduce computational overhead but also allow FOL systems to focus on practical reasoning tasks.

Forward chaining is a reasoning method in First-Order Logic that iteratively infers new facts from a knowledge base (KB) until no further inferences can be made or a goal is reached. At each iteration, it applies Generalized Modus Ponens (GMP) to infer new atomic sentences, checking these against the target query. Forward chaining is sound, as every inference strictly adheres to logical rules, and it is complete for definite knowledge bases, meaning it will always find the answer if the query is entailed by the KB. However, it is only semi-decidable—if a query is not entailed, the process may fail to terminate. For *Datalog queries*, forward chaining operates efficiently with a time complexity of $O(p \cdot n^k)$, where p is the number of predicates, n the number of constants, and k the maximum arity of predicates.
To improve efficiency, forward chaining can limit rule matching to those affected by newly added facts during the previous iteration. This approach prevents unnecessary computations but does not eliminate all performance challenges. Matching conjunctive premises against known facts remains NP-hard, as it involves computationally expensive processes akin to solving graph colorability problems. Despite this, forward chaining is widely used in practical applications such as deductive databases and expert systems, where its ability to iteratively build inferences from structured data is particularly valuable.

Backward chaining is a reasoning approach in First-Order Logic where the system starts with a goal and works backward to determine whether it can be inferred from the knowledge base (KB). At each step, the algorithm selects a goal, identifies a relevant implication (rule) in the KB, and applies Generalized Modus Ponens (GMP) in reverse. This process produces sub-goals, which must themselves be proven true to support the original goal. Backward chaining follows a depth-first recursive proof search, meaning it prioritizes exploring one branch of sub-goals before moving to others. This ensures space efficiency, as the memory requirement is linear in the size of the proof.
Despite its utility, backward chaining has notable limitations. It is incomplete, as it may fall into infinite loops when dealing with recursive rules, such as `P(x) → P(x)` or `Q(f(x)) → Q(x)`, leading to endless derivations like `P(c), P(c), P(c)...` or `Q(c), Q(f(c)), Q(f(f(c)))...`. These issues can be mitigated with loop detection mechanisms.
Additionally, backward chaining can be inefficient due to the repeated evaluation of sub-goals. This inefficiency can be addressed through caching, where previously computed results are stored and reused, though this requires additional memory. Despite these challenges, backward chaining is widely used in logic programming languages like Prolog, where its ability to focus directly on proving specific goals makes it highly effective for solving complex logical problems.

One of the key techniques in FOL inference is converting formulas into ***Conjunctive Normal Form*** (CNF). CNF is a standardized format where a formula is expressed as a conjunction of disjunctions of literals. This conversion involves several steps: eliminating implications and biconditionals, moving negations inward to obtain the Negation Normal Form, standardizing variables to avoid naming problems, ***skolemizing*** (the process of eliminating existential quantifiers in first-order logic by introducing Skolem functions), and finally, distributing conjunctions over disjunctions by applying recursively the DeMorgan rule, and standardizing

again. This process simplifies the structure of formulas, making them easier to handle in automated reasoning systems, while maintaining satisfiability and entailment.
Once the formulas are in CNF, resolution strategies can be applied to perform inference. Resolution is a rule of inference that allows deriving new clauses by resolving pairs of clauses that contain complementary literals. This method is *refutation-complete*, meaning that if a set of clauses is unsatisfiable, the resolution process will eventually derive the empty clause, indicating a contradiction. Various resolution strategies, such as ***ordered resolution*** (define a stable atom ordering and resolve only maximal literals) and ***hyper-resolution*** (divide clauses into nuclei if they have at least one negative literal, and electrons if they only have positive literals, then perform resolution among one nucleus and one electron), help manage the complexity and efficiency of the resolution process.

Agents need a plan to go from the initial state to the goal. *The plan is a sequence of action partially or totally ordered*. For simplicity, we will assume the "classical" single agent situation with a fully observable, deterministic and static environment.
To know if there exists any plan that solves a planning problem (***PlanSAT***) a computation in *PSPACE* has to be resolved. Since the solution is hard to find, it's possible to restrict the question to the existence of a plan of predefined max length that can solve the problem (***Bounded PlanSAT***). This restriction reduces the problem to the *NPSPACE* most of the time.
The ***P****lanning* ***D****omain* ***D****efinition* ***L****anguage* says that a *state is a conjunction of fluents* (ground, function-less atoms), with the assumption that all non-mentioned fluents are false and that distinct names refer to distinct objects. *Actions are defined as a set of action schemas that describe which fluent to change*. The *action schema is finally the combination of action name, list of variables, preconditions and effect/postconditions*, with implicit time and states. We define **AddList** as the set of positive literals in the action's effect and DeleteList the set of negative ones. *A set of action schema defines a planning domain*, which together with an initial state (conjunction of positive literals (ground atoms)) and a goal (conjunction of positive and/or negative literals) composes a *PDDL problem*.
As usual, problems can be resolved via forward/progression or backward/regression search.
Forward search uses a search algorithm to choose which action to do next, based on the satisfied preconditions. It implies a unit cost for each step and a tracking of the action sequence. *Breadth-first* and *best-first* search are sound and complete, meaning that if they return a plan, that plan is a solution, and they will always return the shortest valid plan when possible. The drawback is the exponential memory requirement, which makes them unusable. *Depth-first* and *greedy search*, nevertheless they are not complete, are more used as they are sound,require linear memory, and can be improved with simple loop checking.
Forward search needs a good *heuristic* to guide the search and avoid trying irrelevant actions or branching too much. A possible heuristic for A* search algorithm is the so-called ***problem relaxation***, which makes it easier to search by ignoring some preconditions. Since sometimes it's possible to assume that goals and preconditions contain only positive literals, a useful heuristic is to ignore delete-lists from all actions, achieving a monotonic progress towards the goal. Another simplification is to reduce the search space by clustering nodes, ignoring less-relevant fluents.
Backward search uses a search algorithm to choose which action is most probable to happen before a certain state, starting from the goal. The idea is to stay as general as possible, not defining all the unnecessary details. A more general subgoal is easier to match to previous states. Only needed constants will be instantiated, while the rest remains variables. To choose the best previous state it's best practice to compute the weakest possible preconditions for the current state. The previous action is thus chosen according to whether it is relevant (at least one of the action's effects must unify with an element of the actual state) and consistent (must not undo desired literals of the current action). In other words it has to do something

useful and not do any harm. Backward search usually keeps the branching factor lower than Forward search, but the fact that it reasons with state sets makes it harder to come up with good heuristics.

Planning graphs are a type of data structure used in AI planning to represent possible actions and their effects over time. They are constructed forward in time, starting from an initial state, and grow incrementally level by level. Planning graphs are primarily used to improve heuristics (they provide useful estimates for planning problems, such as determining the minimum number of steps needed to achieve a goal), detect unreachable goals by analyzing the graph structure, and decompose problems by leveraging the divide-and-conquer strategy.

A planning graph is organized into alternating levels:

Literal (state) levels: The set of propositions or facts that can be true at a given point.

Action levels: The actions that can be applied, given the current state.

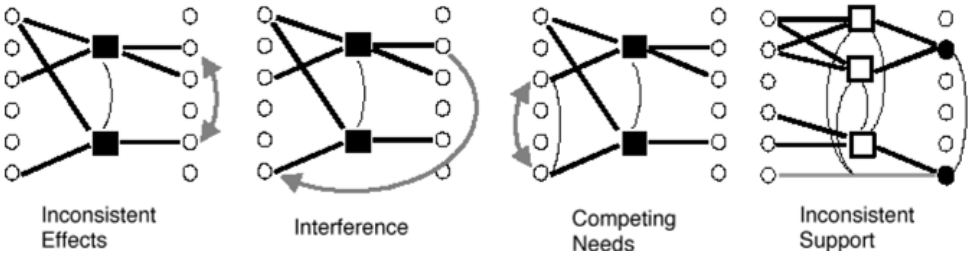
The graph alternates between these levels as it grows, starting with the initial state (a literal level).

Planning graphs operate on grounded representations, meaning that all states and actions are fully instantiated (e.g., specific objects are assigned to variables). Each action includes preconditions (facts that must hold to apply the action) and effects (facts that become true after the action is applied). If no action explicitly negates a literal in a state, it persists into the next level. This persistence is modeled by no-op actions, which essentially maintain the truth of a literal over time.

Mutual Exclusion (Mutex) Relationships/Links identify conflicts between literals or actions. They ensure the graph accurately reflects what is possible:

Mutex actions: Two actions are mutex if they interfere with each other, such as one action negating a precondition of another. Possible causes are inconsistent effects, inconsistent preconditions or interference in effects.

Mutex literals: Two literals are mutex if they present inconsistent support, thus cannot both be true simultaneously in a given state.



The correct way to initialize the graph is to start with the initial state as the first literal level. The next step is, for each action layer, to add all actions whose preconditions are satisfied by the literals in the previous literal level, including no-op actions to persist literals.

After this, at the next literal level, include all literals that result from the actions in the preceding action level, and detect and mark mutex relationships between actions and literals. This process is iterated alternating between literal and action levels until the graph "stabilizes" (i.e., no new literals or actions are added) or the goal is achieved.

Literals and actions increase monotonically, while mutexes decrease monotonically. Once a stable state is reached, if one of the goal literals is missing or is mutex with another goal literal, then the goal is not reachable and the process can be stopped. This is due to the fact that each level represents a set of possible belief states, where two literals connected by a mutex must belong to different belief states. *In planning graphs, a literal appears first in the lowest possible level it can appear in, meaning that if a literal does not appear in one state, it cannot be achieved by any plan*.

The ***level cost*** of a literal is the depth of the level it appears in first, but to get a better estimate it's necessary to force only one action per level to avoid inaccurate estimates.

Given this definition, we can say that the *max-level heuristic* is the maximum level cost of the sub-goals; the *level-sum heuristic* is the sum of the level costs of the goals; and the *set-level heuristic* is the level at which all goal literals appear together, without pairwise mutexes.

A planning graph is polynomial in the size of the problem, where both time and space complexity are $O(n(a + l)^2)$ where n is the number of levels, a the number of actions and l the number of literals. This makes the process of constructing the planning graph very fast.

GraphPlan is a smart way to solve planning problems by building a **planning graph** that maps out possible actions and states over time. It starts from the initial state, growing forward level by level, alternating between facts and actions, while identifying conflicts (like actions that can't happen together). This forward phase checks if the goal is even *possible*.
Once all the goals appear in the graph without conflicts, GraphPlan works **backward**, finding a sequence of actions to achieve the goal. If the backward search fails, it expands the graph further and tries again. This process continues until it either finds a valid plan or proves that no solution exists. It's efficient because it focuses only on viable options and avoids unnecessary exploration. It's like a form of controlled iterative deepening.
The algorithm repeatedly adds a level to a planning graph with the ***EXPAND-GRAPH*** step and if all the goal literals occur in the last level and are non-mutex it searches for a plan that solves the problem with the ***EXTRACT-SOLUTION*** step, otherwise it proceeds with expanding another level and adds the previous goal and level tuple to nogoods.
The EXTRACT-SOLUTION step can be formulated as an *incremental backward-search SAT problem* with one proposition for each ground action and fluent and where clauses represent preconditions, effects, no-ops and mutexes. In this way mutexes found by EXPAND-GRAPH prune paths in the search tree.
If the planning graph is not serialized, it may produce ***partial order plans*** (set of precedence constraints between action pairs) that can be converted into multiple distinct ***total order plans*** (strictly linear sequences of actions).
If the graph and the no-goods have both leveled off, and no solution is found, it means the task is not solvable and can thus return failure. This is true because literals increase monotonically and are finite, while mutexes and no-goods decrease monotonically, so once a stable point is reached we cannot go further.

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

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

To encode the bounded planning problem into a propositional formula and solve it by calls to a SAT solver the first step is the translation: ground fluents & actions at each step are propositionalized. The next step is to define $Init^0$ and $Goal^t$ as conjunctions of literals at step 0 and t. Then $Transition^{i,i+1}$ can be defined to encode the transition from a step to the next one.

```
function SATPLAN(init, transition, goal, T_max) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
          T_max, an upper limit for plan length

  for t = 0 to T_max do
    cnf ← TRANSLATE-TO-SAT(init, transition, goal, t)
    model ← SAT-SOLVER(cnf)
    if model is not null then
      return EXTRACT-SOLUTION(model)
  return failure
```

In ***real world planning*** time and resources scheduling is a fundamental part. A common approach is to plan first and schedule later. To do this we define a ***job*** as a collection of actions with a time duration and with ordering and resources constraints. A problem is thus a complete set of jobs.
Resources differ in *type, number, consumability/reusability*, and how they are *produced* by actions with negative consumption.
The scheduling part specifies the starting time of each action and must satisfy temporal ordering and resource constraints, making the cost function potentially very complicated.

A common approach to solve real world problems is to determine a schedule that minimizes the time needed to complete the problem, called ***makespan***. The result of this process is a ***path***, which is defined as a sequence of actions from Start to Finish. The *Critical Path* is the path with maximum total duration, where actions have $ES = LS$ where the first term is the earliest possible start time and the second term is the latest possible start time. This means that actions cannot be neither anticipated nor postponed. The overall complexity of the path is $O(Nb)$ with N number of actions and b maximum branching factor.
Given that critical path problems without resource allocation are computationally easy, as they are formed by only conjunctions of linear inequalities, adding the resources to the planning problem makes it NP-Hard, as "cannot overlap" constraints are translated into *disjunctions of linear inequalities*.
Some possible techniques to solve this problem are "branch-and-bound", "simulated annealing", "taboo search", reducing to constraint optimization problems, reducing to optimization modulo theories (SAT+LP), or integrating planning and scheduling.

Conditional/(***contingency***) ***planning*** is counterposed to ***sensorless***/(***conformant***) ***planning***, and introduces *uncertain percepts* as part of the problem. The classical planning with closed-world assumption is thus replaced with a partially-observable planning based on *open-world assumption*, where states contain both positive and negative fluents and where *if a fluent does not appear in a state, its value is unknown*.
Belief states are represented by logical formulas and not as explicit sets of states. The belief state corresponds exactly to the set of possible worlds that satisfy the formula representing it. In this case the unknown information can be retrieved via ***sensing***/***percept*** actions added to the plan. After each percept a check is needed to see if the plan requires revision.
Sensorless planning problems can be seen as a belief-state planning problem where the planner deals with factored representations rather than atomic elements, the physical transition model is a collection of action schemata, and the *belief state is represented by a logical formula instead of explicit states*. It has to be noted that all belief states may include unchanging facts (*invariants*) belonging to previous implicit domain knowledge. This

knowledge can be translated via *skolemization* into simple functions. For example instead of saying that each can has a color, we can introduce the function Color of Can.
In a certain belief state it's thus possible to apply every action whose preconditions are entailed by the belief state itself, adding the effects of the action to the belief state with deletion and additions as conjunctions of literals. *If the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals*.
Since in contingent planning sets of belief states are represented as disjunctions of logical formulas, the agent deploys the "then" branch if the belief state entails the conditions of the formula, and the "else" state if it entails the negation of the formula; in any case the algorithm must guarantee that the agent never ends in a belief state where condition's truth value is unknown!
The computation of the result is done with a conditional three-step process:

1) **Prediction** (same as for sensorless): updates the belief state to reflect the effects of an action as if the agent had perfect knowledge of what will happen. It doesn't yet account for sensing or observations—it's purely based on the action's described effects. If the action's effects are uncertain, the belief state becomes more complex, representing all possible outcomes of the action.
   $\widehat{b} = b \backslash Del(a) \cup Add(a) \Rightarrow \widehat{b} = b \wedge Effects(a)$
2) **Observation prediction**: predicts the observations the agent could encounter after performing the action. Observations depend on the action's result and the world's state.If many percepts are possible, the uncertainty remains high. A smaller set means more precise outcomes. It identifies all percept that could occur in the updated belief state and, for each of them, checks if preconditions are satisfied.
   $P = PossiblePercepts(\widehat{b}) = \{p | \widehat{b} \models Preconditions(p)\}$
3) **Update**: refines the belief state using the predicted percepts. For each percept:
   - If it has one condition, add it directly to the belief state. $b_p = p \wedge c$
   - If multiple conditions are possible, add all combinations (disjunction).The updated belief state becomes a complex logical expression (CNF formula)
     $b_p = \vee_{i=1}^{k} (p \wedge c_i)$

The final formula of the result is: $Result(b, a) = \widehat{b} \wedge \bigwedge_{p \in P} b_p$ where $\bigwedge$ is a logical conjunction over all the elements in the set of possible percepts or observations. It means you are combining all the individual conditions for each percept into a single logical expression using "AND." Since it's very hard to deal with CNF formulas, strong approximation is used to guarantee belief states to stay in a cube form.

Given the enormous amount of data processed by a system, it's necessary to define some efficient standards to represent the knowledge base and to integrate automated reasoning on it. **Knowledge Representation & Reasoning** uses logic to represent the most important aspects of the real world.
The activity of *formalizing a specific problem or task domain* is called knowledge engineering, while ontological engineering is the activity of *building general-purpose ontologies*, so ontologies that are applicable with the addition of domain specific axioms, to any special-purpose domain, and that combine different areas of knowledge.
Since *interactions happen at the level of objects*, while *reasoning happens at the level of categories*, Knowledge Representation requires the organization of objects into categories.
Categories can be represented either by *predicates* (*relations*), or as *reification of categories into objects* (*set of objects*), thus introducing ***membership*** and ***subcategories*** (subsets).
Members of a category and subcategories inherits the properties of the parent category.
The relations between categories and subcategories organize categories into ***taxonomic hierarchies***.
FOL allows to state facts about categories: *an object is a member of a category, a category is a subclass of another category, all members of a category have some properties, members of a category can be recognized by some properties, category as a whole has some properties, new categories can be defined by providing necessary and sufficient conditions for membership*.
Derived relations tell that *if two categories have no members in common they are* ***disjoint****, a set of categories is an* ***exhaustive decomposition*** *of another category if all members of this new category are covered by categories of the set, a disjoint exhaustive decomposition is a* ***partition***.
Quantitative properties of objects can be measured and then represented by unit functions, making them orderable and allowing for reasoning by exploiting transitivity of monotonicity.
Quantitative physics is a subfield of AI that investigates how to reason about physical systems without numerical computations, dividing countable objects(has extrinsic properties) from stuff(non countable-has intrinsic properties).

Propositional attitudes are used to represent what an agent knows, believes, wants, etc… and can be used to model also other agents' mental state. This can encounter problems when dealing with it like FOL, since referential transparency can generate wrong inferences in propositional attitudes.
Modal logics are an extension of common logics which introduces special modal operators that take formulas as arguments, instead of terms. Modal logics are based on the axioms:
distribution axiom (A is able to perform propositional inference): $K\colon (K_A\phi \ \wedge \ K_A(\phi \rightarrow \psi)) \rightarrow K_A\psi$
knowledge axiom (A only knows true facts): $T\colon \ K_A\varphi \rightarrow \varphi$
positive-introspection axiom (if A knows a fact, than it knows it knows it) $4\colon \ K_A\varphi \rightarrow K_A(K_A\varphi)$
negative-introspection axiom (if A does not knows a fact, than it knows it does not knows it) $5\colon \ \neg K_A\varphi \rightarrow K_A(\neg K_A\varphi)$
These axioms ensure ***Referential Opacity***, making Modal Logics from *NP-hard* to *PSPACE-hard*.
It also holds that knowledge distributes with AND but not with OR.

Properties in all (normal) modal logics (regardless T, 4, and 5):

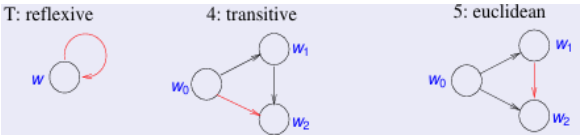
- $K_A(P \wedge Q) \iff K_AP \wedge K_AQ$
- $K_AP \vee K_AQ \models K_A(P \vee Q)$, but $K_A(P \vee Q) \not\models K_AP \vee K_AQ$ (e.g. $K_A(P \vee \neg P) \not\models K_AP \vee K_A\neg P$)
- $\varphi_1 \models \varphi \implies K_A\varphi_1 \models K_A\varphi$
- $\varphi_1 \wedge \ldots \wedge \varphi_n \models \varphi \implies K_A(\varphi_1 \wedge \ldots \wedge \varphi_n) \models K_A\varphi \implies K_A\varphi_1 \wedge \ldots \wedge K_A\varphi_n \models K_A\varphi$
- $\varphi_1 \models \varphi \implies \neg K_A\neg\varphi_1 \models \neg K_A\neg\varphi$
- $\varphi_1 \wedge \ldots \wedge \varphi_n \models \varphi \not\implies \neg K_A\neg\varphi_1 \wedge \ldots \wedge \neg K_A\neg\varphi_n \models \neg K_A\neg\varphi$

The combination of $\neg K_A \neg \varphi$ means that, for agent A, φ is possible, and its noted as $\langle A \rangle \varphi$, while the classic $K_A \varphi$ is noted as $[A]\varphi$.

Kripke models are collections of possible word states, where possible states are connected in a graph by accessibility relations (one relation for each distinct modal operator K_A).

A state is accessible with respect to K_A if everything which holds in the state is consistent with what A knows in the actual state. This means that two possible states may differ just from the beliefs of the agent.

Different modal logics differ by different properties of accessibility, in systems where just T holds, every world can access itself (reflexive), meaning that no more is known; in systems where 4 holds, every world can access itself (reflexive) and can also access other worlds transitively, meaning that if a world w0 can access w1, and w1 can access w2, then w0 can directly access w2. This implies that knowledge or possibility chains are fully connected, reflecting the idea that "knowing something also means knowing what it leads to", in systems where 5 holds, every world can access itself (reflexive), can access other worlds transitively (transitive), and any two worlds accessible from the same world are mutually accessible (Euclidean).



This means that the accessibility relation is highly structured, reflecting a state of perfect knowledge or logical omniscience: if something is possible in one accessible world, it is equally possible in all other mutually accessible worlds. This makes S5 the strongest and most idealized system of modal logic.

On the other hand, it has to be noted that *the less states are accessible, the more precise the knowledge is*, and the set of possible belief states is more limited.

Linear Temporal Logic is propositional logic augmented with temporal operators:
$X\varphi$: will be true in the next time step;
$F\varphi$: will eventually be true in an undefined future time step;
$G\varphi$: is always true in every time step;
$\varphi U \psi$: remains true until ψ occurs;
these operators tell something about φ.

Knowledge bases can be visualized through the use of ***semantic networks***, which are based on efficient algorithms for category membership inference, but they are also very limited in expressivity. A *Semantic Network* is a graph that can represent *individual objects*, *categories*, and *relations*, where *nodes are labeled and correspond to* ***concepts***, (either generic→categories/classes or individual→individual objects) while *binary relations between concepts,* called ***roles****, are represented by labeled arcs*. Two special relationships are always present: IS-A (subsetOf/subclassOf - represents subclass) and IstanceOf (memberOf - represents membership). This structure allows straightforward inheritance detection and the ability to represent default values for categories and exceptions, but cannot represent *negation, disjunction, nested function symbols* and *existential quantification*.

Another way to represent knowledge bases are ***description logics***, which are designed to describe definitions and properties about categories. Their principal inference tasks are:

- **Subsumption:** check if one category is a subset (sub-category) of another
- **Classification:** check whether an object belongs to a category
- **Consistency:** check if category membership criteria are satisfiable

Like SNs, it is based on concepts and roles, with the drawback that defaults and exceptions are lost. Concepts correspond to unary relations and can be either

- $\top$, $\bot$ are universal and empty concepts;
- atomic concepts like “Male”, “Female” etc… (categories)
- operators for the construction of complex concepts (and $\sqcap$, or $\sqcup$, not $\neg$, all $\forall$, some $\exists$, atleast $\geq n$, atmost $\leq n$)
- individual like “John” or other specific individual objects (used in assertions only)

Roles correspond to binary relations and can be combined with operators for constructing complex roles.
An interpretation Δ maps every concept to some more context specific values/categories that better represent the knowledge base.

The **T-Box**, or **Terminological Box**, represents the *terminological knowledge* of the system. It defines the vocabulary of the domain, including **concepts** (which are analogous to classes) and **roles** (relationships between concepts). In essence, the T-Box establishes the "rules" or "ontology" of the knowledge base—what types of things exist and how they relate to one another in a broad, general sense. For example, if we are modeling a zoo, the T-Box would state that *lions are mammals* and that *mammals are animals*. It might also express that a relationship like *hasChild* is a subset of a broader relationship, such as *hasRelative*. These statements in the T-Box reflect **universal truths** about the domain being modeled—they are schema-level declarations rather than facts about specific individuals.
The **A-Box**, or **Assertional Box**, complements the T-Box by capturing *assertional knowledge*. Where the T-Box sets out general rules, the A-Box provides specific **instances**—facts about particular individuals and their relationships. These statements deal with actual individuals and connect them to the concepts and roles defined in the T-Box. Returning to the zoo example, while the T-Box declares that all lions are mammals, the A-Box might assert that *Simba* is a lion, or that *Mufasa has Simba as a child*. Once the T-Box states that all lions are mammals, and the A-Box declares that Simba is a lion, a DL reasoner can **infer** that Simba is also a mammal, even if that fact isn’t explicitly stated.
In practice, the clear separation between the T-Box and A-Box reflects a division between the **schema** and the **data**. The T-Box focuses on what is *universally true* within the domain, independent of specific instances. Meanwhile, the A-Box grounds those rules in real-world data by providing concrete examples. This distinction makes the knowledge base modular and easier to reason over: the schema can evolve independently of the data, and vice versa.

In description logics, key tasks include designing and managing ontologies to ensure logical consistency and coherence, especially when integrating multiple ontologies.
Consistency checking verifies there are no contradictions, while creating hierarchies organizes concepts from general to specific (e.g., "Animal" subsumes "Mammal").
Queries enable reasoning: checking facts for consistency, determining if individuals belong to specific concepts, retrieving individuals satisfying certain conditions, and verifying subsumption relationships (whether one concept is more general than another). Together, these tasks allow effective reasoning and knowledge inference.

Description Logic $\mathcal{ALC}$	Modal Logic $K_{(m)}$
Individual	State
Role	Agent accessibility relation
Concept	Formula
Atomic concept	Atomic formula
Concept connectives: $\sqcap, \sqcup, \neg$	Boolean connectives: $\wedge, \vee, \neg$
Subsuption: $\sqsubseteq$	Implication: $\rightarrow$
Equivalence: $\equiv$	Bi-implication: $\leftrightarrow$
Universal Quantification: $\forall R.(...)$	Positive Modality: $K_A(...)$
Existential Quantification: $\exists R.(...)$	Dual Modality: $\neg K_A \neg(...)$