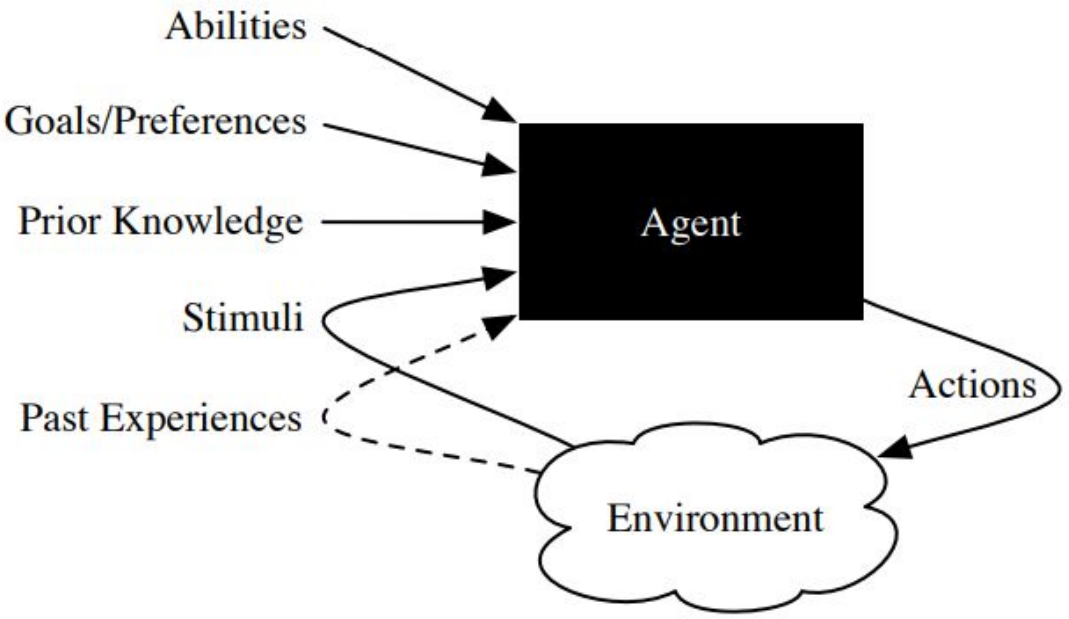
CS3061 Artificial Intelligence

Introduction

ELIZA effect: humans are inclined to see computers as humans.
An AI problem is **AI-complete** if any AI problem is mechanically reducible to it (i.e., it is at least as hard as any other).



An **agent** is something that acts in an environment; it does something.
A **computational agent** is an agent whose decisions about its actions can be explained in terms of computation.
A **knowledge base** is the representation of all of the knowledge that is stored by an agent.
The dynamics in the effect uncertainty dimension can be
• **deterministic** when the state resulting from an action is determined by an action and the prior state, or
• **stochastic** when there is only a probability distribution over the resulting states.
A **model** of a world is a representation of the state of the world at a particular time and/or the dynamics of the world.
The **knowledge base** is its **long-term memory**.
The **belief state** is the **short-term memory** of the agent.

Limits on:

- truth Liar's Paradox: 'I am lying'
- sets/membership ∈ Russell set R = {x | not x ∈ x}
- countability Cantor: Power({0, 1, 2, . . .})
- change Sorites : heap (minus one grain)
- computability Turing: Halting Problem

The Halting Problem

The halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running (i.e., halt) or continue to run forever.

Given a program P and data D, return either 0 or 1 (as output), with 1 indicating that P halts on input D
HP(P, D) := 1 if P halts on D, 0 otherwise

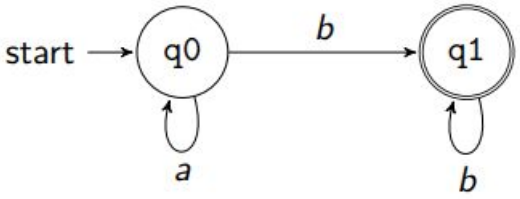
Theorem (Turing) No TM computes HP.
The proof is similar to the Liar's Paradox distributed as follows
H: 'L speaks the truth'
L: 'H lies' with a spoiler L (exposing H as a fraud).

+ Proof that I don't think it's relevant for exam. Or at least hope not.

A **universal Turing machine** is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input.
A **Turing Machine** a mathematical model of a hypothetical computing machine which can use a predefined set of rules to determine a result from a set of input variables.

Finite State Machines

Finite state machine (fsm)



A **fsm** `M` is a triple `[Trans, Final, Q0]` where

- `Trans` is a list of triples `[Q,X,Qn]` such that `M` may, at state `Q` seeing symbol `X`, change state to `Qn`
- `Final` is a list of `M`'s final (i.e. accepting) states
- `Q0` is `M`'s initial state.

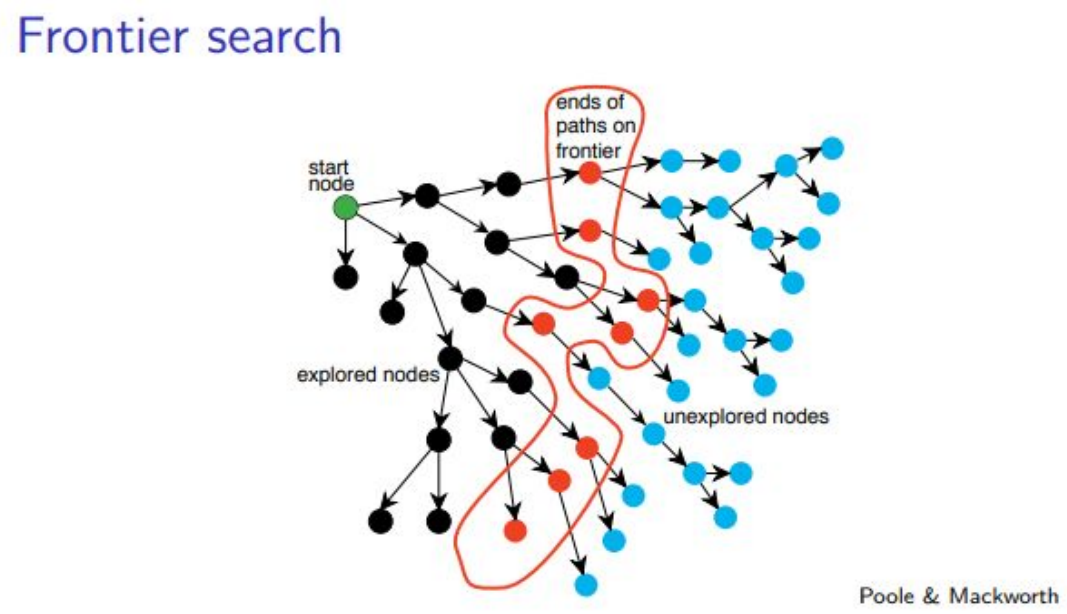
E.g.
```
Trans = [[q0,a,q0],[q0,b,q1],[q1,b,q1]]
Final = [q1]
Q0 = q0
```

Searching

The **forward branching factor** of a node is the number of outgoing arcs from the node. The **backward branching factor** of a node is the number of incoming arcs to the node.
Given goal, arc

```
search(Node) :- goal(Node).
search(Node) :- arc(Node,Next), search(Next)
```

Frontier Search




```
search(Node) :- frontierSearch([Node]).
frontierSearch([Node| ]) :- goal(Node).
frontierSearch([Node|Rest]) :-
                                    findall(Next, arc(Node,Next), Children),
                                    add2frontier(Children,Rest,NewFrontier),
                                    frontierSearch(NewFrontier).
```

Breadth-First

In **breadth-first search** the frontier is implemented as a **FIFO** (first-in, first-out) **queue**. Thus, the path that is selected from the frontier is the one that was added earliest.This method is guaranteed to find a solution if one exists and will find a solution with the fewest arcs.

```
add2frontier(Children,[],Children).
add2frontier(Children,[H|T],[H|More]) :- add2frontier(Children,T,More).
```

Breadth-first search is useful when:

- the problem is small enough so that space is not a problem (e.g., if you already need to store the graph) and
- you want a solution containing the fewest arcs.

It is a poor method when all solutions have many arcs or there is some heuristic knowledge available. It is not used very often for large problems where the graph is dynamically generated because of its exponential space complexity.

Depth-First

In **depth-first search**, the frontier acts like a **LIFO** (last-in, first-out) **stack** of paths. The algorithm selects a first alternative at each node, and it ***backtracks*** to the next alternative when it has pursued all of the paths from the first selection. Some paths may be infinite when the graph has cycles or infinitely many nodes, in which case a depth-first search **may never stop.** For depth-first search, the space used at any stage is linear in the number of arcs from the start to the current node.

```
add2frontier([],Rest,Rest).
add2frontier([H|T],Rest,[H|TRest]) :- add2frontier(T,Rest,TRest).
```

Depth-first search is appropriate when

- space is restricted
- many solutions exist, perhaps with long paths, particularly for the case where nearly all paths lead to a solution or
- the order in which the neighbors of a node are added to the stack can be tuned so that solutions are found on the first try.

It is a poor method when

- it is possible to get caught in infinite paths, which occurs when the graph is infinite or when there are cycles in the graph
- solutions exist at shallow depth, because in this case the search may explore many long paths before finding the short solutions, or
- there are multiple paths to a node, for example, on a n×n grid, where all arcs go right or down, there are exponentially paths from the top-left node, but only n^2 nodes.

Lowest-Cost-First-Search

Implemented by treating the frontier as a priority queue ordered by the cost function. If the costs of the arcs are all greater than a positive constant (bounded arc costs) and the branching factor is finite, the lowest-cost-first search is guaranteed to find an optimal solution – a solution with lowest path cost – if a solution exists.

The bounded arc cost is used to guarantee the lowest-cost search will

find a solution, when one exists, in graphs with finite branching factor. Without such a bound there can be infinite paths with a finite cost. For example, there could be nodes n0,n1,n2,…with an arc ⟨ni−1,ni⟩ for each i>0 with cost ½^i. Infinitely many paths of the form ⟨n0,n1,n2,…,nk⟩ all have a cost of less than 1. If there is an arc from n0 to a goal node with a cost equal to 1, it will never be selected. This is the basis of **Zeno's paradox** that Aristotle wrote about more than 2300 years ago.

Heuristics

A heuristic function h(n), takes a node n and returns a non-negative real number that is an estimate of the cost of the least-cost path from node n to a goal node. The function h(n) is an **admissible** heuristic if h(n) is always less than or equal to the actual cost of a lowest-cost path from node n to a goal.

h(Node) = estimate the minimum cost of a path from Node to a goal node
Prolog search where node = list of propositions to prove, and every arc costs 1

```
h(List) = length(List)
```

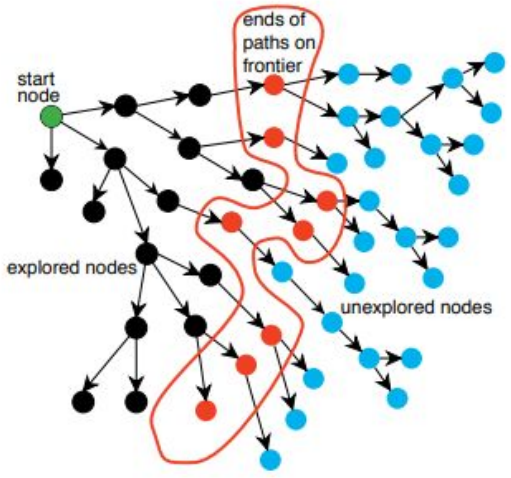
Greedy best-first search

Use a heuristic function to always select a path on the frontier with the lowest heuristic value. This method sometimes works well. However, it can follow paths that look promising because they appear (according to the heuristic function) close to the goal, but the path explored may keep getting longer.

A* Algorithm

A* **search** uses both path cost, as in lowest-cost-first, and heuristic information, as in greedy best-first search, in its selection of which path to expand.

A*



$$\text{solution} = \underbrace{\text{start} \cdots n}_{\text{explored}} \cdots \text{goal}$$

$$f(\text{start} \cdots n) = \text{cost}(\text{start} \cdots n) + h(n)$$

Ensure Frontier = [Head|Tail] where Head has minimal f

Admissibility

If there is a solution, A* using heuristic function h always returns an optimal solution, if:

- the branching factor is finite (each node has a bounded number of neighbors),
- all arc costs are greater than some ϵ>0, and
- h is an **admissible heuristic**, which means that h(n) is less than or equal to the actual cost of the lowest-cost path from node n to a goal node.

add code from assignment

A search algorithm is **complete** if it is guaranteed to find a solution if there is one.

Markov Decision Process

A Markov decision process can be seen as a Markov chain augmented with actions and rewards or as a decision network extended in time. At each stage, the agent decides which action to perform; the reward and the resulting state depend on both the previous state and the action performed.

a 5-tuple <S, A, p,r, γ> consisting of

- a finite set S of states s,s', . . .
- a finite set A of actions a, . . .
- a function p : S × A × S → [0, 1]
  p(s, a,s') = prob(s'|s, a) = how probable is s' after doing a at s
  Σp(s,a,s') =1 for all a $\in A, s \in S$
- a function r : S × A × S → R
  r(s, a,s') = immediate reward at s' after a is done at s
- a discount factor γ ∈ [0, 1]

A **policy** $\pi : S \rightarrow A$ specifies what the agent should do as a function of the state it is in. Given a reward criterion, a policy has an expected value for every state. Let V^{π}(s) be the expected value of following π in state s. This specifies how much value the agent expects to receive from following the policy in that state. Policy π is an **optimal policy** if there is no policy π' and no state s such that V^{π}'(s)> V^{π}(s). That is, it is a policy that has a greater or equal expected value at every state than any other policy.

$$q_0(s,a) := \sum_{s'} [p(s,a,s')r(s,a,s')]$$

$$q_{n+1}(s,a) := \sum_{s'} \left[p(s,a,s') \left(r(s,a,s') + \gamma max_{a'}\, q_n(s',a')\right)\right]$$

Value Iteration

Value iteration is a method of computing an optimal policy for an MDP and its value.

Value iteration starts at the "end" and then works backward, refining an estimate of either Q* or V*. There is really no end, so it uses an arbitrary end point.

Discounted Reward

V=r1+γr2+γ2r3+⋯+γi-1ri+⋯, where γ, the **discount factor**, is a number in the range 0≤γ<1. Under this criterion, future rewards are worth less than the current reward. If γ was 1, this would be the same as the total reward. When γ=0, the agent ignores all future rewards. Having 0≤γ<1 guarantees that, whenever the rewards are finite, the total value will also be finite.

An alternative yet identical q_{n+1}:

$$q_{n+1}(s,a) := q_0(s,a) + \sum_{s'} \left[p(s,a,s') * \gamma max_{a'}\, q_n(s',a')\right]$$

S-Deterministic:
An action is s-deterministic if $p(s,a,s') = 1$ for some s'.
i.e. Only one possible s' given s,a.

Absorbing:
A state s is absorbing if $p(s,a,s) = 1$ for every action a.
i.e. Can never leave this state.

Sink:
A state s is a sink if it is **absorbing** and $r(s,a,s) = m \ \forall a$
i.e. All actions in this absorbing state have the same reward.

S-drain:
An action a is an s-drain if for some sink s',
$p(s,a,s') = 1$ and $r(s,a,s') = m$

Where m = r(s', a, s')

Q-Learning

Q-Learning is a reinforcement-learning technique utilised in machine learning. It does **not** require a model of the environment. It builds its own model based on a series of trials until it converges to a solution. *Q*-learning and related RL algorithms, an agent tries to learn the optimal policy from its history of interaction with the environment.

$$Q_{n+1}(s,a) := (1-\alpha)\,Q_n(s,a) + \alpha\left[r' + \gamma\ max_{a'}Q_n(s',a')\right]$$

where $\alpha \in [0,1]$ is the learning rate (typically $\alpha \approx 0.2$).

Each iteration of q can be be captured as a matrix of dimensions $s \times a$.

Learning Algorithm:

1. Initialise the Q-matrix with zeros.
   Set a value for γ, the discount factor.
   Fill the R-matrix (environment rewards matrix).

2. For each trial, select a **random start state**.

3. Randomly select an action a from all possible actions for the current state s.This will later result in travelling to state s' as a result of action a.

4. From all possible actions from the state s', select the one with the highest Q-value.

5. Update the Q-table for state s and action a using:
   $$Q_{n+1}(s,a) := \alpha\left[r' + \gamma\ max_{a'}Q_n(s',a')\right] + (1-\alpha)\,Q_n(s,a)$$

6. Transition from state s to s' i.e. $s = s'$.

7. If at a **goal state** then end the trial.
   Else go to step 3.

The Q-matrix converges to a more-accurate solution with repeated trials.

Utilising the Q-Matrix:

1. Choose an initial state s.
2. For state s, select the action a in the Q-matrix with the highest Q-value.
3. Transition to state s' using action a. i.e. $s = s'$.
4. Go to step 2.

Exploration vs. Exploitation:
Exploration: We want to learn more. (Learning phase to update Q-matrix.)
Exploitation: We want to do the best we can. (Test the current Q-matrix.)

The explore–exploit dilemma: if an agent has worked out a good course of actions, should it continue to follow these actions (exploiting what it has determined) or should it explore to find better actions? An agent that never explores may act forever in a way that could have been much better if it had explored earlier.
The **ϵ-greedy exploratio**n strategy, where $0 \leq \epsilon \leq 1$, is the explore probability, is to select the greedy action (one that maximizes Q[s,a]) all but ϵ of the time and to select a random action ϵ of the time. It is possible to change ϵ through time. Intuitively, early in the life of the agent, it should act more randomly to encourage initial exploration and, as time progresses, it should act more greedily (reducing ϵ).

One problem with an ϵ-greedy strategy is that it treats all of the actions, apart from the best action, equivalently. If there are a few seemingly good actions and other actions that look much less promising, it may be more sensible to select among the good actions: putting more effort toward determining which of the promising actions is best, and less effort to explore the actions that look worse. One way to do that is to select action a with a probability depending on the value of Q[s,a]. This is known as a **soft-max** action selection.

An alternative is **optimism in the face of uncertainty**: initialize the Q-function to values that encourage exploration. If the Q-values are initialized to high values, the unexplored areas will look good, so that a greedy search will tend to explore. This does encourage exploration.

SARSA (so called because it uses state–action–reward–state–action experiences to update the Q-values) is an *on-policy* reinforcement learning algorithm that estimates the value of the policy being followed. An experience in SARSA is of the form $\langle s,a,r,s',a' \rangle$, which means that the agent was in state s, did action a, received reward r, and ended up in state s', from which it decided to do action a'. This provides a new experience to update Q(s,a). The new value that this experience provides is $r+\gamma Q(s',a')$.
SARSA is useful when deploying an agent that is exploring in the world(e.g a robot moving around that changes policies). If you want to do offline learning, and then use that policy in an agent that does not explore, *Q*-learning may be more appropriate.

Feasibility and non-determinism

Cobham's Thesis:
A problem is feasibly solvable iff some deterministic Turing machine solves it in polynomial time.

P = problems a deterministic TM solves in polynomial time
NP = problems a non-deterministic TM solves in polynomial time

Clearly $P \subseteq NP$.

Whether $P = NP$ is the most celebrated open mathematical problem in computer science.

- $P \neq NP$ would mean that non-determinism **wrecks** feasibility.
- $P = NP$ would mean that non-determinism makes **no difference** to feasibility.

P vs. NP Problem:
There are two forms of non-determinism:

• In don't-care non-determinism, if one selection does not lead to a solution, neither will other selections.
• In don't-know non-determinism, just because one choice did not lead to a solution does not mean that other choices will not. Often we speak of an **oracle** that could specify, at each point, which choice will lead to a solution. Because our agent does not have such an oracle, it has to search through the space of alternate choices.
The class P consists of decision problems solvable in time complexity polynomial in the size of the problem. The class NP, of nondeterministic polynomial time problems, contains decision problems that could be solved in polynomial time with an **oracle** that chooses the correct value at each choice in constant time or, equivalently, if a solution is verifiable in polynomial time. It is widely conjectured that **P≠NP**, which would mean that no such oracle can exist. One pivotal result of complexity theory is that the hardest problems in the NP class are all equally complex; if one can be solved in polynomial time, they all can. These problems are **NP-complete**. A problem is **NP-hard** if it is at least as hard as an NP-complete problem.

Does every problem whose solution can be **quickly verified** by a computer also be **quickly solved** be a computer?

If $P = NP$, then problems that can be verified in polynomial time can also be solved in polynomial time. However, If $P \neq NP$ then that would mean that there are problems in NP (such as NP-complete problems) that are harder to compute than to verify.

SAT (Boolean Satisfiability):
Given a Boolean expression φ with variables $x_1, \ldots, x_n$, can we make φ true by assigning true / false to $x_1, \ldots, x_n$?

Checking that a particular assignment makes φ is easy (P).
Non-determinism (guessing the assignment) puts SAT in NP.
But is SAT in P? There are 2^n assignments to try.

The belief that there is no algorithm to efficiently solve each SAT problem has not yet been proven mathematically, and resolving the question of whether SAT has a polynomial-time algorithm is equivalent to the P vs. NP problem.

Cook-Levin Theorem:
SAT is in P iff $P = NP$.

CSAT:
φ is a conjunction of clauses, where:

- A clause is an OR of literals.
- A literal is a variable x_i or negated variable $\overline{x_i}$.

k-SAT:
Every clause has exactly k literals.

3-SAT is as hard as SAT, 2-SAT is in P.

Horn-SAT: Every clause has at most one positive literal - linear.

<u>Constraint Satisfaction Problem</u>
A CSP is a mathematical problem defined as a set of objects whose state must satisfy a number of constraints and limitations. CSPs represent the entities of a problem as a homogenous collection of finite constraints over variables, which is solved by constraint satisfaction methods.

SAT, SMT and ASP can be thought of as certain forms of the CSP.
e.g. Eight queens, map colouring, Sudoku.

A **constraint satisfaction problem** (CSP) consists of:

- **Variables**: $V = \{V_1, \ldots, V_N\}$
- **Domain**: $D = \{\ldots\}$
  The set of d values that each variable can take e.g. $D = \{R, G, B\}$.
- **Constraints**: $C = \{C_1, \ldots, C_N\}$

Each constraint consists of a **tuple of variables** and a **list of values** that the tuple is allowed to take for this problem:

e.g. $\left[(V_1, V_2), \{(R,B),(R,G),(B,R),(B,G),(G,R),(G,B)\}\right]$

Constraints are usually defined implicitly → A function is defined to test if a tuple of variables satisfied the constraint.
e.g. $V_i \neq V_j$ for every edge (i,j).

Determining whether there is a model for a CSP with finite domains is NP-complete and no known algorithms exist to solve such problems that do not use exponential time in the worst case.

Generate-and-Test:
A simple algorithm that guarantees to find a solution if done so systematically, given that there exists a solution:

1. Generate a possible solution.
2. Test to see if this is the expected solution.
3. If the solution is not found, go back to step 1.

Interpretations and Logical Consequences

Datalog = a subset of Prolog *doesn't accept complex terms*
Datalog based on the following assumptions:

- An agent's knowledge can be usefully described in terms of individuals and relations among individuals.
- An agent's knowledge base consists of definite and positive statements.
- The environment is static.
- There are only a finite number of individuals of interest in the domain.
- An individual can be named.
- A variable starts with upper-case letter.
- A constant starts with lower-case letter or is a sequence of digits (numeral).
- A predicate symbol starts with lowercase letter.
- A term is either a variable or a constant.
- An atomic symbol (atom) is of the form p or p(t1, . . . ,tn) where p is a predicate symbol and ti are terms
- A definite clause is either an atomic symbol (a fact) or of the form:
    - a ← b1 ∧ · · · ∧ bm where a and bi are atomic symbols, a is the head and b's are the body
- query is of the form ?b1 ∧ · · · ∧ bm. knowledge base is a set of definite clauses.

Semantics

A semantics specifies the meaning of sentences in the language. The semantics of propositional calculus is defined below.

An **interpretation** consists of a function Π that maps atoms to {true, false}. If Π(a)=true, atom a is **true** in the interpretation, or the interpretation assigns true to a. If Π(a)=false, atom a is **false** in the interpretation.

An interpretation specifies:

- what objects (individuals) are in the world
- the correspondence between symbols in the computer and objects & relations in world
    - constants denote individuals
    - predicate symbols denote relations

An **interpretation** is a triple $I = \langle D, \phi, \pi \rangle$, where:

- D, the **domain**, is a non-empty set.
  Elements of D are **individuals**.
- ϕ is a **mapping** that assigns to each constant an element of D.
  Constant c denotes individuals $\phi(c)$.
- π is a mapping that assigns to each n-ary predicate symbol a relation:
  A function from D^n into $\{TRUE, FALSE\}$.

A **model** of a knowledge base KB is an interpretation in which all the propositions in KB are true.

Important Things to Note:

- The domain D can contain **real objects** (e.g. a person).
  D cannot necessarily be stored on a computer.
- $\pi(p)$ specifies whether the relation denoted by the n-ary predicate symbol p is true or false for each n-tuple of individuals.
- If the predicate symbol p has **no arguments**, then $\pi(p)$ is either $TRUE$ or $FALSE$.

Truth in an Interpretation:

- A constant c denotes in I the individual $\phi(c)$.
- A **ground atom (variable-free atom)** $p(t_1, ..., t_n)$ is:
    1. **True in interpretation** I if $\pi(p)(t'_1, ..., t'_n) = true$, where t_i denotes t'_i in interpretation I.
    2. **False** otherwise.
- A **ground clause** $h \leftarrow b_1 \wedge ... \wedge b_m$ is
    1. **False in interpretation** I if:
        a. h is false in I and...
        b. Each b_i is true in I
    2. **True in interpretation** I otherwise.

Models & Logical Consequences:

- A knowledge base KB is true in interpretation I iff **every clause** in KB is true in I.
- A **model** of a set of clauses is an interpretation in which **all the clauses are true**.
- If KB is a set of clauses, and g is a conjunction of atoms, then g is a **logical consequence** of KB (written $KB \models g$) if g is true in **every model** of KB.

The description of semantics does not tell us why semantics is interesting or how it can be used as a basis to build intelligent systems. The basic idea behind the use of logic is that, when a knowledge base designer has a particular world to characterize, the designer can choose that world as an intended interpretation, choose meanings for the symbols with respect to that world, and write propositions about what is true in that world. When the system computes a logical consequence of a knowledge base, the designer can interpret this answer with respect to the intended interpretation. A specification of meaning of the symbols is called an **ontology**.

User's View of Semantics:

1. Choose a task domain - **intended interpretation**.
2. Associate **constants** with individuals you want to name.
3. For each relation you want to represent, associate a **predicate symbol** in the language.
4. Tell the system clauses that are true in the intended interpretation - **axiomatising the domain**.
5. Ask questions about the intended interpretation.
6. If $KB \models g$, then g must be true for the intended interpretation.

Computer's View of Semantics:

- The computer doesn't have access to the intended interpretation, all it knows is the **knowledge base** KB.
  The computer cannot determine if a formula is a **logical consequence** of KB.
- If $KB \models g$, then g must be true for the intended interpretation.
- If $KB \not\models g$, then there is a model of KB in which g is false. This could be the intended interpretation.

Proofs:

- A **proof** $\vdash$ is a **mechanical procedure** for deriving a formula g from a knowledge base KB, written $KB \vdash g$.
- Recall that $KB \models g$ means that g is true in **all models** of KB.
- $\vdash$ is **sound** if $KB \models g$ whenever $KB \vdash g$. $(KB \vdash g \Rightarrow KB \models g)$
- $\vdash$ is **complete** if $KB \vdash g$ whenever $KB \models g$. $(KB \models g \Rightarrow KB \vdash g)$

Bottom-Up Ground Proof Procedure:

A **bottom-up proof procedure** can be used to derive all logical consequences of a knowledge base. It is called bottom-up as an analogy to building a house, where each part of the house is built on the structure already completed. The bottom-up proof procedure builds on atoms that have already been established. It should be contrasted with a top-down approach, which starts from a query and tries to find definite clauses that support the query. Sometimes we say that a bottom-up procedure is **forward chaining** on the definite clauses, in the sense of going forward from what is known rather than going backward from the query.

A one-rule derivation, a generalised form of *modus ponens*:
If $h \leftarrow b_1 \vee \ldots \vee b_m$ is a clause in the knowledge base, and each b_i has been derived, then h can be derived. This holds when $m = 0$.

Derives exactly the atoms that logically follow from KB.

- Sound
- Complete

It is efficient.

In mathematical logic, a **Herbrand interpretation** is an interpretation in which all constants and function symbols are assigned very simple meanings.[1] Specifically, every constant is interpreted as itself, and every function symbol is interpreted as the function that applies it.

The definite clause language does not allow a contradiction to be stated. However, a simple expansion of the language can allow proof by contradiction.

Horn clauses and negations

Horn Clause:
Either a **definite clause** or an **integrity clause**. Horn clause has either false or a normal atom as its head.

Negations ($KB \models \neg c$) and **disjunctions** ($KB \models \neg c \vee \neg d$) can follow from KBs containing Horn clauses.

Integrity Constraints:
A clause of the form $false \leftarrow a_1 \vee \ldots \vee a_m$.

Definite Clauses:
Any regular clause (prolog, less cuts) such as $r \leftarrow a_1 \vee \ldots \vee a_m$.
A **knowledge base** is a set of definite clauses.
In general, a definite clause is equivalent to a clause with exactly one positive literal.

Satisfiable:
A formula is:

- **Satisfiable** iff it has a model.
- **Unsatisfiable** iff it has no models.
- **Counter-satisfiable** iff its negation has a model.
- **Valid** iff every interpretation is a model (i.e. a tautology).
- **Logical consequence** of an axiom set if every model of the axiom set is also a model of the formula.

A set of clauses is **unsatisfiable** if it has no models. A set of clauses is provably **inconsistent** with respect to a proof procedure if false can be derived from the clauses using that proof procedure. If a proof procedure is sound and complete, a set of clauses is provably inconsistent if and only if it is unsatisfiable.

It is always possible to find a model for a set of definite clauses. The interpretation with all atoms true is a model of any set of definite clauses. Thus, a definite-clause knowledge base is always satisfiable. However, a set of Horn clauses can be unsatisfiable.

Complete-Knowledge Assumption (CKA):
The **complete knowledge assumption** assumes that, for every atom, the clauses with the atom as the head cover all the cases when the atom is true. Under this assumption, an agent can conclude that an atom is false if it cannot derive that the atom is true. This is also called the **closed-world assumption**. It can be contrasted with the **open-world assumption**, which is that the agent does not know everything and so cannot make any conclusions from a lack of knowledge. The closed-world assumption requires that everything relevant about the world is known to the agent.

Closed World Assumption: any unprovable atom ϕ is false (true : $\neg\phi$) / $\neg\phi$
Negation as failure: ϕ is false if attempting to prove ϕ fails finitely.
N.B. Checking finite failure can be as hard as the Halting Problem.

Monotonicity vs. Non-Monotonicity:
A definite clause is **monotonic** if adding clauses does not invalidate a previous conclusion.
With the complete-knowledge assumption, the system is **non-monotonic**: A conclusion can be invalidated by adding more clauses.

Negation-as-failure:
With the completion, the system can derive negations, and so it is useful to extend the language to allow negations in the body of clauses. A **literal** is either an atom or the negation of an atom. The definition of a definite clause can be

extended to allow literals in the body rather than just atoms. We write the negation of atom a under the complete knowledge assumption as ~a to distinguish it from classical negation that does not assume the completion. This negation is often called **negation as failure**.

Negation-as-failure leads to non-monotonicity.

A **default** is a rule that can be used unless it overridden by an exception.

Rules

$$\frac{\textit{prerequisite } p : \textit{justification } j}{\textit{conclusion } c}$$

$$(\star)\ \frac{\texttt{bird}(X)\ :\ \texttt{fly}(X)}{\texttt{fly}(X)}$$

Let *KB* be

```
bird(robin).
bird(penguin).
false :- fly(penguin).
fly(bee).
```

Conclude:

`fly(robin)` by default rule $(\star)$

but *not* `fly(penguin)`.

An explanation of `fly(bee)` using $(\star)$ is

```
bird(bee)
```

which we can block by adding to *KB* the rule

```
false :- bird(bee).
```

A default rule is normal if its justification is its conclusion (p : c)/c

Abduction, Deduction & Induction:
Abduction and deduction are similar processes, however they start from opposite ends of a proof.

1. **Abduction** starts with statements we know to be true in a given knowledge base, and seeks to find the simplest clauses that make the statement true. (Finding an explanation for the statement, in Tim's words.)
2. **Deduction** starts with a knowledge base, and seeks to deduce as many things as possible.
3. **Inductive reasoning** is reasoning in which the premises are viewed as supplying strong evidence for the truth of the conclusion. While the conclusion of a deductive argument is certain, the truth of the conclusion of an inductive argument in **probable**, based upon the evidence given.

There is debate over whether abductive and inductive reasoning are different.
A knowledge base KB is consistent iff its negation is not a tautology i.e. if there exists a model.

Soundness:
$KB \vdash g \Rightarrow KB \models g$

1. Suppose there is a g such that $KB \vdash g$ and $KB \not\models g$.
2. Let h be the first atom added to C that is not true in every model of KB. Suppose h is not true in model I of KB.
3. There must be a clause in KB of the form:
   $h \leftarrow b_1 \vee \ldots \vee b_m$
4. Each b_i is true in I. So this clause is false in I.
5. Therefore I is not a model of KB.
   Contradiction: No such g exists.

Conditional Probability and Independence

From a Constraint Satisfaction Problem [Var,Dom,Con] to **random variables** with **probabilities** constrained by a graph.
The **domain** (range) of a variable X, written Dom(X), is the set of (possible) values X can take.
A **proposition** α is an equation X = x between a variable X and a value x ∈ Dom(X), or a Boolean combination of such.
A proposition α is assigned a probability through

- a notion |= of a possible world ω **satisfying** α, and
- a **measure** μ for weighing a set of possible worlds.

Conditional Probability and other things similar to Stats + Bayes theorem + independence.

The conditional probability of α' given α is

$$P(\alpha' \mid \alpha) := \frac{P(\alpha' \text{ and } \alpha)}{P(\alpha)}$$

+ Bayes theorem

X **is independent of** Y given Z, written X ⊥⊥ Y | Z,
P(X = x | Y = y ∧ Z = z) = P(X = x | Z = z)

Belief network
A **belief network** gives a probability distribution over a set of random variables.
A belief network consists of:

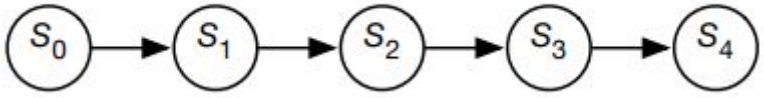
- a directed acyclic graph with nodes = random variables, and

- an arc from the parents of each node into that node
- a domain for each random variable
- conditional probability tables for each variable given its parents respecting (†)

A **Markov chain** is a belief network with random variables in a sequence, where each variable only directly depends on its predecessor in the sequence. Markov chains are used to represent sequences of values, such as the sequence of states in a dynamic system or the sequence of words in a sentence. Each point in the sequence is called **stage**.
Stationary Markov chains are of interest for the following reasons:

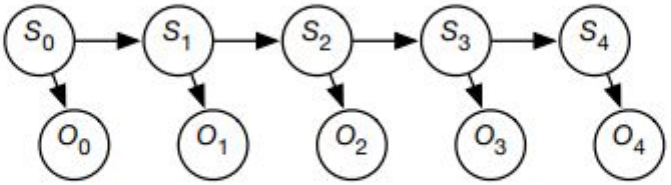
- They provide a simple model that is easy to specify.
- The assumption of stationarity is often the natural model, because the dynamics of the world typically does not change in time. If the dynamics does change in time, it is usually because of some other feature that could also be modeled.
- The network extends indefinitely. Specifying a small number of parameters gives an infinite network. You can ask queries or make observations about any arbitrary points in the future or the past.

In a stationary model: $P(S_{i+1}|S_i) = P(S_1|S_0)$ for all $i \geq 0$



$P(S_0)$ specifies initial conditions $P(S_{t+1}|S_t)$ specifies the dynamics
$P(S_{t+1}|S_{0:t}) = P(S_{t+1}|S_t)$
S_t represents the state at time t, capturing everything about the past ($< t$) that can affect the future ($> t$). => The future is independent of the past given the present.

A Hidden Markov Model (HMM) is a belief network of the form



- $P(S_0)$ specifies initial conditions
- $P(S_{i+1}|S_i)$ specifies the dynamics
- $P(O_i|S_i)$ specifies the sensor model

Pagerank

Google's initial search engine was based on **Pagerank**. Pagerank is a probability measure over web pages where the most influential web pages have the highest probability. It is based on a Markov chain of a random web surfer who starts on a random page, and with some probability d picks a random page that is linked from the current page, and otherwise (if the current page has no outgoing links or with probability 1−d) picks a page at random.

Markov networks

A Markov network or MRF is similar to a Bayesian network in its representation of dependencies; the differences being that Bayesian networks are directed and acyclic, whereas Markov networks are undirected and may be cyclic. Thus, a Markov network can represent certain dependencies that a Bayesian network cannot.

In statistics and machine learning, the **Markov blanket** for a node in a graphical model contains all the variables that shield the node from the rest of the network. This means that the Markov blanket of a node is the only knowledge needed to predict the behavior of that node and its children.

The **Hammersley–Clifford theorem** is a result in probability theory, mathematical statistics and statistical mechanics, that gives necessary and sufficient conditions under which a strictly positive probability distribution can be represented as a Markov network (also known as a Markov random field). It is the **fundamental theorem of random fields**.

Markov Properties

- **Pairwise Markov property**: Any two non-adjacent variables are conditionally independent given all other variables
- **Local Markov property**: A variable is conditionally independent of all other variables given its neighbors
- **Global Markov property**: Any two subsets of variables are conditionally independent given a separating subset