### Introduction

**The Eliza Effect**:  
Humans are inclined to see computers as humans.

**AI Complete**:  
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).

**Moving Target**:

Changing agent and environment (e.g. change in state).

**Tolerance**:

A unary relation is tolerant up to if:

**Sortes Chains**:

A sequence such that holds of but **not** , even though for .

**Cantor’s Theorem**:

For any set, the set of all subsets (the power set) has a greater cardinality that itself.

**Turing Machine**:

An abstract machine that manipulates symbols on a strip of tape according to a table of rules.

An agent can be modelled as a Turing machine where the tape is the environment.

**The Halting Problem**:

Given a program and data , return either 0 or 1, with 1 indicating that halts on input .

The halting problem is undecidable over Turing machines.

The proof is similar to that of the Liar’s paradox: ,

**Semi-Solvability**:

Can we solve for the positive part of the HP?

**Universal Turing Machine**:

A Turing machine that simulates other Turing machines. The input to the UTM is a description a Turing machine and an input for , and the UTM simulates on that input.

A universal Turing machine can be used to meet the positive part of the halting problem (semi-solvability) i.e. looping when the HP asks for 0.

**Non-Determinism**:

A nondeterministic algorithm is an algorithm that, even for the same input, can exhibit **different behaviours** on different runs.

* Symbol manipulation is action, but nondeterminism is the fact that intelligent action requires making choices.
* Paradox of intelligent AI is that intelligence requires choices, but that computation is deterministic.

() is an agent acting intelligently in its environment.

### Finite State Machines

**Church-Turing Thesis**:

A hypothesis about the nature of computable functions. It states that a function on the natural numbers is computable by a human being following an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine.

Anything that can be computed can be computed by a Turing machine.

**Finite State Machines**:

A finite state machine is a triple where:

* is a list of triples such that may, at state and seeing symbol , transition to state .
* is a list of ’s final (accepting) states.
* is ’s initial state.

### Searching

**Search Graphs**:

A graph consists of a set of **nodes**, and a set of ordered pairs of nodes called **arcs**.

* is a **neighbour** of if there is an arc from to i.e. .
* A **path** is a sequence such that .
* Given a set of start nodes and goal nodes, a **solution** is a path from a start node to an end node.

**Graph Searching**:

Generic searching algorithm:

1. Given a graph, start nodes and goal nodes, incrementally explore paths from the start nodes.
2. Maintain a **frontier** of paths from the start node that have been explored.
3. As the search proceeds, this frontier expands into unexplored nodes until a **goal node** is encountered.
4. After the search algorithm returns an answer, it can be asked for **more answers** and the procedure continues.

The way in which the frontier expands defines the **search strategy**.

search(Node) :- goal(Node).

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

**Depth-First Search**:

The frontier is treated as a stack.

Determinised arc(Node, Next) to arcD(NodeList, NextList)

goalD(NodeList) :- member(Node, NodeList), goal(Node).

arcD(NodeList, NextList) :- setof(Next, arcLN(NodeList, Next), NextList).

arcLN(NodeList, Next) :- member(Node, NodeList), arc(Node, Next).

**Bounded**:

search(Node, s(B)) :- goal(Node).

search(Node, s(B)) :- arc(Node, Next), search(Next, B).

**Iterative Deepening**:

iterativeSearch(Node) :- bound(B), search(Node, B).

bound(0).

bound(s(B)) :- bound(B).

**DFS Complexity**:

* DFS is **not** guaranteed to halt i.e. infinite graphs / graphs with cycles.
* The space complexity is linear in the size of the path being explored.
* The search is unconstrained by the goal until it happens to stumble on the goal.

**Breadth-First Search**:

The frontier is treated as a queue.

goalL(H|T) :- goal(H) ; goalL(T).

arcL(NodeList, NextList) :-

findAll(X, (member(N, NodeList), arc(N, X)), NextList).

**BFS Complexity**:

* The **branching factor** of a node is the **number of its neighbours**.
* If the branching factor for all nodes is **finite**, BFS is guaranteed to find a solution.
* It is guaranteed to find the path with **fewest arcs**.
* Time-complexity is exponential in the path length ().
* The space-complexity is exponential in path length ().
* Search is unconstrained by the goal.

**Lowest-Cost-First Search**:

* A **cost** can be associated with an arc.
* The cost of a **path** is the sum of the cost of its arcs.
* The frontier is a **priority queue** ordered by path cost.
* It finds a **least-cost** path to a goal node.

**Heuristic Search**:

*Idea*: Don’t ignore the goal when selecting paths:

* is an estimate of the cost of the shortest path from node to a goal node.
* uses only readily-obtainable information that is easy to compute about a node.
* can be extended to paths .
* is an **underestimate** if there is **no** path from to a goal that has a path length less than .

**Best-First Search**:

Select the path whose end is closest to a goal according to the **heuristic function** using a **priority queue**. Select the lowest -value (minimal -value).

**Complexity**:

* Uses space exponential in path length.
* Not guaranteed to find a solution, even if one exists.
* Does not always find the shortest path.

**Heuristic DFS**:

* Order the neighbours of a node by before adding them to the front of the frontier.
* Not guaranteed to find a solution.

**A\* Search**:

Uses both **path cost** and **heuristic values**:

* is the cost of the path .
* estimate the cost from the end of to a goal.
* estimates the total path cost from a start node to a goal via .

Is a mix of **lowest-cost-first** and **best-first-search**:

* Treats the frontier as a **priority queue** ordered by .
* Always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.

**Admissibility**:  
If there is a solution, A\* always finds an optimal solution if:

1. The branching factor is **finite**.
2. Arc costs are bounded **above zero**.
3. is an **underestimate** of the length of the **shortest** path from to a goal node.

**Don’t Care Nondeterminism**:

The choice can be made arbitrarily. In terms of the logic programming computation model, any goal reduction will lead to a solution, and it does not matter which particular solution is found.

**Don’t Know Nondeterminism**:

The choice matters, but correct one is not known at the time the choice is made.

### Feasibility & Non-Determinism

**Cobham’s Thesis**:

A problem is **feasibly solvable** iff some deterministic Turing machine solves it in polynomial time.

Clearly .

Whether is the most celebrated open mathematical problem in computer science.

* would mean that non-determinism **wrecks** feasibility.
* would mean that non-determinism makes **no difference** to feasibility.

**P vs. NP Problem**:

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

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

**SAT (Boolean Satisfiability)**:

Given a Boolean expression with variables , can we make true by assigning true / false to ?

Checking that a particular assignment makes is easy ().

Non-determinism (guessing the assignment) puts SAT in .

But is SAT in ? There are 2n 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 vs. problem.

**Cook-Levin Theorem**:

SAT is in iff .

**CSAT**:

is a conjunction of clauses, where:

* A clause is an OR of literals.
* A literal is a variable or negated variable .

**k-SAT**:

Every clause has exactly literals.

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

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

**Unifier**:

A substitution is a unifier of expressions and if and are identical.

That is, a unifier of two expressions is a subtraction that when applied to each expression results in the same expression.

### Constraint Satisfaction Problem (CSP)

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.

* **Variables**:
* **Domain**:   
  The set of values that each variable can take e.g. .
* **Constraints**:

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.

Constraints are usually defined implicitly ➝ A function is defined to test if a tuple of variables satisfied the constraint.

e.g. for every edge .

**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.

### Knowledge Bases

A **semantics** specifies the meaning of sentences in the language.

An **interpretation** specifies:

* What objects / individuals are in the world.
* The **correspondence between symbols** in the computer and objects & relations in the world.
  + **Constants** denote **individuals**.
  + **Predicate symbols** denote **relations**.

An **interpretation** is a triple , where:

* , the **domain**, is a non-empty set.  
  Elements of are **individuals**.
* is a **mapping** that assigns to each constantan element of .  
  Constant denotes individuals .
* is a mapping that assigns to each n-ary predicate symbol a relation:  
  A function from into .

If an interpretation assigns the value to a sentence, then the interpretation is a **model** of that sentence.

**Important Things to Note**:

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

**Truth in an Interpretation**:

* A constant denotes in the individual .
* A **ground atom (variable-free atom)**  is:

1. **True in interpretation**  if , where denotes in interpretation .
2. **False** otherwise.

* A **ground clause**  is

1. **False in interpretation** if:
   1. is false in and...
   2. Each is true in
2. **True in interpretation**  otherwise.

**Models & Logical Consequences**:

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

**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 , then 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** .  
  The computer cannot determine if a formula is a **logical consequence** of .
* If , then must be true for the intended interpretation.
* If , then there is a model of in which is false. This could be the intended interpretation.

**Proofs**:

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

**Bottom-Up Ground Proof Procedure**:

A one-rule derivation, a generalised form of *modus ponens*:

If is a clause in the knowledge base, and each has been derived, then can be derived. This holds when .

This is **forward-chaining** on this clause.

Derives exactly the atoms that logically follow from .

* Sound
* Complete

It is efficient.

**Top-Down Proof Procedure**:

Starts with a query to determine if it is a logical consequence of .

It starts with a query and attempts to reduce it to an answer by replacing the with their bodies in .

There is a 1:1 correspondence between top-down and bottom-up proofs:

* Sound
* Complete

**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.

**Horn Clause**:

Either a **definite clause** or an **integrity clause**.

**Negations** () and **disjunctions** () can follow from s containing Horn clauses.

**Integrity Constraints**:

A clause of the form .

**Definite Clauses**:

Any regular clause (prolog, less cuts) such as .

**Conflicts**:

A conflict of a is a set of assumables (an atom that can be assumed in a proof by contradiction) that make it imply false.

A **minimal conflict** is a conflict such that no subset of it is also a conflict.

There can be multiple minimal conflicts with different numbers of assumables.

Example where are the assumables:

|  |
| --- |
| .  .  .  . |

Either is , or is in every model of .

If they were both in some model of , then the first clause would be in , therefore a contraction to being a model of .

* and are minimal conflicts.
* is also a conflict, but not a minimal one.

**Complete-Knowledge Assumption (CKA)**:

Assume that a database of facts contains all possible information about the system modelled. All facts **not listed** are assumed to be false. This is called the **closed-world assumption**.

In the **open-world assumption**, the agent does not know everything and so cannot make any conclusions from a lack of knowledge.

**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**:

A non-monotonic inference rule. Used to derive from failure to derive .

**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.

**Consistency**:

A knowledge base is consistent iff its negation is not a tautology i.e. if there exists a model.

**Soundness**:

1. Suppose there is a such that and .
2. Let be the first atom added to that is not true in every model of .  
   Suppose is not true in model of .
3. There must be a clause in of the form:
4. Each is true in . So this clause is false in .
5. Therefore is not a model of .  
   **Contradiction**: No such exists.

**Fixed Point**:

The generated at the end of the bottom-up algorithm is called a fixed point.

Let be the interpretation in which every element of the fixed point is true and every other atom is false.

is a model of .

Proof:

1. Suppose in is false in .  
   Then is false, and each is true in .
2. Thus can be added to .
3. Contradiction to being the fixed point.

is called a **minimal model**.

**Completeness**:

1. Suppose .  
   Then is true in all models of .
2. Thus is true in the minimal model.
3. Thus is generated by the bottom-up algorithm.
4. Thus .

### Markov Decision Process

A 5-tuple consisting of:

1. A finite set of **states**
2. A finite set of **actions**
3. A function :   
    = How likely is after doing at .
4. A function :   
    = Immediate reward at after is done at .
5. A discount factor

A policy : tells us what action to take .

is the optimal policy.

An alternative yet identical :

**S-Deterministic**:

An action is s-deterministic if for some .

i.e. Only one possible given .

**Absorbing**:

A state is absorbing if for every action .

i.e. Can never leave this state.

**Sink**:

A state is a sink if it is **absorbing** and

i.e. All actions in this absorbing state have the same reward.

**S-drain**:

An action is an s-drain if for some sink ,

and

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.

<http://mnemstudio.org/path-finding-q-learning-tutorial.htm>

where is the learning rate (typically ).

Each iteration of can be be captured as a matrix of dimensions .

**Learning Algorithm**:

1. Initialise the Q-matrix with zeroes.  
   Set a value for , the discount factor.  
   Fill the R-matrix (environment rewards matrix).
2. For each trial, select a **random start state**.
3. Randomly select an action from all possible actions for the current state .  
   This will later result in travelling to state as a result of action .
4. From all possible actions from the state , select the one with the highest Q-value.
5. Update the Q-table for state and action using:
6. Transition from state to i.e. .
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 .
2. For state , select the action in the Q-matrix with the highest Q-value.
3. Transition to state using action . i.e. .
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.)

**SARSA**:

Replace by policy in use.