AIES — Unit II Notes

Game Playing and Knowledge Representation

Table of Contents

1. Game Playing — Introduction
    - Game Tree
2. Minimax Algorithm
3. Alpha-Beta Cutoffs
4. Propositional Logic
5. Predicate Logic (First-Order Logic)
6. Knowledge Representation Structures
    - Frames
    - Conceptual Dependencies (CD)
    - Semantic Networks
    - Scripts
7. Inference — Resolution in Predicate Logic
    - Horn Clauses
    - Unification
    - Forward Chaining
    - Backward Chaining

---

1. Game Playing — Introduction

1. Game playing is a classic AI problem because it provides a well-defined, measurable environment to study **adversarial reasoning** where agents compete against each other with conflicting goals.
2. A game in AI is formalised by six components — initial state, player function (who moves), action function (legal moves), result function (transitions), terminal test, and utility (payoff) function.
3. The **utility function** gives the final numeric value of the game for each player — for chess it is +1 for a win, -1 for a loss, and 0 for a draw.
4. In a **zero-sum game**, one player's gain is exactly equal to the other player's loss — chess and tic-tac-toe are classic zero-sum games.

5. Games are categorised by information (perfect vs imperfect), determinism (deterministic vs stochastic), and number of players (single vs multi-agent).

1a. Game Tree

1. A game tree is a tree where **nodes represent game states** and **edges represent moves** made by players, starting from the initial state at the root.
2. **MAX** is the computer player trying to maximise the utility value, and **MIN** is the opponent trying to minimise it — both players alternate turns.
3. At each level called a **ply**, one player makes a move — MAX plays on even-depth levels and MIN plays on odd-depth levels.
4. **Leaf nodes** of the game tree are terminal states evaluated by the utility function — these concrete payoff values are then propagated back up the tree.
5. The entire game tree for tic-tac-toe shows all 9 possible first moves for MAX and all subsequent opponent responses, covering every reachable game state.

---

2. Minimax Algorithm

1. Minimax provides **perfect play** for deterministic, perfect-information, two-player zero-sum games by selecting the move with the highest guaranteed utility against an optimal opponent.
2. At **MAX nodes**, the algorithm selects the child with the **maximum value** — at **MIN nodes**, it selects the child with the **minimum value**, reflecting each player's goal.
3. Minimax follows a **depth-first search** traversal of the game tree, exploring all the way to terminal nodes before returning values upward to the root.
4. **Time complexity is O(b^m)** where b is the branching factor and m is the maximum depth — this makes it very slow for complex games like chess.
5. **Space complexity is O(bm)** since the algorithm uses depth-first search and only needs to store nodes along the current path.

Minimax Value Definition:

- MINIMAX(n) = UTILITY(n) → if n is a terminal state
- MINIMAX(n) = max(MINIMAX(successor)) → if n is a MAX node
- MINIMAX(n) = min(MINIMAX(successor)) → if n is a MIN node

Property	Value
Completeness	Yes — always finds solution in finite tree
Optimality	Yes — when both players play optimally
Time Complexity	O(b^m)
Space Complexity	O(bm)

---

3. Alpha-Beta Cutoffs

1. Alpha-beta pruning is an optimisation of Minimax that **eliminates branches** of the game tree that cannot possibly affect the final decision, allowing the search to go twice as deep in the same time.
2. It maintains two values — **alpha** (best value found so far for MAX along the current path, starts at −∞) and **beta** (best value found so far for MIN, starts at +∞).
3. The **pruning condition is α ⩾ β** — when the alpha of a MAX node is greater than or equal to the beta of an ancestor MIN node, all remaining children of that node can be safely skipped.
4. Alpha-beta pruning returns **exactly the same optimal move** as full Minimax — pruning only removes nodes that cannot change the final decision.
5. With **perfect move ordering** (best moves explored first), the effective branching factor is reduced to b^(m/2), doubling the search depth achievable in the same time.

Move Ordering	Time Complexity	Description
Worst ordering	O(b^m)	No pruning occurs — same as full Minimax
Ideal ordering	O(b^(m/2))	Maximum pruning — best moves always explored first

How to draw Alpha-Beta on a tree:

- Write α and β values at each node
- When α ⩾ β at any node, cross out (prune) all remaining children of that node
- Propagate values upward — MAX nodes take max of children, MIN nodes take min of children

---

4. Propositional Logic

1. Propositional logic is a symbolic system for manipulating **declarative statements** (propositions) that are either true or false using well-defined connectives.
2. A **proposition** is a declarative sentence whose truth value can be determined — "it is raining" is a proposition but "is it raining?" is not.
3. The logical connectives are **∧ (AND), ∨ (OR), ¬ (NOT), → (implication), ≡ (equivalence)** — used to build compound statements from individual propositions.
4. Propositional logic is **declarative** and allows partial, disjunctive, and negated information — but it cannot talk about individuals or relationships between objects.
5. A key limitation is that propositional logic **cannot express generalisations** like "all humans are mortal" — predicate logic is needed for statements about individuals.

Key Inference Rules:

Rule	Form	Meaning
Modus Ponens	P → Q, P ⊢ Q	If P implies Q and P is true, then Q is true
De Morgan's	¬(P ∨ Q) ≡ ¬P ∧ ¬Q	Negation distributes over connectives
Implication Elimination	P → Q ≡ ¬P ∨ Q	An implication can be rewritten as a disjunction
Double Negation	¬(¬P) ≡ P	Double negation cancels out
Contrapositive	P → Q ≡ ¬Q → ¬P	Equivalent form of an implication

---

5. Predicate Logic (First-Order Logic)

1. Predicate logic (FOL) extends propositional logic by adding **objects, properties, relations, functions, variables, and quantifiers**, making it far more expressive.
2. FOL models the world in terms of **objects** (students, countries), **properties** (blue, tall), **relations** (brother-of, loves), and **functions** (father-of, mother-of).
3. The **universal quantifier ∀** means "for all" — for example ∀x: dolphin(x) → mammal(x) means "every dolphin is a mammal" — typically used with implication (→).
4. The **existential quantifier ∃** means "there exists some" — for example ∃x: mammal(x) ∧ lays-eggs(x) means "there exists a mammal that lays eggs" — typically used with conjunction (∧).
5. Switching the order of two universal quantifiers or two existential quantifiers does not change the meaning — but switching a universal and an existential quantifier **does** change

the meaning.

Feature	Propositional Logic	Predicate Logic (FOL)
Talks about	True/false statements only	Objects, relations, and functions
Variables	No variables	Yes — x, y, z
Quantifiers	No quantifiers	Yes — ∀ and ∃
Expressive power	Low	High
Example	$P \land Q \rightarrow R$	$\forall x: man(x) \rightarrow mortal(x)$

Exam-Style FOL Conversions (Marcus/Caesar):

English Sentence	Predicate Logic Form
Marcus was a man	man(Marcus)
Marcus was a Pompeian	pompeian(Marcus)
All Pompeians were Romans	$\forall x: pompeian(x) \rightarrow roman(x)$
Caesar was a ruler	ruler(Caesar)
All Romans were loyal to Caesar or hated him	$\forall x: roman(x) \rightarrow loyalTo(x, Caesar) \lor hate(x, Caesar)$
Everyone is loyal to someone	$\forall x: \exists y: loyalTo(x, y)$
Marcus tried to assassinate Caesar	tryAssassinate(Marcus, Caesar)

6. Knowledge Representation Structures

6a. Frames

1. A frame, introduced by **Marvin Minsky in 1970**, is a data structure that represents a concept or object as a collection of **slots (attributes)** and their associated **values (fillers)**.
2. Frames differ from semantic networks in that they can contain **procedural knowledge** embedded as values — not just factual associations between concepts.
3. A frame system represents an organised collection of related frames linked by **specialisation and generalisation relationships**, similar to class hierarchies in object-

oriented programming.

4. Slots in a frame can have **default values** that apply unless overridden — for example a "Bird" frame may have "can fly = yes" as a default, overridden by a "Penguin" frame.
5. Frames are easy to understand and extend — they are widely used in natural language processing and machine vision, and they inspired the concept of classes and objects in modern programming.

Frame Structure Example — Person:

```
Frame: Person
  Name:       [value]
  Age:        [value]
  Profession: [value]
  City:       [value]
```

6b. Conceptual Dependencies (CD)

1. Conceptual Dependency theory was developed by **Roger Schank (1973-1975)** to represent the meaning of natural language sentences in a **language-independent** way.
2. CD uses a set of **conceptual primitives** such as ATRANS, PTRANS, MTRANS, MOVE, INGEST, and SPEAK rather than actual words to capture the underlying meaning.
3. The key advantage is that **different sentences with the same meaning** produce the same CD representation — "I gave the man a book" and "The man got a book from me" map to the same structure.
4. CD provides four conceptual categories — **ACT** (actions), **PP** (objects/picture producers), **AA** (modifiers of actions), and **PA** (modifiers of objects).
5. CD representations help in drawing inferences from natural language because the canonical primitive structure makes it easy to apply inference rules across different phrasings.

Primitive	Meaning	Example
ATRANS	Transfer of abstract relationship	Give, take
PTRANS	Transfer of physical location	Go, move
MTRANS	Transfer of mental information	Tell, inform
MBUILD	Building new information from old	Decide, imagine
INGEST	Ingesting an object	Eat, drink
SPEAK	Producing sounds	Say, shout

6c. Semantic Networks

1. A semantic network is an **irregular graph** where nodes represent concepts (objects, events, situations) and arcs represent the relationships between those concepts.
2. Common standard relations include **IS-A** (inheritance), **A-KIND-OF (AKO)**, **an instance of**, and **part-of** — though custom ad-hoc relations can also be defined.
3. Semantic networks are **easy to understand and visually transparent** — they model knowledge in a way that resembles how human memory associates related concepts.
4. A key limitation is that semantic networks **lack standard definitions** for link names, do not support quantifiers like "for all" or "there exists", and can be slow at runtime.
5. Despite limitations, semantic networks convey meaning naturally and are easy to extend by simply adding new nodes and arcs for new knowledge.

Example Semantic Network (Jerry the Cat):

```
Jerry → IS-A → Cat
Cat → IS-A → Mammal
Mammal → IS-A → Animal
Jerry → has-color → White
Jerry → owned-by → Priya
Jerry → lives-in → House
```

(In exam: draw boxes for each node and labeled arrows for each relationship)

6d. Scripts

1. A script is a structured representation of a **stereotypical sequence of events** in a particular

context — it describes what typically happens in a familiar situation like visiting a restaurant.

2. Scripts are closely related to frames but focus on **event sequences** — a restaurant script includes events like entering, being seated, ordering, eating, paying, and leaving.
3. Scripts allow an AI system to **fill in missing details** by assuming the default sequence of events even when not explicitly stated in the input text.
4. A script has five components — **Entry Conditions** (what must be true before), **Results** (what is true after), **Props** (objects involved), **Roles** (participants), and **Scenes** (temporal event sequence).
5. Scripts are useful in **natural language understanding** because they allow reasonable inferences about what happened in a story without being told every detail explicitly.

Component	Description	Restaurant Script Example
Entry Condition	Must be true before script applies	Customer is hungry, restaurant is open
Result	True once the script ends	Customer is not hungry, has less money
Props	Objects involved	Tables, menus, food, bill
Roles	Participants and their actions	Waiter takes order, customer eats and pays
Scenes	Temporal sequence of events	Entering → Ordering → Eating → Paying → Leaving

---

7. Inference — Resolution in Predicate Logic

1. Resolution is a **complete and sound inference method** for predicate logic that uses a single rule — the resolution rule — to prove theorems by refutation (contradiction).
2. Resolution works by converting all statements to **CNF (Conjunctive Normal Form)**, negating the goal, and repeatedly resolving pairs of clauses until an empty clause is found.
3. Two literals are **complementary** if one can be unified with the negation of the other — for example man(x) and ¬man(Himalayas) are complementary when x = Himalayas.
4. The **resolution rule** states — if one clause contains literal L and another contains ¬L, the resolvent is the disjunction of all remaining literals from both clauses with L and ¬L removed.
5. Proving statement S by resolution involves adding **¬S to the knowledge base** and resolving until the empty clause is derived — this proves S is true by contradiction.

Resolution Algorithm:

1. Convert all statements in the knowledge base to CNF form.
2. Negate the goal S, convert ¬S to CNF, and add it to the clause set.
3. Select two parent clauses that contain complementary literals that can be unified.
4. Resolve them to produce a resolvent — disjunction of all other literals after removing the complementary pair.
5. If the resolvent is the empty clause, contradiction is found and proof is complete — otherwise add resolvent to clause set and repeat.

7a. Horn Clauses

1. A **Horn Clause** is a disjunction of literals in which **at most one literal is positive** — for example ¬A ∨ ¬B ∨ C is a Horn clause with exactly one positive literal (C).
2. Horn clauses with exactly one positive literal are called **definite clauses** — the single positive literal is the HEAD and the negative literals form the BODY.
3. A definite clause with **no negative literals** (only a positive literal) is called a **fact** — for example American(West) is a fact with just a head and no body.
4. Definite clauses are equivalent to implications — the clause ¬A ∨ ¬B ∨ C is the same as **A ∧ B → C**, meaning "if A and B are true then C is true."
5. Forward and Backward chaining work specifically with **Horn clause knowledge bases** because their restricted form guarantees sound, complete, and efficient inference.

7b. Unification

1. Unification is a **matching procedure** that takes two logical expressions and determines whether a set of variable substitutions (substitution θ) exists that makes them identical.
2. The result of UNIFY(L1, L2) is either a **substitution set (unifier)** that makes L1 and L2 identical, or **FAILURE** if no such substitution exists.
3. A variable can be unified with any constant, another variable, or a function expression — as long as the variable does not appear within the expression it is being unified with (the **occurs check**).
4. The **Most General Unifier (MGU)** is the most flexible unifier — it uses the minimum number of substitutions needed while keeping as many variables free as possible.
5. Unification **fails** if the two expressions have different predicate symbols, different numbers of arguments, or if a variable must be substituted with an expression containing that same variable.

Unification Algorithm — UNIFY(L1, L2):

1. If L1 or L2 is an atom — return NIL if identical; return {L2/L1} if L1 is a variable not in L2; return {L1/L2} if L2 is a variable not in L1; else return FAILURE.
2. If the lengths of L1 and L2 differ, return FAILURE.
3. Set SUBST to NIL — this holds all substitutions found so far.
4. For each pair of corresponding elements — call UNIFY recursively; if FAILURE return FAILURE; if non-empty substitution, apply it to remaining elements and append to SUBST.
5. Return the final SUBST as the complete unifier.

Example: UNIFY(knows(John, x), knows(y, Leonid)) = **{Leonid/x, John/y}**

7c. Forward Chaining

1. Forward chaining starts with **known facts** in the knowledge base and applies Modus Ponens in the forward direction — deriving all consequences until no new facts can be generated.
2. It follows the pattern — given that P implies Q and P is true, **conclude Q is true** — the system works forward from what is known to what can be concluded.
3. When a new fact is derived, it is **added to the knowledge base** and can trigger other rules whose conditions it satisfies, continuing the chain of inference.
4. Forward chaining is **data-driven** — it is well-suited when new information arrives continuously and the system needs to immediately determine all its consequences.
5. Forward chaining is **sound and complete** for Horn clauses but can be inefficient because it may derive many facts that are irrelevant to the actual goal.

Forward Chaining — Col. West Example:

Given Facts: American(West), Enemy(Nono, America), Missile(M1), Owns(Nono, M1)

Step	Rule Fired	New Fact Derived
1	Missile(x) → Weapon(x)	Weapon(M1)
2	Missile(x) ∧ Owns(Nono,x) → Sells(West,x,Nono)	Sells(West, M1, Nono)
3	Enemy(x, America) → Hostile(x)	Hostile(Nono)
4	American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) → Criminal(x)	**Criminal(West) ✓**

7d. Backward Chaining

1. Backward chaining starts with the **goal to be proved** and works backwards — it finds rules whose conclusion matches the goal, then tries to prove those rules' preconditions as sub-goals.
2. It follows the pattern — to prove G, find a rule whose head matches G, then **prove all conditions in the body** one by one as sub-goals recursively.
3. Backward chaining is **goal-directed** — it is efficient when the goal is specific and well-defined since only facts relevant to the goal are ever derived.
4. It is the basis of **logic programming in PROLOG** — PROLOG performs depth-first backward chaining with Horn clauses using left-to-right evaluation order.
5. A limitation is **incompleteness due to infinite loops** — it can get stuck trying to prove the same sub-goal repeatedly, fixed by maintaining a stack of current sub-goals and checking for duplicates.

Backward Chaining — Col. West Example:

Goal: Prove Criminal(West)

Step	Sub-goal	Result
1	Need: American(West) ∧ Weapon(y) ∧ Sells(West,y,z) ∧ Hostile(z)	Start working backwards
2	American(West)	Found in KB ✓
3	Need Weapon(y) → need Missile(y) → Missile(M1) in KB → Weapon(M1)	✓
4	Need Sells(West,M1,Nono) → need Missile(M1) ∧ Owns(Nono,M1) → both in KB	✓
5	Need Hostile(Nono) → need Enemy(Nono,America) → found in KB	✓
Result	**Criminal(West) proved**	✓

Forward vs Backward Chaining — Comparison

Property	Forward Chaining	Backward Chaining
Direction	Facts → Goal	Goal → Facts

Property	Forward Chaining	Backward Chaining
Driven by	New data and facts arriving	A specific goal to be proved
Efficiency	Can derive many useless facts not related to goal	Only derives facts relevant to the specific goal
Use case	Monitoring systems, event-driven applications	Question answering, diagnosis systems
Completeness	Complete for Horn clauses	Incomplete — infinite loops possible
Example	Medical alert system triggering on patient vitals	PROLOG query resolution

End of AIES Unit II Notes