Computational Intelligence

Dr Binu P Chacko

AI problems

- Intelligence requires knowledge; knowledge captures generalization
- Game playing: chess, checkers – played and improved performance with experience
- Theorem proving: Logic therapist, Gelernter's theorem prover
- Commonsense reasoning using General Problem Solver
- Vision and speech processing (perception), NLP, medical diagnosis, chemical analysis
- Engineering design, robot control, scientific analysis, financial analysis

Applications

Computer Vision

- **Facial Recognition:** Used in smartphones and security systems to identify individuals (PimEyes)
- **Medical Imaging:** Analyzing X-rays, CT scans, and ultrasounds to detect diseases
- **Autonomous Vehicles:** Recognizing road signs, obstacles, and pedestrians for self-driving cars
- **Image and Video Analysis:** Identifying objects, scenes, and activities in images and videos (Gemini)

Natural Language Processing (NLP)

- **Chatbots and Virtual Assistants:** Understanding and responding to user queries, as seen in customer service applications
- **Machine Translation:** Translating text from one language to another
- **Text Summarization:** Condensing long documents into shorter summaries (SciSpace)
- **Sentiment Analysis:** Determining the emotional tone of text

Cont…

Forecasting and Prediction

- **Stock Market Prediction:** Analyzing financial data to forecast future trends (capitalize.ai)
- **Weather Forecasting:** Predicting future weather conditions (WeatherNext)
- **Demand Prediction:** Forecasting consumer demand for products and services (c3.ai)

Healthcare

- **Disease Diagnosis:** Assisting doctors in identifying diseases from medical data.
- **Personalized Treatment Plans:** Developing customized treatment strategies based on a patient's data (Tempus)
- **Drug Discovery:** Analyzing complex data to identify potential new drugs (AIDDISON)

Cont…

Finance

- **Fraud Detection:** Identifying fraudulent transactions by detecting unusual patterns (Feedzai)
- **Risk Management:** Assessing and managing financial risks (Predict360)
- **Trading:** Analyzing trading volume and other market data to make predictions (StockHero)

Other Applications

- **Recommendation Systems:** Suggesting products, movies, or music based on user preferences (Netflix, Amazon).
- **Gaming:** Developing AI agents that can play games (AlphaGo).
- **Data Mining:** Extracting meaningful information from large datasets (Scikit-learn, Pandas)
- **Speech Recognition:** Transcribing spoken words into text and understanding the tone of voice (Google AI)

Define the Problem

- Problem statement: Play chess
- Specify starting position of the chess board
- Rules of legal moves from initial state to goal state
- Board position for a win
- Goal: win the game

Problem of moving around in a state space

- Convert given situation into desired situation using a set of permissible operations
- Solve a problem as a combination of known techniques and search. Explore the space to find some path from current state to goal state

Production System

Consists of

- A set of rules, each consisting of a left side that determines the applicability of the rule, and a right side that describes the operation to be performed
- One or more knowledge/databases
- A control strategy that specifies the order in which the rules will be compared to the database, and the way of resolving the conflicts that arise when several rules match at once
- A rule applier

Control strategies should ….

- Cause motion
- Be systematic

Breadth-First search

- Create a variable called NODE-LIST and set it to the initial state
- Until a goal state is found or NODE-LIST is empty:
  - Remove first element from NODE-LIST and call it E. if NODE-LIST was empty, quit
  - For each way that each rule can match the state described in E do:
    - Apply the rule to generate a new state
    - If new state is a goal state, quit and return this state
    - Otherwise, add the new state to the end of NODE-LIST
- Advantages: It won't get trapped exploring a blind alley
- If there is solution, this search is guaranteed to find it

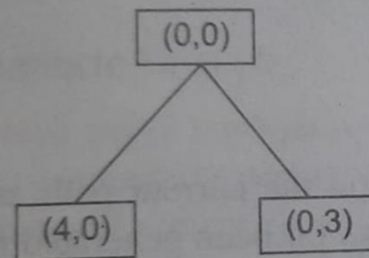


Fig. 2.5 *One Level of a Breadth-First Search Tree*

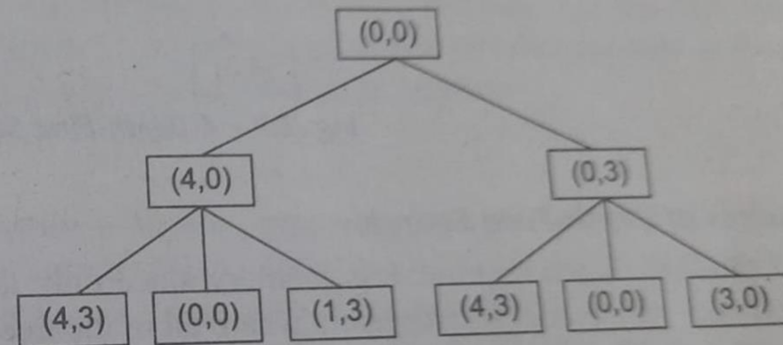


Fig. 2.6 *Two Levels of a Breadth-First Search Tree*

[4] Rule 3, 4, 11, and 12 have been ignored in constructing the search tree.

- If the initial state is a goal state, quit and return success
- Otherwise, do the following until success or failure is signaled
  – Generate a successor E of the initial state. If there are no more successors, signal failure
  – Call Depth-First search with E as the initial state
  – If success is returned, signal success. Otherwise continue in this loop

Advantages

- Requires less memory – only the nodes on the current path are stored
- It may find a solution without examining much of the search space

Depth-First search

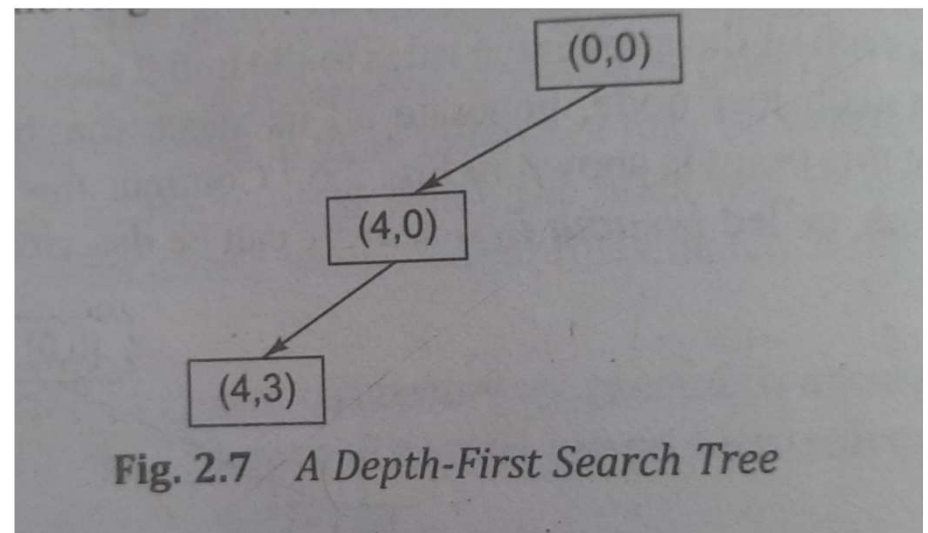


Fig. 2.7 A Depth-First Search Tree

Problem Characteristics

- Is the problem decomposable?
  - Solve each sub-problem using a collection of specific rules
- Can solution steps be ignored or undone?
  - Ignorable, Recoverable, Irrecoverable
- Is the universe predictable?
  - Plan to avoid undo
  - Choose a plan after investigating several plans
  - Problem-solving without feedback from the environment
  - Plan revision

Cont…

- Is a good solution absolute or relative?
  - Travelling salesman problem – any path, best path
- Is the solution a state or a path?
  - Solution to a particular problem is a sequence of operations that produces the final state
- What is the role of knowledge?
  - To constrain the search and to recognize the solution
- Does the task require interaction with a person?
  - Conversational, Solitary (no intermediate communication)
- Problem classification

Production System Characteristics

- Monotonic production system: application of a rule never prevents the application of another rule
- Partially commutative production system: if the application of a particular sequence of rules transforms state x into state y, then any permutation of those rules that is allowable also transforms state x into state y
- Commutative production system: both monotonic and partially commutative
- Non-monotonic production system

Cont…

	Monotonic	Non-monotonic
Partially commutative	Theorem proving	Robot navigation
Non partially commutative	Chemical synthesis	Bridge

- Create process, implemented without the ability to backtrack to previous state
- Good for problems where things don't change
- For problems in which changes occur, can be reversed, order of operation isn't critical
- For problems in which irreversible changes occur
- The order in which operations are performed is important
- Less likely to produce same node many times in the search process

Heuristic Search Techniques

Generate-and-Test

1. Generate a possible solution. It may be a particular point in the problem space, or a path from start state
2. Test to see if it is a solution by comparing the chosen point or the endpoint of the path with a set of acceptable goal states
3. If the solution is obtained, quit. Otherwise go to step 1

- Foundational AI search technique
- Uses domain knowledge to generate better candidate solution
- Reduces search space significantly

Cont…

```
repeat
    generate a candidate solution
    if test(candidate) = success
        return candidate
until no more candidates
return failure
```

8-Queens Problem

- **Goal:** Place 8 queens on a chessboard so that none attack each other

  Generate: Randomly place 8 queens on the board

  Test: Check if any queens attack each other
- **Repeat:** If conflict exists, generate a new arrangement

Hill Climbing

- Simple Hill Climbing: a heuristic local search technique that selects the *first neighbouring state* that improves the evaluation function and continues until no improvement is possible.

1. Start with an initial state
2. Evaluate the current state
3. Generate one neighbouring state
4. If the neighbour is better → move to it
5. Repeat until no better neighbour is found

- Maximize the function, $f(x)=-(x-3)^2+10$

Start with x=0, 1, 2, …

```
current ← initial_state
loop:
   next ← first better successor(current)
   if next does not exist:
      return current
   current ← next
```

Steepest-Ascent Hill Climbing

- At each step, it examines all neighbouring states and moves to the *best one* (steepest upward move)

1. Start with an initial state
2. Generate all neighbouring states
3. Evaluate each neighbour using a heuristic function
4. Select the neighbour with the highest heuristic value
5. If it is better than the current state, move to it
6. Otherwise, stop

Maximize the function, $f(x)=-(x-3)^2+10$

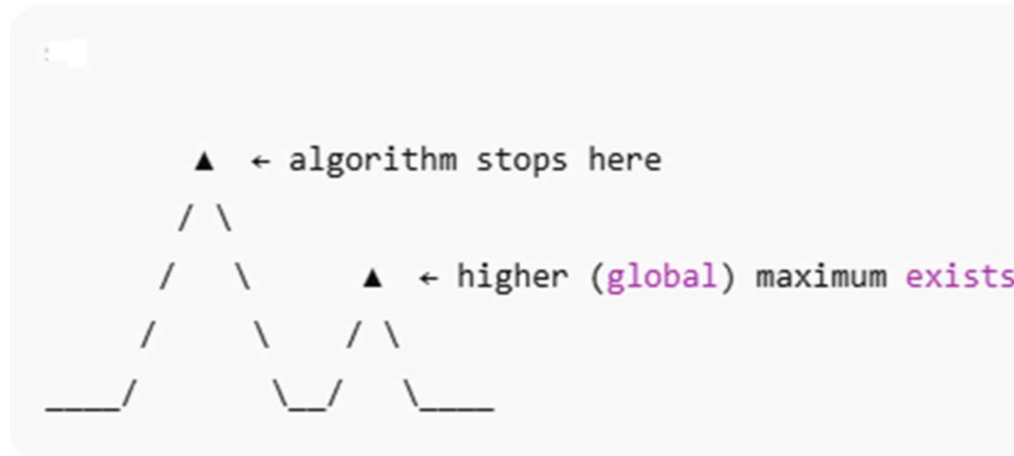
- Initial state, x=1

neighbours, x=0,2

```
current ← initial_state
loop:
   neighbours ← all successors(current)
   best ← argmax(value(neighbours))
   if value(best) ≤ value(current):
      return current
   current ← best
```

Hill Climbing fails due to …

❌ Local Maximum Problem



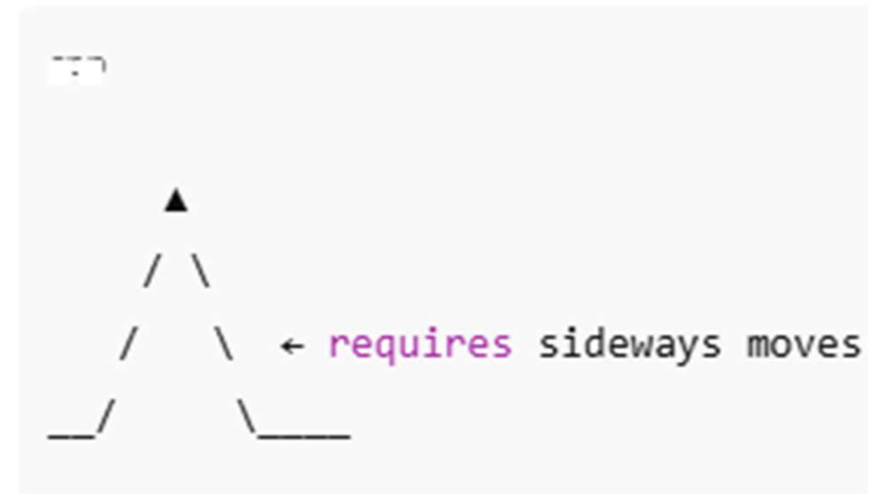
- Algorithm **cannot move down** to reach a better peak
- Stops at the **local maximum**

❌ Plateau Problem

- No improvement found
- Algorithm **stops prematurely**

Solution: Simulated Annealing

❌ Ridge Problem



- Algorithm cannot move diagonally
- Gets stuck

Simulated Annealing

- A probabilistic heuristic search technique, sometimes allowing worse moves
- It is inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to reach a low-energy (stable) state

1. Choose an initial solution
2. Set an initial temperature T
3. Repeat until stopping condition:
   - Generate a random neighbouring solution
   - Compute change in value ΔE
   - If better (ΔE>0), accept it
   - If worse (ΔE<0), accept it with probability:

     $$P=e^{\Delta E/T}$$

   - Reduce temperature T

```
current ← initial_state
T ← initial_temperature
while T > Tmin:
   next ← random_neighbor(current)
   ΔE ← value(next) – value(current)
   if ΔE > 0:
      current ← next
   else if random(0,1) < exp(ΔE / T):
      current ← next
   T ← cool(T)
return current
```

Cont…

- Maximize the function, $f(x)=-(x-4)^2+16$

➢ Cooling schedule, T=T-2

➢ Stop condition: temperature is low, No better neighbours exist

Step	T	Current x	f(x)	Move Type	
0	10	1	7	Start	
1	10	0	0	Worse (accepted)	ΔE=-7 P≈0.49
2	8	1	7	Better	
3	6	2	12	Better	
4	4	3	15	Better	
5	2	4	16	Better	

Best-First search

- A graph/tree search algorithm that always expands the *most promising node first*, according to an evaluation function
- Instead of exploring nodes level-by-level or depth-by-depth, it uses a **priority queue** to decide which node looks "best" to explore next
- In an OR graph, Best-First Search selects and expands the most promising alternative at each step based on a heuristic function, and the parent node is solved when any one of its successor nodes is solved

Cont…

1. Insert start node into priority queue
2. While queue is not empty:
   - Remove node with **lowest f(n)**
   - If node is goal → success
   - Else:
     - Generate successors
     - Add unvisited ones to queue
3. If queue empty → failure

```
BestFirstSearch(start, goal):
   open ← priority queue ordered by f(n)
   closed ← empty set
   insert start into open
   while open not empty:
      n ← remove node with smallest f(n)
      if n is goal:
         return solution path
      add n to closed
      for each successor s of n:
         if s not in closed:
            compute f(s)
            add s to open
   return failure
```

A* algorithm

- An informed search algorithm used to find the *shortest (optimal) path* from a start node to a goal node
- It combines the strengths of Uniform Cost Search and Greedy Best-First Search
- Select the next node based on smallest $f(n)=g(n)+h(n)$

$g(n)$ → actual cost from start to node *n*

$h(n)$ → heuristic (estimated cost from *n* to goal)

$f(n)$ → estimated total cost of solution via *n*

Cont…

- Solution path: S -> A -> C -> G

```
                                   S
                        (Start Node)
                        g=0, h=6
                          f=6
                           |
              -------------------------------
              |                       |
              A                       B
          g=1, h=4                 g=2, h=5
            f=5                   f=7
              |                       |
              C                       D
          g=3, h=2                 g=5, h=3
            f=5                   f=8
              |
              G
          (Goal Node)
          g=5, h=0
            f=5
```

Cont…

1. Place the **start node** in the OPEN list
2. Set g(start) = 0
3. While OPEN is not empty:
   - Remove node n with **lowest f(n)**
   - If n is the goal → return solution path
   - Move n to CLOSED list
   - For each successor s of n:
     - Compute g(s) = g(n) + cost(n, s)
     - Compute f(s) = g(s) + h(s)
     - Add or update s in OPEN list
4. If OPEN is empty → failure

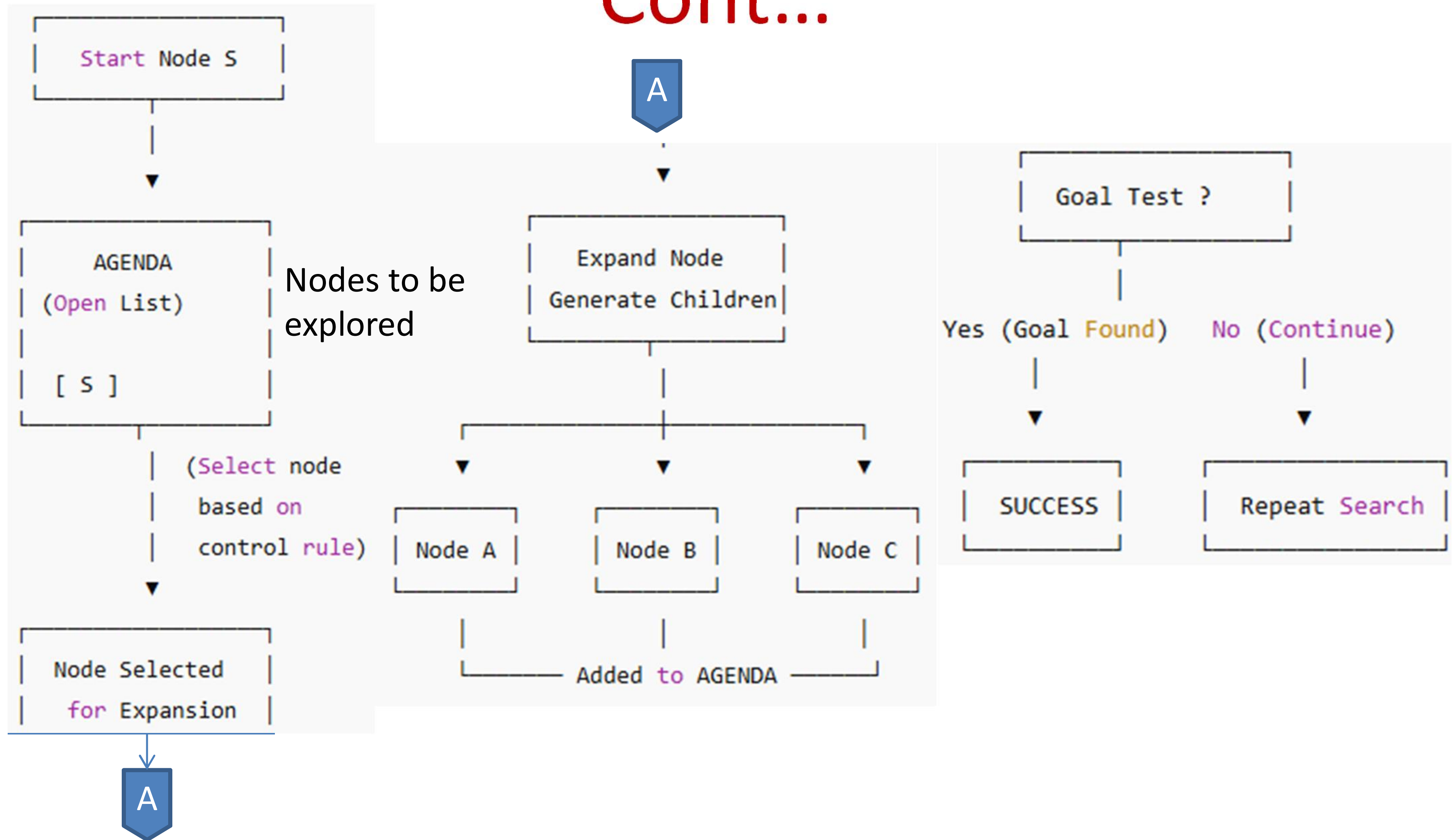
```
A*(start, goal):
   open ← priority queue (ordered by f)
   closed ← empty set
   g(start) = 0
   f(start) = h(start)
   insert start into open
   while open not empty:
      n ← node with smallest f(n)
      if n = goal:
         return path
      remove n from open
      add n to closed
      for each successor s of n:
         if s in closed:
            continue
         tentative_g = g(n) + cost(n, s)
         if s not in open or tentative_g < g(s):
            parent(s) = n
            g(s) = tentative_g
            f(s) = g(s) + h(s)
            insert s into open
   return failure
```

Agenda-Driven search

- It is a search framework in which an agenda (open list) stores unexplored nodes, and the order of node expansion is controlled by a selection rule
- Agenda-driven search is a framework, not a single algorithm.
- Allows easy switching between strategies by changing agenda rules

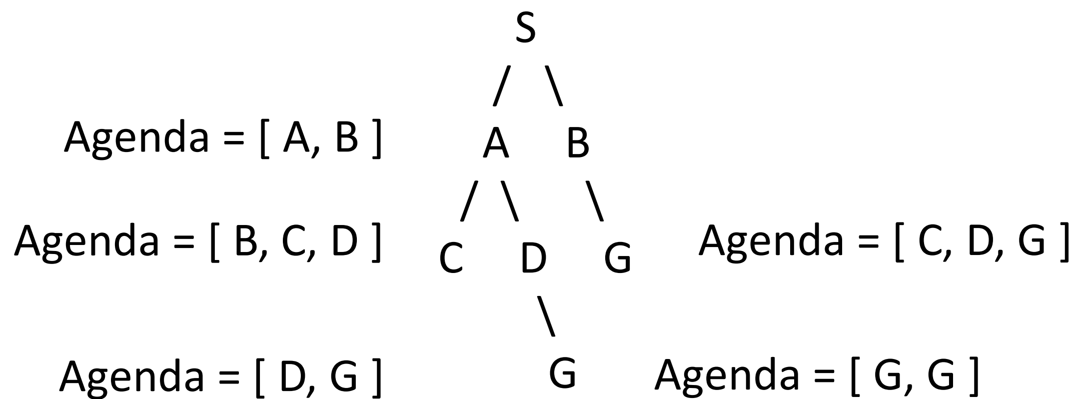
Search Type	
Agenda Structure	Search Strategy
Queue (FIFO)	Breadth-First Search
Stack (LIFO)	Depth-First Search
Priority Queue (h(n))	Best-First Search
Priority Queue (g(n)+h(n))	A* Search

Cont...



Cont…

- Agenda type: Queue (FIFO) - behave like Breadth-First Search
- Agenda (open list) = [S], Goal: G

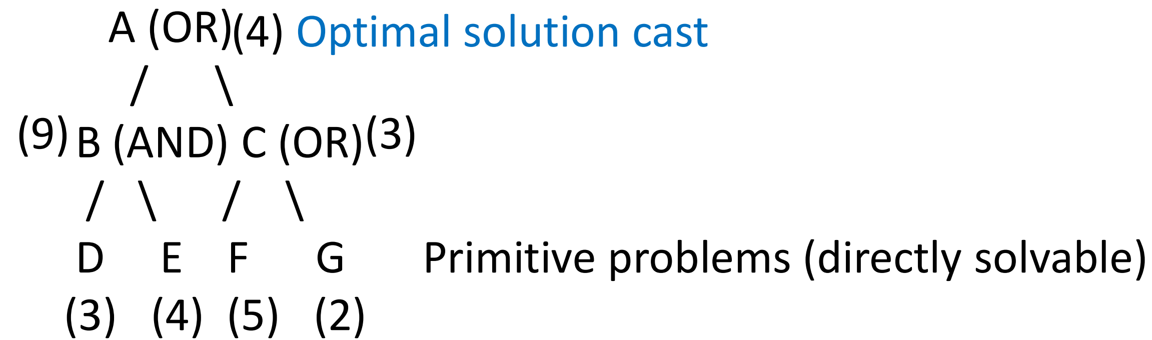


Problem Reduction

- A complex problem is broken down into simpler subproblems -> Each subproblem is solved -> Their solutions are combined to solve the original problem
- AND–OR graph is used to represent this process
- Each *node* represents a **problem or subproblem.** *Edges* show how a problem is **reduced** into smaller problems
- Choose among alternatives (OR) - Solving any one of the children solves the node
- Solve multiple subproblems simultaneously (AND) - All children must be solved to solve the node
- Solution is a subgraph such that all leaf nodes are goal states

Cont…

- Edge cost = 1
- Best choice = A → C → G



AO* algorithm

- Heuristic search algorithm used to find optimal solution graph in an AND–OR graph.

1. Start with the initial node
2. Use heuristic values to select the most promising path
3. Expand the node
4. Update costs backward
5. Mark nodes as *solved* when:
   - OR node: one child solved
   - AND node: all children solved
6. Stop when the start node is solved

- OR node, $cost(n)=min(cost(c_i)+edge_cost)$
- AND node, $cost(n)=\sum(cost(c_i)+edge_cost)$

Cont…

```
AO*(node n)
{
   if n is a terminal node
      mark n as SOLVED
      return cost(n)

   if n is an OR node
      for each child c of n
         cost(c) = AO*(c)
      cost(n) = minimum(cost(c) + edge_cost)
      mark best child
      if best child is SOLVED
         mark n as SOLVED

   if n is an AND node
      for each child c of n
         cost(c) = AO*(c)
      cost(n) = sum(cost(c) + edge_cost)
      if all children are SOLVED
         mark n as SOLVED

   return cost(n)
}
```

Constraint Satisfaction Problem

- Finding values for variables that obey a set of restrictions, where the challenge lies in efficiently navigating a massive space of possibilities

Consists of

- Variables: $X=\{X_1,X_2,\ldots,X_n\}$
- Domains of possible values: $D(X_1)=\{1,2,3\}$
- Constraints: Rules that restrict which combinations of values are allowed, e.g.: $X_1 \neq X_2$

Goal: Assign a value to every variable such that all constraints are satisfied

Sudoku

8					5			
	7		9				4	
		9		7	8	3	2	5
3		1		9			5	
		6				1		
	9			3		6		2
2	8	3	6	5		7		
	1				2		8	
			1					9

- Each cell is a variable, $X_{r,c}$ for $r,c \in \{1,\ldots,9\}$ **X_{13}**
- Domains: Each variable can take a value from:

$D(X_{r,c})=\{1,2,3,4,5,6,7,8,9\}$ $D(X_{1,3})=\{1,2,3,4,5,6,7,8,9\}$

If a cell is pre-filled, its domain is restricted to that single value

- Constrains: Row constraint - All numbers in a row must be different

$D(X_{1,3})=\{1,2,3,4,6,7,9\}$

Column constraint: All numbers in a column must be different $D(X_{1,3})=\{2,4,7\}$

3×3 subgrid constraint: All numbers in a 3×3 box must be different $D(X_{1,3})=\{2,4\}$

These are usually modelled as global AllDifferent constraints Constraint propagation

- Solver looks for the cell with the Minimum Remaining Values, $D(X_{7,6})=\{9\}$
- forced move triggers forward checking
- This may cause New single-value domains, and Chain reactions of forced assignments constraint cascading
- Backtracking may be needed in some cases

Means-Ends analysis

- A goal-oriented problem-solving method that repeatedly reduces the difference between the current state and the desired goal by selecting appropriate actions
- Compare where you are now (means) with where you want to be (ends), identify the difference, and choose an action that reduces the difference

1. **Compare** the current state with the goal state
2. **Identify differences** between them
3. **Select an operator** that can reduce one of the differences
4. **Check preconditions** for the operator
   - If not satisfied → create **subgoals**
5. **Apply the operator**
6. **Repeat** until the goal is achieved

Cont…

```
MEANS_ENDS_ANALYSIS(current_state, goal_state):
  if current_state satisfies goal_state then
    return SUCCESS
  differences ← COMPARE(current_state, goal_state)
  for each difference d in differences do
    operator ← SELECT_OPERATOR(d)
    if operator is applicable then
      if PRECONDITIONS(operator) are satisfied then
        new_state ← APPLY(operator, current_state)
        result ← MEANS_ENDS_ANALYSIS(new_state, goal_state)
        if result = SUCCESS then
          return SUCCESS
      else
        subgoals ← PRECONDITIONS(operator)
        for each subgoal in subgoals do
          result ← MEANS_ENDS_ANALYSIS(current_state, subgoal)
          if result = FAILURE then
            return FAILURE
  return FAILURE
```

Knowledge Representation

- Representation + Mapping = Meaningful Intelligence
- Describe how real-world knowledge is modelled inside a computer so that reasoning and inference are possible

Representation: a formal way of encoding knowledge about the world so that a computer system can store, interpret, and reason about it

- A good knowledge representation should, i) Capture relevant real-world facts, ii) Be unambiguous, iii) Support inference and reasoning, iv) Be computationally efficient

Types of Knowledge Representations

- Logical representations: Use formal logic to represent facts and rules
- Prepositional logic: Raining → WetGround
- First-order predicate logic: Human(Socrates)

  $\forall x$ Human(x) → Mortal(x)
- Precise and expressive. *Can be computationally expensive*
- Semantic networks: Knowledge represented as graphs
- Nodes → concepts, Edges → relationships
- [Dog] --is-an--> [Animal]
- [Dog] --can--> [Run]
- Intuitive and visual. *Limited formal semantics*
- Frames: Structured objects with slots and values
- Good for structured knowledge. *Less suitable for complex reasoning*

```
Frame: Dog
  Type: Animal
  Legs: 4
  Sound: Bark
```

- Production rules: Knowledge as IF–THEN rules
- IF fever AND cough THEN flu
- Easy to understand and modify. *Rule conflicts can arise*
- Ontologies: Formal definitions of concepts and relationships in a domain
- Classes: Student, Course; Relationships: enrollsIn
- Widely used in semantic web and NLP. *Complex to design*

Mappings

- Correspondence between real-world entities and their symbolic representations in a knowledge system

Types of mappings

- World → Symbol Mapping (Encoding): converts real-world facts into symbolic form
- Real world: “Paris is the capital of France”
- Symbolic: Capital(Paris, France)
- Symbol → World Mapping (Interpretation): Interprets symbolic conclusions back into real-world meaning
- Symbolic inference: Mortal(Socrates)
- Interpretation: “Socrates will eventually die”
- Conceptual mapping: Relates different representations or abstraction levels
- “Car” ↔ “Vehicle”
- Used in Ontology alignment, Knowledge integration

Example

- Real World: “Alice is a student enrolled in CS101.”
- Representation (Predicate logic): Student(Alice)

  Enrolled(Alice, CS101)
- Mapping: Alice → a real person

CS101 → a real course

Enrolled → real-world relationship

- Reasoning: ∀x Student(x) → EligibleForLibrary(x)

  EligibleForLibrary(Alice)

Issues in Knowledge Representation

- Representational adequacy: Can KR system represent all necessary knowledge about the domain?
- The real world is complex and rich. Some representations cannot express time, uncertainty, actions, and belief
- Inferential adequacy: Can the system derive new knowledge from existing knowledge?
- Representation should support logical inference
- Given knowledge about Socrates. The system must infer, Mortal(Socrates)
- Inferential efficiency: Can inference be performed efficiently?
- Some representations allow reasoning but are too slow and computationally expensive
- Acquisitional efficiency: How easily can new knowledge be added or modified?
- Knowledge changes over time. Manual encoding is time-consuming and error-prone

Cont…

- Handling Uncertainty and Incompleteness
- E.g. Patient has fever → might have flu (not always)
- Classical logic handles only true/false, not probability
- Consistency and Contradiction: Knowledge bases may contain conflicting facts
- Knowledge Updating and Frame Problem
- Frame problem: When a robot moves, What changes? What remains the same?
- Symbol grounding problem: How do symbols acquire real-world meaning?
- Apple – is it a fruit or company?
- Context and Common-Sense Reasoning: Human reasoning depends heavily on context and common sense
- Can a bird fly? → Depends on species, age, injury, context
- Computers struggle with such implicit knowledge
- Scalability: Can the KR system scale to large knowledge bases?
- Millions of facts. Reasoning must still be fast and accurate

Representing Simple Facts

- A simple fact is a statement that is unambiguously true or false about the world. E.g. Mahatma Gandhi is our father of nation
- Prepositional logic (Boolean logic): Each fact is represented as a proposition (symbol)
- P = "Socrates is a man“. P is True
- First-order predicate logic: This is the standard logic used in KR
- Predicate(Object1, Object2, ...)
- **Components: Constants** → specific objects (Socrates, Paris)
- **Predicates** → properties or relations (Man, Owns)
- **Functions** (optional) → map objects to objects
- **Advantages:** ✔ Represents objects and relations
  ✔ Supports reasoning and inference
  ✔ Widely used in AI and KR

Natural Language	Predicate Logic
Socrates is a man	Man(Socrates)
Paris is a city	City(Paris)
John owns a car	Owns(John, Car)
Alice likes ice cream	Likes(Alice, IceCream)

Cont…

- Quatifiers allow general facts
- Universal Quantifier (∀): "All humans are mortal". $\forall x\ (Human(x) \rightarrow Mortal(x))$
- Existential Quantifier (∃): "There exists a student who likes AI". $\exists x\ (Student(x) \wedge Likes(x, AI))$
- Description logic: used in semantic web
- E.g. $Man \subseteq Human$, $Socrates : Man$, haveChild(John, Mary)
- Frame-based representation: Facts are stored as attribute–value pairs

Representing Instances

- An instance is a specific individual object that belongs to a class (concept)
- E.g. *New Delhi* is an instance of *City*
- Predicate logic: Syntax: ClassName(instance). E.g. City(New Delhi)
- States that the object belongs to a class
- Description logic: syntax: instance : Class. E.g. New Delhi : City
- Clear separation between classes and individuals; Supports automated reasoning
- Frames: Instances are represented as frames with slots

ISA relationship

- It represents inheritance
- E.g. A *Man* **ISA** *Human;* A *Human* **ISA** *Mammal*
- Predicate logic: syntax: ∀x (Subclass(x) → Superclass(x))
- Description logic: Subclass ⊆ Superclass
- E.g. Man ⊆ Human; Human ⊆ Mammal
- Used in ontologies and knowledge graphs; Enables automatic classification
- Semantic networks: Represented as edges between nodes
- Man —ISA—▶ Human —ISA—▶ Mammal
- Frames

Predicate logic
Man(Socrates)
∀x (Man(x) → Human(x))
∀x (Human(x) → Mortal(x))

Inferred facts
Human(Socrates)
Mortal(Socrates)

Frame: Man
ISA: Human
Frame: Human
ISA: Mammal

Property	Description
Transitivity	If A ISA B and B ISA C, then A ISA C
Inheritance	Instances inherit properties of superclasses
Consistency	No cycles in strict hierarchies
Default Reasoning	Subclasses override defaults

Computable Function

- It is a function, f: D -> R, such that for every input in domain D, an algorithm can compute f(x) in finite time
- Computable functions are used to i) Derive new values from known facts, ii) Represent attributes and functional relationships, iii) Support procedural attachments in KR systems
- Age(birthYear) = currentYear – birthYear — Arithmetic function
- Age(Sachin) = 53; Age(Sachin) > 50 — Functional terms
- OlderThan(x, y) ← Age(x) > Age(y) — Predicate logic

Computable Predicate

- It is a predicate, $P(x_1, x_2, \ldots, x_n)$, for which there exists an algorithm that Returns TRUE or FALSE, and Halts for all valid inputs
- Prime(x)
- Parent(John, Mary); computable if the KB is finite and facts are explicit
- Mortal(x); computable if the KB contains a finite set of rules, and Reasoning terminates
- Logic programming: greater(X, Y) : X > Y
- > is a computable predicate; Reasoning is procedural and decidable
- Description logic: Adult ≡ Person ⊓ (age ≥ 18)
- Only restricted computable constructs are allowed; Guarantees decidability

Computable functions compute *values*, computable predicates decide *truth*, and both ensure that reasoning in KR is algorithmically feasible

Resolution

- A rule of inference used mainly in propositional logic and first-order logic to derive new knowledge and prove the truth or falsity of a statement

Clause 1: *It is raining OR it is cold*

Clause 2: *NOT raining*

Conclusion: It is cold

- If one clause contains a literal and another contains its negation, we can resolve them by removing that literal
- Prepositional logic

Given two clauses: (A∨B), (¬A∨C)

The resolvent it: (B∨C)

- First-order logic: resolution requires unification
- Man(x)→Mortal(x)
  ⟶ ¬Man(x) ∨ Mortal(x)
- Man(Einstein)
- After unification: x = Eintstein
- Resolvent: Mortal(Einstein)

Example

- Problem Statement

Given in the knowledge base:

- *If it rains, then the ground is wet*
- *It is raining*

Prove using resolution that: *The ground is wet*

KR in Prepositional logic	
Let R: It is raining; W: The ground is wet Convert the statements: *If it rains, then the ground is wet* It is raining Goal	 R → W R W
Convert sentences to CNF	
Rule Apply it Knowledge base in CNF	A → B ≡ ¬A ∨ B R → W ⟶ ¬R ∨ W R ⟶ already a clause Clause 1: ¬R ∨ W Clause 2: R
Negate the query	
To prove W, negate it and add to the KB Now the clause set is	Negated query: ¬W ¬R ∨ W; R; ¬W
Apply resolutions	
Resolution 1 Resolve: Clause 1: ¬R∨W; Clause 2: R Complementary literals: R and ¬R	Resolution 2 Resolve: Derived clause: W Clause 3: ¬W Complementary literals: W and ¬W Resolvent: □(empty clause

The empty clause (⊥) represents a contradiction: KB ∪ {¬W} ⊨ ⊥
Therefore, the original query *W is true*

Final conclusion: Using resolution, it is proved that **the ground is wet**

Natural Deduction

- It is a system of logical reasoning where *conclusions* (newly inferred knowledge) are derived from *premises* (known facts in the knowledge base) by applying introduction and elimination rules for logical connectives such as ∧ (and), ∨ (or), → (implies), ¬ (not), ∀ (for all), ∃ (there exists)
- ∧ **Introduction:** From P and Q, infer P ∧ Q
- ∧ **Elimination:** From P ∧ Q, infer P (or Q)

Prepositional logic

- Knowledge Base: If it is raining, the ground is wet
  R→W
- It is raining, R
- Using Natural Deduction, Apply → Elimination; $\frac{R \rightarrow W \quad R}{W}$
- Conclusion: The ground is wet

First-order KR

- All humans are mortal, ∀x(Human(x) → Mortal(x))
- Einstein is human, Human(Einstein)
- Deduction: Using universal elimination and implication elimination, Mortal(Einstein)

Rule-based Knowledge Representation

- It expresses knowledge in the form of IF–THEN rules (production rules)
- IF <condition(s)> THEN <action/conclusion>
- IF fever AND cough THEN disease = flu

R1: IF fever AND headache THEN flu

R2: IF flu THEN prescribe_rest

R3: IF flu THEN prescribe_paracetamol

Components

- Rules: Encode domain knowledge; Declarative (state *what* is true)
- Working memory (fact base): Stores facts about the current situation
- Inference engine: Applies rules to known facts to derive new facts; Controls reasoning

fever = true

headache = true

→ flu

→ prescribe_rest

→ prescribe_paracetamol

Types of Rules

- Simple rules: IF A THEN B
- Conjunctive rules: IF A AND B THEN C
- Disjunctive rules: IF A OR B THEN C
- Rules with variables: IF student(X) AND marks(X) > 50 THEN pass(X)

Cont…

Inference mechanism

- Forward chaining (forward reasoning):
- Starts with known facts
- Applies rules to infer new facts
- Stops when goal is reached or no rule applies
- Backward chaining: Starts with a goal
- Tries to find rules that support the goal
- Checks if their conditions are satisfied

Fact: fever = true
Rule: IF fever THEN illness
→ illness = true
Used in expert systems, monitoring systems

Goal: illness
Rule: IF fever THEN illness
Check: Is fever true?
Used in Prolog, diagnostic systems

Variants

- Fuzzy rules: IF temperature is high THEN fever is severe
- Probabilistic rules: IF cough THEN flu (0.7)
- Logic programming (Prolog)
- Expert systems (MYCIN, OPS5)

Aspect	Forward Reasoning	Backward Reasoning
Approach	Data-driven	Goal-driven
Starts from	Known facts	Desired goal
Best for	Prediction, simulation	Diagnosis, queries
Example systems	Rule-based systems	Prolog

Logic Programming

- It focuses on what is true about the world using formal logic (prepositional logic and first-order predicate logic)
- You describe *knowledge* and *relationships*, and the inference engine derives conclusions automatically
- Knowledge is represented as facts, rules, and queries
- Facts describe relationships

```
parent(john, mary).
parent(john, tom).
parent(mary, alice).
male(john).
female(mary).
```

Rules define logical relationships

```
grandparent(X, Y) :-
   parent(X, Z),
   parent(Z, Y).
```

Queries

Ask to Prolog using queries

```
?- parent(john, mary).
Output: true
?- parent(john, X).
X = mary     X = tom
```

```
?- grandparent(john, alice).
Output: true
```

Prolog automatically handles Unification (matching variables), Backtracking (trying alternatives), and Search

Symbolic Reasoning under Uncertainty

- It refers to methods that combine symbolic knowledge representation (logic, rules, symbols) with mechanisms to handle incomplete, noisy, or uncertain information
- Probabilistic logic: Extends logic by attaching probabilities to facts or rules. E.g. P(Flu | Fever) = 0.7
- Bayesian network: uses conditional probabilities
- Nodes = random variables; Edges = causal relationships
- Fever → Flu → Cough
- Inference computes the probability of hypotheses given evidence
- Widely used in diagnosis and decision-making
- Markov logic network: A weighted first-order logic system
- Smokes(x) ⇒ Cancer(x) [weight = 1.5]
- Rules can be violated with penalty
- Bridges symbolic AI and statistical learning

Cont…

- Fuzzy logic: Handles degrees of truth instead of binary truth
- Truth values: $0 \leq truth \leq 1$
- Hot(temperature) = 0.8
- Good for vague concepts like *hot*, *tall*, *near*
- Dempster-Shafer theory: Represents Belief, Plausibility, Ignorance
- Instead of assigning probability to a single event, it assigns belief to sets of events
- Useful when evidence is incomplete or conflicting
- Non-monotonic reasoning: Allows conclusions to be withdrawn when new information appears
- $Bird(x) \rightarrow CanFly(x)$; $Penguin(x) \rightarrow \neg CanFly(x)$
- Models default reasoning under uncertainty

Game Playing

- It involves strategic thinking, deals with uncertainty and adversarial behavior, and requires decision making and searching in large problem space
- Build systems that can represent a game formally, generate possible moves, evaluate game positions, and choose the best move using search strategies

Components

- **Initial State** – Starting configuration of the game
- **Players** – Usually two
- **Actions** – Legal moves
- **Result Function** – Outcome of an action
- **Terminal Test** – Checks if game is over
- **Utility Function** – Numerical value for win(1)/loss(-1)/draw(0)

Types of Games

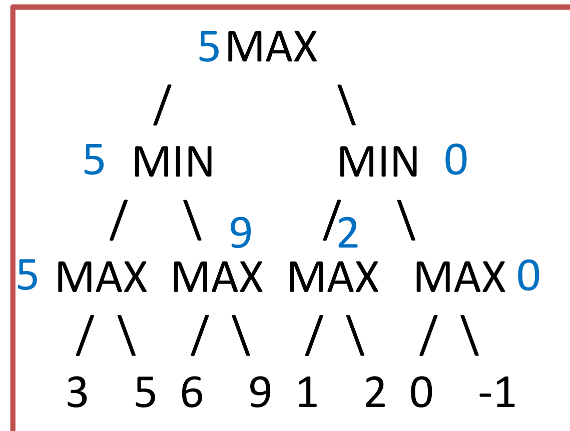
- **Deterministic vs Stochastic**
- Deterministic: No randomness (Chess)
- Stochastic: Has randomness (Backgammon)
- **Perfect vs Imperfect Information**
- Perfect Information: All players see everything (Chess)
- Imperfect Information: Hidden information (Poker)
- **Zero-Sum vs Non-Zero-Sum**
- Zero-Sum: One player's gain = other's loss
- Non-Zero-Sum: Cooperation possible
- AI represents games using a Game Tree consisting of Nodes (Game states), Edges (Moves), Leaves (Terminal states), and Depth (Number of moves ahead)

Minimax Algorithm

- It is a decision-making algorithm used in adversarial games (two-player, zero-sum games) such as chess, tic-tac-toe, and checkers
- The game is deterministic and has perfect information
- Two players (MAX and MIN), both play optimally
- Generate game tree: Start from the current state and generate all possible moves
- Apply terminal test: if The game is over → return utility value
- Depth limit reached → return evaluation function value
- Recursively evaluate children: If it's MAX's turn → choose maximum value
- If it's MIN's turn → choose minimum value
- Backpropagate values: Values are passed upward until reaching the root
- Choose best move: Select the move with the highest minimax value at the root

Example

- Root
- Levels alternate



- Utility values
- MAX can guarantee a score of **5**, assuming MIN plays optimally
- Branching factor (b) = 2, Depth (d) = 3
- Time complexity, $O(b^d)=O(2^3)=O(8)$, evaluated all 8 leaf nodes
- Space complexity, O(bd)

```
Minimax(n) =
   Utility(n)                   if n is terminal
   max(Minimax(child))          if n is MAX node
   min(Minimax(child))          if n is MIN node
```

Alpha-Beta cut-off

- an optimization of the Minimax algorithm that eliminates branches in a game tree that cannot affect the final decision by maintaining two bounds, alpha and beta
- Why pruning? If a node is already worse than a previously explored option, Rational players will never choose it. So we don’t waste time evaluating it.
- α: Best value MAX can guarantee so far; β: Best value MIN can guarantee so far
- Cut the branch if $\alpha \geq \beta$
- Initially: $\alpha = -\infty$; Updated at MAX nodes, Represents the maximum lower bound
- Initially: $\beta = +\infty$; Updated at MIN nodes, Represents the minimum upper bound
- Alpha cutoff occurs at MIN node, if Current value $\leq \alpha$
- Beta cutoff occurs at MAX node, if Current value $\geq \beta$
- Time complexity: Worst case: $O(b^d)$(same as Minimax); Best case (perfect ordering): $O(b^{d/2})$

Additional Refinement

- to improve minimax - reduce computation time and improve practical performance in complex games
- Alpha-beta pruning
- Move ordering: Alpha-beta pruning becomes more effective if best moves are explored earlier
- Better ordering → More pruning → Faster search
- Iterative deepening: depth-limited search is repeatedly applied with increasing depth limits until a solution is found or time runs out
- Search depth = 1, Then depth = 2, Then depth = 3, Continue until time runs out
- Benefits: Always have a best move ready, Helps in move ordering, Works well with time constraints
- Heuristic evaluation function: to estimate the value of a board state
- This allows depth-limited search
- Transposition tables: Store previously evaluated positions. Reuse when same position appears
- Quiescence search: Extend search only in volatile positions. Avoid *horizon effect*
- Forward pruning techniques: Beam search, Null-move pruning

Planning System

- A goal-directed problem-solving framework that generates a sequence of actions to transform an initial state into a desired goal state
- Defined as a 5-tuple: P=(S, A, γ, s_0, G) where: **S** → Set of possible states, **A** → Set of actions, **γ(s, a)** → Transition function, **s_0** → Initial state, **G** → Goal state (or goal conditions)
- Objective: Find a sequence of actions: $\pi=\langle a_1,a_2,\ldots,a_n\rangle$

  Such that: $\gamma(\ldots\gamma(\gamma(s_0, a_1), a_2)\ldots, a_n) \in G$

Components

- State representation: describe the world at a given time
- Prepositional representation, First-order logic representation
- Action representation: defined by: Action=(Preconditions, Effects)
- Each action has:
- **Preconditions** → Conditions that must be true
- **Add List** → Facts added after execution
- **Delete List** → Facts removed after execution

Architecture

Problem Definition

Domain Model (States + Actions)

Planner (Search + Heuristics)

Plan Generation

Execution & Monitoring

Cont…

f the world before planning starts

nditions. Planning system searches for

ions change states,

rates the plan

1. Forward (progression) planning: Start from initial state->Apply actions forward->Stop when goal satisfied
2. Backward (regression) planning: Start from goal, Work backward to initial state, Reduce goal into subgoals
3. Partial order planning: Plan actions without fixing full order, Allows concurrency
4. Heuristic state space planning: Uses heuristics like Relaxed planning graph, Heuristic cost functions

- Heuristic function: $f(n)=g(n)+h(n)$, Where $g(n)$ → cost so far; $h(n)$ → estimated cost to goal
- Plan execution and monitoring: Real-world systems require, Execution module, Monitoring environment changes, Re-planning if needed

Understanding

- Understanding a sentence, situation, or problem consists in finding an interpretation that simultaneously satisfies a set of interacting constraints derived from syntax, semantics, context, and world knowledge
- understanding is like solving a multidimensional puzzle: Each piece restricts where others can go, Many configurations are possible, Only a few are globally consistent, The mind settles on the most stable one
- E.g. The astronomer saw the star with the telescope
- Understanding can be modeled as

astronomer use the telescope?

star have the telescope?

CSP Component	Cognitive Equivalent
Variables	Ambiguous elements (meanings, referents, syntactic roles, intentions)
Domains	Possible interpretations
Constraints	Grammar, semantics, context, world knowledge, discourse coherence
Solution	Coherent interpretation

Cont…

- Variables: Attachment site of prepositional phrase ("with the telescope")
- Domain: Attach to verb phrase, Attach to noun phrase
- Constraints: World knowledge: Stars don't typically carry telescopes
- Plausibility constraints, Statistical linguistic patterns, Syntactic constraints
- Understanding = selecting the interpretation that best satisfies all constraints
- Hard constraint: grammatical violations
- Soft constraint: plausibility, expectations, context
- Weighted constraint satisfaction: Understanding becomes, Find interpretation *I* that maximizes $\sum w_i \cdot C_i(I)$
- Understanding as constraint satisfaction means:

$$\text{Understanding(Input)} = \underset{I \in \mathcal{I}}{\arg\max}\ \text{Coherence}(I)$$

Where coherence is determined by Structural constraints, Semantic constraints, Pragmatic constraints, Background knowledge

Slot and filler structure

- It is a knowledge representation technique used in AI to represent structured information about objects, concepts, or situations
- It is primarily used in Frame-based systems, Semantic networks, Object-oriented knowledge representation, Natural Language Understanding (NLU)
- Knowledge is represented as a collection of objects (frames), each having attributes (slots), which contain values (fillers)

```
Frame: <Frame_Name>
  Slot1: Filler1
  Slot2: Filler2
  Slot3: Filler3
```

```
Frame: Person
  Name: Ravi
  Age: 28            Value slot
  Occupation: Engineer
  Nationality: Indian
```

```
Frame: Animal              Inheritance
  Has_Heart: Yes
  Can_Move: Yes

Frame: Bird
  IS-A: Animal
  Has_Wings: Yes
  Can_Fly: Usually       Default slot –
                         No specific value

Frame: Penguin
  IS-A: Bird
  Can_Fly: No              Override
```

Range constraint: Age: Integer (0-120)

Cardinality constraint: Children: 0..* (allowed fillers)

Procedural constraint

Salary:

IF-NEEDED: Calculate from PayScale

Cont…

- A slot-filler structure can be mathematically represented as:
  Frame $F = \{ (S_1, V_1), (S_2, V_2), \ldots, (S_n, V_n) \}$

Applications

- NLU: Used to fill missing information from text
- Sentence: "Ravi bought a car."
  System fills: Buyer → Ravi, Object → Car, Action → Buy
- Semantic web & ontology: similar to object properties
- Dialogue systems: Used in Slot-filling chatbots, Information extraction systems
- User: "Book a flight to Delhi tomorrow."
  Slots: Destination → Delhi, Date → Tomorrow
- Expert systems

Feature	Slot-Filler	Predicate Logic
Representation	Structured objects	Logical formulas
Inheritance	Yes	No (explicit only)
Default reasoning	Yes	Difficult
Mathematical rigor	Moderate	High

Expert System

- simulate the *decision-making ability* of a human expert in a specific domain
- It is an intelligent computer program that uses *knowledge and inference procedures* to solve complex problems that normally require the expertise and judgment of human specialists. It stores domain knowledge in a *knowledge base* and applies reasoning through an *inference engine* to provide advice, diagnosis, or solutions
- If medical expert system contains the rule

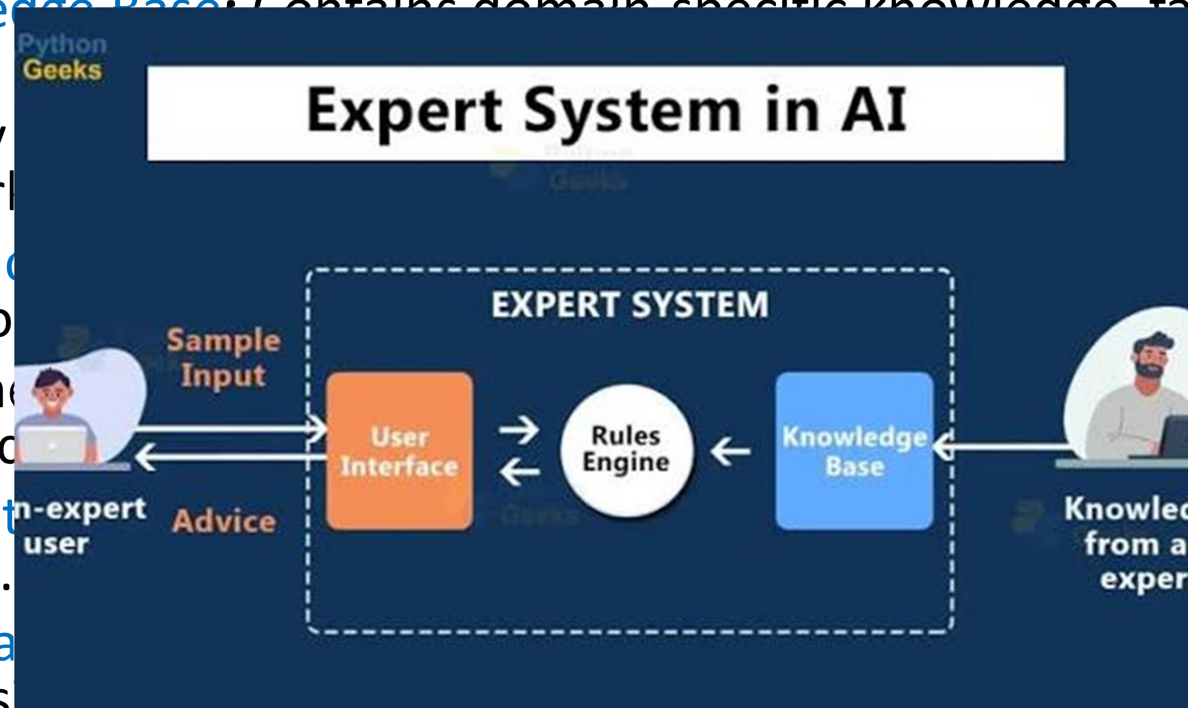
  IF patient has fever AND rash

  THEN possible disease = measles

The inference engine uses patient symptoms to reach a diagnosis

Components

- Knowledge Base: Contains domain-specific knowledge, facts, and rules.
- Usually ... c networ...
- Inferenc... ogical rules to...
- Uses m... **ng** to derive c...
- User Int... he expert system.
- Explana... cision or conclusi...



- Knowledge Acquisition Module: Helps in collecting and updating knowledge from human experts

Characteristics

- High Level of Expertise**:** Expert systems provide solutions with a **high level of skill and knowledge**, comparable to that of a **human specialist** in a specific domain such as medicine, engineering, or finance.
- Domain Specific**:** They are designed to work in a **specific problem area or domain**. For example: medical diagnosis, mineral exploration, or equipment troubleshooting.
- Knowledge-Based System**:** Expert systems rely on a **knowledge base** that contains facts, rules, and information obtained from human experts.
- Reasoning Capability: They use an **inference engine** to apply logical rules and reasoning methods such as: **Forward chaining, Backward chaining**
- This allows the system to draw conclusions from the stored knowledge.
- Explanation Facility**:** Expert systems can **explain how and why** a particular decision or recommendation was made, which helps users understand the reasoning process.
- Reliability and Consistency**:** They provide **consistent answers** for the same set of inputs and do not suffer from fatigue or emotional bias like humans.
- Ability to Handle Complex Problems**:** Expert systems can solve **complex problems that require expert knowledge**, often faster than humans.
- Knowledge Separation**:** The **knowledge base is separated from the control mechanism (inference engine)**, making it easier to update or modify the knowledge without changing the whole system.
- User-Friendly Interface**:** Most expert systems include an **interactive user interface** that allows users to easily communicate with the system

Representing Domain Knowledge

- KR is the method used to encode expert knowledge in a form that a computer can process. Knowledge is usually stored in a *Knowledge Base*
- Rule-based representation
- Semantic networks: KR as a *graph structure* consisting of nodes (objects or concepts) and links (relationship b/w concepts)
- Animal → has → legs; Bird → is-a → Animal; Bird → can → fly
- Good for representing relationships, Easy visualization
- Frame-based representation
- Logic-based representation: Uses formal logic, mainly predicate logic
- Ontologies: formally define concepts and relationships within a domain
- Components: classes, relationships, properties, constraints

Using Domain Knowledge

- Once knowledge is represented, the expert system uses it through an *inference mechanism*
- Inference engine: Inference allows systems to derive new knowledge from existing knowledge
- Match facts with rules, Apply logical reasoning, Generate new knowledge
- Forward chaining (data-driven reasoning): Facts → Rules → Conclusion
- Backward chaining (goal-driven reasoning): Goal → Check rules → Verify facts. Used in Prolog
- Knowledge acquisition: The process of collecting domain knowledge from human experts, Databases, Documents, Machine learning systems
- Techniques: Interviews, Questionnaires, Protocol analysis, Observation, Data mining
- Roles involved: Domain expert, Knowledge engineer, System developer

Knowledge Engineering

- It is the process of designing, constructing, and maintaining *knowledge-based systems* (KBS) by capturing knowledge from domain experts and encoding it in a form that machines can use for reasoning and decision-making

Objectives

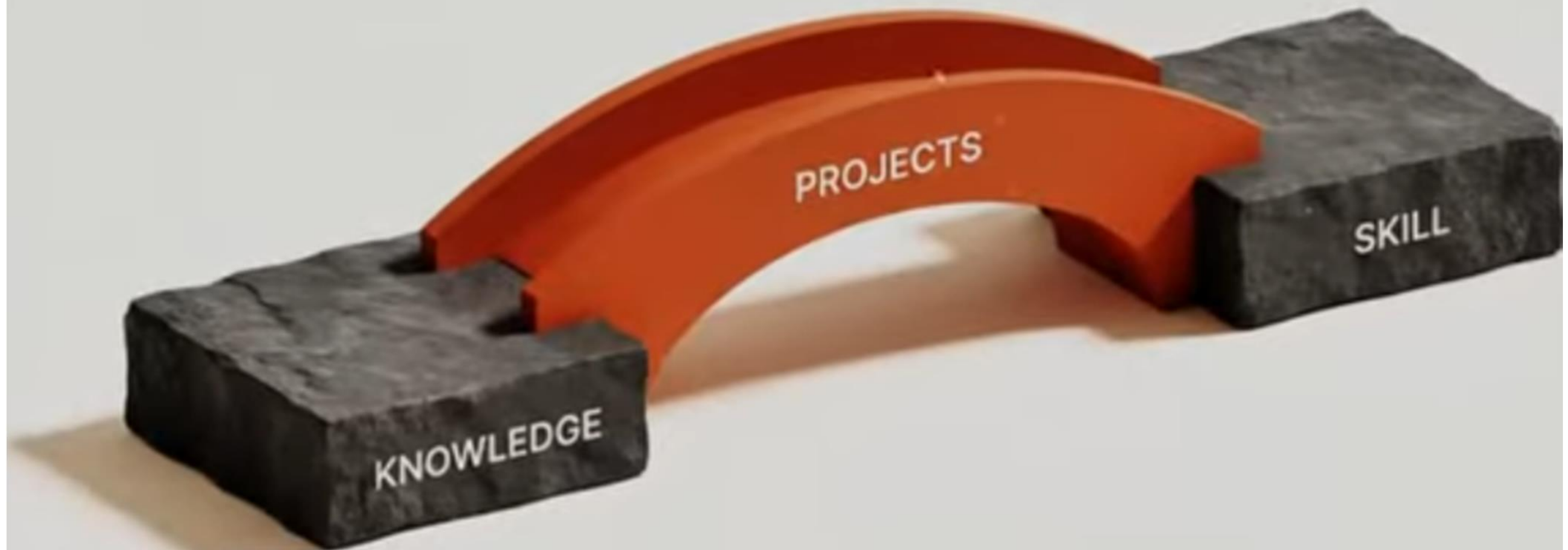
- **Knowledge Acquisition** – Extract knowledge from experts or data.
- **Knowledge Representation** – Represent knowledge in a structured form.
- **Knowledge Validation** – Ensure correctness and consistency.
- **Knowledge Inference** – Use reasoning mechanisms to derive new knowledge.
- **Knowledge Maintenance** – Update knowledge when domain knowledge evolves

Process

Stages in KE life cycle

- Knowledge acquisition
- Knowledge modeling: Knowledge is structured into models describing the domain. E.g. Concepts, Entities, Relationships, Rules, Constraints
- Knowledge representation: encoded in machine-readable forms
- Inference mechanisms
- Applications: Expert systems, DSS (business intelligence), Semantic web (Google knowledge graph), Robotics, NLP, Healthcare (clinical decision systems)

Projects are where learning becomes skill.



Expert System Life Cycle

- It refers to the systematic process used to develop, deploy, and maintain an expert system
- Problem identification: define problem domain and system objectives
- Objectives: Determine whether the problem can be solved using an expert system (e.g. ES for diagnosing infectious diseases), Identify the domain expert, Define the scope and limitations
- Activities: Identify the type of problem (diagnosis, classification, planning, prediction), Determine knowledge sources, Analyze economic feasibility
- Deliverables: Problem definition document, Feasibility report, Project plan
- Knowledge acquisition: Types of knowledge
- Declarative knowledge – facts and concepts
- Procedural knowledge – rules and procedures
- Heuristic knowledge – expert intuition or rules of thumb

Example
Disease
 |
Symptoms → Fever, cough
 |
Diagnosis Rules

Cont…

Architecture
User
 |
User Interface
 |
Inference Engine
 / \
Knowledge Base Explanation System

- Knowledge conceptualization: acquired knowledge is organized into conceptual structures (model)
- Tasks: Identify key concepts, Define relationships, Develop problem-solving strategies
- This stage acts as a bridge between raw knowledge and formal representation
- Knowledge representation
- System design
- Design decisions: Choose inference method (forward/ backward chaining), Select development tools (Prolog, Python), Decide system structure

Cont…

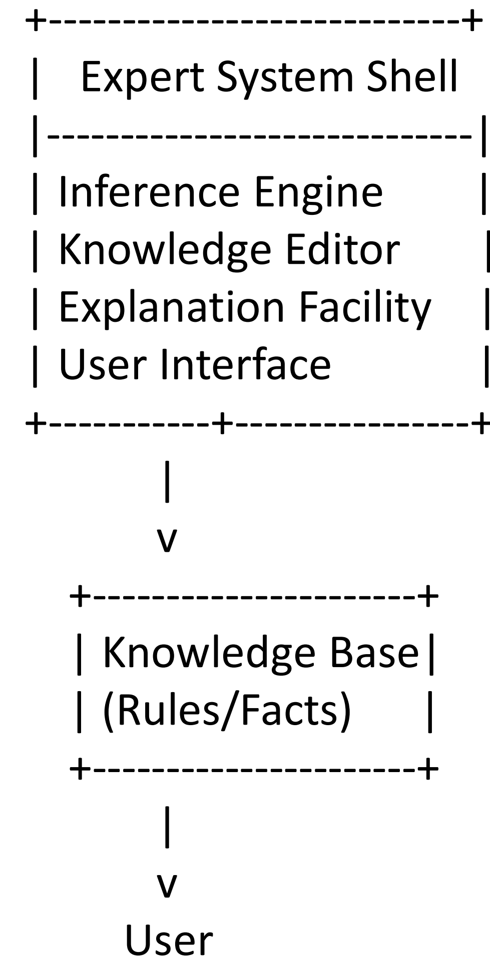
- Implementation: ES is built
- Tasks: Encode knowledge into rules, Implement inference engine, Develop user interface, Integrate system modules
- Tools used: Expert system shells (CLIPS, Jess), Logic programming languages (Prolog), AI frameworks
- Testing: types of testing
- Verification: Ensures system is implemented correctly
- Validation: Ensures system solves the right problem
  - Methods: Compare system results with human expert decisions, Test with real-world cases
- Performance testing: Checks system efficiency
- Deployment: installed in the real-world environment
- Activities: User training, System installation, Documentation, Integration with existing systems
- Maintenance: Knowledge domains evolve, so expert systems must be continuously updated. E.g. New disease discovered → new rule added
- Maintenance ensures long-term usefulness and accuracy
- Tasks: Update knowledge base, Add new rules, Improve reasoning methods, Fix errors

Expert System Shell

- It is a development tool that contains th
interface, explanation facility, and other
create an expert system, but does not in
knowledge

Architecture

- Inference engine: core reasoning mecha
known facts to derive conclusions
- Functions: Rule matching, Pattern recog
Forward or backward chaining
- Knowledge representation tools: knowle
knowledge in forms such as Production
Frames, Semantic Networks, Logic expre
- Knowledge base editor: a tool for Updat
and modifying rules
- User interface: Features - Question-answ
driven input, Natural language interface



Cont.

Conclusion: Patient has flu
Explanation:
Rule 12 fired because:
- Fever = Yes
- Cough = Yes

Domain Knowledge → Knowledge Base

Expert System Shell

Inference Engine

Conclusion

- Explanation facility: Provides justification of decisions made by the system
- *Why* is the system asking this question?

How did the system reach this conclusion?

- Knowledge acquisition tools: capture knowledge from experts through Interactive rule building, Learning modules, Knowledge editing tools
- E.g. C Language Integrated Production System (NASA, 1985) is used for Rule-based programming, AI prototyping, Knowledge-based systems
- Applications: build systems for Medical diagnosis, Fault detection, Financial decision support, Configuration systems, Engineering design

Expert System Tools

```
(defrule fever-diagnosis
  (symptom fever)
  (symptom headache)
  =>
  (assert (disease flu))
)
```

- Software environments or shells that provide facilities such as knowledge representation, inference mechanisms, and user interfaces to develop expert systems efficiently without building them from scratch
- CLIPS: free and open source, forward chaining inference, procedural/COOL
- Components: knowledge base, inference engine (pattern matching using Rete algorithm), user interface, explanation facility
- Applications: Medical diagnosis, Fault detection, Industrial monitoring, Training systems
- EXSYS: Commercial expert system development tool
- Features: Rule-based knowledge representation, Decision tree modelling, Web deployment
- Applications: Customer support systems, Risk assessment, Business decision support

Cont…

```
(defrule cold-diagnosis
  (symptom sneezing)
  (symptom cough)
  =>
  (assert (disease cold))
)
```

```
IF patient has fever
AND patient has cough
THEN disease = flu
CF = 0.8
```

- Java Expert System Shell: Developed by Ernest Friedman-Hill based on CLIPS syntax but designed for Java integration
- Components: Rule/fact base, inference engine (forward chaining), Java API, explanation facility
- Applications: Intelligent agents, Decision support systems, Web-based expert systems
- EMYCIN: First expert system shell, Derived from MYCIN, Uses Certainty Factor (CF) model to represent uncertain knowledge
- Components: Knowledge Base, Inference Engine, Explanation system, Knowledge acquisition module
- Applications: medical diagnosis, Clinical decision support
- Automated Reasoning Tool: Developed by Inference Corporation, Used for large industrial expert systems
- Features: Rule-based reasoning, Object-oriented programming, Knowledge base management
- Applications: Financial analysis, Manufacturing systems, Defense applications

Cont…

- PROLOG: Based on First Order Predicate Logic, Uses backward chaining, Used to build knowledge-base and expert systems
- Components: Facts: fever(john). cough(john).
- Rule: flu(X) :- fever(X), cough(X)
- Query: ?- flu(john).
- Applications: NLP, Automated reasoning, KR
- Knowledge Engineering Environment: Developed by Intellicorp, Runs on LISP machines, Combines frame-based and rule-based systems, Included graphical tools
- Components: Knowledge base, Frame system, Inference engine, User interface
- Applications: Business decision systems, Military systems, Complex knowledge-based applications

MYCIN

- Rule based expert system developed by Edward Shortliffe (Stanford university, 1970)
- to assist physicians in diagnosing bacterial infections of the blood (bacteremia) and meningitis and to recommend appropriate antibiotic treatments and determine dosage based age, weight, allergies

Architecture

- Knowledge base: Around 450–600 IF–THEN rules, Represented expert knowledge from infectious disease specialists
- Inference engine: reasoning using Backward chaining
- Start with a goal (hypothesis)
  Example: Identify the infecting organism
- Search for rules that support the hypothesis
- Ask the user questions to obtain missing facts
- Continue until sufficient evidence is gathered

Cont…

Output
Organism: Streptococcus (CF = 0.75)

Recommended therapy:
Penicillin
Dosage: 4 million units every 4 hours

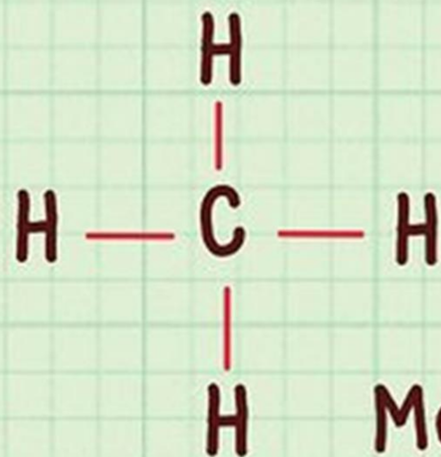
- Working memory (Fact base): Stores patient-specific data such as Symptoms, Lab results, Culture test results, Patient history
- These facts are used during rule evaluation
- Explanation facility increases trust and transparency in medical decision-making
- Certainty Factor model (handling uncertainty): Each rule has a confidence value: $CF \in [-1,+1]$, +1 (completely certain), 0 (unknown), -1 (completely false)
- IF organism is gram-positive

  THEN Streptococcus (CF = 0.7)
- Characteristics: Modular knowledge representation, Easy to update rules, Suitable for expert knowledge encoding, High diagnostic accuracy
- Consultation process: Physician enters patient data, MYCIN asks questions, System applies rules, Produces diagnosis and treatment recommendation

DENDRAL

Architecture

- designed at Stanford university (1960) to infer molecular structures of organic co
- Generate-a
  molecular
- **Test** them
- Structure (… r a
  given Mole
- Uses graph
- Applies co … ures
  early
- Heuristic e … ndidate
  structures
- Encodes ru … y,
  Stability of
- Employs h
- Meta-DEN … **es**
  automatic



Knowledge-base

- Induces fragmentation rules from known molecular structures and spectra, Output: Generalized rules
- 👉This is an early example of **machine learning integrated with symbolic AI**

Cont…

- Knowledge representation: rule based symbolic representation
- IF a molecule contains a carbonyl group

THEN expect a specific fragmentation peak

- Search strategy: DENDRAL combines Exhaustive search (Systematically explores candidate structures), Heuristic pruning (Eliminates implausible structures early), Constraint satisfaction (Ensures chemical validity at every step)

DENDRAL

Input:

Molecular formula + Mass spectrometry data

Process:

Generate candidate structures

↓

Apply chemical constraints

↓

Use expert rules to prune

↓

Simulate spectra

↓

Rank candidates

Output:

Most likely molecular structure(s)

Machine Learning

1980

Expert Systems

- Experts created systems that could mimic human decision-making. These "expert systems" used rules and knowledge from humans to help make decisions.
- Example: Early systems helped doctors diagnose diseases by asking questions.
- Example: Think of how apps like "WebMD" ask you about your symptoms to help figure out what might be wrong with you. Those early systems worked in a similar way.

2000s-present

Modern AI

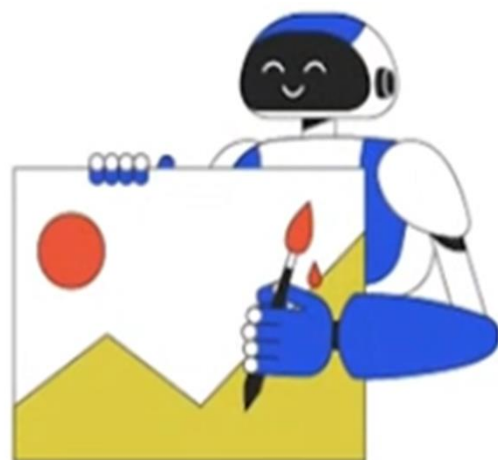
- Starting in the 2000s, AI got smarter with the rise of machine learning (ML) and deep learning.
- AI systems began learning from large amounts of data, becoming more capable.
- Neural networks, inspired by the human brain, power these advancements.
- AI can now recognize faces in photos (like Facebook tagging your friends), or help self-driving cars understand the world around them.
- Example: Netflix recommending movies based on what you've watched before—AI is learning from patterns in the data!

From rule-based systems to Generative AI

- Type: Animal
- Legs: 4
- Color: Brown
- Barks: Yes

Traditional Programming

Machine Learning



Generative AI

Dog

Dog

What is ML?

Learning from Data

ML enables computers to learn patterns from data and make decisions without explicit programming.

Key concept

Features are measurable properties or characteristics of data.
Example: In a customer dataset, features could be age, income, and purchase history.

Two Main Types of Problems:
Classification – Predicts categories (e.g., spam detection, disease diagnosis).
Regression – Predicts continuous values (e.g., house price prediction, stock forecasting).

Types

Supervised Learning: Learns from labeled data (e.g., fraud detection).

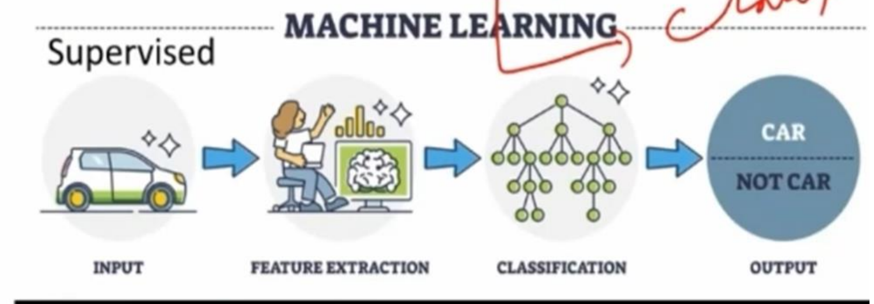
Unsupervised Learning: Identifies patterns in unlabeled data (e.g., customer segmentation).

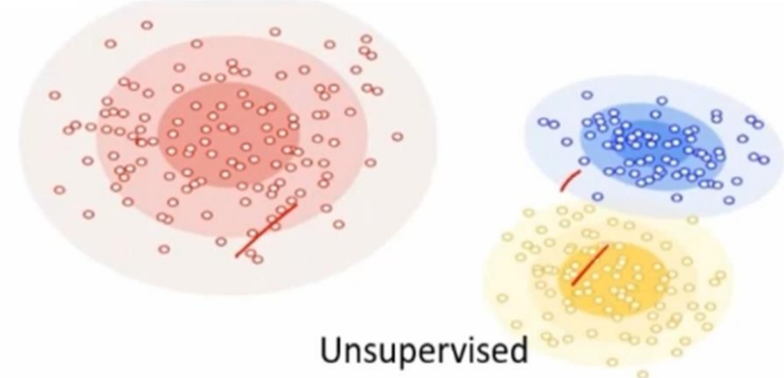
Reinforcement Learning: Learns through rewards and penalties (e.g., robotics, trading algorithms).

Introduction.mp4

Machine Learning:

- Study of algorithms that improve their performance at some task with experience
- Optimize a performance criterion using example data or past experience.





AI learns from data to make predictions or decisions
Example 1: Predict which students may need extra attention
Example 2: Cluster students based on marks to provide targeted support or special classes

Learning by Taking Advice

- It's about formally incorporating guidance (advice) into the learning process to improve model performance, generalization, and efficiency
- Traditional ML: Learn f:X→Y from data D   Training data
- With advice: Learn f:X→Y using (D,A)   Advice
- Types of advice: label advice - Additional labels or corrections
- Feature-level advice: "Feature X is important"
- Constraint-based advice: Add constraints to hypothesis space, $f(x_1) \leq f(x_2)$
- Policy advice (Reinforcement learning): Expert provides action suggestions
- Similarity/Dissimilarity advice: "These two samples should behave similarly"

Cont…

- Advice can Reduce sample complexity, Improve generalization bounds
- It introduces **bias** that Narrows hypothesis space, Speeds up convergence

How advice is incorporated …

- Regularization-based approach: Add advice as a penalty term, $\min_f L(D, f) + \lambda \cdot R(A, f)$

Where L = standard loss, R = advice-based penalty, λ = trade-off parameter

- Bayesian framework: Advice acts as a prior, $P(f|D, A) \propto P(D|f) \cdot P(f|A)$
- Constraint optimization: solve, $\min_f L(D, f)$ subject to advice constraints
- Teacher-student framework: Teacher provides soft guidance. Student learns from Data, Teacher outputs

Learning in Problem Solving

- It refers to the process by which a system improves its ability to solve problems by acquiring better representations, heuristics, and strategies through experience
- It is about improving the efficiency of search and decision-making through experience and abstraction
- A problem-solving system can be defined as ⟨S,A,T,G⟩ where S: State space, A: Actions, T: Transition function, G: Goal condition
- The system learns a function, $\pi: S \rightarrow A$ (policy) or

$V(s): S \rightarrow R$ (value function)

- Types of learning: Learning by experience - improve via interaction with environment
- Used in Reinforcement Learning. Learns optimal decisions through feedback (reward)
- Learning search heuristics: Problem solving often involves *search (e.g.* BFS, DFS)
- Learning improves Heuristic function h(n). E.g. Learning heuristics for Game playing (Chess, Go)

Cont…

- Explanation based learning: Learn from a single example using domain knowledge
- Process: Solve a problem using rules, Generalize the solution, Store as a reusable rule
- E.g. From one proof → derive general theorem
- Leaning by analogy/Case based reasoning: Use past problems to solve new ones
- Steps: Retrieve similar case, Reuse solution, Revise if needed, Retain new experience
- Learning macro operators: Combine sequences of actions into higher-level operators
- Instead of A → B → C → D, Learn, Macro: A → D
- Learning problem representations: Performance depends on representation.
- Learning includes Feature extraction, State abstraction, Dimensionality reduction

Learning from Examples

- It is the process by which a machine learning model infers a general function or concept from a finite set of labelled instances (supervised learning)
- Framework: Training set, $D=\{(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)\}$

Where $x_i \in X$: input (feature vector), $y_i \in Y$: output (label/target)

- Goal: Learn $f:X \rightarrow Y$ such that $f(x) \approx y$
- Hypothesis space: $H=\{h:X \rightarrow Y\}$
- Set of all candidate models. E.g. Neural networks, $h(x)=f(Wx+b)$
- Inductive Bias determines What patterns can be learned, How well the model generalizes
- Learning objective, True risk (generalization): minimize empirical risk (training error), but care about true risk

Cont…

- Learning as Generalization: Generalizing from finite data to unseen instances
- **Generalization gap** is controlled by Sample size *n*, Hypothesis complexity, Data distribution
- **Overfitting**: memorizes examples
- **Underfitting**: fails to capture pattern

Learning paradigms

- Supervised learning: Learn mapping x→y
- Unsupervised learning: Learn structure from x only
- Semi-supervised learning: Combine labelled and unlabelled data
- Reinforcement learning: Learn from interaction (state-action-reward tuples)

Explanation Based Learning

- system uses **prior domain knowledge** to construct a **general explanation** of why a training example satisfies a target concept, and then **generalizes that explanation** into a reusable rule

Components

- Training example: A positive instance of a target concept: $x \in C$
- Target concept: A predicate to be learned: Goal(x)
- Domain theory: A set of **logical rules/axioms**
- Operationality criteria: Defines which predicates are **allowed in the final learned rule**

Process

- Explanation: Construct a **proof** that the example satisfies the target concept using the domain theory; Domain Theory$\vdash$Goal(x)
- This is typically done using Resolution, Backward chaining, Deductive inference
- Generalization: Generalize the proof by Replacing constants with variables, Removing irrelevant conditions, Keeping only necessary constraints
- This produces a **general rule** that explains *all similar cases*
- Operalization: Transform the generalized explanation into a rule that Uses only **operational predicates**

Cont…

Characteristics

- Deductive: uses domain theory to prove correctness
- Inductive: generalizes from a specific example
- Data efficiency: Can learn from **very few examples**
- Knowledge-intensive: Requires a **strong, correct domain theory**, Performance depends heavily on knowledge quality
- Speed-up learning: Learning to improve efficiency rather than accuracy

Formal Learning Theory

- Mathematical framework for understanding What it means to learn from data, when learning is possible, and how well a learned model generalizes to unseen data
- Learning consists of Input space: X, Output space: Y, Unknown distribution: D over X×Y
- Training set: $S=\{(x_1,y_1),\ldots,(x_n,y_n)\}\sim D^n$
- We choose a hypothesis: $h\in H, h:X\rightarrow Y$
- **True Risk (Generalization Error),** $R(h)=E_{(x,y)}\sim D[\ell(h(x),y)]$
- **Empirical Risk (Training Error)** $\hat{R}_S(h) = \frac{1}{n}\sum_{i=1}^{n} l(h(x_i), y_i)$
- Since D is unknown, learning is based on Empirical Risk Minimization
- Generalization problem: Under what conditions does minimizing empirical risk lead to low true risk?

Cont…

- Probably Approximately Correct learning framework: A hypothesis class H is PAC-learnable if For any ϵ>0, δ>0, there exists an algorithm such that: $\Pr(R(h) \leq \epsilon) \geq 1-\delta$

with sample complexity: $$n = O\left(\frac{complexity(\mathrm{H}) + \log(1/\delta)}{\varepsilon}\right)$$

- uniform convergence $\sup_{h \in \mathrm{H}} \left| R(h) - \hat{R}_S(h) \right| \leq \varepsilon$
- This ensures that the Empirical risk approximates true risk **uniformly over all hypotheses**

Learning = **approximate unknown function from data**
Generalization = **core challenge**
Capacity control = **key mechanism**
PAC framework = **foundational model**
Modern ML = **extends beyond classical guarantees**

Connectionist Models

- computational models inspired by the structure and function of the human brain
- cognition emerges from large networks of simple, interconnected **processing units (neurons)**
- knowledge is Distributed across many units, Encoded in connection **weights, Learned through experience** rather than explicitly programmed
- Network architecture: A graph G=(V,E), where V: neurons (units), E: weighted connections
- Common architectures: Feedforward networks (MLPs), Convolutionasl neural networks (CNNs), Hopfield network, Self organizing maps
- Neuron model: Each neuron computes,

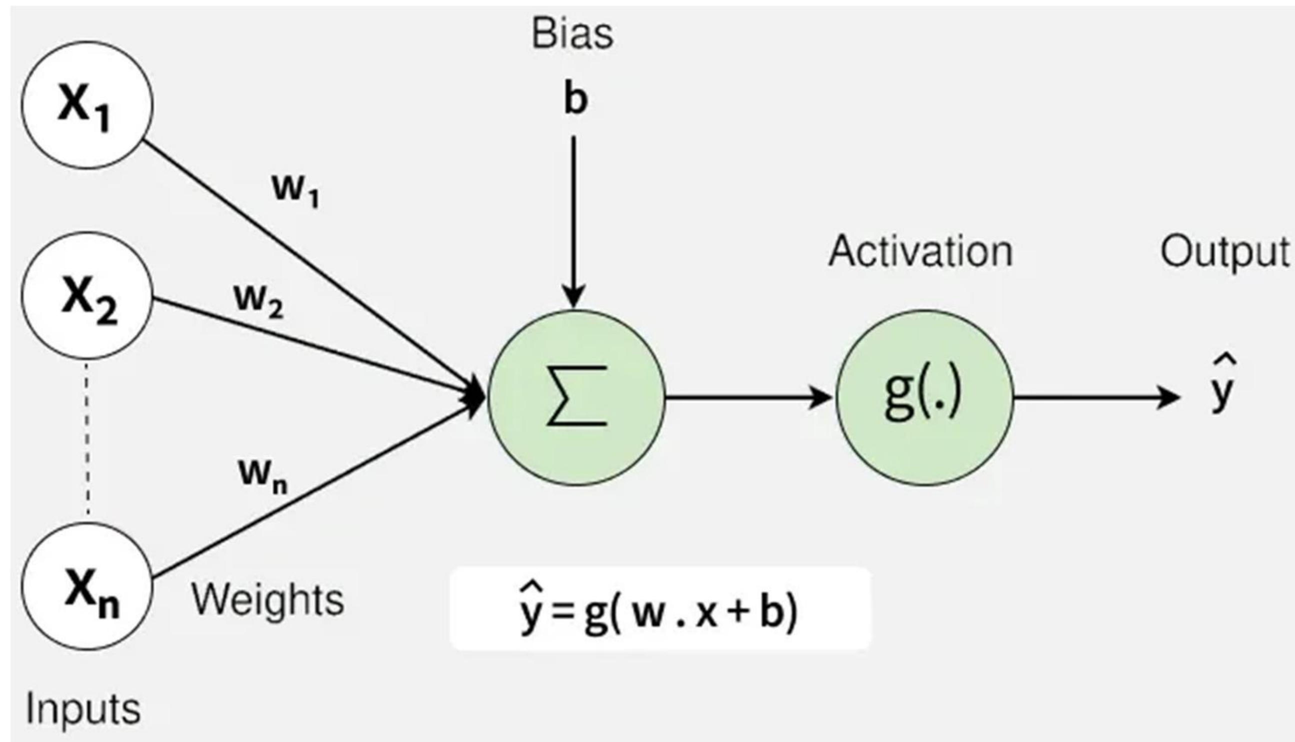
$$a_j = \phi\left(\sum_i w_{ij} x_i + b_j\right)$$

Where x_i: inputs, w_{ij}: weights, b_j: bias, ϕ: activation function (e.g., ReLU, sigmoid, tanh)

- Learning adjusts weights to minimize a loss function L: $w_{ij} \leftarrow w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$

η: learning rate

Cont…



- Used to explain memory representation, Language processing, Pattern recognition
- Properties: parallel distributed processing, Robust to noise/damage, Generalization ability

Hopfield Network

- It is a type RNN
- Used as associate memory (stores patterns and recall them from partial or noisy inputs)
- Structure: Network neurons; Symmetric weights, $w_{ij} = w_{ji}$; Neuron has binary state, s_i Ɛ {-1, 1} or {0, 1}
- Energy function: $E = -\frac{1}{2}\sum_{i,j} w_{ij}s_i s_j + \theta_i s_i$ θ_i is the neuron threshold
- Neuron try to reduce E, and converges to a stable state which corr: to stored memory
- Hebbian learning rule: $w_{ij} = \frac{1}{N}\sum_{\mu=1}^{P} \xi_i^{\mu}\xi_j^{\mu}$, $w_{ii}=0$
- This ensures that each stored pattern is a stable attractor
- Applications: Pattern recognition, Error correction, Optimization problems

Genetic Algorithm – Machine Learning

- A **genetic algorithm** is a **heuristic search** algorithm that is inspired by Charles Darwin's theory of natural evolution.
  Commonly used for solving complex optimization and search problems
- This algorithm reflects the process of **natural selection** where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.
- Genetic Algorithms are being widely used in different **real-world applications**, for example, **image processing**, **Designing electronic circuits**, **code-breaking**, and **artificial creativity**.

How Genetic Algorithm works?

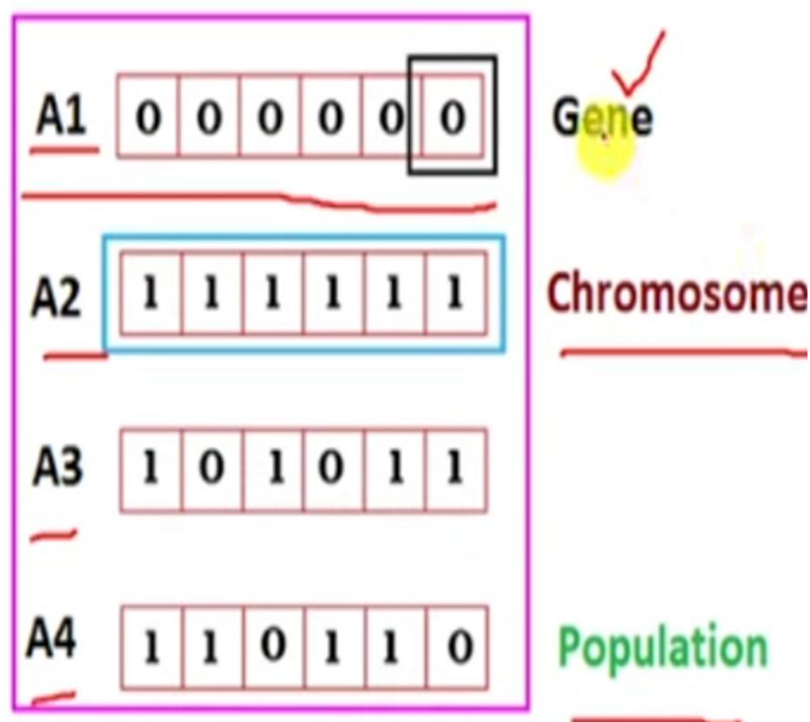
There are five phases in Genetic Algorithm:

- Initialization
- Fitness Assignment
- Selection
- Crossover (Reproduction)
- Termination

How Genetic Algorithm works?

Initial Population

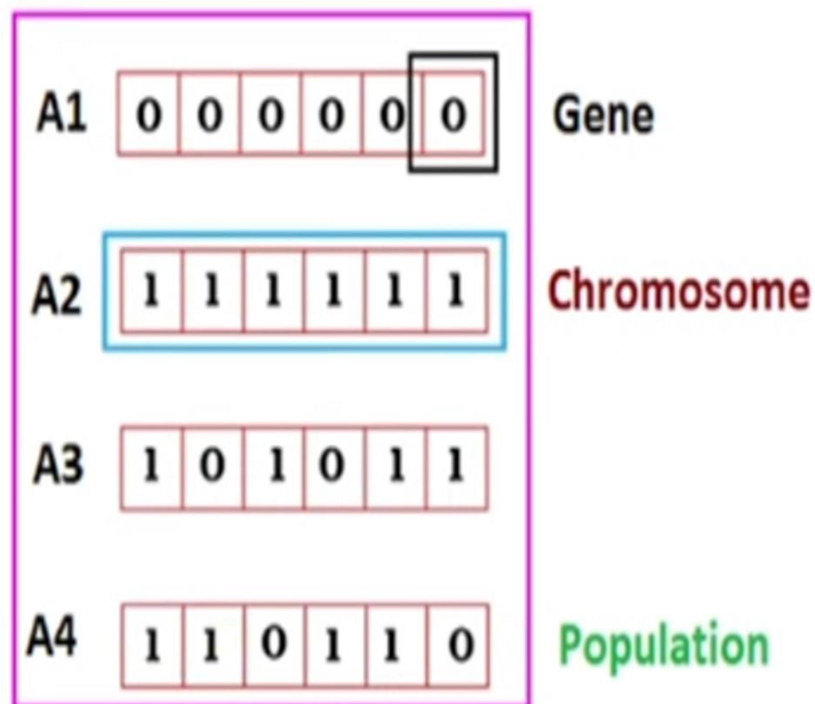
- The process begins with a set of individuals which is called a Population.
- Each individual is a solution to the problem you want to solve known as Chromosome.
- An individual is characterized by a set of parameters (variables) known as Genes.
- Genes are joined into a string to form a Chromosome (solution).



How Genetic Algorithm works?

Fitness Function

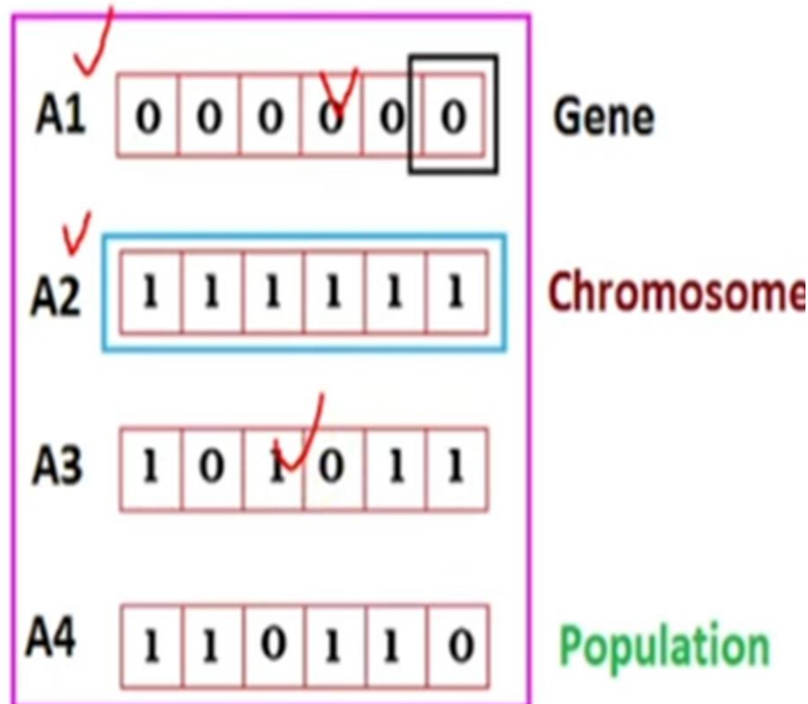
- The fitness function determines **how fit** an individual is? (the ability of an individual to compete with other individuals).
- It gives a fitness score to each individual.
- The probability that an individual will be selected for reproduction is based on its **fitness score**.



How Genetic Algorithm works?

Selection

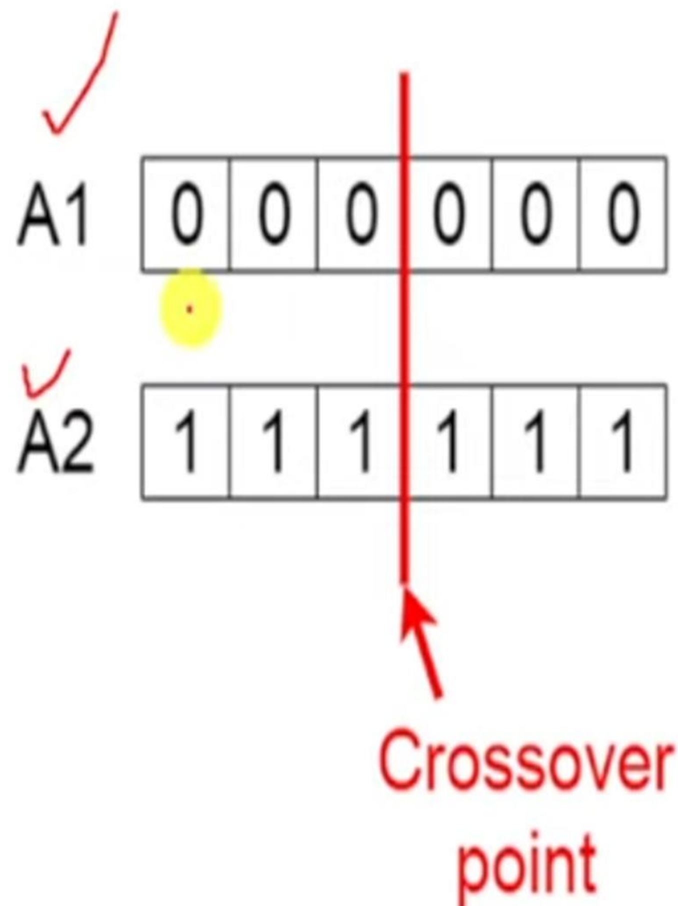
- The idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation.
- A pair of individuals (parents) are selected based on their fitness scores.
- Individuals with high fitness have more chance to be selected for reproduction.



How Genetic Algorithm works?

Crossover

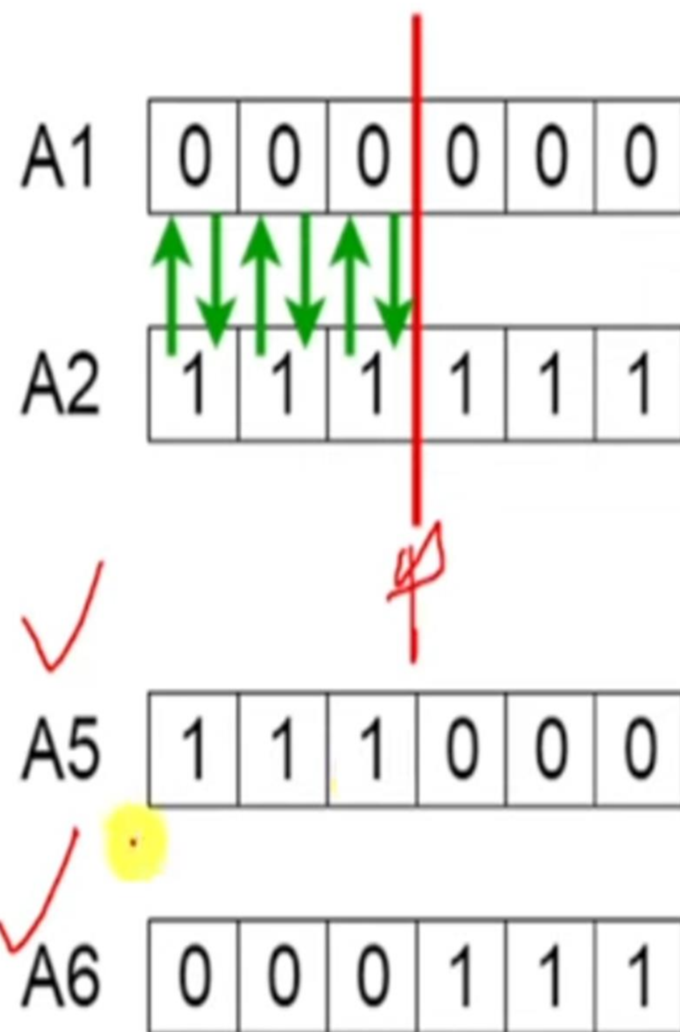
- Crossover is the most significant phase in a genetic algorithm.
- For each pair of parents to be mated, a crossover point is chosen at random from within the genes.
- For example, consider the crossover point to be 3 as shown.



How Genetic Algorithm works?

Offspring

- **Offspring** are created by exchanging the genes of parents among themselves until the crossover point is reached.
- The new offspring are added to the population.



How Genetic Algorithm works?

Mutation

- In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability.
- This implies that some of the bits in the bit string can be flipped.

Before Mutation

A5	1	1	1	0	0	0

After Mutation

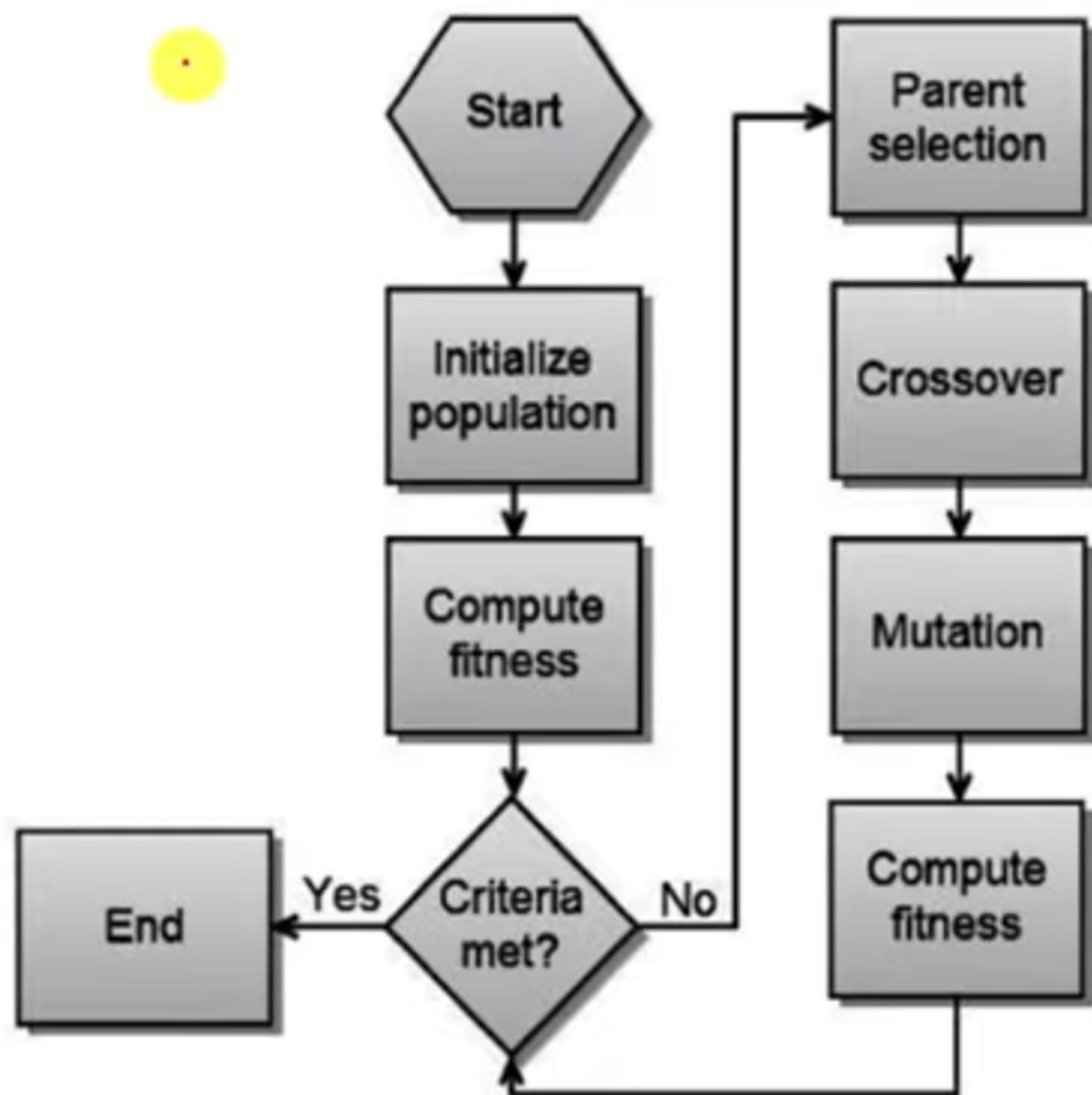
A5	1	1	0	1	1	0

How Genetic Algorithm works?

Termination

- The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation).
- Then it is said that the genetic algorithm has provided a set of solutions to our problem.

How Genetic Algorithm works?



Genetic Algorithm

- Problem: We consider an optimization problem:

We aim to $\max_{x \in X} f(x)$ (or minimization via transformation)

X: search space (often large, discrete, non-convex)

f(x): fitness function (objective)

- Representation (Encoding): A solution $x \in X$ is encoded as a **chromosome**:
- Binary strings (classic): $x \in \{0,1\}^n$
- Real-valued vectors (continuous GAs)
- Permutations (e.g., TSP)
- Trees (Genetic Programming)
- Population: A multiset $P_t = \{x_1, x_2, \ldots, x_N\}$ at generation t
- Fitness function: Maps each individual to a scalar: $F(x) = f(\phi(x))$
- May include penalties for constraint violations
- Selection operator: common methods are
- Fitness-proportionate (roulette wheel): $P(x_i) = \dfrac{F(x_i)}{\sum_j F(x_j)}$
- Tournament selection, Rank-based selection

Cont…

- Genetic operators: Crossover (recombination)
- Combines genetic material from parents: Single-point crossover, Multi-point crossover, Uniform crossover
- Formally: $(x^{(1)},x^{(2)}) \rightarrow (y^{(1)},y^{(2)})$
- Mutation: Random perturbation: Bit-flip mutation (binary), Gaussian mutation (real-valued)
- Mutation ensures **ergodicity** (ability to explore entire search space)
- Replacement strategy: Defines how new generation is formed: Generational replacement, Steady-state replacement, Elitism (preserve best individuals)
- Applications: Combinatorial optimization (TSP, scheduling), Neural architecture search, Feature selection, Engineering design optimization, Game AI and strategy evolution

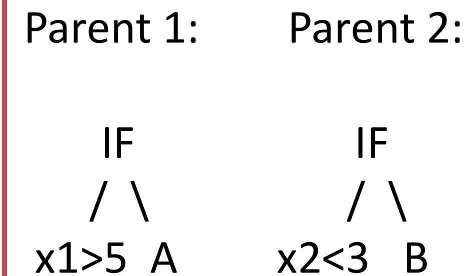
Genetic Programming based Classifier

- A program synthesis problem over a hypothesis space of executable decision structures
- A classifier system aims to learn a mapping: $h: X \rightarrow Y$
- X: input space (features), Y: class labels
- GP-based classifier systems evolve **programs or rules** that act as classifiers
- Representation: Tree-based: Each individual is a **program tree**
- This is represented as a tree: *Internal nodes*: functions/operators (IF, >, <, logical ops), *Leaves*: features, constants, class labels
- Two main paradigms: *Pittsburg approach*
- Each individual = **complete classifier;** Fitness = performance on entire dataset
- *Michigan approach*: Each individual = **single rule;** Population = rule set

```
IF (x1 > 5) THEN
   IF (x2 < 3) THEN Class A
   ELSE Class B
ELSE Class C
```

Learning Mechanism

- Initialization: Generate random programs (trees): Random conditions, Random structure
- Fitness evaluation: $F(h)=\alpha \cdot accuracy+\beta \cdot parsimony\ penalty$
- Where Accuracy = classification correctness, Parsimony penalty = discourages overly large trees (bloat control)
- Genetic operators: Crossover (subtree exchange)
- Two parent trees exchange subtrees
- Swap subtrees → new classifiers
- This enables recombination of **decision logic**
- Mutation: Replace subtree, Modify condition (e.g., change threshold), Change class label
- Selection: Bias toward better classifiers: Tournament selection common, Promotes high-accuracy programs
- Iteration: Over generations: Programs become more accurate, Structures adapt to data patterns



Role of GP

- Automatic feature extraction: GP can evolve expressions like: $(x_1 \cdot x_2)+\log(x_3)$
- Rule discovery: GP discovers rules such as:
  IF (age > 40 AND cholesterol > 200) → disease
- Unlike fixed models: No predefined structure, Rules emerge dynamically
- Model structure learning: GP simultaneously learns: Model parameters, Model structure
- Key challenges: Bloat (code growth)
- Programs grow without improving fitness
- Mitigation: Depth limits, Parsimony pressure: $F' = F - \lambda \cdot \text{size}(h)$
- Overfitting: Programs memorize training data
- Solutions: Cross-validation, Regularization (tree size control)
- Closure and Sufficiency: Function set must satisfy:
- **Closure**: all operations valid for all inputs
- **Sufficiency**: can represent target function

Artificial Life

- It is the study of man-made systems that exhibit behaviours characteristic of natural living systems
- ALife is "bottom-up" - You define local rules for individual agents, and complex, global behaviour emerges from their interactions
- It uses Genetic Algorithms (GAs) or Evolution Strategies (ES)

Types of Artificial Life

- Soft ALife (computational): Uses simulations and algorithms. E.g. GA
- Hard ALife (robotics): Physical robots exhibiting life-like behaviour. E.g. Swarm robots
- Wet ALife (biochemical): Synthetic biology and chemical systems. E.g. artificial cells

Key Concepts

- Emergence: Global behaviour arises from local interactions
- Self organization: Systems organize without central control
- Evolution: Adaptation via Mutation, Selection, Reproduction
- Adaptation: Agents learn or evolve based on environment

Society Based Learning

- It is a paradigm where learning emerges through interaction among agents in a social environment

Components

- Agents: Autonomous entities with Goals, Knowledge, Learning capability
- Environment: Shared space where agents interact
- Social interaction: Collaboration, Negotiation, Imitation

Learning mechanisms

- Imitation learning: Agents learn by copying others
- Reinforcement learning: Rewards influenced by group behaviour
- Evolutionary learning: Strategies evolve across populations
- Cultural learning: Knowledge spreads like memes
- Applications: Smart cities, Traffic systems, Distributed AI systems, Recommendation systems, Social simulations