AI unit 1

PAGE NO.
DATE

Turing Test and Rational Agent Approaches
State Space Representation of Problems
Heuristic Search Techniques
Game Playing → Min Max Search
Alpha Beta Cut off Procedures

AI is the branch of computer science that deals with creating machines or software that can think, learn, and act intelligently like humans.
AI → giving machines the ability to perform tasks that normally requires human intelligence.

Key features of AI
Perception : Understanding input from environment (eg. vision, speech)
Reasoning : Drawing conclusions and making decisions
Learning : Improving performance from past experience (ML)
Problem-solving : Finding solutions using search, heuristics, or planning
Acting : Taking rational actions in real-world environments

Types of AI
→ based on capabilities
(i) Narrow AI (weak AI) : specialized for a single task or specific task or a narrow range of tasks.
- Highly specialized and operate within a limited context.
- Cannot generalize their knowledge or apply it to tasks outside their designated function

Ex. Chatbots, Google Maps, Face recognition

(ii) General AI (strong AI) : can perform any intellectual task that a human can do with the ability to generalize knowledge and apply it to different contexts.

AI is the science of making machines act intelligently — to perceive, reason, learn, and decide like humans.

- It possess the ability to understand, learn and apply knowledge across a wide range of tasks.
- Broad intelligence → can perform a variety of tasks, making it versatile and adaptable.
- Human-like reasoning
- Self-learning → capable of learning and improving over time, adapting to new situations and acquiring new skills without human intervention

Ex. General AI is theoretical, not yet achieved fully, but research is ongoing.

iii) Super AI : Most advanced form of AI, surpassing human intelligence
Super AI is still a theoretical concept

Examples of AI Applications
- Virtual assistants (Siri, Alexa)
- Self driving cars
- Medical diagnosis
- Game playing (Chess, Go)
- Fraud detection in Banking.

Turing Test

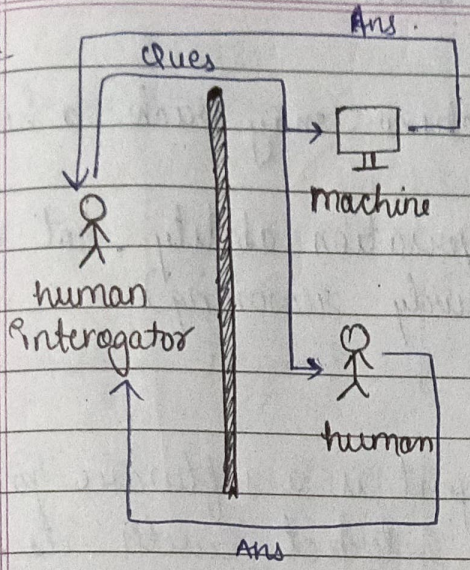
- Proposed by Alan Turning (1950)
- Turing asked "Can machines think?"

Instead of directly answering, he proposed an "limitation game" → known as Turing Test

Test is when a person can't tell whether ans is given by machine or human

Turing test is a widely recognized benchmark for evaluating a machine's ability to demonstrate human like intelligence

Intelligence → ability to exhibit human-like behavior.



Setup: A human interrogator communicates via text with two participants
1. A human
2. A machine (computer)

Both try to convince the interrogator that they are human.

Goal: If the interrogator can not reliably tell the machine from the human, the machine is said to have passed the Turing Test

Criteria for Turing Test

The test does not require the machine to be correct or logical in its response but rather to be convincing in simulating human conversation.

The test is fundamently about deception — the machine must fool the judge into believing that it is human.

Key criteria:

(i) Natural Language Processing (NLP): machine must understand and generate human language fluently.

(ii) Knowledge Representation: Machine needs to handle and manipulate knowledge to provide contextually relevant responses.

(iii) Reasoning: Machine should demonstrate some form of logical reasoning, even if flawed, to sustain a convo.

(iv) Learning: Machine should learn from its environment

Ex Judge: Are you a computer? → No.
Multiply 29878978 to 198810498 → (After long pause) wrong answer

Add 9187 with 6843 → (After 20sec) gives right answer

Turing Test is still philosophically important but not a practical benchmark today.
Modern AI is evaluated by performance metrics (accuracy, efficiency) not just by imitation

PAGE NO.
DATE

Limitations

1. Deception over intelligence : A machine may trick a human without truly 'understanding'.
2. Narrow measure : only tests conversation ability, not other forms of intelligence (vision, creativity, reasoning)

Rational Agents in AI

An AI agent is a software program that can interact with its environment, collect data and use data to perform self-directed tasks.

An agent is anything that perceives its environment through sensors and acts upon that environment using actuators

Agent = perception + Action

Key features of Agent

- Autonomous : Act without constant human input and decide next steps from past data
- Goal driven : Optimize for defined objectives
- Perceptive : Gather info from sensors, inputs or APIs
- Adaptable : Adjust strategies when situation change
- Collaborative : work with humans or other agents towards shared goals.

Agent function : Maps percepts (inputs) to actions

$$f : P^* \rightarrow A$$

Agent program : Implementation of the agent function that runs on the agent's hardware.

A rational agent is one that chooses actions that maximize its expected performance measure given the knowledge and percepts it has.

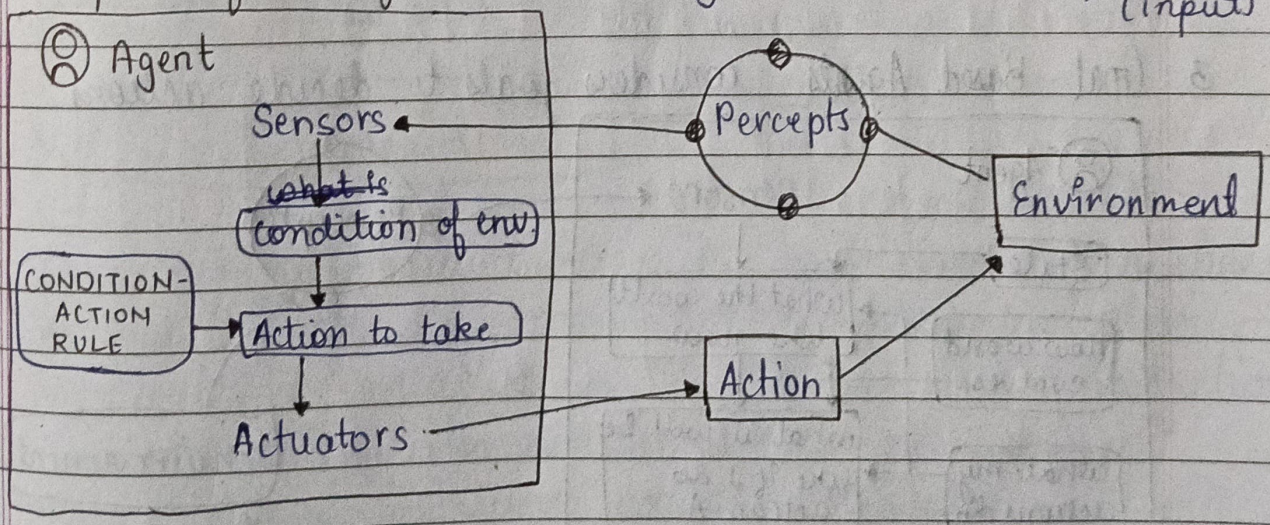
PAGE NO.
DATE

AI Agents Classification

- Reactive agents : respond to immediate environmental stimuli without foresight or planning
- Proactive agents : anticipate future states and plan actions to achieve long-term goals
- Single-agent system : One agent solves a problem independently
- Multi-agent systems : Multiple agents interact, coordinate or compete to achieve goals ; may be homogeneous (same ~~goals~~ roles) or heterogeneous (diverse roles)
- Rational agents : Choose actions to maximize expected outcomes using both current and historical information

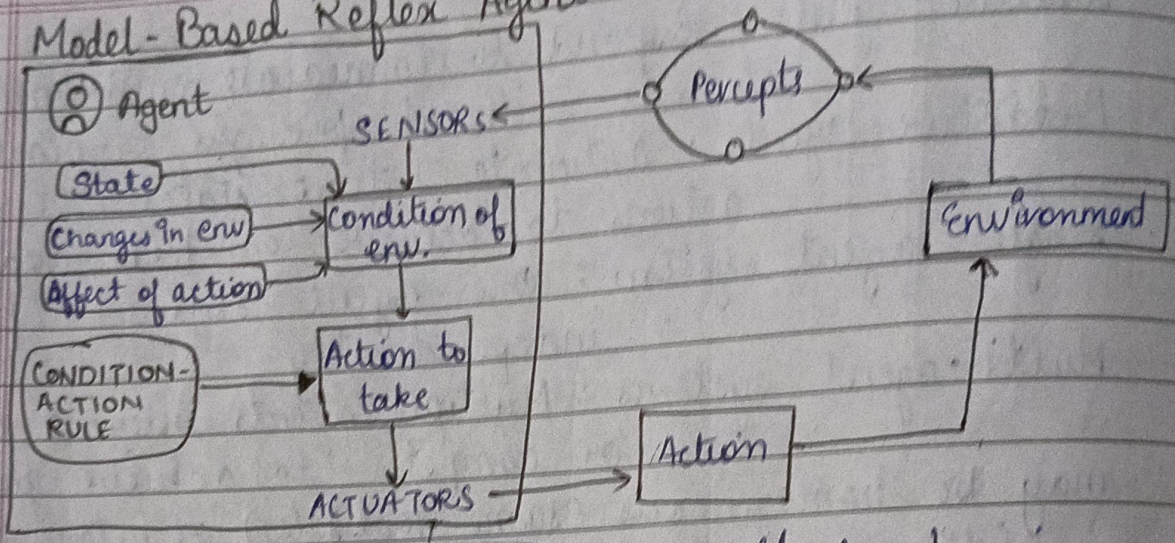
Types of Agents

1. Simple Reflex Agents : Acts only on the current perception (input)



These agents respond directly to stimuli without considering past experiences or potential future states.
Uses condition-action rules ("If-then" rules)
No memory of past states
Not very intelligent → works on fixed rules.
Ex - Traffic light controls on current traffic situation
- Thermostat ("If temp < 18°C → then turn on heater")

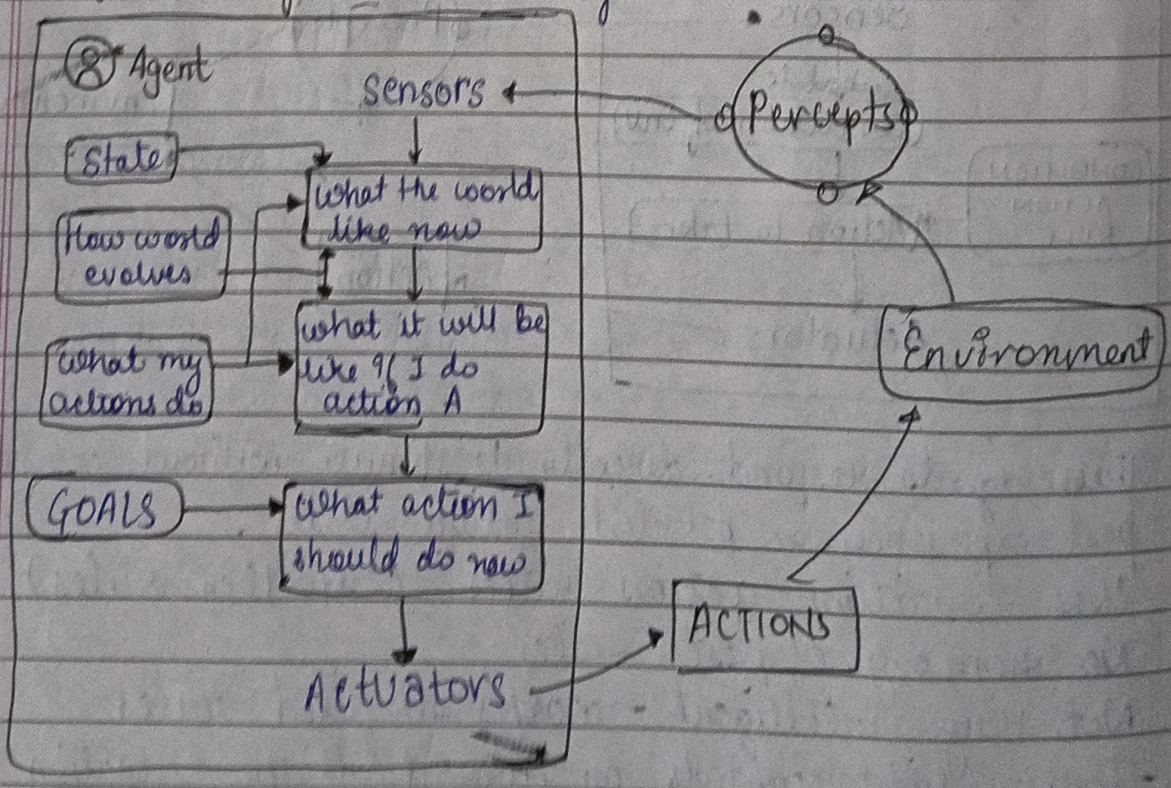
2. Model-Based Reflex Agent



- Maintains an internal model of world (env.)
- Function effectively in partially observable environments. This internal model helps them make more informed decisions by considering how the world evolves and how their actions affect it.

Ex. Robot vacuum cleaners that map rooms and tracks cleaned areas.

3. Goal-Based Agents : considers goals to decide actions

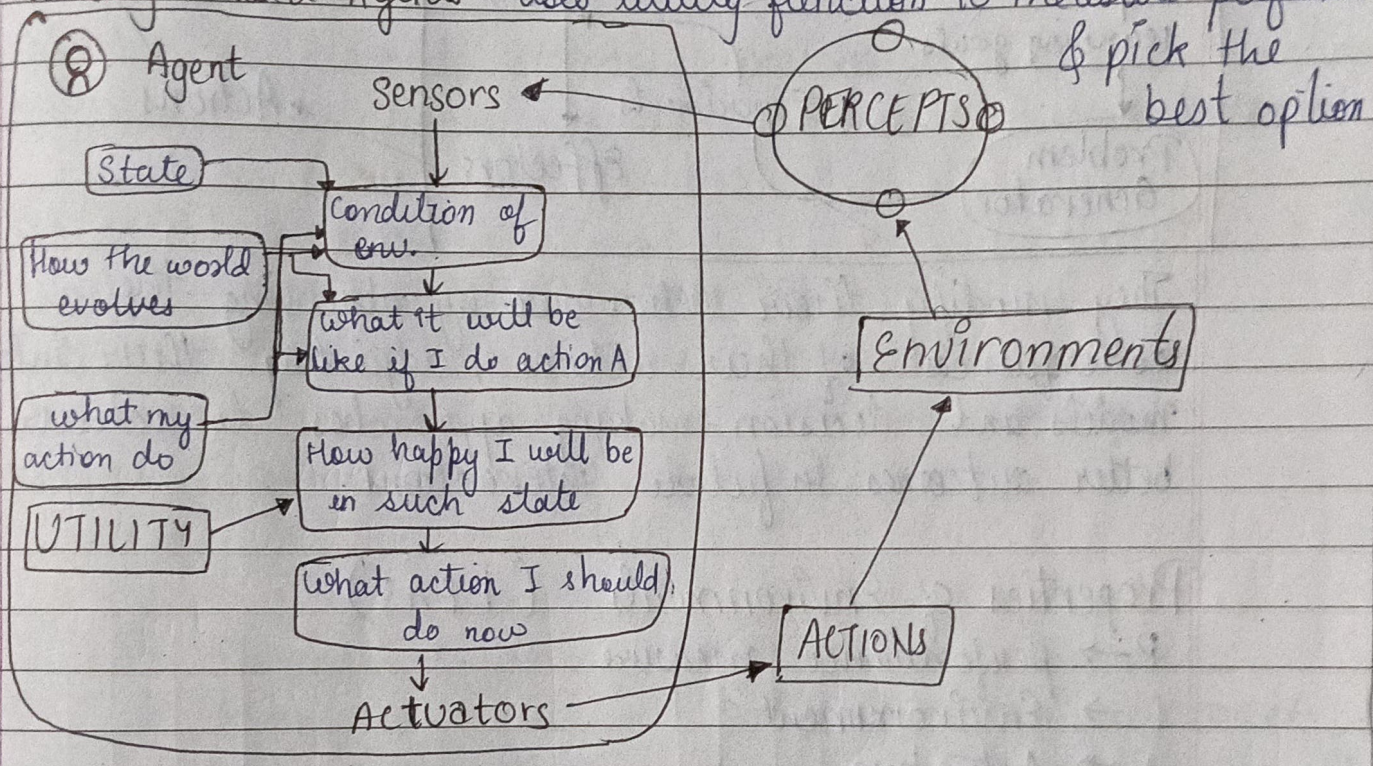


PAGE NO.
DATE

- Not flexible → have fixed goal
- Evaluate how different action sequences might lead towards their defined goal, selecting the path that appears most promising.
- Employ search and planning mechanisms.
- May explore multiple possible routes to goal

Ex GPS navigation system choosing a path to destination

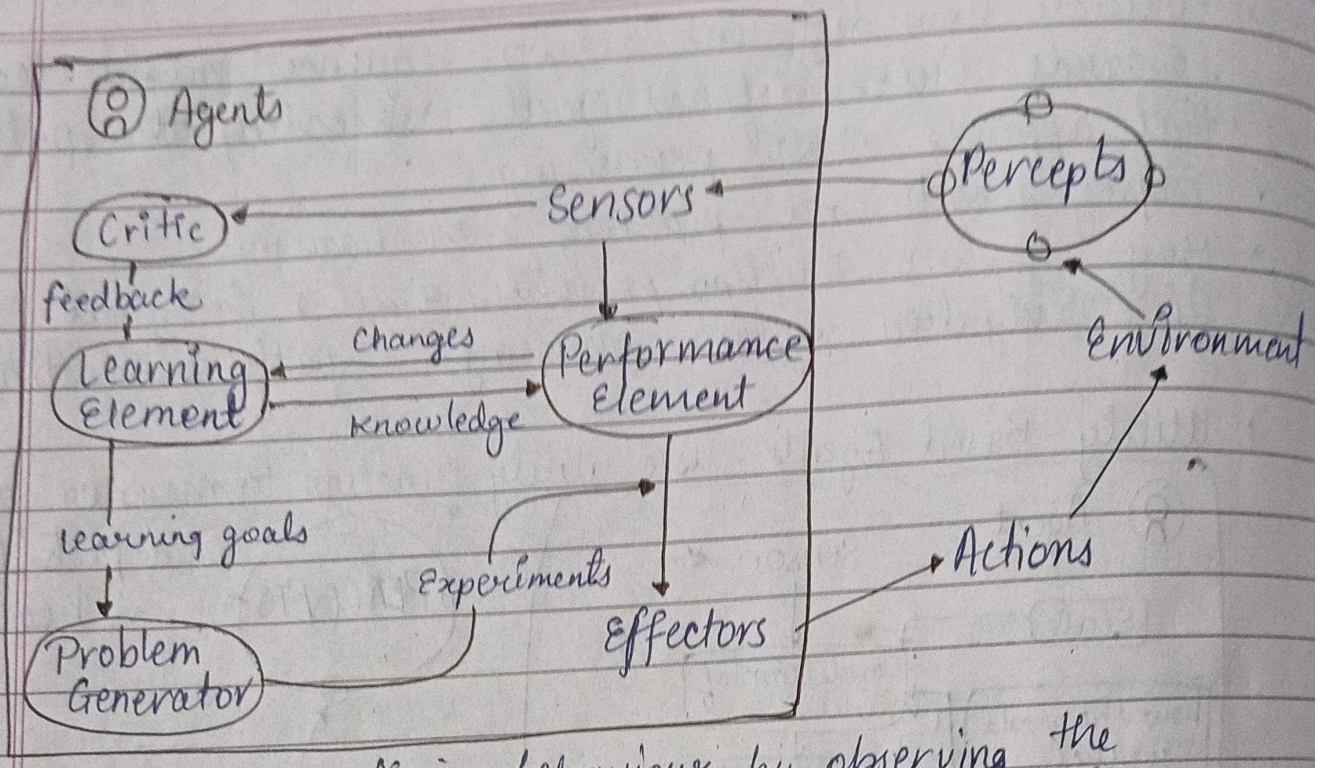
4. Utility-Based Agents : uses utility function to measure performa & pick the best option



- It extend goal-based thinking by evaluating actions based on how well they maximize a utility function — essentially a measure of "happiness" or "satisfaction".
- Handle probabilistic and uncertain environments
- Make rational decisions under constraints

Ex Investment AI maximizing profit, minimizing risk.

5. Learning Agents : Improves performance with experience
- Has components
  - Learning element (improves with feedback)
  - Performance element (selects actions)
  - Critic (gives feedback)
  - Problem generator (suggests exploratory actions)



They modify their behaviour by observing the consequences of their actions, adjusting their internal models and decision-making approaches to achieve better outcomes in future interactions

Properties of Environments (PEAS)

P → performance measure
E → Environment
A → Actuators
S → Sensors

Components of Rational Agents

- Perception: The ability to perceive the environment through sensors
- Knowledge base: Information the agent has about the environment and itself
- Decision-making process: Algorithms and rules that guide the agent's actions
- Action: The ability to perform actions that affect the environment through actuators

→ Ex. for self-driving car

Design Principles of Rational Agents

- Performance measure: evaluates how well the agent is achieving its goals.
  + Speed, safety, passenger comfort
- Rationality: acting optimally given the info and computational resources available.
  → slowing down when a pedestrian is crossing
  Sensors → cameras, GPS, radars
  Actuators → steering, brakes, accelerator
- Autonomy: can make decisions without human intervention

Applications of Rational Agents

- Robotics (operate robots to perform tasks)
- Financial trading (make buy-sell decisions)
- Health Care (assist in diagnosis & treatment planning)
- Game AI (controls non-player characters) NPC

Early AI (Turing test) focused on mimicking humans.
But AI today is built on rational agent approach.
→ Instead of asking "Can machines think like humans?", it asks → "Can machines act rationally to achieve goals?"

Advantages over Turing Test	
• Turing Test	• Rational Agent Approach
- Focused on human like conversation	- General framework (not restricted to conversation)
- Subjective, Imitation-based, limited	- Works for any environment (games, robotics, healthcare etc.)
	- Objective → maximizing measurable performance

In AI, problem solving is often modeled as a search through a state space. ohh, that's why we study searching algo.

State Space Representation → formal way of defining a problem as search

A State Space is the set of all possible states (configurations) that can be reached in the problem.
[problem solving → moving from the initial state to the goal state using operators (actions)]

Components of a state space problem

- Initial state → The starting point of the problem
- Goal state → Desired condition to be achieved
- State space → The set of all possible states reachable from initial state by applying actions.
- Actions → Rules for moving from one state to another.
- Path cost → Sequence of states from start to goal, path cost helps compare solutions

Ex. 8 puzzle problem

- Initial state: A scrambled 3×3 board with tiles 1-8 and a blank space

6	7	1
2	8	5
3		4

3×3

- Goal state: Tiles arranged in order (1-8) with blank space in bottom-right

1	2	3
4	5	6
7	8	

- Operators (Actions): Move blank tiles UP / DOWN / LEFT / RIGHT
- State space: All possible arrangements of tiles (9! = 362880 states)
- Path cost: no. of moves

Advantages of State Space Representation

→ Provides formal framework for problem solving
→ Makes it easier to apply search algorithms (DFS, BFS, A*)
→ Useful for heuristic design

State space representation defines a problem as search through states, from an initial state to a goal state, using operators, forming the basis of AI problem solving.

Limitations:
→ State space may be too large (combinational explosion)
→ Requires efficient search strategies (heuristic, pruning)

Unit 2

UNINFORMED v/s INFORMED SEARCH

- Uniformed search (Blind search) (no heuristic)
  - Search strategies that do not use any domain knowledge
  - They only use the problem definition (initial state, action, goal state)

  ⇒ Explore the state space blindly until the goal is found.

Ex BFS, DFS
+ Gauranteed to find optimal solution (if exist)
+ Simple to implement
- May explore huge parts of state space unnecessarily
- Very inefficient for large problems (high time, space complexity) ~~Time consu~~

- Informed Search (Heuristic Search)
  - Search strategies that use problem-specific knowledge (heuristic) to guide search towards the goal

  ⇒ Use a heuristic function $h(n)$ = estimated cost from node n to goal

Ex A^* Search ($f(n) = g(n) + h(n)$), Best first search
+ Much more efficient (reduce search space) (less time, space complexity)
+ Often find optimal solution faster
- Requires a good heuristic (designing heuristics can be hard)
- Performance depends on accuracy of heuristic

State space representation defines a problem as search through states, from an initial state to a goal state, using operators, forming the basis of AI problem solving.

Limitations:
→ State space may be too large (combinational explosion)
→ Requires efficient search strategies (heuristic, pruning)

Unit 2

UNINFORMED v/s INFORMED SEARCH

- Uniformed search (Blind search) (no heuristic)
  - Search strategies that do not use any domain knowledge
  - They only use the problem definition (initial state, action, goal state)
  - ⇒ Explore the state space blindly until the goal is found.

Ex BFS, DFS

+ Guaranteed to find optimal solution (if exist)
+ Simple to implement
- May explore huge parts of state space unnecessarily
- Very inefficient for large problems (high time, space complexity)
  ~~Time consuming~~

- Informed search (Heuristic Search)
  - Search strategies that use problem-specific knowledge (heuristic) to guide search towards the goal
  - ⇒ Use a heuristic function $h(n)$ = estimated cost from node n to goal

Ex A^* Search ($f(n) = g(n) + h(n)$), Best first Search

+ Much more efficient (reduce search space) (less time, space complexity)
+ Often find optimal solution faster
- Requires a good heuristic (designing heuristics can be hard)
- Performance depends on accuracy of heuristic

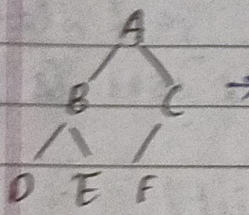
unit 2

BFS (Breadth first search)

- Uninformed (blind) search algorithm which explores the state space level by level

Algorithm

1. Put the start node (root → initial state) in queue (FIFO)
2. Repeat until queue is empty
   → remove from front node
   → if its goal state, return solution
   → otherwise, add all of its unvisited children to the queue



→ A B C D E F

- Complete → always find solution if exists
- Optimal → give shortest path
- Time Complexity : $O(b^d)$ — b → branching factor
- Space Complexity : $O(b^d)$ — d → depth of shallowest factor

Use case → to find shortest path
when state space is not too large (memory expensive)

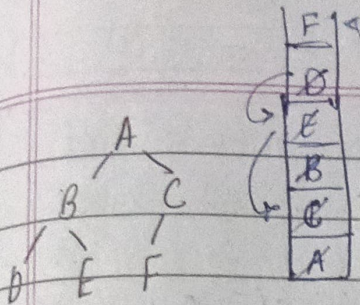
unit 2

DFS (Depth first Search)

- Uniformed (blind) search algorithm which explores a state space by going as deep as possible along one branch before backtracking.

Algorithm

1. Put the start node (root → initial state) in stack (LIFO)
2. Expand the node
   → if its the goal state, stop
   → otherwise, visit the first unvisited child
3. If no children left, backtrack to explore other paths



- Incomplete (may get stuck in infinite paths in infinite depth trees)
- Not optimal (may find longer path before the shortest one)
- Time complexity : $O(b^d)$
- Space complexity : ~~$O(b^d)$~~ $O(bd)$

Use case → when memory is limited
when the search tree is very wide but not very deep

unit 2

BIDIRECTIONAL SEARCH

- Runs two simultaneous searches :
  - One forward from initial state
  - One backward from goal state

When two searches meet in the middle, a path is found

→ Instead of searching depth d, each search goes about d/2, making it faster

A — B — C — D — E

Start Goal

forward search : A → B → C
Backward search : E → D → C
Meet at C, path found ≈

Algorithm

1. Initialize two frontiers (queues) → forward (start) & Backward (goal)
2. Expand nodes alternately from both frontiers
3. If a node is found in both frontiers → path is complete
4. Reconstruct solution by joining forward and backward paths

- Optimal (if using BFS both sides)
- Time complexity : $O(b^{d/2} + b^{d/2}) = O(b^{d/2})$
- Space complexity : $O(b^{d/2})$

Disadvantage → need to know goal state in advance

~~BEST FIRST SEARCH~~.

DEPTH LIMITED SEARCH (DLS)

DLS is a DFS with a cutoff depth limit l

→ it explores nodes only up to a fixed depth l

→ if the goal is beyond that depth, it will not be found

Helps avoid getting stuck in infinte paths (a problem in DFS.)

~~Algorithm~~ Algorithm

1. Start at root node
2. Perform DFS but stop if the curred depth = limit l
3. If goal is found within depth $\leq l$, return solution
4. Otherwize return cutoff (meaning deeper search required)

- ~~to~~ not complete, if limit < depth of solution
- Not optimal
- Time Comp.: $O(b^l)$
- Space Comp: $O(bl)$.

\+ prevents infite loop (unlike DFS)

\+ less memory than BFS

\- If limit is too small → may miss the sol.

\- If limit is too large → behave like DFS

Heuristic Search Approach

Misplaced tiles
Manhattan dist
Euclidean dist
Chebyshev dist

A Heuristic is a problem ~~searching~~ specific rule or function that provides an estimate of how close a given state is to the goal

→ $h(n)$ = estimated cost from node n to goal

It does not gaurantee accuracy, but helps guide the search efficiently.

Heuristic search is a search strategy that uses heuristic information (domain knowledge) to prioritize which path to explore

PAGE NO.
DATE

Unlike uninformed search (blind), heuristic search is goal-directed.

Heuristic Algorithms
Greedy Best First Search ($f(n) = h(n)$ → manhattan dist.)
A* Search ($f(n) = g(n) + h(n)$)
Beam Search, Hill Climbing

Properties
- Complete (A* is complete, hill climbing is not)
- Optimal (depends on heuristic)
- Time complexity (less than uninformed search, but depends on heuristic quality)
- Space complexity (often high → A* stores all nodes in memory)

Advantages
- Much more efficient than uninformed methods
- Finds solution faster by focusing on promising paths

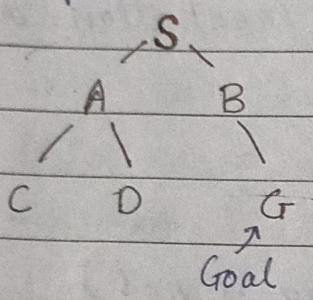
Disadvantages
- Requires good heuristic function (designing one can be hard)
- Bad heuristic can make search worse than uninformed search.

ITERATIVE DEEPENING SEARCH (Iterative deepening DFS)
IDS is a hybrid of BFS and DFS.
- it performs depth-limited search (DLS) repeatedly with increasing depth limits
- at depth limit 0, 1, 2 .... d until the goal is found

⇒ It combines the space efficiency of DFS with the completeness & optimality of BFS

Working

1. Start with depth limit = 0 → run DLS
2. If goal not found, increase depth limit to 1 → run DLS again
3. Repeat until goal is found.



Depth = 0 ; only checks S
Depth = 1 ; Checks S, A, B
Depth = 2 ; Checks S, A, B, C D, G
found G → goal state.

Properties

Complete (like BFS)
Optimal (
Time complexity : $O(b^d)$, same as BFS
Space complexity : $O(bd)$, same as DFS (much better than BFS)

Advantages

- Uses much less memory than BFS
- Gaurantees findings the optimal solution (unlike DFS)
- Works well in large, infinite, or unknown depth problems.

Disadvantages

- Some nodes are re-expanded multiple times, which increase overhead.
- But since most nodes are near the deepest level, extra cost is not too high

BEST FIRST SEARCH (heuristic search)

It is a heuristic search that explores the most promising node first, based on an evaluation function:

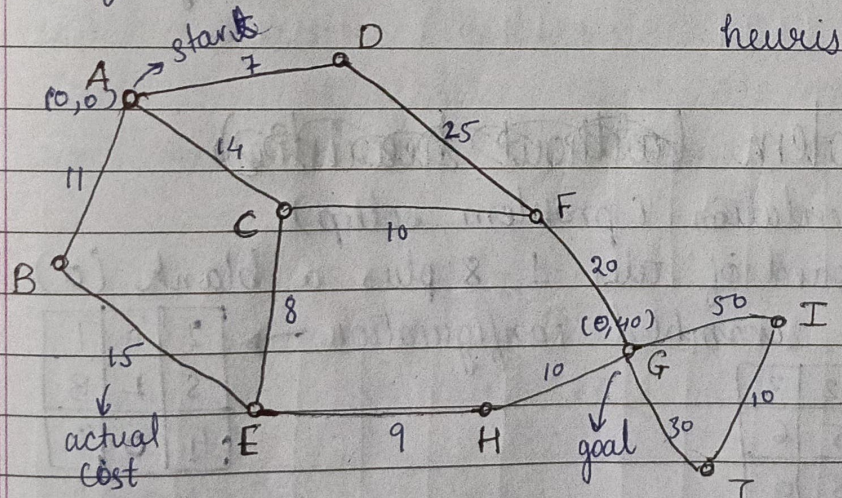
$$f(n) = h(n)$$

where, $h(n)$ = heuristic estimate of cost from node n to goal

→ It expands the node that appears to be closest to the goal

Algorithm

1. Put the start node into a priority queue
2. While the queue is not empty.
   - Remove the node with lowest heuristic value $h(n)$
   - If its goal → stop.
   - Else expand its children and add them to the priority queue.

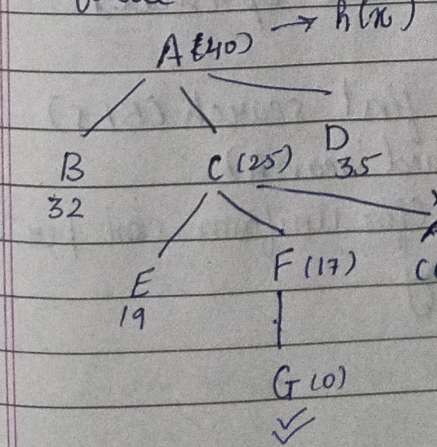


heuristic func = manhattan dist

$$h(x) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

A → G = 40
B → G = 32
C → G = 25
D → G = 35
E → G = 19
F → G = 17
G → G = 0 (Goal)
H → G = 10

priority queue = A C F G



path = A → C → F → G
cost = 14 + 10 + 20 = 44

other possible paths (without heuristic)
A → B → E → H → G
cost = 41

- Complete (if graph is finite)
- Optimality not guaranteed (may choose a locally good path, but miss global optimum)
- Time Complexity : $O(b^d)$ worst case
- Space Complexity : Keeps all nodes in memory (can be large)

\+ faster than blind searches (BFS/DFS)
\+ Uses heuristic, so guided towards goal
\- Not always optimal
\- Depends heavily on the quality of heuristic

Best first Search = Greedy heuristic search → expands node with the lowest h(n) first
"always go where it looks closest to the goal."

8-puzzle problem ~~(without heuristic)~~

→ State Space representation (problem setup):
State: a 3x3 board of tiles 1-8 plus a blank (0)
Initial state : any scrambled configuration →

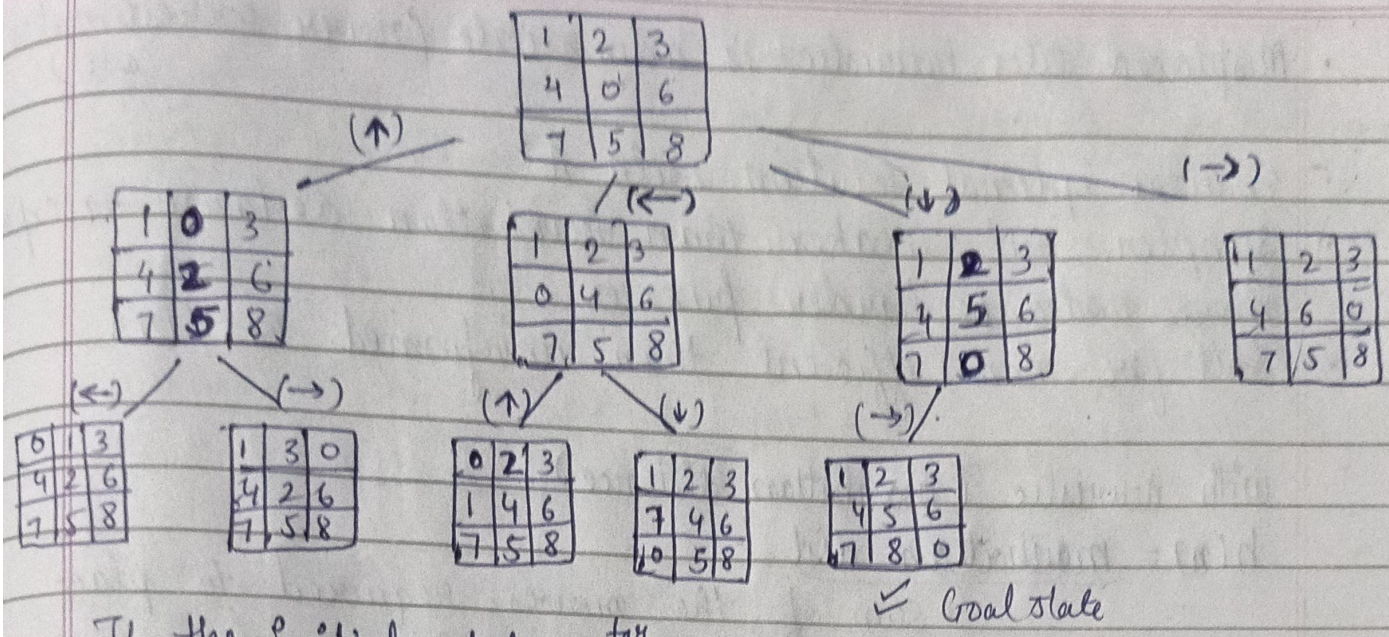
2	6	7
5	1	3
4	0	8

Goal state :

1	2	3
4	5	6
7	8	0

Actions : move blank UP(↑), DOWN(↓), LEFT(←), RIGHT(→)
Path cost : 1 per move (shortest path are optimal)

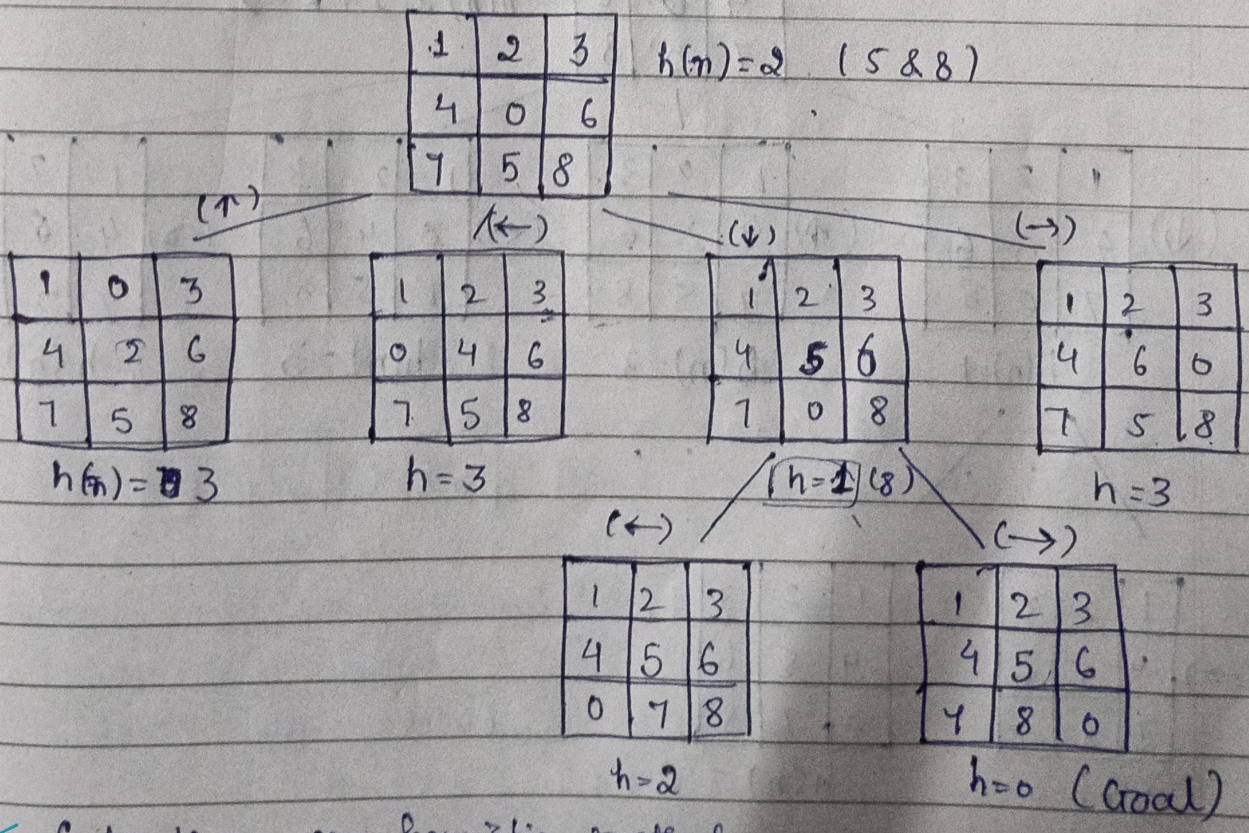
Without heuristic → use Breadth first search (BFS) (uniformed search)
BFS gaurantees shortest solution (for uniform cost per step)
BFS may consume a lot of memory



✓ Goal state

If the initial state is far from goal (large depth), BFS may be infeasible due to memory and time.
Thus we use heuristic. (A^*)

with heuristic (Misplaced Tiles)
Count the no. of tiles which are not in correct place, and will expand the branch with lowest no. of misplaced tiles



faster than non heuristic method

- Misplaced tiles heuristic is admissible (never overestimates cost)
- Ensures optimal solution with A*
- Simpler but weaker than Manhattan distance. → expands more nodes in harder puzzles.
- Still far more efficient than uninformed BFS/DFS

with heuristic (manhattan distance)

h(n) = manhattan dist

↳ sum of the moves required to place a misplaced tile in right position.

Ex

2	1	3
5	6	4
7	8	0

h(n)=6)

2 → 1 move | 5 → 1 move | 1 → 0 move
1 → 1 move | 6 → 1 move | 8 → 6 "
3 → 0 move | 4 → 2 move

h(n) = 1+1+0+1+1+2+0+0+~~0~~ = 6

1	2	3
4	0	6
7	5	8

h(n) = 1+1 = 2

(↓)

1	2	3
4	5	6
7	0	8

h(n)=1

(↑)

1	0	3
4	2	6
7	5	8

h(n)=3

(←)

1	2	3
0	4	6
7	5	8

h(n)=3

(→)

1	2	3
4	6	0
7	5	8

h(n)=3

(→)

1	2	3
4	5	6
7	8	0

h(n)=0

Goal

1	2	3
4	5	6
0	7	8

h(n) = 2

Manhattan dist: counts total number of horizontal + vertical steps each tiles is away from its goal.

PAGE NO.
DATE

- Manhattan is generally stronger, so it typically expands fewer nodes, making the search faster on harder problems
- Complexity: exponential (worst-case)
- Always admissible (never overestimates) → ensures optimal solution

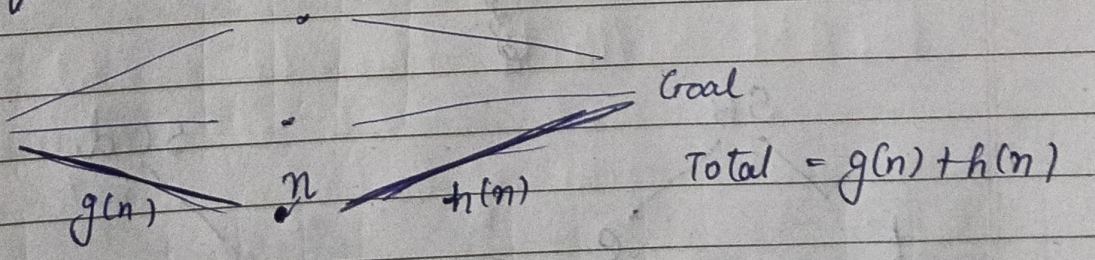
$$h(n) = \sum_{i=1}^{8} |r_{current}(i) - r_{goal}(i)| + |c_{current}(i) - c_{goal}(i)|$$

A* Search (informed searching) (heuristic search)

A* is a best-first search algorithm that finds the shortest path to a goal.

It combines:
- → Uniform Cost Search: explores cheapest path so far → g(n)
- → Greedy Best-first Search: explores the node closest to the goal → h(n)

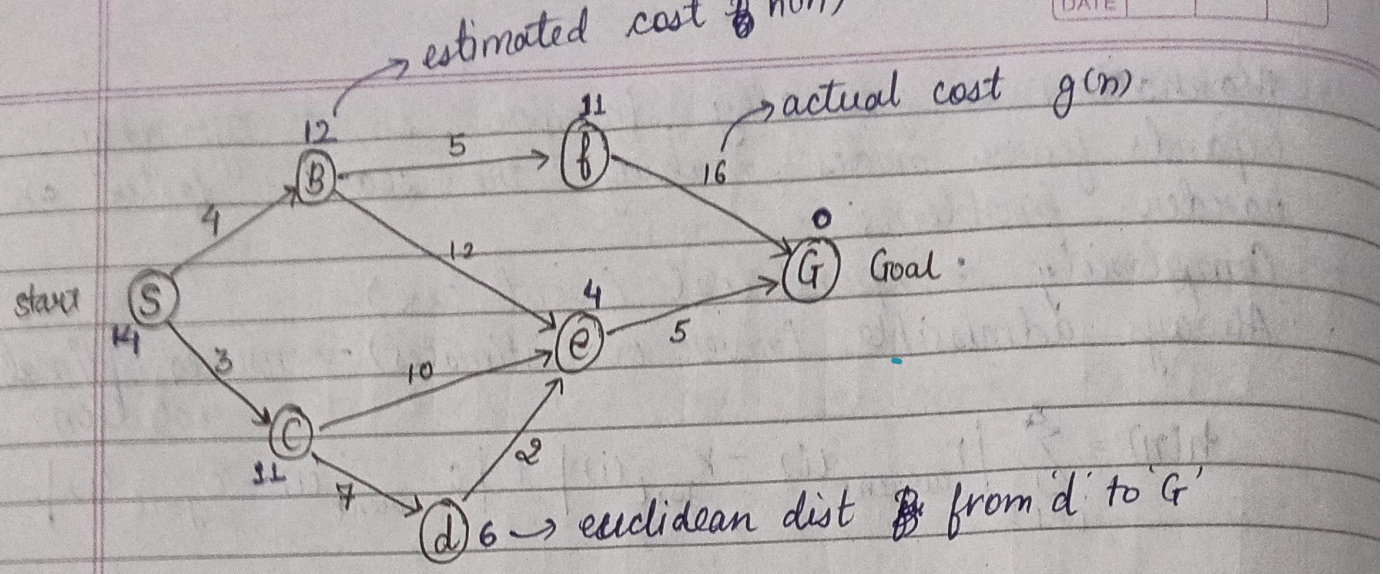
$$f(n) = g(n) + h(n)$$



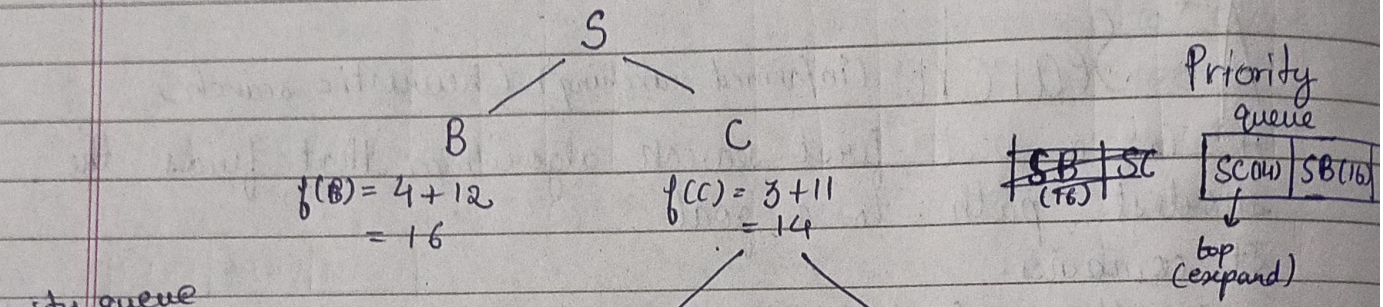
g(n): cost from start to node n
h(n): estimated cost from n to goal (heuristic)
f(n): estimated total cost of solution through n.

- Optimal (if h(n) is admissible, never overestimates)
- Complete (if branching factor is infinite)
- Efficient when heuristic is strong.

admissible → never overestimates.



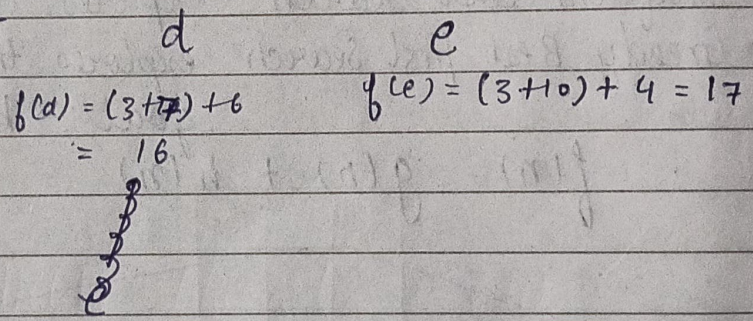
d → euclidean dist from 'd' to 'G'

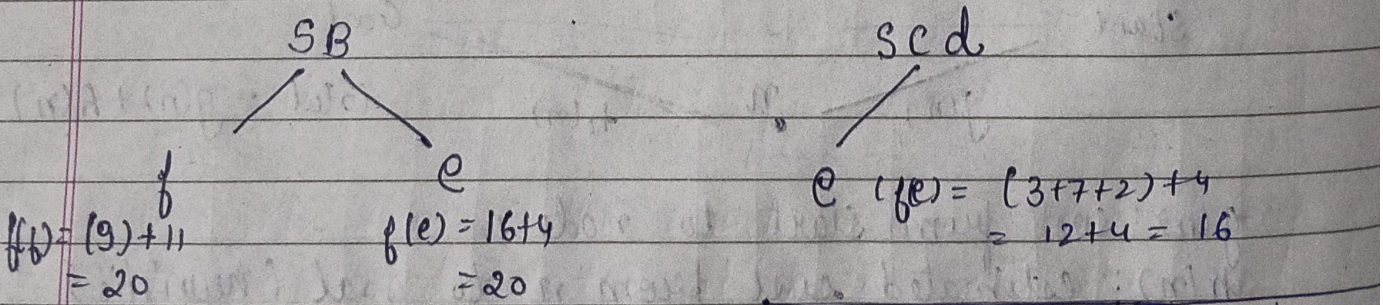


Priority queue

SB(16)	Scd(16)	Sce(17)	..

expand any





PQ = | Scde(16) | Sce(17) | SBf(20) | SBe(20)

Scde(16) ↓ expand

SBe(20) ↓ skip this one, because S→e is already discovered with lower cost

Scde

|

G

f(n) = (3+7+2+5)+0 = 17

path: S → C → d → e → G (cost = 17)

PAGE NO.
DATE

A* Algorithm

1. Put the start node in OPEN list (priority queue sorted by f)
2. Remove the node with smallest $f(n)$ from OPEN
3. If it's goal → stop, return path
4. Otherwise, expand it:
   - → Generate successors (valid moves)
   - → Compute $f = g + h$ for each
   - → Add them to OPEN (skip if already discovered with lower cost)
5. Repeat until goal found or OPEN is empty

Advantages

1. Guarantees optimal solution (with admissible h)
2. Explores fewer nodes than BFS / UCS if heuristic is strong

Limitations

1. Can use a lot of memory (keep all nodes in OPEN)
2. Exponential in worst-case
3. Heuristic quality is crucial (poor h → degenerates to UCS).

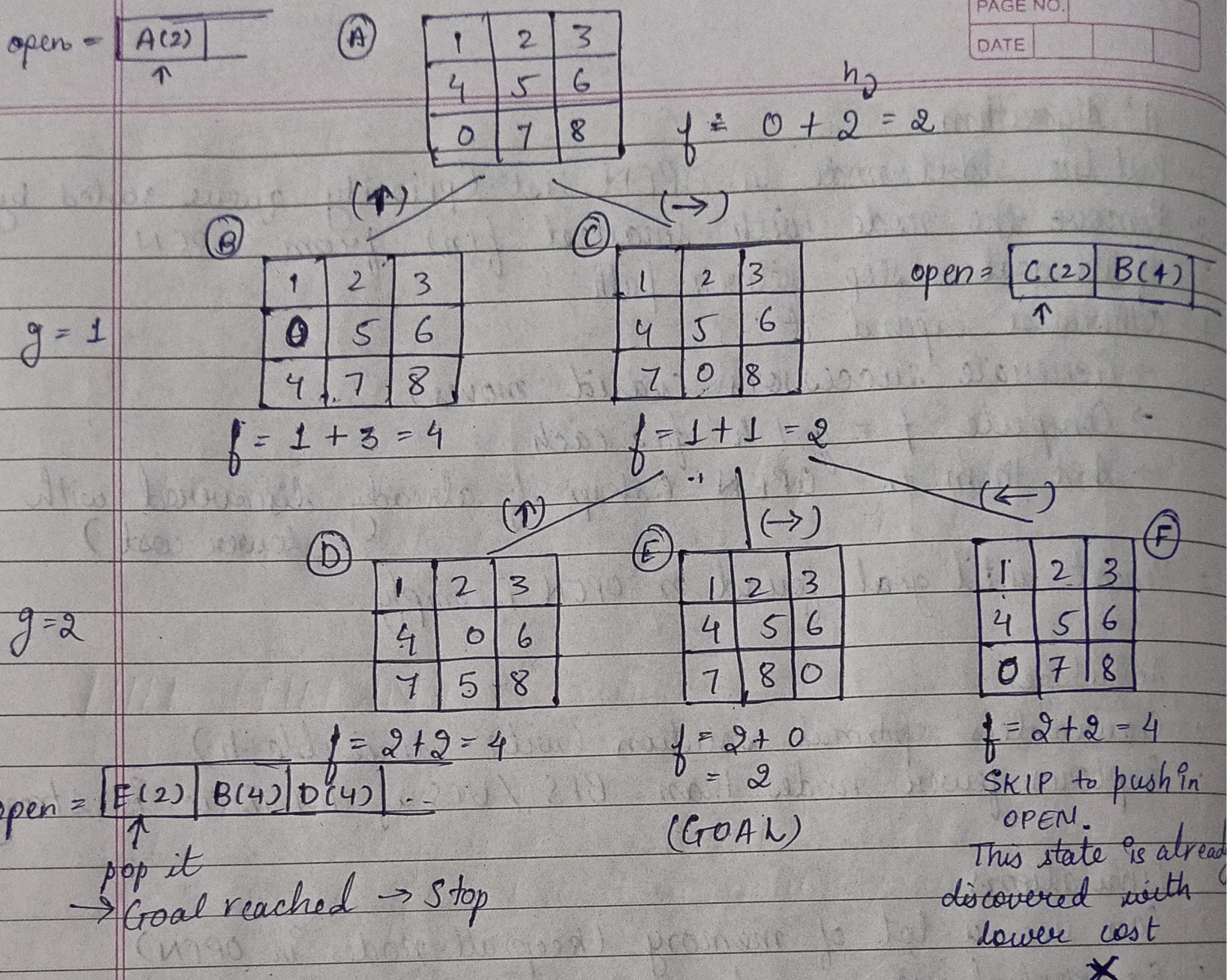
8 puzzle with A*

$f(n) = g(n) + h(n)$
↳ manhattan dist

Initial state = (1, 2, 3, 4, 5, 6, 0, 7, 8) →

1	2	3
4	5	6
0	7	8

Goal state = (1, 2, 3, 4, 5, 6, 7, 8, 0)
Actions = UP, DOWN, RIGHT, LEFT
State space = 9!
Path cost = 1 per move
heuristic = manhattan dist = $\sum_{i=1}^{8} |r_{curr}(i) - r_{goal}(i)| + |c_{curr}(i) - c_{goal}(i)|$



BEAM SEARCH (heuristic search)

It is a heuristic search algo. that is a memory efficient version of Best-first Search

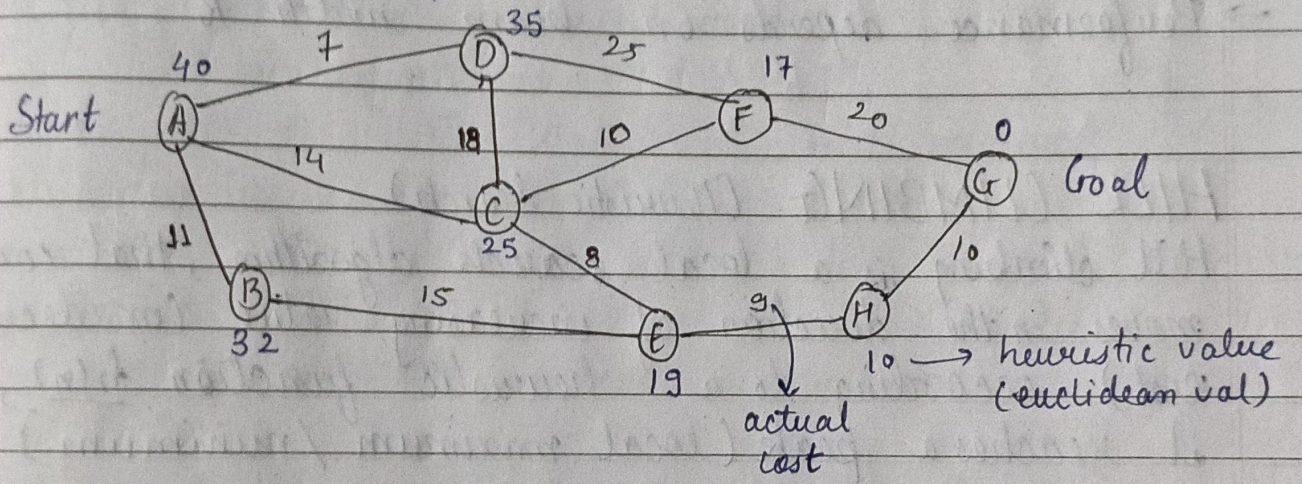
- At each level of the search tree, instead of keeping all generated nodes, it keeps only the best K-nodes (based on heuristic value)

→ Here, K = beam width (a fixed integer)

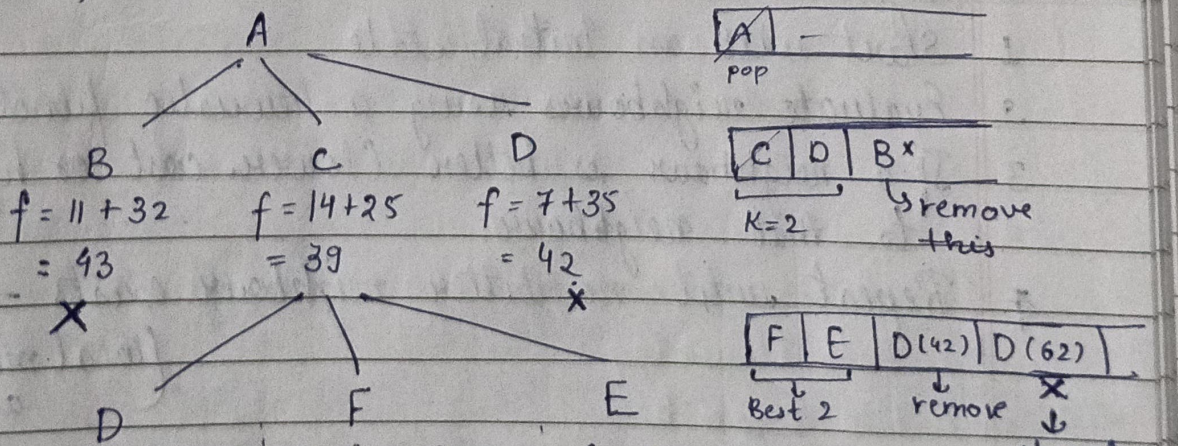
This reduces memory usage and search time but risks losing the optimal solution if the correct path is discarded

Algorithm

1. Start from initial node
2. Generate successors (children)
3. Evaluate them using a heuristic function $h(n)$
4. Keep only the k best nodes (lowest h values) and discard others.
5. Expand those nodes in the next level
6. Repeat until the goal is found (or no nodes remain)



Beam width, $k = 2$



Goal state reached, but it didn't stop here, because, their are paths remaining with lower cost

G

$f = (14+10+20)+0 = 44$

H

$f = (14+8+9)+10 = 41$

E | G

H | G

G → Goal with lower cost

$f = (14+8+9+10)+0 = 41$ ✓ G

- Not complete (it may discard the path that leads to goal)
- Not optimal (may miss the best path)
- Space complexity : $O(k \cdot d) \approx$ constant (d = depth of tree)
- Time complexity : $O(k \cdot d \cdot d)$ (b = branching factor)

→ Uses less memory than Best first or A*
→ Faster because few nodes are expanded
→ Works well when exact optimality is not critical
→ Performance depends on beam width k.

HILL CLIMBING (Heuristic Search)

Hill climbing is a local search algorithm, that continuou moves in the direction of increasing value (or decreasin cost) according to a heuristic function $h(n)$, until it reaches a peak (local maximum / minimum)

Algorithm

1. Start with an initial state
2. Evaluate neighbours using a heuristic function
3. If a neighbour is better (lower cost or higher value), move to that neighbour
4. Repeat until no better neighbour exists → stop (local maximum / minimum achieved)

Types of hill climbing

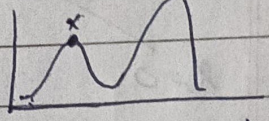
1. Simple hill climbing
   → selects the first better neighbour and moves there.
   → fast but may miss the best part.
2. Steepest-Ascent hill climbing
   → evaluates all neighbours, moves to the one with the best improvement.
   → slower but ~~simple~~ better than simple.

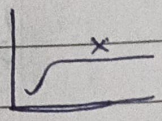
Hill climbing = Beam search ($k=1$)

PAGE NO.
DATE

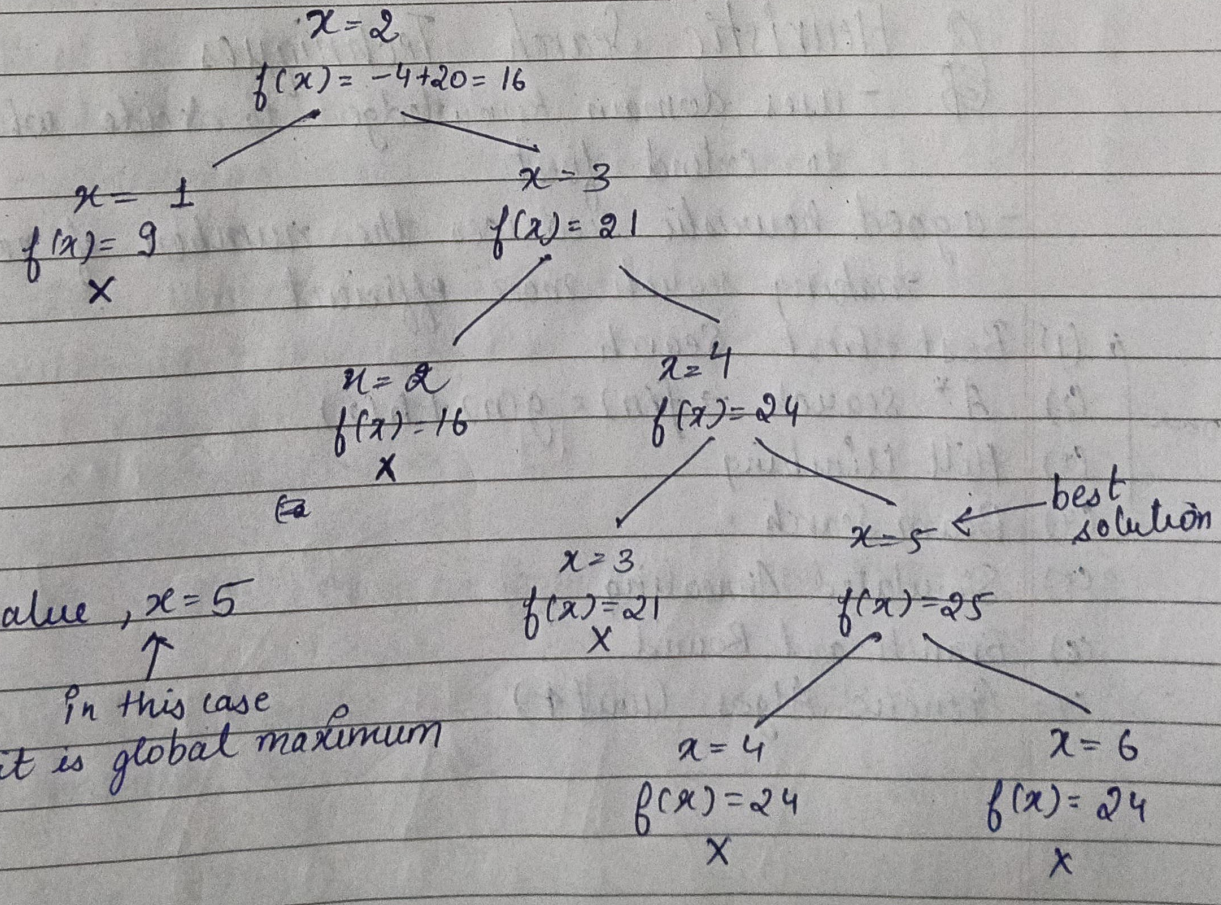
3. Stochastic hill climbing
→ chooses randomly among better neighbors (not always the best one)
→ helps escape loops

Problems with ~~the~~ hill climbing
- Local maxima : algo stops at a peak that is lower than the global maximum
- Plateaus : stop where all neighbours have equal value
- Ridges : path to better states exists but requires moving away temporarily.





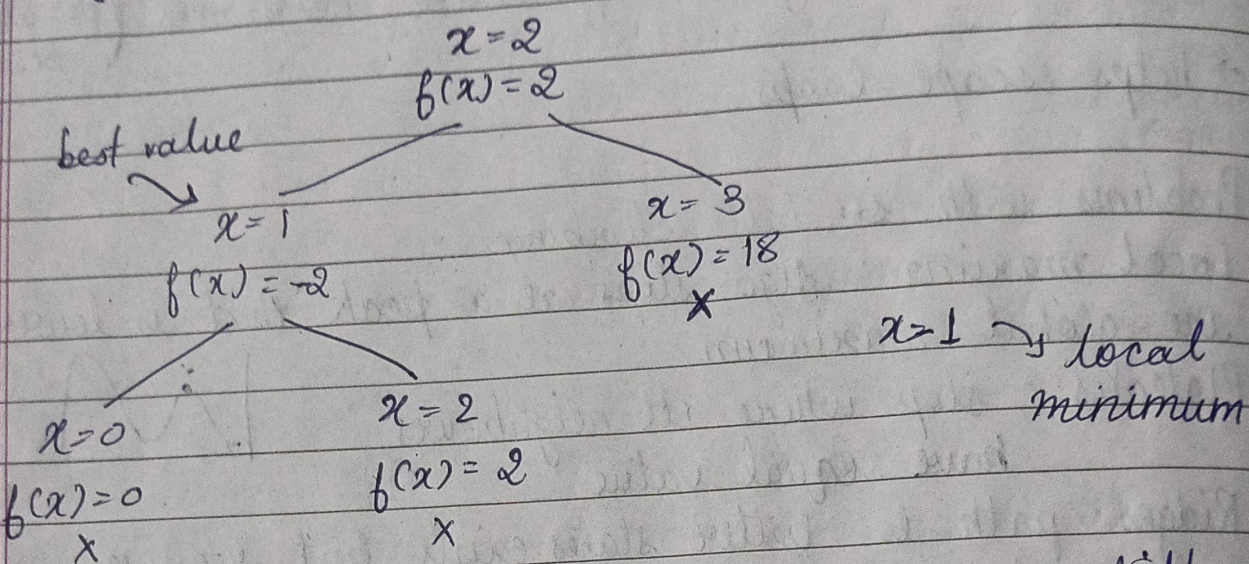
Ex Simple hill climbing
$f(x) = -x^2 + 10x$ → maximize value.



max value, $x=5$
↑
In this case it is global maximum

PAGE NO.
DATE

$f(x) = x^3 - 3x$ → ~~maximize~~ minimize



x=1 → local minimum

If the global optimum were further away, hill climbing would never reach it
It simply halts at first peak or valley → (equal vals)

Heuristic Search Techniques

- uses domain knowledge to decide which path to extend first
- a good heuristic reduces the number of nodes explored making search more efficient.

(1) Best-First Search
(2) A* search → $f(n) = g(n) + h(n)$
(3) Hill Climbing
(4) Beam Search
(5) Simulated Annealing
(6) Branch and Bound
(7) Genetic Algos (Unit 4)

DONE

→ have adversarial search problems.

PAGE NO.
DATE

GAME PLAYING → Chess, Tic-Tac-Toe

- Game playing in AI is the process of building systems that can play strategic games against a human or another AI.
- These are multi-agent environments
  - Player 1 (Maximizer)
  - Player 2 (Minimizer)

Types of Games

1. Single-player games
   - Only one agent, no oponent

Ex 8-puzzle, Sudoku

Algos → search Algos (BFS, DFS, A* etc)

2. Two-player games
   - Involved competition

Ex Chess, Checkers, Tic-Tac-Toe *

Algos → Minimax algo, Alpha-Beta pruning

Game tree / search tree / space search graph

- Representation of all possible moves in a game.
- Nodes → states of game
- Edges → actions (moves)
- ply → depth of graph
- Terminal nodes → win / loss / draw state

→ We use a heuristic evaluation function to estimates the "goodness" of a position

Ex. Deep Blue (Chess)
Alpha Go (Go)

Minimax evaluates all possible moves and outcomes
- Max (X) chooses best utility move
- Min (O) blocks / reduce score

Backtracking → This ensures optimal play

PAGE NO.
DATE

Minimax Algorithm (adversarial search)

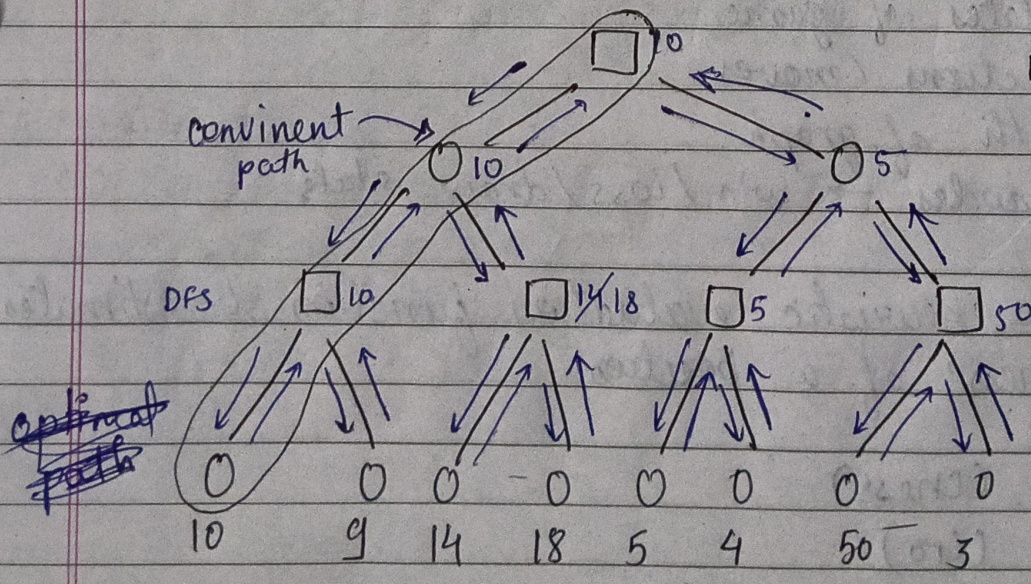
- Decision-making algo used in 2-player games
- It assumes
  - → Player 1 (Maximizer) tries to maximize the score
  - → Player 2 (Minimizer) tries to minimize the score
- Both players are rational and play optimally

Working

- Build a game tree of all possible moves
- Assign utility values (+1 → win, -1 loss, 0 draw)
- Backtrack values up the tree
  - At MAX nodes → choose the maximum value
  - At MIN nodes → choose the minimum value

Algorithm

1. Generate the game tree to a certain depth (or terminal states)
2. Apply an evaluation function at terminal nodes.
3. Backtrack values
   - MAX node = choose maximum child
   - MIN node = choose minimum child
4. Root node's value = best move for the current player



MAX aims to maximize its score, while MIN aims to minimize it.

PAGE NO
DATE

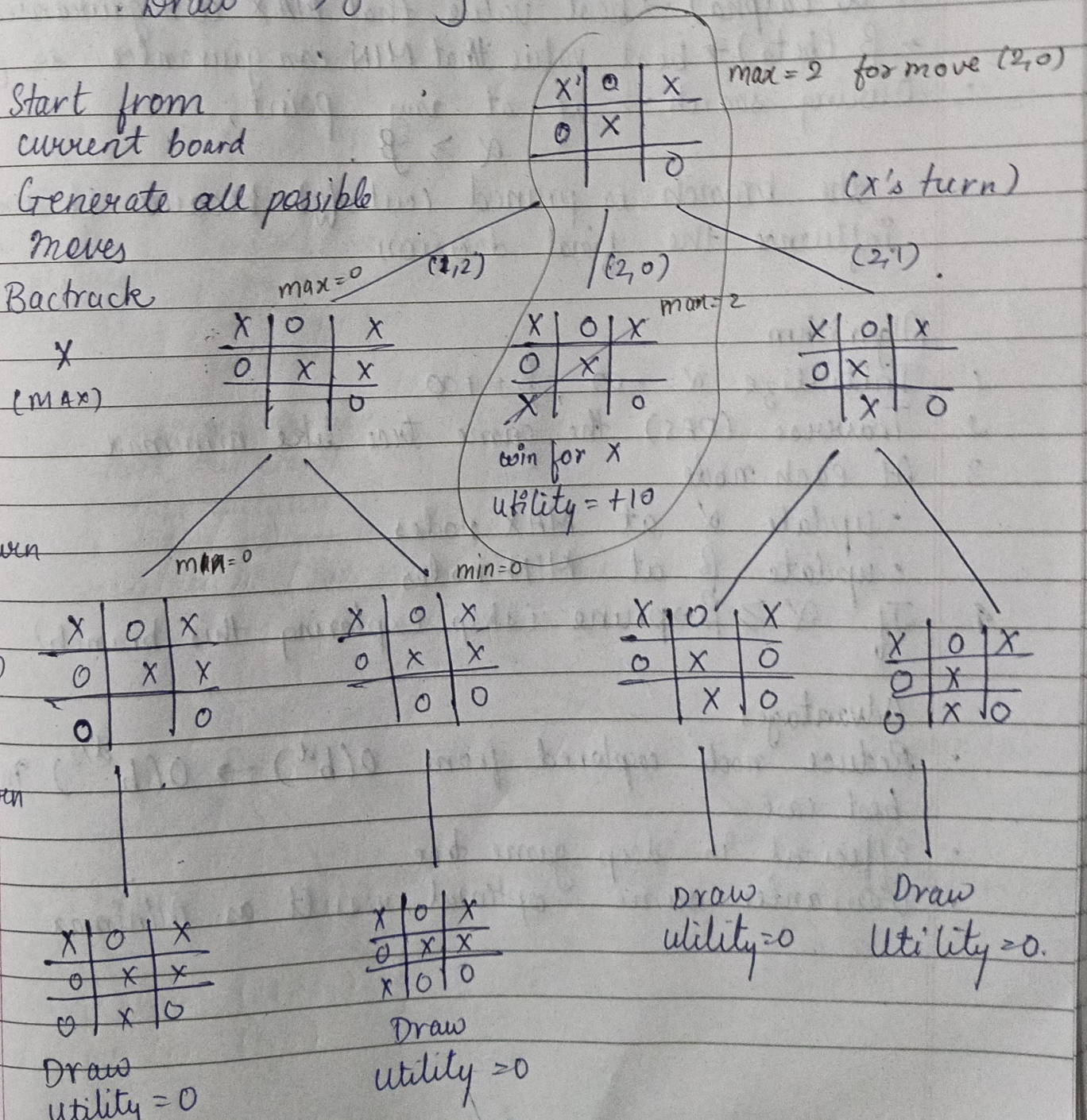
Limitations
Exponential complexity $O(b^d)$ — b → branching factor, d → depth
Not feasible for large games like Chess → use Alpha-Beta

MINIMAX FOR TIC-TAC-TOE

- Player X (AI) → MAX
- Player O → MIN
  - X wins → +10
  - O wins → -10
  - Draw → 0

} Utility values

1. Start from current board
2. Generate all possible moves
3. Backtrack



PAGE NO.
DATE

optimized version of minimax

Alpha-Beta Pruning

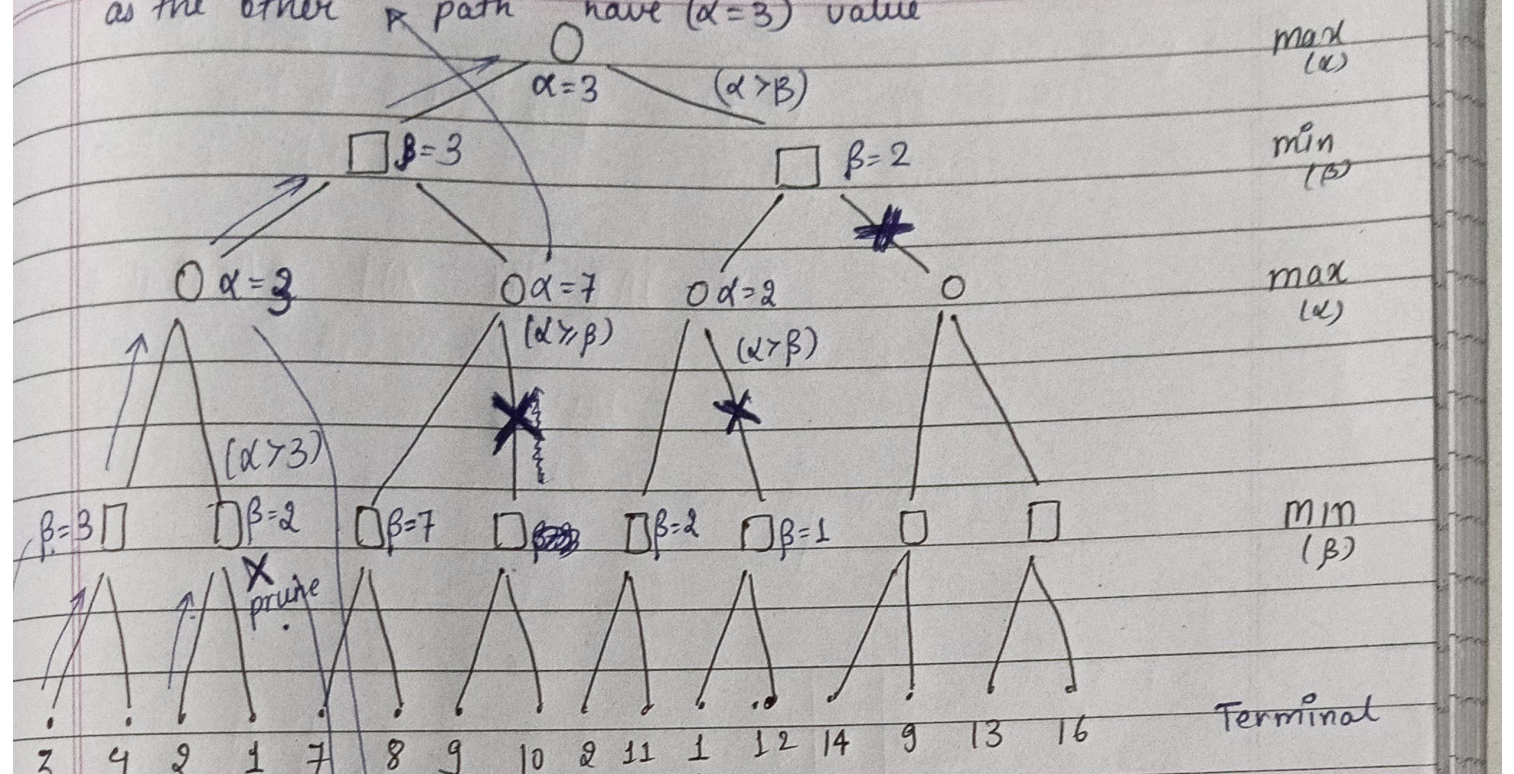
- Optimization of minimax algo
- It eliminates branches in the game tree that cannot affect the final decision
- This makes the search faster without changing the result.

Keep track of two values while searching
→ α (alpha) → best value that MAX can gaurantee so far.
→ β (beta) → best value that MIN can gaurantee so far.
- During search → if at any point;
  $\alpha \geq \beta$.
  the branch is pruned (cut off) because it won't influence the final decision

Algorithm
1. Initialize $\alpha = -\infty$, $\beta = +\infty$
2. Traverse (DFS) the game tree like minimax
3. At each node
   - update α at MAX nodes
   - update β at MIN nodes
4. If $\alpha \geq \beta$ → prune (skip exploring this branch)

Advantages
- Reduce node explored from $O(b^d) \rightarrow O(b^{d/2})$ in best case.
- Efficient in deep game tree.
- Gaurantee same optimal result as Minimax

no need to explore this branch further as the MIN node will not choose this path as the other path have ($\alpha = 3$) value



MIN value of this node will not exceed 3 (at max 3)

the MAX value of this node will not be less than 3 (at least 3)

no need to explore this branch further because it can give at max value of 2 (less than current α).

Ex