Definition
Problem-solving in artificial intelligence (AI) involves identifying a sequence of actions that lead from an initial state to a desired goal state. This process is often conceptualized as a search through a state space, where each state represents a possible configuration of the system.

Problem as a State Space Search
In this framework, a problem is defined by:

Initial State: The starting point of the problem.

Goal State: The desired outcome.

State Space: The set of all possible states reachable from the initial state.

Operators: Actions that transition the system from one state to another.

Path Cost: A numerical value representing the cost associated with a path from the initial to the current state.

The objective is to find a path from the initial state to the goal state with minimal path cost.

Problem Formulation
Formulating a problem involves:

Defining the State Space: Identifying all possible states.

Specifying the Initial State: Determining where the problem-solving process begins.

Defining the Goal Test: A function to determine if a given state is the goal state.

Identifying Operators: Actions available to transition between states.

Assigning Path Costs: Evaluating the cost associated with each action or sequence of actions.

A well-formulated problem provides a clear roadmap for the search process.

Well-Defined Problems
A problem is well-defined if it has:

Clear Initial State: Unambiguously specified starting point.

Explicit Goal State: Clearly defined desired outcome.

Defined State Space: Comprehensive set of all possible states.

Known Operators: Clearly specified actions for state transitions.

Deterministic Outcomes: Actions have predictable effects.

Examples include puzzles like the 8-queens problem or the traveling salesman problem.

Constraint Satisfaction Problem (CSP)
A CSP is a problem where the solution must satisfy a set of constraints or conditions. Each state represents a partial solution, and operators add components to the solution while ensuring constraints are not violated.
Components of CSP:

Variables: Elements to be solved for.

Domains: Possible values each variable can take.

Constraints: Restrictions on the values that variables can simultaneously take.

Example: In the 8-queens problem, variables represent queen positions, domains are the board coordinates, and constraints ensure no two queens threaten each other.

Uninformed Search Techniques
These strategies do not utilize additional information about states beyond what is provided in the problem definition.

Depth-First Search (DFS): Explores as far down a branch as possible before backtracking.

Characteristics:
Uses a stack data structure (LIFO).
Can get trapped in deep or infinite branches.
Memory efficient but not guaranteed to find the shortest path.

Breadth-First Search (BFS): Explores all nodes at the present depth before moving to nodes at the next depth level.

Characteristics:
Uses a queue data structure (FIFO).
Guaranteed to find the shortest path in an unweighted graph.
Can consume significant memory for wide or infinite state spaces.

Depth-Limited Search (DLS): DFS with a predetermined depth limit to prevent infinite descent.

Characteristics:
Limits how deep the search can go.
Addresses infinite path issues but may miss solutions deeper than the limit.

Iterative Deepening Search (IDS): Combines the benefits of DFS and BFS by incrementally increasing the depth limit.

Characteristics:
Performs DLS repeatedly with increasing depth limits.
Finds the shortest path with less memory usage than BFS.

Bidirectional Search: Simultaneously searches forward from the initial state and backward from the goal state, meeting in the middle.

Characteristics:
Can significantly reduce search time.
Requires both forward and backward search capabilities.

Informed Search Techniques
These strategies utilize problem-specific knowledge (heuristics) to find solutions more efficiently.

Greedy Best-First Search: Selects the node that appears to be closest to the goal based on a heuristic.

Characteristics:
Uses a priority queue ordered by the heuristic function.
Can be fast but is not guaranteed to find the optimal solution.

A Search:* Combines the cost to reach a node and the estimated cost from that node to the goal.

Hill climbing search
Hill climbing search is sometimes compared to climbing Mount Everest in thick fog with amnesia i.e. destined to reach top of Everest but due to thick fog (poor visibility) and forgetting what top of Everest is like, peak lower than top of top of Everest might be considered as top of Everest.
• Hill Climbing Search initiates a loop that continuously moves in the direction of increasing value
• Terminates when it reaches a “Peak”
• Doesn’t maintain a search tree so the current node data structure needs only record the state and its objective function value.
• Hill climbing does not look ahead beyond the immediate neighbors of the current state.
• Hill climbing is also called greedy local search sometimes because it grabs a good neighbor state without thinking ahead about where to go next
• One move is selected and all other nodes are rejected and are never considered.
• Halts if there is no successor.
• Problem: depending on initial state, can get stuck in local maxima

Drawback of Hill Climbing Search
This simple policy has three well-known drawbacks:
a. Local Maxima: a local maximum as opposed to global maximum.
b. Plateaus: An area of the search space where evaluation function is flat, thus requiring random walk. Search might be unable to find its way of plateau.
c. Ridge: Where there are steep slopes and the search direction is not towards the top but towards the side.

Simulated Annealing Search
• Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency
• Instead of restarting from a random point, we allow the search to take some downhill steps to try to escape local maxima.
• A random pick is made for the move
• If it improves the situation, it is accepted straight away
• If it worsens the situation, it is accepted with some probability less than 1 which decreases exponentially with the badness of the move i.e. for bad moves the probability is low and for comparatively less bad moves, it is higher.

• Probability of downward steps is controlled by temperature parameter.
High temperature implies high chance of trying locally "bad" moves, allowing nondeterministic exploration.
• Low temperature makes search more deterministic (like hill-climbing).
• Temperature begins high and gradually decreases according to a predetermined annealing schedule.
• Initially we are willing to try out lots of possible paths, but over time we gradually settle in on the most promising path.
• If temperature is lowered slowly enough, an optimal solution will be found.
• In practice, this schedule is often too slow and we have to accept suboptimal solutions.
Applications of Simulated Annealing Search
VLSI layout problem
Factory scheduling
Travelling salesman problem
Local beam search
• A path based algorithm
• Keeps track of k states rather than just one
• Starts with k randomly generated states
• At each iteration, all the successors of all k states are generated.
• If anyone is a goal state, stop, else select the k best successors from the complete list and repeat.

Adversarial Search
Competitive environments in which the agents goals are in conflict, give rise to adversarial search, often known as games.
In AI, games means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which utility values at the end of the game are always equal and opposite.

• Eg. If first player wins, the other player necessarily loses
Opposition between the agent's utility functions make the situation adversarial.
Game
A game can be formally be defined as a kind of search problem with the following components

• Initial state
• A successor function
• A terminal test
• A utility function
Minimax Algorithm
• It is a recursive algorithm for choosing the next move in a n-player game, usually a two player game
• A value is associated with each position or state of the game
• The value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach the position.
• The player then makes the move that maximizes the minimum value of the position from the opponents possible moves called maximizing player and other player minimize the maximum value of the position called minimizing player.
MiniMax Game Search
• It is a Depth-first search with limited depth.
• Use a static evaluation function for all leaf states.
• Assume the opponent will make the best move possible.
Algorithm
minimax(player,board)
if(game over in current board position)
return winner
children = all legal moves for player from this board

if(max's turn)
return maximal score of calling minimax on all the children
else (min's turn)
return minimal score of calling minimax on all the children

Alpha-beta pruning
Minimax search explores some parts of tree it doesn't have to
• To avoid this, minimax algorithm is modified with two values alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured respectively.
• Alpha-beta pruning algorithm explores a branch only if there is possibility of producing superior alternatives.
• It reduces the time required for the search and it must be restricted so that no time is to be wasted searching moves that are obviously bad for the current player.
• The exact implementation of alpha-beta keeps track of the best move for each side as it moves throughout the tree.
• It adapts minimax to consider the values of the nodes and whether exploration down a branch could possibly be fruitful.
• It proceeds from the idea "If you have an idea that is surely bad, don't take the time to see how truly awful it is (Pat Winston)"

Properties of α-β
• Pruning does not affect final result
• Good move ordering improves effectiveness of pruning
• With "perfect ordering," time complexity = O(bm/2)
  doubles depth of search
• A simple example of the value of reasoning about which computations are relevant

Alpha Beta Procedure
• At each non-leaf node, store two values— alpha and beta.
• Let alpha be the best (i.e., maximum) value found so far at a "max" node.
• Let beta be the best (i.e., minimum) value found so far at a "min" node.
• Initially assign alpha = – and beta = at the root.
• Note alpha is monotonically non-decreasing and beta is monotonically non-increasing as you travel up the tree.
• Given a node n, cut off the search below that node (i.e., generate no more children) if
o n is a "max" node and alpha(n) beta(i) for some "min" ancestor i of n, or
o n is a "min" node and beta(n) alpha(j) for some "max" ancestor j of n.

Algorithm
function alphabeta(node, depth, α, β, maximizingPlayer)
if depth = 0 or node is a terminal node
return the heuristic value of node
if maximizingPlayer
for each child of node
α := max(α, alphabeta(child, depth - 1, α, β, FALSE))
if β ≤ α
break (* β cut-off *)
return α
else
for each child of node
β := min(β, alphabeta(child, depth - 1, α, β, TRUE))
if β ≤ α
break (* α cut-off *)
return β