

UNIT 2

Problem Solving Methods

(12 Hours) — Artificial Intelligence (CACS410)

⚠ Please Note

These notes were prepared for quick revision and **may contain errors or omissions**. Please take care: cross-check important definitions, formulas, algorithms and worked examples against your prescribed textbook, class notes and teacher's instructions before relying on them (especially for exams).

2.1 Problems, State Spaces and Problem Formulation

Definition of a Problem

A **problem** is a situation in which an agent has an **initial state** and wants to reach a **goal state** by applying a sequence of permitted actions. Solving the problem means finding such a sequence.

Problem Formulation and Well-Defined Problems

Problem formulation is deciding which states and actions to consider. A problem is **well-defined** when it is specified by these five components:

Component	Meaning	8-puzzle example
Initial state	Where the agent starts	A given arrangement of tiles
Actions / operators	What the agent can do in a state (successor function)	Move the blank Left, Right, Up, Down
Transition model	The state that results from an action	Swap the blank with the adjacent tile
Goal test	Is this state a goal?	Tiles are in the target order
Path cost	Numeric cost of a path	1 per move

A **solution** is a path from the initial state to a goal state; an **optimal solution** has the lowest path cost. Problems with vague goals or unspecified operators (e.g., “make a delicious meal”) are **ill-defined**. Before searching, the agent goes through four phases: *goal formulation* → *problem formulation* → *search* → *execution*.

Problem as State Space Representation

The **state space** is the set of all states reachable from the initial state by any sequence of actions. It is drawn as a graph: **nodes** are states, **edges** are actions. For the 8-puzzle there are $9!/2 = 181,440$ reachable states.

Constraint Satisfaction Problems (CSP)

A CSP is defined by a set of **variables** $X_1 \dots X_n$, a **domain** of allowed values for each variable and a set of **constraints** restricting the values that combinations of variables may take. A **solution** is a complete assignment that satisfies all constraints. CSPs are usually solved by **backtracking search** (assign a variable, check constraints, undo on failure) improved by heuristics:

- **MRV** (minimum remaining values): choose the variable with the fewest legal values left.
- **Degree heuristic**: choose the variable involved in the most constraints on unassigned variables.
- **Least constraining value**: try first the value that rules out the fewest choices for neighbours.
- **Forward checking / arc consistency**: remove values from neighbours' domains that can no longer be legal, and detect failure early.

Water Jug Problem

Example — Water Jug Problem

You have a 4-litre jug and a 3-litre jug, no markings, and an unlimited water supply. Measure exactly **2 litres** in the 4-litre jug.

State = (x, y) where x = water in the 4 L jug (0...4) and y = water in the 3 L jug (0...3). Start (0, 0); goal (2, n).

Rule	Condition	Action
1	$x < 4$	Fill the 4 L jug: $(x, y) \rightarrow (4, y)$
2	$y < 3$	Fill the 3 L jug: $(x, y) \rightarrow (x, 3)$
3	$x > 0$	Empty the 4 L jug: $(x, y) \rightarrow (0, y)$
4	$y > 0$	Empty the 3 L jug: $(x, y) \rightarrow (x, 0)$
5	$x + y \geq 4, y > 0$	Pour 3 L into 4 L until full: $(x, y) \rightarrow (4, y - (4 - x))$
6	$x + y \geq 3, x > 0$	Pour 4 L into 3 L until full: $(x, y) \rightarrow (x - (3 - y), 3)$

Rule	Condition	Action
7	$x + y \leq 4, y > 0$	Pour all of 3 L into 4 L: $(x, y) \rightarrow (x + y, 0)$
8	$x + y \leq 3, x > 0$	Pour all of 4 L into 3 L: $(x, y) \rightarrow (0, x + y)$

Solution (two possible paths)

Path 1: $(0,0) \rightarrow (0,3)$ [fill 3 L] $\rightarrow (3,0)$ [pour into 4 L] $\rightarrow (3,3)$ [fill 3 L] $\rightarrow (4,2)$ [pour until 4 L full] $\rightarrow (0,2)$ [empty 4 L] $\rightarrow (2,0)$ [pour 2 L into 4 L].

Path 2: $(0,0) \rightarrow (4,0) \rightarrow (1,3) \rightarrow (1,0) \rightarrow (0,1) \rightarrow (4,1) \rightarrow (2,3)$. Either way, exactly 2 L ends up in the 4 L jug (or the 3 L jug).

N-Queens Problem

Place **N queens** on an $N \times N$ chessboard so that no two queens attack each other, i.e., no two share a **row, column or diagonal**. **Incremental formulation:** state = any arrangement of $0 \dots N$ queens; place one queen per column from left to right in a square that is not attacked; goal = N queens, none attacked. The 4-queens problem has exactly 2 solutions and 8-queens has 92 (12 distinct up to symmetry).

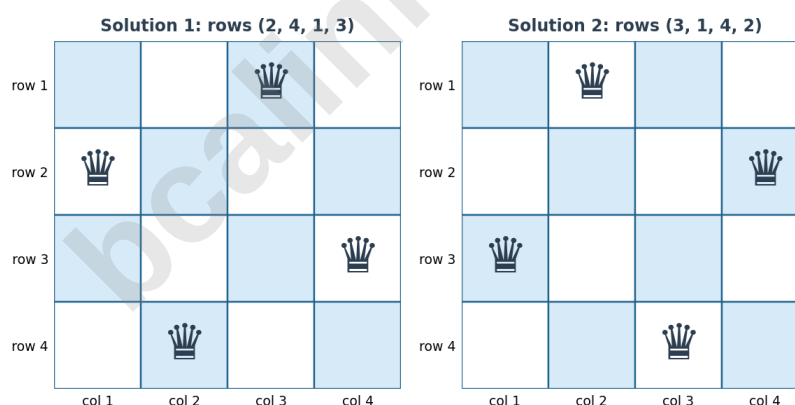


Figure 2.1 — The two solutions of the 4-queens problem (row number of the queen in each column)

Cryptarithmic Problem

Each letter stands for a different digit (0–9); the arithmetic must hold and leading letters cannot be 0. It is a CSP where letters are variables, digits are values and the sum is a constraint.

Example — SEND + MORE = MONEY

```

S E N D
+ M O R E
-----

```

Solution (reasoning)

1. Two 4-digit numbers sum to at most 19998, so the carry into the 5th digit is 1: **M = 1**.
2. In the thousands column $S + M (+ \text{carry})$ must give $O + 10$. With $M = 1$ this forces **S = 9** (or 8 with a carry) and **O = 0**.
3. Hundreds column: $E + O + \text{carry} = N \Rightarrow N = E + 1$ (a carry exists).
4. Tens column: $N + R + \text{carry} = E + 10 \Rightarrow$ with $N = E + 1$ we get **R = 8** (9 is used).
5. Units column: $D + E = Y + 10$. Trying the remaining digits {2,3,4,5,6,7}: $E = 5$, $N = 6$, $D = 7$ gives **Y = 2**.

Answer: $S=9, E=5, N=6, D=7, M=1, O=0, R=8, Y=2$, i.e., $9567 + 1085 = \mathbf{10652}$.

Graph (Map) Coloring Problem

Color the nodes of a graph so that adjacent nodes get different colors using the fewest colors. The smallest number needed is the **chromatic number**. As a CSP: variables = regions, domain = {Red, Green, Blue}, constraint = neighbours differ.

Example — Coloring the map of Australia (7 regions)

Neighbours: WA-NT, WA-SA, NT-SA, NT-Q, SA-Q, SA-NSW, SA-V, Q-NSW, NSW-V.
Tasmania (T) has no neighbours.

Solution

WA = Red, NT = Green, SA = Blue (SA touches both WA and NT), Q = Red, NSW = Green, V = Red, T = any color.

Three colors are enough, and since WA, NT and SA are mutually adjacent, at least 3 are needed.

2.2 Problem Solving by Searching

The agent builds a **search tree** whose root is the initial state; expanding a node generates its children using the actions available. Nodes waiting to be expanded form the **frontier** (OPEN list); expanded states are kept in the **explored set** (CLOSED list) to avoid repeated states. Each node stores: state, parent, action, path cost $g(n)$ and depth.

General State Space Search

```

OPEN <- [start]; CLOSED <- []
while OPEN is not empty:
    n <- remove a node from OPEN          # which one = search strategy
    if GOAL(n): return path to n
    put n in CLOSED
    for each successor s of n not in OPEN or CLOSED:
        set parent(s) = n; add s to OPEN
return FAILURE

```

The only thing that differs between search algorithms is **which node is removed from OPEN**: a queue gives BFS, a stack gives DFS and a priority queue ordered by cost or heuristic gives uniform-cost, greedy or A* search.

Types of Search

Type	Description	Examples
Uninformed (blind)	No knowledge beyond the problem definition; only generates successors and tests for goal	BFS, DFS, DLS, IDDFS, Bidirectional
Informed (heuristic)	Uses problem-specific knowledge (a heuristic) to guide the search	Greedy best-first, A*
Local search	Keeps only the current state and improves it; path is irrelevant	Hill climbing, Simulated annealing
Adversarial	Opponents with conflicting goals	Minimax, Alpha-Beta

Measuring Problem-Solving Performance

Measure	Question answered
Completeness	Is the algorithm guaranteed to find a solution when one exists?
Optimality	Does it find the least-cost solution?
Time complexity	How many nodes are generated?
Space complexity	How many nodes must be stored in memory?

Complexity is expressed with **b** = branching factor (max successors of a node), **d** = depth of the shallowest goal, **m** = maximum depth of the tree and **l** = depth limit.

2.3 Uninformed (Blind) Search

Breadth-First Search (BFS)

Expands the **shallowest** unexpanded node first, using a **FIFO queue**; all nodes at depth k are expanded before any at depth $k+1$.

- **Complete:** yes (if b is finite). **Optimal:** yes, if all step costs are equal.
- **Time:** $O(b^d)$. **Space:** $O(b^d)$ — the main drawback; memory runs out before time does.

Depth-First Search (DFS)

Always expands the **deepest** node in the frontier, using a **LIFO stack** (or recursion). It goes down one branch to a leaf, then backtracks.

- **Complete:** no (fails in infinite-depth or looping spaces). **Optimal:** no.
- **Time:** $O(b^m)$. **Space:** $O(bm)$ — very small, its main advantage.

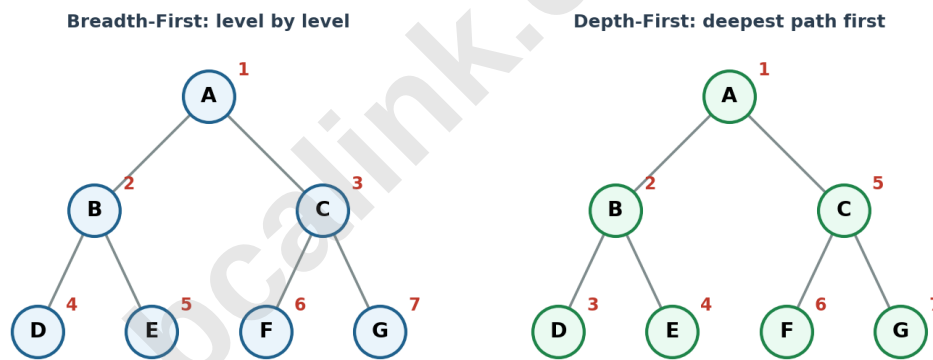


Figure 2.2 — Order in which nodes are visited by BFS and DFS (red numbers)

Example — Visit order

On the tree in Figure 2.2 with goal G: **BFS** visits A, B, C, D, E, F, G. **DFS** visits A, B, D, E, C, F, G.

Depth-Limited Search (DLS)

DFS with a predetermined **depth limit l** ; nodes at depth l are treated as if they have no successors. It solves the infinite-path problem of DFS. It can return *failure* (no solution within the limit) or *cutoff* (limit reached). **Complete:** no if $l < d$. **Optimal:** no. **Time:** $O(b^l)$. **Space:** $O(bl)$.

Iterative Deepening Depth-First Search (IDDFS)

Runs DLS repeatedly with limit 0, 1, 2, 3, ... until a goal is found. It combines the **small memory** of DFS with the **completeness and optimality** of BFS (for unit step costs). Repeating the upper levels looks wasteful, but most nodes are in the bottom level. For $b = 10$, $d = 5$: IDDFS generates about 123,456 nodes versus about 111,111 for BFS — only about 11% extra.

- **Complete:** yes. **Optimal:** yes (unit cost). **Time:** $O(b^d)$. **Space:** $O(bd)$.

Note: IDDFS is the preferred uninformed method when the search space is large and the depth of the solution is unknown.

Bidirectional Search

Runs **two searches at once** — one forward from the start, one backward from the goal — and stops when the two frontiers meet. Since $b^{(d/2)} + b^{(d/2)}$ is much smaller than b^d (for $b = 10$, $d = 6$: 2,000 vs 1,000,000 nodes), it is very efficient. It requires an explicitly known goal state and the ability to compute predecessors of a state.

Comparison of Uninformed Strategies

Algorithm	Complete?	Optimal?	Time	Space
BFS	Yes	Yes (unit cost)	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
Depth-limited	No (if $l < d$)	No	$O(b^l)$	$O(bl)$
Iterative deepening	Yes	Yes (unit cost)	$O(b^d)$	$O(bd)$
Bidirectional	Yes	Yes (unit cost)	$O(b^{(d/2)})$	$O(b^{(d/2)})$

*Note: **Uniform-cost search** (not in the syllabus) generalizes BFS to varying step costs: it expands the node with the lowest path cost $g(n)$ and is complete and optimal.*

Solving the N-Queens Problem with Uninformed Search

N-queens is naturally solved by **DFS with backtracking**: all solutions lie at depth N and memory stays tiny, while BFS would store enormous frontiers. Place a queen in the next column; if no safe row exists, backtrack to the previous column.

Example — 4-Queens by backtracking (position = row number of the queen in each column)

Column 1 = row 1 → column 2: rows 1, 2 attacked; row 3 safe → (1, 3). Column 3: rows 1, 2, 3, 4 are all attacked → **dead end, backtrack**.

Column 2 = row 4 $\rightarrow$ (1, 4). Column 3 = row 2 $\rightarrow$ (1, 4, 2). Column 4: all rows attacked $\rightarrow$ **backtrack**; column 3 has no other safe row $\rightarrow$ backtrack to column 1.

Column 1 = row 2 $\rightarrow$ column 2 = row 4 $\rightarrow$ column 3 = row 1 $\rightarrow$ column 4 = row 3. **Solution (2, 4, 1, 3)** found.

Puzzle problems (8-puzzle): BFS or IDDFS find a solution with the fewest moves, but the number of states is large, so these blind methods become impractical for bigger puzzles. This motivates heuristic search.

2.4 Informed (Heuristic) Search

Informed search uses a **heuristic function $h(n)$** = estimated cost of the cheapest path from node n to a goal, with $h(\text{goal}) = 0$. It lets the agent prefer promising nodes.

- **Admissible** heuristic: never overestimates the true cost, i.e., $h(n) \leq h^*(n)$ (optimistic).
- **Consistent (monotone)** heuristic: $h(n) \leq c(n, a, n') + h(n')$ for every successor n' (a triangle inequality). Every consistent heuristic is admissible.

Greedy Best-First Search

Evaluation function **$f(n) = h(n)$** : always expands the node that *appears* closest to the goal. It is fast but can be misled.

- **Complete:** no (can loop in graphs without checking). **Optimal:** no. **Time and space:** $O(b^m)$ worst case, much better with a good h .

A* Search

Evaluation function **$f(n) = g(n) + h(n)$** , where $g(n)$ = cost from the start to n and $h(n)$ = estimated cost from n to the goal; $f(n)$ is the estimated cost of the cheapest solution through n . A* always expands the node with the lowest f .

```
OPEN <- [start], g(start) = 0
loop:
  if OPEN empty: return FAILURE
  n <- node in OPEN with smallest f = g + h
  if GOAL(n): return path
  move n to CLOSED
  for each successor s of n:
    new_g = g(n) + cost(n, s)
    if s is new, or new_g < g(s): set g(s) = new_g, parent(s) = n; put s in OPEN
```

- **Complete:** yes. **Optimal:** yes if h is admissible (tree search) or consistent (graph search).

- **Time:** exponential in the worst case. **Space:** $O(b^d)$ — keeps all generated nodes (its main weakness).

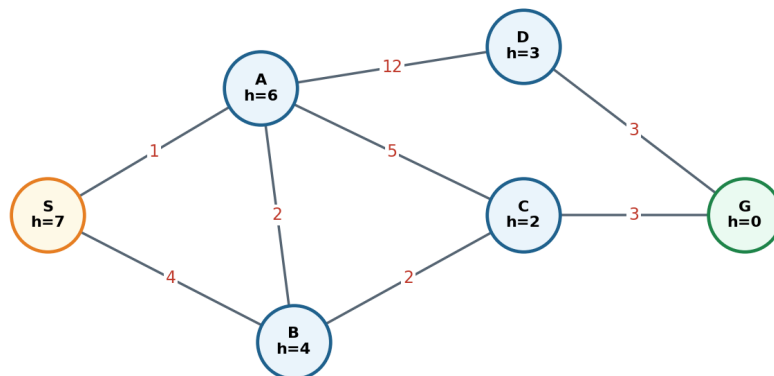


Figure 2.3 — Example graph (edge costs in red, heuristic h inside each node)

Example — Greedy best-first vs. A* on the graph of Figure 2.3 (start S, goal G)

Greedy ($f = h$): from S choose B ($h = 4$) rather than A ($h = 6$); from B go to C ($h = 2$); then G. Path S–B–C–G has cost $4 + 2 + 3 = 9$ (not optimal).

Step	Expanded node	Nodes generated, with g and $f = g + h$	OPEN after step
1	S ($f = 7$)	A: $g=1, f=7$; B: $g=4, f=8$	A(7), B(8)
2	A ($f = 7$)	B via A: $g=3, f=7$ (better than 4); C: $g=6, f=8$; D: $g=13, f=16$	B(7), C(8), D(16)
3	B ($f = 7$)	C via B: $g=5, f=7$ (better than 6)	C(7), D(16)
4	C ($f = 7$)	G: $g=8, f=8$	G(8), D(16)
5	G — goal!	Path S–A–B–C–G, cost 8	—

A* returns the optimal path S–A–B–C–G with cost 8, while greedy search returned a path of cost 9.

Optimality of A*

Claim: if h is admissible, A* (tree search) returns an optimal solution. **Proof sketch:** let C^* be the optimal cost and suppose A* is about to select a suboptimal goal G_2 from the frontier. Then $f(G_2) = g(G_2) > C^*$, because $h(G_2) = 0$. But some node n of an optimal path is still on the frontier, and $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = C^*$ by admissibility. Hence $f(n) \leq C^* < f(G_2)$, so n would be chosen before G_2 — a contradiction. With a **consistent** h , f never decreases along a path, so the first goal selected for expansion is optimal in graph search as well.

Note: If $h_2(n) \geq h_1(n)$ for all n (both admissible), h_2 **dominates** h_1 and A^* with h_2 expands no more nodes than with h_1 .

Heuristics for the 8-Puzzle

- h_1 = **number of misplaced tiles** (not counting the blank).
- h_2 = **sum of Manhattan distances** of tiles from their goal positions. h_2 dominates h_1 and both are admissible.

Example — A* on an 8-puzzle with h = Manhattan distance

Start: 1 2 3 / 4 _ 6 / 7 5 8 Goal: 1 2 3 / 4 5 6 / 7 8 _

$h(\text{start}) = 2$ (tile 5 and tile 8 are each 1 move away). Successors of the start state, $f = g + h$:

Move blank	Resulting state	g	h	f
Up	1 _ 3 / 4 2 6 / 7 5 8	1	3	4
Left	1 2 3 / _ 4 6 / 7 5 8	1	3	4
Right	1 2 3 / 4 6 _ / 7 5 8	1	3	4
Down	1 2 3 / 4 5 6 / 7 _ 8	1	1	2

A* expands the Down state ($f = 2$); moving the blank Right gives the goal with $g = 2$, $h = 0$, $f = 2$. The solution has **2 moves**. An uninformed search would have expanded many more states.

Local Search

Local search operates on a **single current state** and moves only to neighbouring states, forgetting the path. It uses very little memory and is useful for optimization and when only the final state matters (e.g., N-queens, scheduling, circuit layout). Terms in the **state-space landscape**: *global maximum*, *local maximum*, *plateau* (flat area) and *shoulder*.

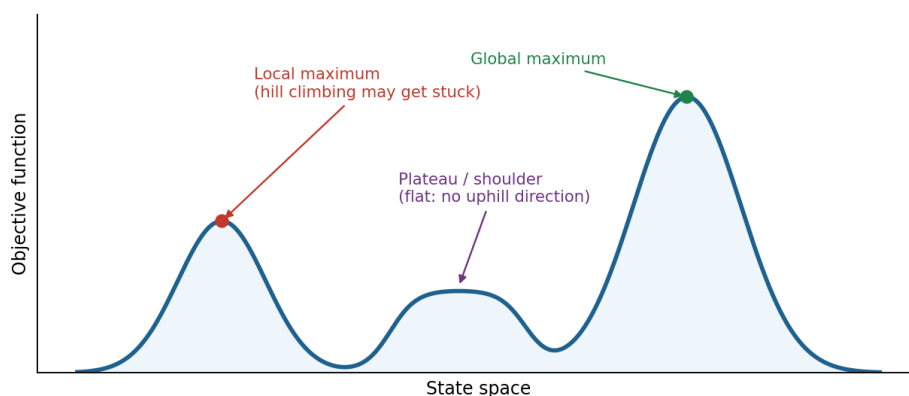


Figure 2.4 — A one-dimensional state-space landscape

Hill Climbing

Repeatedly moves to the neighbour with the best value; stops when no neighbour is better (a “peak”). It is like climbing a mountain in thick fog with amnesia.

```
current <- initial state
loop:
  neighbor <- highest-valued successor of current
  if value(neighbor) <= value(current): return current
  current <- neighbor
```

Problem	Why it happens	Remedy
Local maxima	A peak higher than its neighbours but lower than the global maximum	Random-restart hill climbing
Plateaus / shoulders	Flat region: all neighbours have equal value	Allow sideways moves (with a limit)
Ridges	A sequence of local maxima hard to climb with single moves	Stochastic or first-choice variants

Variants: **steepest-ascent** (best neighbour), **stochastic** (random uphill move), **first-choice** (first better neighbour) and **random-restart** (restart from random states; complete with probability 1). On 8-queens (h = number of attacking pairs) steepest-ascent gets stuck about 86% of the time but runs very fast.

Simulated Annealing

Inspired by **annealing** in metallurgy (heating then slowly cooling a metal). To escape local maxima it sometimes accepts **downhill** moves. A random neighbour is chosen; if it is better it is accepted, otherwise it is accepted with probability $e^{(\Delta E/T)}$, where $\Delta E \leq 0$ is the change in value and T is the “temperature”. T starts high (almost a random walk) and decreases by a schedule toward 0 (almost hill climbing). If T falls slowly enough, the algorithm finds the global optimum with probability close to 1.

```
current <- initial state
for t = 1, 2, 3, ...:
  T <- schedule(t)
  if T = 0: return current
  next <- random successor of current
  dE <- value(next) - value(current)
  if dE > 0: current <- next
  else: current <- next with probability  $e^{(dE/T)}$ 
```

Example — Acceptance probability

A move worsens the value by 2 ($\Delta E = -2$). At $T = 1$: probability = $e^{-2} \approx \mathbf{0.135}$. At $T = 10$: probability = $e^{(-0.2)} \approx \mathbf{0.82}$. So bad moves are accepted often when hot and rarely when cold.

Solving N-Queens and Puzzles with Informed Search

- **N-queens (local search)**: use a complete-state formulation — start with N queens, one per column, and let h = number of pairs of queens attacking each other. Move a queen within its column to reduce h (hill climbing / min-conflicts); simulated annealing or random restarts avoid getting stuck. Goal: $h = 0$.
- **8-puzzle (A*)**: use $f = g + h$ with h = Manhattan distance; it finds the shortest solution while expanding far fewer nodes than BFS.

Algorithm	Evaluation function	Complete?	Optimal?
Greedy best-first	$f(n) = h(n)$	No	No
A*	$f(n) = g(n) + h(n)$	Yes	Yes (admissible / consistent h)
Hill climbing	Value of the current state	No	No
Simulated annealing	Value + temperature T	Yes (with slow cooling)	Yes (with slow cooling)

2.5 Game Playing

Games are **adversarial search** problems: two or more agents with conflicting goals alternate moves. The usual setting is a **two-player, turn-taking, zero-sum game with perfect information** (e.g., chess, tic-tac-toe): what is good for one player (**MAX**) is equally bad for the other (**MIN**). A game is formulated by:

- **Initial state**, **PLAYER(s)** (whose turn), **ACTIONS(s)** (legal moves), **RESULT(s, a)** (transition model).
- **TERMINAL-TEST(s)** (is the game over?) and **UTILITY(s, p)** (final payoff, e.g., +1 win, 0 draw, -1 loss).

The **game tree** has game states as nodes and moves as edges; one move by one player is one **ply**.

Optimal Decisions in Games — Minimax Algorithm

MAX wants to maximize and MIN wants to minimize the final utility. The **minimax value** of a state is the utility of being in that state, assuming both players play optimally to the end:

- $\text{MINIMAX}(s) = \text{UTILITY}(s)$, if s is terminal.
- $\text{MINIMAX}(s) = \max$ over actions of $\text{MINIMAX}(\text{RESULT}(s, a))$, if it is MAX's turn.
- $\text{MINIMAX}(s) = \min$ over actions of $\text{MINIMAX}(\text{RESULT}(s, a))$, if it is MIN's turn.

```
function MINIMAX-DECISION(state): return action a maximizing MIN-VALUE(RESULT(state, a))

function MAX-VALUE(s):
    if TERMINAL(s) return UTILITY(s)
    v = -infinity
    for each action a:
        v = max(v, MIN-VALUE(RESULT(s, a)))
    return v

function MIN-VALUE(s):
    if TERMINAL(s) return UTILITY(s)
    v = +infinity
    for each action a:
        v = min(v, MAX-VALUE(RESULT(s, a)))
    return v
```

- **Complete:** yes (finite tree). **Optimal:** yes against an optimal opponent.
- **Time:** $O(b^m)$. **Space:** $O(bm)$ (depth-first). For chess $b \approx 35$, $m \approx 100$, so exact minimax is infeasible.

Example — Minimax on the tree of Figure 2.5

MAX moves at the root; MIN moves at the middle level; the leaves hold utilities 3, 12, 8 | 2, 4, 6 | 14, 5, 2.

MIN nodes take the **minimum** of their leaves: $\min(3, 12, 8) = 3$, $\min(2, 4, 6) = 2$, $\min(14, 5, 2) = 2$. The MAX root takes the **maximum**: $\max(3, 2, 2) = 3$. MAX should play the left move.

Alpha-Beta Pruning

Minimax examines every node of the game tree. **Alpha-beta pruning** returns the *same* result but skips branches that cannot influence the decision. It keeps two values along the current path:

- α = value of the best (highest) choice found so far for MAX.
- β = value of the best (lowest) choice found so far for MIN.
- **Prune** the remaining children of a node as soon as $\alpha \geq \beta$.

```
function ALPHA-BETA(s, alpha, beta) # MAX node (MIN node is symmetric)
    if TERMINAL(s): return UTILITY(s)
    v = -infinity
    for each action a:
        v = max(v, ALPHA-BETA(RESULT(s, a), alpha, beta))
        if v >= beta: return v # prune
        alpha = max(alpha, v)
    return v
```

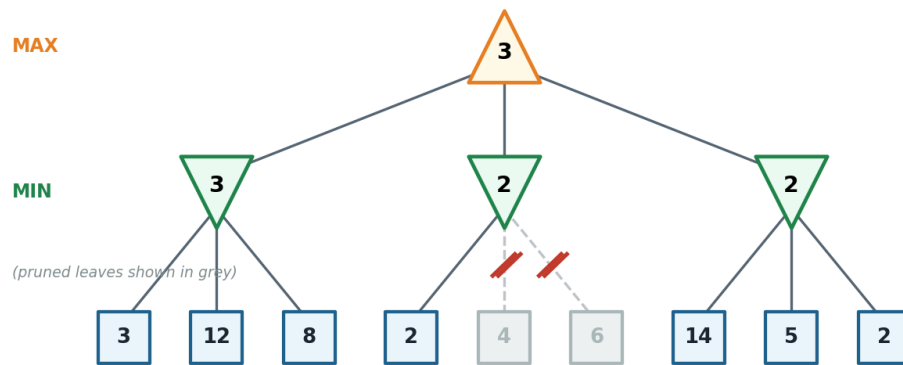


Figure 2.5 — Minimax values and alpha-beta pruning (leaves 4 and 6 are never examined)

Step	Node examined	Effect
1	MIN node 1: leaves 3, 12, 8	Value 3 $\Rightarrow$ root $\alpha = 3$
2	MIN node 2: first leaf 2	$\beta = 2 \leq \alpha = 3 \Rightarrow$ prune leaves 4 and 6; node $2 \leq 2$
3	MIN node 3: leaves 14, 5, 2	β drops $14 \rightarrow 5 \rightarrow 2$; value 2
4	Root	$\max(3, 2, 2) = 3$ — same answer as minimax

- Pruning never changes the final minimax value. Its effectiveness depends on **move ordering**.
- With perfect ordering the time is $O(b^{(m/2)})$ — the search can go twice as deep in the same time; random ordering gives about $O(b^{(3m/4)})$.

Evaluation Functions and Cutoffs

Real games are too big to search to terminal states. The search is **cut off** at a fixed depth and a heuristic **evaluation function** EVAL(s) estimates the utility of non-terminal states (e.g., in chess a weighted sum of material: pawn 1, knight/bishop 3, rook 5, queen 9, plus position features).

Tic-Tac-Toe Problem

Two players, X (MAX) and O (MIN), alternately mark a 3×3 grid; three in a row wins. Utility: +1 if X wins, 0 for a draw, -1 if O wins. The game tree has at most $9! = 362,880$ move sequences (255,168 distinct games and 5,478 distinct board positions). With optimal play by both, the game is always a **draw**.

Example — Evaluation function for tic-tac-toe

$E(s) = (\text{number of rows, columns and diagonals still open for X}) - (\text{number still open for O})$.

Suppose X is in the centre and O in a corner (top-left).

Lines open for X (no O in them): $8 - 3 = 5$. Lines open for O (no X in them): row 1, row 3, column 1, column 3 = 4. $E = 5 - 4 = +1$, a slight advantage to X. Larger E means a better position for MAX.

Steps of minimax search: (1) generate the game tree to a depth limit, (2) apply EVAL to the leaves, (3) back values up using max at MAX levels and min at MIN levels, (4) play the move with the best backed-up value, (5) repeat after the opponent moves. Alpha-beta makes this faster.

Stochastic Games

Games with a chance element (dice in backgammon, shuffled cards) have **chance nodes** in the tree in addition to MAX and MIN nodes. Exact minimax values are replaced by **expectiminimax** values: at a chance node the value is the **expected value** = $\sum P(\text{outcome}) \times \text{value}(\text{outcome})$.

Example — Expectiminimax

MAX chooses between two moves, each leading to a chance node: Move A gives value 2 with probability 0.9 and 3 with probability 0.1; Move B gives 1 with probability 0.9 and 4 with probability 0.1.

Expected values: $A = 0.9 \times 2 + 0.1 \times 3 = 2.1$; $B = 0.9 \times 1 + 0.1 \times 4 = 1.3$. MAX chooses Move A.

Note: Chance multiplies the branching factor, so deep lookahead becomes much costlier; evaluation functions must also be scaled carefully. Card games with hidden cards are partially observable.

Key Takeaways

- ✓ A well-defined problem has an initial state, actions, transition model, goal test and path cost; the state space is a graph of states.
- ✓ CSPs (N-queens, cryptarithmic, map coloring) are solved by backtracking with MRV/forward checking.
- ✓ BFS: complete and optimal but memory-hungry. DFS: low memory but incomplete. IDDFS: best blind choice. Bidirectional: $O(b^{(d/2)})$.
- ✓ Informed search uses $h(n)$. Greedy $f = h$ (fast, not optimal); $A^* f = g + h$ (complete and optimal for admissible/consistent h).
- ✓ Hill climbing can get stuck on local maxima, plateaus and ridges; simulated annealing escapes using probability $e^{-(\Delta E/T)}$.

- ✓ Minimax gives optimal play in two-player zero-sum games; alpha–beta pruning gives the same result faster; chance nodes use expected values.

Review Questions

- Define a well-defined problem. Formulate the 8-puzzle and the water jug problem.
- Solve $\text{SEND} + \text{MORE} = \text{MONEY}$, and color the map of Australia with 3 colors.
- Compare BFS, DFS, DLS, IDDFS and bidirectional search on completeness, optimality, time and space.
- Explain A* search. Prove that A* is optimal with an admissible heuristic.
- Differentiate greedy best-first search and A*. What is an admissible heuristic?
- Explain hill climbing and its problems. How does simulated annealing overcome them?
- Explain the minimax algorithm and alpha–beta pruning with an example tree.
- Solve the 4-queens problem with backtracking.

— *End of Unit 2* —

Reminder: these notes may contain errors — verify with your textbook and teacher.