Module 2: Searching

How to systematically explore possibilities to find solutions — from blind search to game-playing.

PECST522 — Artificial Intelligence | B.Tech CSE S5 (2024 Scheme)

Module 2 — Roadmap

10 contact hours (largest module!) | Maps to CO2 | Difficulty: ☆☆☆

2.1	**What Is Search?**	Motivation, search tree, key terminology
2.2	**Uninformed Search**	BFS, DFS, IDS — blind exploration strategies
2.3	**Informed Search**	Heuristics, Greedy, A* — smart exploration
2.4	**CSP**	Constraint Satisfaction Problems — assignment, not paths
2.5	**Adversarial Search**	Minimax, Alpha-Beta — games with opponents

2.1

What Is Search?

From problem formulation (Module 1) to actually finding the solution.

Why Do We Need Search?

Module 1 defined problems (states, actions, goals). But defining ≠ solving.

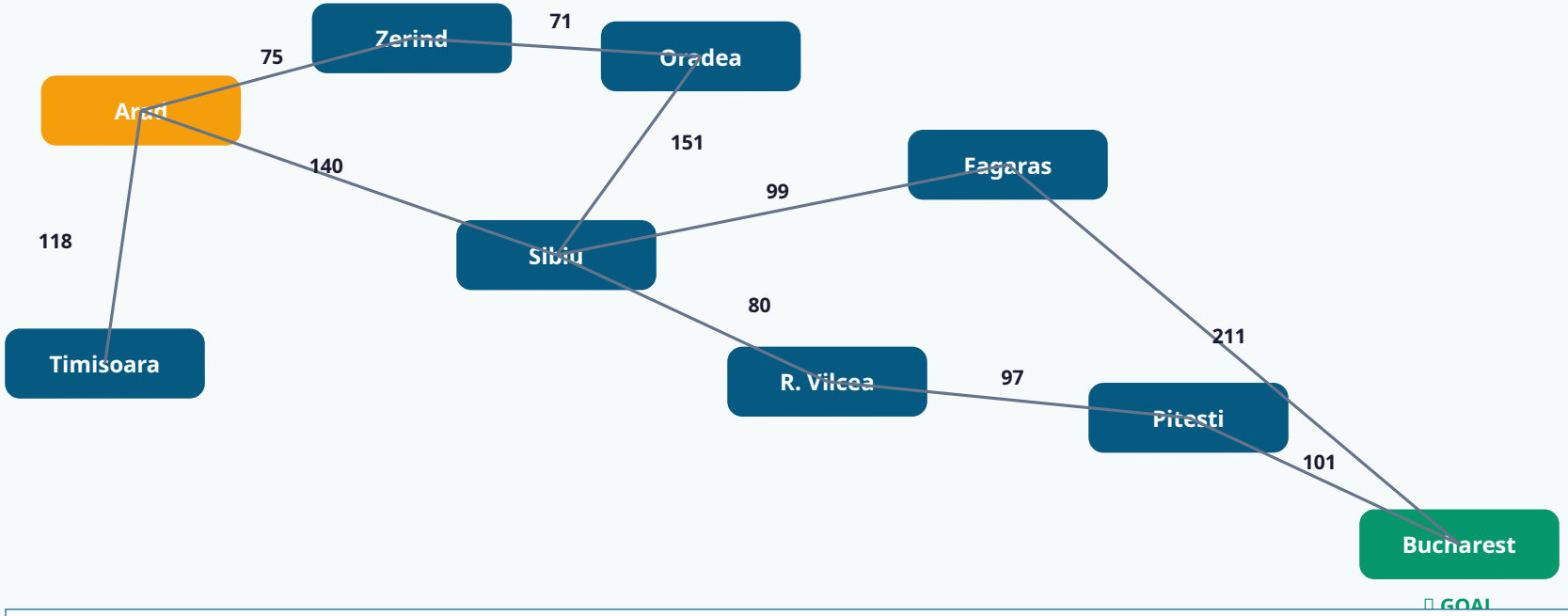
We need a systematic way to explore the state space and find a path from start to goal.

🗺 Analogy

You're in Palakkad, want to reach Kochi. Roads exist (actions), you know what Kochi looks like (goal test). But you still need to NAVIGATE — try routes, backtrack from dead ends, pick the shortest path. That navigation IS search.

Search = systematic exploration of the state space to find a path from initial state to goal state.

The Romania Road Map — Our Running Example

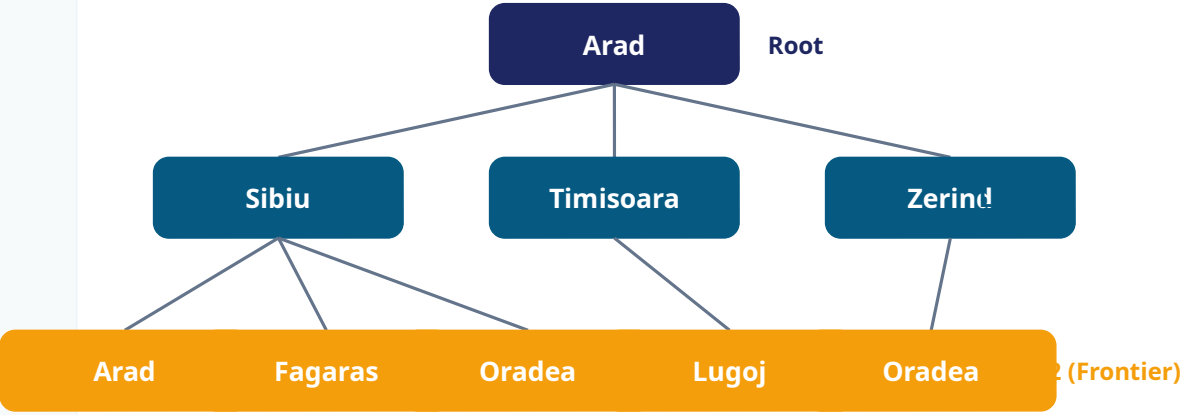


This is a road network. Each city = a state. Each road = an action. The numbers = distances (costs). When we 'search', we explore this map systematically — that exploration forms a SEARCH TREE.

Source: AIMA (Russell & Norvig) — the classic AI textbook example

The Search Tree

Starting from Arad on the Romania map, here's how the search tree grows as we explore:



Root — Initial state

Expanded — States explored

Frontier — Discovered, not yet expanded

Branches — Actions connecting states

BFS grows WIDE (level by level). DFS grows DEEP (one branch fully before backtracking).

Key Search Terminology

Term	Meaning
Node	A state in the search tree (includes parent pointer, action, path cost)
Expand	Generate all children of a node (apply all available actions)
Frontier	Set of nodes discovered but not yet expanded (a.k.a. fringe / open list)
Explored set	Nodes already expanded (a.k.a. closed list) — avoids revisiting states
Solution	Path from root to a goal node
Optimal solution	Solution with the lowest total path cost

2.2

Uninformed (Blind) Search

BFS, DFS, and IDS — algorithms that know nothing about goal distance.

Breadth-First Search (BFS)

Explore ALL nodes at depth d before ANY node at depth d+1. Uses a FIFO queue.

Like searching for a lost key: check every room on Floor 1 before going to Floor 2.

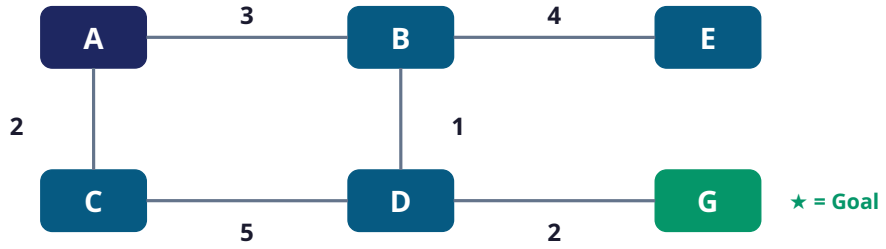
Algorithm:

1. Put start node in the queue
2. Take the FRONT of the queue, check if it's the goal
3. If not, expand it → add all children to the BACK of the queue
4. Repeat until goal found or queue empty

⚠ BFS finds the SHALLOWEST goal (fewest actions), not necessarily the cheapest-cost one.

BFS — Worked Example

Graph (edges = costs):



BFS Trace (FIFO queue — no costs used for exploration):

Step	Queue (FIFO)	Expand	Children Added	Parent
0	[A]	—	—	—
1	[B, C]	A	B, C	A→B, A→C
2	[C, D, E]	B	D, E	B→D, B→E
3	[D, E]	C	(D already seen)	—
4	[E, G]	D	G	D→G
5	**[ ]**	**G = GOAL ✓**	**—**	**—**

Path (trace parents backward): G ← D ← B ← A → A → B → D → G | Cost (computed AFTER): 3+1+2 = 6

⚠ BFS does NOT use costs to decide what to explore. It expands level-by-level (FIFO). Cost is calculated AFTER the path is found.

BFS — Properties

Complete?	Yes — always finds a solution if one exists
Optimal?	✓ ONLY if all edge costs are equal (finds shallowest goal = fewest hops, NOT lowest cost)
Time	$O(b^d)$ — b = branching factor, d = depth of shallowest goal
Space	$O(b^d)$ — must store entire frontier 😱

⚠ The Memory Problem

With b=10, d=12: frontier = 10^{12} nodes = terabytes of RAM. BFS is impractical for deep problems!

Depth-First Search (DFS)

Go as DEEP as possible down one branch before backtracking. Uses a LIFO stack.

Like exploring a maze: follow one corridor to its end, then backtrack to the last junction.

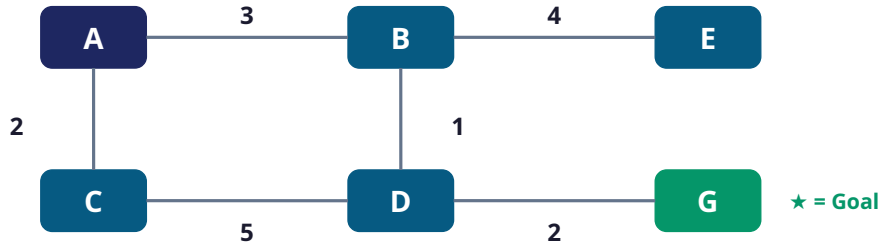
Algorithm:

1. Put start node on the stack
2. Take the TOP of the stack, check if it's the goal
3. If not, expand it → push all children onto the TOP of the stack
4. Repeat until goal found or stack empty

Why use DFS? MEMORY. Only keeps one path in memory: O(b·m) space vs BFS's $O(b^d)$.

DFS — Worked Example

Same graph (children pushed in alphabetical order):



DFS Trace (LIFO stack — no costs used for exploration):

Step	Stack (LIFO)	Expand	Children Pushed	Parent
0	[A]	—	—	—
1	[B, C]	A	B, C (C on top)	A→B, A→C
2	[B, D]	C	D	C→D
3	[B, G]	D	G	D→G
4	**[B]**	**G = GOAL ✓**	**—**	**—**

Path (trace parents): G ← D ← C ← A → A → C → D → G | Cost (computed AFTER): 2+5+2 = 9 (NOT optimal!)

⚠ DFS also doesn't use costs — it goes deep first (LIFO stack). Cost is only computed AFTER the path is found.

DFS — Properties

Complete?	No! Can get stuck in infinite branches (yes if finite + explored set)
Optimal?	No — finds A goal, not necessarily the best one
Time	$O(b^m)$ — m = maximum depth (could be much > d)
Space	O(b·m) — only stores ONE path + siblings ✓✓✓

Trade-off: DFS uses minimal memory but may find suboptimal paths or get lost in deep branches.

Iterative Deepening Search (IDS)

> **Run DFS with limit 1, then limit 2, then limit 3... combining BFS's optimality with DFS's low memory.**

Why the re-work isn't as bad as it seems:

- Most nodes are at the DEEPEST level (b=10 → 90% of nodes at last level)
- Overhead of repeating shallow levels: only ~11% extra work for b=10
- You get: Complete ✓ | Optimal (uniform costs) ✓ | O(b·d) memory ✓

 "If the exam asks which uninformed strategy is preferred, the answer is usually IDS."

IDS — Visual Walkthrough

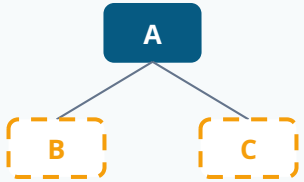
How Depth-Limited Restarts Work (Goal Test vs Expansion)

▤ **Expanded** (depth < limit: pop → goal test → generate children) ☐**Goal-Tested Only** (depth = limit: pop → goal test → STOP, no children) ▧**Goal Found**

d = 0

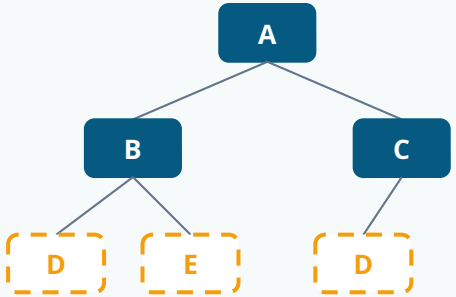
A ≠ G → STOP
(at limit, no expand)

d = 1



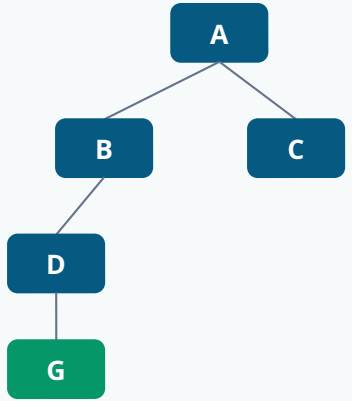
Pop B: B≠G, at limit → no expand
Pop C: C≠G, at limit → no expand

d = 2



D,E,D: all at limit
goal-tested → none is G → fail

d = 3



Pop G: G=G ✓ FOUND!

⚡ **Key Insight: Goal Test ≠ Expansion**
Every node popped from the stack is **goal-tested** (cheap comparison: is this node = G?). But only nodes with **depth < limit** get **expanded** (children generated and pushed to stack).
Nodes AT the limit are checked and discarded — that's how IDS knows the goal isn't at depth d.

IDS Worked Example

Iterative Deepening Search — Same Graph as BFS/DFS

Iteration Trace

Depth Limit	Nodes Expanded (order)	Nodes at Frontier	Goal Found?
d = 1	A	B, C	✗
d = 2	A → B, C	D, E, D	✗
d = 3	A → B → D, E → C → D	G, …	✓ G!

Result at d = 3

Path found: A → B → D → G | Cost = 3 + 1 + 2 = **6**

IDS finds the shallowest goal (like BFS) but uses only O(b·d) memory.

Graph (edge costs)

A–B:3 A–C:2 B–D:1
B–E:4 C–D:5 D–G:2

Goal: G Start: A

Key Insight

Repeated work?
Yes — most nodes re-expanded.

But overhead is small!
Last iteration dominates:
Total ≈ $b^d \times b/(b-1)$

Memory: O(b·d)
— same as DFS!

BFS vs DFS vs IDS — Comparison

Criterion	BFS	DFS	IDS
Complete?	✓ Yes	✗ No (infinite paths)	✓ Yes
Optimal?	✓ (uniform cost)	✗ No	✓ (uniform cost)
Time	$O(b^d)$	$O(b^m)$	$O(b^d)$
Space	$O(b^d)$ 😱	$O(b \cdot m)$ ✓	$O(b \cdot d)$ ✓
When to use	Shallow goals, memory OK	Deep solutions, memory tight	**DEFAULT choice for blind search**

Uniform Cost Search (UCS)

Always Expand the Cheapest Node So Far

Definition

Expand the node with the lowest **g(n)** (path cost so far).
Uses a priority queue ordered by g(n).

Evaluation function: **`f(n) = g(n)`** (no heuristic!)

Analogy

"Always take the cheapest next step, regardless of direction."

Properties

Complete: ✓ **(step costs > 0)**
Optimal: ✓ **(cheapest path)**

Time:
`O(b^(1+⌊C*/ε⌋))`
Space: **`same as time`**

C=opt cost, ε=min step*

Key Insight

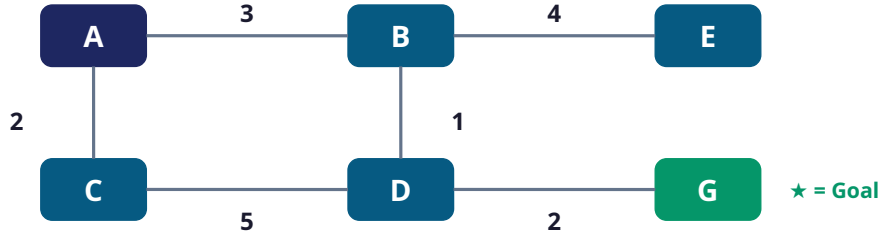
BFS with uniform costs = UCS
UCS = A* with h(n) = 0
Bridge between blind search and A.*

⚠ Warning

UCS is optimal but **SLOW**
Explores all directions without guidance.
Need
heuristics → A*

UCS Worked Example

Same graph as BFS / DFS (edges = costs):



Priority Queue Trace (ordered by g(n))

Step	Priority Queue [node : g(n)]	Expand	g	Children / Updates
1	[A : 0]	A	0	B:3, C:2
2	[C:2], [B:3]	C	2	D:7 (2+5)
3	[B:3], [D:7]	B	3	D:4✓(3+1) E:7(3+4)
4	[D:4], [E:7]	D	4	G:6 (4+2)
5	[G:6], [E:7]	G ✓	6	GOAL REACHED

Key: D updated 7→4 C found D at cost 7 (2+5), but B found a cheaper path: 3+1 = 4. UCS always keeps the lowest g(n).

✓ **Optimal: A → B → D → G** Cost = 3+1+2 = **6**

BFS found same path (cost 6) — got lucky! **UCS GUARANTEES** optimality.

UCS — Why Does g(n) Get Updated?

The distinction that makes UCS optimal

✓ Node POPPED from Queue

Its g(n) is FINAL and OPTIMAL.

Why? The priority queue always pops the lowest-cost node. So by the time a node is popped, **no cheaper path can exist** — every unexplored path costs at least as much (they're still in the queue behind it).

⟳ Node STILL IN Queue

Its g(n) is TENTATIVE — can be lowered.

When expanding a neighbour, if we find a **cheaper path** to a node already in the queue, we **update (relax)** its g(n). The old, costlier entry is replaced.

Example from the Worked Trace

Step 2: C is popped (g=2), discovers D → adds `D : g(n) = 7` to queue. D is **in the queue** — tentative.
Step 3: B is popped (g=3), discovers D via cost 3+1=4. Since **`4 < 7`**, D's g(n) is **`updated 7 → 4`** ✓
Step 4: D is popped (g=4). Now its cost is **FINAL** — no cheaper path possible.

Takeaway: "Discovered" ≠ "Optimal." Popping settles the cost. The queue is where relaxation happens.

2.3

Informed (Heuristic) Search

Giving search a 'sense of direction' with heuristic functions.

Key Definitions — Heuristics

Heuristic h(n)

Estimate of cost from node n to nearest goal. h(goal)=0, h(n)≥0.
Example: straight-line distance from n to goal.

Admissible

h(n) NEVER OVERESTIMATES the true cost. h(n) ≤ actual cost.
Why: A* with admissible h is guaranteed optimal.

Consistent

h(n) ≤ cost(n→n') + h(n') for every successor n'.
Consistent ⟹ admissible. Most natural heuristics are consistent.

Generate and Test

The Simplest Search Strategy

Definition

1. Generate a candidate solution
2. Test if it is the goal
3. Repeat until goal found or all candidates exhausted

Two Versions

Systematic (Exhaustive)
Try every possibility
E.g., Dictionary attack

Heuristic (Guided)
Use domain knowledge
E.g., Common passwords first

Examples of Generate & Test

 4-Digit PIN: Try 0000→9999 (10,000 combos)
 Password Cracking: Try all char combinations (26^n)
 Drug Discovery: Screen millions of compounds

All brute-force — no structure exploited
→ Motivates organized search (BFS, DFS, A*)

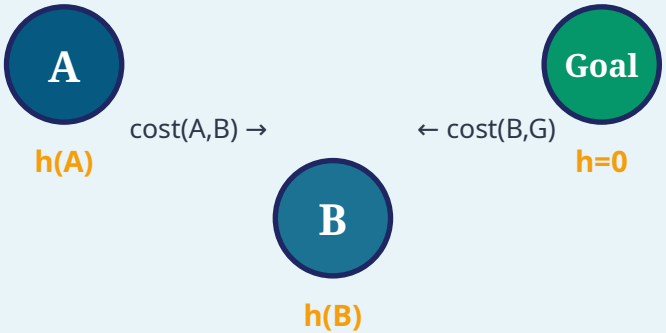
⚠ Limitation

Does not use problem structure
No notion of "promising direction"
Search space explodes combinatorially

→ We need organized search strategies:

Admissible vs Consistent Heuristics — Visual Proof

Consistency: Triangle Inequality



h(A) ≤ cost(A,B) + h(B)

Why Consistency Matters

- Consistent ⟹ Admissible (proof by induction on path length)
- f-values along any path never decrease → A* never re-expands nodes
- Most natural heuristics (SLD, Manhattan) are consistent
- ⚡ Consistency → optimal efficiency with graph search (closed list)

→ See next 2 slides for Manhattan Distance worked example

Key Insight: Consistent ⟹ Admissible (but not vice versa). Most natural heuristics (SLD, Manhattan) are consistent.

Admissible = never overestimates. Consistent = obeys triangle inequality at EVERY step.

Manhattan Distance — 8-Puzzle Example

Manhattan Distance: For each tile, compute |row_current − row_goal| + |col_current − col_goal|. Sum over all tiles (excluding blank). Each tile's distance is the number of horizontal + vertical moves needed — ignoring other tiles.

Current State

row ↓ \ col→	0	1	2
0	1	2	5
1	3	4	6
2	7	8	

Goal State

row ↓ \ col→	0	1	2
0	1	2	3
1	4	5	6
2	7	8	

$$h_2(n) = \Sigma \ \ |\text{row\_i(current)} - \text{row\_i(goal)}| + |\text{col\_i(current)} - \text{col\_i(goal)}| \quad \text{for each tile } i \in \{1..8\}$$

Manhattan Distance — Tile-by-Tile Calculation

Tile	Current (row, col)	Goal (row, col)	|Δrow| + |Δcol|	Distance
1	(0, 0)	(0, 0)	|0−0| + |0−0|	0
2	(0, 1)	(0, 1)	|0−0| + |1−1|	0
3	(1, 0)	(0, 2)	|1−0| + |0−2|	**3**
4	(1, 1)	(1, 0)	|1−1| + |1−0|	**1**
5	(0, 2)	(1, 1)	|0−1| + |2−1|	**2**
6	(1, 2)	(1, 2)	|1−1| + |2−2|	0
7	(2, 0)	(2, 0)	|2−2| + |0−0|	0
8	(2, 1)	(2, 1)	|2−2| + |1−1|	0
			$h_2(n)$ =	**6**

Tiles that need moves: Tile 3 (3) + Tile 4 (1) + Tile 5 (2) = 6

Why admissible? Each tile needs ≥ Manhattan moves (ignoring others only underestimates).

Consistency check: 1. Each move costs exactly 1 (slide one tile into the blank).
2. That move changes only one tile's position — by exactly 1 step.
3. So Manhattan total changes by at most 1 (that tile's distance ±1).
4. Therefore: h(n)−h(n') ≤ 1 = cost(n→n'). ✓

Greedy Best-First Search

Always expand the node that APPEARS closest to goal (lowest h(n)). Ignores path cost so far.

Evaluation function: f(n) = h(n)

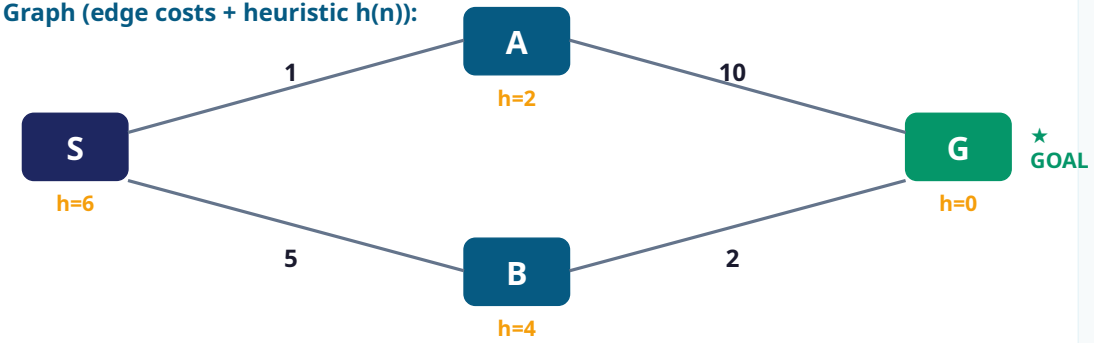
Properties:

- NOT optimal — might find a suboptimal path that "looks close"
- NOT complete — can get stuck in loops
- Often FAST — reaches goal quickly in practice
- Like DFS + heuristic — dives toward goal but can be misled

⚠ Greedy = fast but unreliable. It's the motivation for combining with g(n) → A*

Greedy Best-First Search — Worked Example

Graph (edge costs + heuristic h(n)):



Greedy Expansion:

Expand S
A (h=2), B (h=4)
Pick A ← lowest h

Expand A
G (h=0)
Pick G ← GOAL!

⚡ **Greedy never expands B!**

✗ Greedy Path: S → A → G

Cost = 1 + 10 = 11

Chose A because h(A)=2 < h(B)=4

✓ Optimal Path: S → B → G

Cost = 5 + 2 = 7

Greedy MISSED this — saved 4 units!

⚠ Why did Greedy fail?

Greedy ONLY sees h(n). It chose A because h(A)=2 looks closer than h(B)=4. But the actual edge A→G costs 10, while B→G costs only 2. Greedy ignores g(n) (cost so far), so it gets misled by an optimistic heuristic.

Compare — A* on this graph:

f(A)=1+2=3, f(B)=5+4=9. A* expands A first but keeps B open. When f(G via A)=11 > f(B)=9, A* expands B → finds S→B→G=7. Greedy commits; A* reconsiders.

Non-Completeness:

Without cycle detection, if h leads into a loop (e.g. A→C→A where h(C)<h(A)) Greedy loops forever.

A* Algorithm ☆

THE most important algorithm in Module 2

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

g(n) = actual cost from START to n | h(n) = estimated cost from n to GOAL

Algorithm:

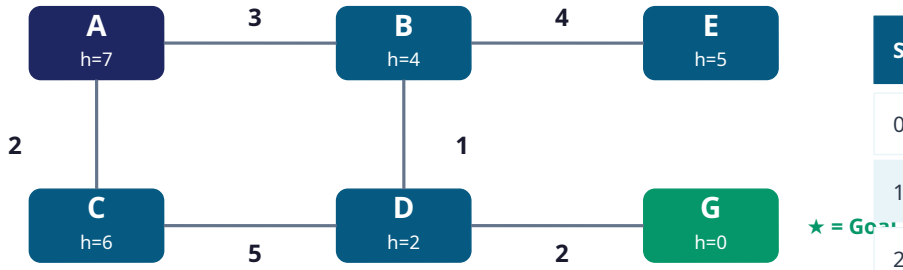
1. Start with initial node in priority queue (ordered by f(n))
2. Remove node with LOWEST f(n)
3. If it's the goal → done!
4. Expand: for each child, g(child) = g(n) + step_cost, f(child) = g(child) + h(child)
5. Add children to queue (update if cheaper path found). Repeat.

🗺 *g(n)=cost so far | h(n)=est. remaining | f(n)=total est. trip cost*

A* with admissible h → guaranteed optimal. The MOST important algorithm in this module.

A* — Complete Worked Example

Graph (edges = cost, h-values in nodes):



A* Trace:

Step	Expand	f=g+h	Frontier
0	—	—	A:7
1	A(f=7)	g(A)=0	B:7, C:8
2	B(f=7)	g(B)=3	D:6, C:8, E:12
3	D(f=6)	g(D)=4	G:6, C:8, E:12
4	**G(f=6)**	**✓ GOAL!**	**—**

Optimal path: A → B → D → G | Cost = 3 + 1 + 2 = 6 ✓

A* — Properties & Heuristic Accuracy

Complete?	Yes (if step costs > 0, finite branching)
Optimal?	Yes, IF h is admissible
Time	Exponential worst case, but much less with good h
Space	$O(b^d)$ — keeps all nodes in memory (same as BFS)

Effect of heuristic accuracy:

- h(n) = 0 → A* becomes Uniform Cost Search (no guidance)
- h(n) = perfect → A* goes straight to goal, O(d) time
- Better h → fewer nodes expanded → faster search

8-Puzzle Heuristics:

h_1 = Misplaced tiles — count tiles NOT in goal position (admissible: each needs ≥1 move)

h_2 = Manhattan distance — sum of horizontal+vertical distances (admissible: can't beat Manhattan)

h_2 dominates h_1 ($h_2 \geq h_1$ everywhere) → h_2 expands fewer nodes → h_2 is better

2.4

Constraint Satisfaction Problems

Finding assignments that satisfy all constraints — not paths, but configurations.

CSP — Three Components

Some problems aren't about finding a PATH — they're about finding an ASSIGNMENT of values satisfying all constraints.

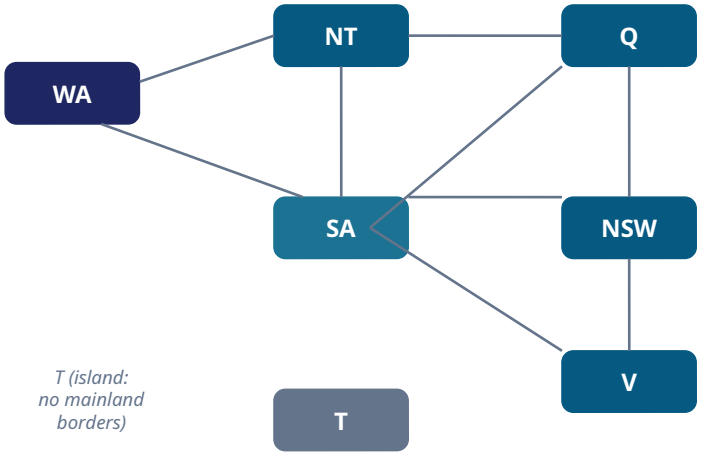
Variables (X)	Domains (D)	Constraints (C)
Things to assign values to	**Possible values for each variable**	**Rules restricting combinations**
{WA, NT, Q, NSW, V, SA, T} (Australian states)	{Red, Green, Blue} for each state	Adjacent states must have different colours

A solution = assignment of every variable such that ALL constraints are satisfied.

Map Colouring — Solving a CSP Step by Step

We defined Variables, Domains & Constraints — now how do we find a valid assignment?
Backtracking: assign one variable at a time; if a constraint is violated, undo (backtrack) and try the next value.

Australian states (adjacency graph):



Backtracking trace:

1. WA = Red	✓ No constraints yet
2. NT = Green	✓ Must ≠ WA(Red)
3. SA = Blue	✓ Must ≠ WA(Red), ≠ NT(Green)
4. Q = Red	✓ Must ≠ NT(Green), ≠ SA(Blue)
5. NSW = Green	✓ Must ≠ Q(Red), ≠ SA(Blue)
6. V = Red	✓ Must ≠ NSW(Green), ≠ SA(Blue)
7. T = Red	✓ No mainland constraints

✓ **SOLUTION: WA=Red, NT=Green, SA=Blue, Q=Red, NSW=Green, V=Red, T=Red**

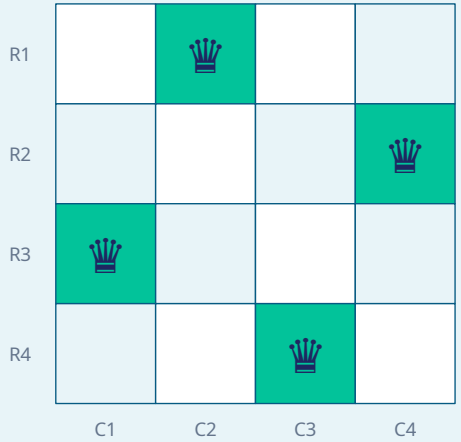
CSP Backtracking — Full Worked Trace

4-Queens Problem (N = 4)

Step-by-Step Trace

1	$Q_1 = 1$	✓ Assign
2	$Q_2 = 3$	✓ Assign (col 1 ✗, diag safe)
3	$Q_3 = ?$	1 ✗col 2 ✗diag 3 ✗col 4 ✗diag → DEAD END
4	$Q_2 \rightarrow 4$ (backtrack)	↩ Retry Q_2 with next domain value
5	$Q_3 = 2$	✓ Assign (cols {1,4}, diag safe)
6	$Q_4 = ?$	1 ✗diag 2 ✗col 3 ✗diag 4 ✗diag → DEAD END
7	$Q_1 \rightarrow 2, Q_2 \rightarrow 4, Q_3 \rightarrow 1, Q_4 \rightarrow 3$	✓ SOLUTION FOUND!

Solution Board



Key Insight Backtracking = DFS + constraint checking at each assignment. Prunes entire subtrees when a dead end is detected.

CSP Techniques

Backtracking Search

Assign variables one at a time. If constraint violated → undo last assignment, try next value. Like Sudoku — if stuck, erase and retry.

Forward Checking

After assigning a variable, remove inconsistent values from UNASSIGNED neighbours' domains. Detects failure EARLY.

MRV Heuristic

Minimum Remaining Values — assign the variable with fewest legal values remaining next. "Fail-first" principle.

Arc Consistency (AC-3)

For every connected pair, ensure every value in one has a compatible value in the other. Shrinks domains before search.

Forward Checking — Worked Example

Australia Map Colouring | Variables: WA, NT, SA, Q, NSW, V, T | Domains: {R, G, B}

Step	Assign	Neighbours Updated	Domains After
1	**WA = Red**	NT, SA	`NT:{G,B}  SA:{G,B}  Q:{R,G,B}  NSW:{R,G,B}  V:{R,G,B}`
2	**NT = Green**	SA, Q	`SA:{B} ← forced!  Q:{R,B}  NSW:{R,G,B}  V:{R,G,B}`
3	**SA = Blue**	Q, NSW, V	`Q:{R}  NSW:{R,G}  V:{R,G}`
4	**Q = Red**	NSW	`NSW:{G}  V:{R,G}`
5	**NSW = Green**	V	**`V:{R}  → V = Red  ✓ SOLUTION`**

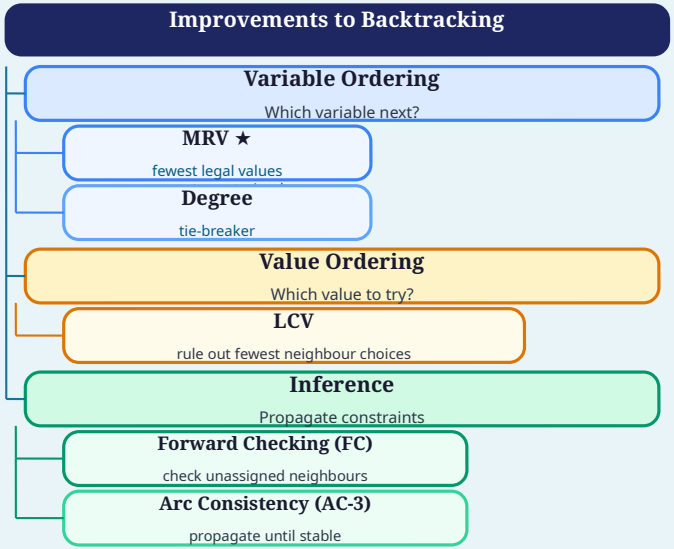
 Forward Checking detects failure IMMEDIATELY when a domain becomes empty — no wasted exploration of doomed subtrees.

Without FC: plain backtracking assigns ALL variables before discovering a conflict → exponentially more work.

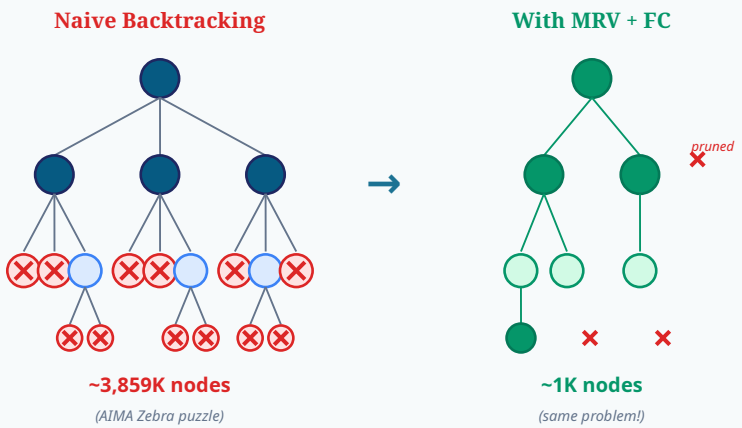
Heuristic Improvements to Backtracking

Fail-First Principle: *"To succeed, try first where you are most likely to fail."* — detect dead ends early, prune more.

Taxonomy of CSP Heuristics



Search Tree Comparison



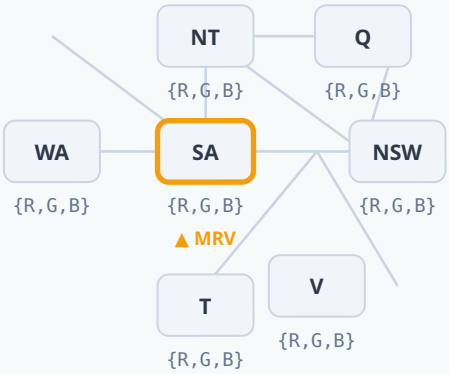
MRV picks the

LCV picks the

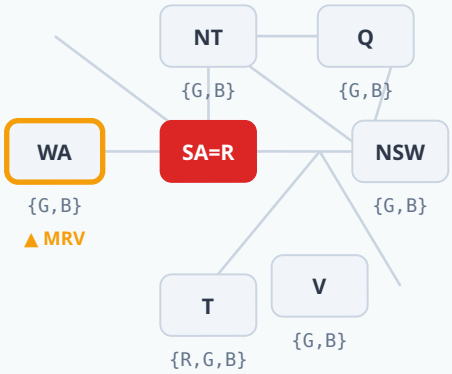
Together confront difficulty but

Best exam combination: **MRV + FC** *(shown in next slide →)*

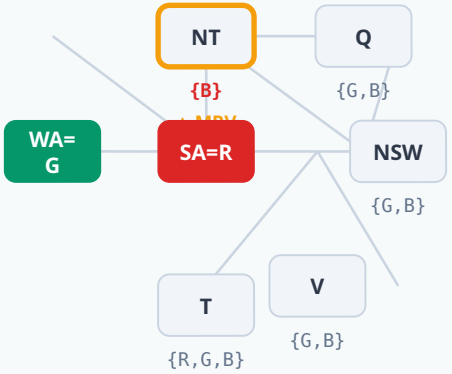
MRV + FC — Visual Trace (Steps 1–3)



All |D|=3 → Degree: SA (5 nbrs)



SA=Red → 5 nbrs lose R



WA=Green → NT:{B} forced!

■ Colour-filled = assigned □ Grey = unassigned (domain below) ▲ Gold border = MRV pick ● Red text = forced

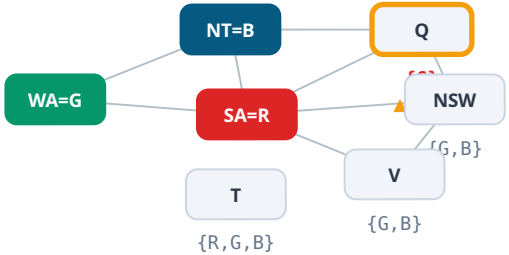
Step 1: All |D|=3 → MRV tie. Degree heuristic: SA has 5 neighbours (most connected) → assign SA=R

Step 2: SA=R → Forward Checking removes R from WA,NT,Q,NSW,V → |D| drops to 2. MRV 5-way tie → degree picks WA

Step 3: WA=G → FC removes G from NT (already lost R via SA) → NT:{B} forced! MRV picks NT (|D|=1)

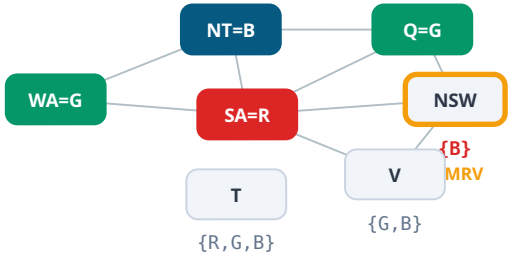
MRV + FC — Visual Trace (Steps 4–Solution)

After Step 3: NT = Blue



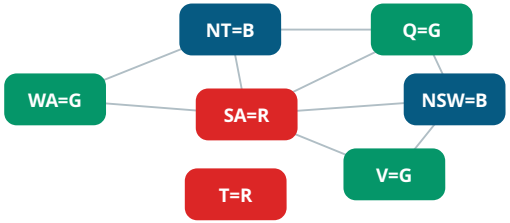
NT=Blue → Q:{G} forced!

After Step 4: Q = Green



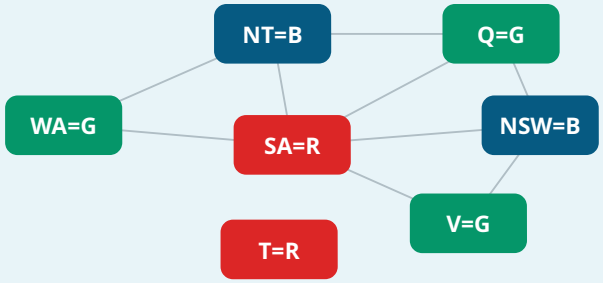
Q=Green → NSW:{B} forced!

After Step 5: NSW = Blue



NSW=Blue → V:{G} → DONE!

★ SOLUTION



✓ COMPLETE — 0 Backtracks!

MRV + FC cascade: Each forced assignment (|D|=1) triggers the next. No search, no backtracking needed.

Final Colouring:

`SA=R` `WA=G` `NT=B` `Q=G` `NSW=B` `V=G` `T=R`

Naive BT: may explore 20+ nodes (many failures)

MRV + FC: 6 assignments, 0 failures — optimal path!

Exam Note: Show domain sizes + MRV selection reason at EACH step. Highlight forced assignments (|D|=1) in red for full marks.

Arc Consistency — AC-3 Algorithm

AC-3 Steps

1. Init queue with ALL arcs (Xi, Xj)

2. Pop arc (Xi, Xj). Remove values from Di that have NO support in Dj

3. If Di changed → re-add all (Xk, Xi) to queue

4. Repeat until queue empty (✓ consistent) or domain empty (✗ no solution)

Example: Failure Detection

`X1∈{1,2}, X2∈{1,2}, X3∈{1,2}`
`X1≠X2, X2≠X3, X1≠X3`

Arc (X1,X2): 1 has support 2 ✓, 2→1 ✓
Arc (X2,X3): same ✓
Arc (X1,X3): 1→2 ✓, 2→1 ✓

Looks OK! But 3 vars, 2 values, all≠/
→ AC-3 alone can't detect this failure
(need backtracking or higher-order k-consistency)

Forward Checking: assigned → unassigned only | AC-3: ALL pairs including unassigned ↔ unassigned

⚡ AC-3: 6/8 CST401 papers (3–14 marks). Must know: algorithm steps + trace a small example.

2.5

Adversarial Search (Game Playing)

When an opponent tries to prevent you from winning.

Optimal Decisions in Games

What Does 'Winning' Mean for an AI?

Game = multi-agent environment where agents have CONFLICTING goals

Key Assumptions

- Two players: **MAX** (us) and **MIN** (opponent)
- Both play **OPTIMALLY** (worst-case reasoning)
- Zero-sum: my gain = your loss
- Perfect information (both see full board)

The Question

If both players are perfectly rational, what is the **BEST move** MAX can make?

Answer: The move leading to the highest-value terminal state, ASSUMING MIN will try to minimize it.

This is the **MINIMAX VALUE**

🏷 Syllabus topic: "Optimal decisions in games" — this is the theoretical foundation for the Minimax algorithm (next slide)

Minimax Algorithm

> **Assume opponent plays perfectly. You MAXIMIZE your score; opponent MINIMIZES it. Alternate MAX/MIN layers.**

The idea:

- MAX nodes (your turn): pick child with HIGHEST value
- MIN nodes (opponent's turn): pick child with LOWEST value
- Leaf nodes: have actual utility values (end-of-game score)

🗺️ *Normal search = solo road trip (you choose all turns). Game search = chess (opponent chooses THEIR turns to make YOUR trip bad).*

Minimax — Setting Up the Game Tree

Before we solve: what does the tree mean?

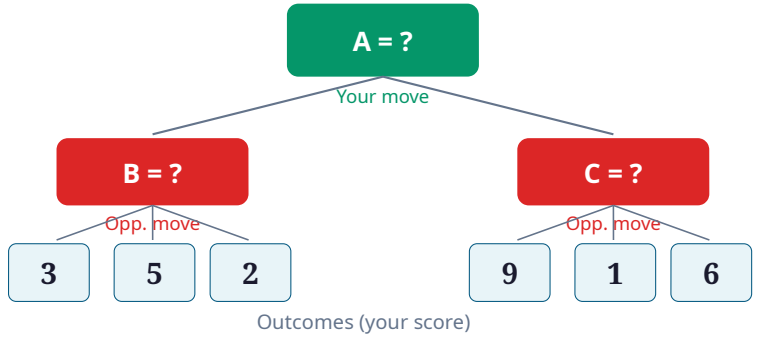
🎮 A Simple Game

You (MAX) pick branch B or C

Opponent (MIN) picks one of three outcomes from that branch

Leaf numbers = **YOUR score** at game end.
Higher is better for you; worse for opponent.

The Unsolved Tree



🤔 The Question:

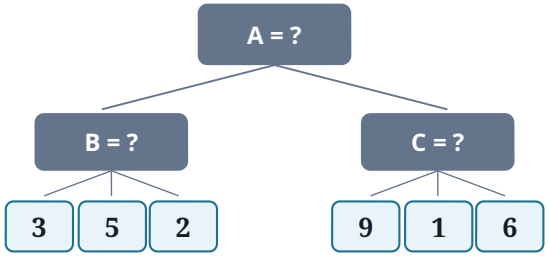
Should you go to B or C?
Branch B has outcomes 3, 5, 2. Branch C has outcomes 9, 1, 6.

Trap: C has the highest single score (9) — but your opponent controls that choice!
A rational opponent at C will pick 1 (lowest), not 9. At B they'd pick 2.

Minimax — Solving Bottom-Up

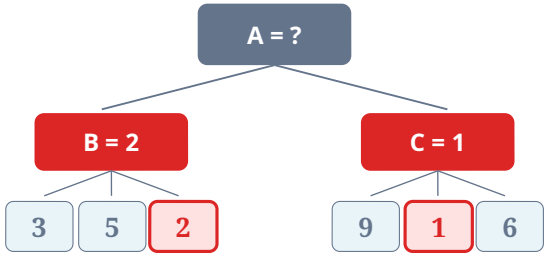
Three steps: leaves → MIN → MAX

Step 1: Read Leaves



These are known — game-end scores. Start here.

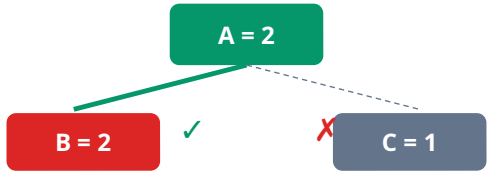
Step 2: MIN Picks Lowest



min(3,5,2) = 2 **min(9,1,6) = 1**

Opponent picks worst-for-you option from each branch.

Step 3: MAX Picks Highest



max(2, 1) = 2 → Go to B

You guarantee at least 2, no matter what opponent does.

Key Insight

C has the highest leaf (9) but it's a TRAP — opponent would give you just 1. Branch B's worst case (2) beats C's (1).

Algorithm pattern: always solve BOTTOM → UP. Leaves are known → propagate through MIN/MAX → root gives optimal move.

Alpha-Beta Pruning

Skip branches that CAN'T affect the final decision. Same answer as minimax, fewer nodes explored.

The parameters:

α (alpha)	Best value MAX can guarantee so far (lower bound). Starts at $-\infty$.
β (beta)	Best value MIN can guarantee so far (upper bound). Starts at $+\infty$.
Prune when	$\alpha \geq \beta \rightarrow$ the window is empty, no useful moves remain

 α = "I (MAX) already guarantee at least THIS much" | β = "Opponent (MIN) limits to at most THIS much"

Alpha-Beta — Worked Example

Same tree: MAX(A) → MIN(B,C) → leaves [3,5,2] and [9,1,6]

Trace:

1. Expand B (left): children 3, 5, 2 → MIN picks 2. B = 2.

2. Back at A (MAX): α = max(−∞, 2) = 2. A guarantees ≥ 2.

3. Expand C (right): α=2 passed down. First child = 9 → β at C = 9.

4. Second child = 1 → C's value ≤ 1. Now β=1. Check: α(2) ≥ β(1)? YES → PRUNE! ✂

5. Third child (6) is NEVER evaluated!

PRUNED TREE:

A = 2 → B = 2 (fully evaluated) | C ≤ 1 (PRUNED after seeing 1; node 6 never checked)

Result: Same optimal value (2, choose B) but skipped evaluating node 6.

Handling Deep Game Trees — Cutoff & Evaluation

The Problem

Chess: ~10^{47} states, branching ~35
Go: ~10^{170} states, branching ~250

Can't search to terminal nodes!

Need to stop early and estimate.

The Solution

Replace:
`TERMINAL-TEST → CUTOFF-TEST`
(stop at depth limit d)

`UTILITY → EVAL(s)`
(heuristic score for non-terminal state)

Evaluation Function — EVAL(s)

Maps state → numeric score (positive = good for MAX, negative = good for MIN)

Chess Example: `EVAL(s) = (Q×9 + R×5 + B×3 + N×3 + P×1) − (opponent same)`
+ positional bonuses (center control, king safety, pawn structure)

Requirements: ① Fast to compute ② Correlate with winning probability ③ Agree with UTILITY at terminals

Better eval function = stronger play at same depth

Modern: AlphaZero / Stockfish NNUE use neural networks as EVAL

Alpha-Beta — Effectiveness

Best case: reduces branching factor from b to √b (squares the searchable depth!)

Chess example (b = 35, depth = 8):

Without pruning: $35^8 \approx$ 2 trillion nodes

With perfect ordering: $35^4 \approx$ 1.5 million nodes ← 1000× fewer!

Key facts:

- Does alpha-beta change the answer? NEVER — same result as minimax, just faster.
- Depends on move ordering — examine best moves first for maximum pruning.
- What if we can't search to end of game? Use cutoff + evaluation function (how chess engines work).

Module 2 — Summary: All Search Algorithms

Algorithm	Key idea	Complete?	Optimal?	Complexity
BFS	FIFO queue	✓ Complete	✓ Optimal (uniform)	$O(b^d)$ memory 😱
DFS	LIFO stack	✗ Complete	✗ Optimal	$O(b \cdot m)$ memory ✓
IDS	Repeated DFS	✓ Complete	✓ Optimal (uniform)	$O(b \cdot d)$ memory ✓
Greedy	$f(n) = h(n)$	✗ Complete	✗ Optimal	Fast in practice
A*	$f(n) = g(n)+h(n)$	✓ Complete	✓ Optimal (admissible h)	$O(b^d)$ memory
CSP/BT	Assignment	✓ (finite domains)	N/A (satisfaction)	Backtrack + prune
Minimax	Game tree	✓ (finite game)	✓ (optimal play)	$O(b^m)$ time
α-β	Minimax + prune	✓ (same as MM)	✓ (same as MM)	$O(b^{m/2})$ best case

Self-Check Questions

1. **Trace BFS and DFS on a given graph. Show the frontier at each step.** ⚡
2. **What is IDS? Why is it preferred over BFS and DFS?** ⚡
3. Compare BFS, DFS, IDS in a table (complete, optimal, time, space).
4. Define: heuristic function, admissible, consistent. Why does admissibility matter for A*?
5. **Write f(n) = g(n) + h(n). Explain each term. Trace A* on a graph.** ⚡
6. What happens to A* if h(n) = 0 for all n? What if h(n) = perfect?
7. Define CSP (variables, domains, constraints). Formulate map-colouring as a CSP.
8. What is forward checking? What is MRV?
9. **Explain minimax with a game tree example. Who picks what at each level?** ⚡
10. **Trace alpha-beta on a game tree. Show which nodes get pruned and WHY.** ⚡
11. Does alpha-beta change the minimax value? What's its best-case time saving?

⚡ = Exam hot topics (Part A: 3 marks | Part B: 9 marks)

Module 2 Complete

Next: Module 3 — Knowledge & Reasoning (Logic, Propositional & First-Order)