

CCES – College of Computer Engineering and Sciences

Prince Sattam Bin Abdulaziz University

CS600: Advanced AI

Multiagent Search

Chapter 3 — Adversarial Games: Minimax, α - β , and MCTS

Dr. Khalil Chebil

Artificial Intelligence Course

Chapter Outline | What we will cover

Multiagent Environments
AND-OR Search Trees
Evaluation & Depth-Limited Search

Minimax
Alpha-Beta & MCTS
Choosing a Method & Summary

Learning Objectives

- ▶ Explain why other agents turn **plans** into **strategies**, and model this with **AND-OR trees**
- ▶ Design **evaluation functions** for depth-limited search
- ▶ Compute **minimax** values by hand and run **alpha-beta** pruning step by step
- ▶ Explain **MCTS** and the UCT formula, and choose the right method for a problem

Section 1

Multiagent Environments

From Plan Finding to Strategy Selection | Other agents react to your moves

Single-agent search (Chapter 2)

An action determines the next state. A solution is a fixed **sequence** of actions.

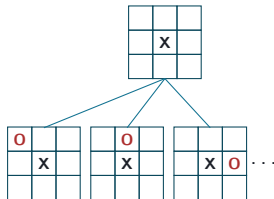
Multiagent search

After your move, someone else moves. One action opens a **set of possible futures**, so a solution is a **strategy**: a move for every reply the others might make.

Key challenges

Others have their own goals and information; outcomes depend on their responses; the tree grows with every reply.

Tic-Tac-Toe: you play X in the centre...



... and O has 8 possible replies

Types of Multiagent Environments | The relationship between agents decides the method

Competitive

Conflicting objectives: one agent's gain is another's loss.

Chess, Go, Tic-Tac-Toe, attacker vs. defender

⇒ **adversarial search** (this chapter)

Independent

Act without coordinating, but still affect one another.

Self-driving cars at an intersection, traders in a market

Cooperative

Share information or a joint objective.

Warehouse robots, rescue drones, multiplayer team games

Game property	Chess / Go	Backgammon	Poker
Deterministic?	yes	no (dice)	no (cards)
Perfect information?	yes	yes	no (hidden hands)
Zero-sum?	yes	yes	yes

Minimax and alpha-beta assume **deterministic, perfect-information, two-player, zero-sum** games.

Section 2

AND-OR Search Trees

AND-OR Search Trees | Our choices vs. outcomes we must handle

Two kinds of nodes

OR node: our turn; **one** working action suffices.

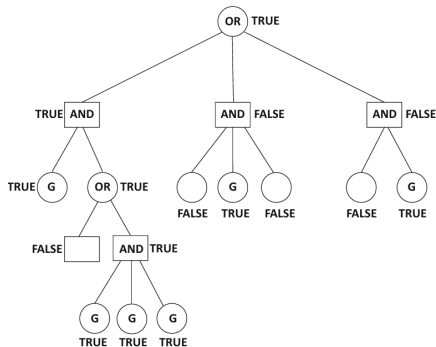
AND node: not our choice; the plan must work for **every** outcome.

Where AND nodes come from

Opponent replies (games), or the possible results of an action (a wheel slips, a packet is lost).

A solution is a sub-tree

One branch at each OR node, all branches at each AND node, every leaf a goal.



Values flow upward: an OR node is TRUE if *some* child is; an AND node only if *all* children are. Only the left move is safe.

AO-Search | Find a guaranteed path to the goal

```
def ao_search(state):  
    if goal_test(state):  
        return True  
    if state.is_or_node():    # our turn  
        return any(ao_search(s)  
                    for s in successors(state))  
    else:                     # opponent  
        return all(ao_search(s)  
                   for s in successors(state))
```

Logic

OR = any: one winning action is enough.

AND = all: every reply must still lead to the goal.

More than two agents

In a 3-agent game (A, B, C) seen by A:
OR nodes = A's choices; AND nodes = the
combined choices of B and C.

Branching blows up

If B and C each have 5 moves, every AND
node has $5 \times 5 = 25$ children. Depth 4 (two
rounds) already gives $(b \cdot 25)^2$ leaves.

Real systems need loop detection: a repeated
state on the current path counts as failure.

Section 3

Evaluation & Depth-Limited Search

Why We Cannot Search to the End | The numbers behind game trees

Game	b	d	size
Tic-Tac-Toe	≤ 9	9	549,946 nodes
Connect Four	7	≤ 42	$\sim 10^{21}$
Chess	≈ 35	≈ 80	$35^{80} \approx 3 \times 10^{123}$
Go	≈ 250	≈ 150	$\approx 10^{360}$

b = branching factor, d = game length; size $\approx b^d$
(Connect Four: positions)

Compare

There are about 10^{80} atoms in the observable universe.

The practical recipe

Search d moves ahead, then **stop** and score the frontier with an **evaluation function**: “how promising does this position look?”

Trade-off

Deeper search sees more tactics but costs exponentially more. A better evaluation function lets a shallower search play well.

Designing Evaluation Functions | Weighted features of a position

Typical form

$$\text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_k f_k(s)$$

Chess features: material (pawn 1, knight 3, rook 5, queen 9), king safety, centre control, mobility.

Tic-Tac-Toe (notebook `ttt_eval`)

For every row, column and diagonal:
+3 two of mine and one empty; +1 one of mine and two empty;
−3, −1 for the opponent's equivalents.

Powerful but imperfect

Few features summarise a rich position; tactics deeper than the cut-off are invisible (the *horizon effect*); decisions are only as good as the evaluation.

Connect Four (notebook)

Reward the centre column, open 2- and 3-in-a-rows; penalise the opponent's threats. Depth-4 alpha-beta from the empty board: play the centre column.

Modern engines (AlphaZero) *learn* the evaluation with a neural network.

Two Agents, Two Objectives | Each agent scores futures with its own function

General-sum version

Return a *pair* of losses; each agent picks the successor that is best *for itself*.

```
def search(state, depth, agent):
    if depth == 0 or is_terminal(state):
        return (loss_a(state), loss_b(state))
    other = 'B' if agent == 'A' else 'A'
    i = 0 if agent == 'A' else 1
    return min((search(s, depth-1, other)
               for s in successors(state)),
               key=lambda pair: pair[i])
```

Why a pair?

If A simply minimised whatever number B's search returned, A would be minimising B's loss, choosing the move that is best *for B*. Each level must compare futures by the mover's own objective.

Zero-sum special case

If $\text{loss}_B = -\text{loss}_A$, one number suffices: A minimises it, B maximises it. Rename A's utility and you get **minimax**.



Section 4

Minimax

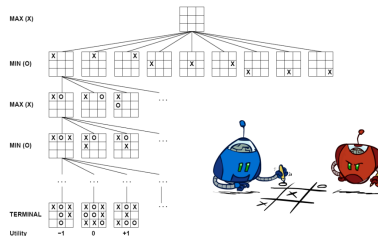
Minimax | Optimal play in two-player zero-sum games

Idea

MAX picks the child with the highest value; **MIN** the lowest. Terminal states give the true payoff; other states take the value of the best move for whoever is to move.

```
def max_value(s, d):
    if d == 0 or terminal(s):
        return utility(s)
    return max(min_value(c, d-1) for c in succ(s))

def min_value(s, d):
    if d == 0 or terminal(s):
        return utility(s)
    return min(max_value(c, d-1) for c in succ(s))
```

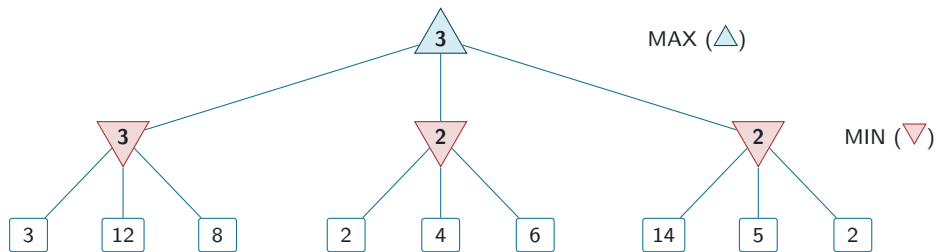


Tic-Tac-Toe: X is MAX, O is MIN; utility
+1 / 0 / -1 for X.

Guarantee

The best outcome MAX can force against a *perfect* opponent.

Minimax Example | Read the tree from the leaves upward



Step by step

MIN nodes: $\min(3, 12, 8) = 3$, $\min(2, 4, 6) = 2$, $\min(14, 5, 2) = 2$
MAX: $\max(3, 2, 2) = 3$: play the left move.

Common mistake

“Go right, there’s a 14!” MIN will never let MAX reach 14: it answers with 2. Look at the *worst* reply, not the best leaf.

Worked Example: Emergency Routing | Dispatcher (MAX) vs. worst-case traffic (MIN)

Route (base score)	none	incident	signal fail	MIN
highway (80)	80	55	65	55
arterial (70)	70	45	55	45
local (60)	60	35	45	35

Penalties: incident -25 , signal failure -15 .

Minimax decision

MAX picks $\max(55, 45, 35)$: **take the highway**, guaranteeing a score of at least 55.

Discussion

Is traffic really an adversary? Minimax assumes the *worst case*. If incidents are rare, an *expected* value (chance nodes) may be a better model.

Section 5

Alpha-Beta & MCTS

Alpha-Beta Pruning | Same answer as minimax, fewer nodes

Bounds

α : the best value MAX can already guarantee on this path.

β : the best value MIN can already guarantee.

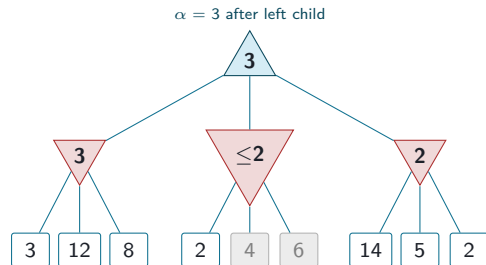
When $\alpha \geq \beta$, the rest of this branch cannot affect the decision: **prune**.

On the minimax example

Left MIN finishes at 3, so $\alpha = 3$ at the root.

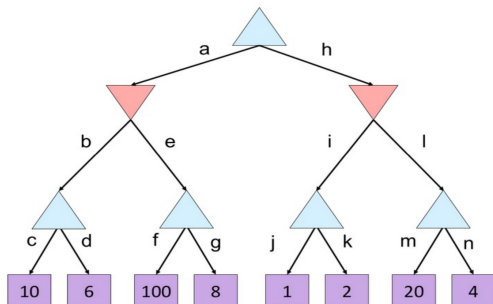
Middle MIN sees leaf 2: its value is now $\leq 2 < 3$. MAX will never choose it, so skip leaves 4 and 6.

Right MIN: 14, 5, 2: all needed (it only drops to 2 at the last leaf).



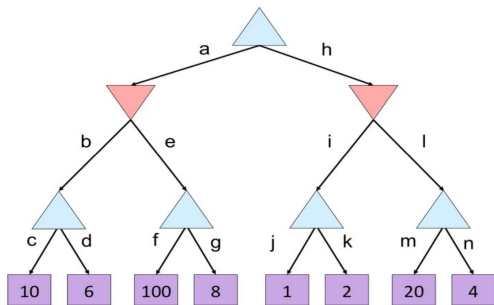
7 of 9 leaves evaluated. The decision (3, left) is unchanged.

Alpha-Beta Quiz | Which branches are never examined? (left to right)

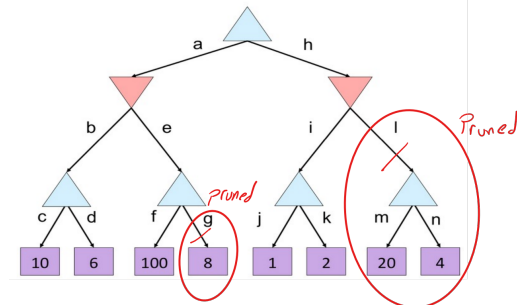


MAX at the root and bottom level; MIN in the middle.

Alpha-Beta Quiz | Which branches are never examined? (left to right)



MAX at the root and bottom level; MIN in the middle.



Why: left MAX gets $\max(10, 6) = 10$, so MIN's $\beta = 10$. The next MAX sees $100 \geq 10$: MIN won't allow it, prune g . Left MIN = 10, root $\alpha = 10$. Right side: MAX gets $\max(1, 2) = 2$, so this MIN is $\leq 2 < 10$: prune all of l . Root value **10**; 5 of 8 leaves evaluated.

Move Ordering Matters | The lab's cyber-defense game

Defense (MAX)	phish	cred.	lateral	MIN
patch critical	60	75	45	45
harden network	70	55	80	55
increase monitoring	85	65	60	60

Minimax: **increase monitoring** (value 60).

Listed order: no pruning at all

Each MIN node's value only drops to its minimum at the last leaf, so all **9** leaves are evaluated.

Best move first: 3 leaves pruned

Monitoring first gives $\alpha = 60$. Patch: first leaf $60 \leq \alpha$, prune the other two. Harden: 70, then $55 \leq 60$, prune 80. Only **6** leaves evaluated.

With perfect ordering alpha-beta costs about $b^{d/2}$ instead of b^d : twice the depth for the same effort.

Alpha-Beta in Practice | Measured on Tic-Tac-Toe from the empty board (notebook)

Algorithm	Nodes	Time	Move
Minimax, full depth	549,946	14.7 s	corner
Alpha-beta, full depth	18,297	0.67 s	corner
MCTS, 1,000 simulations	1,001	0.38 s	centre

Both exact methods find value 0. Alpha-beta: **30× fewer nodes**, same answer.

Reading the results

Corner and centre are both optimal openings (value 0). Minimax and alpha-beta always agree; MCTS picks a strong move from statistics.

Connect Four, depth-4 alpha-beta

737 nodes, 540 cut-off evaluations, 137 prunes: column 3 (the centre).

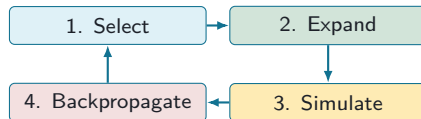
Monte Carlo Tree Search (MCTS) | Estimate by simulation, not exhaustion

Four phases, repeated

1. **Selection**: walk down the tree using UCT
2. **Expansion**: add a new child
3. **Simulation**: play random moves to the end
4. **Backpropagation**: update wins and visits on the path

Why it works

No evaluation function needed; effort goes to promising moves; *anytime*: stop whenever, more simulations give better estimates.



Famous use

AlphaGo (2016) combined MCTS with neural networks to beat world champion Lee Sedol at Go, where minimax is hopeless ($b \approx 250$).

The UCT Formula, Worked | Exploit what looks good, explore what is uncertain

Upper Confidence bound for Trees

$$u_i = \underbrace{\frac{w_i}{n_i}}_{\text{exploit}} + c \underbrace{\sqrt{\frac{\ln N_i}{n_i}}}_{\text{explore}}$$

w_i wins, n_i visits of child i ; N_i visits of the parent; $c \approx \sqrt{2} = 1.41$.

Unvisited children get $u = \infty$, so each is tried once.

Parent visited $N = 20$ times:

Child	w/n	exploit	explore	u
A	6/10	0.600	0.772	1.372
B	3/4	0.750	1.220	1.970
C	1/6	0.167	0.996	1.163

Select B

B has the best win rate *and* few visits, so it gets the bonus. A is more certain but no longer favoured.

Section 6

Choosing a Method & Summary

Choosing the Right Search Method | No method dominates everywhere

Situation	Method	Reason
Contingent planning, uncertain outcomes	AND-OR / AO-search	must cover every continuation
Two-player zero-sum game, small tree	Minimax	optimal under perfect play
Same, deeper trees	Alpha-beta + good move ordering	minimax answer, roughly $b^{d/2}$ work
Huge tree, weak or no evaluation function	MCTS	simulation focuses effort
Hard time limit	iterative deepening	always has a move ready

Beyond board games

Autonomous driving (anticipating other drivers), cybersecurity (attacker vs. defender), robotics (cooperating or blocking robots), auctions and pricing (strategic competitors).

Common Pitfalls | From the labs and exams

1. Chasing the best leaf

MIN will never let you reach it; take the max of the mins.

2. Thinking pruning changes the answer

Alpha-beta returns exactly the minimax value.

3. Ignoring move ordering

Bad order can mean no pruning at all.

4. Mixing perspectives

Every value must be scored from one consistent viewpoint (or a pair, in general-sum games).

5. Treating nature as an adversary

Worst case can be too pessimistic; consider chance nodes.

6. Trusting a shallow search

The horizon effect hides threats just past the cut-off.

Practice Problems | Answers on the next click

Problem	Answer
1 MAX root, three MIN children with leaves (4, 8), (7, 2), (5, 6). Minimax value?	
2 Same tree, alpha-beta left to right: which leaf is pruned?	
3 Reverse the children: (5, 6), (7, 2), (4, 8). Pruned leaves?	
4 UCT, $N = 20$: child with $4/5$ vs. child with $7/12$. Which is selected ($c = 1.41$)?	
5 Why can MCTS beat minimax in Go?	

Practice Problems | Answers on the next click

Problem	Answer
1 MAX root, three MIN children with leaves (4, 8), (7, 2), (5, 6). Minimax value?	$\max(4, 2, 5) = 5$: third move
2 Same tree, alpha-beta left to right: which leaf is pruned?	None: child 2 drops to 2 only at its last leaf, and child 3 never goes below $\alpha = 4$
3 Reverse the children: (5, 6), (7, 2), (4, 8). Pruned leaves?	$\alpha = 5$; child 2: 7 then $2 < 5$ (no more leaves); child 3: $4 < 5$, prune 8
4 UCT, $N = 20$: child with 4/5 vs. child with 7/12. Which is selected ($c = 1.41$)?	4/5: $0.80 + 1.09 = 1.89$ vs. $0.58 + 0.70 = 1.29$
5 Why can MCTS beat minimax in Go?	$b \approx 250$ and no good evaluation; simulations need neither

Chapter Summary | Key takeaways

Modelling

Other agents turn plans into strategies. AND-OR trees capture contingencies; evaluation functions make huge trees tractable.

Algorithms

Minimax: optimal in zero-sum games. Alpha-beta: same answer, far fewer nodes, especially with good ordering. MCTS: simulation for huge trees.

Remember

Think about the worst reply, keep a consistent perspective, and order moves well.

Looking ahead

Ch. 4: **propositional logic**, representing knowledge and reasoning with it.

Thank you!

Questions & Discussion

Next: Chapter 4 — Propositional Logic