

Adversarial Search

Game Playing

CIS 32

Functionally

HW 1 is Up on the Site. Questions? (Paper, PDF, or E-mail submission)

Today:

Adversarial Search (Game Playing)

Adversarial Search

Why Search?

- One of the reasons we use search in AI is to help agents figure out what to do.
- Considering how sequences of actions can be put together allows the agent to **plan**.

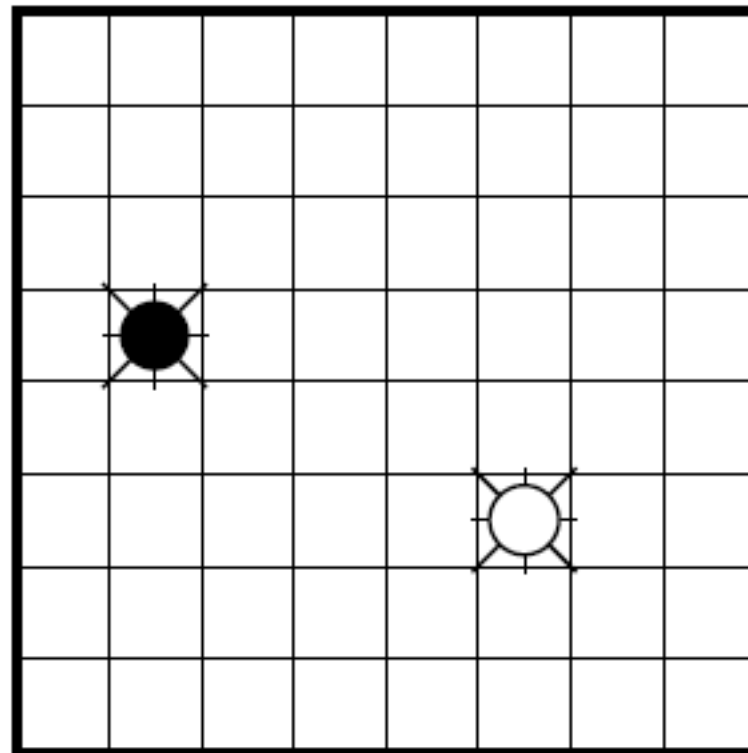
So far, an agent that search can figure out what to do when:

- It knows exactly how the world is;
- Each action only has one outcome; and
- The world only changes when the agent makes it change.

2 Agent Search

- In other words we can plan when the world is:
 - Accessible;
 - Deterministic; and
 - Static
- Obviously these are unrealistic *simplifications*.
- Here we will consider how to handle one kind of dynamism:
 - Other agents messing with the world.
- (Later lectures will look at other kinds of complication.)

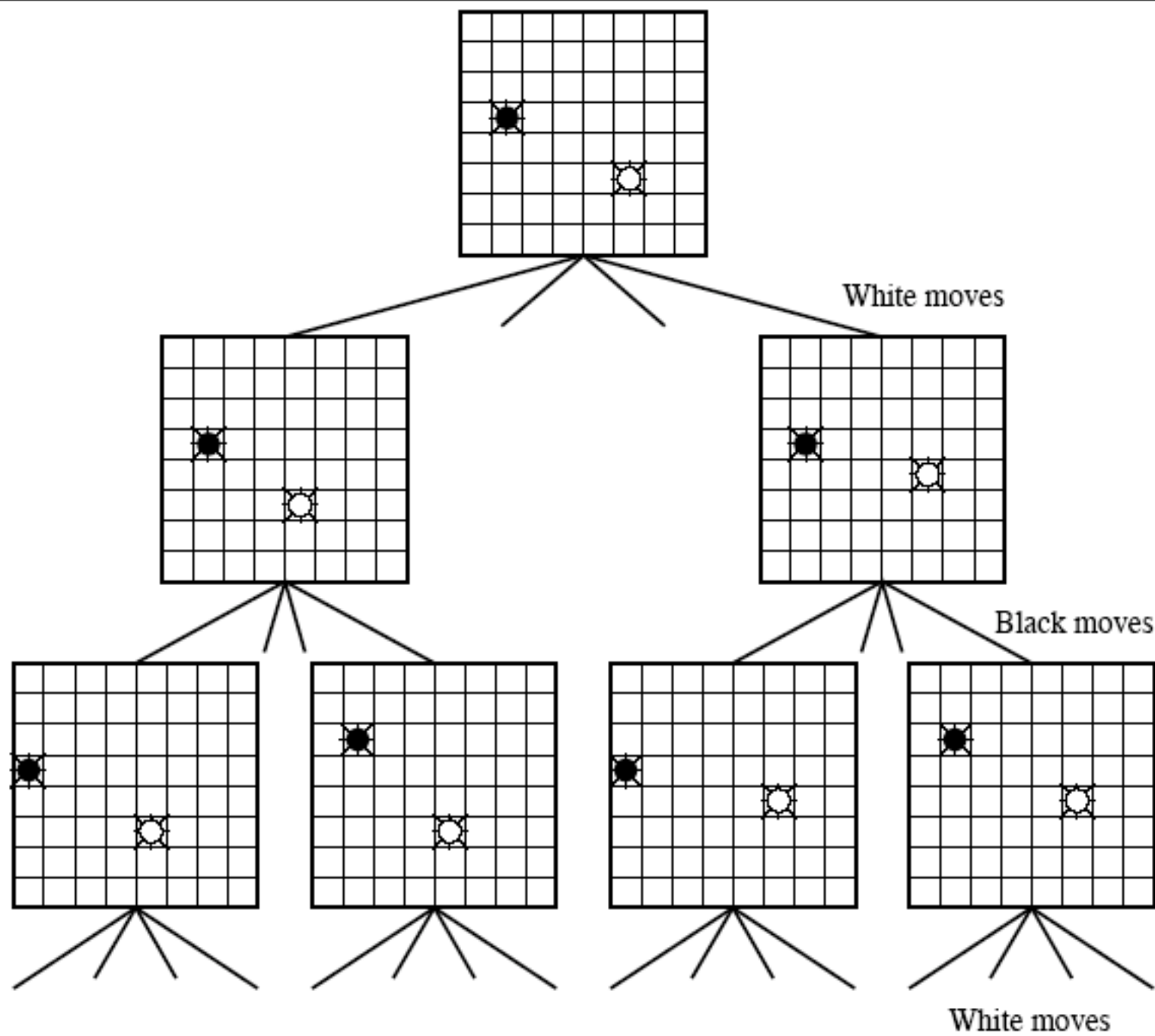
Consider a set up where there are two agents in the world:



© 1998 Morgan Kaufman Publishers

Assumption: Agents take turns when they move.

- One typical kind of scenario which fits this profile is a two-person game.
- This Grid-World Example could be moves in a chess endgame:
- Consider that White wants to be in the same cell as Black.
- Black wants to avoid this.
- **Agent Motivation:** What each agent wants is a move that guarantees success not matter what the other agent does.
- **Agent Point of View:** Usually all they can find is a move that improves things from their point of view.



© 1998 Morgan Kaufman Publishers

White wants to be in the same cell as Black.

Computers and Games

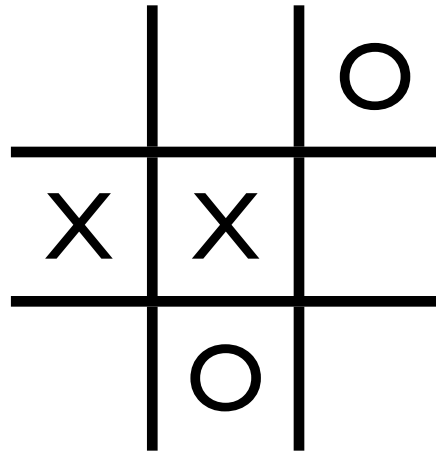
- This example is a *two person, perfect information, zero sum game*.
- **Perfect information:**
 - Both players know exactly what the state of the game is.
- **Zero sum:**
 - What is good for one player is bad for the other.
- This is also true of chess, draughts, go, othello, connect 4, ...

Two Person, Perfect Information, Zero-Sum

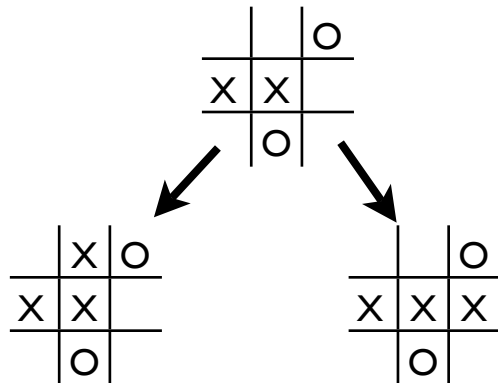
- These games are relatively easy to study at an introductory level.
- They have been studied just about as long as AI has been a subject.
- Some games are easily “solved”:
 - Tic-Tac-Toe
- Others have held out until recently.
 - Checkers (CHINOOK - Univ. of Alberta, Canada)
 - Chess (DEEP BLUE - IBM)
- Yet others are far from being mastered by computers.
 - Go
- Chance provides another complicating element.
 - Risk, Monopoly

State Space Representation

Games State spaces are iconic and have natural representations:



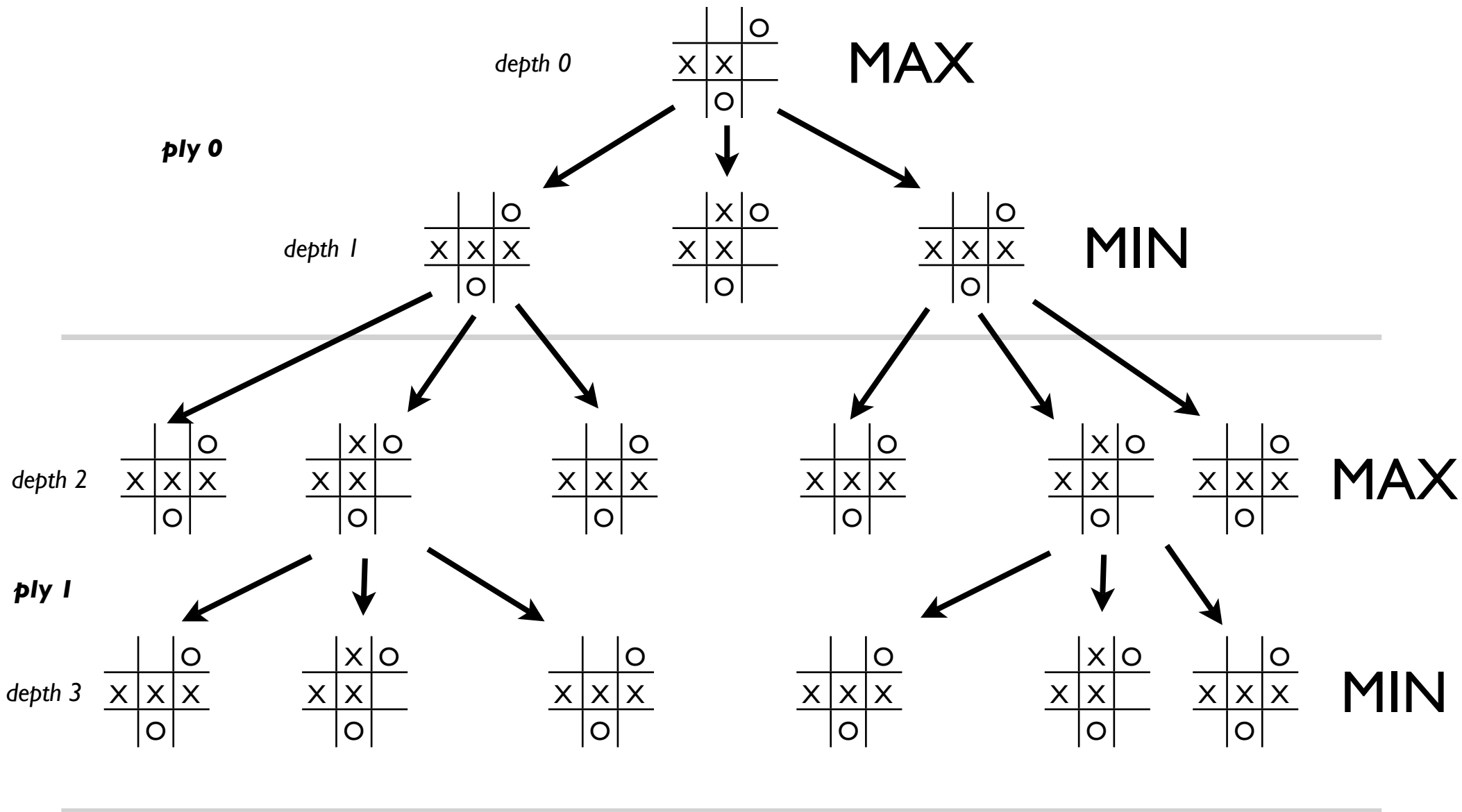
- State space operators are ***player's moves***.
- Search trees can be built much as before.



- However, we use different techniques to choose the optimal moves.

Minimax Procedure

- Typically we name the two players MAX and MIN.
- MAX moves first, and we want to find the best move for MAX.
- Since MAX moves first, even numbered layers are the ones where MAX gets to choose what move to make.
- The first node is on the zeroth layer.
- We call these “MAX nodes”.
- “Min nodes” are defined similarly.
- A *ply of ply-depth k* are the nodes at *depth $2k$ and $2k + 1$* .
- We usually estimate, in ply, the depth of the “lookahead” performed by both agents.



2 Ply Look-a-head

Search Scope

- We can't search the whole tree:
 - Chess: 10^{40} nodes
 - 10^{22} centuries to build search tree.
- So just search to a **limited horizon** (like depth-bounded).
- Then evaluate (using some heuristic) the leaf nodes.
- Then extract the best move at the top level.
- The Question now becomes: How do we do this (and how do we take into account the fact that MIN is also trying to win)?
- We use the minimax procedure.

MAX's Desires

- Assume our heuristic gives nodes high positive values if they are good for MAX
- And low values if they are good for MIN (can be negative even!).
- Now, look at the **leaf nodes** and consider which ones MAX wants:
 - Ones with high values.
- MAX could choose these nodes if it was his turn to play.
- So, the value of the MAX-node parent of a set of nodes is the *max of all the child values*.

MIN's Desires

- Similarly, when MIN plays she wants the node with the *lowest value*.
- So the MIN-node parent of a set of nodes gets the min of all their values.
- We back up values until we get to the children of the start node, and MAX can use this to decide which node to choose.
- There is an assumption (another!) which is that the evaluation function works as a better guide on nodes down the tree than on the direct successors of the start node.
- Let's look at a concrete example—Tic-Tac-Toe.

Evaluation Function for Crosses

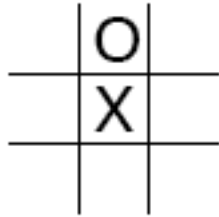
- Let MAX play crosses (X) and go first.
- Breadth-first search to depth 2.
- evaluation function (our heuristic) $e(p)$:

$$e(p) = \begin{cases} \infty & \text{if } p \text{ is a win for MAX} \\ -\infty & \text{if } p \text{ is a win for MIN} \\ val & \text{otherwise} \end{cases}$$

where

$$val = (\text{possible winning rows, columns, diagonals for MAX}) \\ - (\text{possible winning rows, columns, diagonals for MIN})$$

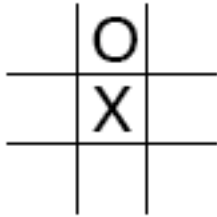
Evaluation



Scores?

© 1998 Morgan Kaufman Publishers

Evaluation

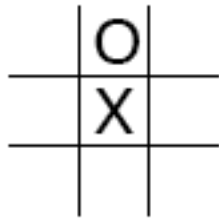


Scores?

$$6 - 4 = 2$$

© 1998 Morgan Kaufman Publishers

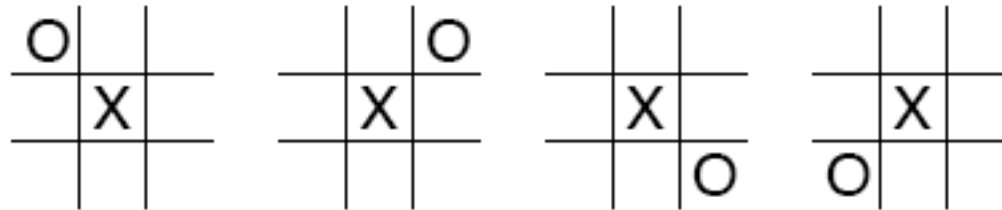
Evaluation



Scores? $6 - 4 = 2$

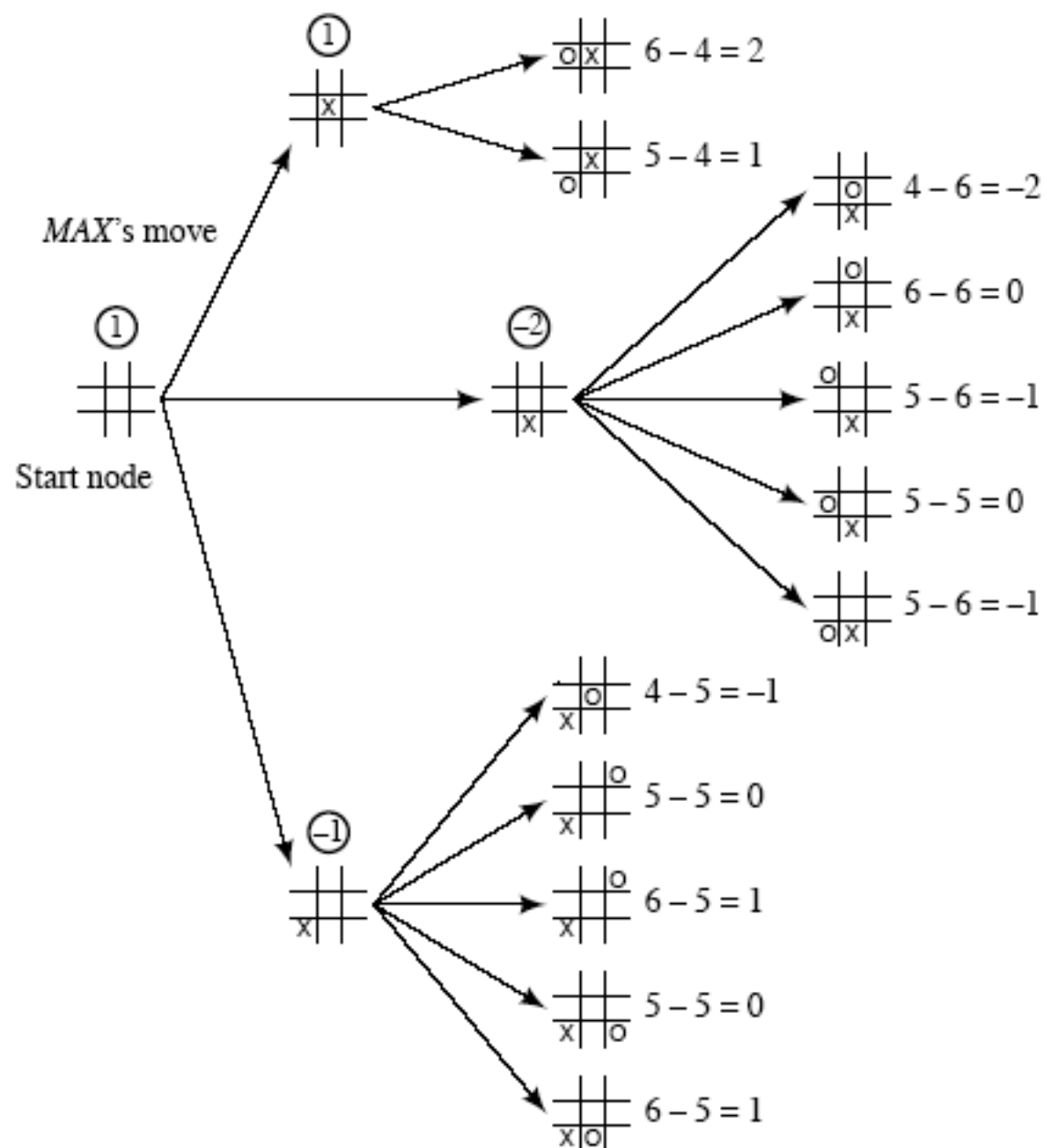
© 1998 Morgan Kaufman Publishers

- We also use symmetry to avoid having to generate loads of successor states, so are all equivalent:



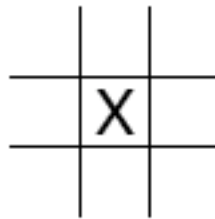
© 1998 Morgan Kaufman Publishers

- So, run the depth-2 search, evaluate, and back up values of the leaf nodes.



MAX's Best Move

- Unsurprisingly (for anyone who ever played Tic-Tac-Toe as a kid):

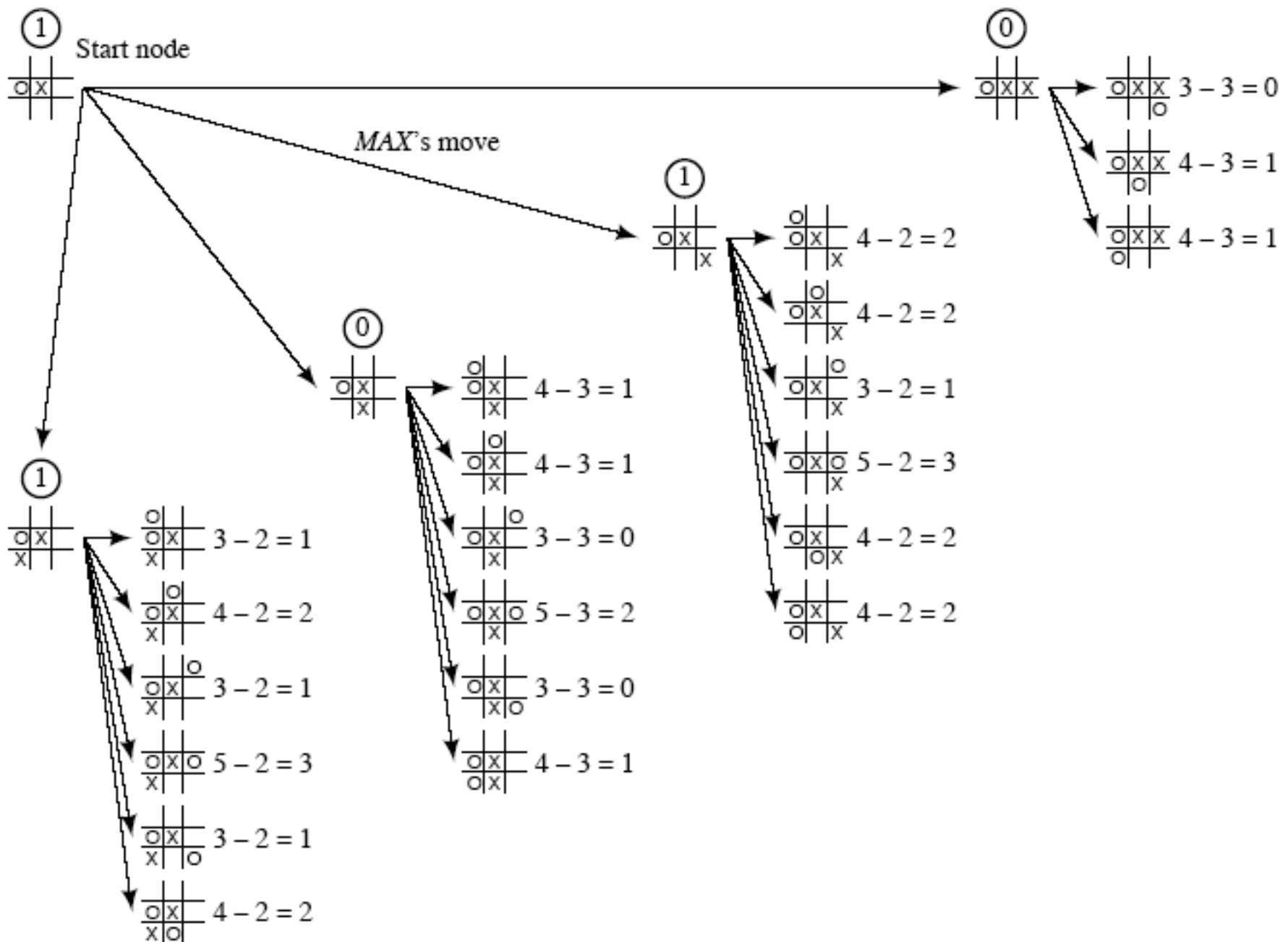


© 1998 Morgan Kaufman Publishers

Is the best move.

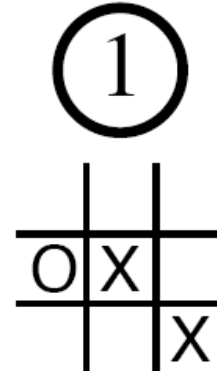
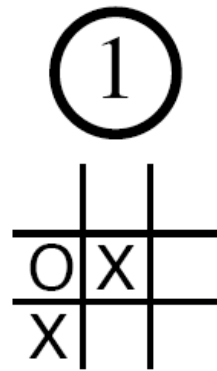
- So MAX moves and then MIN replies, and then MAX searches again:

MAX's Second Move

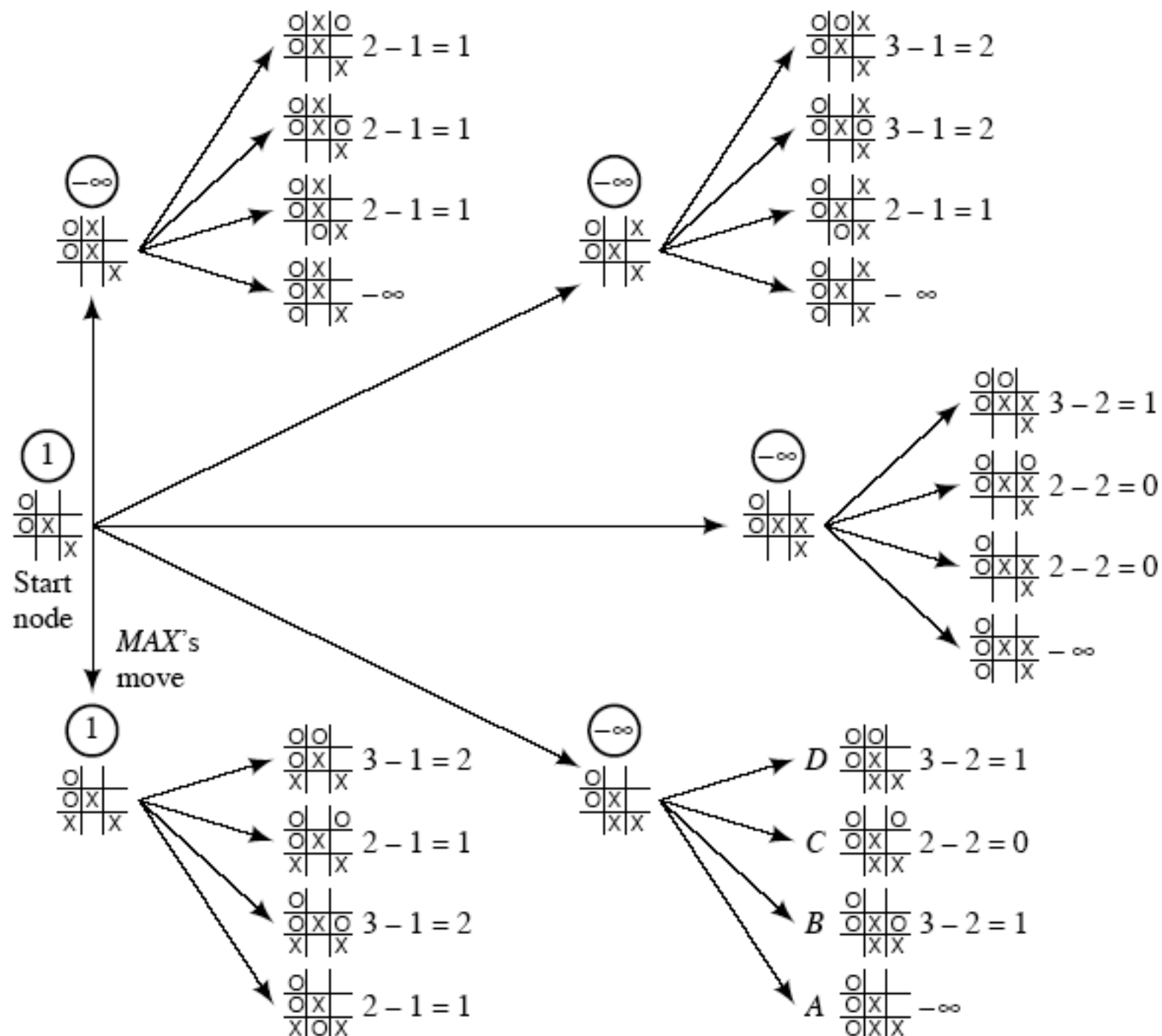


Second Move

- Here there are two equally good best moves.

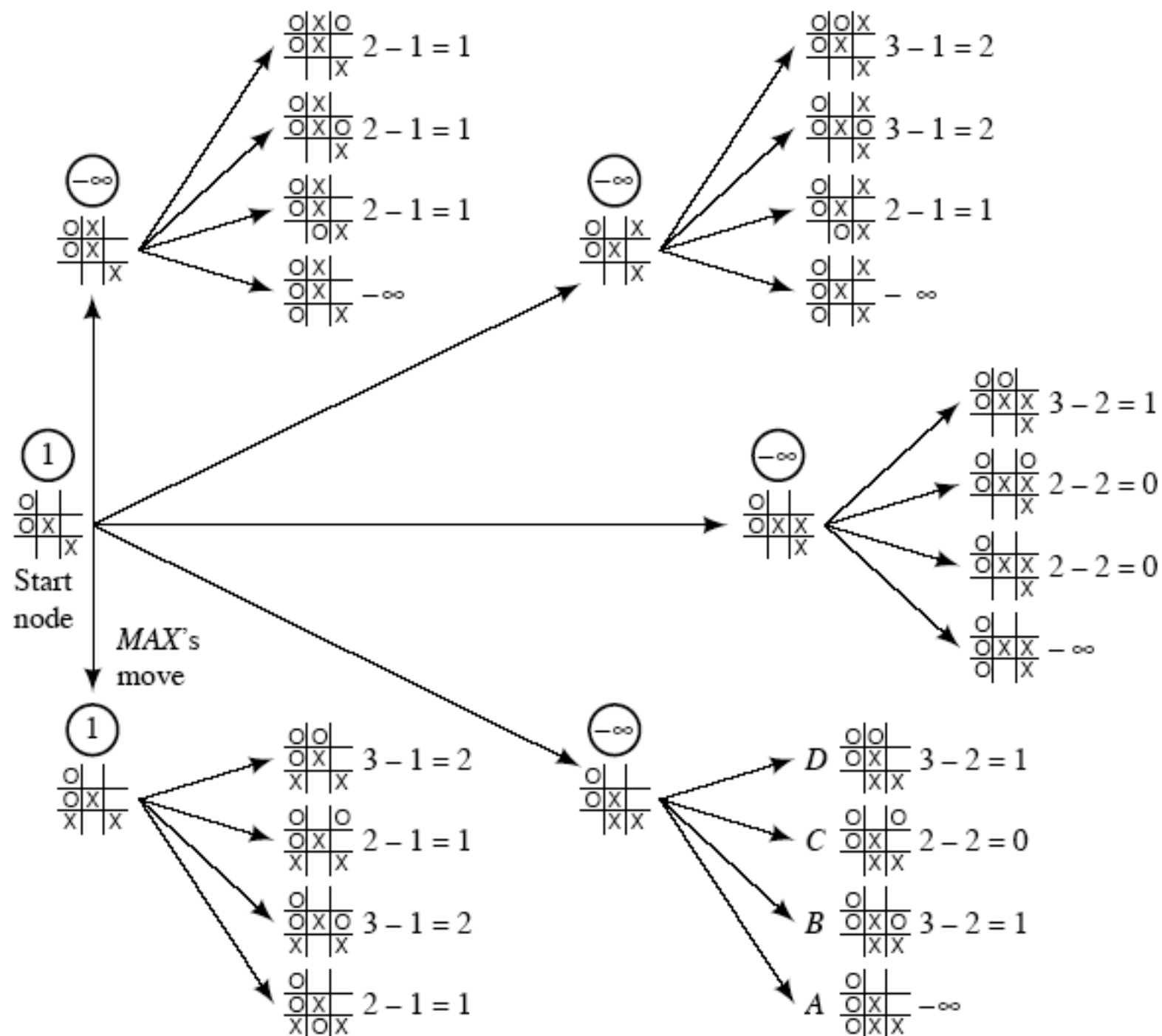


- So we can break the tie randomly.
- Then we let MIN move and do the search again.



Alpha-Beta search

- Minimax works very neatly, but it is inefficient.
- The inefficiency comes from the fact that we:
 - Build the tree FIRST,
 - THEN back up the values
- If we combine the two we get massive savings in computation.
- How do we manage this?
- Consider the last move in our MINIMAX example:



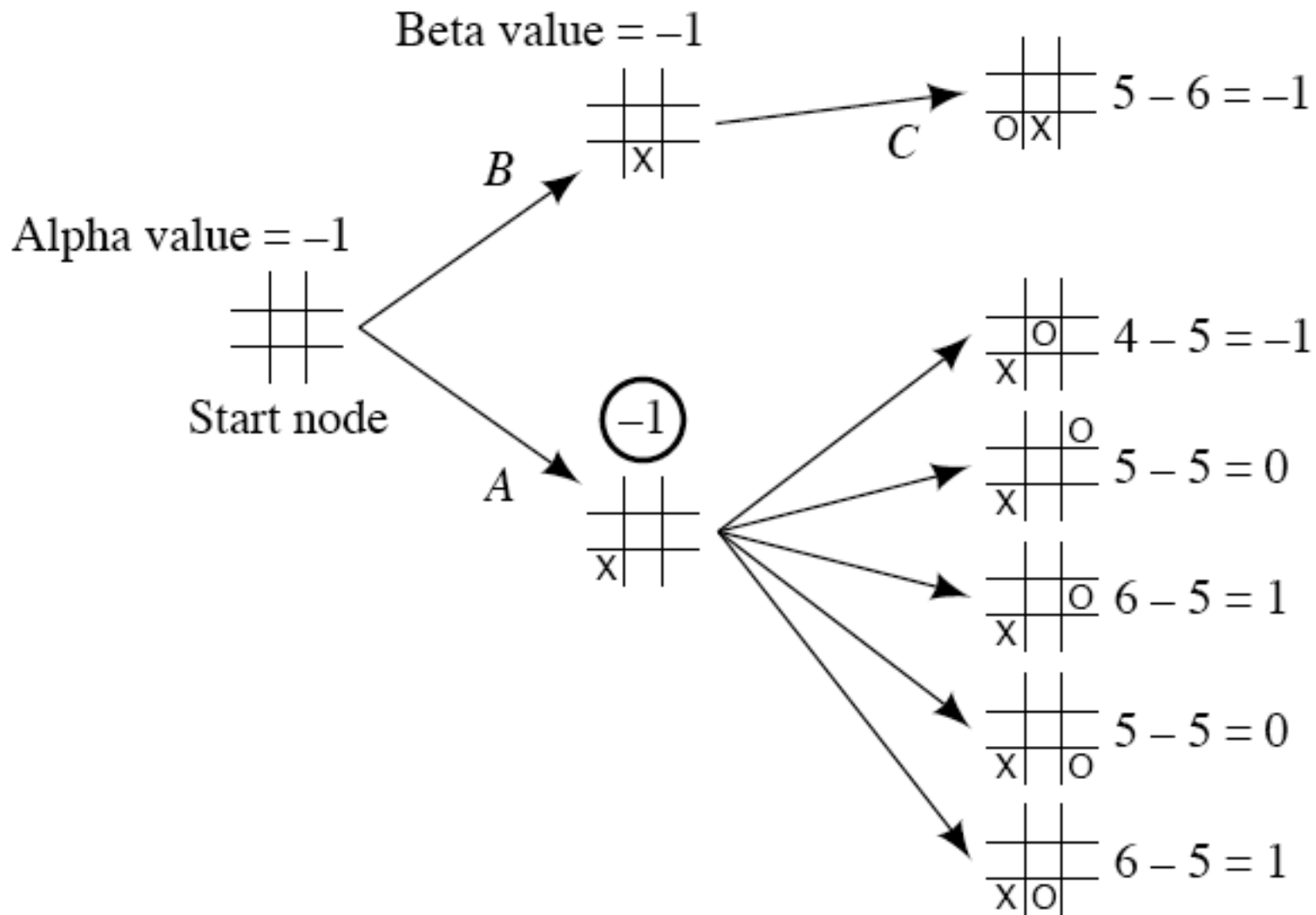
Cut-off at node A

- Well, when we get to node A, we don't have to expand any further.
 - Since we now that MIN will choose the winning game.
- So we save the evaluation of B, C and D.
- We also don't have to search any of the nodes below these nodes.
 - MAX will not find a better path/move below the MIN node..
- It also works when we don't have a winning move for MIN.
- Consider the following (earlier) stage of Tic-Tac-Toe.

Alpha-Beta Pruning

- Node A has backed-up value -1.
- Thus the start node cannot have a lower value than -1.
- This is the *alpha value*.
- Now let's go on to B and C.
- Since C has value -1, B cannot have a greater value than -1.
- This is the *beta value*.
- In this case, because B cannot ever be better than A, we can stop the expansion of B's children.

Earlier Stage in Crosses



- In general:
 - Alpha values are associated with MAX nodes and can never **decrease**.
 - Beta values are associated with MIN nodes and can never **increase**.
- Thus we can stop searching below:
 - Any MIN node with a beta value less than or equal to the alpha value of one of its MAX ancestors.
 - The backed up value of this MIN can be set to its beta value.
 - Any MAX node with an alpha value greater than or equal to any of its MIN node ancestors.
 - The backed up value of this MAX node can be set to its alpha value.

Alpha-Beta Values Computed

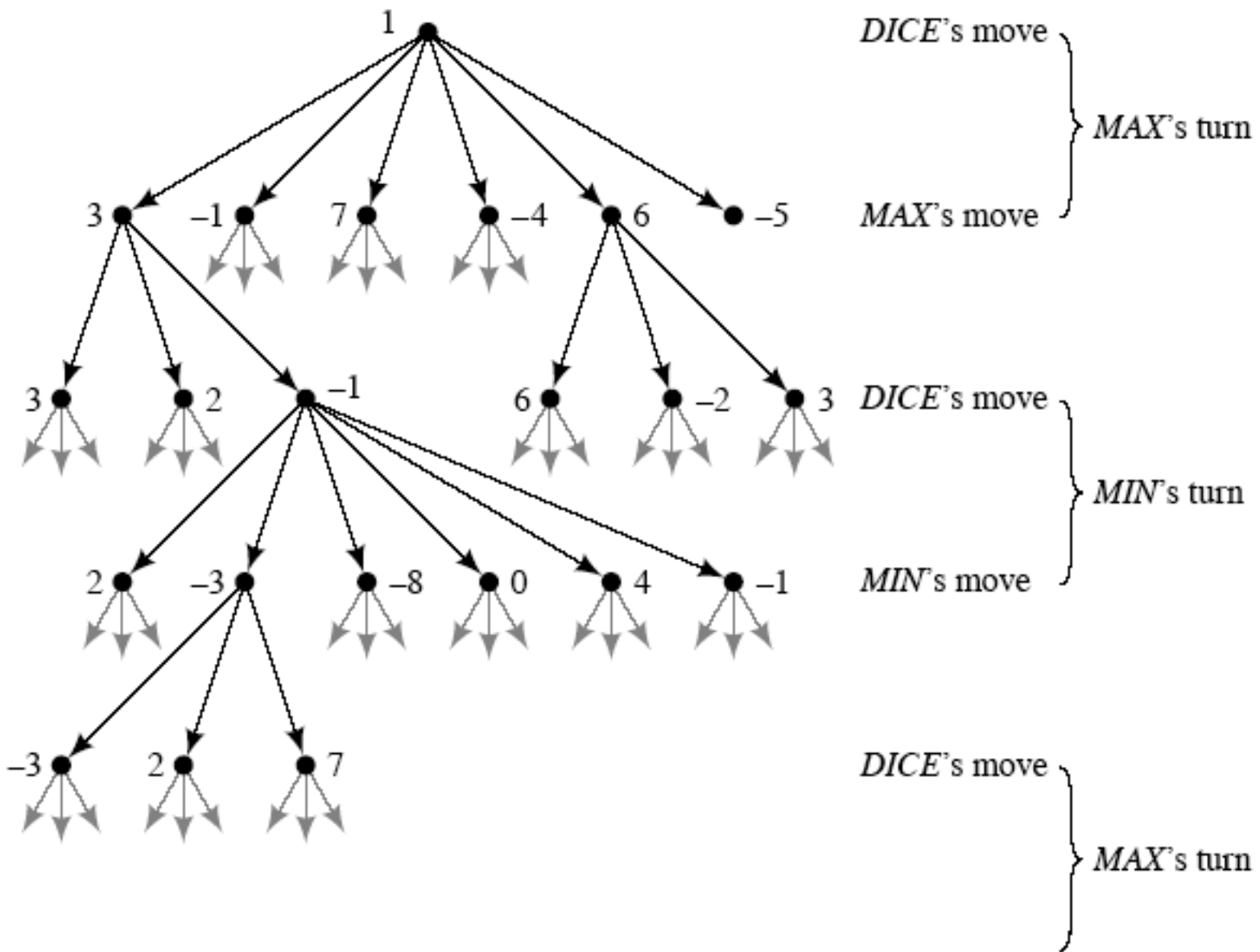
- We compute the values as:
 - *Alpha*: current **largest** final backed-up value of successors.
 - *Beta*: current **smallest** final backed-up value of successors.
- We keep searching until:
 1. we meet the “stop search” cut-off rules, or
 2. we have backed-up values for all the successors of the start node.
- Doing this always gives the **same best move** as full minimax.
- However, often (*usually*) this alpha-beta approach involves less searching.

Horizon Effects

- How do we know when to stop searching?
- What looks like a very good position for MAX might be a very bad position just over the horizon.
- Stop at *quiescent* nodes (value is the same as it would be if you looked ahead a couple of moves).
- Can be exploited by opponents; pushing moves back behind the horizon.
- A similar problem occurs because we assume that players always make their best move:
 - “Bad” moves can mislead a minimax-style player.

Games of chance

- How do we handle dice games?
- A neat trick is to model this as a another player DICE.
- We back up values in the usual way, **maximizing** for MAX and **minimizing** for MIN.
- For DICE moves, we back up the expected (weighted average) of the moves.
- For a single die, the weight is $1/6$.
- For more complex situations we use whatever probability distribution is indicated.



Summary

- We have looked at game playing as adversarial state-space search.
- Minimax search is the basic technique for finding the best move.
- Alpha/beta search gives greater efficiency.
- Games of chance can be handled by adding in the random player DICE.

Summary

- This lecture has looked at some techniques for refining the search space:
 - uniform cost search;
 - greedy search; and
 - A* search.
- When these work they explore just the relevant part of the search space.
- There are also techniques that go further than those we have studied.

(Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. 1984)

- There are three directions we will take from here:
 - Adversarial search
 - Learning the state space.
 - Adding in more knowledge about the domain.