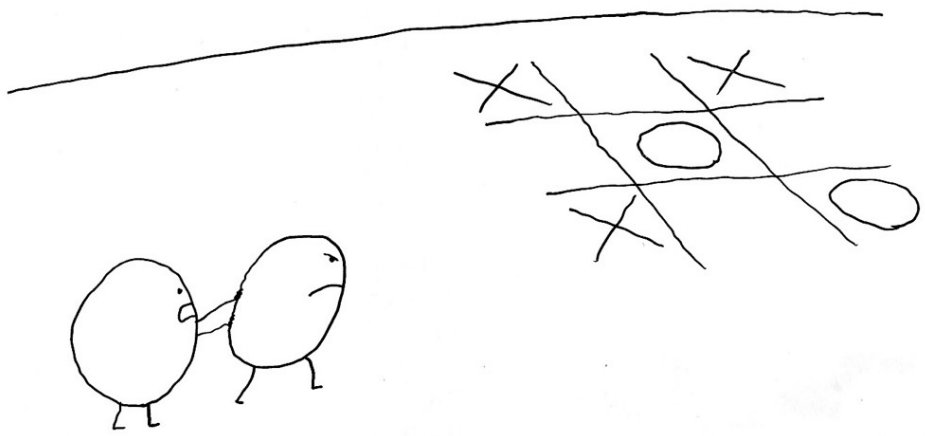
CS 4100: Introduction to AI

Wayne Snyder
Northeastern University

Lecture 9: A* Algorithm; Adversarial Search and Game Playing



"Larry! No!"

A*

The A* algorithm applies to graphs with weighted edges (non-negative) and a heuristic function h which estimates the distance of a node from the goal.

We use three different scores or values during the search:

g(N) = best known (exact) distance of N from start (sum of edge weights in best known path)

h(N) = heuristic estimate of remaining shortest distance to goal

f(N) = g(N) + h(N) = combination of exact and estimated distance of path through node N, increasingly more exact as approach goal; f(N) is used to order a priority queue controlling the search.

A* combines two approaches to search that we have already studied:

Breadth-First Search (single path version of Dijkstra's All Pairs Shortest Path algorithm):

f(N) = g(N)

Best-First Search:

f(N) = g(N)

A*: f(N) = g(N) + h(N)

where h is assumed to be admissible = never overestimates the remaining distance

A*

h(N) = a specified heuristic function

start, goal = specified nodes in the state space

Closed stores nodes for which a shortest path has been determined (may have to be updated)

Nodes are stored with the length g(N) of the path and back pointers for the paths.

Open stores nodes currently being explored

Open is a priority queue ordered by f(N)

A*

A* Algorithm Pseudo-Code

```
g(start) = 0          # so f(start) = h(start)
Open = [ (f(start), start)  ];      Closed = { (start, g(start), Null) }
while Open not empty:
    Current =  Open.pop()          # get minimal element from Open
    for each Child of Current:
        if Child not in Closed:           # never seen before
            g_child = g(Current) + cost(Current,Child);  f_child = g_child + h(Child)
            Add (g_child, Child, Current) to Closed            # record tentative shortest path
            Add (f_child, Child) to Open
        else if g_child < g(Child):                        # found a shorter path than the one previous stored in Closed
            Add (g_child, Child, Current) to Closed            # record better shortest path
        if Child == goal:
            Report success and reconstruct the solution path back to start
Report Failure
```

A*

A*

A*

A* Example

Games and Computer Science

A **GAME** is "a competitive activity involving skill, chance, or endurance on the part of two or more persons who play according to a set of rules, usually for their own amusement or for that of spectators" (Webster's)

Game Theory is a branch of mathematics that studies multi-player skill and chance games to understand economics, political science, networking, etc. (situations where entities compete for resources or rewards).

Video Game Programming is a combination of physical simulation, animation, video, audio, movie-making, etc. with some AI Game Programming.

AI Game Programming is a branch of Artificial Intelligence which attempts to simulate human behavior in a board game such as chess, checkers, backgammon, Othello, Go, Connect-4, etc. Games involving chance are more difficult....

AI Game Programming

We will study BOARD GAMES and investigate how to use TREE SEARCH to simulate how humans play such games.

Computers play board games by simulating how humans play: they search possible moves and predict where their opponent might move.

This is called Adversary Search

AI Game Programming

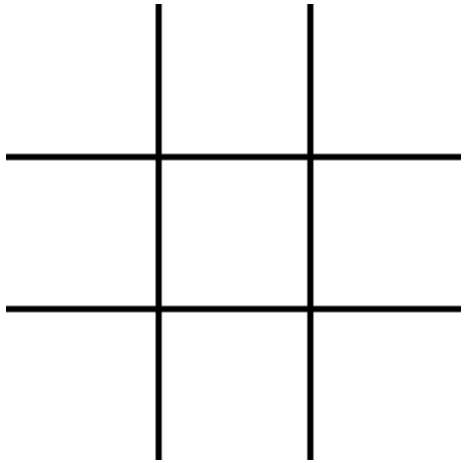
This approach works best on games in which

- There is a board where pieces are moved around following simple rules.
- There are two players (the computer is one player);
- Each player knows everything ("Perfect Information");
- There are no dice or other elements of random behavior;
- SO: The number of possible moves at any position is relatively limited, and the computer can search the tree of all possible moves.....

Adversary Search on Trees

How to describe the possible moves of such a game? Let's take the example of Tic Tac Toe:

Board:



Pieces: O X

Rules:

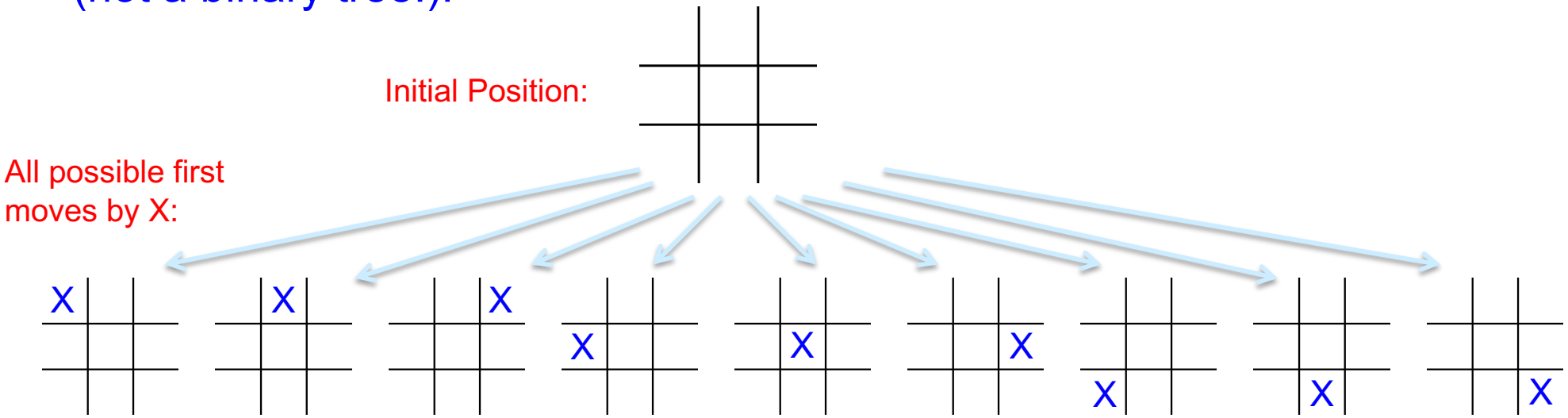
- Players alternate placing X's and O'x in the squares; X goes first;
- The first player to get three pieces in a row, column, or diagonal wins.

Possible representation for Tic Tac Toe board and pieces:

```
int [][] board = int[3][3];
int blank = -1;
int O = 0;
int X = 1;
```

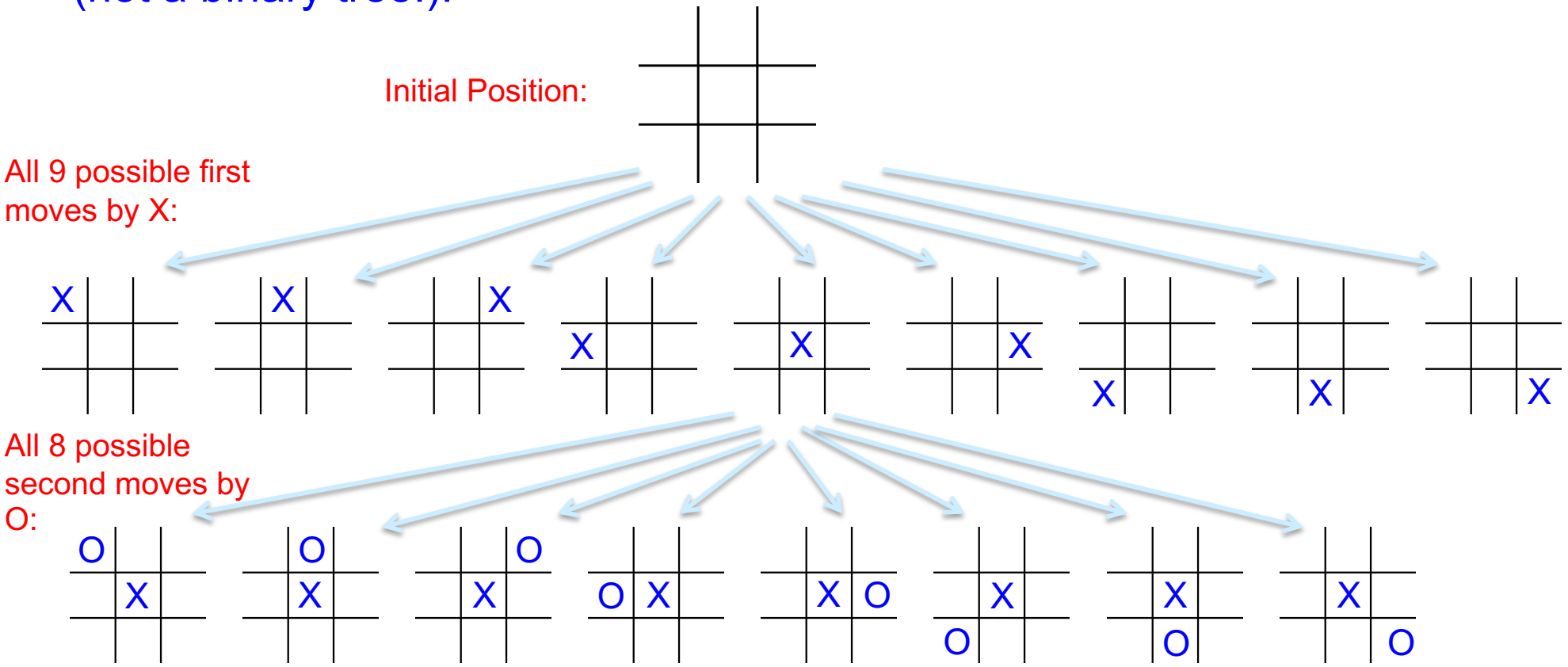
Adversary Search on Trees

The collection of all possible game positions can be described by a tree (not a binary tree!):



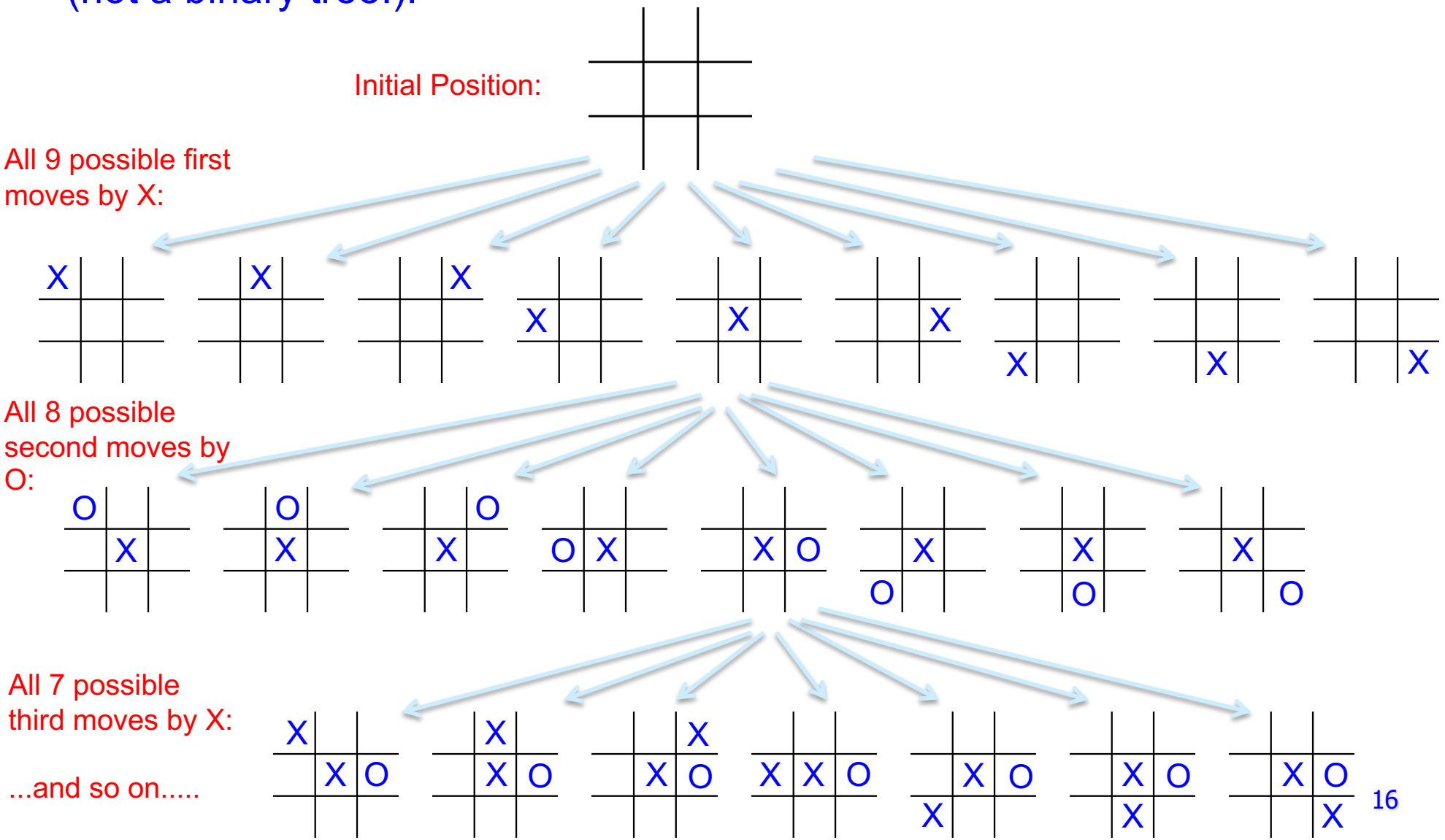
Adversary Search on Trees

The collection of all possible game positions can be described by a tree (not a binary tree!):



Adversary Search on Trees

The collection of all possible game positions can be described by a tree (not a binary tree!):



Adversary Search on Trees

You end up with a tree with 9! = 392,880 games/paths/leaves, one for each possible sequence of moves.

Ply	Branching Factor	Number of Nodes in this ply
0	9	1
1	8	9
2	7	72
3	6	504
4	5	3,024
5	4	15,120
6	3	60,480
7	2	181,440
8	1	362,880
9	0	362,880
Total:		986,410

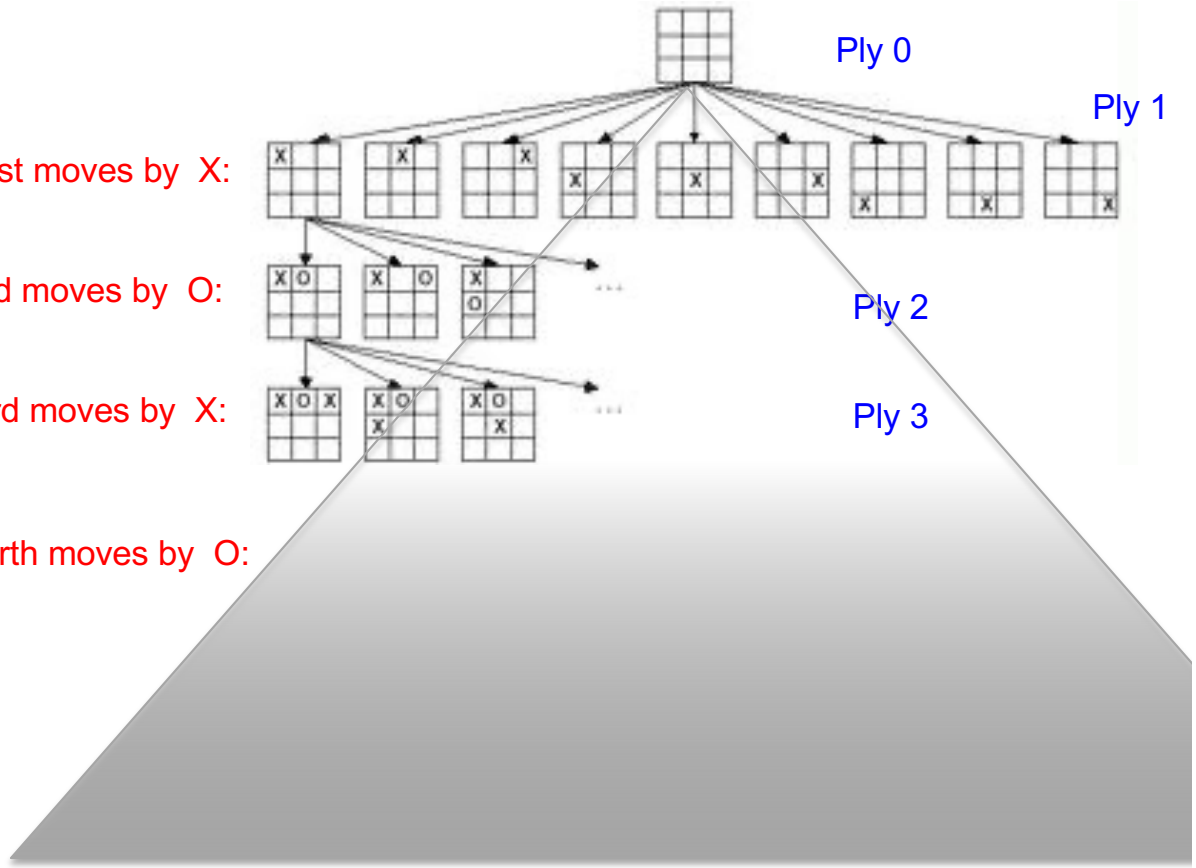
But there is a lot of duplication!

There are only 6045 distinct boards, and 126 distinct leaves.

We will talk about how to avoid such duplication when we discuss graphs. For now, we'll punt.....

Sometimes the levels are called "Plies" and numbered:

Initial Position:



Adversary Search on Trees

The number of board positions for the games we will consider is finite, but sometimes very large....

With 32 pieces and 64 squares, a generous upper bound on the number of possible chess boards is

$64^{33} = 2^{196} \sim 10^{60} =$

100000000000000000000000000000000000000000000000000000000000

(some chess blogs put the number closer to 10^{50}) and the paths in the tree are not finite (since you can move back to a previous position).

For Go, the number is apparently 10^{360}. (For comparison, there are about 10^{82} atoms in the visible universe.)

The most serious problem, however, is the "branching factor" and the resultant combinatorial explosion of nearby positions to test.....

Number of distinct chess positions

	uniquely realizable	all
ply 0	1	1
ply 1	20	20
ply 2	400	400
ply 3	1862	5362
ply 4	9825	72078
ply 5	53516	822518
ply 6	311642	9417681
ply 7	2018993	96400068
ply 8	12150635	988187354
ply 9	69284509	9183421888
ply 10	382383387	85375278064

Adversary Search on Trees

This is why we consider only two player games of “perfect information”—others are too difficult!

Every time you roll 2 dice, your game tree branches by 12;

If your game has N > 2 players,you would have to consider all possible outcomes for a sequence of N-1 moves between your own;

Without perfect information (e.g., cards), you would have to consider all possible hidden pieces of information.

Combinatorial explosion makes it impossible to explore very much of such search spaces.....

Let’s go back to Tic Tac Toe!

Adversary Search on Trees

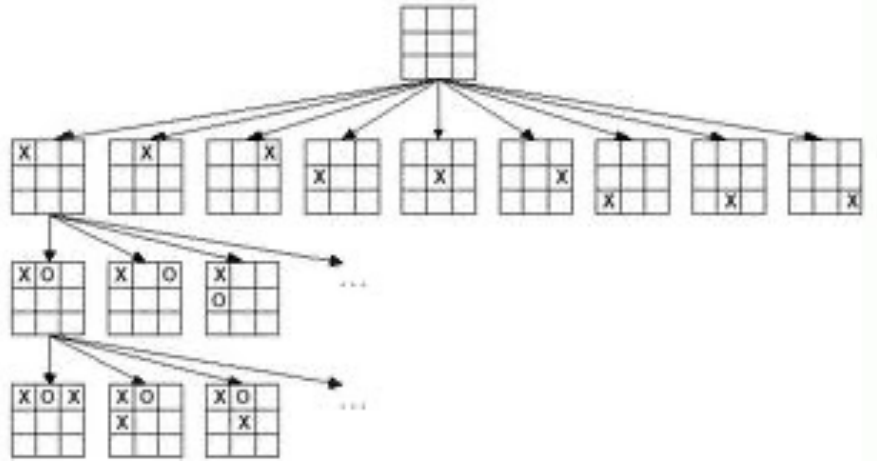
In order to search for the best move at a given board position, we have to know what we are looking for (the winning board positions) and when we are getting close to a win:

An Evaluation Function rates board positions:

Eval(board) => numeric score (how good is this board for me)

with an important assumption of symmetry:

What's good for me is bad for my opponent in equal measure:
Eval(B) for me is the same as -Eval(B) for my opponent.



Typically we use ints.....

A winning score is ∞

[-Integer.MIN_VALUE, ... 0 ..... ,Integer.MAX_VALUE]

A losing score is $-\infty$

Punchline: One global eval function is used: one player tries to maximize and one tries to minimize!

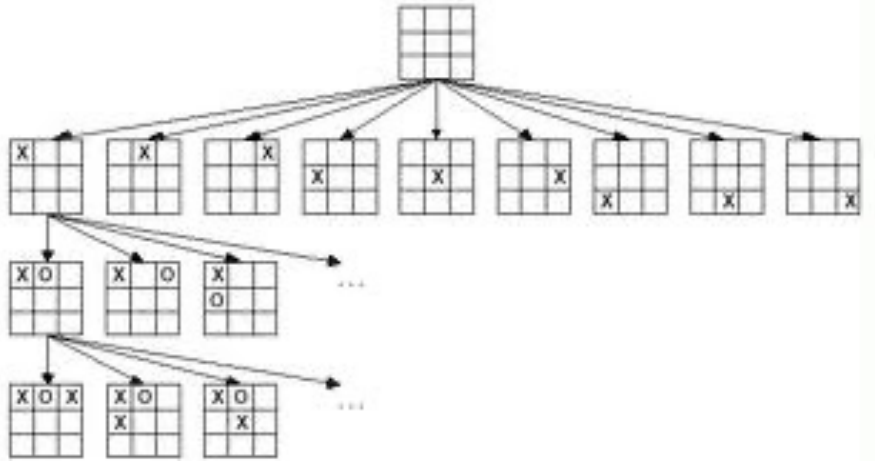
Adversary Search on Trees

Typically, we create an evaluation function by

Counting pieces, perhaps weighted by position or other characteristic (e.g., mobility, number of pieces it can attack) and

Counting patterns (parts of winning configurations), again perhaps weighted by importance.

NOTE: The Eval function must always be symmetric (win for me is loss or you), so typically you add the count for yourself, and subtract for your opponent. You want big positive score, and your opponent wants a big negative score.



Example: For Tic Tac Toe, we could count the number of possible rows, columns, and diagonals where we could possible move in the future to get a win; to make it symmetric, we subtract the similar number for our opponent:

X		
	X	O
	O	

Adversary Search on Trees

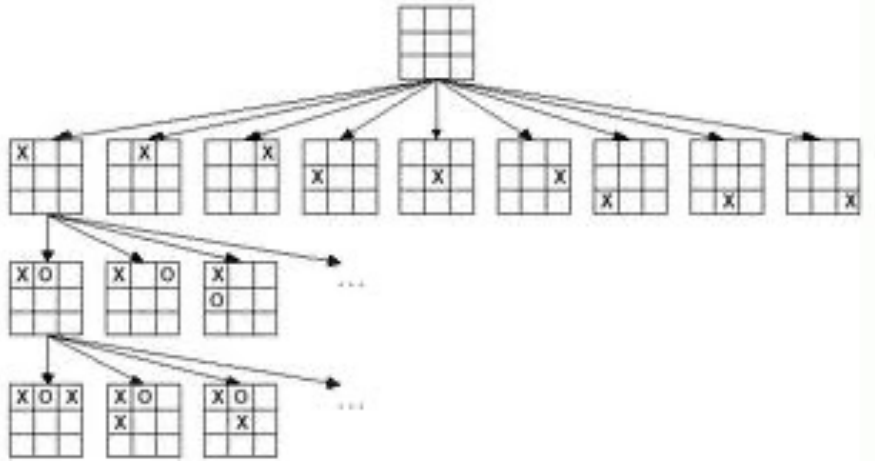
Typically, we create an evaluation function by

Counting pieces, perhaps weighted by position or other characteristic (e.g., mobility, number of pieces it can attack) and

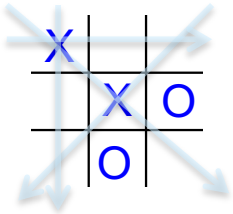
Counting patterns (parts of winning configurations), again perhaps weighted by importance.

NOTE: The Eval function must always be symmetric (win for me is loss or you), so typically you add the count for yourself, and subtract for your opponent. You want big positive score, and your opponent wants a big negative score.

Example: For Tic Tac Toe, we could count the number of possible rows, columns, and diagonals where we could possibly move in the future to get a win and weight it by the number of our pieces in the sequence; to make it symmetric, we subtract the similar number for our opponent:



1 + 1 + 1 + 2 (for X)



Adversary Search on Trees

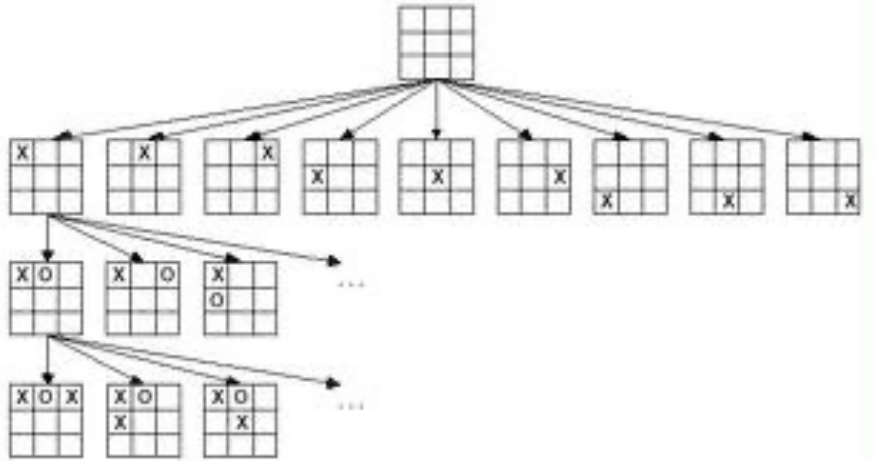
Typically, we create an evaluation function by

Counting pieces, perhaps weighted by position or other characteristic (e.g., mobility, number of pieces it can attack) and

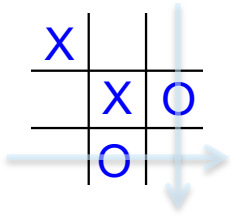
Counting patterns (parts of winning configurations), again perhaps weighted by importance.

NOTE: The Eval function must always be symmetric (win for me is loss or you), so typically you add the count for yourself, and subtract for your opponent. You want big positive score, and your opponent wants a big negative score.

Example: For Tic Tac Toe, we could count the number of possible rows, columns, and diagonals where we could possibly move in the future to get a win and weight it by the number of our pieces in the sequence; to make it symmetric, we subtract the similar number for our opponent:



5 (for X) – (1 + 1) (for O) = 3



Eval(B) = Infinity or –Infinity if there is a win!

Adversary Search on Trees

For chess, we could

Weight each piece by kind (pawn = 1, rook = 10, etc.) and by mobility;

Look for good and bad patterns:

- Many of our pieces in the middle of the board(good);
- Pieces which can attack other pieces (good); Etc.

Add your sum and subtract your opponent's.

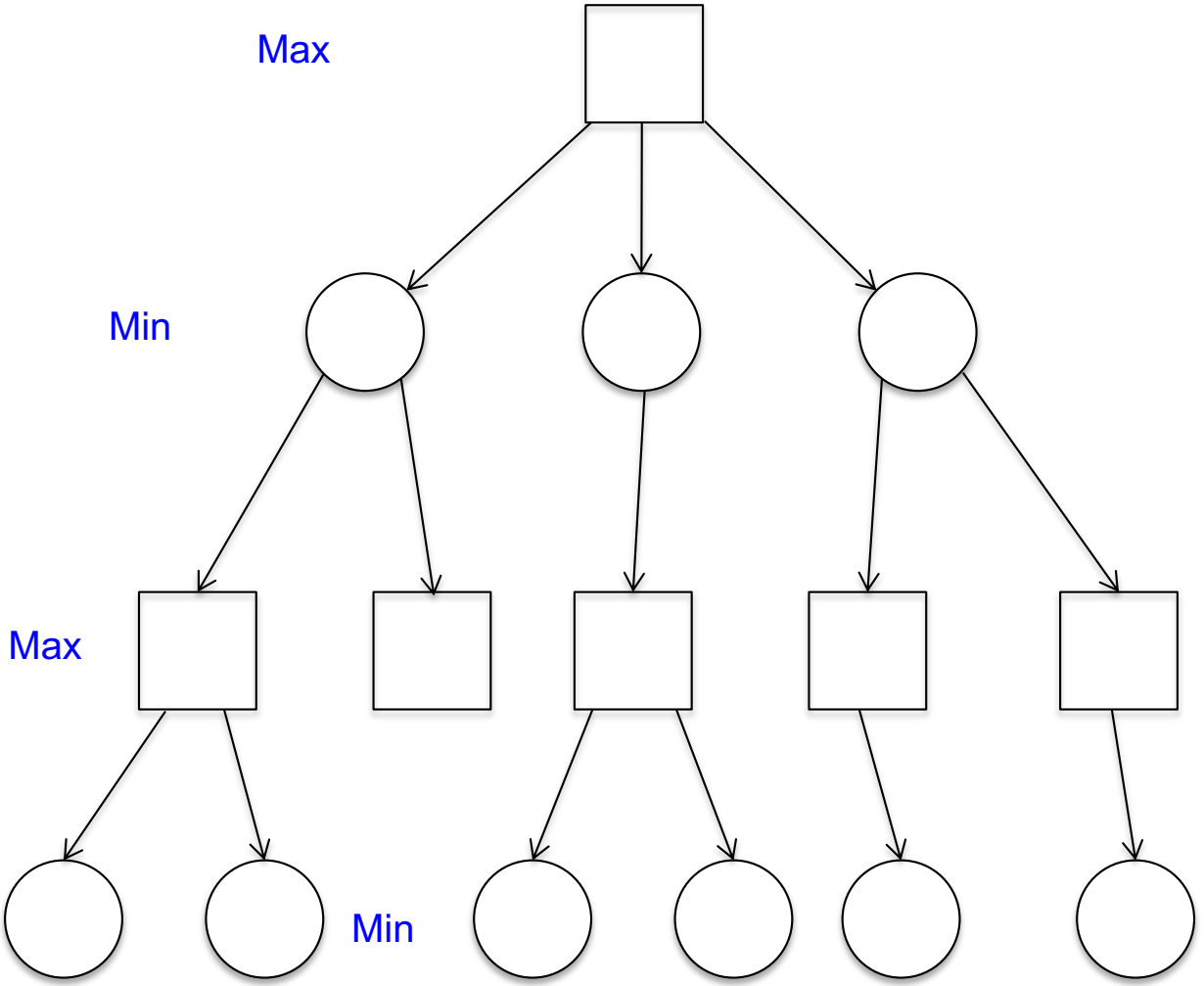
Min Max Trees

The symmetry of the game creates alternating layers in the tree, with

Max Levels (□) – It's the max player's move, and he wants the highest score; and

Min Levels(○) – It's the min player's move, and he wants the lowest score.

It does not matter who is Max and who is Min, but let's assume that the computer is always Max; the tree thus represents what the program sees when considering its next move…



Min Max Trees

The Algorithm:

(1) Generate the tree (more on this later!);

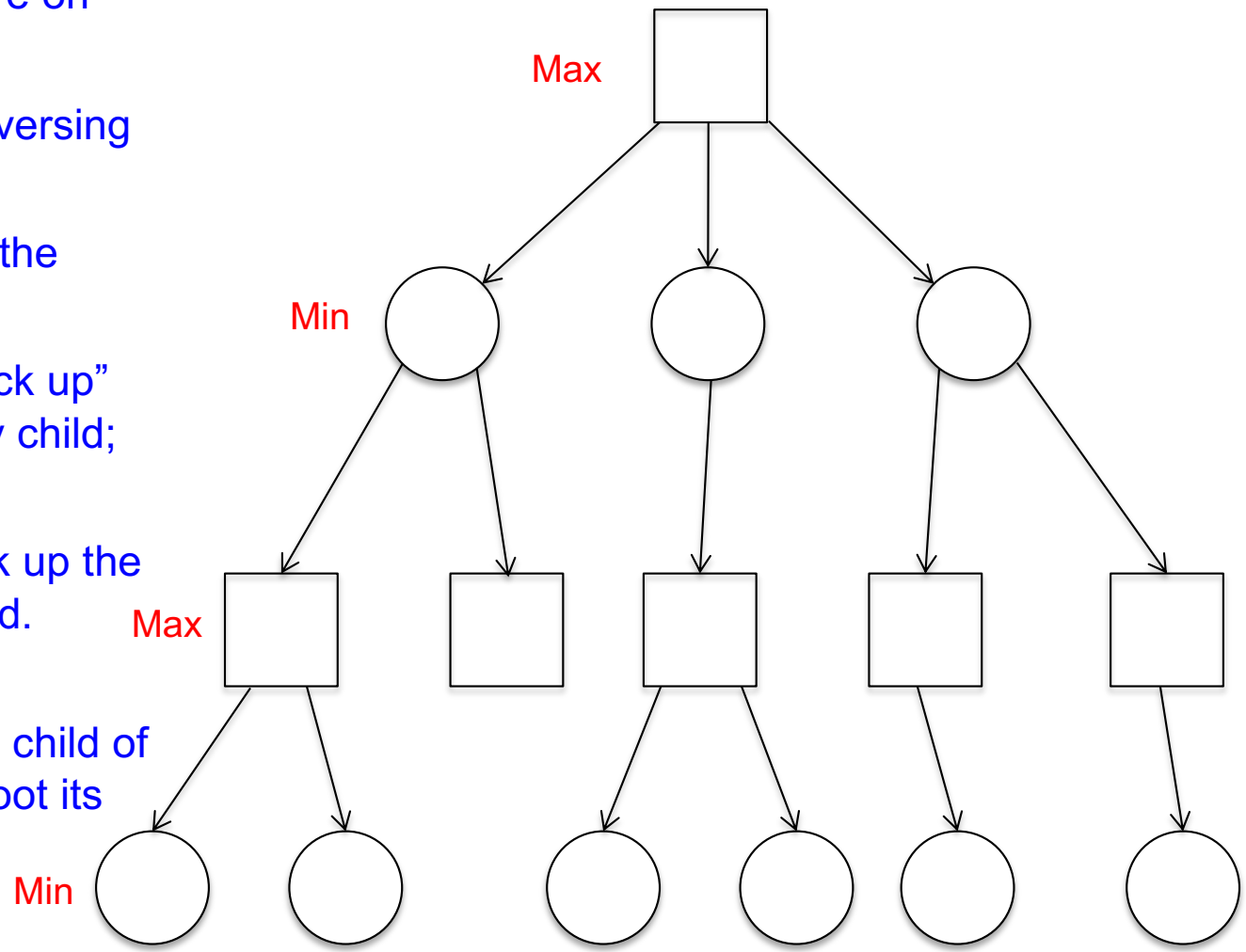
(2) Label the nodes by traversing the tree post-order and:

(A) At leaf nodes, use the eval() function;

(B) At Max nodes, “back up” the maximum value of any child; and

(C) At Min nodes, back up the minimum value of any child.

(3) Choose the move that corresponds to the largest child of the root (which gave the root its value).



Min Max Trees

The Algorithm:

(1) Generate the tree (more on this later!);

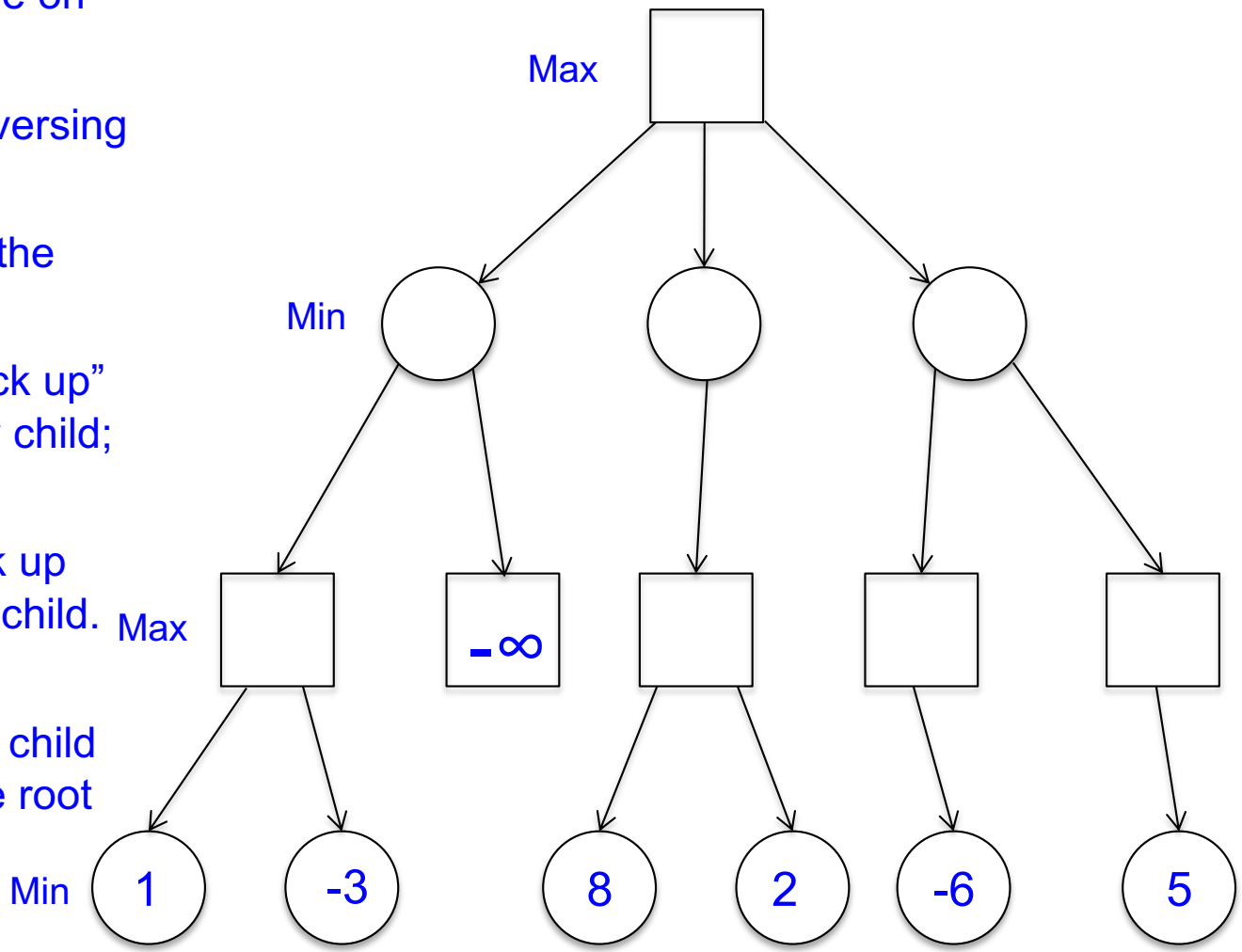
(2) Label the nodes by traversing the tree post-order and:

(A) At leaf nodes, use the eval() function;

(B) At Max nodes, "back up" the maximum value of any child; and

(C) At Min nodes, back up the minimum value of any child.

(3) Choose the move that corresponds to the largest child of the root (which gave the root its value).



Min Max Trees

The Algorithm:

(1) Generate the tree (more on this later!);

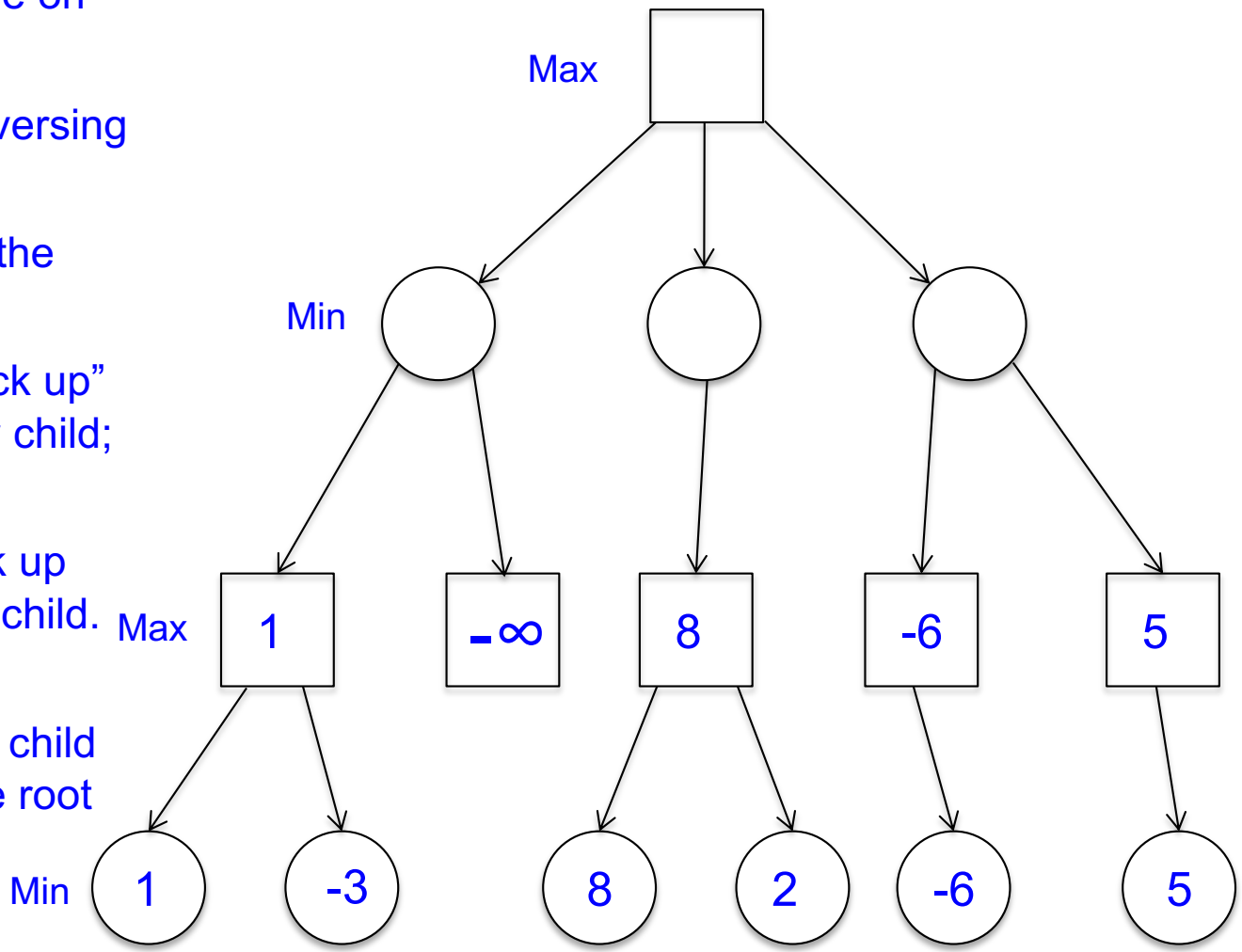
(2) Label the nodes by traversing the tree post-order and:

(A) At leaf nodes, use the eval() function;

(B) At Max nodes, “back up” the maximum value of any child; and

(C) At Min nodes, back up the minimum value of any child.

(3) Choose the move that corresponds to the largest child of the root (which gave the root its value).



Min Max Trees

The Algorithm:

(1) Generate the tree (more on this later!);

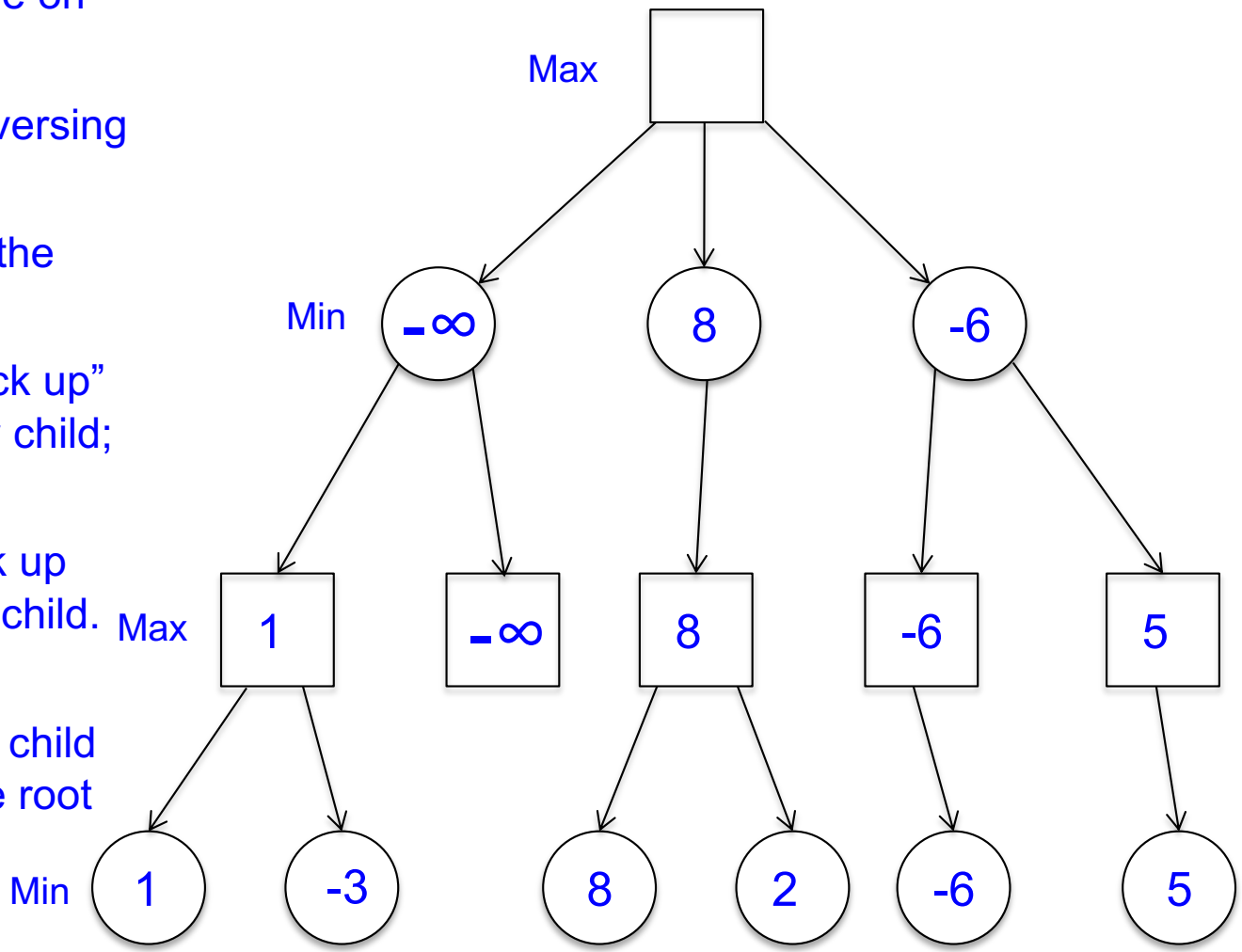
(2) Label the nodes by traversing the tree post-order and:

(A) At leaf nodes, use the eval() function;

(B) At Max nodes, “back up” the maximum value of any child; and

(C) At Min nodes, back up the minimum value of any child.

(3) Choose the move that corresponds to the largest child of the root (which gave the root its value).



Min Max Trees

The Algorithm:

(1) Generate the tree (more on this later!);

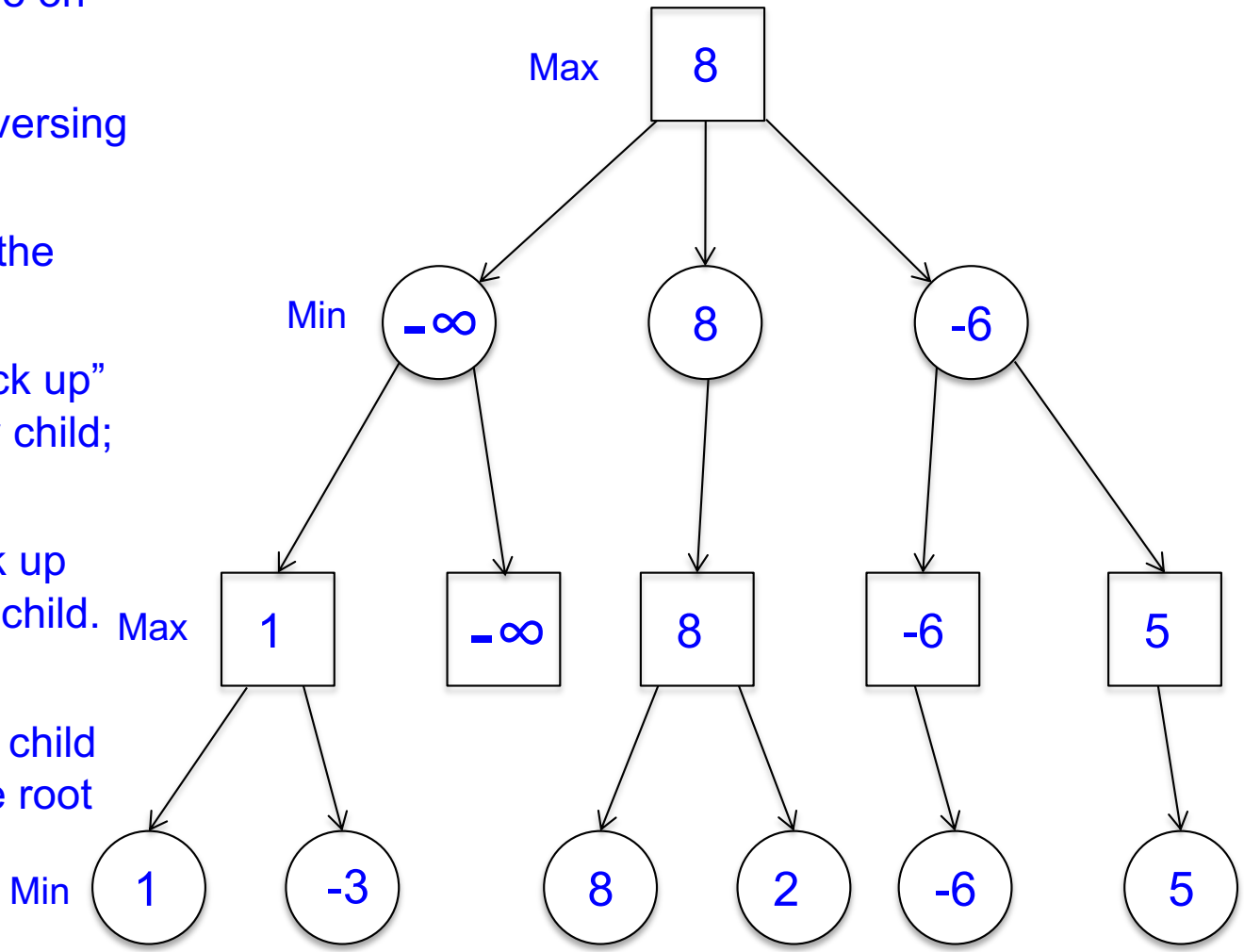
(2) Label the nodes by traversing the tree post-order and:

(A) At leaf nodes, use the eval() function;

(B) At Max nodes, "back up" the maximum value of any child; and

(C) At Min nodes, back up the minimum value of any child.

(3) Choose the move that corresponds to the largest child of the root (which gave the root its value).



Min Max Trees

The Algorithm:

(1) Generate the tree (more on this later!);

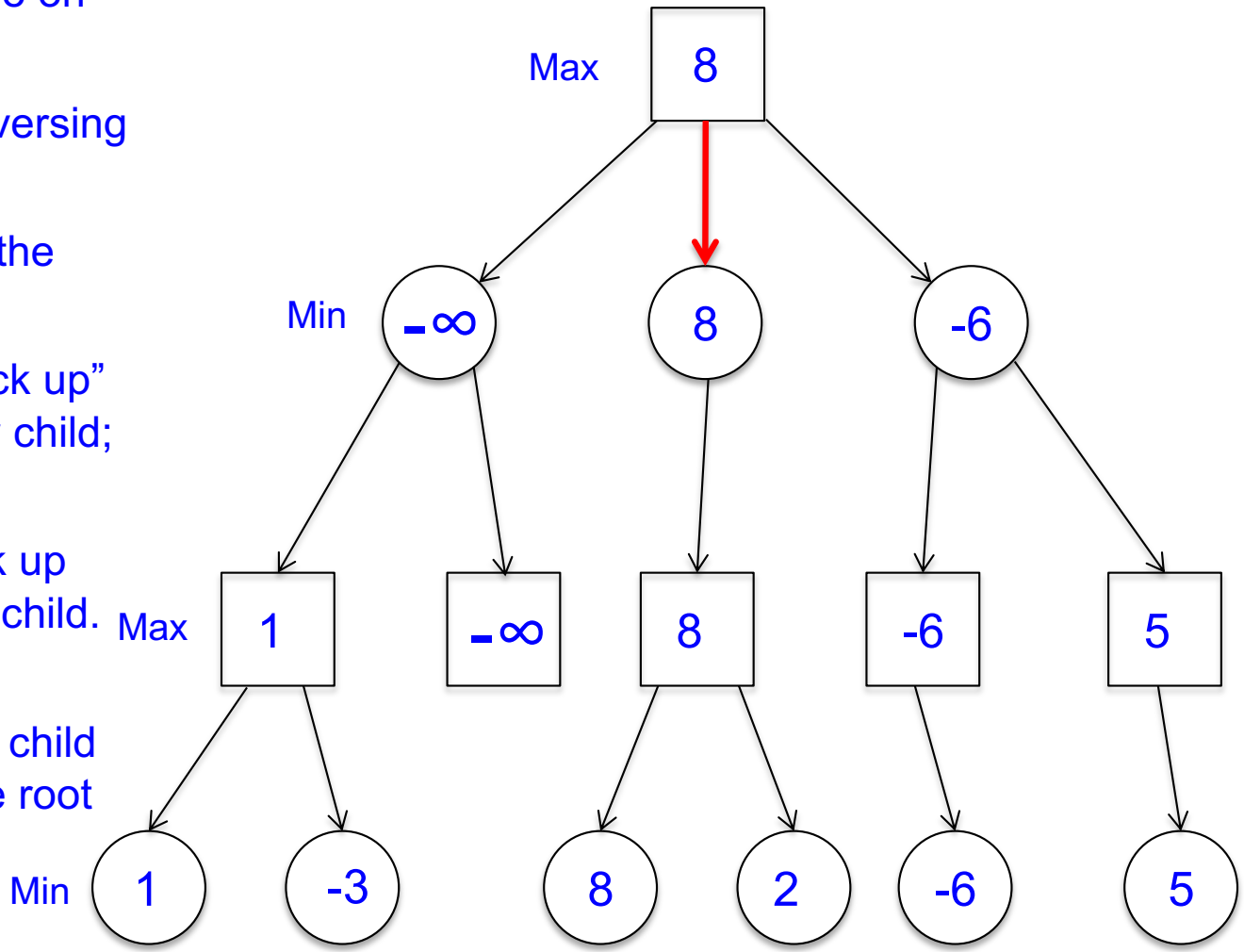
(2) Label the nodes by traversing the tree post-order and:

(A) At leaf nodes, use the eval() function;

(B) At Max nodes, "back up" the maximum value of any child; and

(C) At Min nodes, back up the minimum value of any child.

(3) Choose the move that corresponds to the largest child of the root (which gave the root its value).



Min Max Trees

The Algorithm:

(1) Generate the tree (more on this later!);

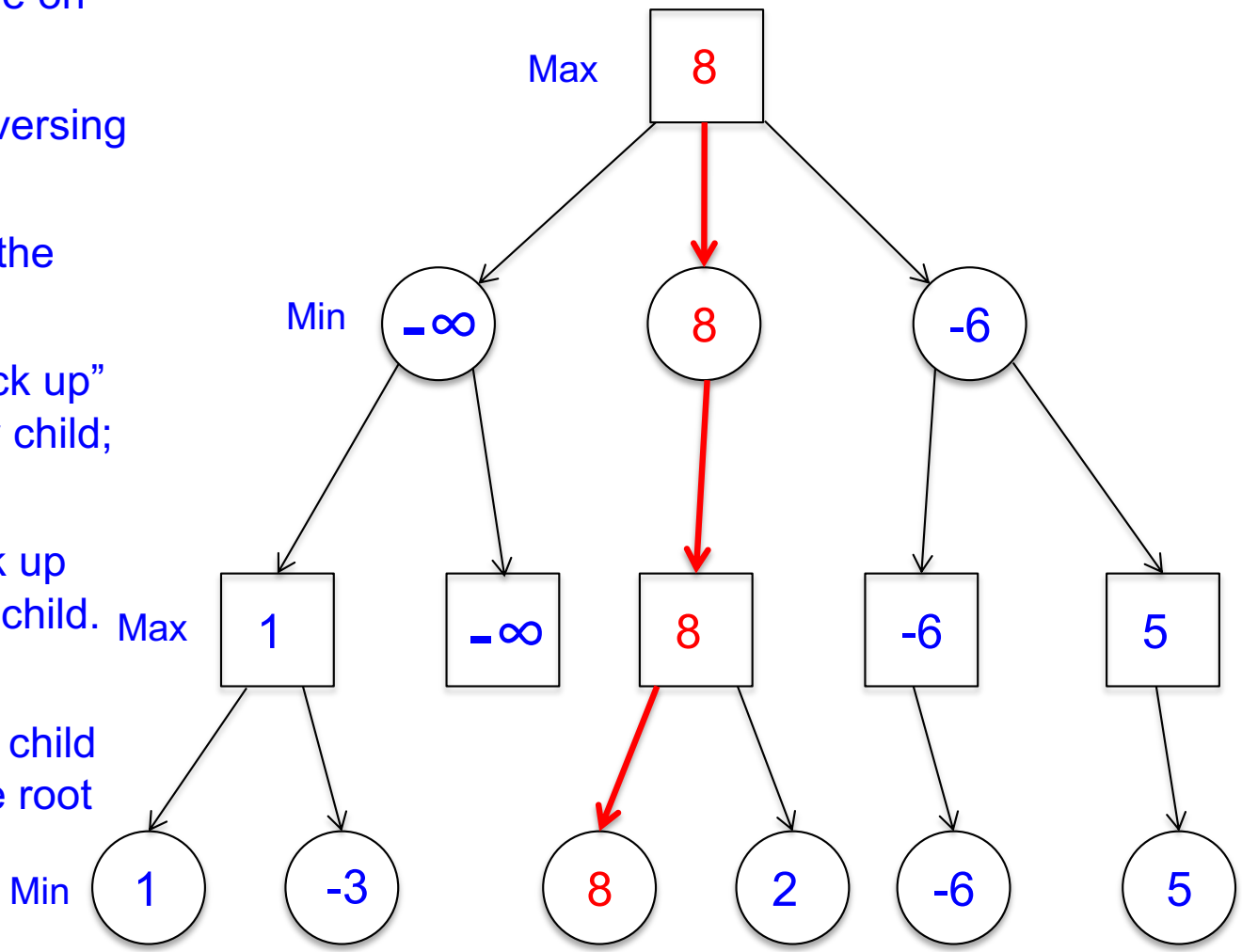
(2) Label the nodes by traversing the tree post-order and:

(A) At leaf nodes, use the eval() function;

(B) At Max nodes, "back up" the maximum value of any child; and

(C) At Min nodes, back up the minimum value of any child.

(3) Choose the move that corresponds to the largest child of the root (which gave the root its value).



Note that all values come from the leaves, and the root's value comes from the best sequence of moves from the Max point of view

Min-Max Trees

Big Question: How to generate the tree?

Simplest Answer: Generate tree down to some fixed depth D:

```
Move chooseMove(Node t) {
     int max = -Inf;      Move best;
     for(each move m to a child c of t) {
        int val = minMax( c, 1 );
        if(val > max) {
           best = m; max = val;
        }
     }
     return best;
}

int minMax(Node t, int depth) {
  if( t is a leaf node || depth == D)  // leaf node could be
     return eval(t);                   // because eval(t) = ± Inf
  else if( t is max node ) {
     int val = -Inf;
     for(each child c of t) {
       val = max(val, minMax( c, depth+1 ) );
     }
     return val;
  } else {                          // is a min node
     int val = Inf;
     for(each child c of t)
       val = min(val, minMax( c, depth+1 ) );
     return val;
  }
}
```