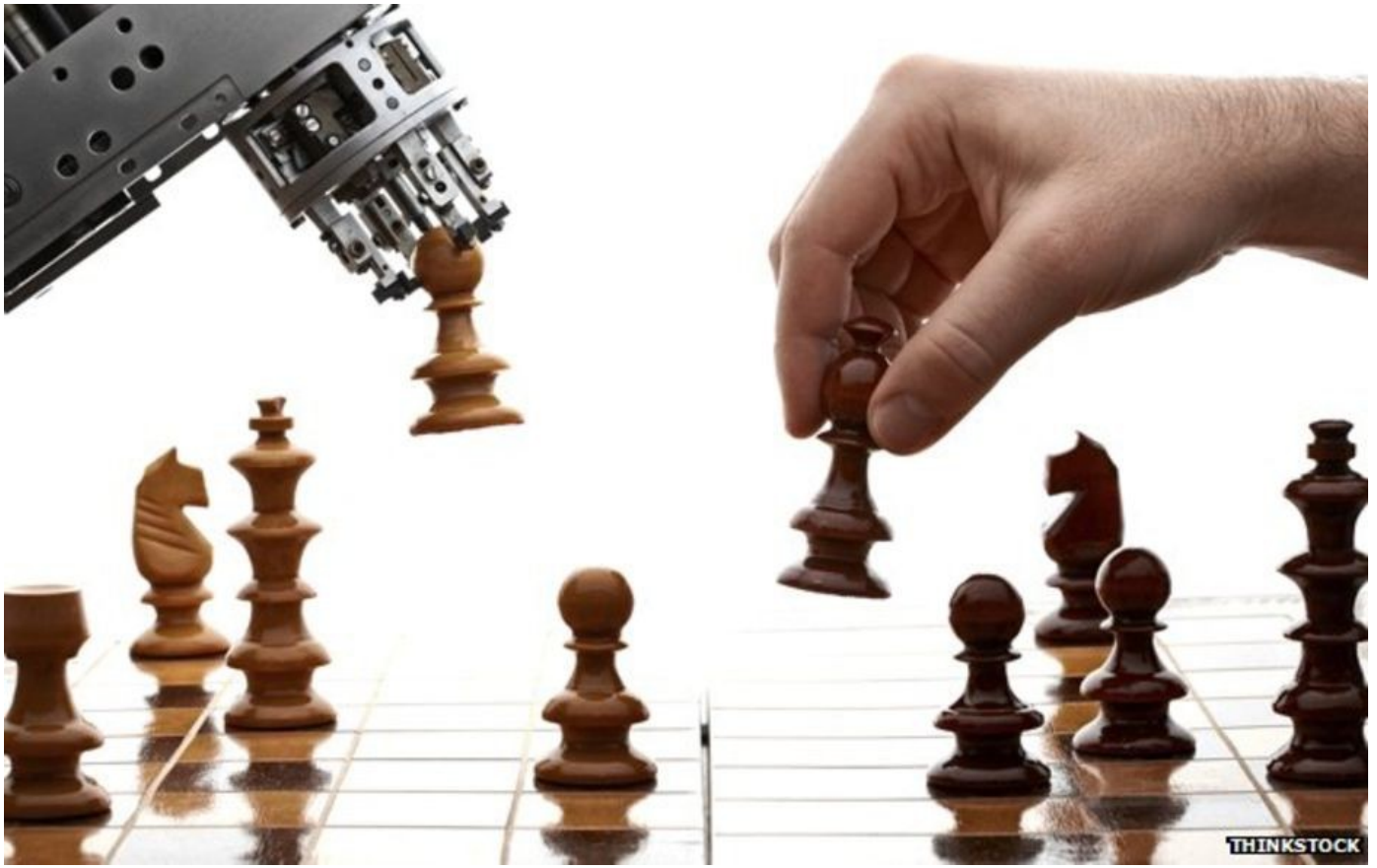
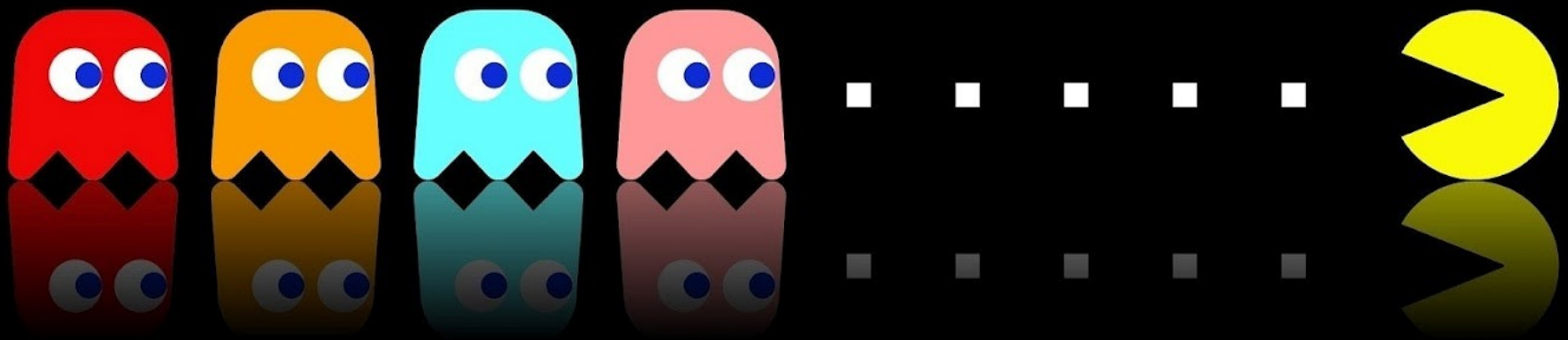


Introduction to Artificial Intelligence

COSC 4550 / COSC 5550

Professor Cheney

9/18/17




Wyoming Global
Technology Summit

adversarial search
(i.e. game playing!)

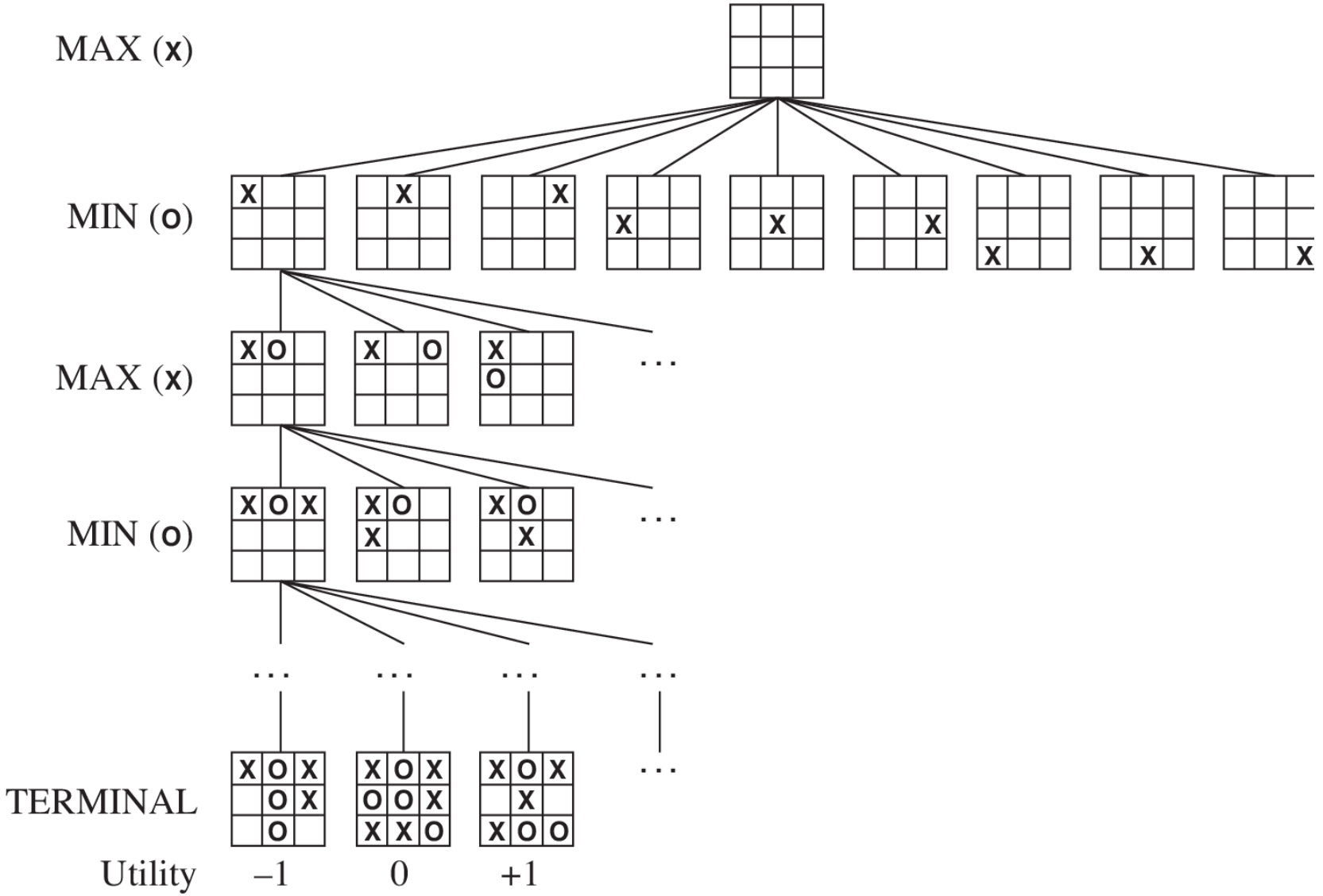
we will focus largely on:

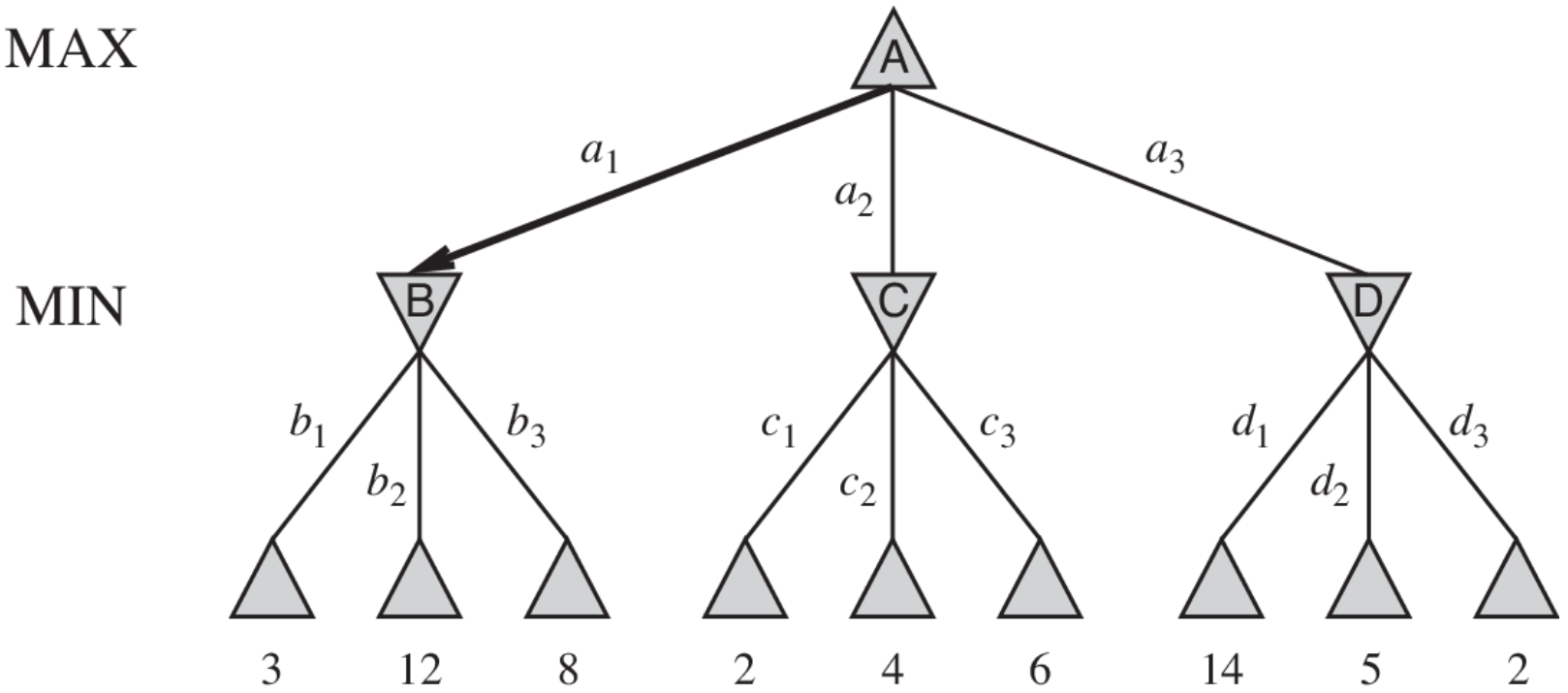
two-player

zero-sum games

with perfect information

(but game theory includes much more than this)


MAX (x)
MIN (o)
MAX (x)
MIN (o)
TERMINAL
Utility
–1
0
+1



▲ ~ max node (try to increase final score/utility)

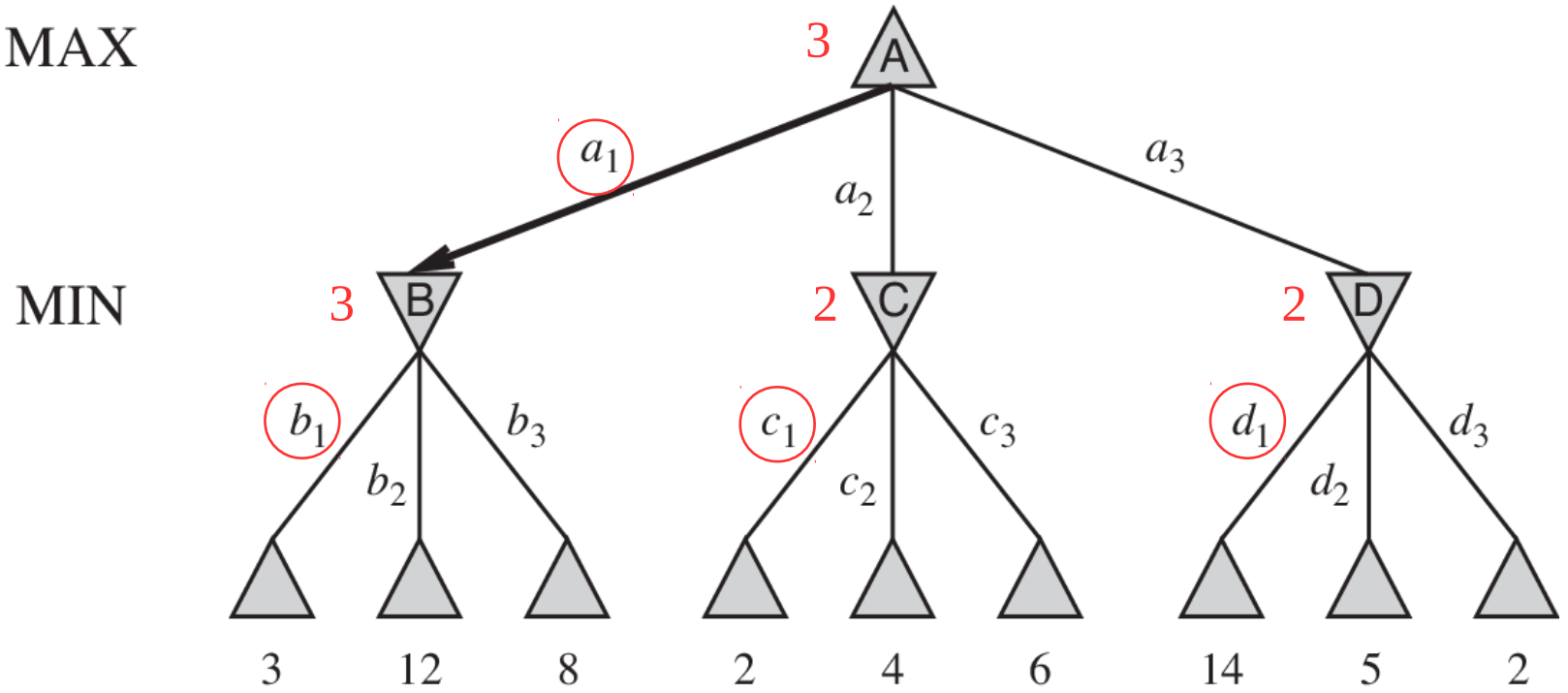
▼ ~ min node (try to decrease final score/utility)

minimax algorithm

starting from the leaf nodes (final score values),
find the value for intermediate nodes by
“backing up” the value of the node that
the player (who's turn it is) would choose

these intermediate values are called “minimax values”

actions are then taken to maximize (or minimize)
the expected minimax values at the next layer (“ply”)



minimax is a depth-first tree search, so:

time complexity: $O(b^m)$

space complexity: $O(bm)$

we now have an algorithm for game playing!

but it's expensive... $O(b^m)$

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

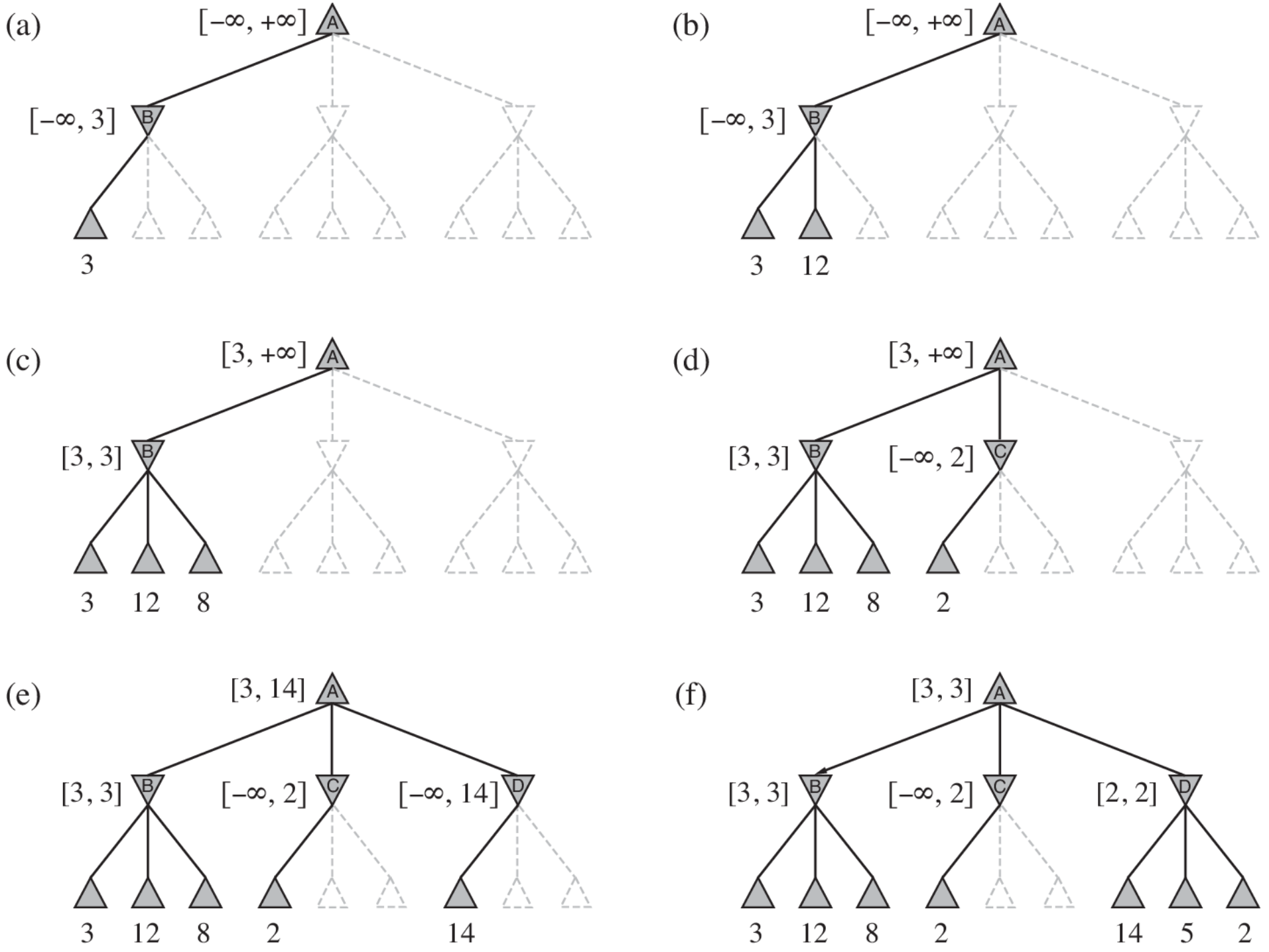
Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

alpha-beta pruning

how can we ignore (“prune”)
some branches of our tree
to save us the computation of
searching over all possible states?

let's keep a range of the possible
minimax value for each node:

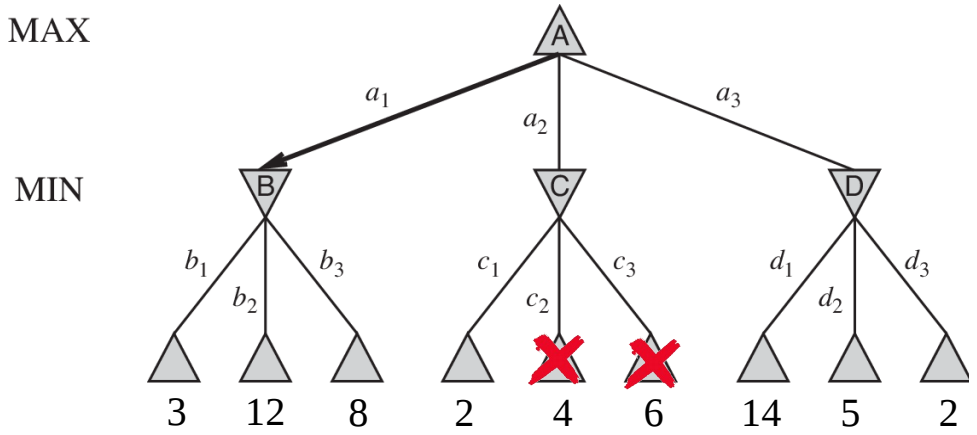
[minPossibleValue, maxPossibleValue]
[β , α]


(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2
(e)
[3, 14]
A
[3, 3]
B
[−∞, 2]
C
[−∞, 14]
D
3
12
8
2
14
(f)
[3, 3]
A
[3, 3]
B
[−∞, 2]
C
[2, 2]
D
3
12
8
2
14
5
2

of the 9 possible paths, we only had to
follow 7 of them until termination
(i.e. to finish the game and get actual scores)

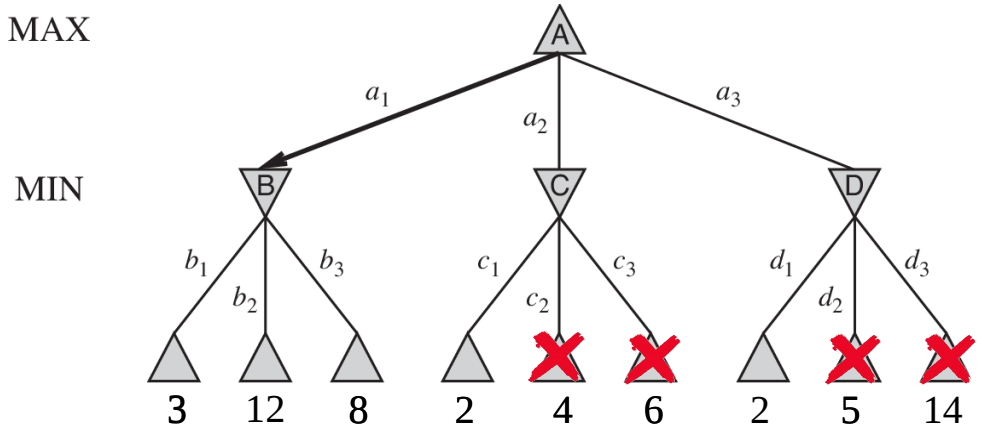
what would make this ratio even better (or worse)?

original:



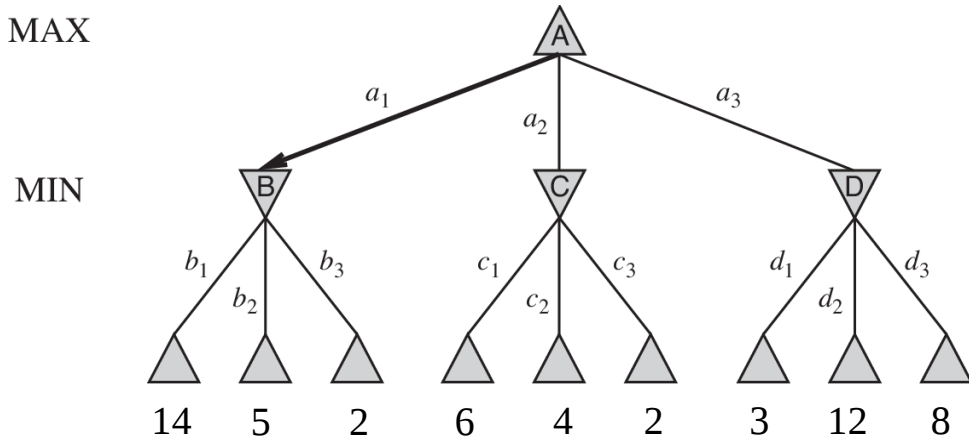
7 / 9

best:



4 / 9

worst:



9 / 9

move ordering matters!

having your highest minimax value on your first branch allows you to prune more effectively on later branches

so does performing the best action on each branch first

if you have an idea of what your “killer move” is going to be – try it first!

without alpha-beta pruning,
minimax is just depth-first search: $O(b^m)$

with pruning on perfectly ordered branches,
we can perform search in ~ $O(b^{m/2})$

but it's impossible to perfectly order branches
(if we knew their values already, we wouldn't have to do search)

on a random ordering of branches
alpha-beta pruning results in ~ $O(b^{3m/4})$

since the branches we are pruning are
parts of the game that would never be reached
(by two optimal agents playing each other),

the efficiency gains of alpha-beta pruning
do not come at any loss of optimality

(i.e. time savings for free!)

even though alpha-beta pruning lets us
evaluate trees/paths that are twice as long,

often it's still impossible to reach the goal state
(e.g. games like chess or go)

we need a way to differentiate between non-goal states

let's use a heuristic function!
(a.k.a evaluation function, value function)

humans use heuristic function to evaluate actions
(we can't see 100+ moves into the future,
but we can guess compare future non-goal states)

e.g. in chess, we often know that a move that captures
a queen is worth more than one that captures a pawn

(even though we haven't followed those paths to completion)

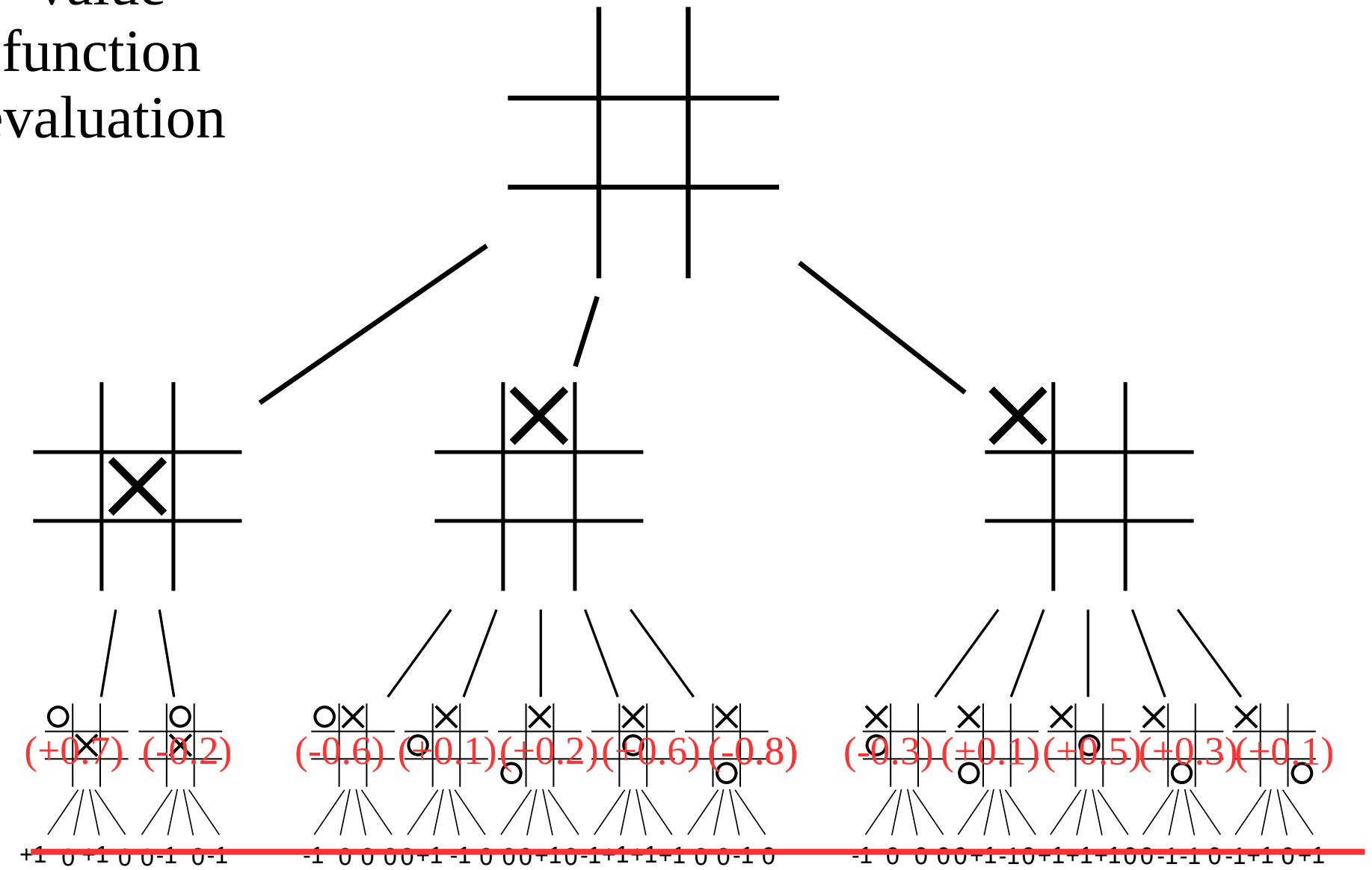
what pieces are on the board are “features” of that state

we often create heuristic value functions based on the features of the board (environmental) state

e.g. value of a board state = weighted sum of pieces on board:

value = #pawns * 1 + #bishops * 3 + #knights * 3 + #rooks * 5 + #queens * 9

value function evaluation



linear weighted sums are fast, so they're used often
(the whole point of a heuristic function is to save time!)

but this case assumes independent (non-interacting) pieces

and doesn't account for different layouts/arrangements

(you could have a different feature for each
combination of piece and position)
64^7 = over 100,000,000,000 features...

how do we efficiently build flexible and robust feature sets?

optimization on non-linear function approximators

(e.g. machine learning with artificial neural networks)
(more on this later…)