# EECS 391: Introduction to AI

Soumya Ray

Website: http://vorlon.case.edu/~sray/eecs391_sp12/index.html

Email: sray@case.edu

Office: Olin 516

Office hours: Tue 2:30-4:00 or by appointment

# Today

- Adversarial Search (Chapter 5)

# Games

- We consider zero-sum games of perfect information

- Perfect Information: Everything except the other agent's strategy is known

- Zero-sum: when one agent wins, the other loses (by the same amount)

# Zero-sum games

- A and B are playing a game
- Let's define a utility function for each state of the game for A:

$$U_A(s) = \begin{cases} c \text{ if A wins in } s, c > 0 \\ -c \text{ if B wins in } s \\ 0 \text{ otherwise} \end{cases}$$

- Therefore, A will try to maximize $U_A(s)$ and B will try to minimize it
  - A is also called "MAX" and B is called "MIN"
  - $U(s)$ can be also be numeric

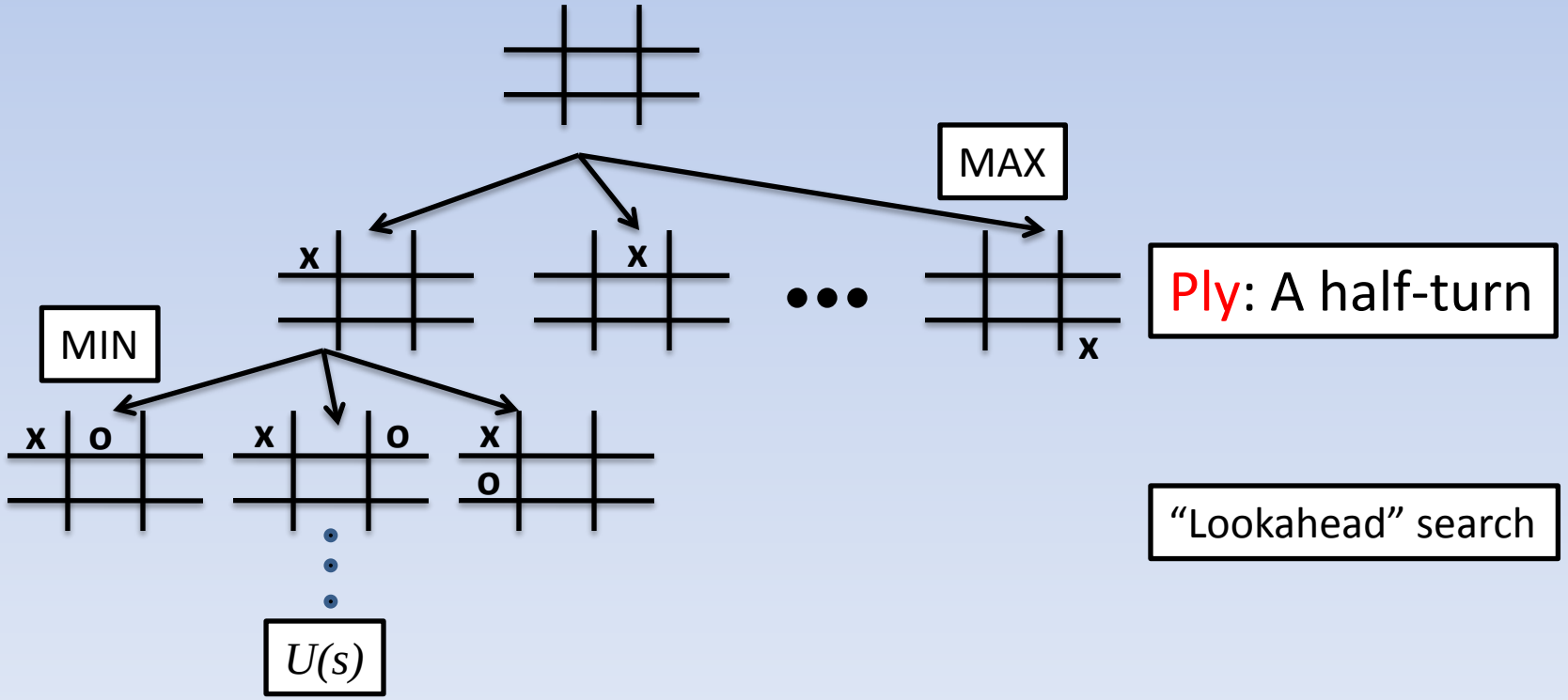
# Games as Search

- Initial state=initial configuration of pieces, cards etc
- Goal state=?
- Search Operators=?
- Cost=?
- Utility function

But…what about the other agent?

# Game Tree

- Start at initial state and generate all legal successors with A playing, then B, …



# Solving a Game (ideal scenario)

- While not (game over)
  - Build the game tree from current state
  - Compute *optimal strategy*
  - Play first move of optimal strategy
  - Wait for opponent to respond
  - Repeat

# Optimal Strategy

- Starting from any node in the game tree, what is the optimal strategy for MAX?
  - The best that MAX can do, *no matter what* MIN does

- To determine this, we need to "propagate" $U(s)$ from the leaves of the game tree up to this node

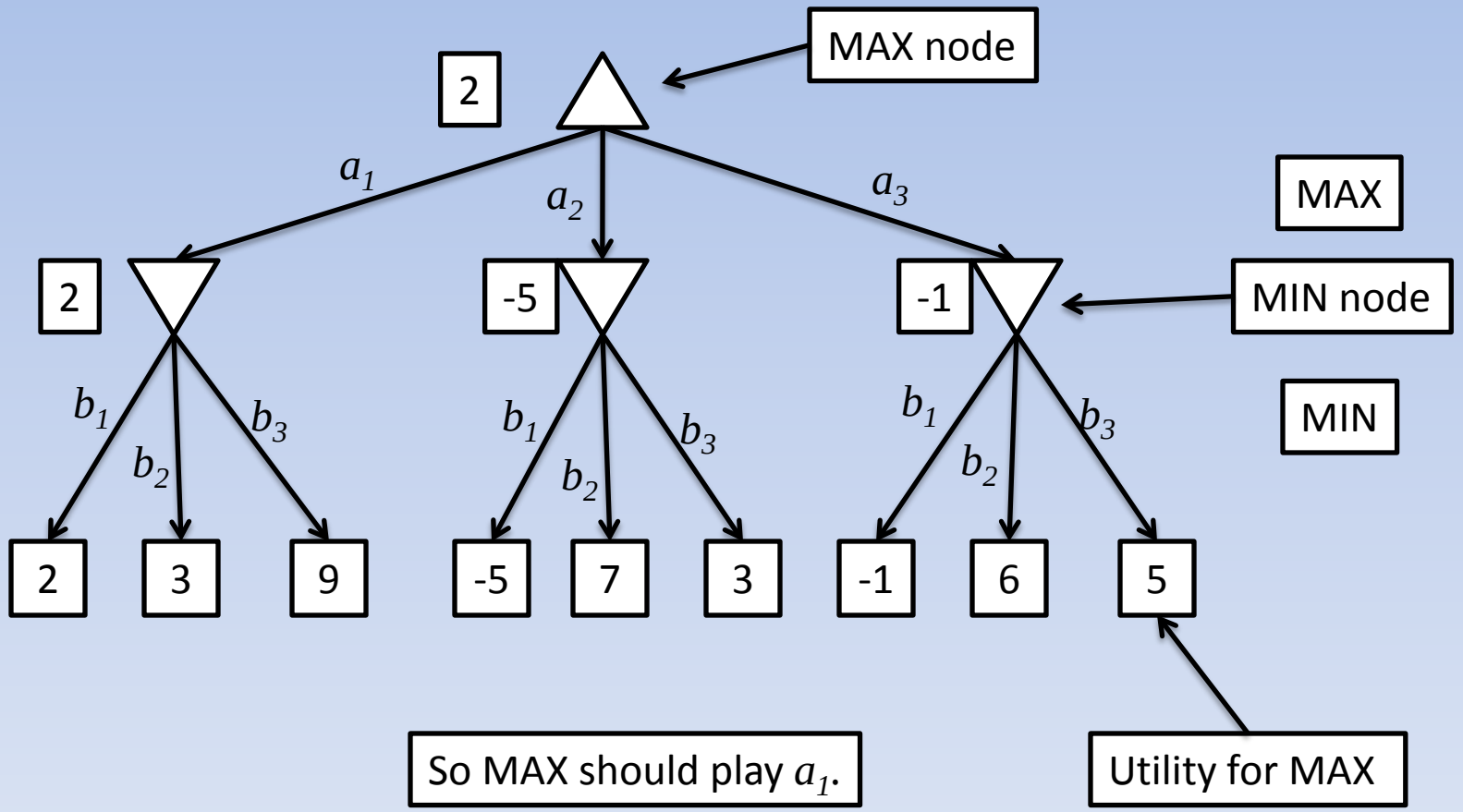
# *Minimax(s)*

- Define *minimax(s)* to be the "value" of some state *s*, *assuming both players play optimally from that point on*
  - The value is defined with respect to whichever agent (MAX or MIN) whose turn it is to play, or *U(s)* for MAX for terminal states

# Calculating *minimax(s)*

$$mm(s) = \begin{cases} U(s) \text{ for MAX if } s \text{ terminal} \\ \max_{s' \in successor(s)} (mm(s')) \text{ if MAX to play} \\ \min_{s' \in successor(s)} (mm(s')) \text{ if MIN to play} \end{cases}$$

# Example


MAX node
2
$a_1$
$a_2$
$a_3$
MAX
2
-5
-1
MIN node
$b_1$
$b_2$
$b_3$
MIN
2
3
9
-5
7
3
-1
6
5
So MAX should play $a_1$.
Utility for MAX

# Minimax algorithm

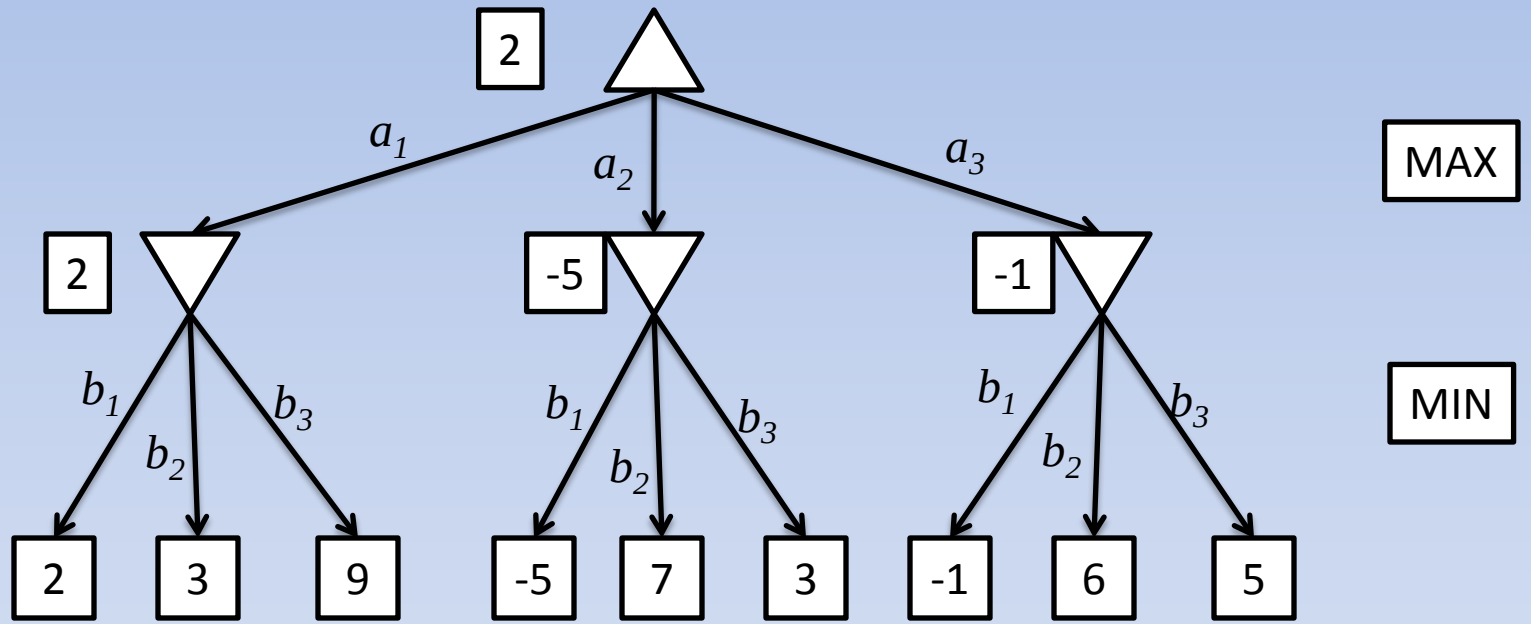
- Recursively compute $mm(s)$ at each $s$
  - 2 functions, MAX and MIN, that call each other

- The recursion effectively performs a depth-first exploration of the game tree
  - As the recursion unwinds, the values of the lowermost states are "backed up" to the ancestors

# Suboptimal play

- What if MIN does not play optimally to MAX's strategy?
  - Then MAX will do at least as well as if MIN had played optimally
  - However, minimax may not result in the optimal strategy vs MIN's strategy
    - But then again, the optimal strategy against MIN's suboptimal play would not be optimal against an optimal player

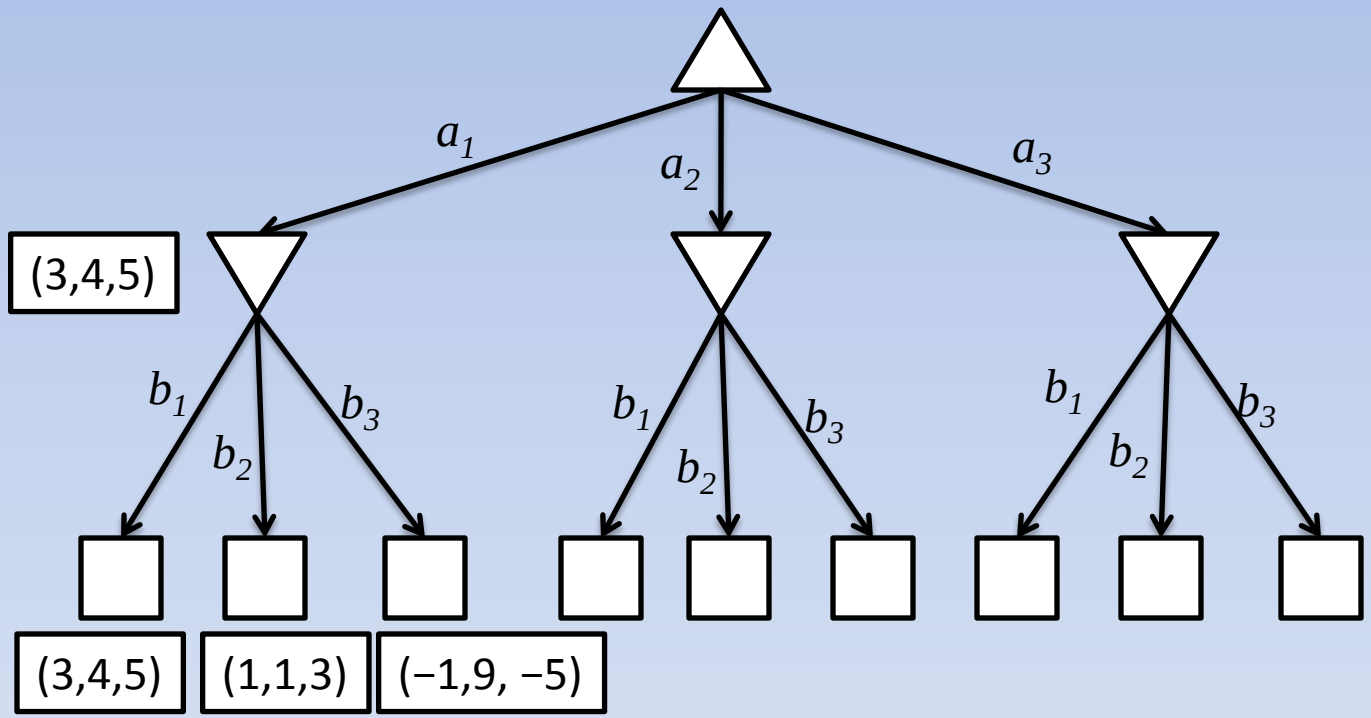
# Example



# Multiplayer games

- Conceptually the same algorithm
- Maintain a vector of utilities, one per agent, at each state
- The backup operation backs up the highest utility of the agent moving that turn

# Example

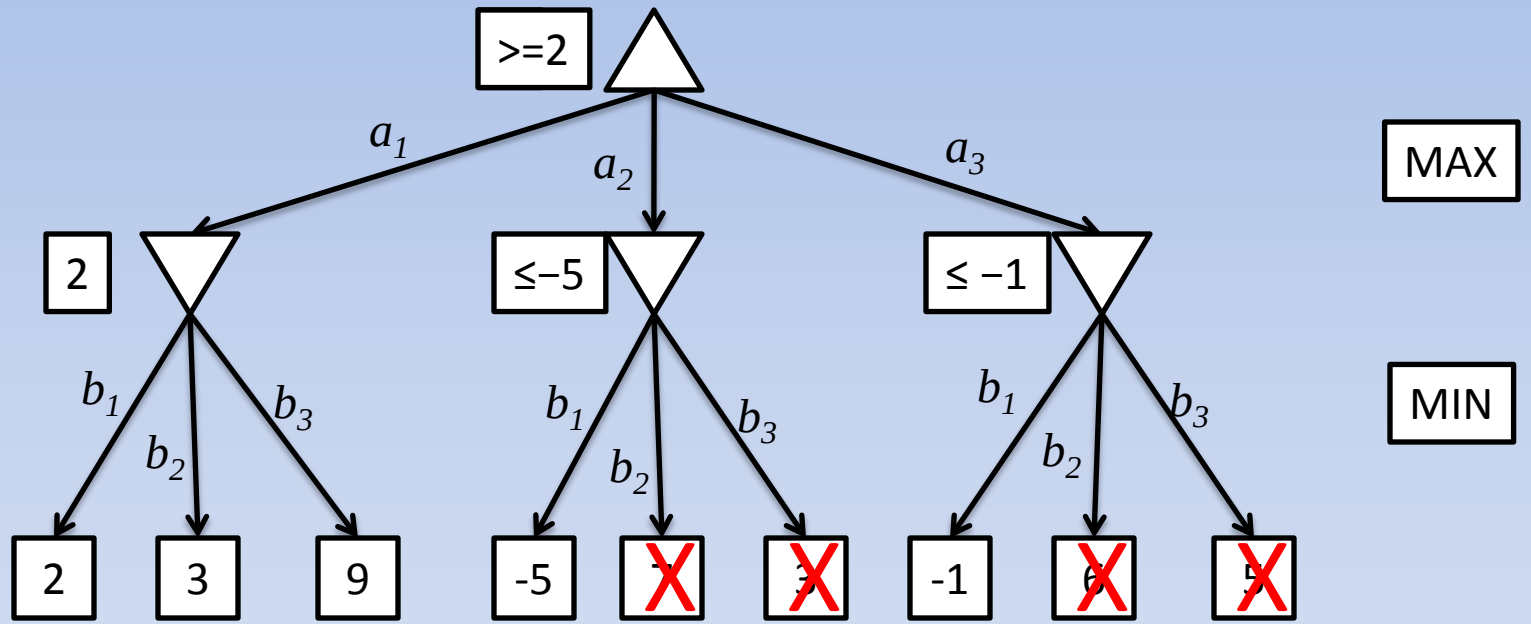


# α-β Pruning

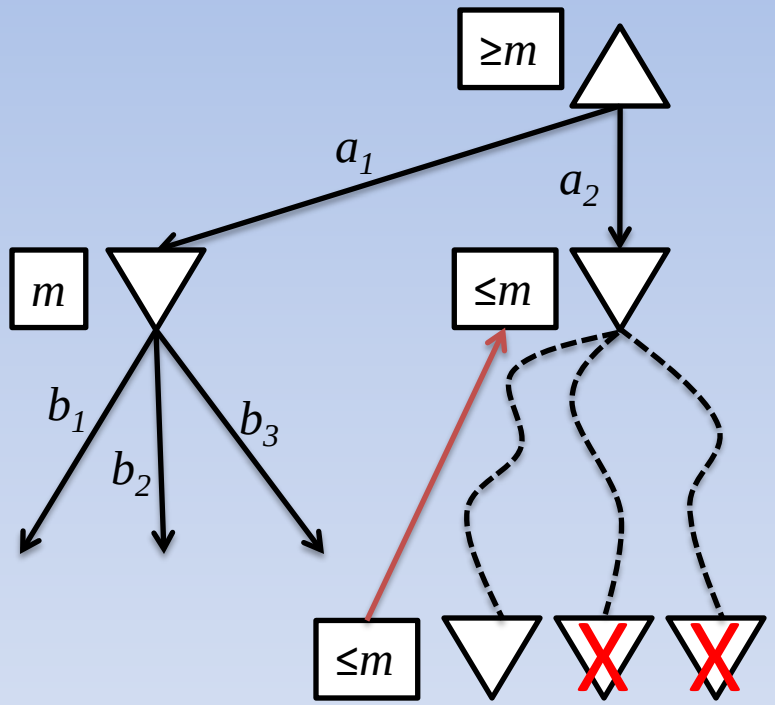
- Minimax is expensive (exponential in number of moves)

- We can in fact obtain the optimal strategy without looking at all the nodes
  - A method of pruning the game tree by maintaining bounds on the minimax values

# Example



# General Idea



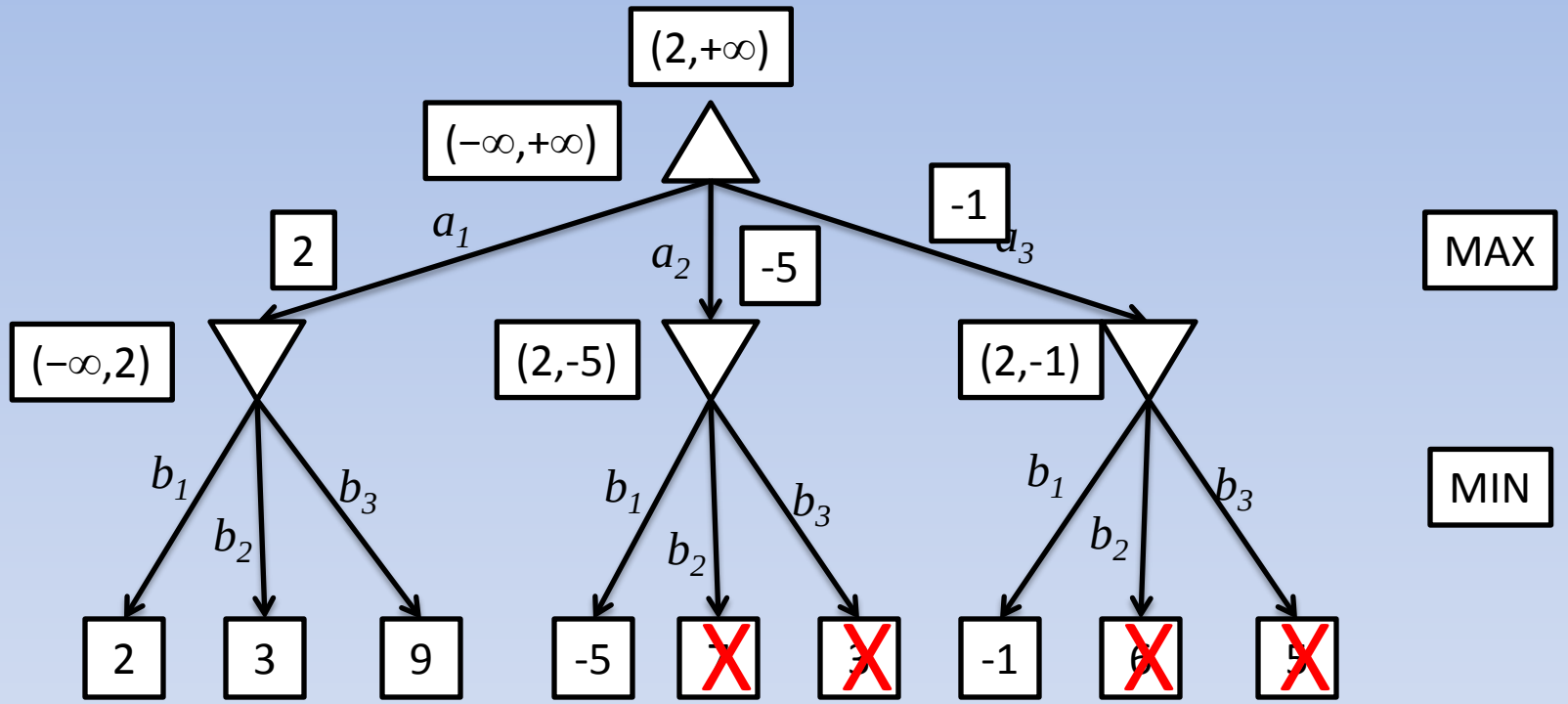
# α-β Pruning

- Maintain two parameters:
  - α = the best choice so far along the path for MAX
  - β = the best choice so far along the path for MIN
- Update these values as the search proceeds
- If we are at a MAX (MIN) node and its value is larger (smaller) than current β (α), prune

# Algorithm

- Set $(\alpha,\beta)=(-\infty,+\infty)$
- Perform standard minimax backup for each successor in turn
- If at a MIN node:
  - If backed-up value smaller than $\alpha$, prune other successors
  - Else replace $\beta$ with smaller of $\beta$ and backed-up value
- If at a MAX node:
  - If backed-up value larger than $\beta$, prune other successors
  - Else replace $\alpha$ with larger of $\alpha$ and backed-up value

# Example



# Effectiveness of α-β

- How much we can prune with α-β depends on the order in which we see nodes
- Using knowledge of the game, we can construct heuristic criteria that help choose the best nodes first
- In the ideal case, will expand $O(N^{1/2})$ nodes, where $N$ is the number of nodes expanded by minimax

# Real-time Games

- So far, we have assumed both agents have enough time to exhaustively search the game tree
- In most games this is impossible, so they will be forced to stop early and make a move
  - How to best do this?
    - Iterative Deepening
    - Evaluation functions

# Evaluation Functions

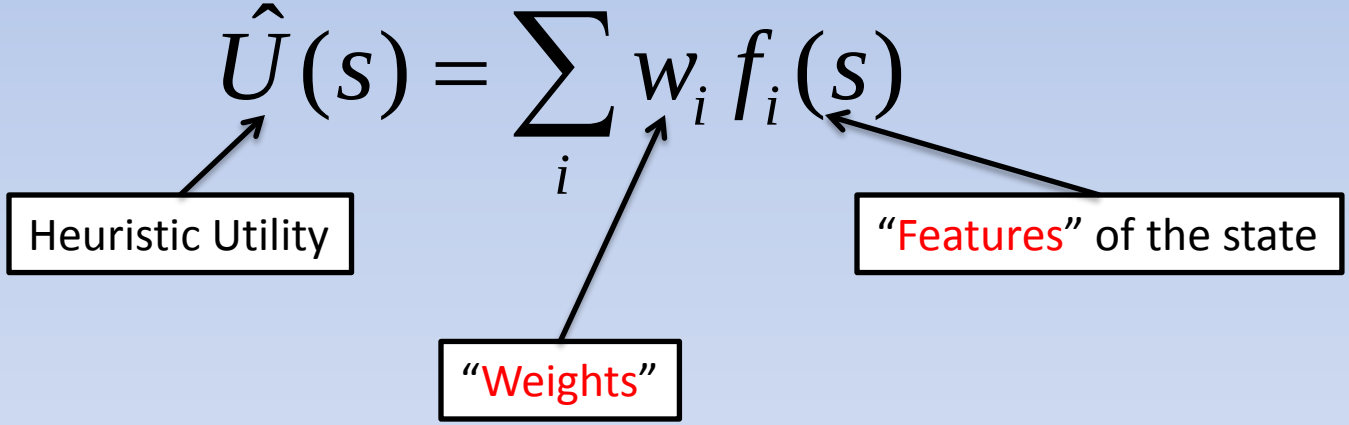
- If we are forced to stop the search at a non-terminal state, we can use *heuristic evaluation functions* to estimate what the backed up utility of that state could be
  - Similar idea as heuristics in goal-directed search
  - Except now we "look inside" states to determine the functions

# State Features

- To estimate the utility of a state, we compute *features* of a state
  - These are characteristics that we think might contribute to the overall utility
  - For example, in chess, we could have a features that measured the advantage in pieces and control of the center

# Linear Evaluation Functions

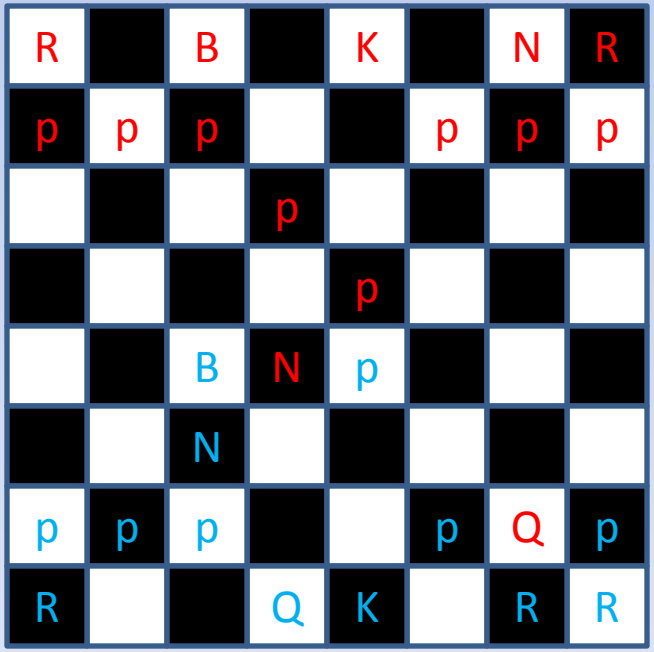
- A simple way to combine state features and express the heuristic utility of a state




- Assumes state features are additive
  - More complex functions are nonlinear e.g. neural networks (later)

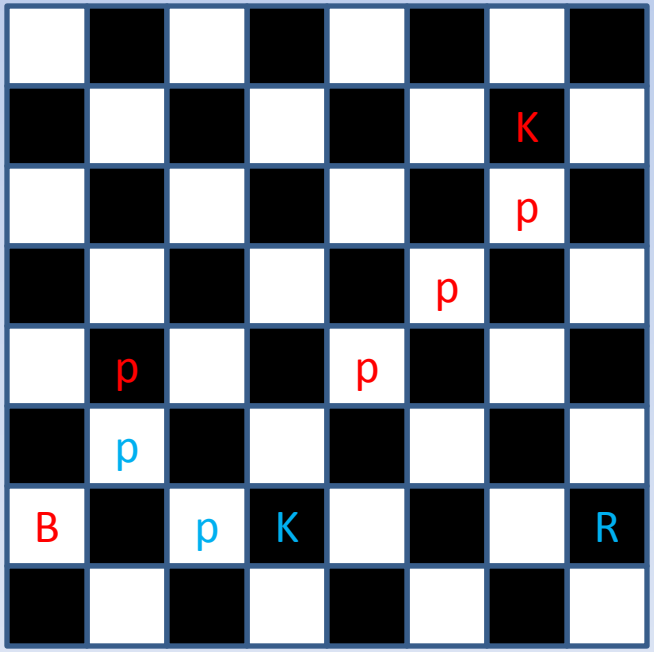
# Stopping the Search: Quiescent States

- The very next move could cause the evaluation function to change a lot
  - Figure 5.8 in book



# Stopping the Search: Horizon Effect

- A sequence of moves can postpone an inevitable bad outcome
  - Figure 5.9 in book

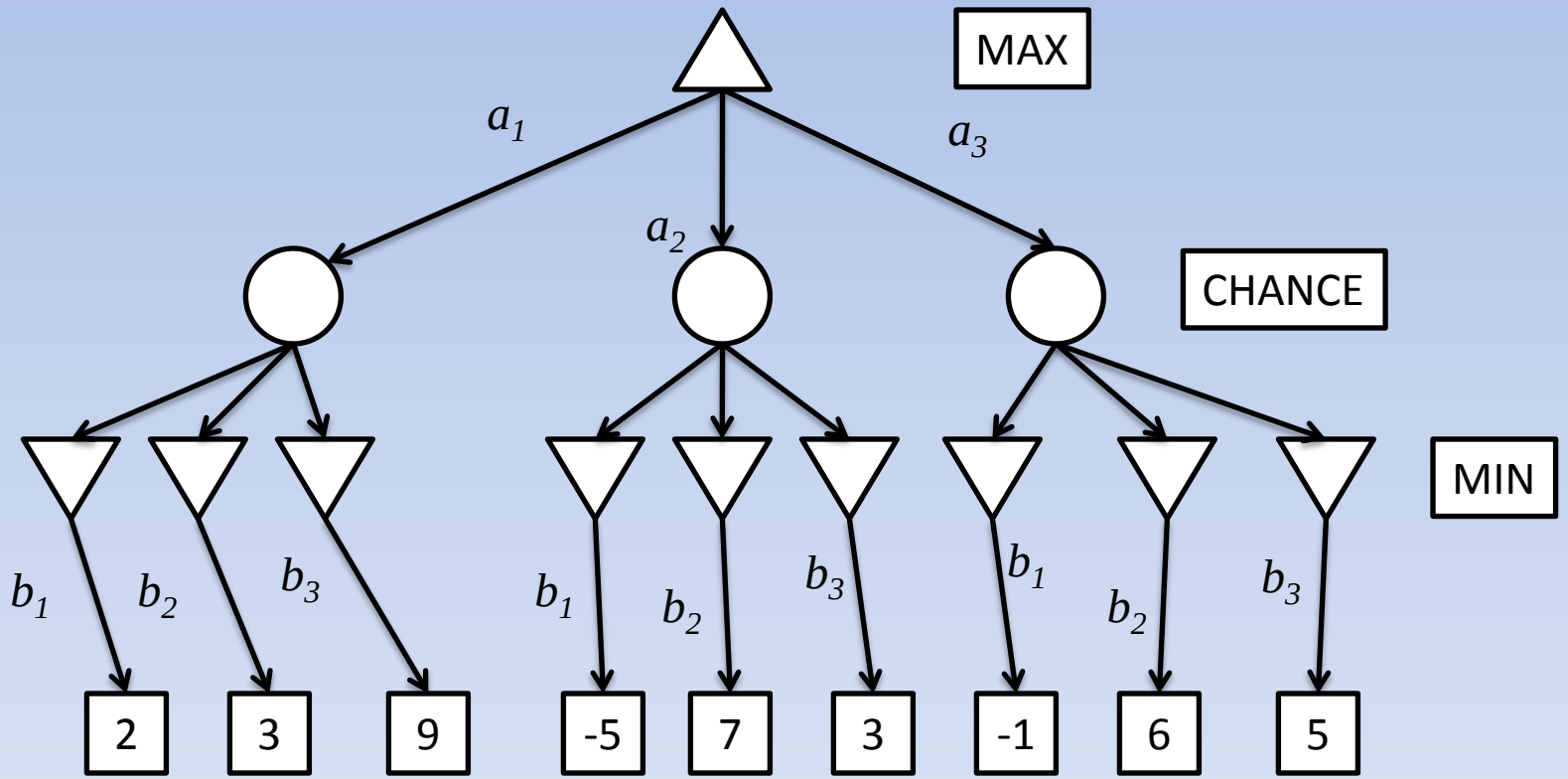


# Games with Chance

- Suppose the game is not deterministic and includes moves based on a die roll
  - How do we take this into account?

- We can construct game trees with special "chance nodes" that represent the die roll

# Example

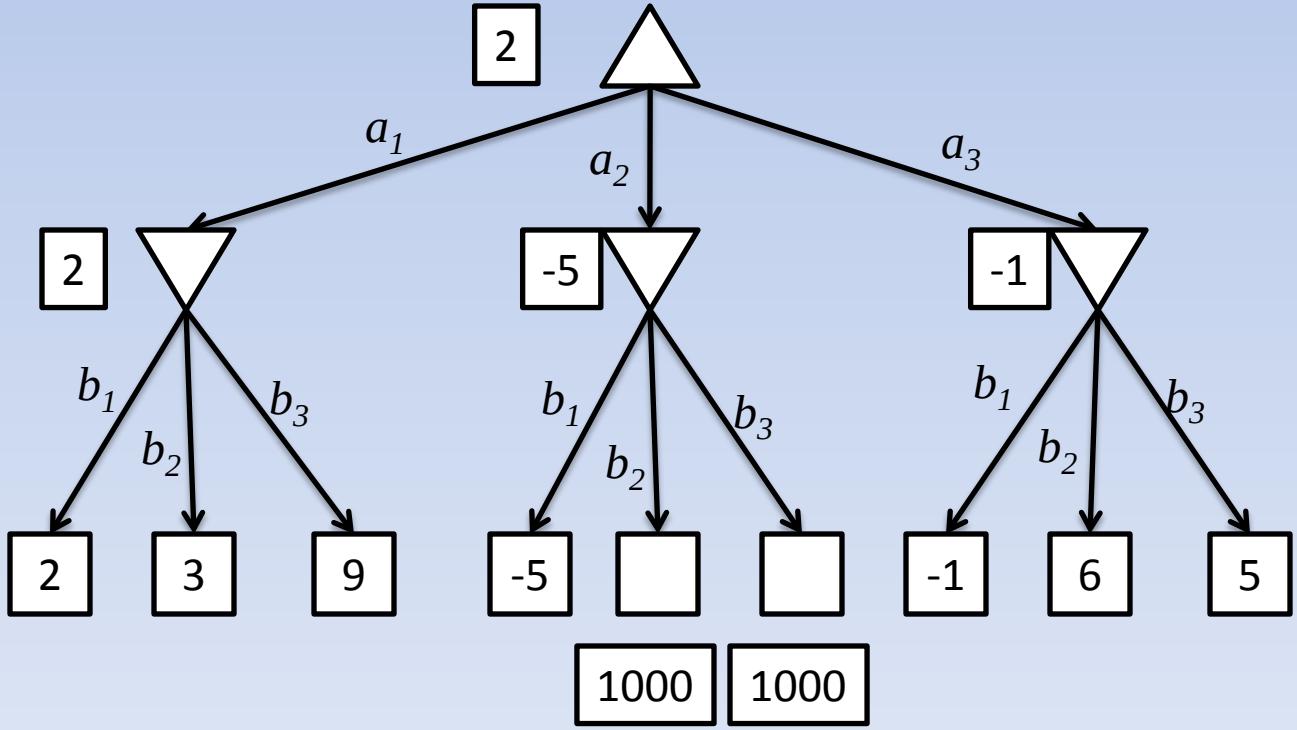


# “Expectiminimax”

$$emm(s) = \begin{cases} U(s) \text{ for MAX if } s \text{ terminal} \\ \max_{s' \in successor(s)} (emm(s')) \text{ if MAX to play} \\ \min_{s' \in successor(s)} (emm(s')) \text{ if MIN to play} \\ \sum_{s' \in successor(s)} \Pr(s') emm(s') \text{ if } s \text{ CHANCE} \end{cases}$$

# Issues with minimax (1)

- If the utilities are approximate, may not be a good idea



# Issues with minimax (2)

- In some cases, there may be a “clear favorite” node, but minimax/α-β will still expand many other legal moves
- Along with a utility function, could have a function that evaluates the utility of expanding a node given what it has seen so far
- “Metareasoning”---thinking about the method of solution rather than the solution

# Issues with minimax (3)

- Do humans play like this?
  - Maybe to some extent, but often not
  - Humans are more goal-directed (long term strategic thinking)
  - This requires reasoning about the state rather than blind search---will see more of this in planning

# Searching with Constraints (Ch 6)

- Sometimes problems have constraints the solution should respect (e.g. Sudoku puzzles)
- “Constraint Satisfaction Problems”
- Solvable with standard search, but
  - Need to track constraints at each step
  - Can take advantage of constraints to prune large parts of search space

# Summary

- We learned about:
  - Zero sum games of perfect information
  - Search for games
  - Minimax algorithm
  - Alpha-beta pruning
  - Real-time games
  - Games with chance
  - Issues with minimax