CS5100 Foundations of Artificial Intelligence

Module 09 Lesson 14

Adversarial Search

Some images and slides are used from CS188 UC Berkeley/AIMA with permission
All materials available at http://ai.berkely.edu / http://aima.cs.berkeley.edu

Overview

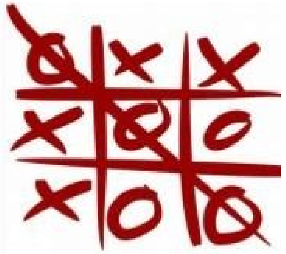
Minimax Algorithm

a-β (Alpha-Beta) Pruning

Evaluation functions

Expectiminimax Algorithm

In this module…



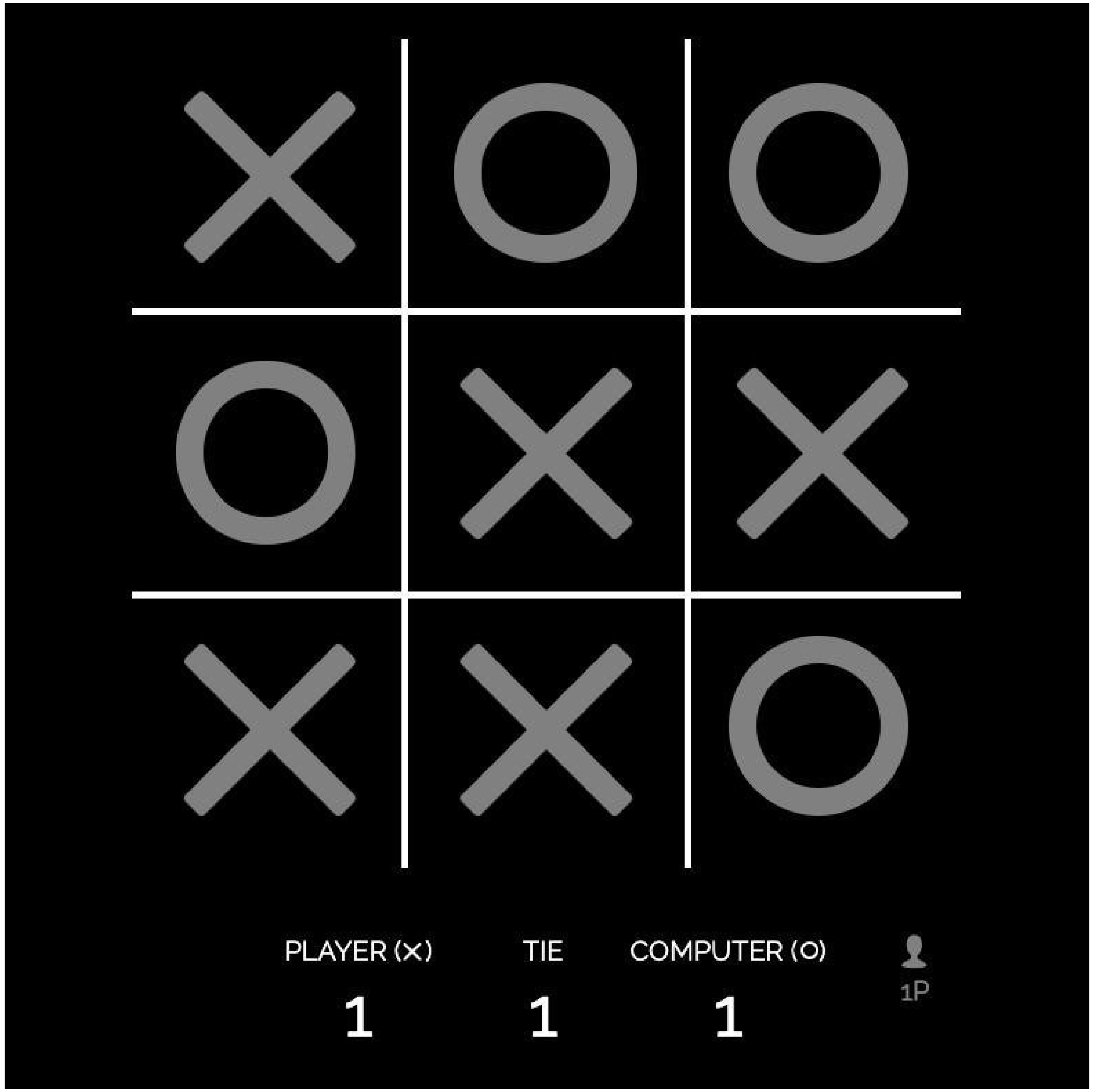
- We consider
  - Multiple agents
  - Competitive (zero-sum), Sequential, and Deterministic (and later Stochastic) environments
  - Using atomic representations

  - AI interested in: Deterministic, two-player, perfect info, zero-sum, sequential games
  - E.g. tic-tac-toe (noughts and crosses), chess, checkers, etc.

  - Bridge, poker etc. are different – imperfect information, multi-player
    - Some info not visible to all players

Tic-tac-toe or Noughts and Crosses

https://playtictactoe.org/



On Checkers

Computer Checkers
Early research dominated by Samuel's seminal work
First public man-machine competition in 1963
Exhibition match against World Champion in 1968
Machine learning milestone

On Chess - Garry Kasparov versus Deep Blue

On Go

Search vs. Games

Search

- Single agent
- Bad path or bad heuristics ➔ slow to get to solution,
  or no solution

Adversarial Search / Games

- At least 2 agents, may be more
- Bad heuristics/path ➔ (big) loss

Why Games?

Hard to solve!

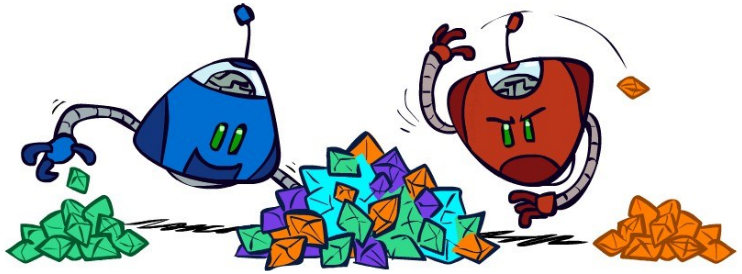
Can model real-life situations

Rules are clear, and the world is bounded (somewhat!)

- Branching factor for chess = 35,
  depth of game ~ 100 (50 moves by each player)
  So $35^{100} \sim 10^{154}$ nodes
- Need to make *some* decision, even if not able to calculate optimal decision
- Penalty is severe

- Research into decision-making, and how to make the best use of time
  - Tradeoff: cost to *compute* solution vs. cost of solution

Zero-Sum Games



Zero-Sum Games

- Agents have opposite utilities (values on outcomes) – win for one is a loss for the other
- Think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition

General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition etc.: all possible

Different Types of Games

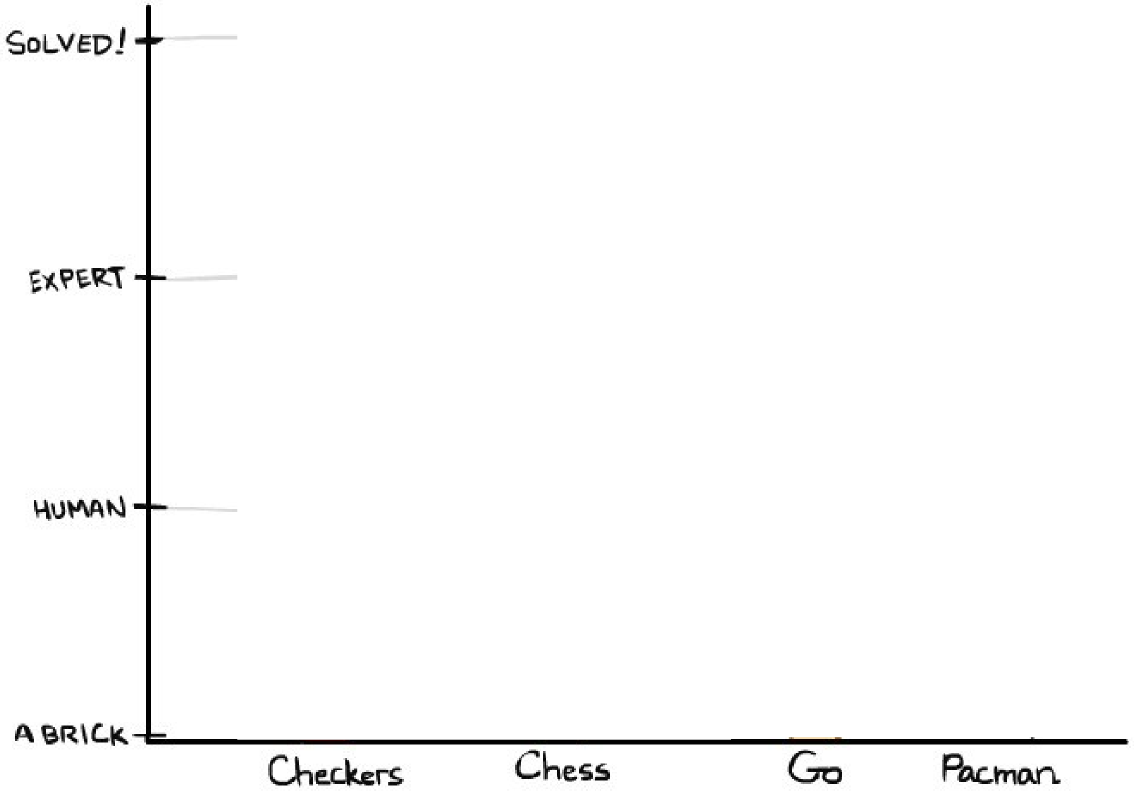
Dimensions:

- **Deterministic** or stochastic?
- Single agent or **multi-agent**?
  One, two, or more players?
- **Perfect information** (can you see the state)?
- **Sequential** vs. Episodic?
- **Zero sum**?

AI interest in: Deterministic, two-player, perfect info, zero-sum, sequential games

- Wanted: a strategy (policy) which recommends a move from each state
- Most games like tic-tac-toe, chess, checkers, etc. fall into this category
- Bridge, poker etc. are different – imperfect information, multi-player
  - Some info not visible to all players
- Physical games (hockey, baseball, tennis) more complicated, lots of actions, imprecise rules (why else would you need a ref?)

Game Playing State-of-the-Art

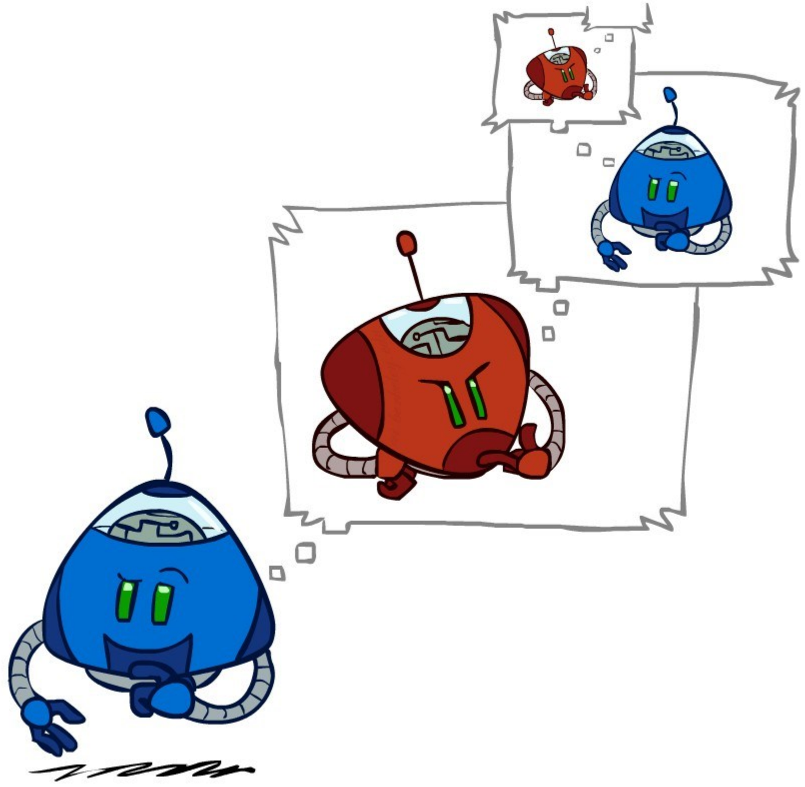


1994: Chinook beat Marion Tinsley. 2007: Checkers is solved!

1997: Deep Blue beats Garry Kasparov

2016-2017-2019 AlphaGo, AlphaZero, MuZero …

Adversarial Search

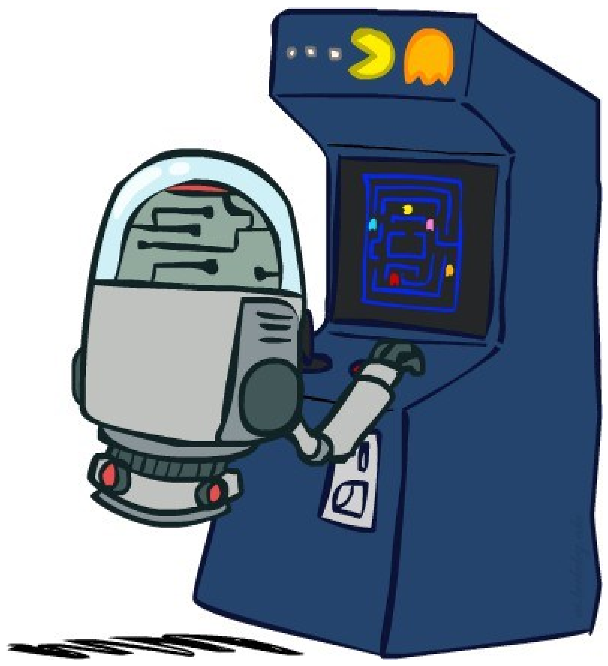


Deterministic Games

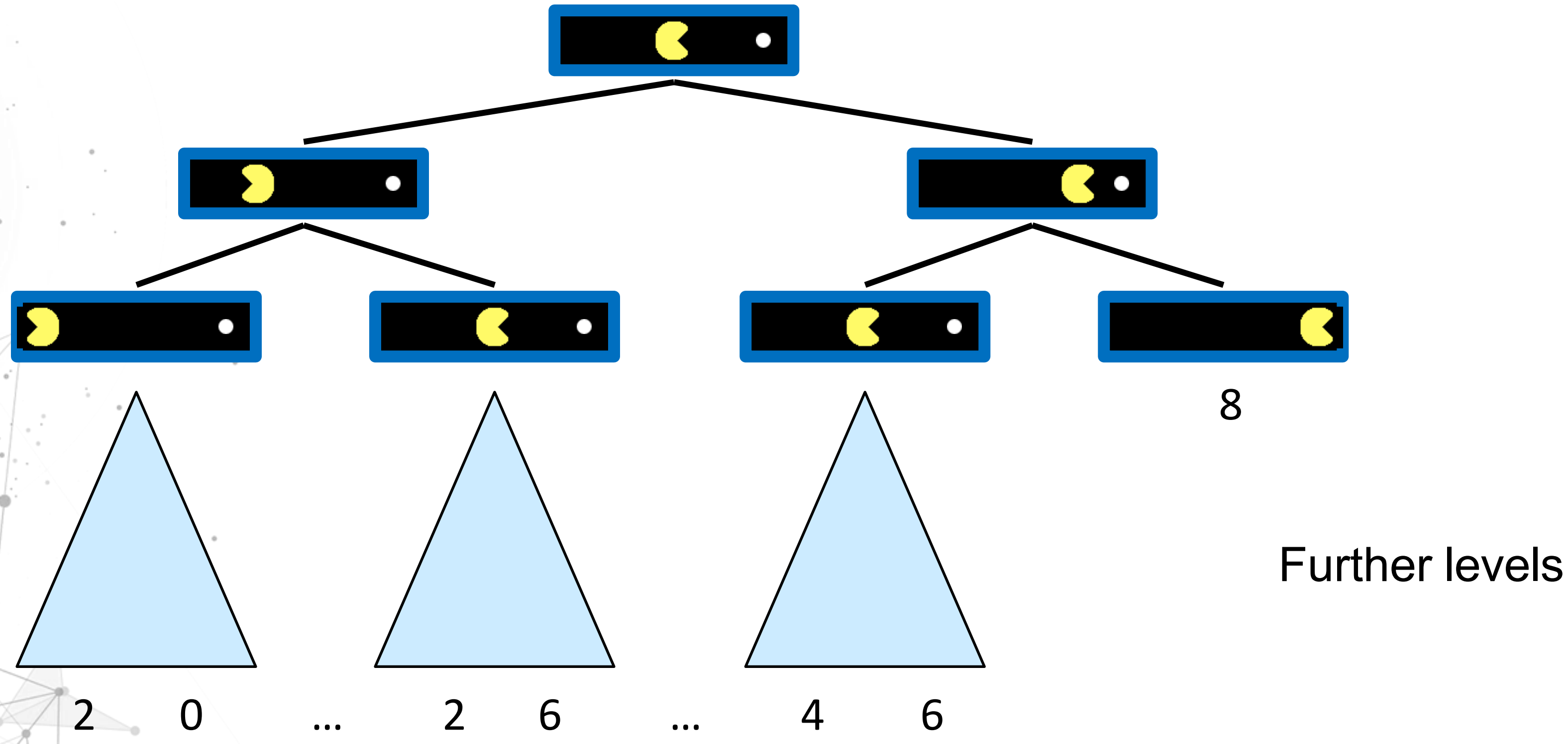
Many possible formalizations, one is:

- States S, Initial State s_0
- Players(s): P={2...N} (usually take turns)
- Actions(s): A (may depend on player / state)
- Results(s, a) or Transition Function: $SxA \to S$
- Terminal-Test(s): $S \to \{t,f\}$
- Utility function: $SxP \to R$

Solution for a player is a policy: $S \to A$



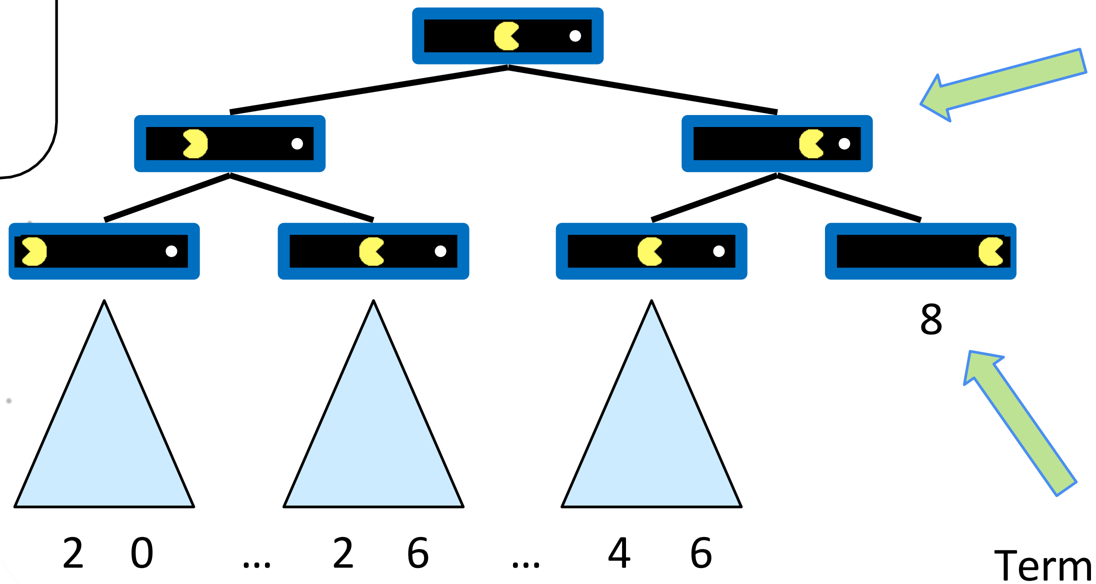
Single-Agent Search Trees



Value of a State

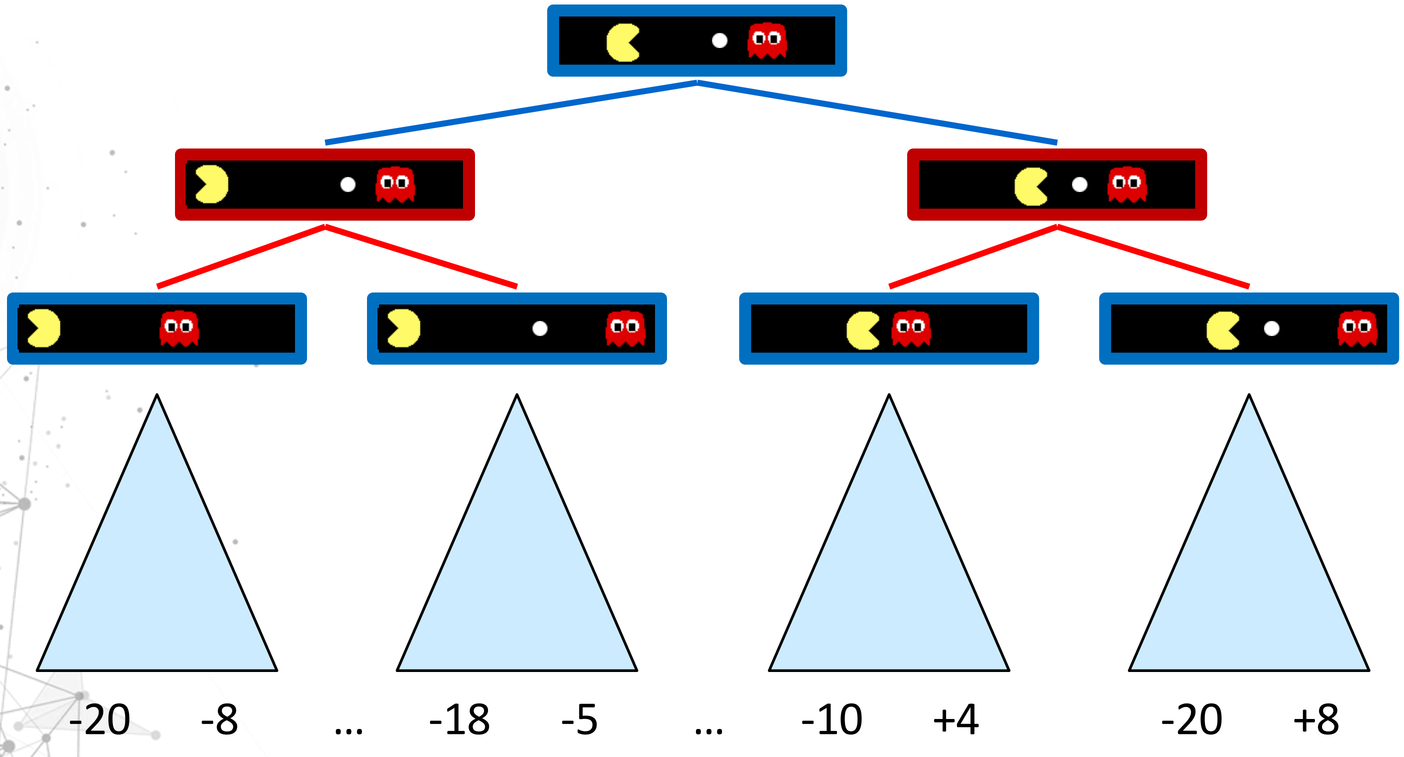
Value of a state:
The best
achievable
outcome (utility)
from that state

Non-Terminal States: $V(s) = \max_{s' \in \text{children}(s)} V(s')$



Terminal States:
$V(s) = \text{known}$

Adversarial Game Trees



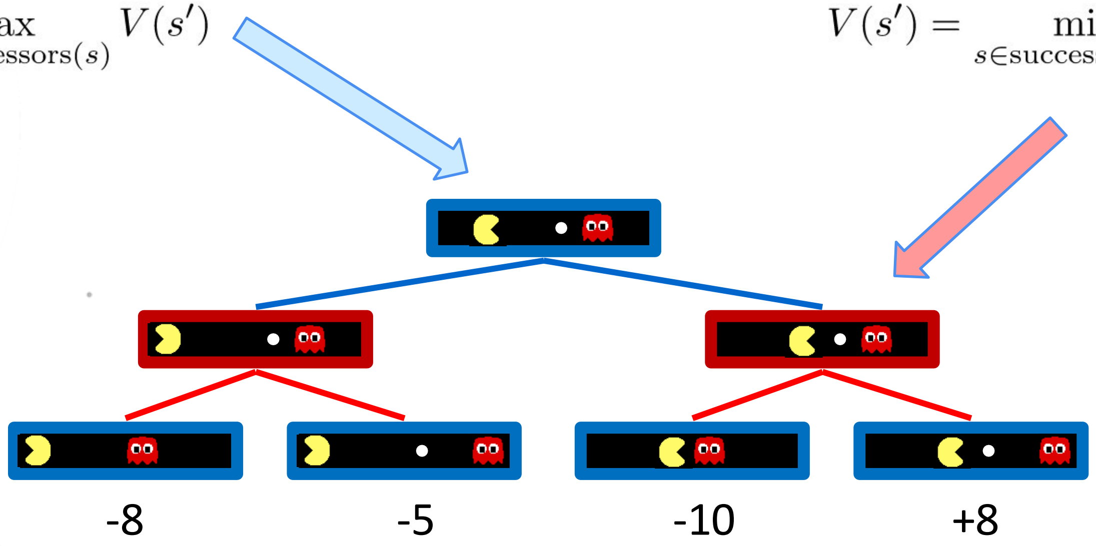
Minimax Values
Agent's Point of View (here: Pacman's POV)

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

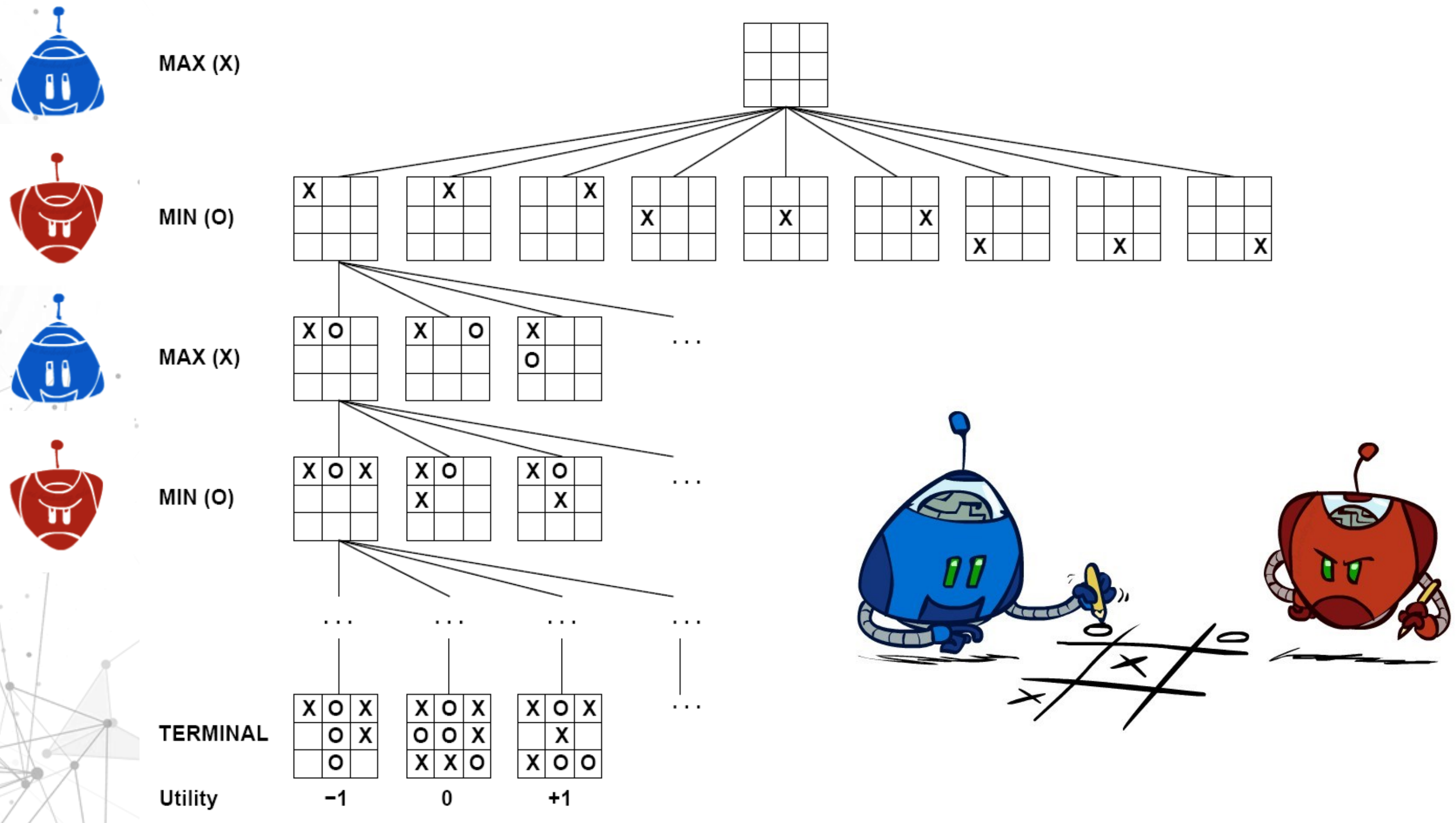
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

$$V(s) = \text{known}$$

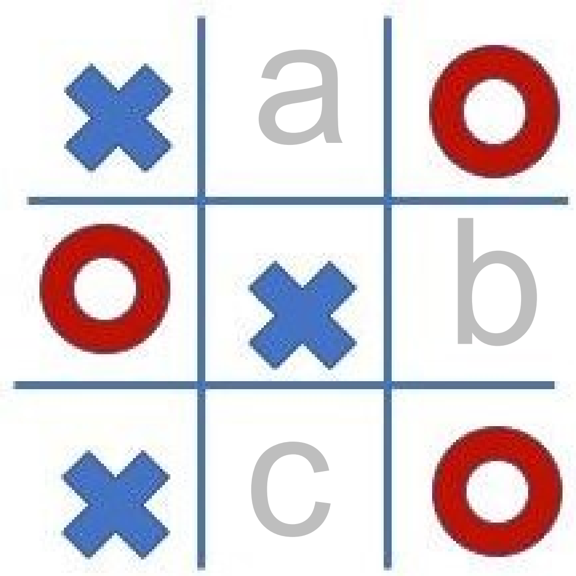
Tic-Tac-Toe Game Tree: Blue's Perspective



Exercise
Play Tic-Tac-Toe

- Consider this TTT game, after 3 moves by each player
- Assume it is X’s turn
- Assume both players are rational

- What should X play?
  - Start with what X could play.
  - Then, what might O respond with?
  - How will it play out…
  - How can you decide from the tree of possibilities?



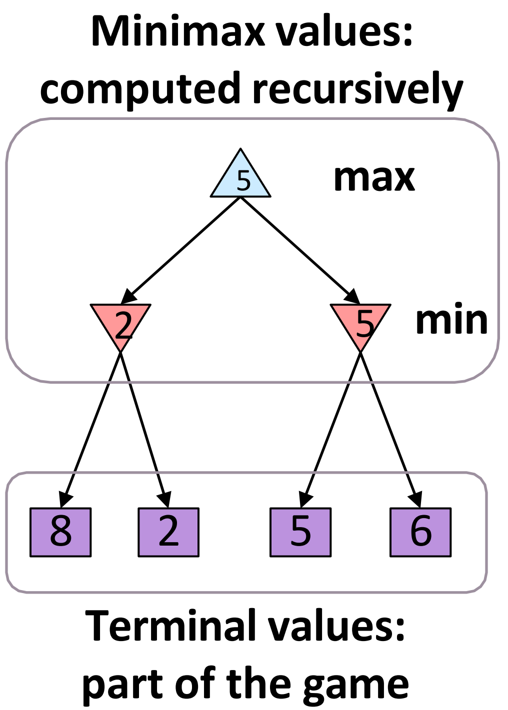
Approach to Solution

1. Play out the game
2. Label bottom-most states (boards) with +1, 0, -1
    a. +1 is a win (for the current player)
    b. 0 is a draw
    c. -1 is a loss
3. Then label boards one level up, keeping in mind whose move it is
4. Move up level by level

Minimax algorithm, implemented as DFS

Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



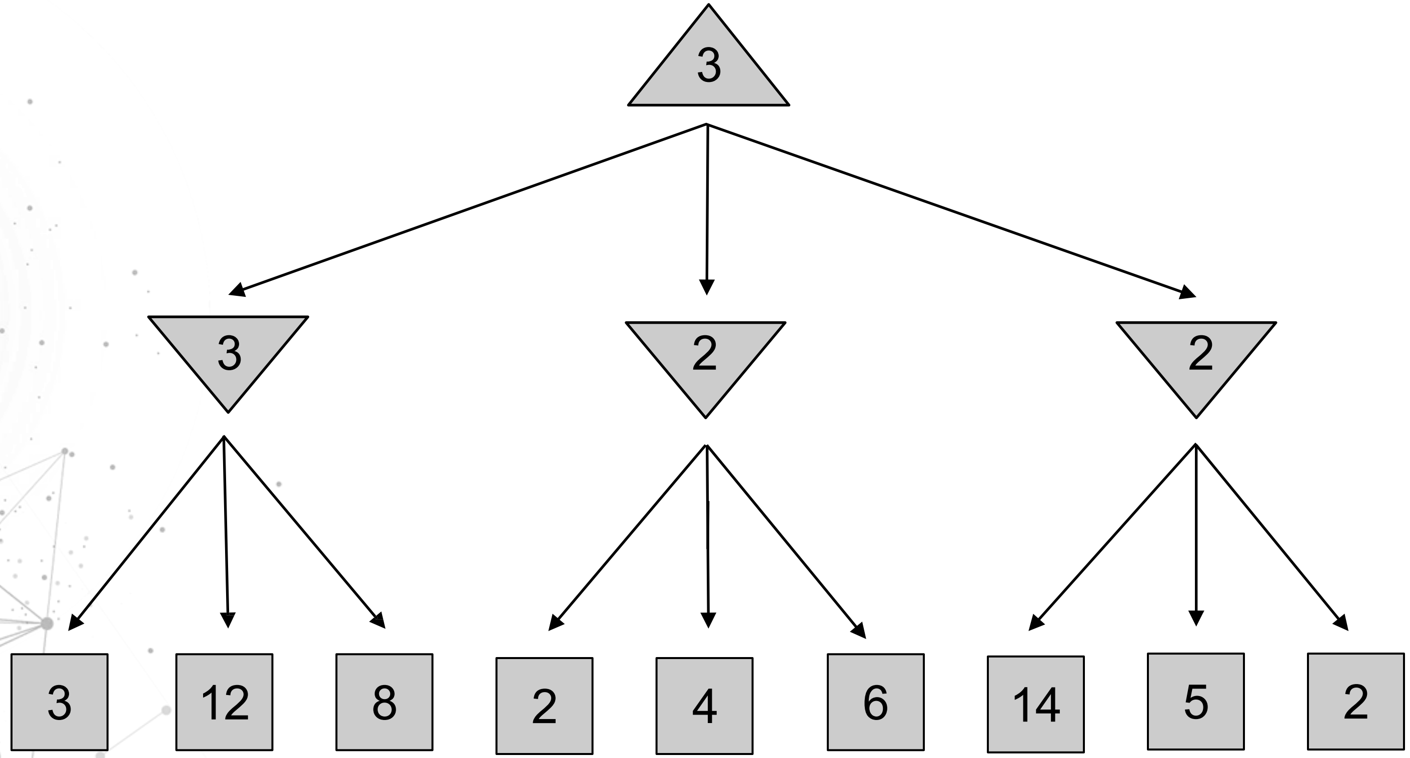
Minimax Implementation

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

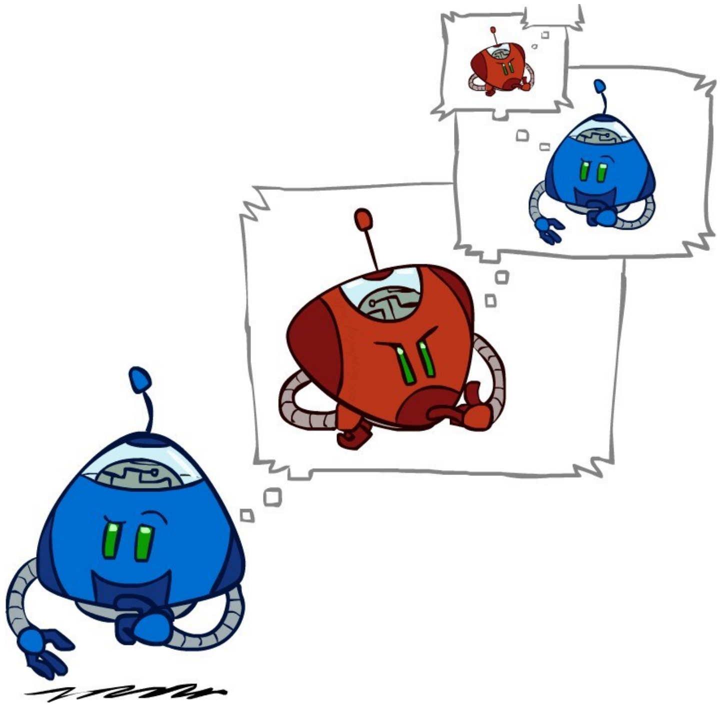
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

Minimax Example



Minimax Efficiency

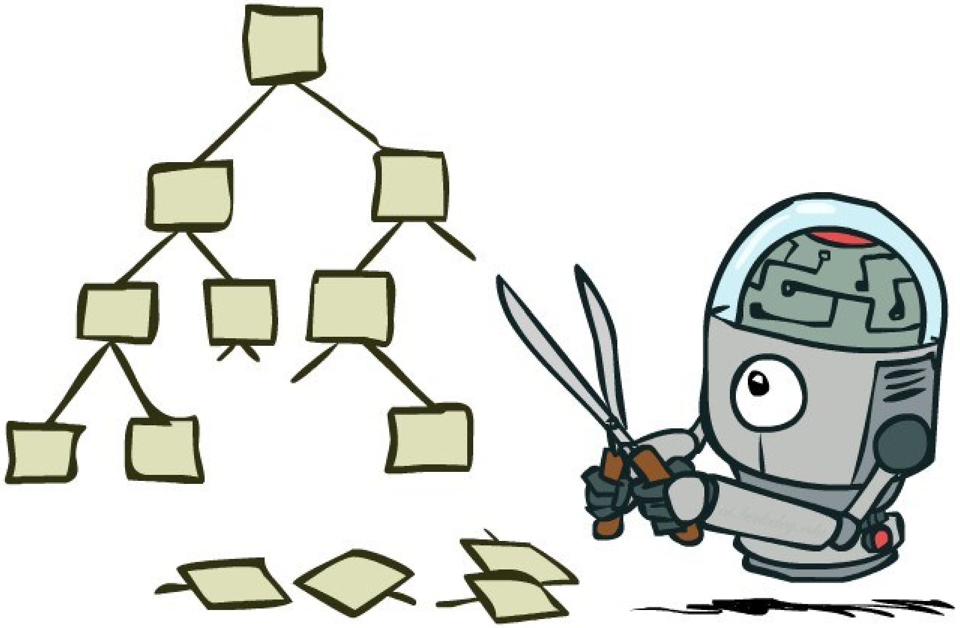
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$



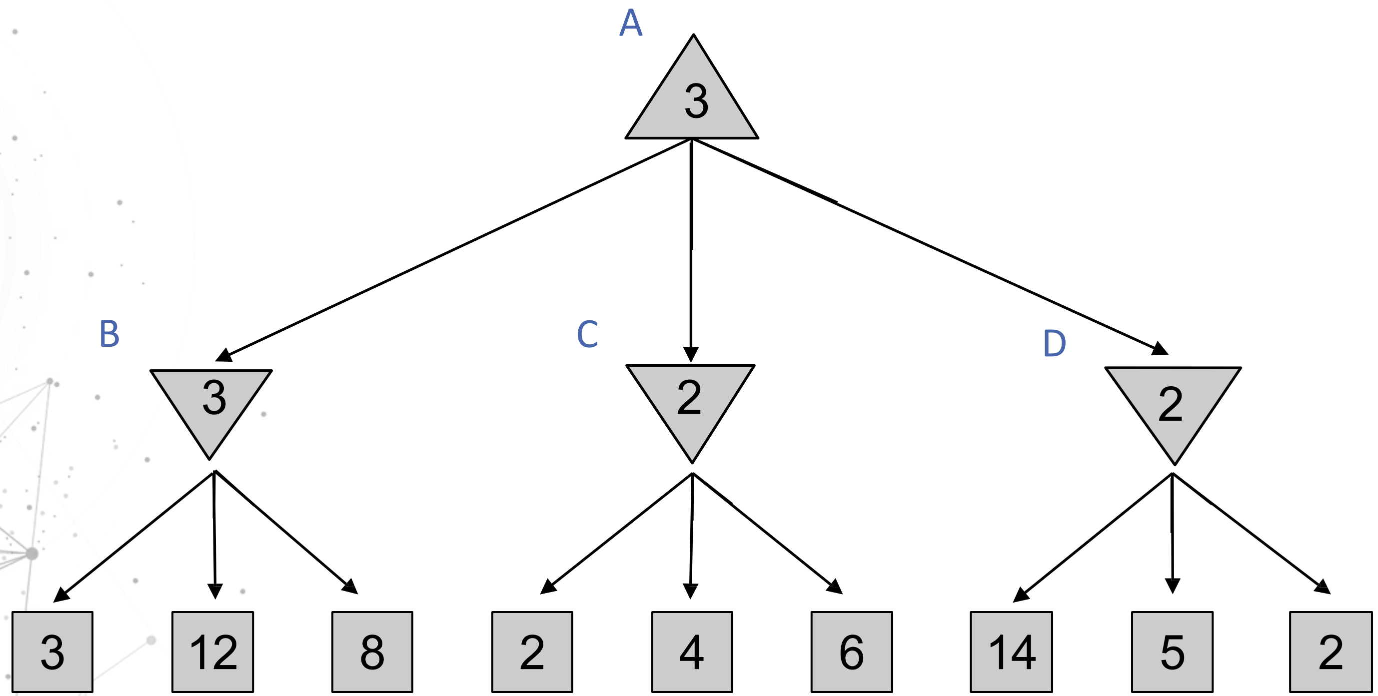
Minimax for Chess?

- Does it make sense to use minimax for chess?
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But: do we need to explore the whole tree?

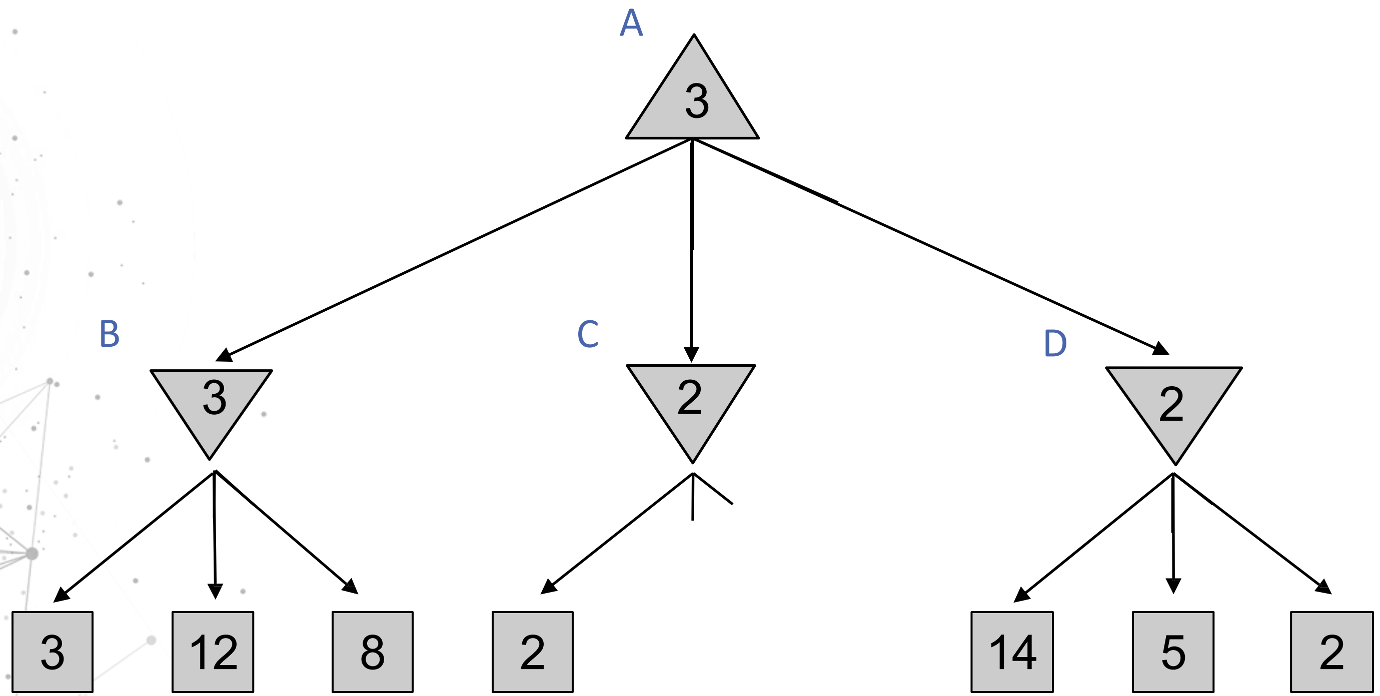
Game Tree Pruning



Minimax Example

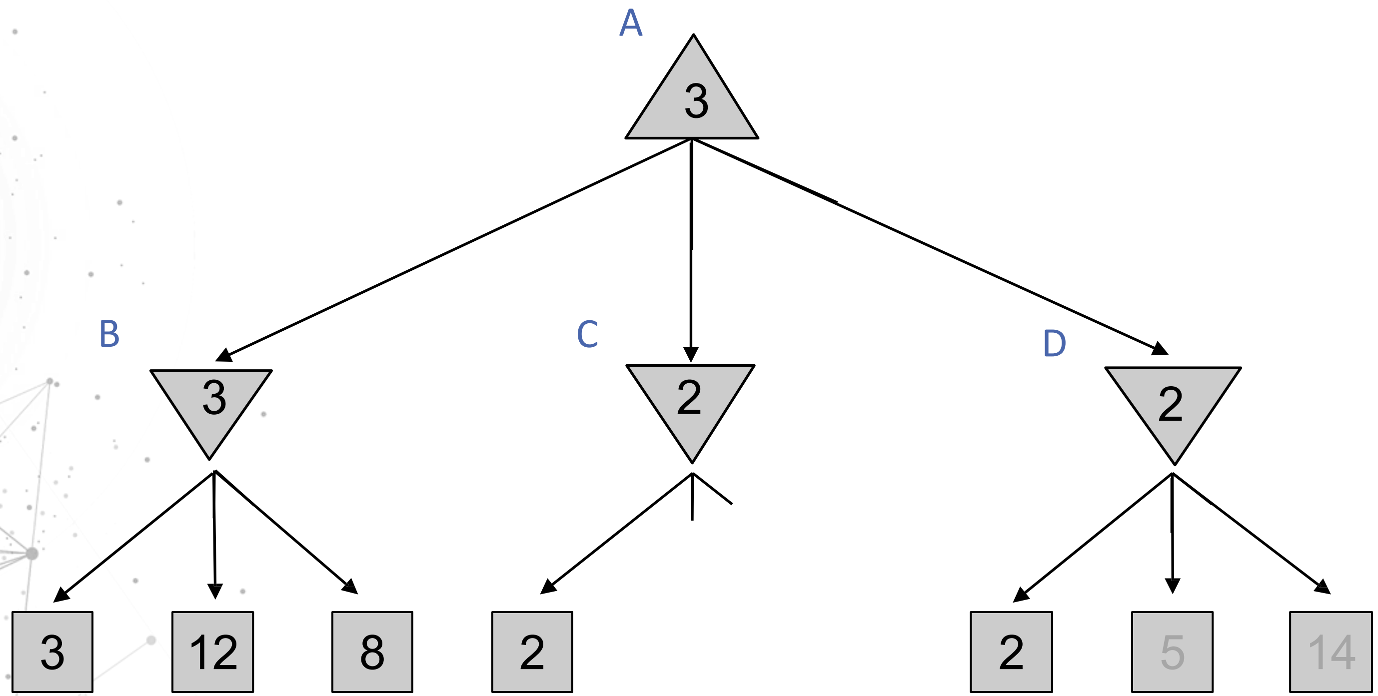


Alpha-Beta Pruning

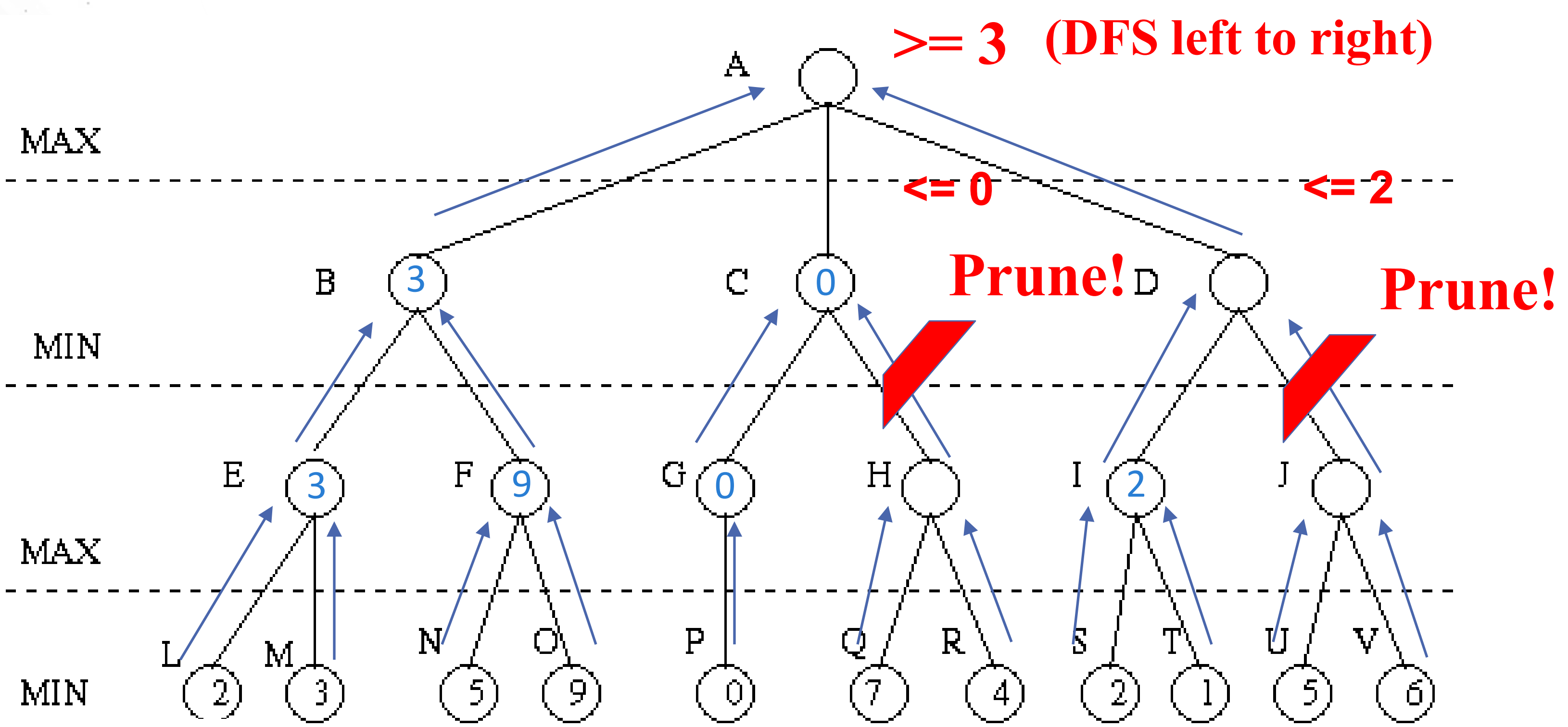


At each node:
Alpha = at least, Beta = at most

Alpha-Beta Pruning: Reordered subtree

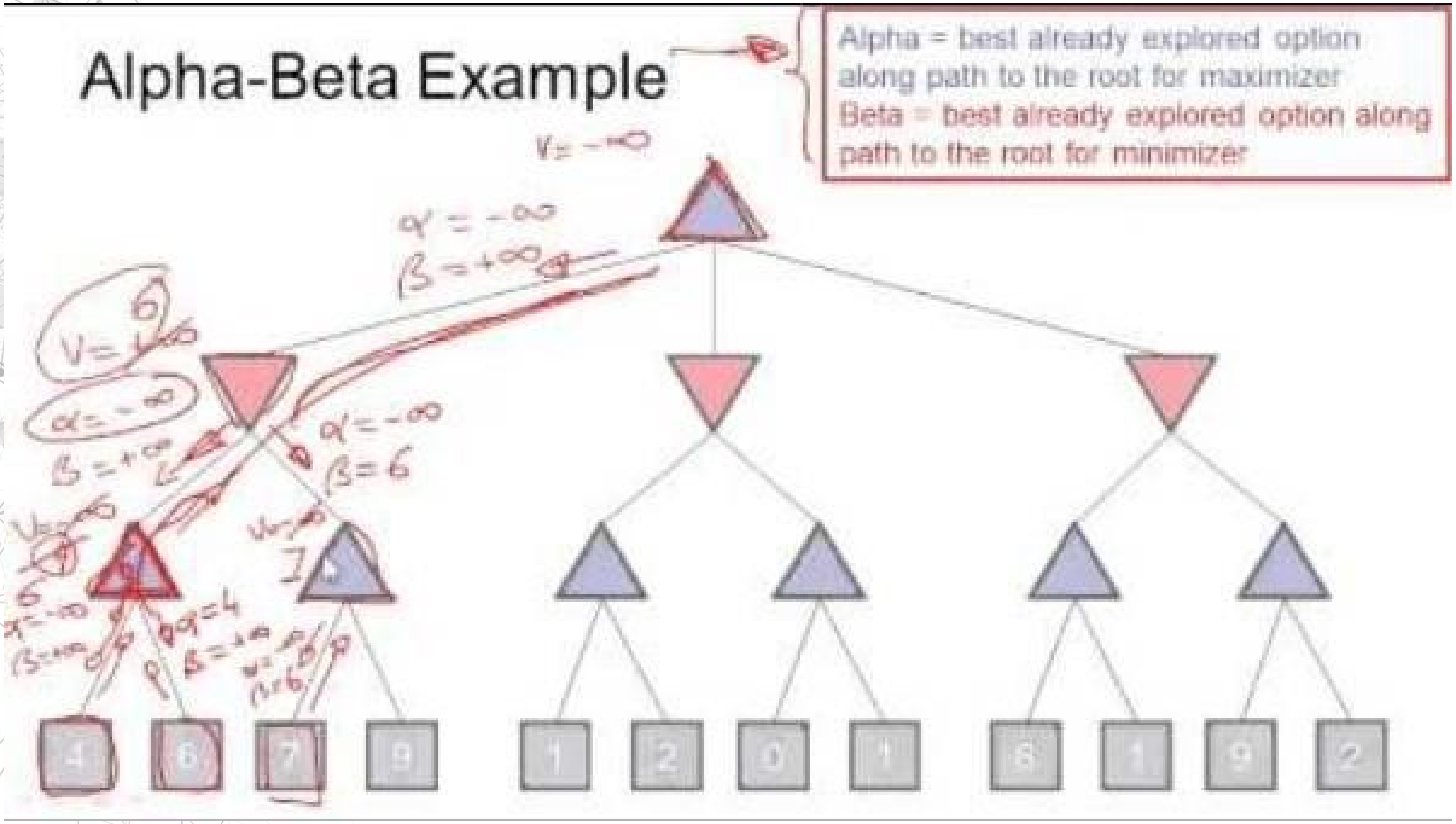


Minimax Algorithm with α-β Pruning



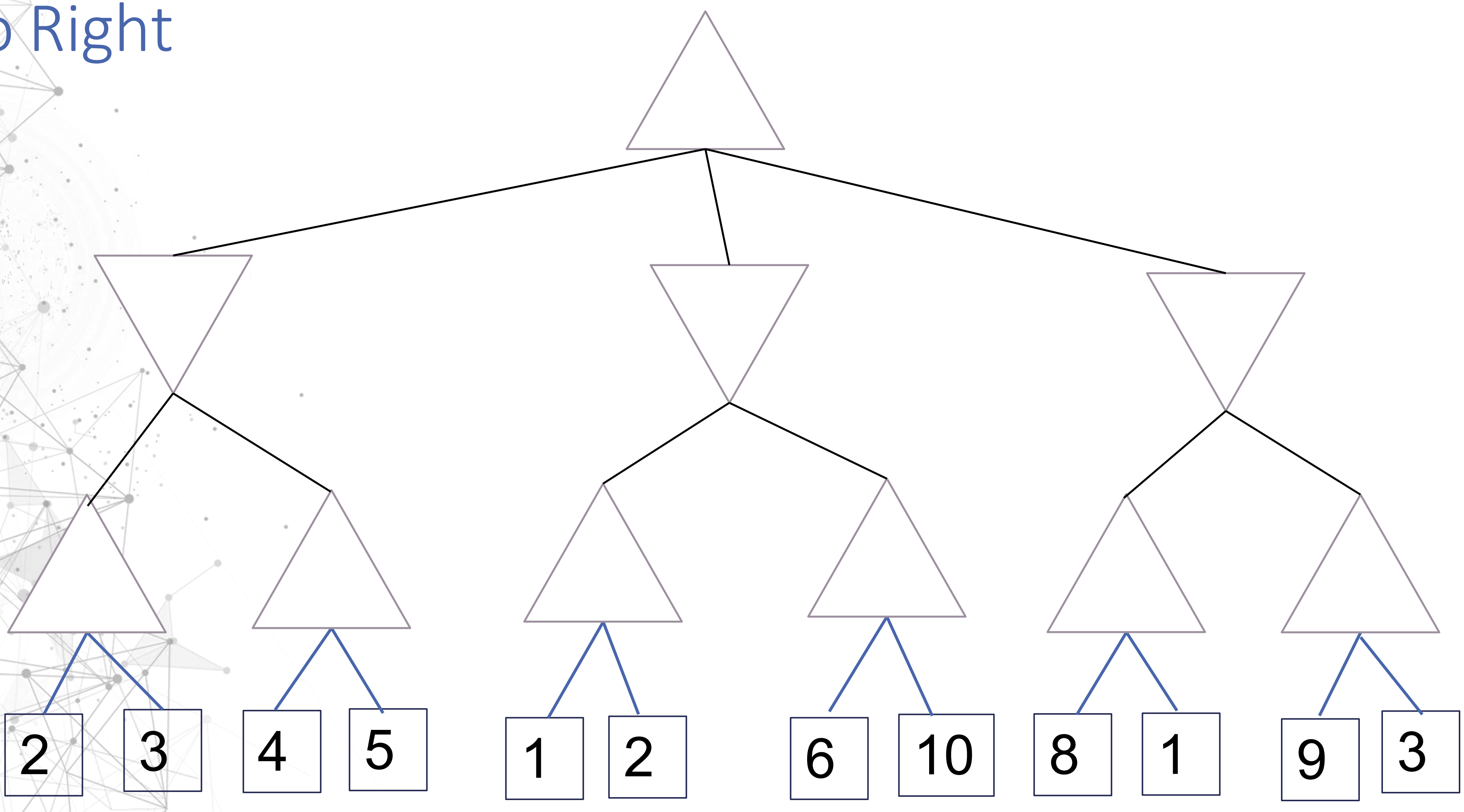
Payoff for Max

Berkeley Video



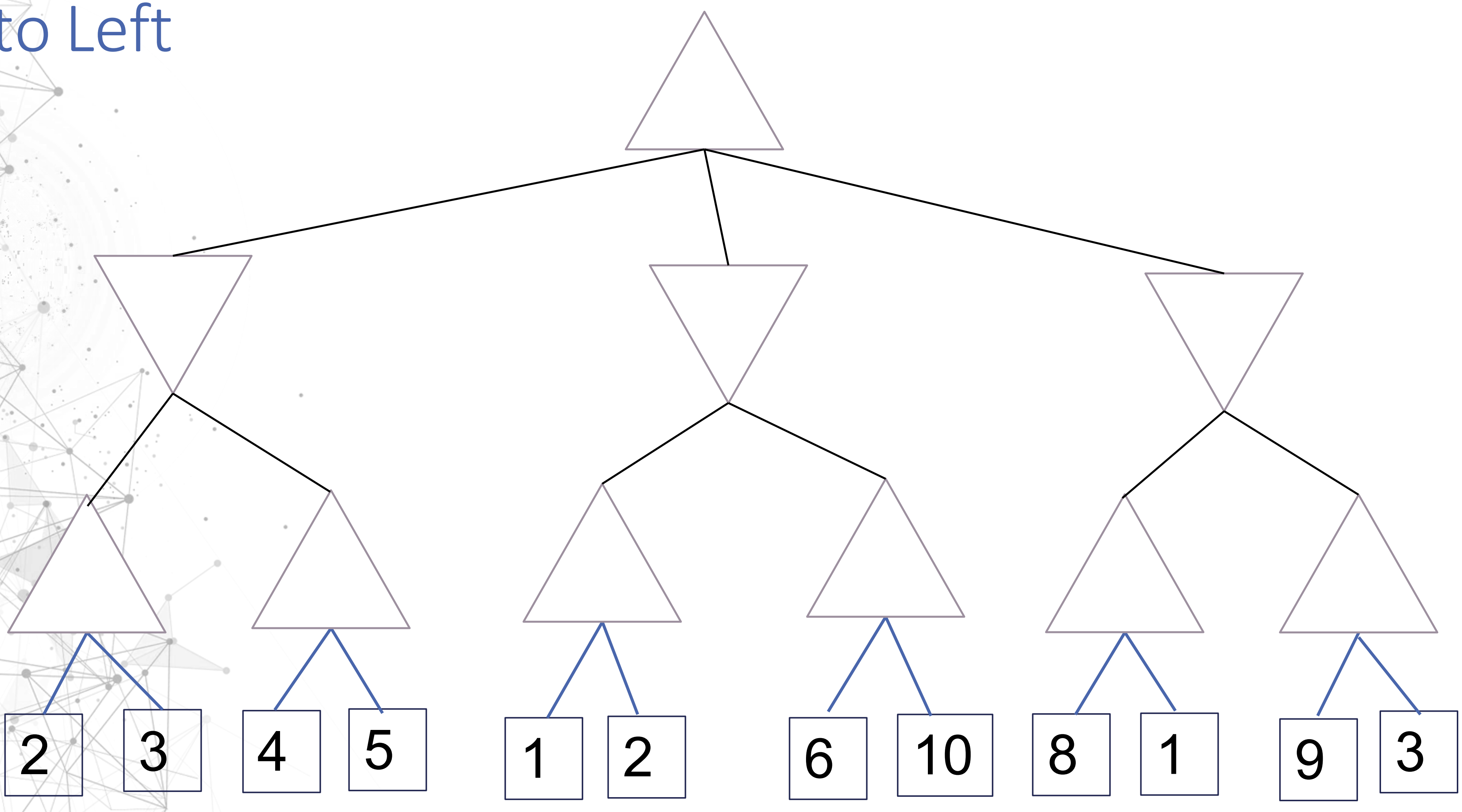
Alpha-Beta Trials … 1

Left to Right



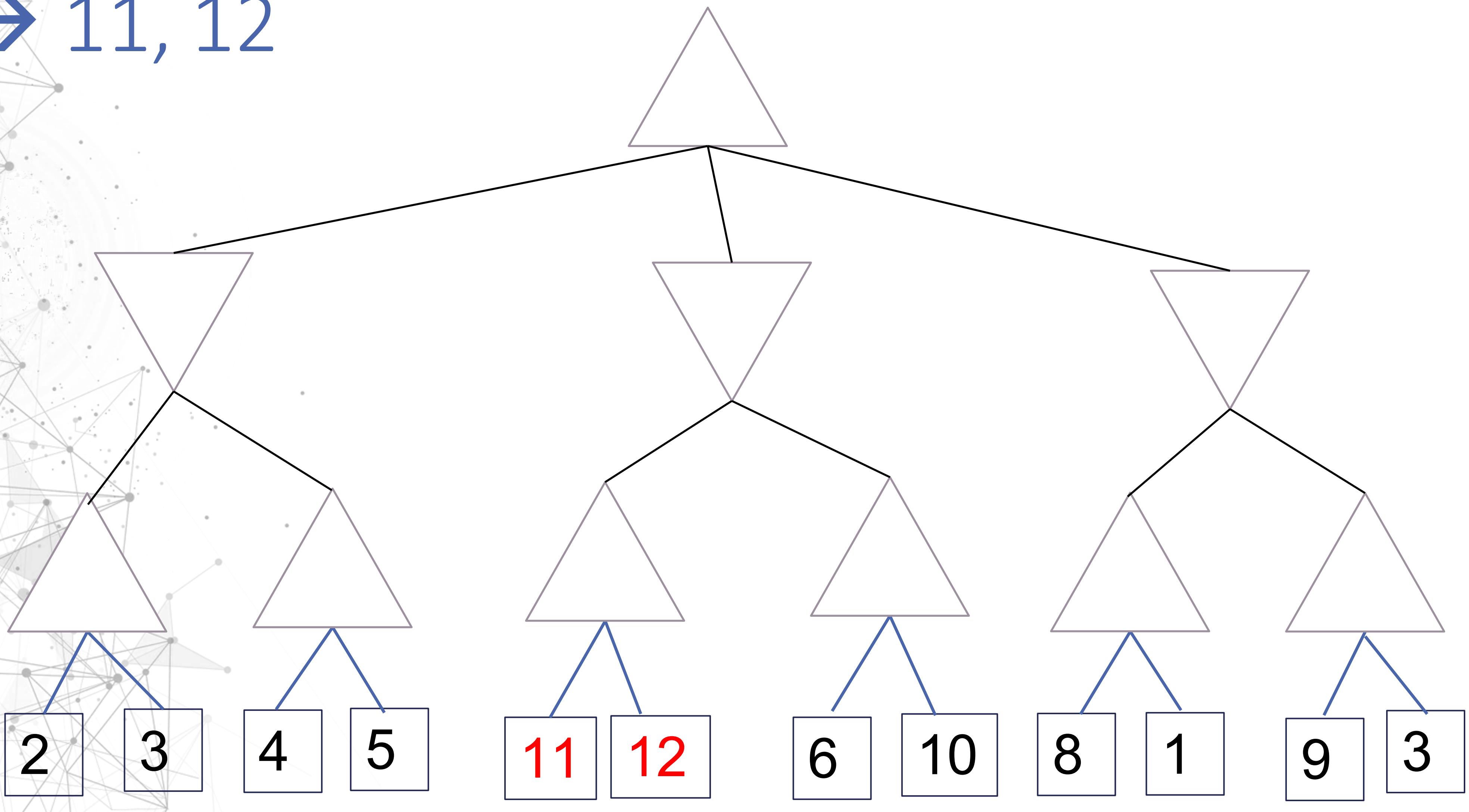
Alpha-Beta Trials … 2

Right to Left

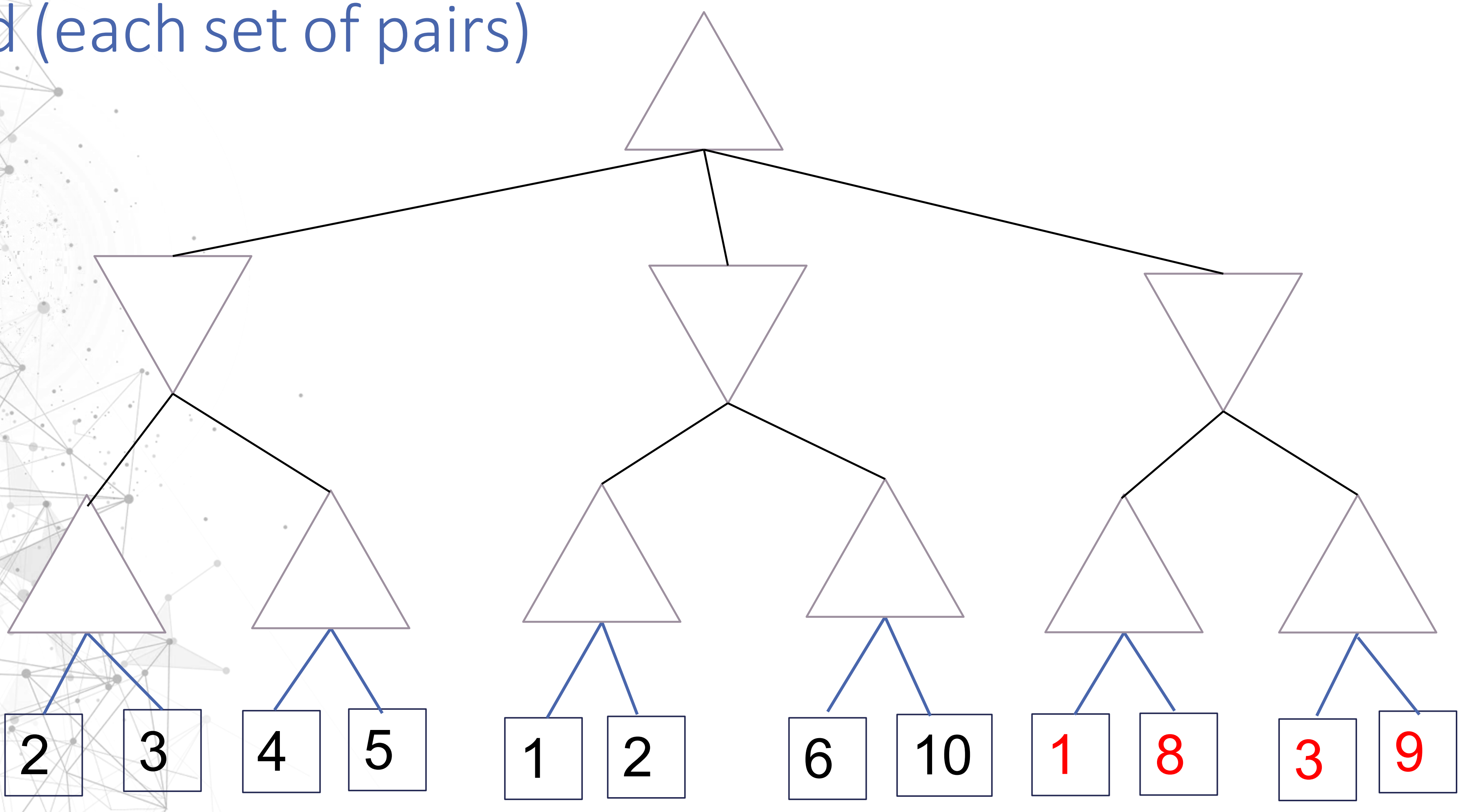


Alpha-Beta Trials … 3

1,2 → 11, 12

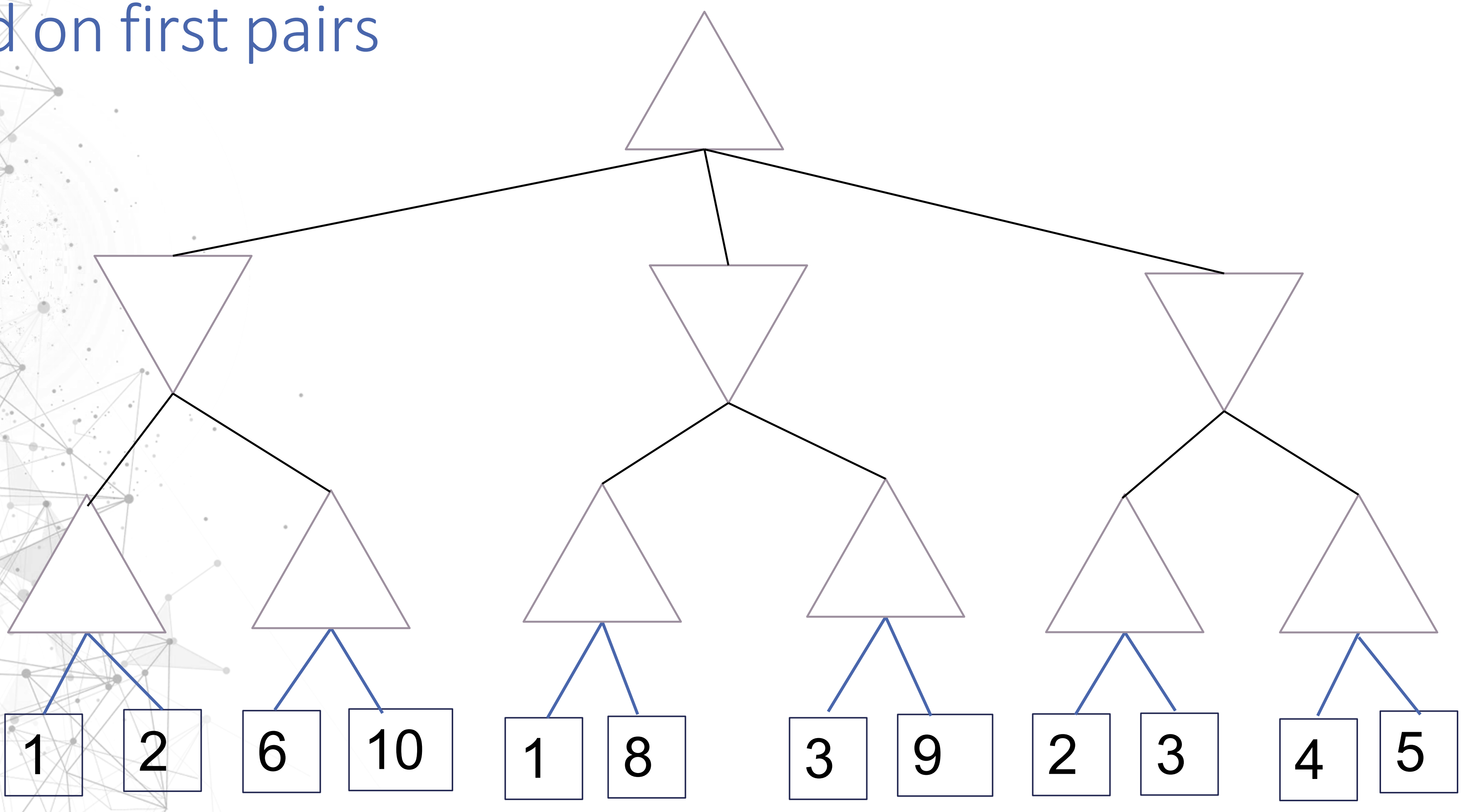


Alpha-Beta Trials … 4
Sorted (each set of pairs)



Alpha-Beta Trials … 5

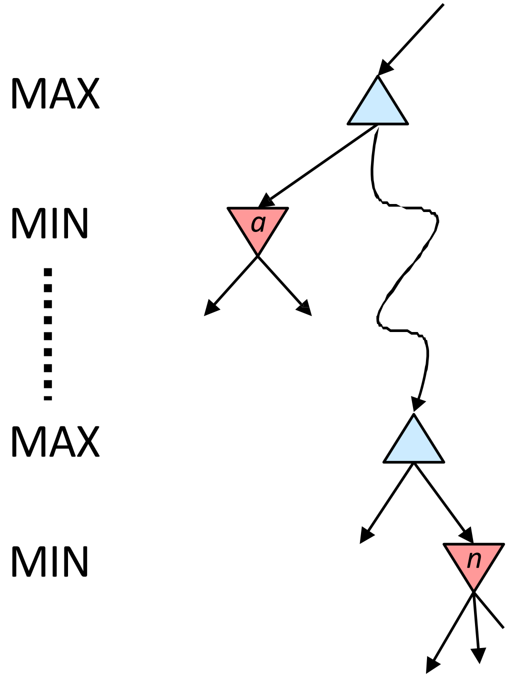
Sorted on first pairs



Alpha-Beta Pruning

General configuration (MIN version)

- We're computing the MIN-VALUE at some node *n*
- We're looping over *n*'s children
- *n*'s estimate of the children's min is dropping
- Who cares about *n*'s value? MAX
- Let *a* be the best value that MAX can get at any choice point along the current path from the root
- If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)



MAX version is symmetric

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Pruning Properties

1. This pruning has no effect on minimax value computed for the root!

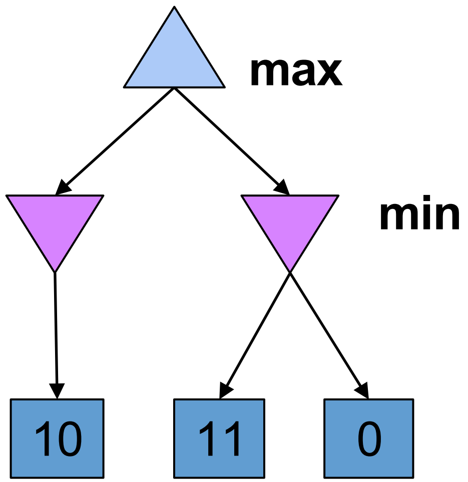
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection

2. Good child ordering improves effectiveness of pruning

With "perfect ordering":

- Time complexity drops to $O(b^{m/2})$
- Doubles solvable depth!
- Full search of, e.g. chess, is still not feasible/sensible…

➔ Simple example of metareasoning (computing about what to compute)



A.L. Samuel/Alpha-Beta 1959!

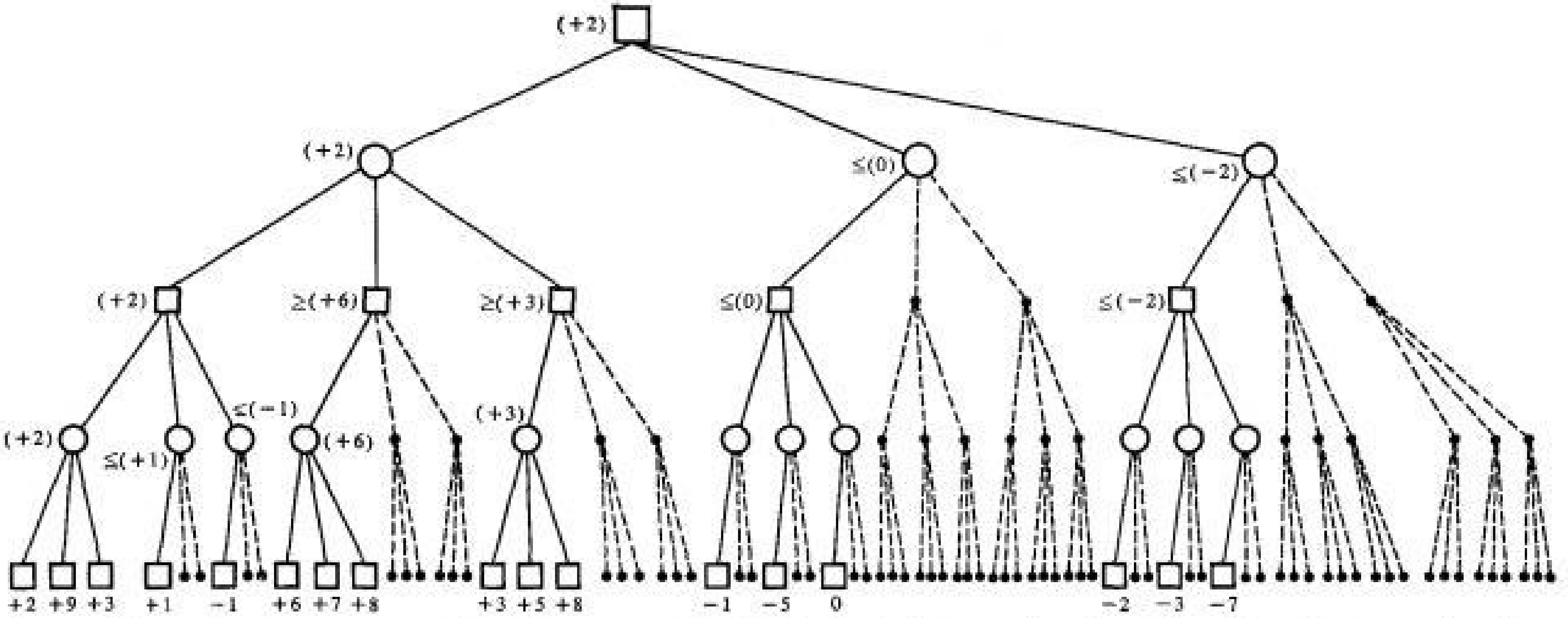
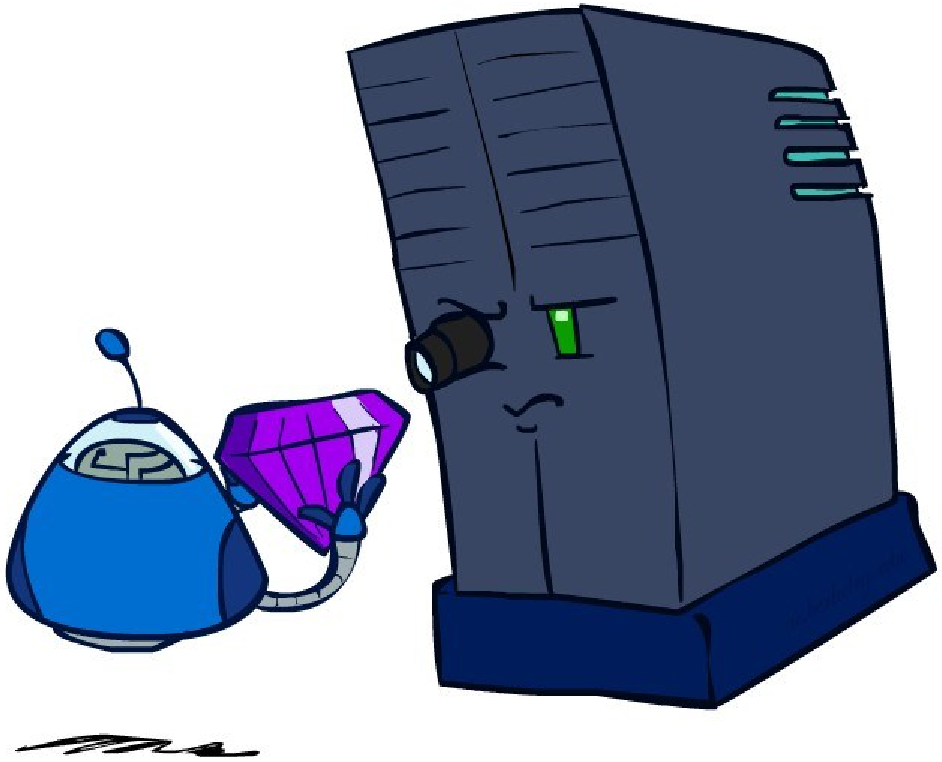


Figure 1 A (look-ahead) move tree in which alpha-beta pruning is fully effective if the tree is explored from left to right. Board positions for a look-ahead move by the first player are shown by squares, while board positions for the second player are shown by circles. The branches shown by dashed lines can be left unexplored without in any way influencing the final move choice.

A.L. Samuel, 1959: *Some Studies in Machine Learning Using the Game of Checkers* , IBM Jnl, Vol. 3, pp 211-229.

Imperfect Real-Time Decisions & Evaluation Functions



Can't always search all the way to the leaf nodes

You can't just hold up a game till you explore the depths of the tree.

In such cases we may have to make imperfect decisions in real-time.

Basic Idea

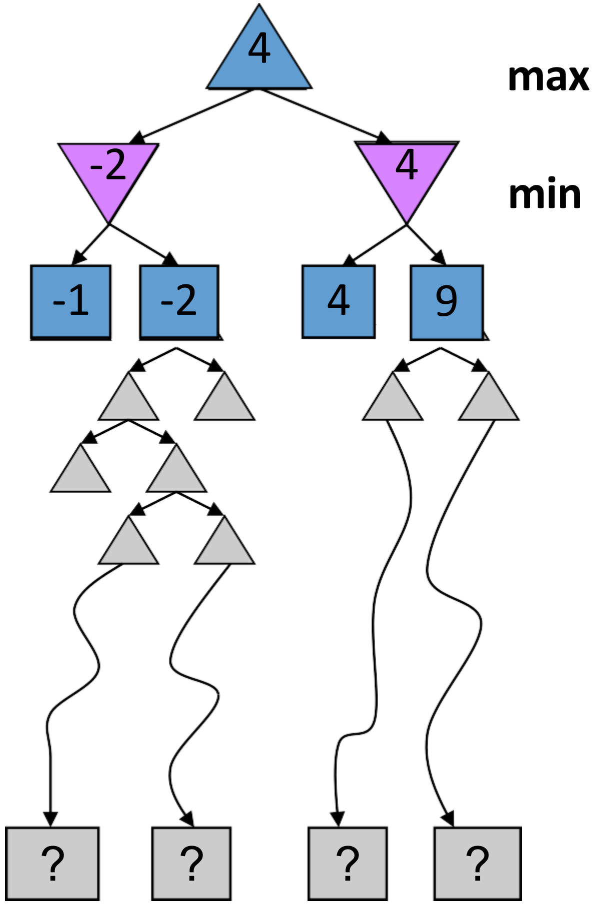
Change mimimax or alpha-beta

1. Replace utility function with a heuristic evaluation function EVAL
   which **estimates** position's utility

2. Replace terminal test by a cutoff test
   which decides when to apply EVAL,
   e.g., when depth = some depth d, or if terminal state

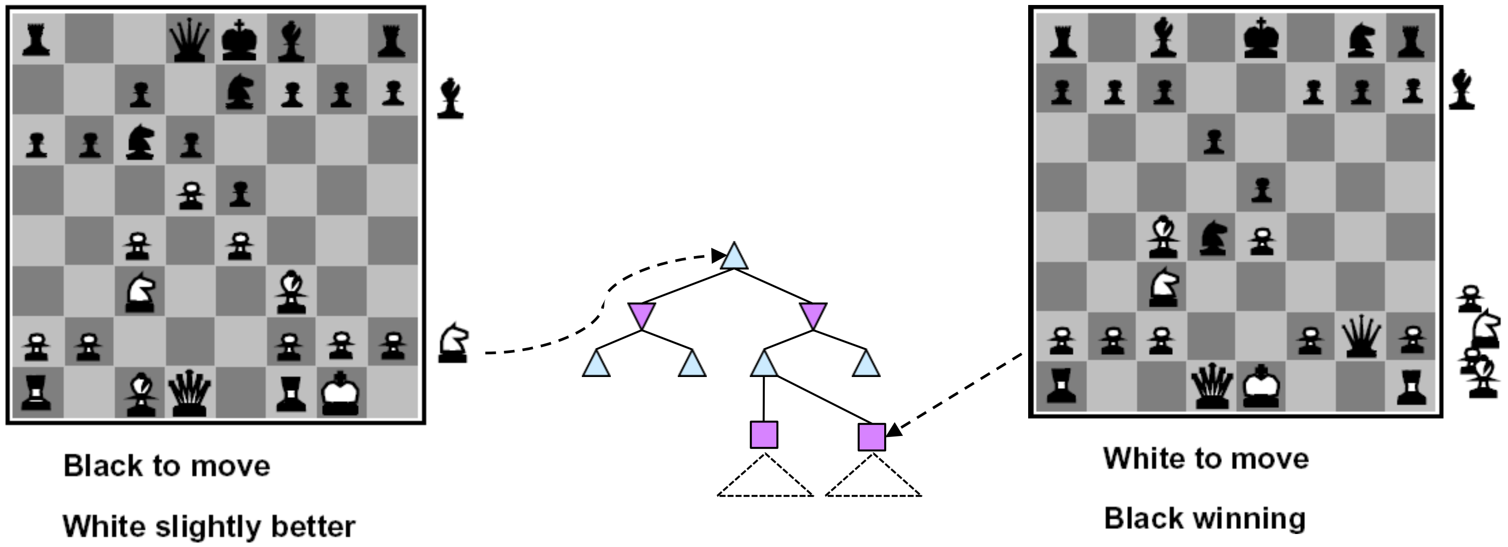
Resource Limits

- **Problem**: In real games, cannot search to leaves!
- **Solution**: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - At that depth, replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, and we can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference



Evaluation Functions

Evaluation functions score non-terminals in depth-limited search



EVAL function properties

Ideal function: returns the actual minimax value of the position

In practice, the function:

- Must emulate real utility in ordering terminal states
- Must be fast to compute
- For non-terminal states, must be strongly correlated with chances of winning

EVAL function

In practice: typically, a weighted linear sum of features:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g. $f_1(s)$ = (#white pawns – #black pawns) + (…) …

or (difference in piece-count) + (…)

etc.

Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

Stochastic Games: Dealing with chance

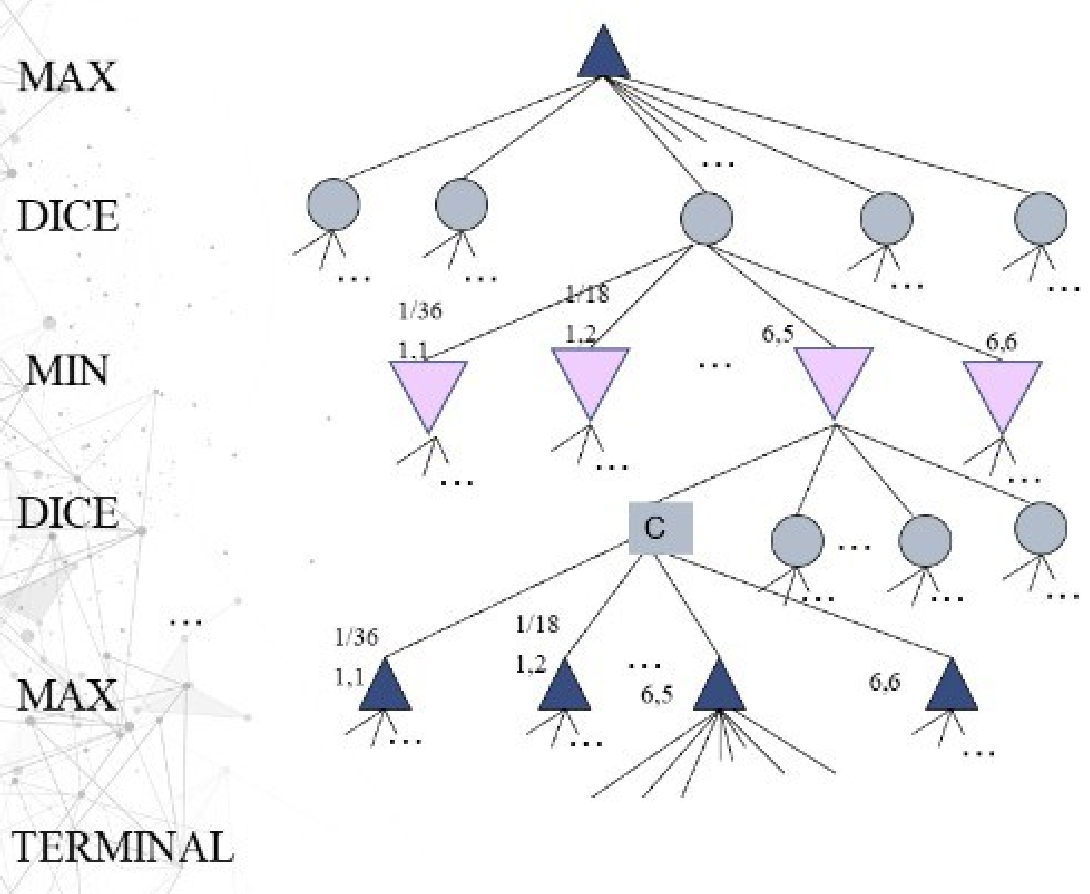
Expectiminimax .. 1

- Generalization of Minimax for games involving chance
- Includes chance nodes between
  MAX and MIN nodes
- MAX and MIN nodes determined as earlier
- Chance nodes evaluated as “expected value” (weighted average over all possible dice rolls or chance events)

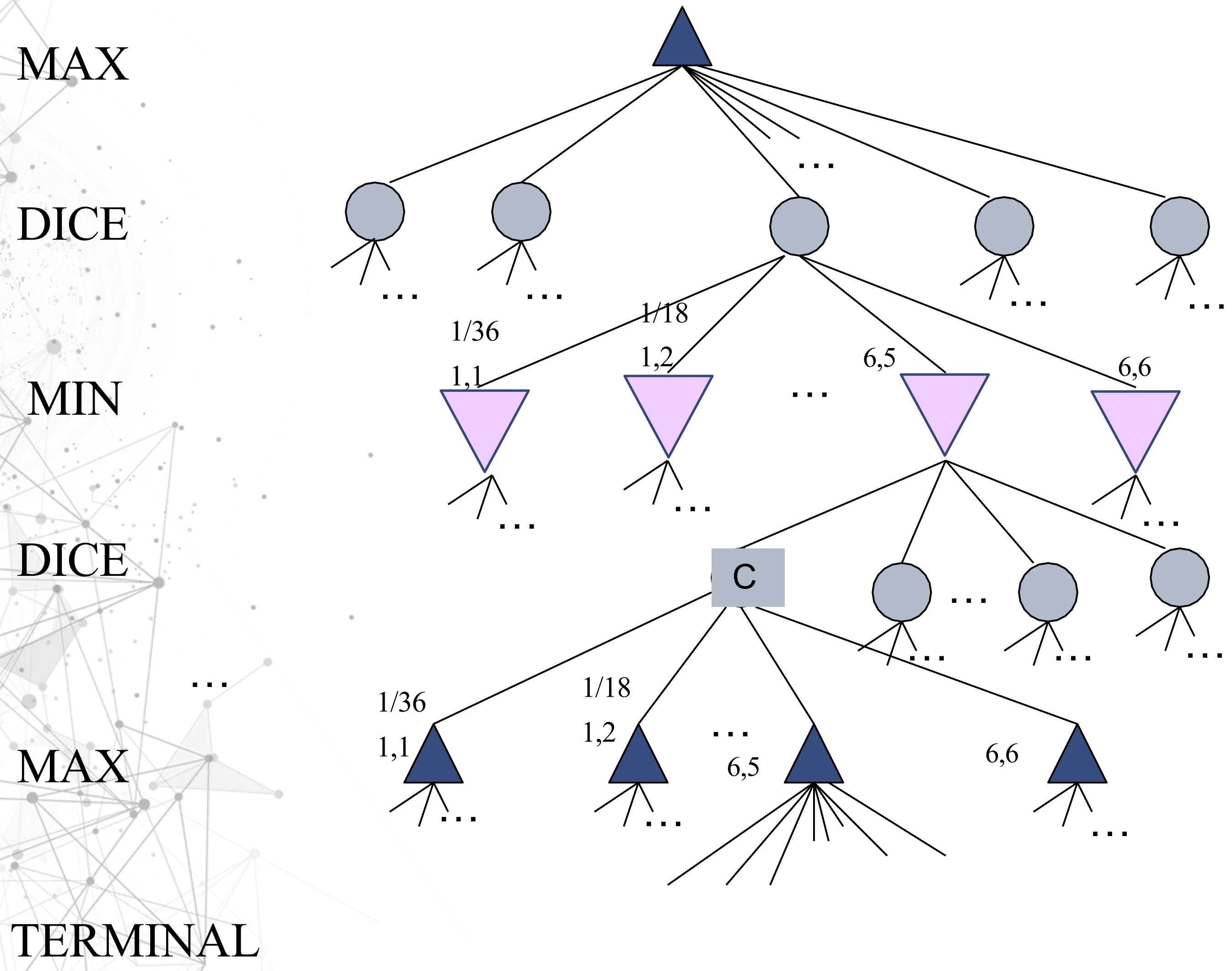
Expectiminimax .. 2

- Includes chance (DICE) nodes between MAX and MIN nodes
- MAX and MIN nodes determined as earlier
- Chance nodes evaluated as "expected value" (weighted average over all possible dice rolls or chance events)

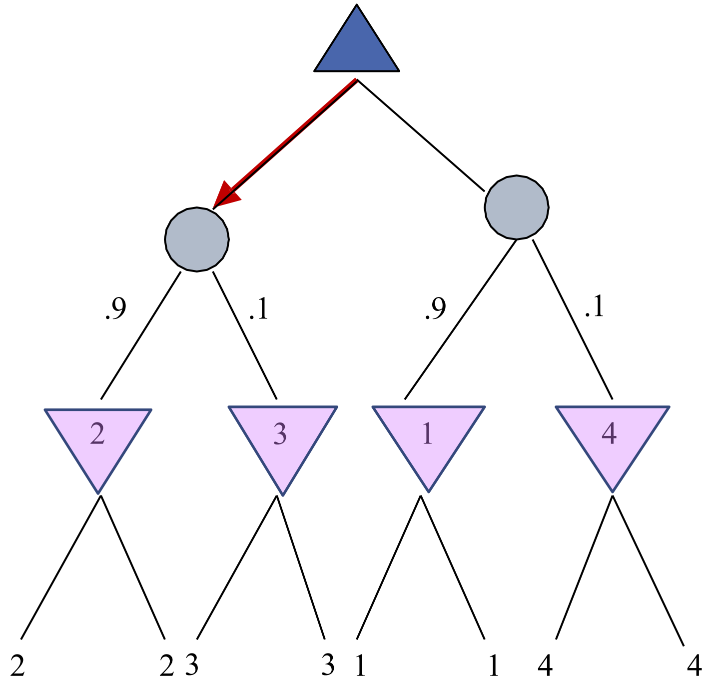
- Branches to chance nodes labeled with probability
  - 6 * 6 = 36 combinations of dice values, but only 21 distinct
    - 6 doubles at probability 1/36 each, 15 other combinations at 1/18 each



Backgammon Game Tree



Computing "Expected Value"



2 * 0.9 + 3 * 0.1 = 1.8 + 0.3 = 2.1
1 * 0.9 + 4 * 0.1 = 0.9 + 0.4 = 1.3

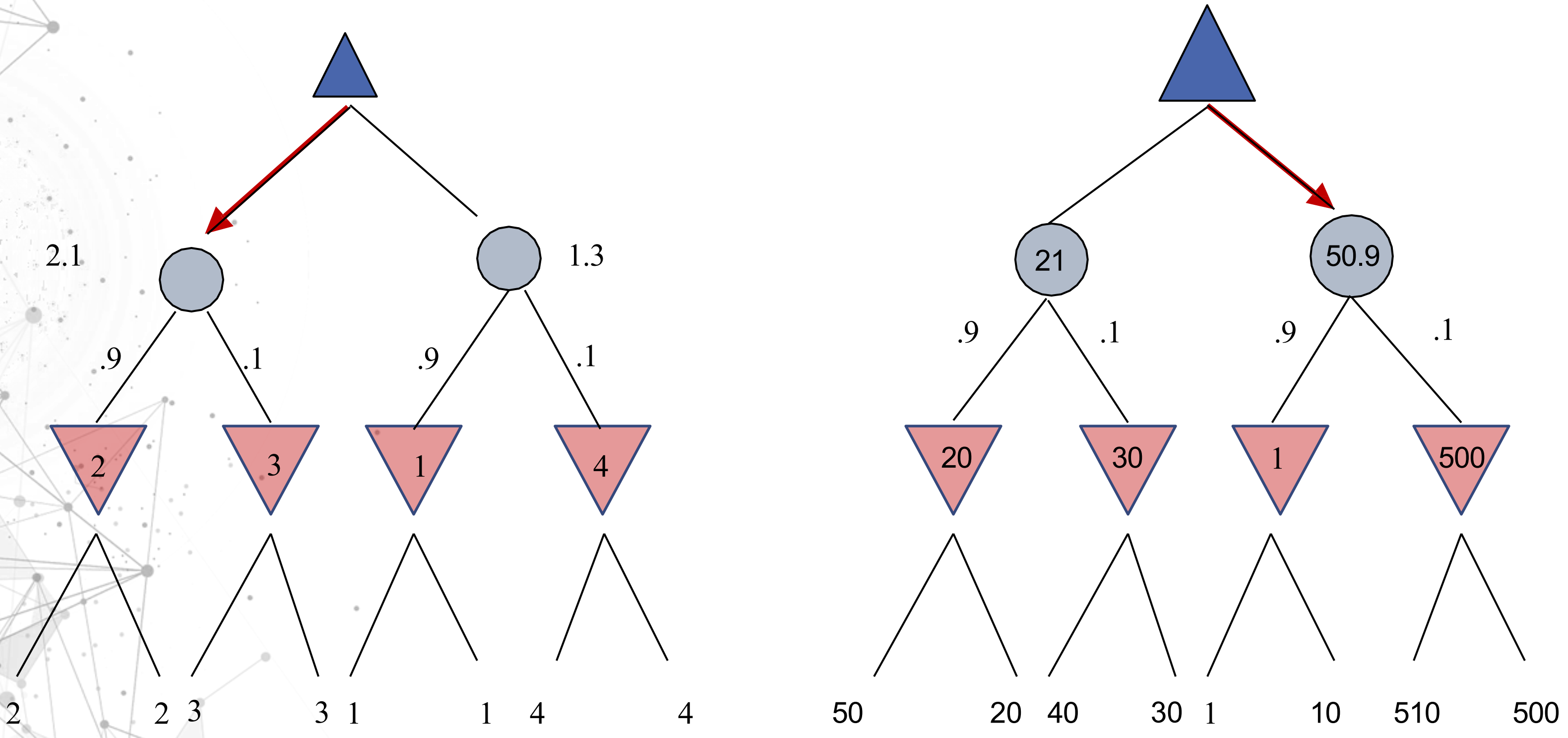
Expectiminimax

Expectiminimax(n) =

Utility(n)	for n, a terminal state
$max_{s \in Succ(n)}\ \mathrm{expectiminimax}(\boldsymbol{s})$	for n, a Max node
$min_{s \in Succ(n)}\ \mathrm{expectiminimax}(\boldsymbol{s})$	for n, a Min node
$\Sigma_{\boldsymbol{s} \in \boldsymbol{Succ}(\boldsymbol{n})} \boldsymbol{P}(\boldsymbol{s}) * \mathrm{expectiminimax}(\boldsymbol{s})$	for n, a chance node

Variations of Expectiminimax possible.

Evaluation functions for Expectiminimax



Very different behavior with values at different scales, emphasizing high payoff

Evaluation functions for games of chance

- Again, cut off search at some level and apply evaluation function to each leaf node
- Dice rolls make things different

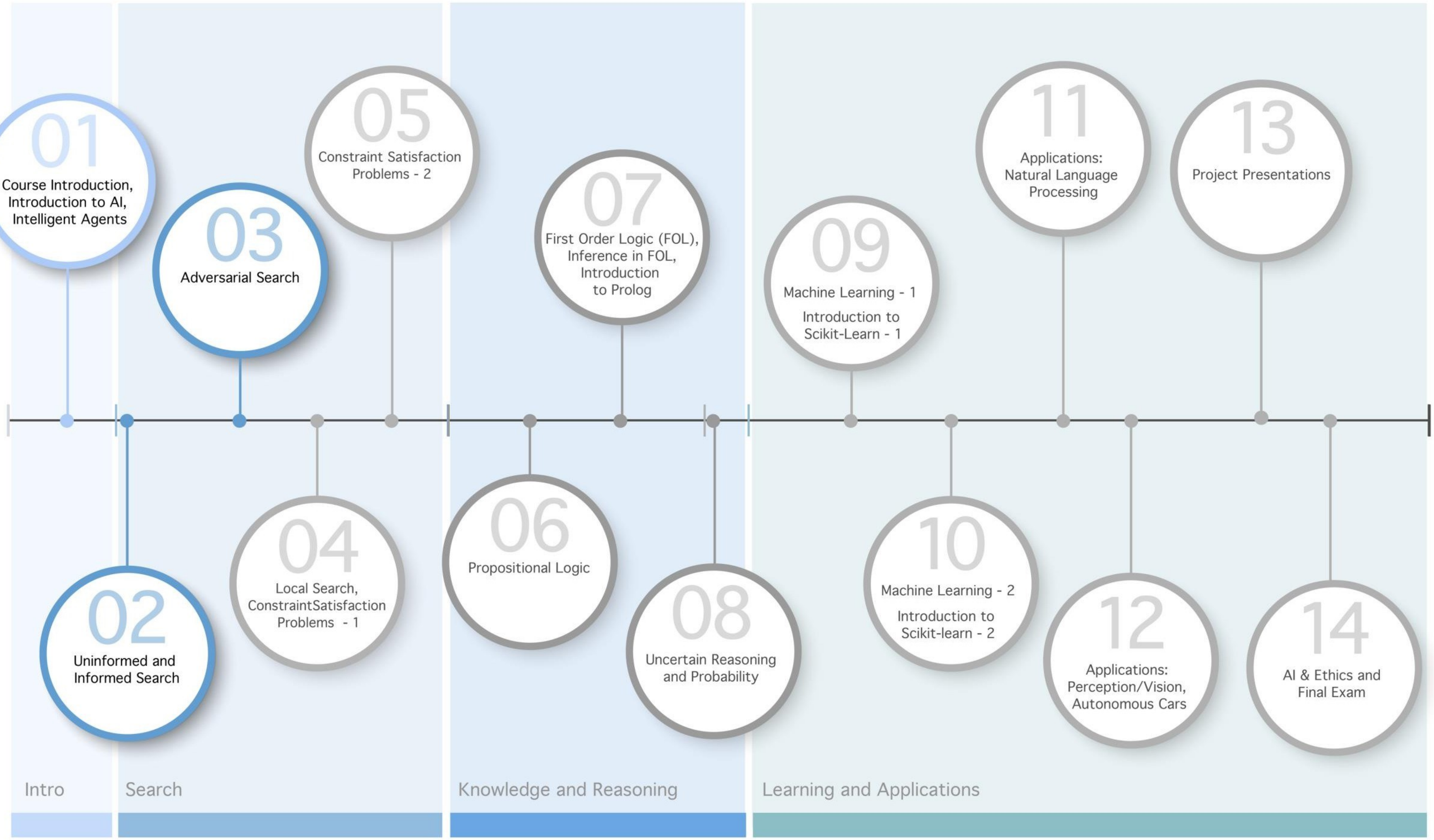
- Minimax b^m time

- With chance nodes, need to consider all possible dice rolls sequences,
  $b^m * N^m$ where N = # of distinct dice

Card Games

- Bridge, Poker etc. stochastic and partially observable
- Too many possible deals to consider

- Use Monte Carlo approximation, take random sample of N deals, compute best outcome
- Bidding adds even more complexity


01
Course Introduction, Introduction to AI, Intelligent Agents
02
Uninformed and Informed Search
03
Adversarial Search
04
Local Search, ConstraintSatisfaction Problems - 1
05
Constraint Satisfaction Problems - 2
06
Propositional Logic
07
First Order Logic (FOL), Inference in FOL, Introduction to Prolog
08
Uncertain Reasoning and Probability
09
Machine Learning - 1
Introduction to Scikit-Learn - 1
10
Machine Learning - 2
Introduction to Scikit-learn - 2
11
Applications: Natural Language Processing
12
Applications: Perception/Vision, Autonomous Cars
13
Project Presentations
14
AI & Ethics and Final Exam
Intro
Search
Knowledge and Reasoning
Learning and Applications