Function Approximation of State Spaces

- Q-Learning collects Q-Values for all explored state-action pairs (s,a) => Q-Learning maintains a Q-table
- Is the state of observation the state space for making decision?
  - state spaces are often exponential in the number of variables
  - similar states usually require similar actions
- basic Q-Learning does not generalize from observations to states

Idea: Function Approximation

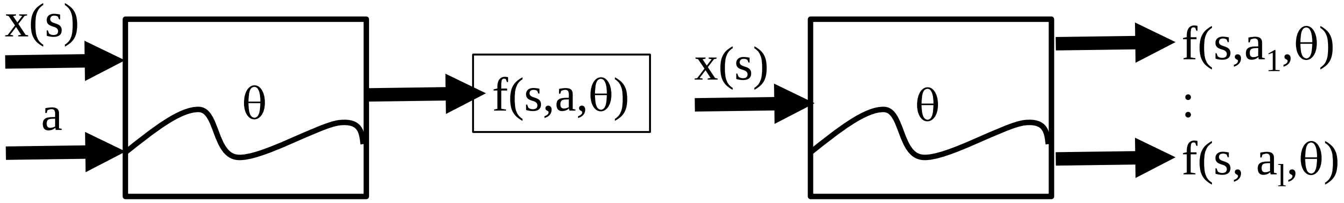
Treat the set of states as a (continuous) vector of factors and learn a regression function $f(s,a,\theta)$ predicting $Q^*(s,a)$.

Q-value function approximation

Given: A mapping *x(s)* describing *s* in IR^d.

Goal: Learn a function $f(x(s),a,\theta)$ predicting the true Q-value Q*(s,a) for any value of x(s).

- similar to supervised learning, but not exactly:
  - Where to put the action *a* in our prediction function?



  - Samples from the same trajectory are not independent and identical distributed (IID)
  - true Q*(s,a) is not known for training

    => targets are constantly changing

Learning using Function Approximation

- we want to learn a function f(x(s),a,θ) over the state-action space by optimizing the function parameters θ.

$$f(x(s), a, \theta) \approx Q^{*}(s, a)$$

- to learn f we need a loss function, e.g. MSE between $f(s, a, \theta)$ and observed values Q*(s,a).

$$L(\theta) = E\left[\left(Q^{*}(s, a) - f(x(s), a, \theta)\right)^{2}\right]$$

- optimization using stochastic gradient descent

$$-\frac{1}{2}\nabla L(\theta) = \left(Q^{*}(s, a) - f(x(s), a, \theta)\right)\nabla_{\theta} f(x(s), a, \theta)$$

$$\Delta\theta = \alpha\left(Q^{*}(s, a) - f(x(s), a, \theta)\right)\nabla_{\theta} f(x(s), a, \theta)$$

- update: $\theta \leftarrow \theta + \Delta\theta$

Linear Prediction Functions

A simple function approximation might be linear

- Linear Functions over $s \in IR^d$:

$$f(x(s), a, \mathrm{W}) = \mathrm{x(s)}^{\mathrm{T}}\mathrm{W} = \sum_{j=1}^{n} x(s)_j^T w_j$$

- Loss function:

$$L(W) = E[(Q^*(s,a) - \mathrm{x(s)}^{\mathrm{T}}\mathrm{W})^2]$$

- Stochastic Gradient Descent on L(w):

$$\nabla W\ f(x(s), a, \mathrm{W}) = \mathrm{x(s)}^{\mathrm{T}}$$

$$-\frac{1}{2}\nabla L(\theta) = \big(Q^*(s,a) - f(x(s), a, \theta)\big)x(\mathrm{s})^{\mathrm{T}}$$

$$\Delta\theta = \alpha\big(Q^*(s,a) - f(x(s), a, \theta)\big)x(\mathrm{s})^{\mathrm{T}}$$

Further Directions

- other prediction functions:
  - (deep) neural networks
  - decision trees
  - nearest neighbor
  - ...
- DQN: uses a deep neural network and works with an experience buffer to make the learning target more stable
- Policy Gradients: Uses function approximation for selecting the best action (not the Q-values)
- Actor-Critic methods: Combine value function approximation and policy gradient.

Why is AI important for Games?

Computer games are an optimal sand-box for developing AI techniques:

- games are queryable environments
- rewards and actions are known
- states are parts or views on the game state

But, why is reinforcement learning interesting for managing and mining Computer Games ?

- develop intelligent AI opponents/collaborators
- micro-management for small granularity games
- learn optimal strategies for teaching players or balancing
- mimic real behavior within a game

Imitation Learning

- use reinforcement learning to make an agent behave like a teacher (e.g. a pro gamer)
- Learning from experience: teacher provides (s,a,r,s') samples of good behavior (reward is known)
- Learning from demonstration: teacher provides (s,a,s') samples.
  - reward is not explicitly known
  - success is expected based on the reputation of the player

Challenge:

- predicting the action for states with sufficient samples is easy (policy follows the distribution of observed actions)
- predicting proper actions for undersampled states is hard.

=> approximation function must generalized from observed states to unobserved ones.

Imitation learning in Games

possible applications:

- make a player behave like a real one
  (e.g. adapt player styles for football games)
- learn policies for hard opponents to analyze their weaknesses
- when training an agent learn from human experts
  (first Alpha Go version)
- learn policies for your own behavior and find out where it deviates from the optimal policy

Note, this is an active field of research with many unsolved problems:

- policies depend on the agents/players capabilities
- capability of the imitating agent in unknown states is hard to evaluate
- reward functions might not be the same for teacher and imitating agent

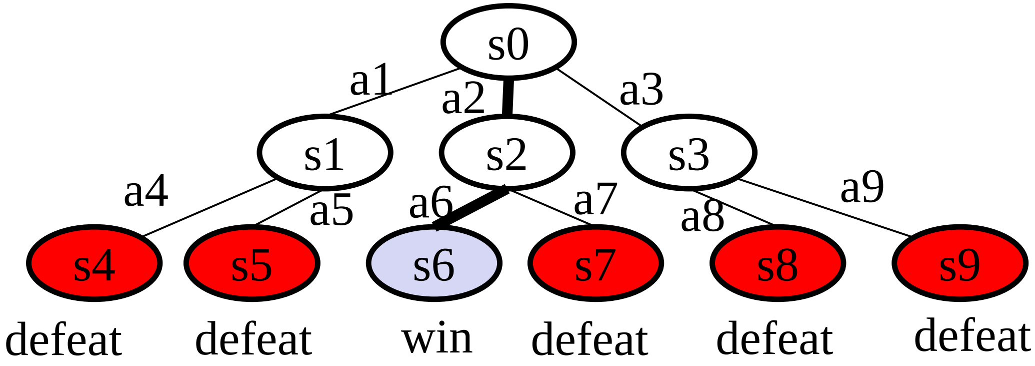
Techniques for Multiple Agents

Consider an MDP (S,A,T,R):

- often the uncertainty of state transitions T is completely caused by the actions of other independent agents (opponent or team members)

  examples: chess, GO, etc.

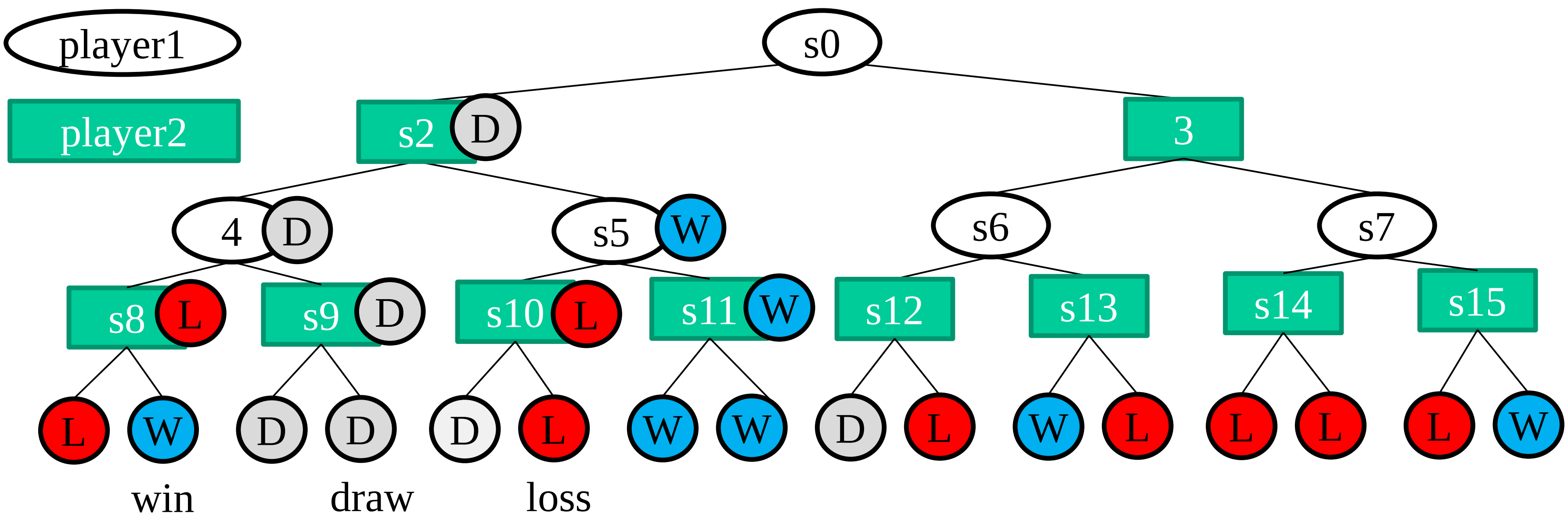
- if you would know the policy of the other agents, optimal game play could be achieved with deterministic search.



Antagonistic Search

- assume that there is a policy π^* which both player follows
- in antagonistic games, the reward of player p1 is the negative reward of player p2. (zero-sum game)

   => player1 maximizes rewards
   player2 minimizes the rewards



Antagonistic Search

- generally it is not possible to search until the game ends (search grows exponential with available actions)

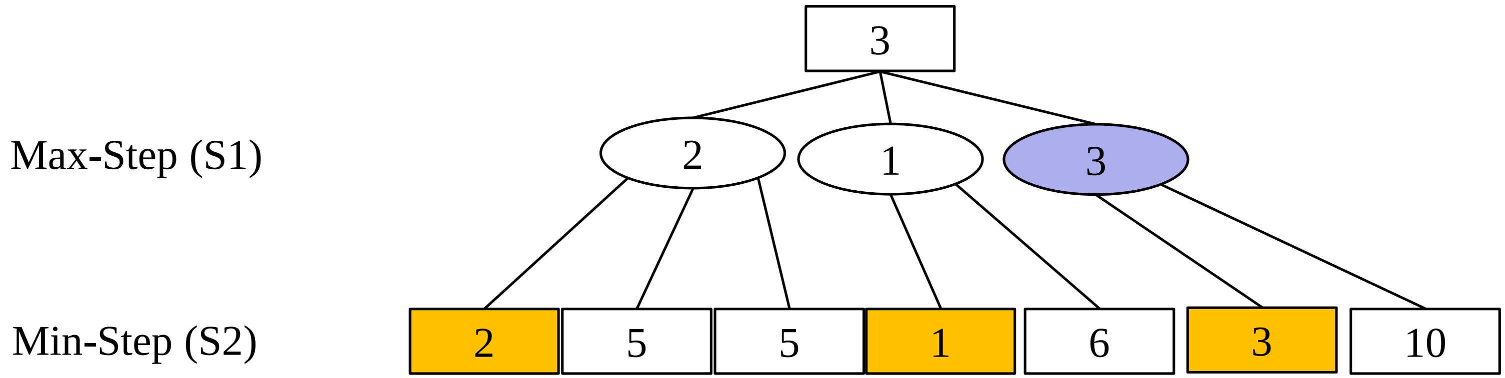
$\Rightarrow$ stop searching at a certain level and user another reward corresponding to the chance of success

Types of rewards:

- heuristics (figures, flexibility, strategic positions etc.)
- prediction functions (input game state ->win probability)
- databases (opening or end game libraries)

Min-Max Search in antagonistic Search Trees

- select action a that maximizes R(s) for S1 after S2's reaction
- Search depth:
  - Given Number of Turns
  - ⇒ Time may vary and is hard to estimate
  - ⇒ Turbulent positions make cutting of some branches unfavorable
  - Iterative Deepening:
    - Multiple calculations with increasing search depth
    - On Time-Out: Abort and use of last complete calculation
      (since expense doubles on average, double the expense can be estimated)
  - turbulent positions: single branches are being expanded if leaves are turbulent.



Alpha-Beta Pruning

Idea: If a move already exists, that can be valuated with even after a counter reaction, all branches creating a value less than can be cut.

- : S1 reaches at least α on this sub-tree ($R(s) > \alpha$)
- : S2 reaches at most β on this sub-tree ($R(s) < \beta$)

Algorithm:

- Traverse Search-Tree with deep search and fill inner nodes on the way back to the last branching
- For calculating inner nodes:

  If $\beta < \alpha$ then
  - Cut off remaining sub-tree
  - set β-value for the sub-tree if it's root is a min-node
  - set α -value for the sub-tree if it's root is a max-node

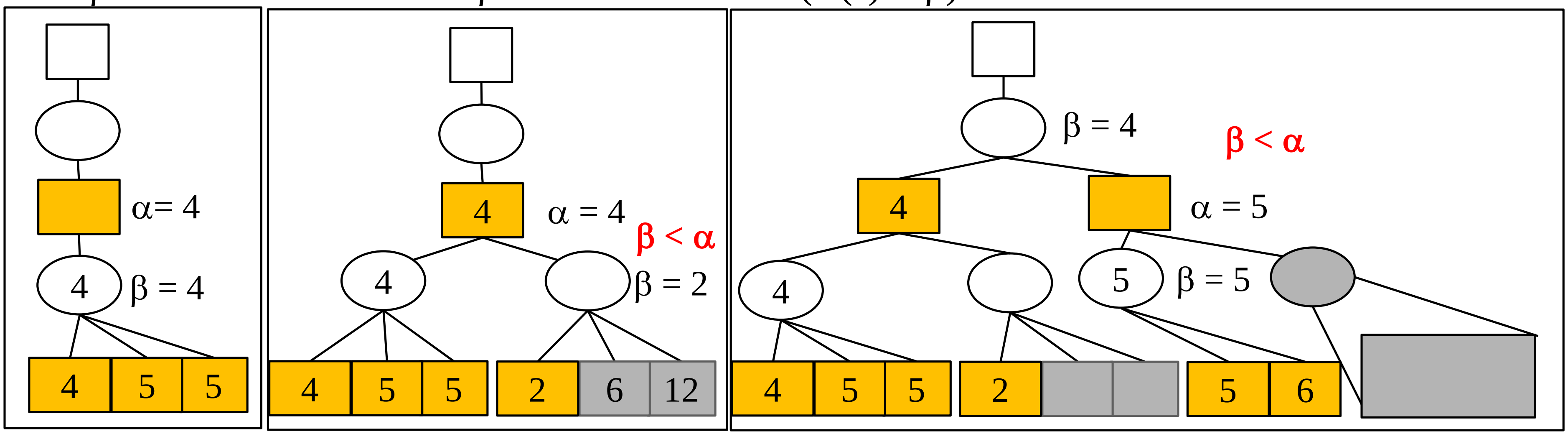
  Else set β-value to the minimum of min-nodes

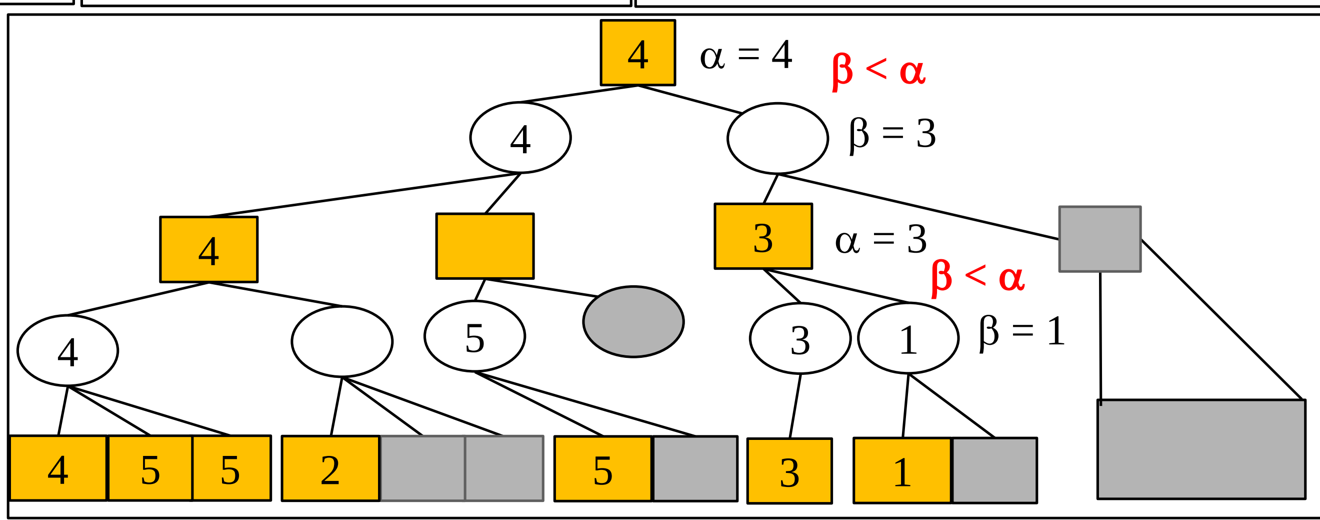
  set α-value to the maximum of max-nodes

Alpha-Beta Pruning

Idea: If a move already exists, that can be valuated with α even after a counter reaction, all branches creating a value less than α can be cut.

- α: S1 reaches at least α on this sub-tree ($R(s) > \alpha$)
- β: S2 reaches at most β on this sub-tree ($R(s) < \beta$)





Monte Carlo Tree Search

- for games with high branching factors MinMax does not scale
- heuristics are often hard determine and require expert knowledge
- machine learning depends on the available data sets (biased to human play style)

Monte Carlo Tree Search:

- samples tree based on Monte Carlo Learning of simulated play outs
- uses an exploration/exploitation scheme to systematically search the first k-layers of the search tree.
- simulation can be based on different opponent agents strategies

UBC1

- selects actions w.r.t. reasonable exploration and exploitation trade-offs
- consider a situation where you had N tries and I actions
- for each action a_i you know the number of wins and number of samples (allows to calculate mean win rate)
- based on Hoeffding's inequality, it can be shown that the following bound for mean win rate holds: $c_{n,n_i} = \sqrt{\frac{2 \ln n}{n_i}}$
- the bounds gets narrower the more samples for a_i become available, but the bounds for all actions aj (i≠j) become wider
- now always select action $a = argmax_i(\mu_i + c_{n,n_i})$

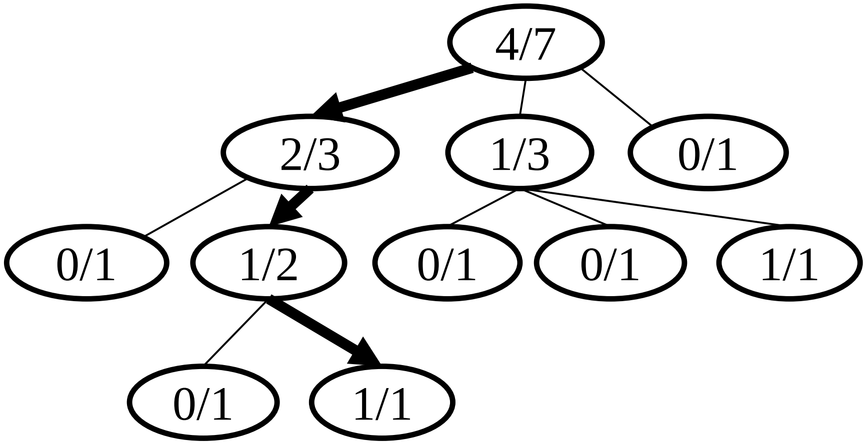
Monte Carlo Tree Search with UCT

- use UBC1 for sampling the first k levels of the search tree
- if no samples are available apply a random search or some light-weight policy.
- to evaluate leafs at the leaf level, simulate game until terminal state is reached

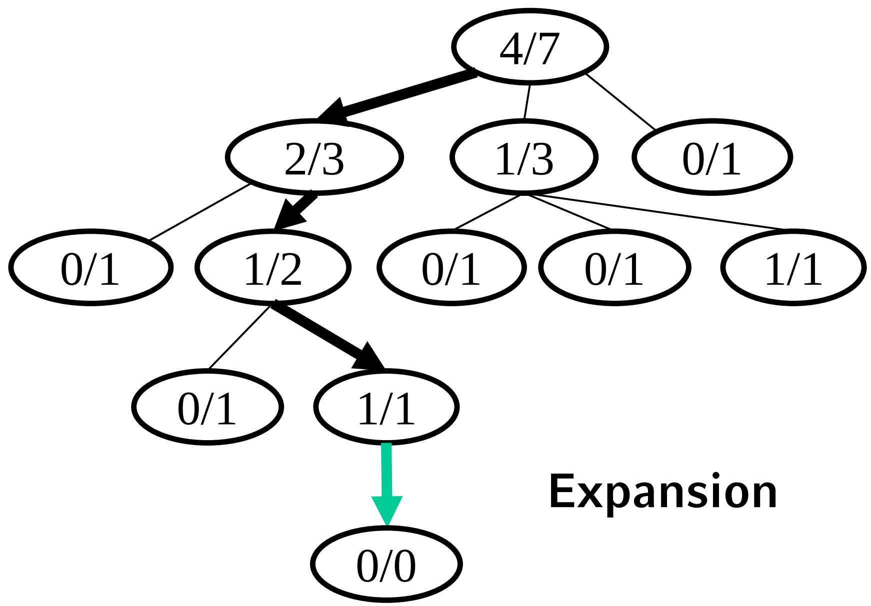
The algorithm runs in 4 phases:

- selection: search tree based on UBC1
- expansion: randomly select an action when UBC1 does not work
- simulation: simulate a further game trajectory
- backpropagation: backup the value along the path to the root

Example

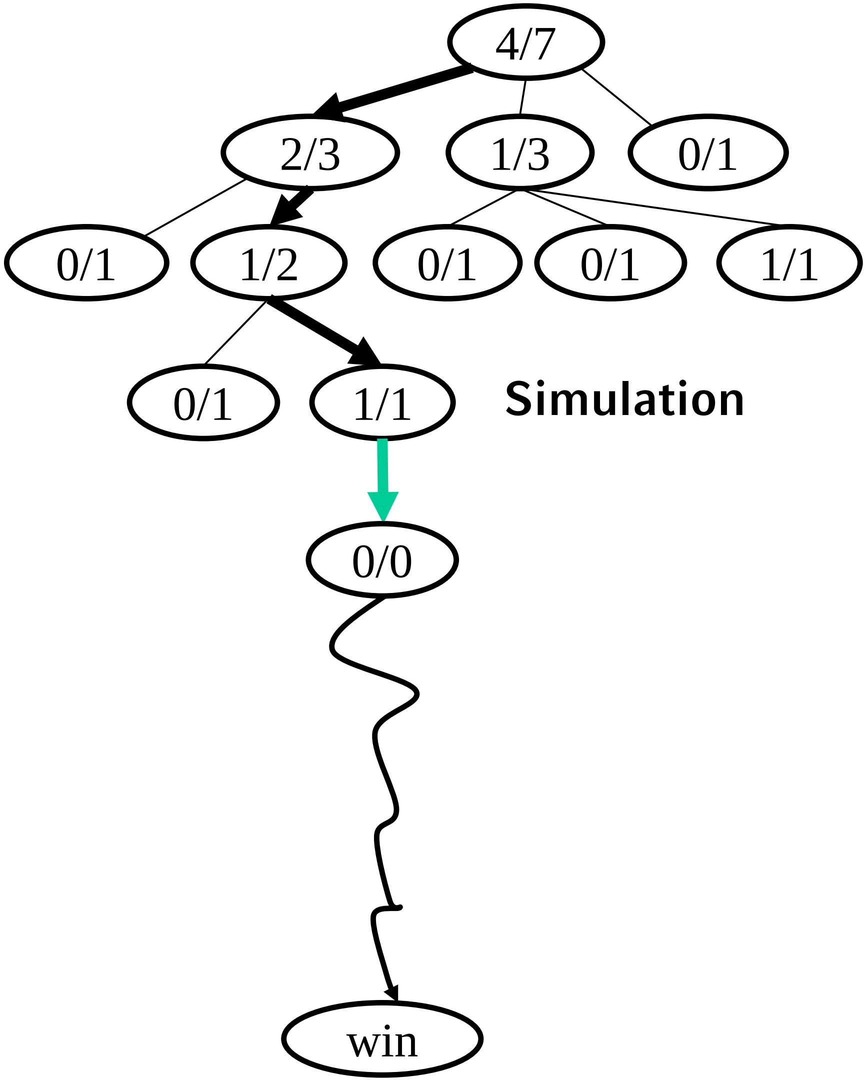


Selection

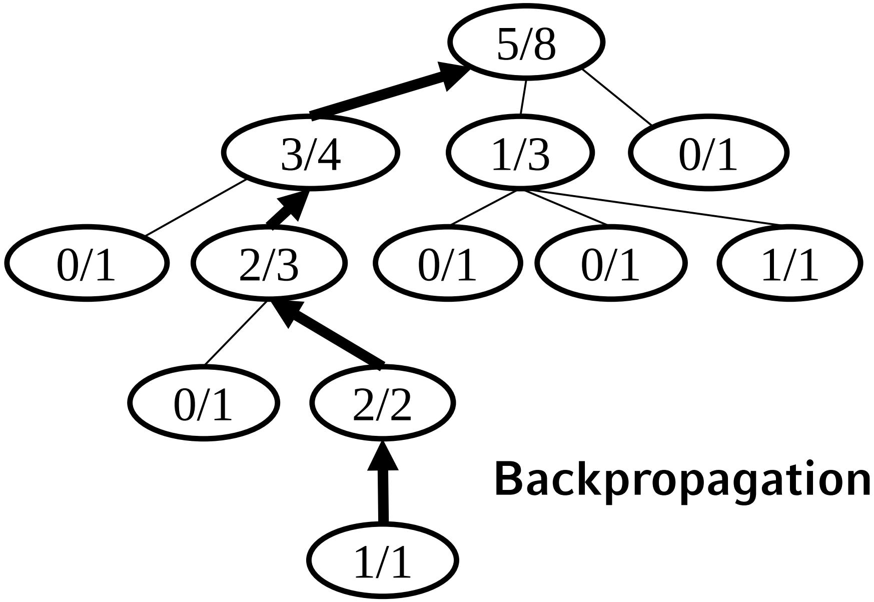


Expansion

Example



4/7
2/3
1/3
0/1
0/1
1/2
0/1
0/1
1/1
0/1
1/1
Simulation
0/0
win



5/8
3/4
1/3
0/1
0/1
2/3
0/1
0/1
1/1
0/1
2/2
Backpropagation
1/1

Monte Carlo Tree Search

- applicable to antagonistic search but not restricted to it
- can handle stochastic games and games partially observable game states
- the 4 steps can be iterated until a given time budget is spend: the longer the search is done the better is the result.
- a general question is to perform simulation to determine the possible outcomes
- Monte Carlo Tress Search is used in Alpha Go to allow lookahead together with convoluational neural networks and deep reinforcement learning

Learning Goals

- agents and environments for sequential planning
- deterministic search
- building decision graph for routing in open environments
- Markov Decision Processes
- Policy and Value Iterations
- Model-free approaches and Q-Learning
- Function Approximation
- Antagonistic Search
- MiniMax Search and Alpha-Beta Pruning
- Monte Carlo Tree Search with UCT

Literature

- Nathan R. Sturtevant: **Memory-Efficient Abstractions for Pathfinding** In Artificial Intelligence and Interactive Digital Entertainment, Conference (AIIDE), 2007.
- Lecture notes D. Silver: Introduction to Reinforcement Learning (http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html)
- S. Russel, P. Norvig: Artificial Intelligence: A modern Approach, Pearson, 3rd edition, 2016
- Levente Kocsis and Csaba Szepesvári: Bandit based monte-carlo planning. In Proceedings of the 17th European conference on Machine Learning (ECML'06), 282-293, 2006
- V. Mnih, K. Kavokcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller: Playing Atari with Deep Reinforcement Learning, NIPS-DLW 2013.