Advanced Topics on Artificial Intelligence

Alban Grastien

The Australian National University

Second Semester, 2020

Summary

- Last time:
  - Asynchronous VI: value iteration that does not backup all states all the time
  - Topological VI: use the structure of the state space to choose what to backup when
  - RTDP: Check the new video!
    Lesson:
    - Use optimistic heuristics in combination with greedy search
    - Update only the region searched
- Today:
  - Monte Carlo Tree Search
  - Some heuristics

Too Many States

- The greedy graph of a policy is still too large: billions of states or more.

- The number of states close to the starting state is smaller than the number of states close to the goal

→ Better to concentrate on these states

- Also, they pertain to the current decisions, rather than later decisions.

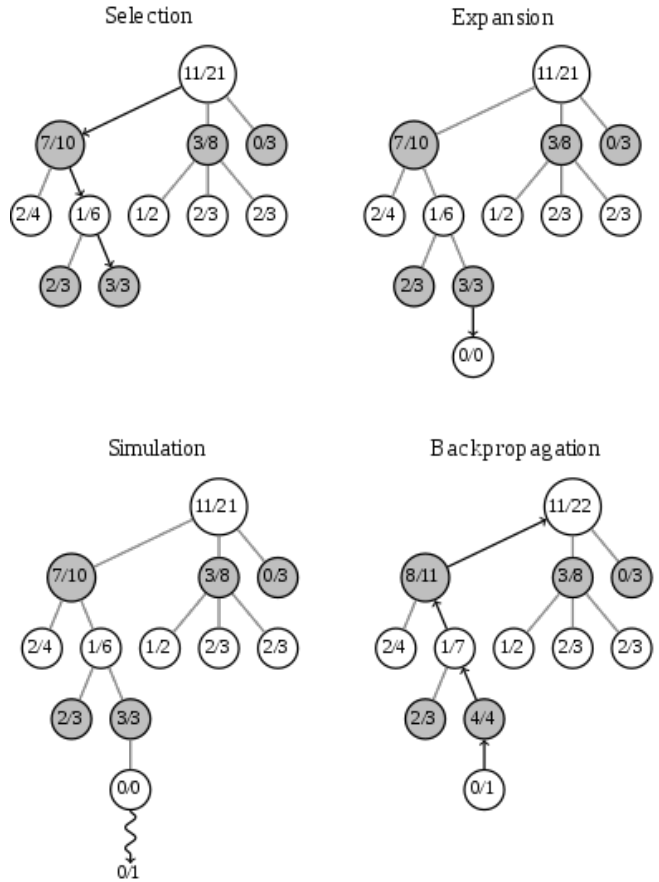
- Monte Carlo Tree Search estimates the value of states close to the initial state, and uses a greedy policy to simulate the value of other states.

Monte Carlo Tree Search

MCTS

Repeat

- Select a node
- Expand the node
- Simulate from the node
- Backpropagate



(Image from wikipedia.)

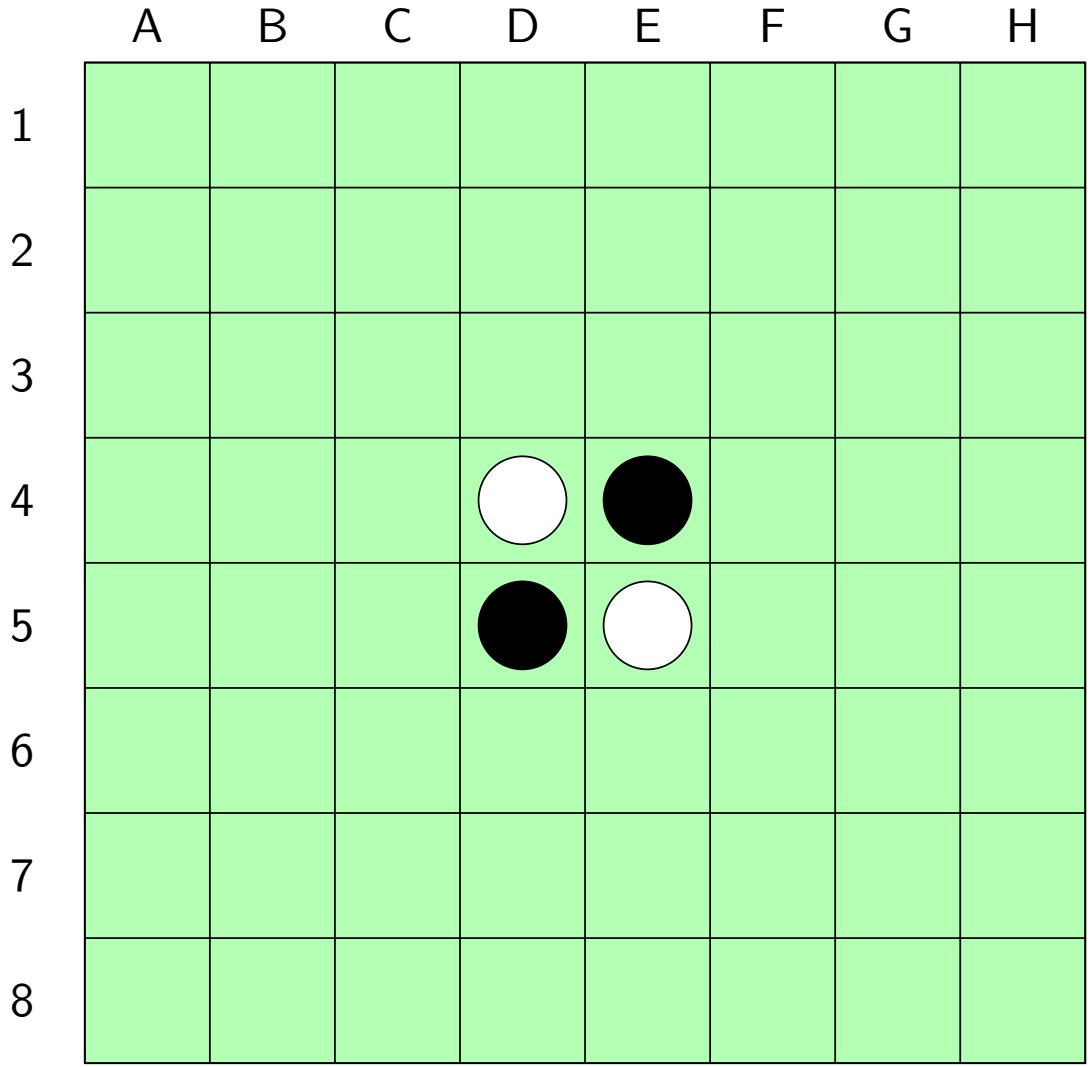
Intuition

- MCTS does not keep a value $V(s)$ for each state, but only for the states that
  1. are close to s_0
  2. look promising

- For the other states, MCTS uses a heuristic.

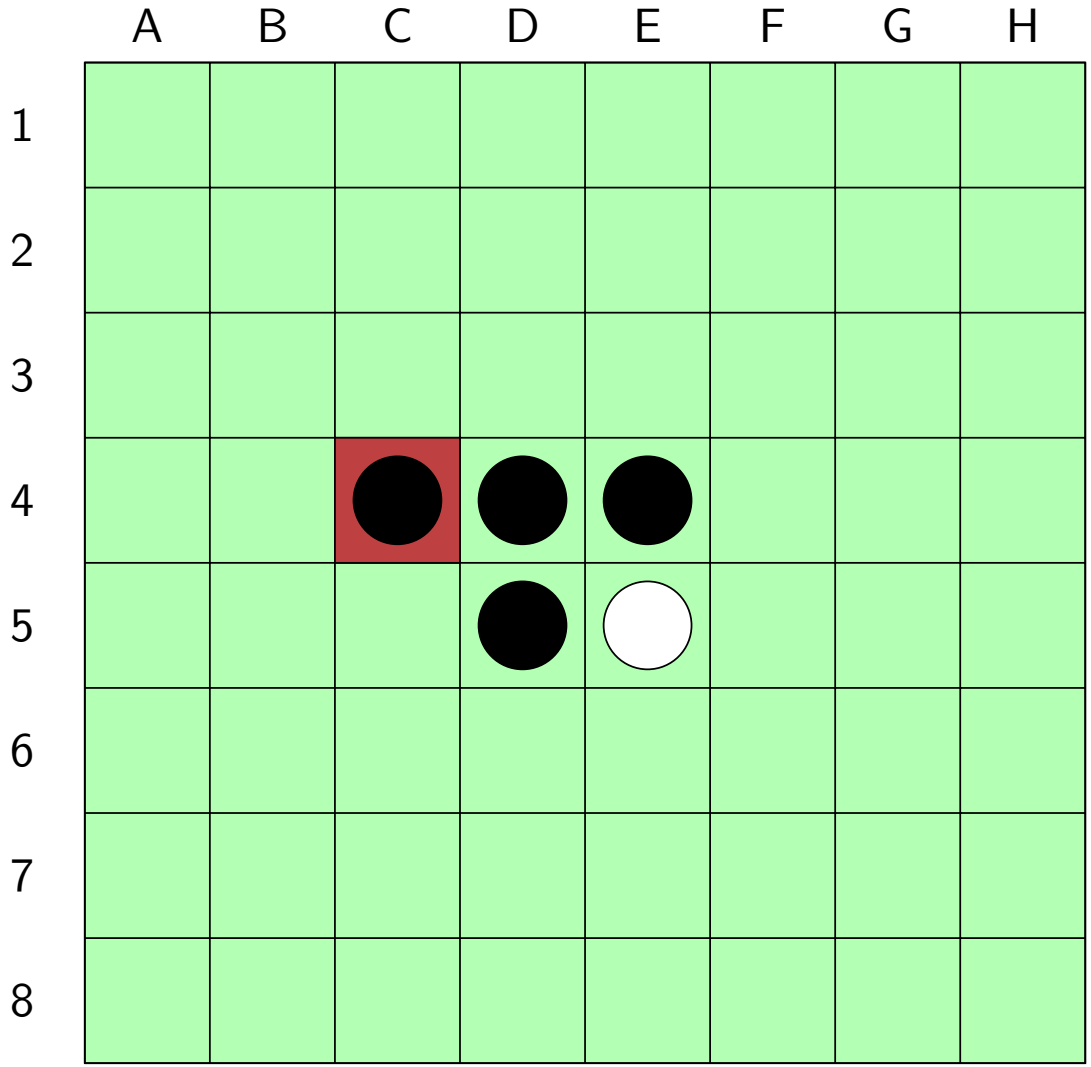
  The algorithm iteratively adds states with estimates $V(s)$

Example: Reversi



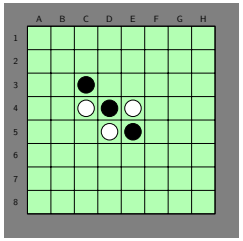
- Players place disk one at a time
- If a line of disks of one colour ends with two disks of the different colour (one of which was just placed), their colour is flipped.
- A move is legal only if it flips at least one disk.
- Example: move $C4$

Example: Reversi



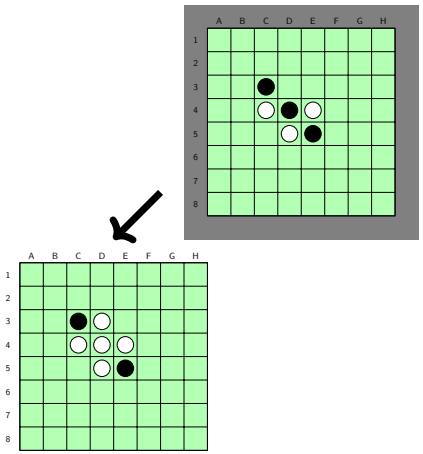
- Players place disk one at a time
- If a line of disks of one colour ends with two disks of the different colour (one of which was just placed), their colour is flipped.
- A move is legal only if it flips at least one disk.
- Example: move $C4$

MCTS on Reversi



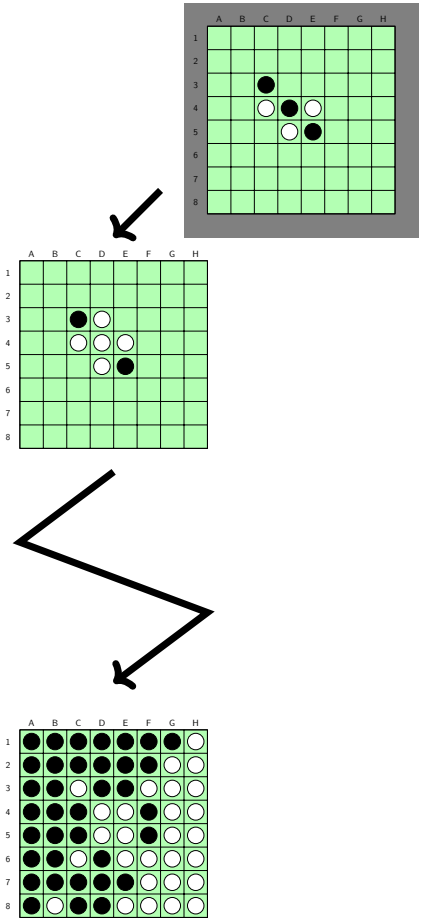
1. Selection

MCTS on Reversi



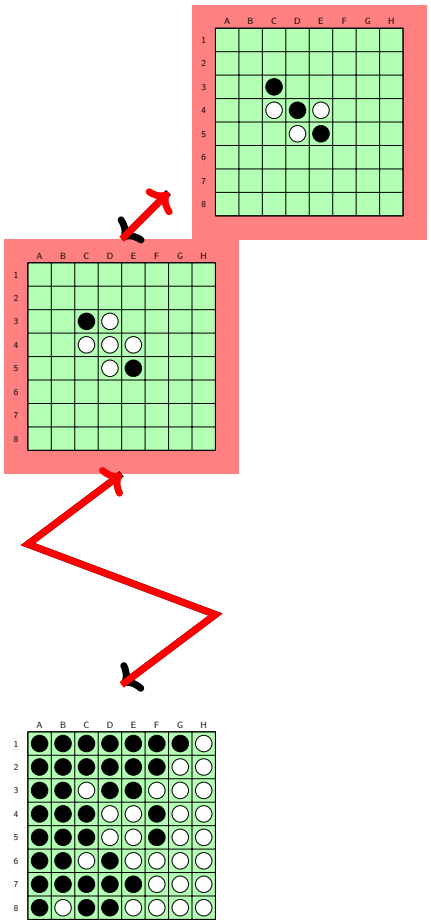
1. Selection
2. Expansion

MCTS on Reversi



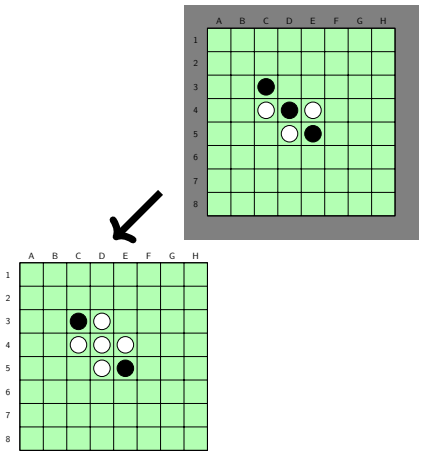
1. Selection
2. Expansion
3. Simulation

MCTS on Reversi



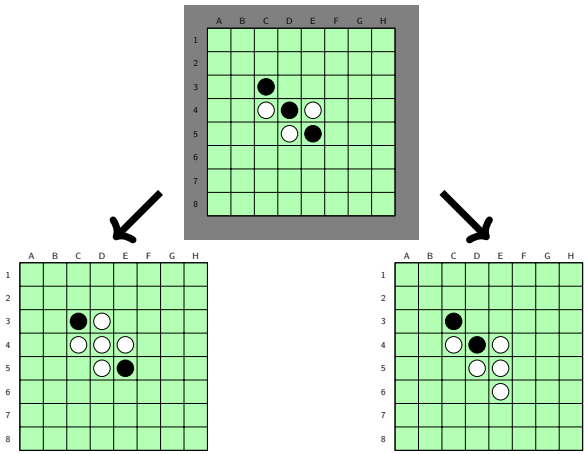
1. Selection
2. Expansion
3. Simulation
4. Backpropagation

MCTS on Reversi



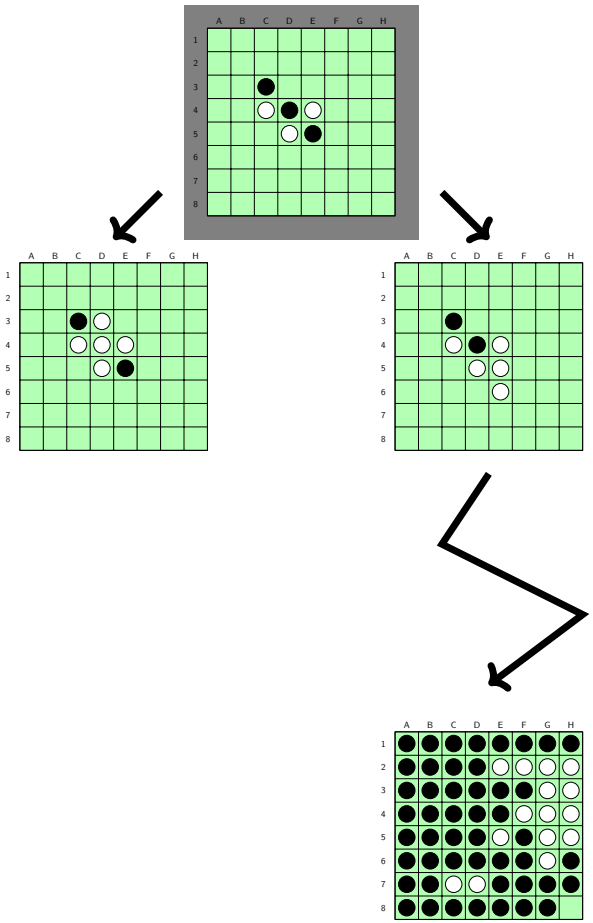
1. Selection

MCTS on Reversi



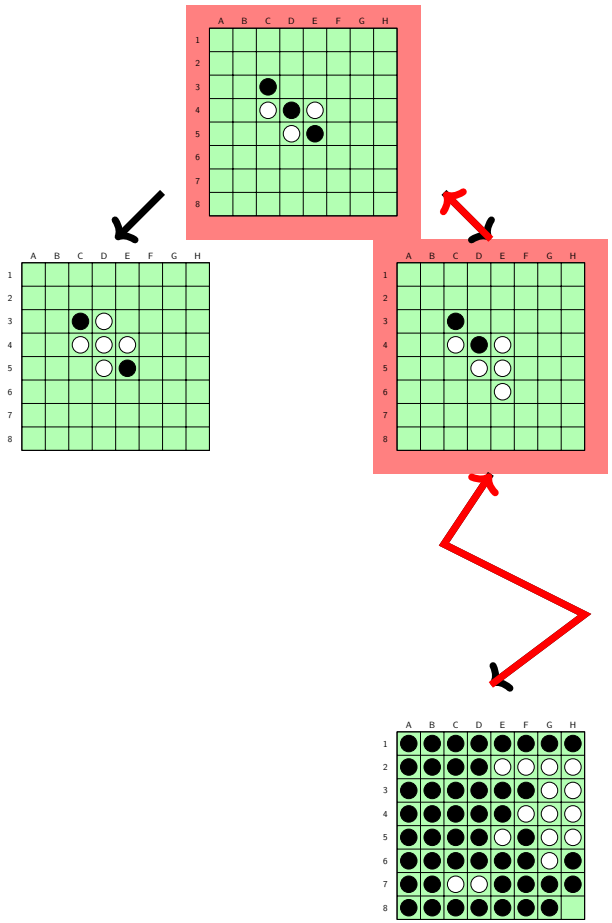
1. Selection
2. Expansion

MCTS on Reversi



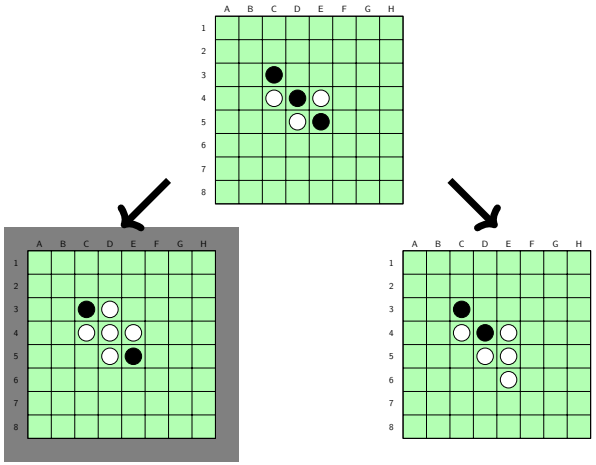
1. Selection
2. Expansion
3. Simulation

MCTS on Reversi



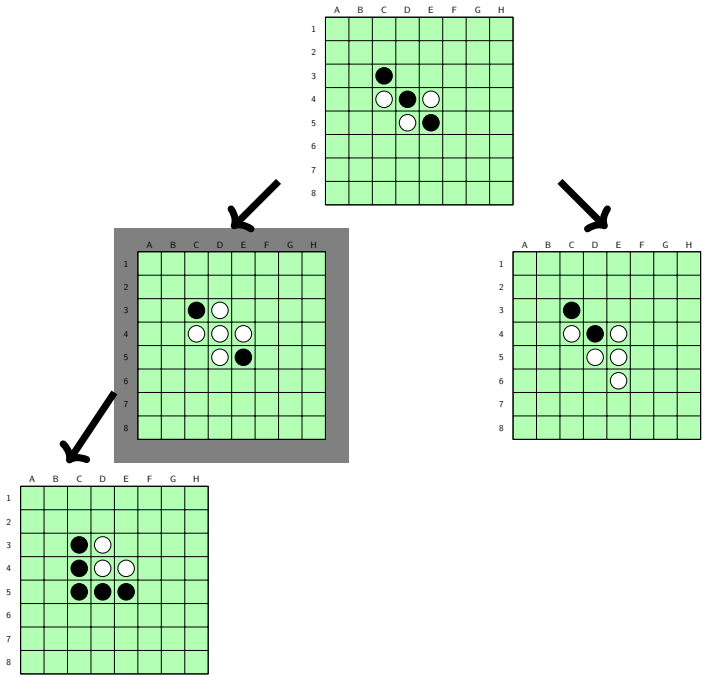
1. Selection
2. Expansion
3. Simulation
4. Backpropagation

MCTS on Reversi



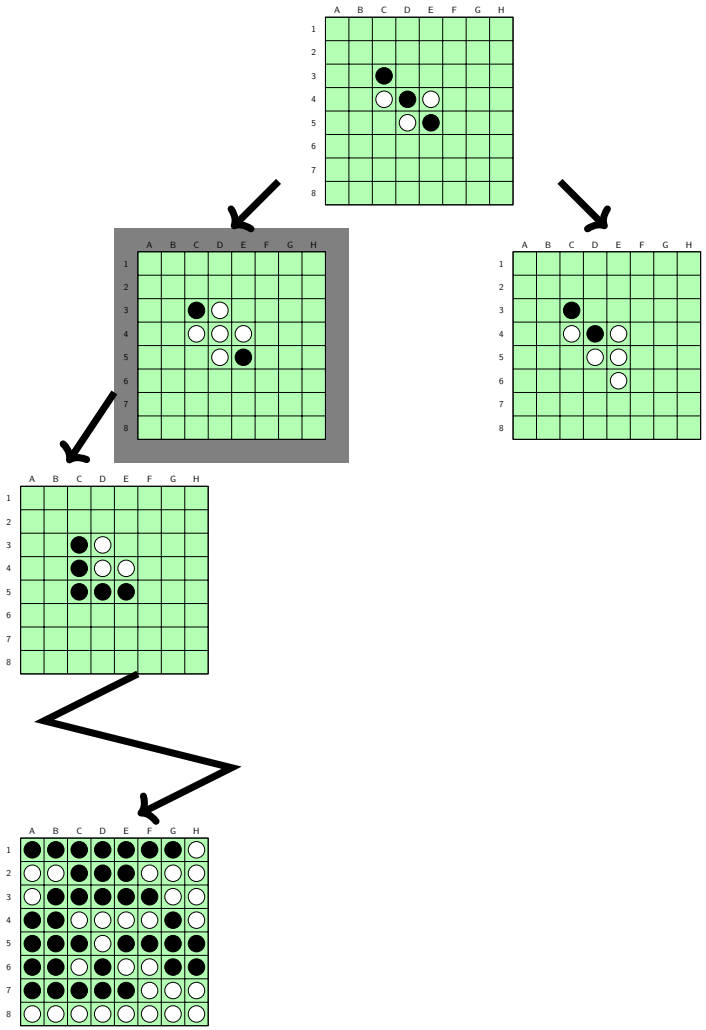
1. Selection

MCTS on Reversi



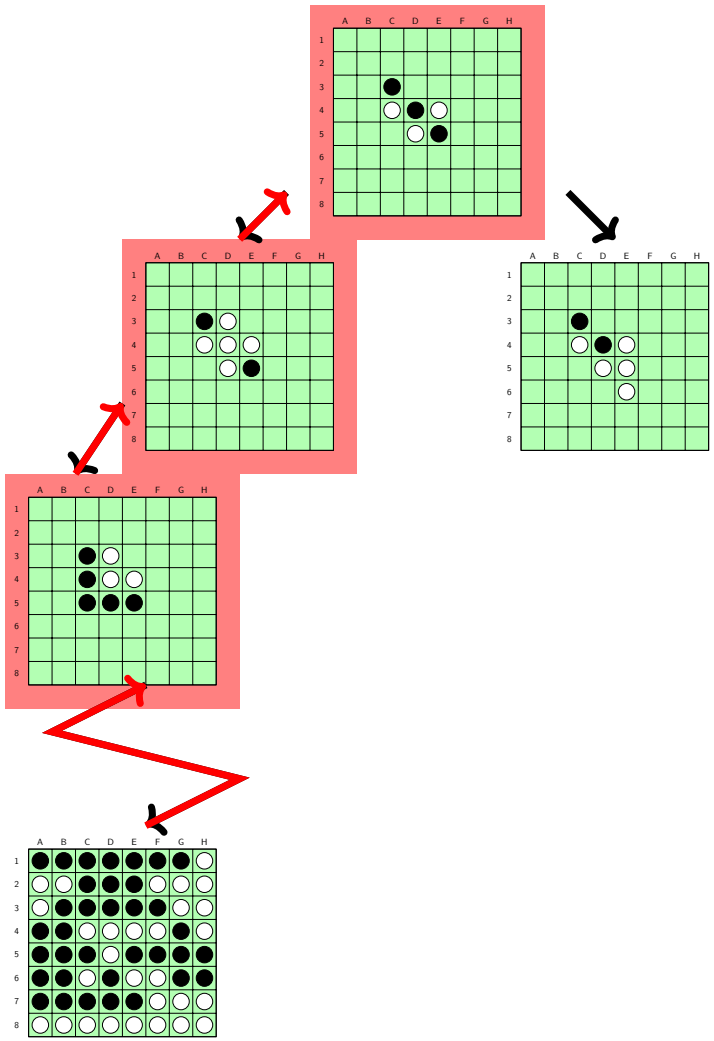
1. Selection
2. Expansion

MCTS on Reversi



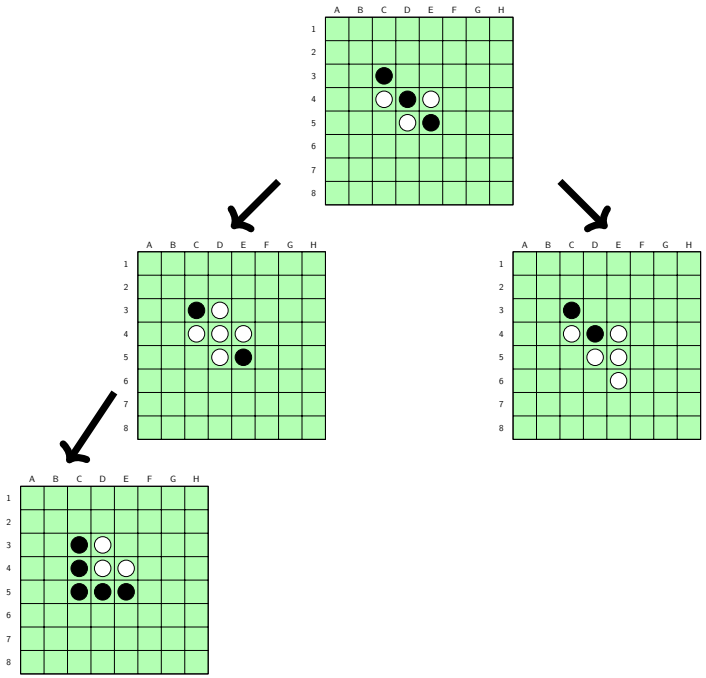
1. Selection
2. Expansion
3. Simulation

MCTS on Reversi



1. Selection
2. Expansion
3. Simulation
4. Backpropagation

MCTS on Reversi



1. Selection

Selection

- In this step, we select the part of the search tree that will be explored
- Which node should we select?

Selection (RTDP vs MCTS)

In RTDP

- we choose the best looking node because we know that the estimated value of each node is optimistic
- if we explore a suboptimal node, its estimate will worsen
- so, eventually, we will find the optimal node

In MCTS

- the estimate of a node depends on the outcome of the simulation
- it is unreliable, and neither optimistic nor pessimistic
- so, an unlucky simulation could give a bad evaluation to a good node

$\rightarrow$ we will have the same problem in reinforcement learning

Selection (in MCTS)

Exploration / exploitation dilemma:

- Exploration: you want to explore all possibilities in order to detect solutions that are not obvious
  - Cf. move 37 of the second game and move 78 in the fourth game between AlphaGo and Lee Sedol
    `https://www.youtube.com/watch?v=WXuK6gekU1Y`

- Exploitation: you want to concentrate on nodes that look good

Selection

Exploration / exploitation dilemma:
Starting from the root node, choose the child node that maximises a function that balances exploration and exploitation

$$\arg \max_{n' \in children(n)} function_1(value(n')) + function_2(nb\_evaluations(n'))$$

UCT

UCT (Upper Confidence Bound 1 applied to trees) is a version of MCTS that uses the Upper Confidence Bound 1 criterion for selecting nodes

The value of the child is:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

where

- w_i is the number of victories from the child node
- n_i is the number of times the node n_i was explored
- N_i is the number of times the current node was explored
- c is a factor that balances exploration vs exploitation

Expansion

Once a leaf of the tree has been selected, it is expanded

- the idea is to make sure we improve our knowledge of this node, so the best way to do that is to look more closely at the next states from this node

Simulation

From the current leaf node, we simulate how the system will behave

How to do the simulation?

- Random actions
- Greedy policy based on a heuristic (not necessarily better than random, because it is biased)
- Default (hand-written) policy
- Reinforcement learning (later in this course)

Backpropagation

Back-propagate to the parents of the current leaf node the result of the simulation

Heuristics

Some heuristics:

- Determinisation
- hmax
- idual

Determinisation

Consider:

- Stochastic Shortest Path: minimise cost, no action has a zero cost
- We want an admissible heuristic, i.e., that underestimates the values of each $V(s)$
- Remember the Bellmann Equations:

$$V(s) = \min_{a \in A(s)} Q(s, a)$$

and

$$Q(s, a) = \Sigma_{s' \in S} \; P(s, a, s') \times \Big(C(s, a, s') + \gamma \, V(s') \Big)$$

$\Rightarrow$ Pick the state s' that minimises $C(s, a, s') + h(s')$

- E.g., if the value is
  - $(0.5 \times 10) + (0.25 \times 15) + (0.25 \times 6)$

  then pick 6

$h_{\text{m-m}}$

- $h_{\text{m-m}}(s) = 0$ if s is a goal state
- $h_{\text{m-m}}(s) = \min_{a \in A(s)} \; C(s, a, s') + \gamma \min_{s' | P(s,a,s') > 0} h_{\text{m-m}}(s')$

- Finding the heuristic value is an optimal classical planning problem
- Still to hard to solve in general

hmax

Variable-based heuristics

- Remember: a state is a defined as a set of predicates
  - Examples: $(\mathit{RobotAt} = \ell_1)$, $(\mathit{Fuel} = 50)$, etc.
- Rather than reason on states, we reason on predicates
- Why?
  - If there are n predicates, there are up to 2^n states

- From state s, what is the minimal cost of achieving predicate p?
  - if p already holds in s, the cost is zero: $g_s(p) = 0$
  - otherwise, we need to apply an action a that produces p, and we need to satisfy the preconditions of that action

$$g_s(p) = \min_{a \in A(p)} \left(C(a) + \max_{p' \in \mathsf{precond}(a)} g_s(p') \right)$$

- If the goal G is defined as a set of predicates, we must satisfy all predicates:
  - $h_{\max}(s) = \max_{p \in G} \; g_s(p)$

- Replacing $\max$ with Σ (sum) yields h_{add}, which is not admissible

Dual-based heuristics

Linear Programming

A linear program is an optimisation problem defined over a set of rational-valued variables with linear inequalities.

Example:
Maximise x subject to

- $x + a \leq 2$
- $a + b \geq 10$
- $b \leq 9$

LP is only (weakly) P-COMPLETE

Linear Programming Formulation

SSP with no discount, trying to minimise the cost

Primal-Linear-Program

Variables: W_s for all $s \in S$

Maximise: $\Sigma_{s \in S}\ \alpha_s W_s$ where $\alpha_s > 0$ for all $s \in S$

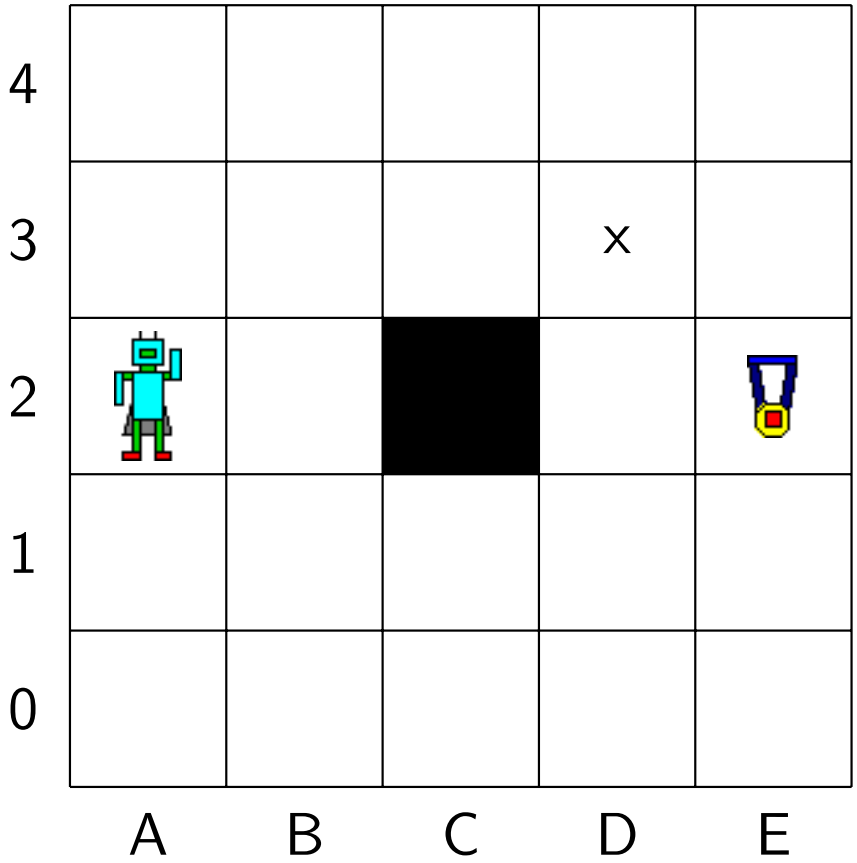
Constraints:

$W_s = 0$ for all $s \in G$

$W_s \leq C(a) + \Sigma_{s' \in S}\left(P(s, a, s') \times W_{s'}\right)$ for all $s \in S$ and all $a \in A_s$

- W_s in the optimal solution is $V^*(s)$

Formally: Little Robot



Example for $D3$

- Up: $W_{D3} \leq 1 + W_{D2}$
- Down: $W_{D3} \leq 1 + W_{D4}$
- Left: $W_{D3} \leq 1 + W_{C3}$
- Right:
  $W_{D3} \leq 1 + (.5 \times W_{E3} + .5 \times W_{E2})$

Why Does it Work?

- First, notice that $W := V^*$ is a solution (satisfies the constraints)
- Assume now that there is another solution W with a larger objective: $\Sigma_{s \in S}\ \alpha_s W_s > \Sigma_{s \in S}\ \alpha_s V^*(s)$
  - Then, for some s, $W_s > V^*(s)$.
  - Let $q \in S \setminus G$ be the state that maximises $m = W_q / V^*(q) > 1$. Notice $mV^*(s) \geq W_s$ for all states s.
  - Let $a := \pi^*(q)$.
  - Then:

$$
\begin{aligned}
W_q = mV^*(q) &= m\left[C(a) + \left(\Sigma_{q' \in S} P(q,a,q')\, V^*(q')\right)\right] \\
&= mC(a) + \left(\Sigma_{q' \in S} P(q,a,q') m V^*(q')\right) \\
&> C(a) + \left(\Sigma_{q' \in S} P(q,a,q') m V^*(q')\right) \\
&> C(a) + \left(\Sigma_{q' \in S} P(q,a,q')\, W_{q'}\right) \text{ QED}
\end{aligned}
$$

Complexity

Deciding the value of a state is (weakly) P-COMPLETE with an enumerated representation.

Caveat:

- The enumerated representation is exponentially large compared to the number of state variables.
- And the number of linear constraints is $O(\mathit{NumberOfStates} \times \mathit{NumberOfActions})$.

Dual Linear Program

- Dual of the (Primal) Linear Program $\rightarrow$ cf. theory of optimisation
- Essentially, the variables in the dual program are not the value of the states, but the number of times a state is accessed on average.
- Understand the dual linear program as a flow:
  - $in(s)$ and $out(s)$ is the number of times the state s is reached and exited;
  - $x_{s,a}$ is the number of times that the action a is performed in state s.

Dual Linear Program

Dual-Linear-Program

Variables:

- $in(s)$ and $out(s)$ for each state;
- $x_{s,a}$ for each state s and action a applicable in s

Minimise: $\Sigma_{s \in S, a \in A} \; x_{s,a} \times C(s,a)$

Constraints:

$x_{s,a} \geq 0$	for all $s \in S$ and $a \in A(s)$
$in(s) = \Sigma_{s' \in S, a \in A(s')} \; x_{s',a} \times P(s',a,s)$	for all $s \in S$
$in(s) = out(s)$	for all $s \in S \setminus (G \cup \{s_0\})$
$out(s_0) - in(s_0) = 1$	
$out(s) = \Sigma_{a \in A(s)} \; x_{s,a}$	for all $s \in S \setminus G$
$\Sigma_{s \in G} \; in(s) = 1$	

Problem with Dual LP

- The number of variables is $O(\mathit{NumberOfStates} \times \mathit{NumberOfActions})$
- The number of constraints is $O(\mathit{NumberOfStates})$

Simplification

- Instead of keeping track of the states, we keep track of the assignments $v = \nu$
- We project the SSP on each state variable v
- We define the Dual Linear Program for each state variable
  - LP variables $x_{s,a}$ are replaced with $x_{v,\nu,a}$
- We link the dual LPs as follows:
  - for all action a, state variable v, state variable v' $\Sigma_{\nu \in \text{domain}(v)} x_{v,\nu,a} = \Sigma_{\nu \in \text{domain}(v')} x_{v',\nu,a}$

Example

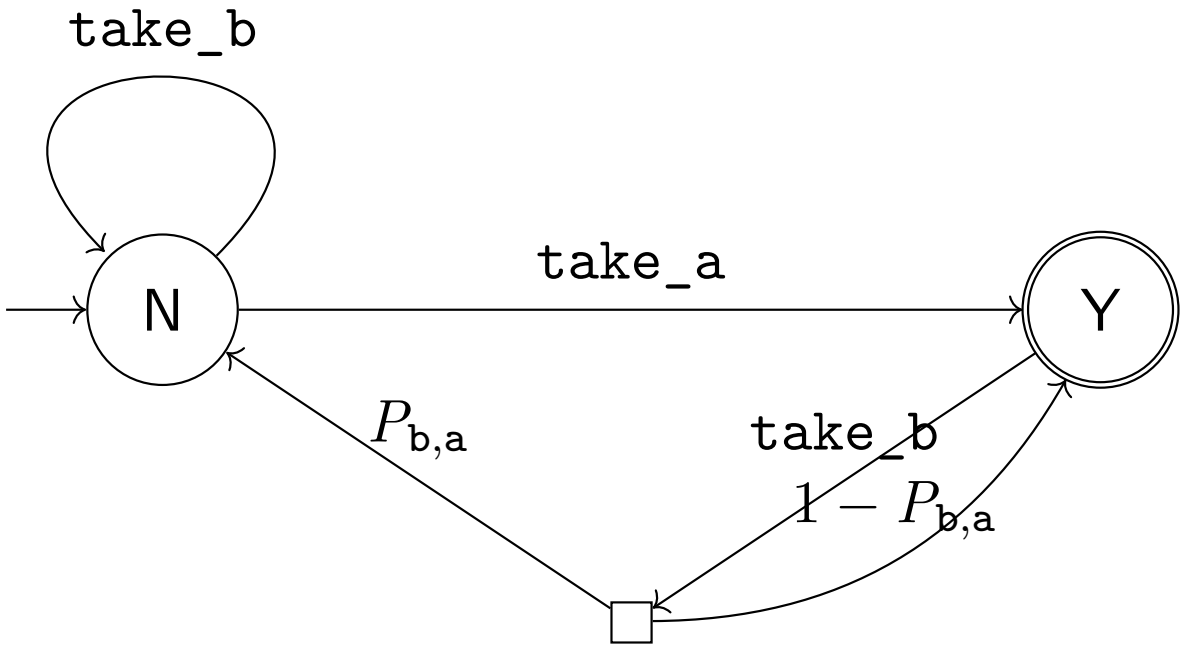
- Problem: need to pick up four items A, B, C, D
  - Four variables: `holds_a`, `holds_b`, etc. with domain $\{Y, N\}$
  - Initial state: `holds_x` $\rightarrow N$ for all `x`
  - Goal state: `holds_x` $\rightarrow Y$ for all `x`

- Four actions: `take_a`, `take_b`, etc.
  Effect of `take_x`
  - `holds_x` $:= Y$ with probability 1
  - for `y` $\neq$ `x`, if `holds_y` $= Y$, then `holds_y` $:= N$ with probability $P_{\texttt{x,y}}$

- Why it is "difficult":
  - Need to find in which order to pick-up the items

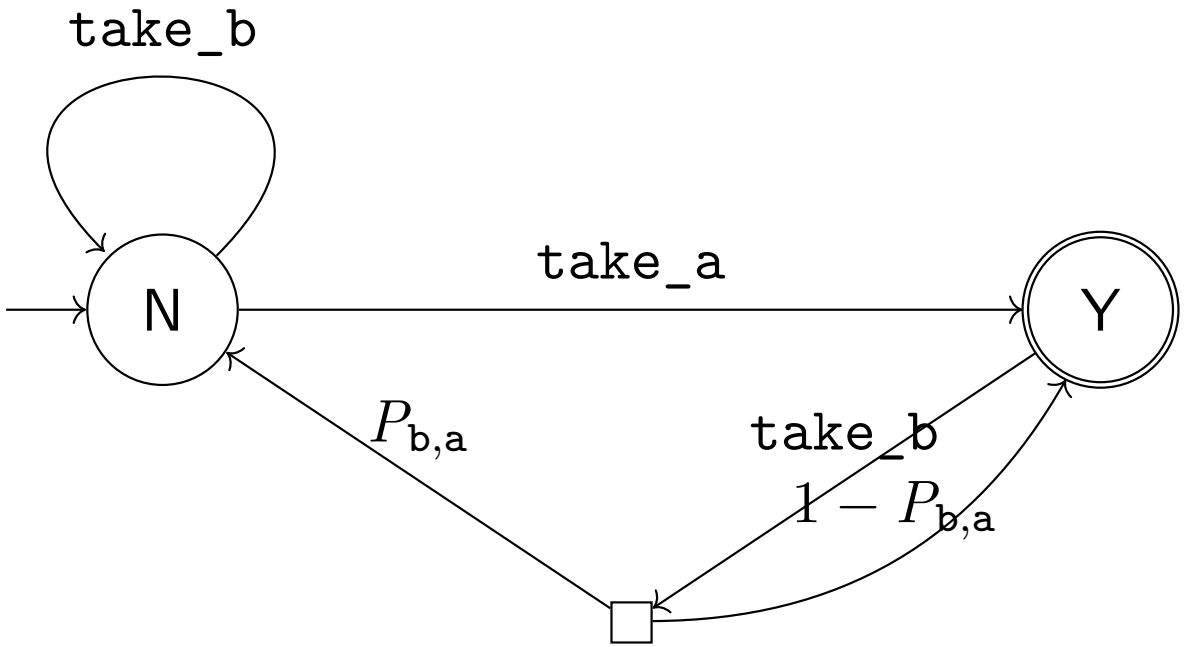
Projection

Variable `holds_a` (not showing `take_c` and `take_d`)



Projection

Variable `holds_a` (not showing `take_c` and `take_d`)



Example of constraint:

- $in(A, Y) = \left(x_{A,N,\texttt{take\_a}} \times 1\right) + \left(x_{A,Y,\texttt{take\_b}} \times (1 - P_{\texttt{b,a}})\right) + \left(x_{A,Y,\texttt{take\_c}} \times (1 - P_{\texttt{c,a}})\right) + \left(x_{A,Y,\texttt{take\_d}} \times (1 - P_{\texttt{d,a}})\right)$

Tying it Together

- $x_{A,Y,\texttt{take\_c}} + x_{A,N,\texttt{take\_c}} = x_{B,Y,\texttt{take\_c}} + x_{B,N,\texttt{take\_c}}$

The h^{pom} Heuristic

- Put all the constraints together:
  - Number of variables:
    $O(NumberOfActions \times NumberOfVariables \times DomainSize)$
  - Number of constraints: $O(NumberOfActions \times NumberOfVariables^2)$

- Why is it only a heuristic?
  - It does not verify what is the current state when the actions are performed
    (cf. the constraint in the previous slide)