Deliberative Agents

Boi Faltings

Laboratoire d'Intelligence Artificielle
boi.faltings@epfl.ch
http://moodle.epfl.ch/

Fall 2018

Deliberative architecture

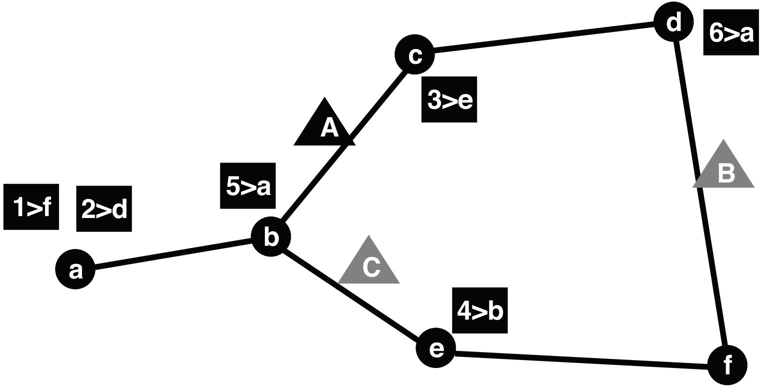
Reactive architecture: difficult to

- plan over time
- act with varying goals (e.g. receive instructions)
- coordinate with other agents

Explicit consideration of action outcomes
⇒ deliberative architecture

"*... the art and practice of thinking before acting.*" - P. Haslum

Example: delivery problem



Agent A delivers packages 1..6 to their destinations

Requirements

Reactive agents = Decision processes:

every state is assumed to be reachable!

State encodes many variables (carrying packages, location of objects, etc.) $\Rightarrow$ combinatorial explosion:

$$\begin{aligned}\mathcal{S} &= pos(robot) \times pos(1) \times holding(1) \times \ldots \\ &= \{a..f\} \times \{a..f\} \times \{T, F\} \times \ldots\end{aligned}$$

$|\mathcal{S}| = 6^7 \cdot 2^6 = 17'915'904$
$|A| \simeq 10 \Rightarrow |T| = 3'209'796'161'372'160$ entries
$\Rightarrow$ does not fit into any computer memory!

Requirements (2)

1. Changing goals
   Example: new packages to be delivered
   Effect: changing reward structure
2. Actions of other agents
   Example: Agents B,C also move packages, block pathways
   Effect: changing transition and reward structure

$\Rightarrow$ requires recomputing entire policy each time.

Reactive ⇒ deliberative agents

Observation:

> *When the state space is large, or when a problem is solved only once, only a small subset of the states are actually visited*

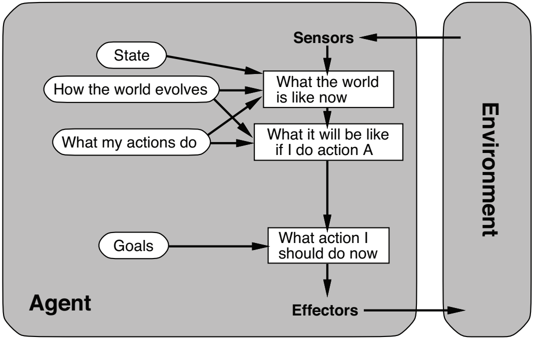
⇒ computing optimal actions for every state is a waste of effort!

Examples:

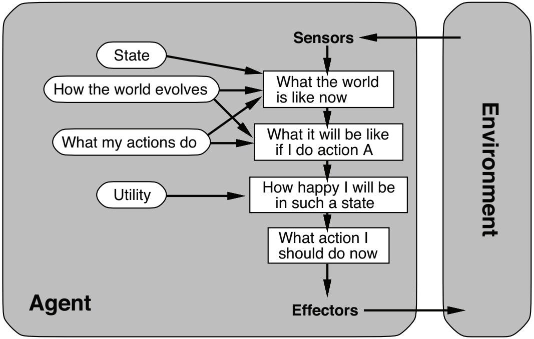
- Package 5 should never be moved to to c,d,e, or f
- Package 3 will never be moved to a
- if A starts in a, it will not leave both packages 1 and 2 in a

⇒ unnecessary states

Agent with explicit goals



Utility-based agent



Deliberative solution of decision processes

State	s_1	s_1	s_1	s_1
Action	a_1	a_2	a_3	a_4
Reward	-1	-2	0	**2**
Next state	s_2	s_1	s_2	$\mathbf{s_2}$
State	s_2	s_2	s_2	s_2
Action	a_1	a_2	a_3	a_4
Reward	1(+2)	**2(+2)**	-3(+2)	-1(+2)
Next state	s_4	$\mathbf{s_3}$	s_1	s_4
State	s_3	s_3	s_3	s_3
Action	a_1	a_2	a_3	a_4
Reward	1(+4)	2(+4)	-2(+4)	**4(+4)**
Next state	s_1	s_4	s_4	$\mathbf{s_1}$

Deliberative solution of decision processes

- focus only on current state and successors.
- need to search all possible sequences $\Rightarrow$ tree search.
- state space grows exponentially $\Rightarrow$ stop at bounded depth.

Example

Initial state:

$$s_0 : pos(A) = a, pos(1) = a, pos(2) = a, pos(3) = c, \ldots$$

Goal states:

$$\begin{aligned} s_{t1} : \quad & pos(A) = a, pos(1) = f, pos(2) = d, pos(3) = e, \ldots \\ s_{t1} : \quad & pos(A) = b, pos(1) = f, pos(2) = d, pos(3) = e, \ldots \end{aligned}$$

...

Actions:

- pick up/drop off package
- move along the graph

State-based planning algorithms

Assume initial state is known, e.g.:

s_0: (pos(A)=a, pos(1)=a, pos(2)=a, pos(3)=c, pos(4)=e, pos(5)=b, pos(6)=d)

Consider only states that can be reached from known states:

s_1: (**pos(A)=b**, pos(1)=a, pos(2)=a, pos(3)=c, pos(4)=e, pos(5)=b, pos(6)=d)
s_2: (**pos(A)=b**, **pos(1)=b**, pos(2)=a, pos(3)=c, pos(4)=e, pos(5)=b, pos(6)=d)
s_3: (**pos(A)=b**, pos(1)=a, **pos(2)=b**, pos(3)=c, pos(4)=e, pos(5)=b, pos(6)=d)

$\Rightarrow$ every step multiplies number of states by *branching factor* b

Number of states at level l (assuming no duplicates):

$$c(l) = \sum_{i=1}^{l} b^i = \frac{b^{l+1} - 1}{b - 1} \leq b^{l+1}$$

Exponential growth with each level, but often less states than for value/policy iteration: $b = 3, l = 10 \Rightarrow c(l) = 88'573$.

Formalization

Algorithm = search:

systematically generate possible action sequences and test each whether it leads from initial to goal state.

Search node:

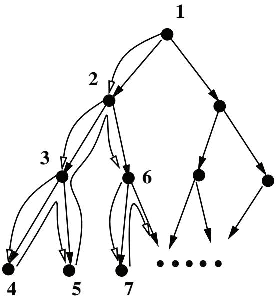
n = *a state*

Successor function:

$succ(n)$ = *list of nodes (states) reached from* n *by simulating all different actions.*

Depth-first search

The search space is considered as a tree:



Depth-first: always expand the first node found until there are no more successors (or a final node is found)
⇒ backtrack: return to the last level with an untried possibility and continue from there.

Properties (depth-first search)

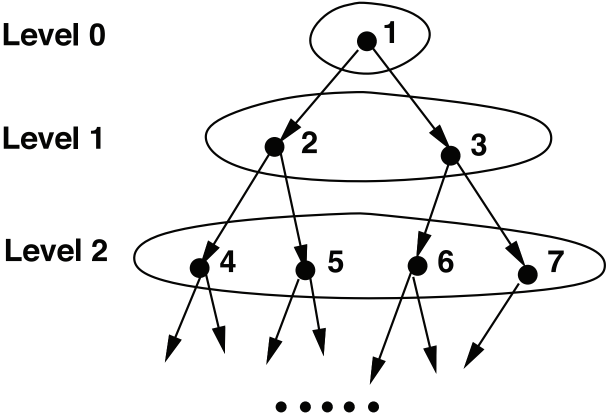
Advantage:

- memory required = list of "open" nodes
  grows linearly with depth of search
  (logarithmic in size of search space)

Disadvantages:

- Plan found may not be the shortest
- Heavy-tailed distributions: may be stuck in unpromising branch

Breadth-first search



Breadth-first: exploring the tree layer by layer

Properties (breadth-first search)

Advantages:

- Always finds the shortest plan.
- Not penalized by bad initial decisions.

Disadvantage:

- Requires large amounts of memory to store all nodes of the tree at each level.

Algorithms

Depth-first:

```
1: Function DFS (InitialNode)
2: Q ← (InitialNode)
3: repeat
4:    n ← first(Q), Q ← rest(Q)
5:    if n is a goal state, return n
6:    S ← succ(n)
7:    Q ← append(S, Q)
8: until Q is empty
9: return FAIL
```

Breadth-first: exchange the order in step 7:

```
7.    Q ← append(Q, S)
```

Depth-limited search

- Usually, breadth-first search requires too much memory to be practical.
- Main problem with depth-first search:
  can follow a dead-end path very far before this is discovered.

$\Rightarrow$ impose a *depth limit* l:
never explore nodes at depth $> l$

Depth-Limited Search (DLS)

```
Function DLS (InitialNode,l)
depth-limit(InitialNode) ← l
Q ← (InitialNode)
repeat
    n ← first(Q), Q ← rest(Q)
    if n is a goal node, return n
    S ← succ(n)
    for nn ∈ S do
        depth-limit(nn) ← depth-limit(n)-1
    if depth-limit(n) > 0 then Q ← append(S,Q)
until Q is empty
return FAIL
```

Q: What is the right depth limit *l*?

Iterative Deepening

Increase the limit:

Function Iterative-deepening(InitialNode)
$l \leftarrow 2$
repeat
    $solution \leftarrow DLS(InitialNode, l)$
    $l \leftarrow l + 1$
until $solution \neq \{\}$

Every step repeats earlier search steps....

Isn't this costly?

Complexity

If the algorithm finds a solution at depth l, it has explored all subspaces of depth l, $l-1$, ..., 2.
Let there be $c(i) = b^{i+1} - 1$ nodes in the search space up to depth i, then the total complexity is:

$$
\begin{aligned}
\sum_{i=2}^{l} c(i) &= \sum_{i=2}^{l} (b^{i+1} - 1) \\
&= \left(b^{l+1} \sum_{i=0}^{l-2} b^{-i} \right) - (l-1) \\
&< \left(b^{l+1} \cdot \frac{b}{b-1} \right) - (l-1) \leq 2c(l)
\end{aligned}
$$

$\Rightarrow$ as long as $b \geq 2, l \geq 3$, complexity is no more than doubled!

Finding the optimal plan

- cost of plan = sum of costs of actions.
- step from node n' to successor n has a cost c(n',n):
- thus, the cost $g(n)$ of node n is:

$$g(n) = c(n', n) + g(n') = c(n', n) + \sum_{n', n'' \in ancestors(n)} c(n', n'')$$

Optimization in DFS

- Keep track of the best goal node found so far.
- Insight: for any node n, cost of a successor can never be lower than cost of n.

$\Rightarrow$ any node that has higher cost than the best solution found so far can not lead to a better solution.

$\Rightarrow$ all such nodes can be ignored (branch-and-bound).

- optimal plan = best plan when queue is empty.

Heuristic search

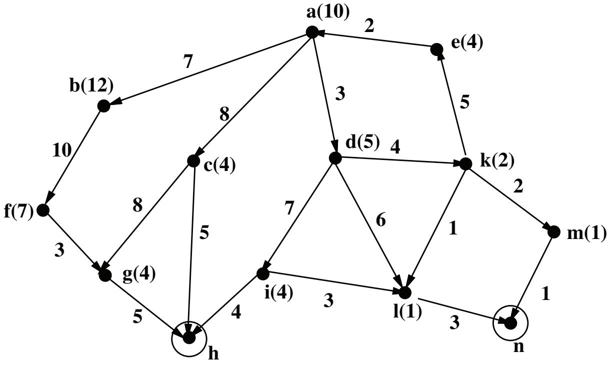
- Search should be guided to first explore the most promising solutions.
- This can be done using a *heuristic* function:
  h(n) = estimate of the minimal cost from node n to a goal node.
- Define g(n) = cost of the best path to node n, then:
  $$f(n) = g(n) + h(n)$$
  is an estimate of the cost of the best path from initial to goal node *passing through node n*.

$\Rightarrow$ first explore nodes with a low value of f(n).

Algorithm A* (best-first)

```
Q ← initial node
C ← empty
repeat
   if Q is empty, return failure
   n ← first element of Q, Q ← rest(Q)
   if n is a final node, return n
   if n ∉ C, or has lower cost than its copy in C then
      add n to C
      S ← succ(n)
      S ← sort(S,f)
      Q ← merge(Q,S,f)
  (Q is ordered in increasing order of f(n) = g(n) + h(n))
   endif
end
```

Example of a heuristic search



Order of exploration

(a(10)) $\Rightarrow$
(d(8),c(12),b(19)) $\Rightarrow$
(k(9),l(10),c(12),i(14),b(19)) $\Rightarrow$
(l(9),m(10),c(12),i(14),e(16),b(19)) $\Rightarrow$
(m(10),n(11),c(12),i(14),e(16),b(19)) $\Rightarrow$
(n(10),c(12),i(14),e(16),b(19)) $\Rightarrow$
solution!

Beam search

- List `Q` in A* algorithm may get very long
- Idea: limit `Q` to only keep `n` best nodes, throw away the others.

⇒ *beam* search with width `n`

- Incomplete: can miss the best solution!
- Iteratively increasing limit does not solve the problem of suboptimal solution.

Optimality in A*

- The performance of A* depends largely on the quality of the heuristic function.
- If we always have `h(n) = 0`, nodes are explored in the order of their cost: any path found will always be the shortest possible, thus optimal.
- If the function `h(n)` *overestimates* the true cost $h^*(n)$ remaining from n to a final node, we can have:
  `g(n') + h(n') < g(n) + h(n) (> g(n) + h*(n))`
  even if total cost of path through n is less than that through n':
  $$g(n') + h^*(n') > g(n) + h^*(n)$$
  $\Rightarrow$ the solution found might be sub-optimal!
- One can prove that A* always finds the optimal solution as long as `h(n)` **under-estimates** the true cost.

Planning with an adversary

Planning world may include other agents that

- do random actions
- compete, or
- are direct adversaries

$\Rightarrow$ plan has to take their actions into account

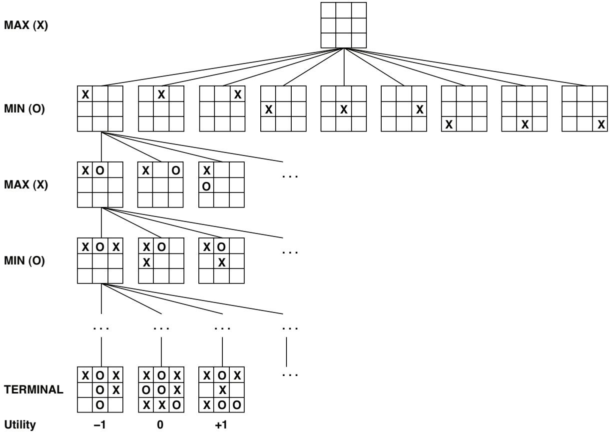
Purest form: games

Games as search

- state =
  board position + turn
- successor function =
  moves that can be made by the next player
- goal states =
  positions where one or the other has won, possibly with a payoff (cost) function

$\Rightarrow$ possible games = search tree

Example of a game tree

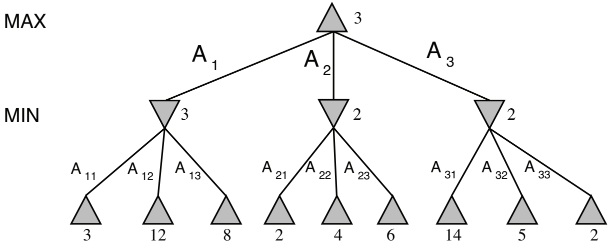


Minimax search

Each player controls only certain layers of the outcome ⇒ assume:

- when agent is in control, it will **max**imize payoff
- when others are in control, they will **min**imize payoff

⇒ backpropagation to earlier states



Imperfect decisions

- Real games do not allow generating the entire game tree!
- Chess:

$$d = 50 \textit{ moves, } b = 35 \textit{ possibilities} \Rightarrow 35^{50} \textit{ states!}$$

$\Rightarrow$ can only search to a certain depth!

- Assume that *horizon states* are final states, evaluate with a heuristic

Evaluating horizon states

Encode features of board position $= x_i$:

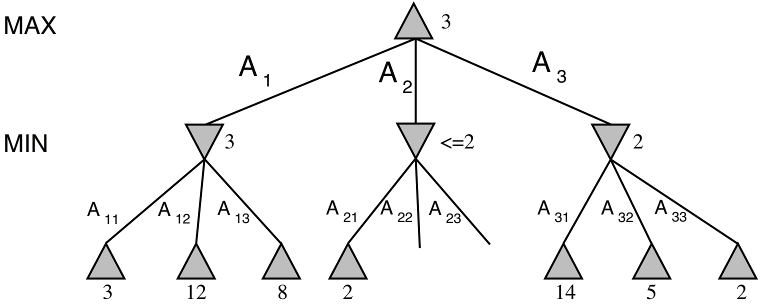
- number of pieces
- number of pieces threatening another
- points already won
- ...

State evaluation = heuristic based on board position:

$$f = a_1 \cdot x_1 + a_2 \cdot x_2 + ... + a_n \cdot x_n$$

Alpha-Beta pruning

Minimax search can be optimized:



$\Rightarrow$ abandon A_2 without searching A_{22}, A_{23}

Principle

DFS algorithm keeps record of:

- α = best choice found on this path for *MAX*
- β = best choice found for *MIN*

Abandon a branch as soon as:

- *MAX* $> \beta$: the opponent would never allow us to get there
- *MIN* $< \alpha$: we have already found a branch where the opponent can do us less harm

Algorithm (Alpha-Beta pruning)

function MAX-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*
  inputs: *state*, current state in game
    game, game description
    α, the best score for MAX along the path to *state*
    β, the best score for MIN along the path to *state*

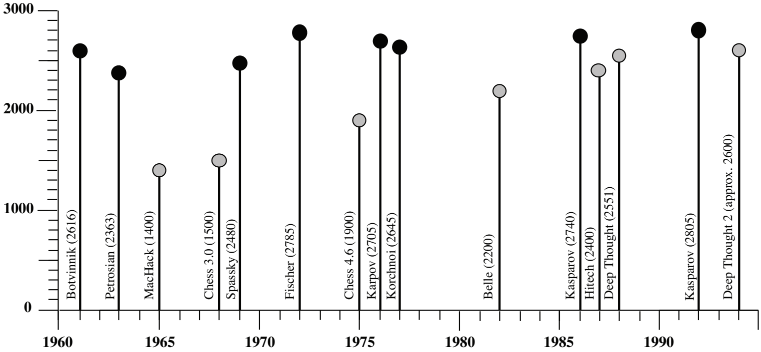
  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\alpha \leftarrow$ MAX(α, MIN-VALUE(*s*, *game*, α, β))
    if $\alpha \geq \beta$ **then return** β
  end
  return α

---

function MIN-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*

  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\beta \leftarrow$ MIN(β, MAX-VALUE(*s*, *game*, α, β))
    if $\beta \leq \alpha$ **then return** α
  end
  return β

Performance of chess programs



Beyond Alpha-Beta

Stochastic models can simplify complex deterministic games.
Example: Go (Weiqi, Baduk)

- Very large branching factor / search space:
  d = 150 moves, b = 361 possibilities $\Rightarrow 361^{150}$ states!
- Replace details by stochastic models.

Go: Traditional Approaches

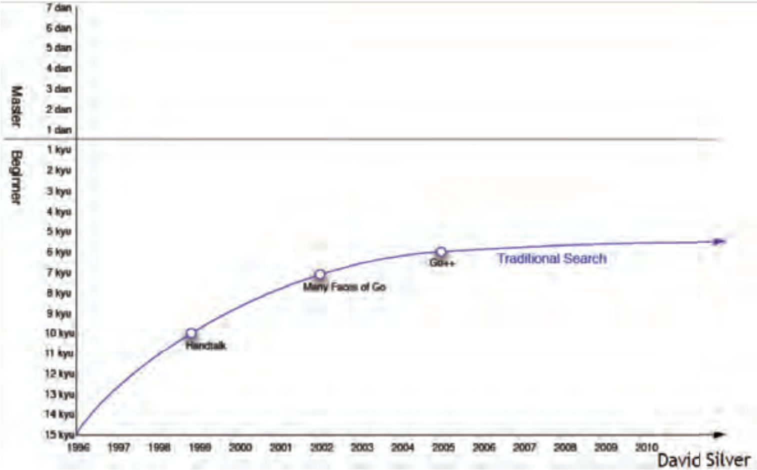
Alpha-beta search: combine global and local searches.

- Large branching factor.
- Difficult to define boundaries for local search.

Selective search: encoding human expert-knowledge.

- Require human expertise.
- Require extensive manual tuning.

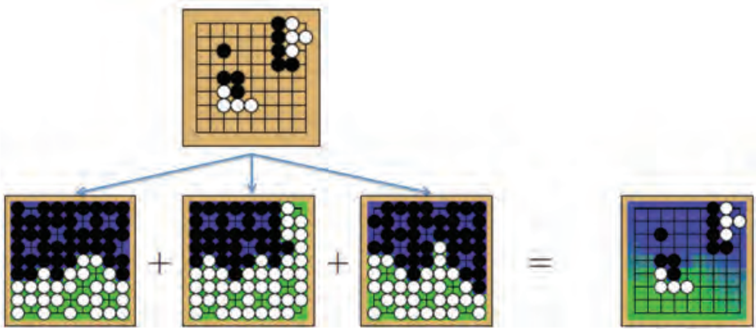
Go: Traditional Approaches



A New Approach for Go

State evaluation using *Monte Carlo* Algorithm:

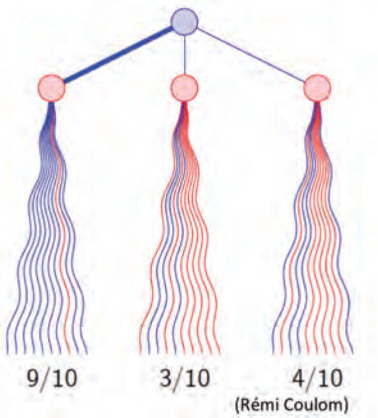
- Given a state, 'randomly roll-out' sequences of legal moves to the end of the game.
- Estimate winning probabilities by statistics over many roll-outs.



(Rémi Coulom)

Simple Monte Carlo Search

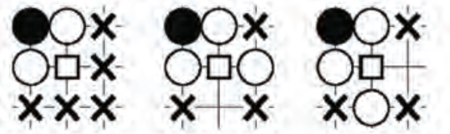
Evaluate all possible next states using Monte Carlo algorithm with N roll-outs, pick the best move.



Roll-out policy

Problem: many “random” moves are simply useless.
Solution: Assign a simple roll-out policy to avoid useless/harmful moves, and play important/winning moves:

- Non-suicidal
- Handicrafted pattern matching (MoGo 2006)
- Learnt from expert game database (CrazyStone 2007)



Default cut patterns in MoGo, if the pattern from the left is matched, and the pattern from the middle or the right is not matched, then black plays in the central square. Gelly 2006

Selective Sampling

Problem: Simple Monte Carlo samples many poor moves.

WANTED

A way to focus on "promising moves".

We already know something about the moves from our Monte Carlo sampling. Therefore, at each move, we need to consider:

- Exploitation: moves optimized using existing estimates.
- Exploration: moves that have not been sufficiently sampled.

This can be formulated as a Multi-Armed Bandit problem.

Upper Confidence Bound Applied to Trees (UCT)

Apply upper confidence bound to game trees [Kocsis and Szepesvari, 2006]:

- Select player's move with the greatest upper confidence bound on its value.
- Select opponent's move with the minimal lower (sign reversed) confidence bound on its value.

Upper Confidence Bound for Move i

$$UCB_i = \frac{W_i}{N_i} + c\sqrt{\frac{log(N)}{N_i}}$$

W_i = win with move i; N_i = roll outs with move i;
N = total roll outs; c = exploration parameter.

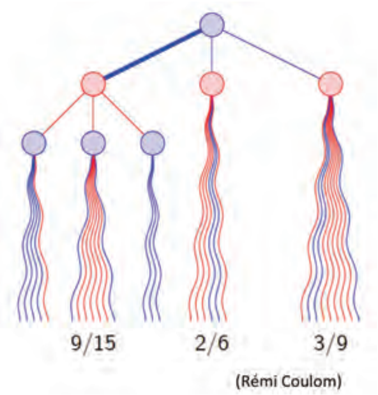
Monte Carlo Tree Search (MCTS)

A hybrid search algorithm:

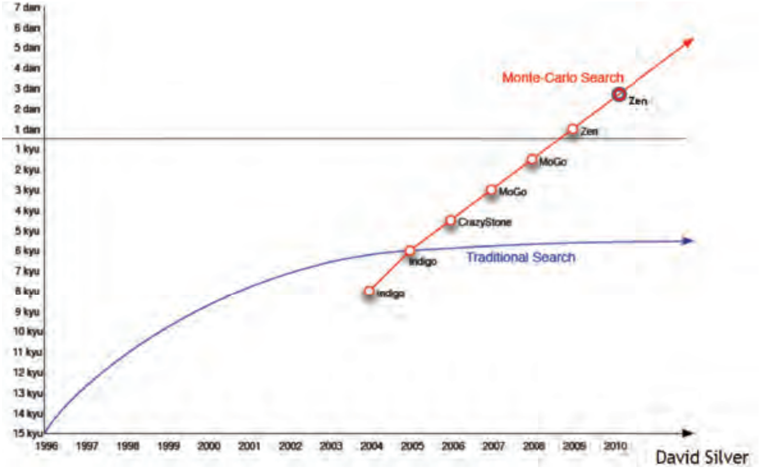
- Use UCT to build a game tree.
- Monte Carlo for evaluating leaf nodes.

Leading Computer Go programs based on MTCS:

- CrazyStone, MoGo, Zen, AlphaGo
- Different heuristics are used to guide UCT and Monte Carlo.



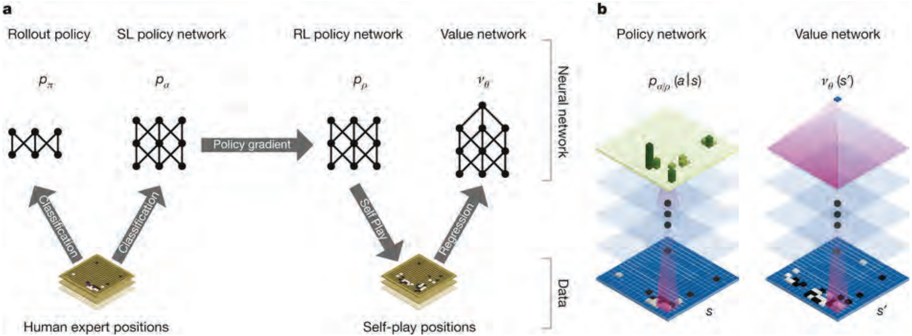
MTCS in Practice



AlphaGo

- AlphaGo combines Monte-Carlo search with deep neural nets.
- Uses deep neural networks for:
  - evaluating the quality of a board position.
  - suggesting next moves to try during random rollout.
- First trained on many many human games, then further improved by playing against itself.
- Neural nets good at recognizing patterns $\Rightarrow$ very powerful extension.
- Beat best human player (Lee Sedol).

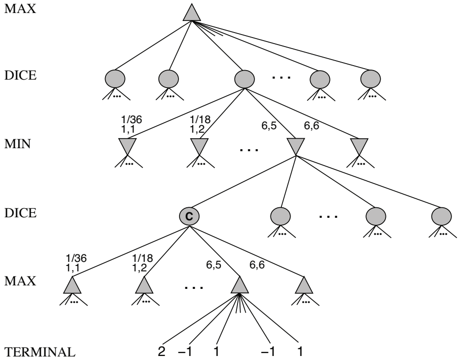
Deep Learning in AlphaGo



(from Nature 529, 2016)

Games with chance

Games include throwing a dice ⇒ game tree includes chance levels:



Outcomes known to all players.

Strategies

- Simultaneously learn player and opponent's strategies using repeated self-play.
- Optimize using *regret*: difference in reward between actual play and best possible actions (in hindsight).
  - play according to expected return
    = select action that minimizes expected regret.
    (expectiminimax)
  - minimize *exploitability* = possibility of opponent to benefit from "bad luck"
    = randomized strategy (as in adversarial bandits).
- Multiple sequences of moves need to be considered simultaneously ⇒ much more complex than ordinary game tree search.

Counterfactual Regret Minimization

How to compute probabilities in a randomized strategy:

- like Monte-Carlo search: estimate quality of moves by sampling outcomes of chance events.
- counterfactual regret $cf(a)$ = regret of having played another action a' instead of a.
- determine *randomized* strategy by *regret matching*: play action a with probability proportional to its counterfactual regret.
- will converge to an *equilibrium* where no agent can do better given the strategy of the other agent.

Information Sets

- Uncertain information can be known only to one agent: for example, cards dealt in poker game.

$\Rightarrow$ multiple *information sets* = sets of states with identical private information.

- Strategies of agent *a* can depend only on information set known to agent *a*, except...
- Opponent's choice of action may reveal its private information, but...
- Opponent could bluff to mislead.

Computer Poker

- Complexity in poker comes from uncertainty, not size of the game.
- Heads-up limit Texas hold'em: 10^{13} decision points, solved near-optimally using abstraction and CFR in 2012.
- Heads-up no limit Texas hold'em: 10^{161} decision points (similar to GO), best AI players (LIBRATUS, DeepStack) reliably beat best human players (2017).
- Libratus uses: game abstraction (blueprint solving), detailing subgames, and self-improvement to exploit opponent's weaknesses.
- DeepStack uses no abstraction, but approximates values of horizon states using a deep neural network.

Significant to real applications such as stock trading or fighting diseases.

Summary

- Limitations of reactive agents $\Rightarrow$ deliberative agents.
- Planning generates and optimizes strategy for a particular scenario.
- Can deal with bigger scenarios than MDP/POMDP.
- Planning with adversaries: game playing algorithms.
- Monte Carlo tree search scales to much larger problems.
- Games with uncertainty: learn probabilistic strategies with counterfactual regret minimization.