

Deliberative Agents

Boi Faltings

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

Deliberative architecture

Reactive architecture: difficult to

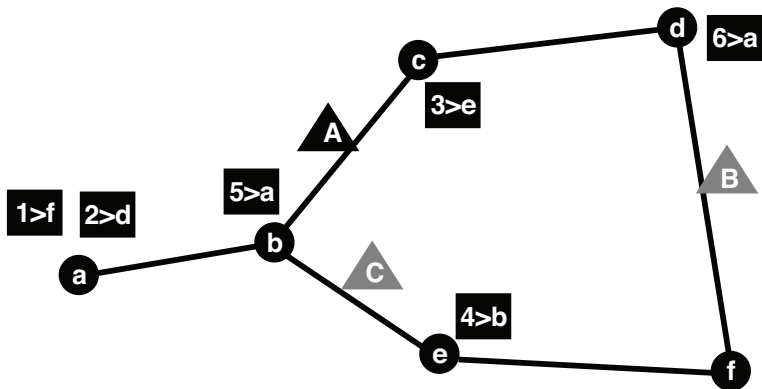
- plan over time
- act with varying goals (e.g. receive instructions)
- coordinate with other agents

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:

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

$$|\mathcal{S}| = 6^7 \cdot 2^6 = 17'915'904$$

$$|A| \simeq 10 \Rightarrow |T| = 3'209'796'161'372'160 \text{ entries}$$

$\Rightarrow$ 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

⇒ requires recomputing entire policy each time.

Reactive $\Rightarrow$ 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

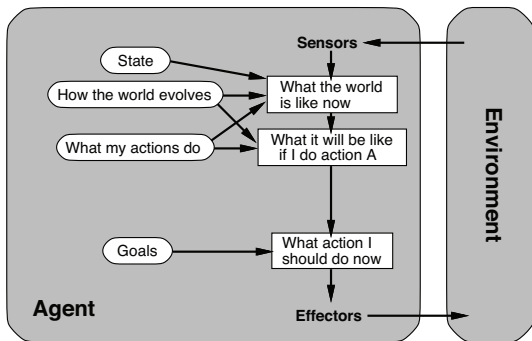
$\Rightarrow$ computing optimal actions for every state is a waste of effort!

Examples:

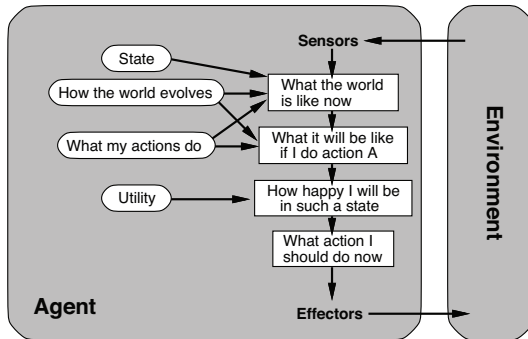
- Package 5 should never be moved to c,d,e, or f
- Package 3 will never be moved to a
- if A starts in a, it will not leave both packages 1 and 2 in a

$\Rightarrow$ unnecessary states

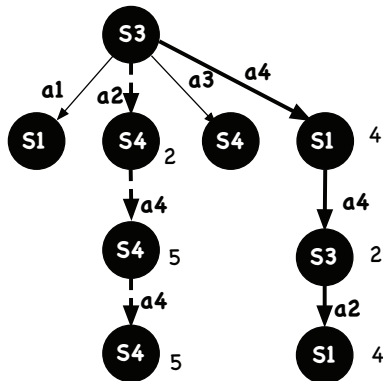
Agent with explicit goals



Utility-based agent



Deliberative solution of decision processes



Deliberative solution of decision processes

- Focus only on current state and successors.
- Myopic search does not find best plan.
- Need to search all possible sequences $\Rightarrow$ tree search.
- State space grows exponentially $\Rightarrow$ stop at bounded depth.

Example

Initial state:

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

Goal states:

$$s_{t1} : pos(A) = a, pos(1) = f, pos(2) = d, pos(3) = e, \dots$$

$$s_{t1} : pos(A) = b, pos(1) = f, pos(2) = d, pos(3) = e, \dots$$

...

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 up to level l (assuming no duplicates):

$$c(l) = \sum_{i=0}^l b^i = \frac{b^{l+1} - 1}{b - 1} < b^{l+1}$$

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

Formalization

Algorithm = search:

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

Search node:

$n = a \text{ state}$

Successor function:

$\text{succ}(n) = \text{list of nodes (states) reached from } n \text{ by simulating all different actions.}$

Properties (depth-first search)

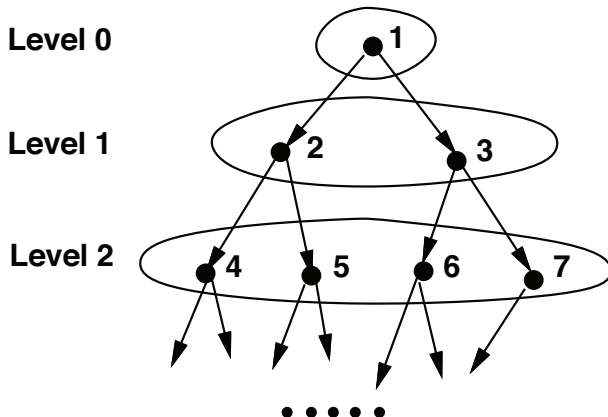
Advantage:

- 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: $Q \leftarrow (\text{InitialNode})$
- 3: **repeat**
- 4: $n \leftarrow \text{first}(Q)$, $Q \leftarrow \text{rest}(Q)$
- 5: **if** n is a goal state, **return** n
- 6: $S \leftarrow \text{succ}(n)$
- 7: $Q \leftarrow \text{append}(S, Q)$
- 8: **until** Q is empty
- 9: **return** FAIL

Breadth-first: exchange the order in step 7:

7. $Q \leftarrow \text{append}(Q, S)$

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.
- ⇒ impose a *depth limit* l :
never explore nodes at depth $> l$

Depth-Limited Search (DLS)

```
1: Function DLS (InitialNode,l)
2: depth-limit(InitialNode)  $\leftarrow$  l
3: Q  $\leftarrow$  (InitialNode)
4: repeat
5:   n  $\leftarrow$  first(Q), Q  $\leftarrow$  rest(Q)
6:   if n is a goal node, return n
7:   S  $\leftarrow$  succ(n)
8:   for nn  $\in$  S do
9:     depth-limit(nn)  $\leftarrow$  depth-limit(n)-1
10:  if depth-limit(n) > 0 then Q  $\leftarrow$  append(S,Q)
11: until Q is empty
12: return FAIL
```

Q: What is the right depth limit /?

Iterative Deepening

Increase the limit:

- 1: Function Iterative-deepening(InitialNode)
- 2: $l \leftarrow 2$
- 3: **repeat**
- 4: $solution \leftarrow DLS(InitialNode, l)$
- 5: $l \leftarrow l + 1$
- 6: **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:

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

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 .
- ⇒ any node that has higher cost than the best solution found so far can not lead to a better solution.
- ⇒ 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 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* .

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

Algorithm A* (best-first)

```
1: Function ASTAR (InitialNode)
2: Q  $\leftarrow$  (InitialNode)
3: C  $\leftarrow$  ()
4: repeat
5:   n  $\leftarrow$  first(Q), Q  $\leftarrow$  rest(Q)
6:   if n is a goal state, return n
7:   if n  $\notin$  C, or has lower cost than its copy in C then
8:     C  $\leftarrow$  append(n,C)
9:     S  $\leftarrow$  succ(n)
10:    S  $\leftarrow$  sort(S,f)
11:    Q  $\leftarrow$  merge(Q,S,f) {Q is ordered by  $f(n) = g(n) + h(n)$ }
12: until Q is empty
13: return FAIL
```

Example of a heuristic search

Search order (without heuristic):

(S1(0))

(S2(1),S3(2))

(S3(2),S4(3))

(S4(3),S5(5))

(S5(5),G(13))

(G(6),G(13))

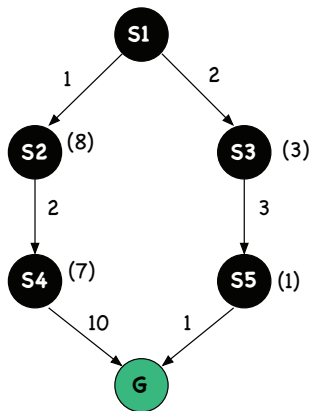
Search order (A*):

(S1(0+0)=0)

(S3(2+3=5),S2(1+8=9))

(S5(5+1=6),S2(9))

(G(6),S2(9))



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 plan found will always have the lowest possible cost, and is thus optimal.
- If the function $h(n)$ *overestimates* the true cost $h^*(n)$ remaining from n to a goal state, we can have:
$$g(n') + h(n') < g(n) + h(n) (> g(n) + h^*(n))$$
even if total cost of the 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 $h(n)$ **never over-estimates** the true cost.

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.

Planning with an adversary

Planning world may include other agents that

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

⇒ plan has to take their actions into account

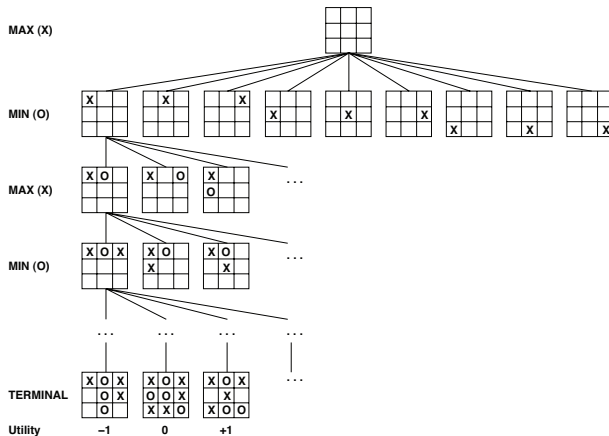
Purest form: games

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

⇒ possible games = search tree

Example of a game tree

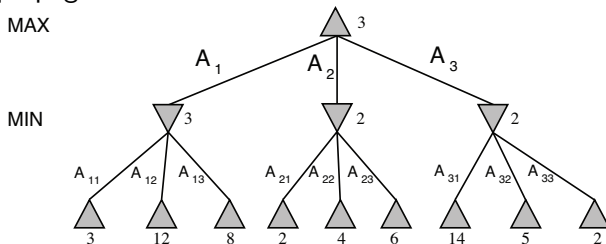


Minimax search

Each player controls only certain layers of the outcome $\Rightarrow$ assume:

- when agent is in control, it will **maximize** payoff
- when others are in control, they will **minimize** payoff

$\Rightarrow$ backpropagation to earlier states



Imperfect decisions

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

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

⇒ 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:

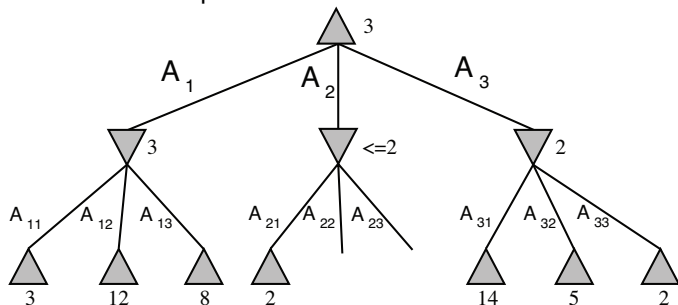
$$f = a_1 \cdot x_1 + a_2 \cdot x_2 + \dots + a_n \cdot x_n$$

Alpha-Beta pruning

Minimax search can be optimized:

MAX

MIN



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

Principle

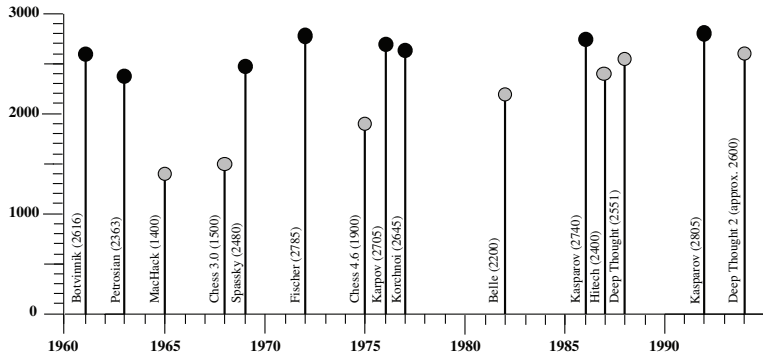
DFS algorithm keeps record of:

- 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

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

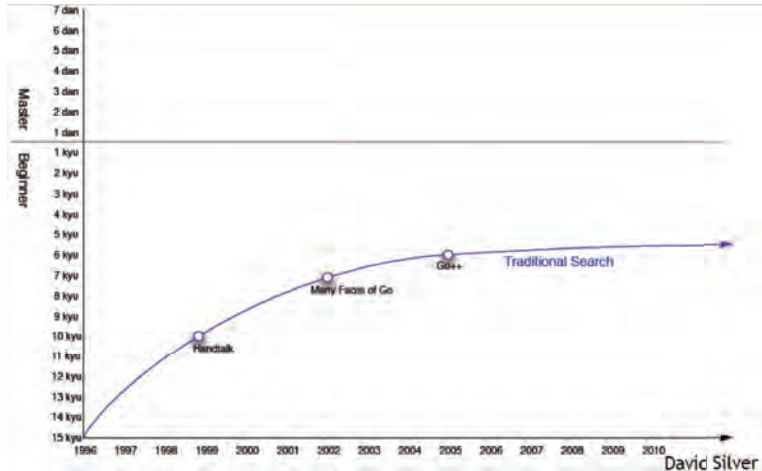
Alpha-beta search: combine global and local searches.

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

Selective search: encoding human expert-knowledge.

- Require human expertise.
- Require extensive manual tuning.

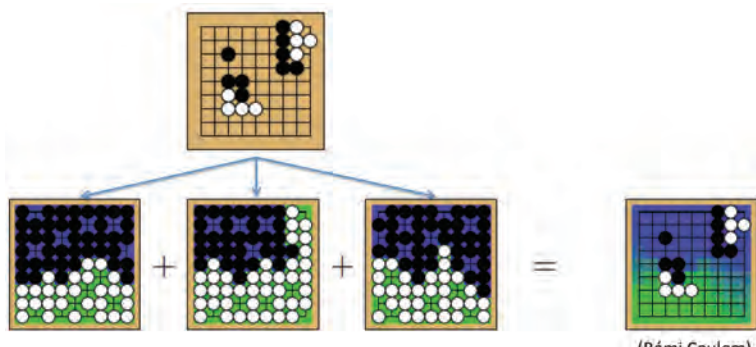
Go: Traditional Approaches



A New Approach for Go

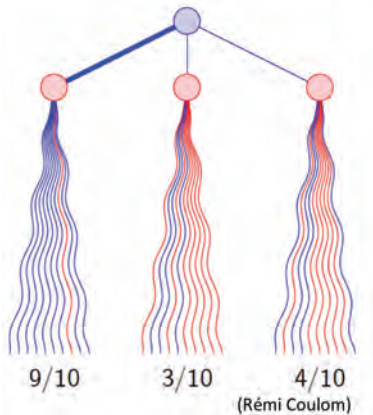
State evaluation using *Monte Carlo* Algorithm:

- 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.



Simple Monte Carlo Search

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

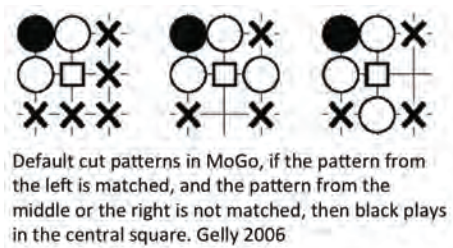


Roll-out policy

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)



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)

Upper Confidence Bound for Move i

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

exceeded with prob. $< \exp(-c^2 t/2)$

W_i = win with move i ; N_i = roll outs with move i ;

N = total roll outs; c = exploration parameter.

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

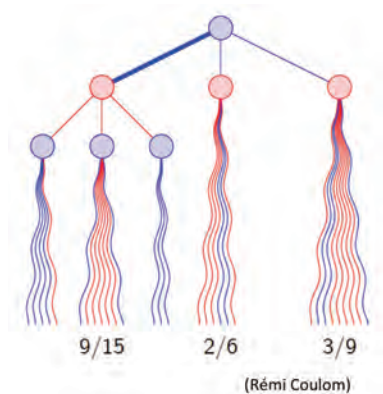
Monte Carlo Tree Search (MCTS)

A hybrid search algorithm:

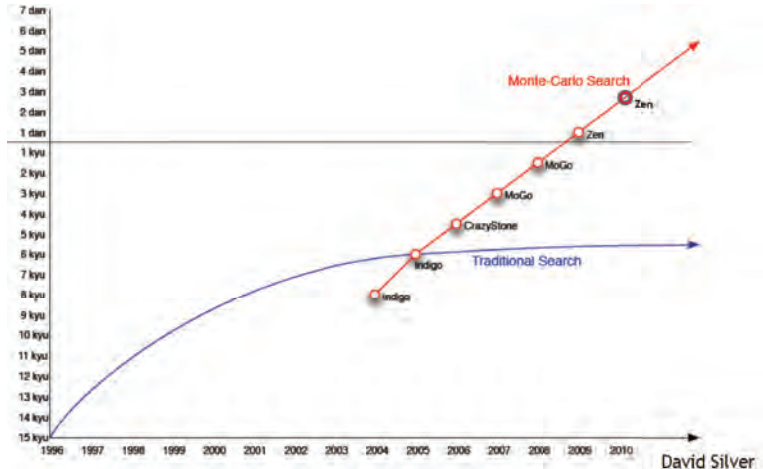
- Use UCT to build a game tree.
- Monte Carlo for evaluating leaf nodes.

Leading Computer Go programs based on MTCS:

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



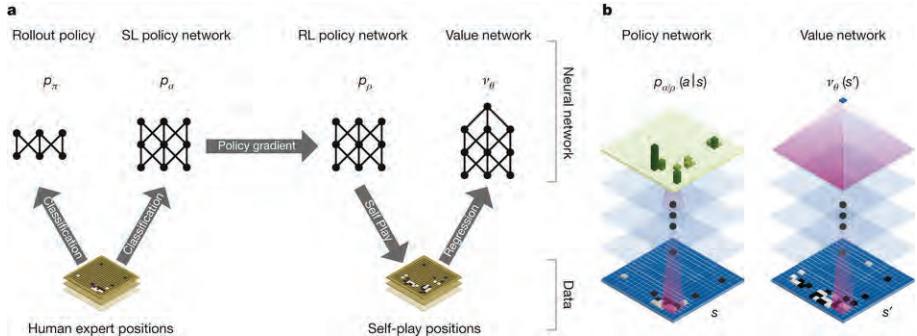
MCTS in Practice



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).

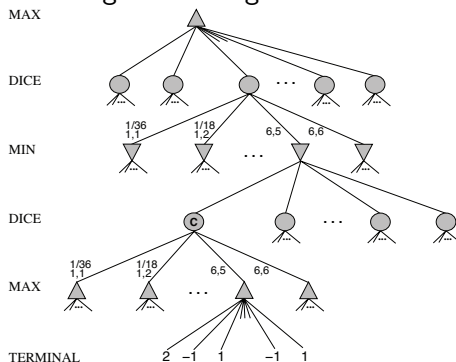
Deep Learning in AlphaGo



(from Nature 529, 2016)

Games with chance

Games include throwing a dice $\Rightarrow$ 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 $\Rightarrow$ 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.
- ⇒ multiple *information sets* = sets of states with identical private information.
- Strategies of agent a can depend only on information set known to agent a .
- ⇒ 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.