

Monte Carlo Search

Tristan Cazenave

LAMSADE CNRS
Université Paris-Dauphine
PSL

Tristan.Cazenave@dauphine.fr

Outline

- Monte Carlo Tree Search
- Nested Monte Carlo Search
- Nested Rollout Policy Adaptation
- Playout Policy Adaptation
- Zero Learning (Deep RL)
- Imperfect Information Games

INTELLIGENCE ARTIFICIELLE

Une approche ludique

Tristan Geurtsen



Monte Carlo Tree Search

Monte Carlo Go

- 1993 : first Monte Carlo Go program
 - Gobble, Bernd Bruegmann.
 - How nature would play Go ?
 - Simulated annealing on two lists of moves.
 - Statistics on moves.
 - Only one rule : do not fill eyes.
 - Result = average program for 9x9 Go.
 - Advantage : much more simple than alternative approaches.

Monte Carlo Go

- 1998 : first master course on Monte Carlo Go.
- 2000 : sampling based algorithm instead of simulated annealing.
- 2001 : Computer Go an AI Oriented Survey.
- 2002 : Bernard Helmstetter.
- 2003 : Bernard Helmstetter, Bruno Bouzy,
Developments on Monte Carlo Go.

Monte Carlo Phantom Go

- Phantom Go is Go when you cannot see the opponent's moves.
- A referee tells you illegal moves.
- 2005 : Monte Carlo Phantom Go program.
- Many Gold medals at computer Olympiad since then using flat Monte Carlo.
- 2011 : Exhibition against human players at European Go Congress.

UCT

- UCT : Exploration/Exploitation dilemma for trees [Kocsis and Szepesvari 2006].
- Play random random games (playouts).
- Exploitation : choose the move that maximizes the mean of the playouts starting with the move.
- Exploration : add a regret term (UCB).

UCT

- UCT : exploration/exploitation dilemma.
- Play the move that maximizes

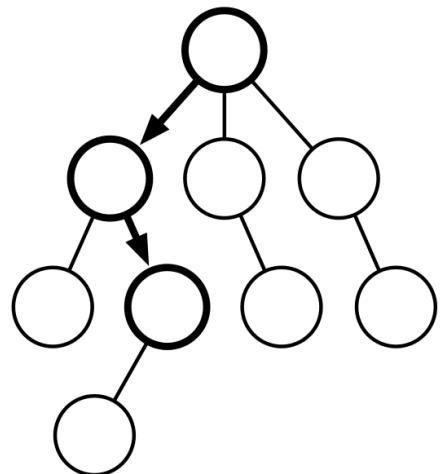
$$\frac{w_i}{n_i} + c \sqrt{\frac{\ln t}{n_i}}$$

In which

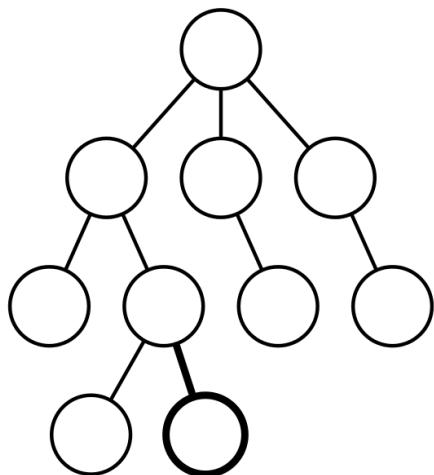
- w_i = number of wins after the i -th move
- n_i = number of simulations after the i -th move
- c = exploration parameter (theoretically equal to $\sqrt{2}$)
- t = total number of simulations for the parent node

UCT

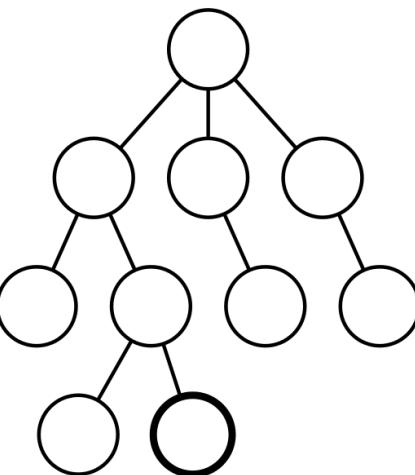
Selection



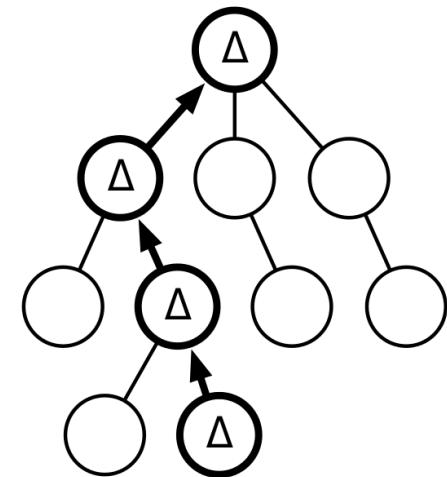
Expansion



Sampling



Backpropagation



Tree Policy

Default Policy



RAVE

- A big improvement for Go, Hex and other games is Rapid Action Value Estimation (RAVE) [Gelly and Silver 2007].
- RAVE combines the mean of the playouts that start with the move and the mean of the playouts that contain the move (AMAF).

RAVE

- Parameter β_m for move m is :

$$\beta_m \leftarrow p\text{AMAF}_m / (p\text{AMAF}_m + p_m + \text{bias} \times p\text{AMAF}_m \times p_m)$$

- β_m starts at 1 when no playouts and decreases as more playouts are played.

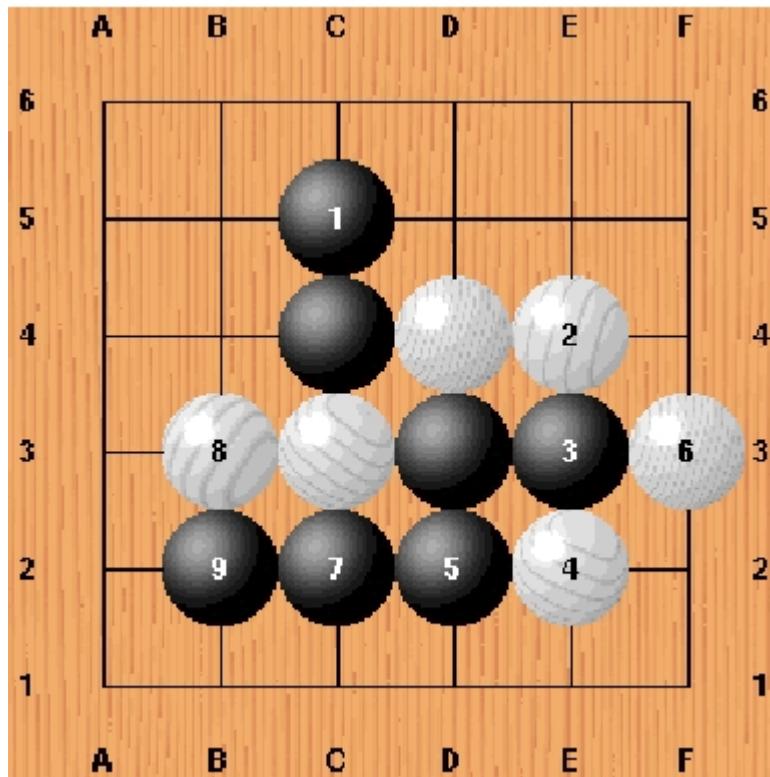
- Selection of moves in the tree :

$$\operatorname{argmax}_m ((1.0 - \beta_m) \times \text{mean}_m + \beta_m \times \text{AMAF}_m)$$

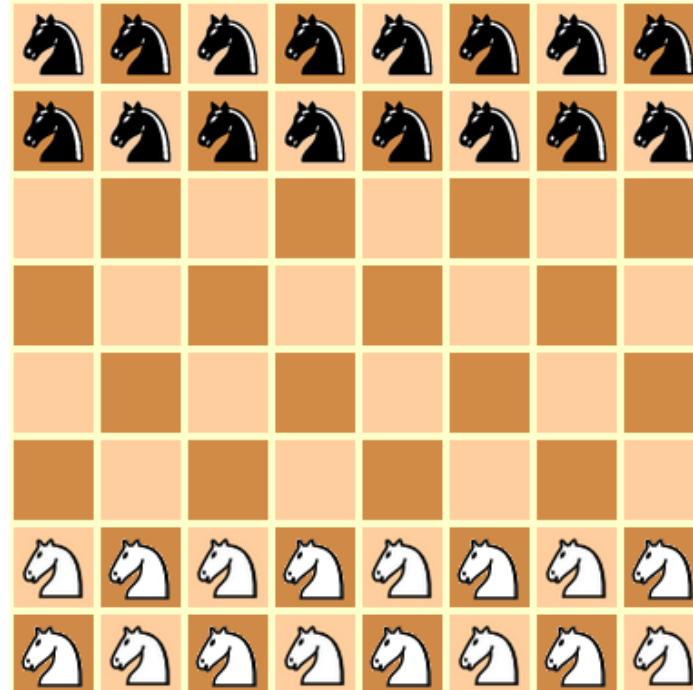
GRAVE

- Generalized Rapid Action Value Estimation (GRAVE) is a simple modification of RAVE.
- It consists in using the first ancestor node with more than n playouts to compute the RAVE values.
- It is a big improvement over RAVE for Go, Atarigo, Knightthrough and Domineering [Cazenave 2015].

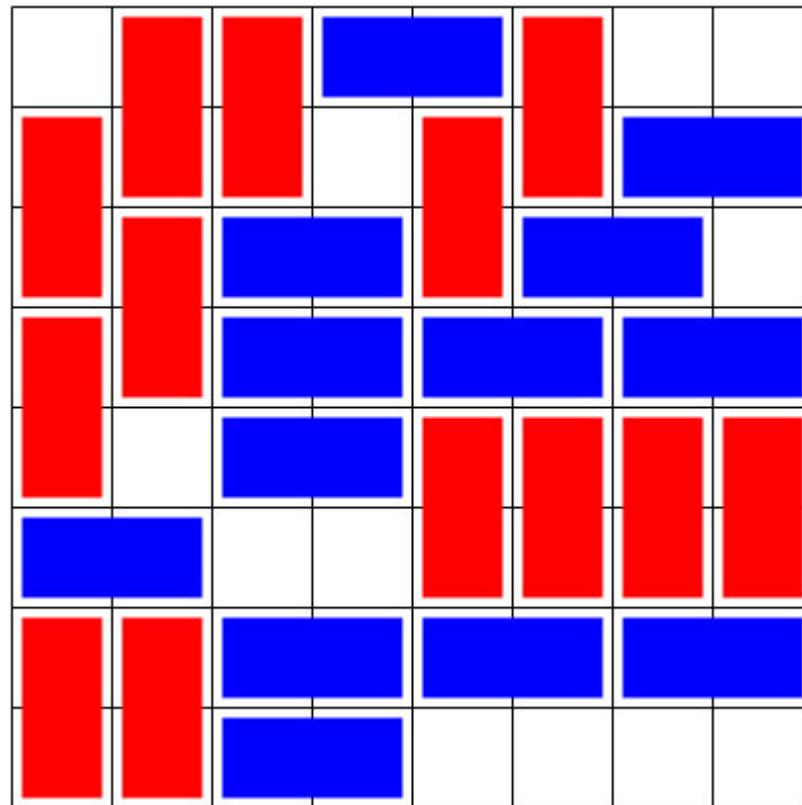
Atarigo



Knightthrough



Domineering



Go



RAVE vs UCT

Game	Score
Atarigo 8x8	94.2 %
Domineering	72.6 %
Go 9x9	73.2 %
Knightthrough	56.2 %
Three Color Go 9x9	70.8 %

GRAVE vs RAVE

Game	Score
Atarigo 8x8	88.4 %
Domineering	62.4 %
Go 9x9	54.4 %
Knightthrough	67.2 %
Three Color Go 9x9	57.2 %

Parallelization of MCTS

- Root Parallelization.
- Tree Parallelization (virtual loss).
- Leaf Parallelization.

MCTS



- Great success for the game of Go since 2007.
- Much better than all previous approaches to computer Go.

AlphaGo

Lee Sedol is among the strongest and most famous 9p Go player :



AlphaGo has won 4-1 against Lee Sedol in March 2016
AlphaGo Master wins 3-0 against Ke Jie, 60-0 against pros.
AlphaGo Zero wins 89-11 against AlphaGo Master in 2017.

General Game Playing



- General Game Playing = play a new game just given the rules.
- Competition organized every year by Stanford.
- Ary world champion in 2009 and 2010.
- All world champions since 2007 use MCTS.

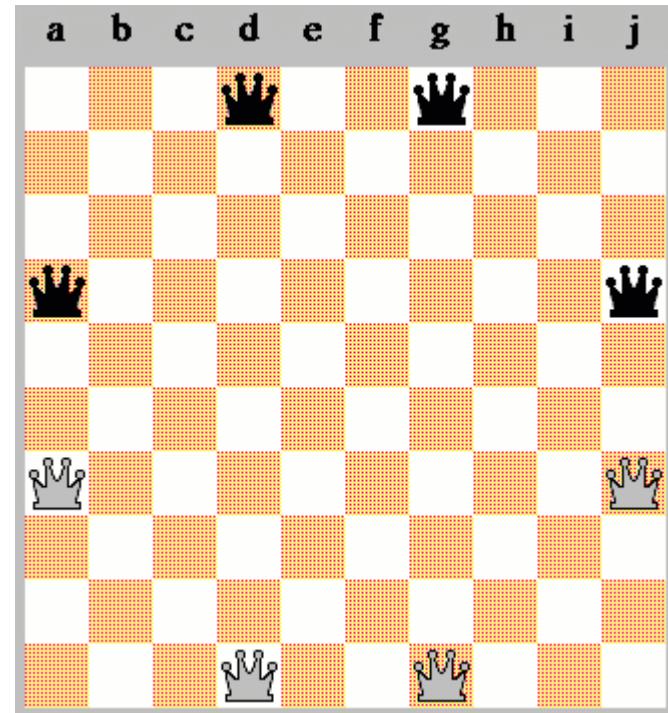
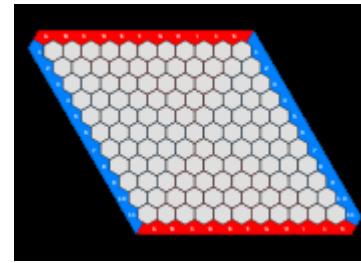
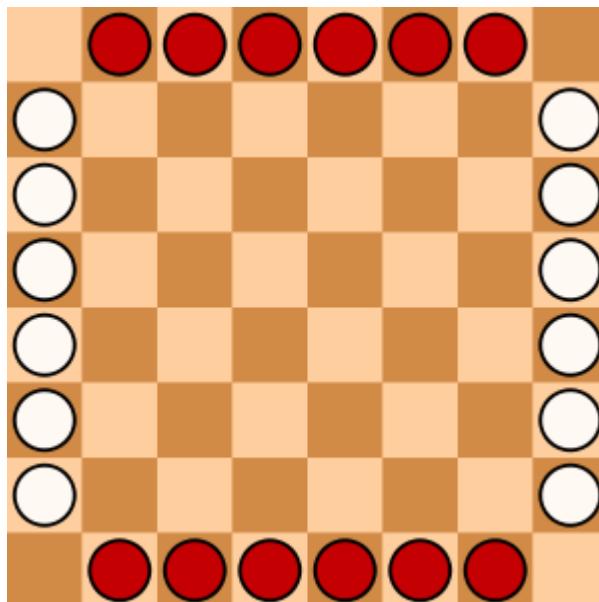
General Game Playing



- Eric Piette combined Stochastic Constraint Programming with Monte Carlo in WoodStock.
- World champion in 2016 (MAC-UCB-SYM).
- Detection of symmetries in the states.

Other two-player games

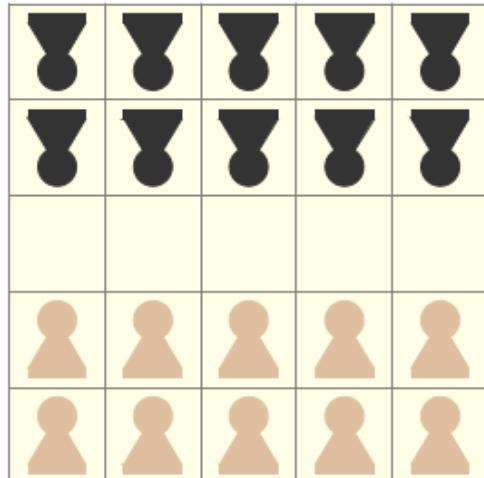
- Hex : 2009
- Amazons : 2009
- Lines of Action : 2009



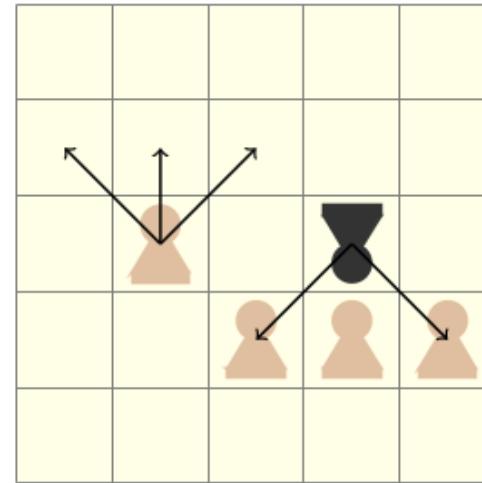
MCTS Solver

- When a subtree has been completely explored the exact result is known.
- MCTS can solve games.
- Score Bounded MCTS is the extension of pruning to solving games with multiple outcomes.

Breakthrough



(a) Starting position on size 5×5 .



(b) Possible movements.

- Write the Board and Move classes for Breakthrough 5x5.
- Write the function for the possible moves.
- Write a program to play random games at Breakthrough 5x5.

Playouts

- Write, in the Board class, a score function to score a game (1.0 if White wins, 0.0 else) and a terminal function to detect the end of the game.
- Write, in the Board class, a playout function that plays a random game from the current state and returns the result of the random game.

Flat Monte Carlo

- Keep statistics for all the moves of the current state.
- For each move of the current state, keep the number of playouts starting with the move and the number of playouts starting with the move that have been won.
- Play the move with the greatest mean when all the playouts are finished.

UCB

Choose the first move at the root according to UCB before each playout:

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

In which

- w_i = number of wins after the i -th move
- n_i = number of simulations after the i -th move
- c = exploration parameter (theoretically equal to $\sqrt{2}$)
- t = total number of simulations for the parent node

UCB vs Flat

- Make UCB with 1 000 playouts play 100 games against Flat with 1 000 playouts.
- Winrate ?
- Tune the UCB constant (hint 0.4).

Table de transposition

- Chaque état est associé à un hash code.
- On utilise le hachage de Zobrist.
- Chaque case de chaque pièce est associée à un nombre aléatoire différent.
- Le code d'un état est le XOR des nombres aléatoires des pièces présentes sur le plateau de jeu.
- Pourquoi utilise-t-on un XOR ?
- Combien utilise-t-on de nombres aléatoires différents pour coder un échiquier ?

Table de transposition

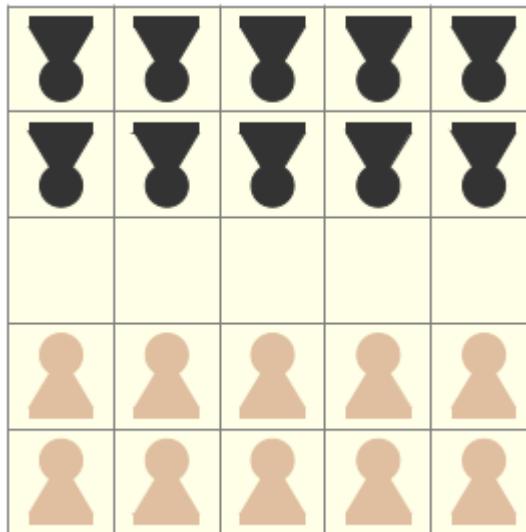
- On utilise un XOR parce que :
- Le XOR de valeurs bien réparties est bien réparti.
- Le XOR est une opération très rapide sur les bits.
- $(b \text{ XOR } a) \text{ XOR } a = b$
- Pour ajouter une pièce ou pour la retirer il suffit de faire le XOR avec le nombre aléatoire de la pièce
=> on peut calculer le code d'une position efficacement à partir du code du parent.

Table de transposition

- Aux échecs :
- pièces * cases = $12 * 64 = 768$
- droits de roque 4
- prise en passant 16
- trait 1
- total 789
- A Breakthrough 5x5 : 50 + 1 for the turn

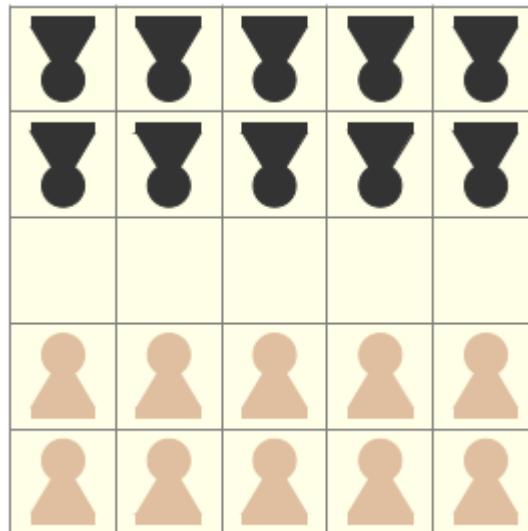
Table de transposition

- En supposant que les nombres aléatoires pour Breakthrough 5x5 vont de 1 à 25 pour noir et de 26 à 50 pour blanc (la première ligne vaut 1, 2, 3, 4, 5 pour noir et 26, 27, 28, 29, 30 pour blanc) :



- Quel est le hashcode h_1 de la position de départ ?
- Comment calculer le hash h_2 de la position où on a avancé le pion blanc le plus à gauche, à partir du hash h_1 de la position de départ ?

Table de transposition



$$h1 = 0$$

$$h2 = h1 \wedge 41 \wedge 36 = 13$$

Table de transposition

- Ecrire le code pour le tirage des nombres aléatoires associés aux cases et aux pions.
- Modifier le programme pour qu'un damier soit toujours associé à un hashcode de Zobrist calculé incrémentalement lorsqu'on joue un coup.

Transposition Table

- An entry of a state in the transposition table contains :
 - The hashcode of the stored state.
 - The total number of playouts of the state.
 - The number of playouts for each possible move.
 - The number of wins for each possible move.

Transposition Table

- First Option (C++ like) :
 - Write a class TranspoMonteCarlo containing the data associated to a state.
 - Write a class TableMonteCarlo that contains a table of list of entries.
 - Each entry is an instance of TranspoMonteCarlo. The size of the table is 65535. The index in the table of a hashcode h is $h \& 65535$.
 - The TableMonteCarlo class also contains the functions :
 - look (self, board) which returns the entry of board.
 - add (self, t) which adds en entry in the table.

Transposition Table

- Alternative : use a Python dictionary with the hash as a key and lists as elements.
- Each list contains 3 elements :
 - the total numbers of playouts,
 - the list of the number of playouts for each move,
 - the list of the number of wins for each move.
- Write a function that returns the entry of the transposition table if it exists or else None.
- Write a function that adds an entry in the transposition table.

UCT

```
procedure UCTSEARCH( $s_0$ )
  while time available do
    SIMULATE( $board$ ,  $s_0$ )
  end while
   $board.SetPosition(s_0)$ 
  return SELECTMOVE( $board$ ,  $s_0$ , 0)
end procedure
```

```
procedure SIMULATE( $board$ ,  $s_0$ )
   $board.SetPosition(s_0)$ 
   $[s_0, \dots, s_T] = \text{SIMTREE}(board)$ 
   $z = \text{SIMDEFAULT}(board)$ 
  BACKUP( $[s_0, \dots, s_T]$ ,  $z$ )
end procedure
```

UCT

```
procedure SIMTREE(board)
    c = exploration constant
    t = 0
    while not board.GameOver() do
        st = board.GetPosition()
        if st  $\notin$  tree then
            NEWNODE(st)
            return [s0, ..., st]
        end if
        a = SELECTMOVE(board, st, c)
        board.Play(a)
        t = t + 1
    end while
    return [s0, ..., st-1]
end procedure
```

UCT

```
procedure SIMDEFAULT(board)
  while not board.GameOver() do
    a = DEFAULTPOLICY(board)
    board.Play(a)
  end while
  return board.BlackWins()
end procedure
```

```
procedure BACKUP([s0, ..., sT], z)
  for t = 0 to T do
    N(st) = N(st) + 1
    N(st, at) += 1
    Q(st, at) +=  $\frac{z - Q(s_t, a_t)}{N(s_t, a_t)}$ 
  end for
end procedure
```

UCT

```
procedure SELECTMOVE(board, s, c)
    legal = board.Legal()
    if board.BlackToPlay() then
         $a^* = \operatorname{argmax}_{a \in \text{legal}} (Q(s, a) + c \sqrt{\frac{\log N(s)}{N(s, a)}})$ 
    else
         $a^* = \operatorname{argmin}_{a \in \text{legal}} (Q(s, a) - c \sqrt{\frac{\log N(s)}{N(s, a)}})$ 
    end if
    return a
end procedure
```

```
procedure NEWNODE(s)
    tree.Insert(s)
    N(s) = 0
    for all a  $\in \mathcal{A}$  do
        N(s, a) = 0
        Q(s, a) = 0
    end for
end procedure
```

Algorithm 4 The UCT algorithm

```
UCT (board, player, policy)
moves  $\leftarrow$  possible moves on board
if board is terminal then
    return winner (board)
end if
t  $\leftarrow$  entry of board in the transposition table
if t exists then
    bestValue  $\leftarrow -\infty$ 
    for m in moves do
        t  $\leftarrow t.totalPlayouts$ 
        w  $\leftarrow t.wins[m]$ 
        p  $\leftarrow t.playouts[m]$ 
        value  $\leftarrow \frac{w}{p} + c \times \sqrt{\frac{\log(t)}{p}}$ 
        if value > bestValue then
            bestValue  $\leftarrow value$ 
            bestMove  $\leftarrow m$ 
        end if
    end for
    play (board, bestMove)
    player  $\leftarrow$  opponent (player)
    res  $\leftarrow$  UCT (board, player, policy)
    update t with res
else
    t  $\leftarrow$  new entry of board in the transposition table
    res  $\leftarrow$  playout (board, player, policy)
    update t with res
end if
return res
```

UCT

- Exercise : write the Python code for UCT.
- The available functions that are problem specific are :
- board.playout () that returns the result of a playout.
- board.legalMoves () that returns the list of legal moves for the board.
- board.play (move) that plays the move on board.
- look (board) that returns the entry of the board in the transposition table.
- add (board) that adds an empty entry for the board in the transposition table.

AMAF

- All Moves As First (AMAF).
- AMAF calculates for each possible move of a state the average of the playouts that contain this move.

$$\Pr(z = 1 \mid s_t = s, a_t = a) = Q^\pi(s, a),$$

$$\Pr(z = 0 \mid s_t = s, a_t = a) = 1 - Q^\pi(s, a),$$

$$\Pr(z = 1 \mid s_t = s, \exists u \geq t \text{ s.t. } a_u = a) = \tilde{Q}^\pi(s, a),$$

$$\Pr(z = 0 \mid s_t = s, \exists u \geq t \text{ s.t. } a_u = a) = 1 - \tilde{Q}^\pi(s, a).$$

AMAF

- Exercise :
- Write a playout function memorizing the played moves.
- Add an integer code for moves in the Move class.
- Add AMAF statistics to the Transposition Table entries.
- Update the AMAF statistics of the Transposition Table.

RAVE

$$Q_\star(s,a) = \left(1 - \beta(s,a)\right) Q\left(s,a\right) + \beta(s,a) \tilde{Q}\left(s,a\right)$$

$$\mu = Q\left(s,a\right),$$

$$\tilde{\mu} = \tilde{Q}\left(s,a\right),$$

$$\mu_\star = Q_\star(s,a),$$

$$b = Q^\pi\left(s,a\right) - Q^\pi\left(s,a\right) = 0,$$

$$\tilde{b} = \tilde{Q}^\pi\left(s,a\right) - Q^\pi\left(s,a\right) = \tilde{B}(s,a),$$

$$b_\star = Q_\star^\pi\left(s,a\right) - Q^\pi\left(s,a\right),$$

$$\sigma^2 = \mathbb{E}\big[\big(Q\left(s,a\right) - Q^\pi\left(s,a\right)\big)^2 \bigm| N(s,a) = n\big],$$

$$\tilde{\sigma}^2 = \mathbb{E}\big[\big(\tilde{Q}\left(s,a\right) - \tilde{Q}^\pi\left(s,a\right)\big)^2 \bigm| \tilde{N}(s,a) = \tilde{n}\big],$$

$$\sigma_\star^2 = \mathbb{E}\big[\big(Q_\star(s,a) - Q_\star^\pi(s,a)\big)^2 \bigm| N(s,a) = n, \tilde{N}(s,a) = \tilde{n}\big],$$

$$e_\star^2 = \mathbb{E}\big[\big(Q_\star(s,a) - Q^\pi\left(s,a\right)\big)^2 \bigm| N(s,a) = n, \tilde{N}(s,a) = \tilde{n}\big].$$

RAVE

$$\begin{aligned} e_\star^2 &= \sigma_\star^2 + b_\star^2 \\ &= (1 - \beta)^2 \sigma^2 + \beta^2 \tilde{\sigma}^2 + (\beta \tilde{b} + (1 - \beta)b)^2 \\ &= (1 - \beta)^2 \sigma^2 + \beta^2 \tilde{\sigma}^2 + \beta^2 \tilde{b}^2. \end{aligned}$$

Differentiating with respect to β and setting to zero,

$$0 = 2\beta \tilde{\sigma}^2 - 2(1 - \beta)\sigma^2 + 2\beta \tilde{b}^2,$$

$$\beta = \frac{\sigma^2}{\sigma^2 + \tilde{\sigma}^2 + \tilde{b}^2}.$$

RAVE

$$\sigma^2 = \frac{Q^\pi(s, a)(1 - Q^\pi(s, a))}{N(s, a)} \approx \frac{\mu_\star(1 - \mu_\star)}{n},$$

$$\tilde{\sigma}^2 = \frac{\tilde{Q}^\pi(s, a)(1 - \tilde{Q}^\pi(s, a))}{\tilde{N}(s, a)} \approx \frac{\mu_\star(1 - \mu_\star)}{\tilde{n}},$$

$$\beta = \frac{\tilde{n}}{n + \tilde{n} + n\tilde{n}\tilde{b}^2/\mu_\star(1 - \mu_\star)}.$$

In roughly even positions, $\mu_\star \approx \frac{1}{2}$, we can further simplify the schedule,

$$\beta = \frac{\tilde{n}}{n + \tilde{n} + 4n\tilde{n}\tilde{b}^2}.$$

RAVE

```
procedure Mc-RAVE( $s_0$ )
  while time available do
    SIMULATE( $board, s_0$ )
  end while
   $board.SetPosition(s_0)$ 
  return SELECTMOVE( $board, s_0, 0$ )
end procedure

procedure SIMULATE( $board, s_0$ )
   $board.SetPosition(s_0)$ 
   $[s_0, a_0, \dots, s_T, a_T] = \text{SIMTREE}(board)$ 
   $[a_{T+1}, \dots, a_D], z = \text{SIMDEFAULT}(board, T)$ 
  BACKUP( $[s_0, \dots, s_T], [a_0, \dots, a_D], z$ )
end procedure
```

RAVE

```
procedure SIMDEFAULT(board, T)
    t = T + 1
    while not board.GameOver() do
        at = DEFAULTPOLICY(board)
        board.Play(at)
        t = t + 1
    end while
    z = board.BlackWins()
    return [aT+1, ..., at-1], z
end procedure
```

```
procedure SIMTREE(board)
    t = 0
    while not board.GameOver() do
        st = board.GetPosition()
        if st  $\notin$  tree then
            NEWNODE(st)
            at = DEFAULTPOLICY(board)
            return [s0, a0, ..., st, at]
        end if
        at = SELECTMOVE(board, st)
        board.Play(at)
        t = t + 1
    end while
    return [s0, a0, ..., st-1, at-1]
end procedure
```

RAVE

```
procedure SELECTMOVE(board, s)
    legal = board.Legal()
    if board.BlackToPlay() then
        return argmaxa ∈ legal EVAL(s, a)
    else
        return argmina ∈ legal EVAL(s, a)
    end if
end procedure

procedure EVAL(s, a)
    b = pretuned constant bias value
    
$$\beta = \frac{\tilde{N}(s,a)}{N(s,a) + \tilde{N}(s,a) + 4N(s,a)\tilde{N}(s,a)b^2}$$

    return  $(1 - \beta)Q(s,a) + \beta\tilde{Q}(s,a)$ 
end procedure
```

RAVE

procedure BACKUP($[s_0, \dots, s_T], [a_0, \dots, a_D], z$)

for $t = 0$ **to** T **do**

$N(s_t, a_t) += 1$

$Q(s_t, a_t) += \frac{z - Q(s_t, a_t)}{N(s_t, a_t)}$

for $u = t$ **to** D **step** 2 **do**

if $a_u \notin [a_t, a_{t+2}, \dots, a_{u-2}]$ **then**

$\tilde{N}(s_t, a_u) += 1$

$\tilde{Q}(s_t, a_u) += \frac{z - \tilde{Q}(s_t, a_t)}{\tilde{N}(s_t, a_t)}$

end if

end for

end for

end procedure

procedure NEWNODE($board, s$)

$tree.Insert(s)$

for all $a \in board.Legal()$ **do**

$N(s, a), Q(s, a), \tilde{N}(s, a), \tilde{Q}(s, a) = \text{HEURISTIC}(board, a)$

end for

end procedure

RAVE

- Exercise :
- Compute the AMAF statistics for each node.
- Modify the UCT code to implement RAVE.

GRAVE

- State of the art in General Game Playing (GGP)
- Best AI of the Ludii system (<https://ludii.games/>)
- Simple modification of RAVE
- Uses statistics both for Black and White at all nodes
- “In principle it is also possible to incorporate the AMAF values, from ancestor subtrees. However, in our experiments, combining ancestor AMAF values did not appear to confer any advantage.”

GRAVE

- Use the AMAF statistics of the last ancestor with more than n playouts instead of the AMAF statistics of the current node.
- More accurate when few playouts.
- Published at IJCAI 2015.
- GRAVE is a generalization of RAVE since GRAVE with $n=0$ is RAVE.

Algorithm 1 The GRAVE algorithm

```
GRAVE (board, tref)
  moves  $\leftarrow$  possible moves
  if board is terminal then
    return score(board)
  end if
  t  $\leftarrow$  entry of board in the transposition table
  if t exists then
    if t.layouts  $>$  ref then
      tref  $\leftarrow$  t
    end if
    bestValue  $\leftarrow -\infty$ 
    for m in moves do
      w  $\leftarrow$  t.wins[m]
      p  $\leftarrow$  t.layouts[m]
      wa  $\leftarrow$  tref.winsAMAF[m]
      pa  $\leftarrow$  tref.layoutsAMAF[m]
       $\beta_m \leftarrow \frac{pa}{pa + p + bias \times pa \times p}$ 
      AMAF  $\leftarrow \frac{wa}{pa}$ 
      mean  $\leftarrow \frac{w}{p}$ 
      value  $\leftarrow (1.0 - \beta_m) \times mean + \beta_m \times AMAF$ 
      if value  $>$  bestValue then
        bestValue  $\leftarrow$  value
        bestMove  $\leftarrow$  m
      end if
    end for
    play(board, bestMove)
    res  $\leftarrow$  GRAVE(board, tref)
    update t with res
  else
    t  $\leftarrow$  new entry of board in the transposition table
    res  $\leftarrow$  playout(player, board)
    update t with res
  end if
  return res
```

GRAVE

- Exercise :
- Modify the RAVE code to implement GRAVE.

Sequential Halving

- Sequential Halving [Karnin & al. 2013] is a bandit algorithm that minimizes the simple regret.
- It has a fixed budget of arm pulls.
- It gives the same number of playouts to all the arms.
- It selects the best half.
- Repeat until only one move is left

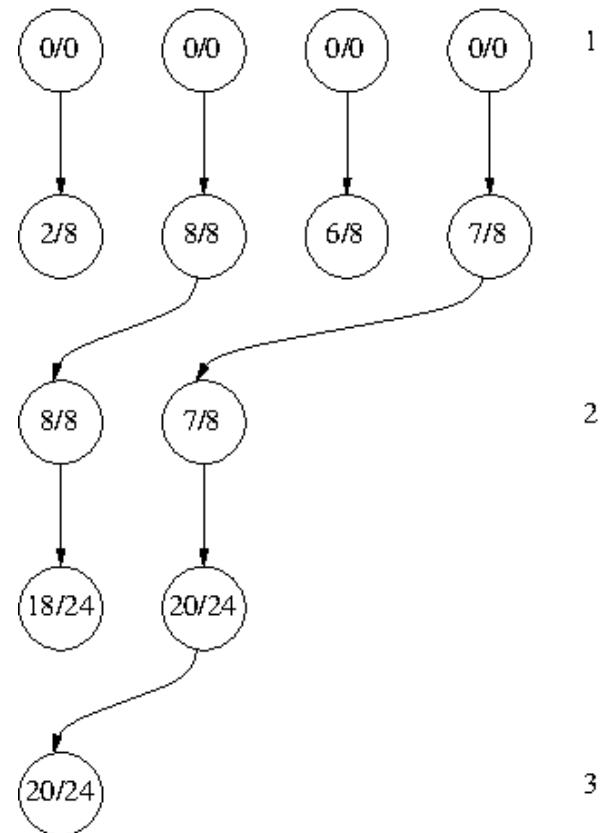
Sequential Halving

```
 $S \leftarrow [possibleMoves]$ 
while  $|S| > 1$  do
    for each move  $m$  in  $S$  do
        play ( $m$ )
        perform  $\left\lfloor \frac{budget}{|S| \times \lceil \log_2(|possibleMoves|) \rceil} \right\rfloor$  playouts
        undo ( $m$ )
    end for
     $S \leftarrow$  set of  $\left\lceil \frac{|S|}{2} \right\rceil$  moves in  $S$  with the largest
    empirical average
end while
```

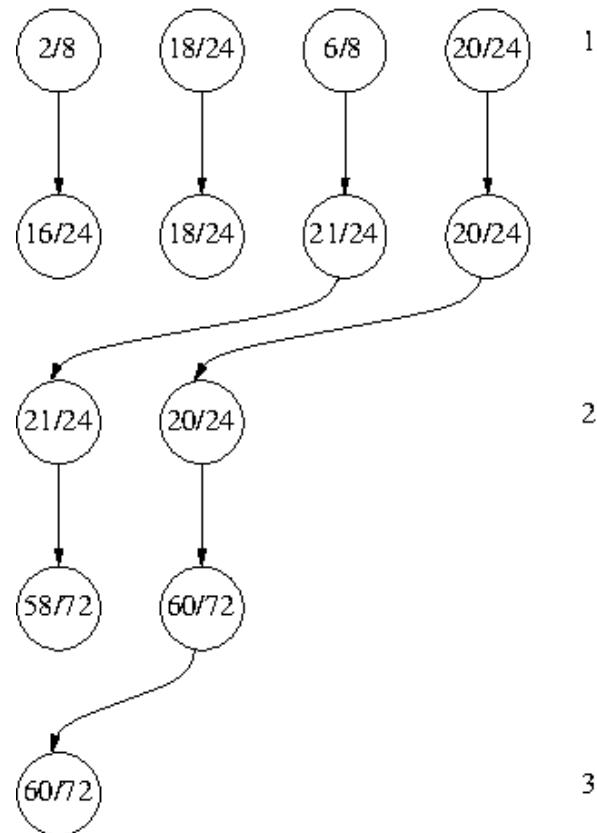
SHOT

- SHOT is the acronym for Sequential Halving Applied to Trees [Cazenave 2015].
- When the search comes back to a node it considers the spent budget and the new budget as a whole.
- It distributes the overall budget with Sequential Halving.

SHOT



SHOT



SHOT

- SHOT gives good results for Nogo.
- Combining SHOT and UCT :
 - SHOT near the root
 - UCT deeper in the tree
- The combination gives good results for Atarigo, Breakthrough, Amazons and partially observable games.

Sequential Halving

- Exercise:
- Write the code to perform Sequential Halving at the root on top of UCT.

SHUSS

- Sequential Halving combined with other statistics such as AMAF statistics.
- Instead of selecting the best half with the mean (μ_i), use:

$$\mu_i + c * \text{AMAF}_i / p_i$$

with p_i the number of playouts of move i and $c \geq 128$.

- Combining SH with AMAF = SHUSS (Sequential Halving Using Scores) [Fabiano et al. 2021]

SHUSS

Table 1: Comparison of Hybrid-SHUSS with AMAF score against RAVE.

Game	Playouts	0	128	256	512	1024	2048	4096	8192	16384	∞
Atarigo 7x7	10 000	44.2	47.2	49.6	50.2	50.0	49.6	45.2	47.8	46.4	45.2
Atarigo 9x9	10 000	35.6	41.4	40.0	38.2	41.0	41.2	43.4	41.4	36.4	40.2
Ataxx 8x8	10 000	30.2	33.6	35.2	34.2	42.0	46.2	55.0	62.4	62.0	71.8
Breakthrough 8x8	10 000	54.0	57.8	56.8	56.0	56.6	55.2	53.8	51.0	55.0	52.4
Domineering 8x8	10 000	41.4	47.8	44.8	49.0	46.2	47.2	46.2	45.6	43.0	42.4
Go 7x7	10 000	45.2	49.2	46.2	53.8	58.6	50.2	42.6	33.2	31.0	15.8
Go 9x9	10 000	43.4	53.2	58.2	52.2	50.8	43.8	35.6	26.4	19.0	12.2
Hex 11x11	10 000	15.8	43.0	43.4	51.4	48.4	50.2	46.4	46.6	43.4	42.6
Knightthrough 8x8	10 000	61.0	61.6	65.0	63.8	62.2	60.2	54.2	54.4	56.2	52.8
NoAtaxx 8x8	10 000	91.0	87.4	76.8	72.0	62.8	55.2	53.8	44.6	45.8	43.2
NoBreakthrough 8x8	10 000	37.8	40.8	44.0	46.2	51.4	44.2	46.4	44.0	50.0	46.6
NoDomineering 8x8	10 000	40.4	45.6	49.4	46.0	48.4	50.0	47.6	47.4	45.0	47.6
NoGo 7x7	10 000	38.8	40.8	45.6	44.0	50.8	47.6	50.8	49.4	47.6	51.8
NoGo 9x9	10 000	30.0	37.8	38.8	40.0	41.0	42.0	42.8	45.0	45.8	37.4
NoHex 11x11	10 000	46.4	48.0	48.6	49.0	49.2	48.6	48.6	49.2	48.8	49.2
NoKnightthrough 8x8	10 000	29.0	36.8	38.8	39.6	47.8	46.2	46.0	45.2	48.2	47.6

SHUSS

Algorithm 2 Sequential Halving USing Scores (SHUSS)

Parameter: cutting ratio λ, \tilde{t}'

Input: total budget T , set of arms S , online scores $\tilde{X}_r^{(i)}$

$S_0 \leftarrow S, T_0 \leftarrow T$

$R \leftarrow$ number of rounds before $|S_R| = 1$

for $r = 0$ **to** $R - 1$ **do**

$$t_r \leftarrow \lfloor \frac{T_r}{|S_r|(R-r)} \rfloor$$

$$T_{r+1} \leftarrow T_r - t_r |S_r|$$

sample each arm in S_r t_r times, giving an empirical mean $p_r^{(i)}$ to arm i out of t_r^+ trials

$$q_r^{(i)} = p_r^{(i)} + \frac{\tilde{t}'}{t_r^+} \tilde{X}_r^{(i)}$$

$S_{r+1} \leftarrow S_r$ without the fraction $1 - \lambda$ of the worst arms in terms of $q_r^{(i)}$

end for

Output: arm in S_R

SHUSS

- Exercise:

Write the code to perform SHUSS at the root on top of GRAVE.

PUCT

- MCTS used in AlphaGo and AlphaZero.
- A neural network gives a policy and a value.
- No playouts, evaluation with the value at the leaves.
- $P(s,a)$ = probability for move a of being the best.
- Bandit for the tree descent:

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

Algorithm 1 The PUCT algorithm.

```
1: PUCT (board, player)
2:   moves  $\leftarrow$  possible moves on board
3:   if board is terminal then
4:     return evaluation (board)
5:   end if
6:   t  $\leftarrow$  entry of board in the transposition table
7:   if t exists then
8:     bestValue  $\leftarrow -\infty$ 
9:     for m in moves do
10:      t  $\leftarrow$  t.totalLayouts
11:      mean  $\leftarrow$  t.mean[m]
12:      p  $\leftarrow$  t.layouts[m]
13:      prior  $\leftarrow$  t.prior[m]
14:      value  $\leftarrow$  mean + c  $\times$  prior  $\times \frac{\sqrt{t}}{p}$ 
15:      if value > bestValue then
16:        bestValue  $\leftarrow$  value
17:        bestMove  $\leftarrow$  m
18:      end if
19:    end for
20:    play (board, bestMove)
21:    player  $\leftarrow$  opponent (player)
22:    res  $\leftarrow$  PUCT (board, player)
23:    update t with res
24:   else
25:     t  $\leftarrow$  new entry of board in the transposition table
26:     res  $\leftarrow$  evaluation (board, player)
27:     update t
28:   end if
29:   return res
```

PUCT

- Exercise :

Modify the UCT code into PUCT.

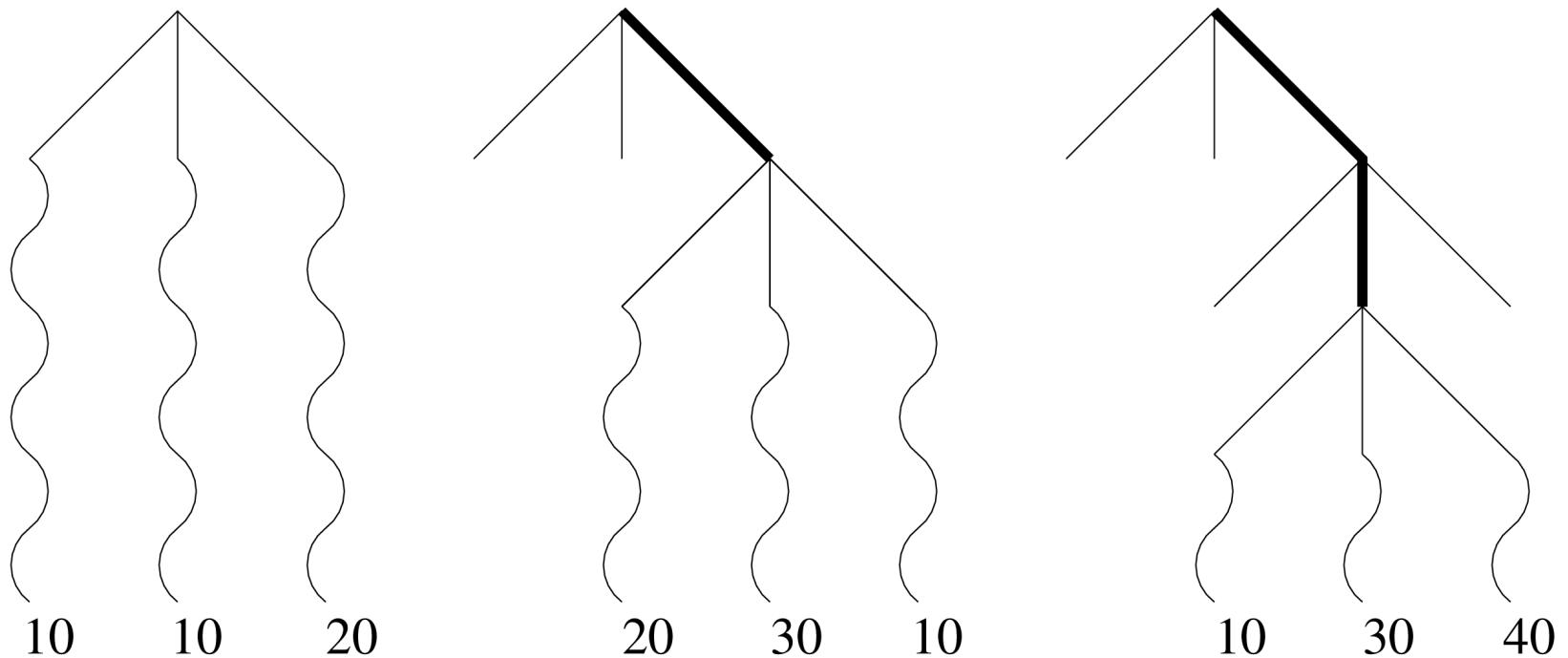
Suppose a random policy and a random value.

Nested Monte Carlo Search

Single Agent Monte Carlo

- UCT can be used for single-agent problems.
- Nested Monte Carlo Search often gives better results.
- Nested Rollout Policy Adaptation is an online learning variation that has beaten world records.

Nested Monte-Carlo Search



Nested Monte-Carlo Search

- Play random games at level 0
- For each move at level $n+1$, play the move then play a game at level n
- Choose to play the move with the greatest associated score
- Important : memorize and follow the best sequence found at each level

Algorithm 4 The NMCS algorithm.

NMCS (*state, level*)
if *level* == 0 **then**

return playout (*state, uniform*)

end if

BestSequenceOfLevel $\leftarrow \emptyset$

while *state* is not terminal **do**

for *m* in possible moves for *state* **do**

s \leftarrow play (*state, m*)

NMCS (*s, level - 1*)

update *BestSequenceOfLevel*

end for

bestMove \leftarrow move of the *BestSequenceOfLevel*

state \leftarrow play (*state, bestMove*)

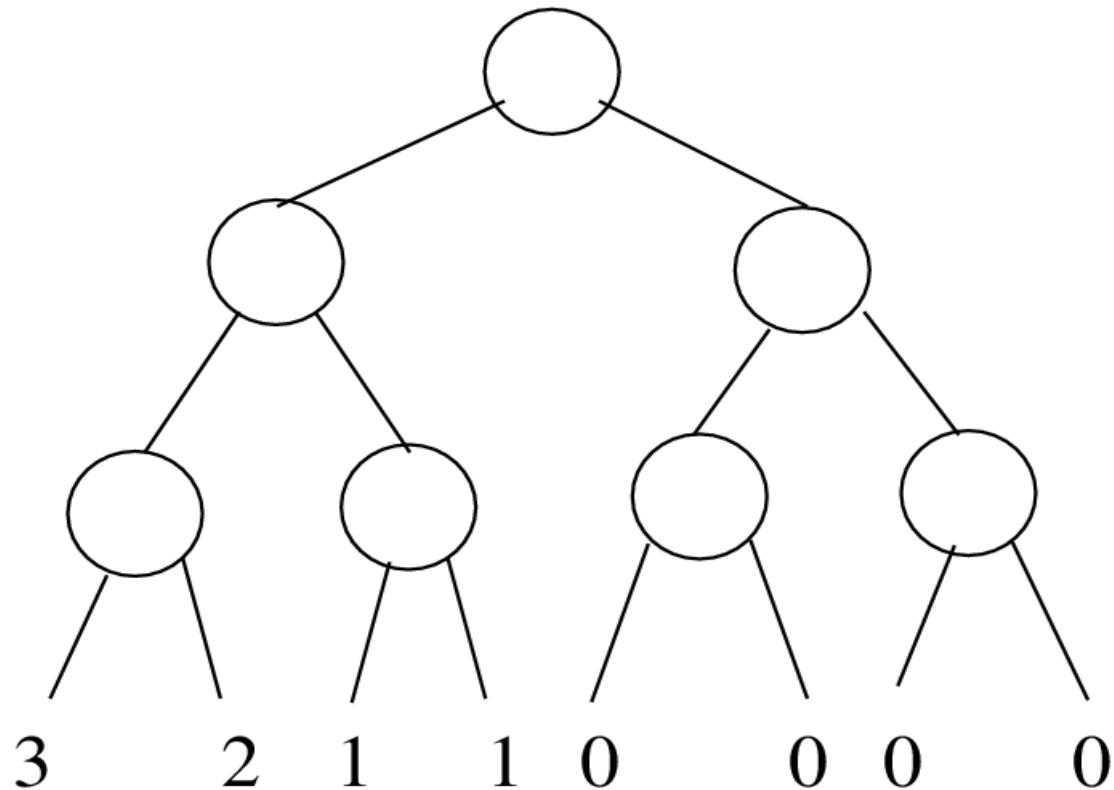
end while

Analysis

- Analysis on two very simple abstract problems.
- Search tree = binary tree.
- In each state there are only two possible moves: going to the left or going to the right.

Analysis

- The scoring function of the leftmost path problem consists in counting the number of moves on the leftmost path of the tree.

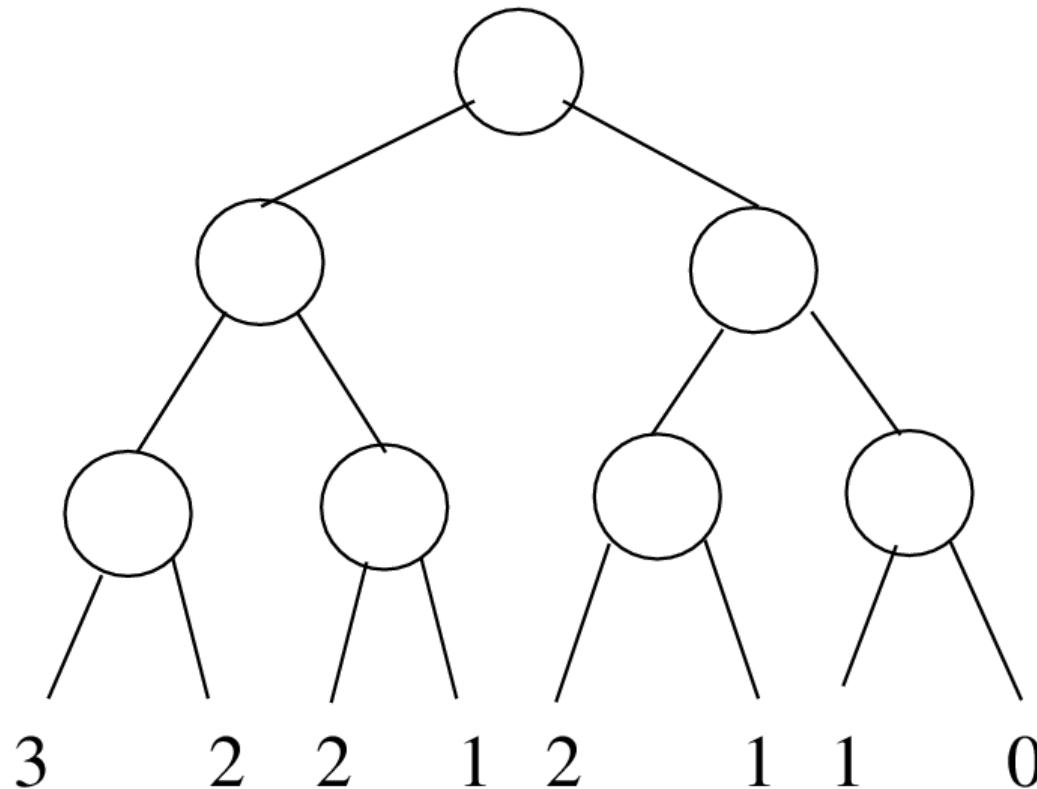


Analysis

- Sample search : probability 2^{-n} of finding the best score of a depth n problem.
- Depth-first search : one chance out of two of choosing the wrong move at the root, so the mean complexity $> 2^{n-2}$.
- A level 1 Nested Monte-Carlo Search will always find the best score, complexity is $n(n-1)$.
- Nested Monte-Carlo Search is appropriate for the leftmost path problem because the scores at the leaves are extremely correlated with the structure of the search tree.

Analysis

- The scoring function of the left move problem consists in counting the number of moves on the left.

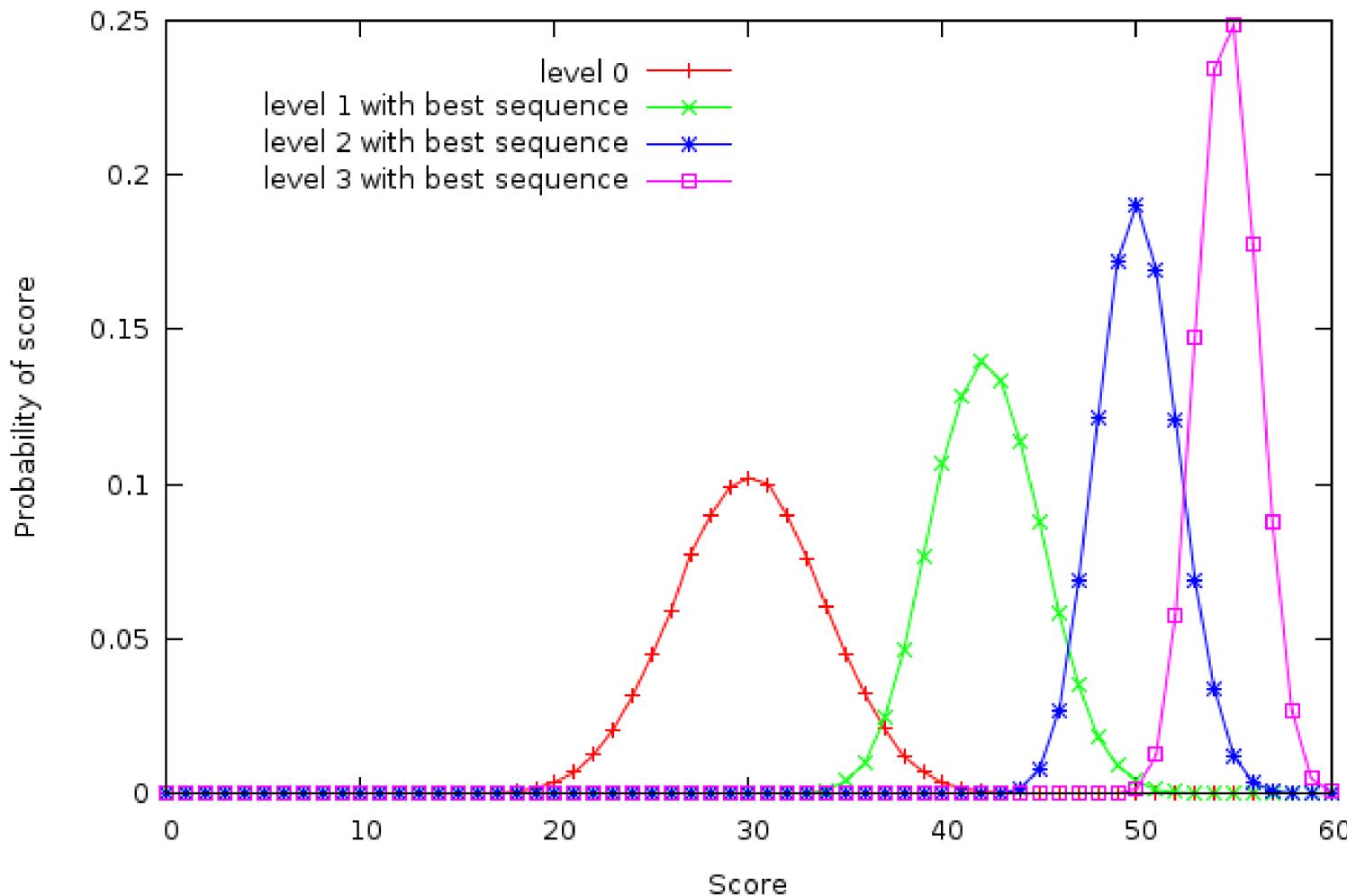


Analysis

- The probability distribution can be computed exactly with a recursive formula and dynamic programming.
- A program that plays the left move problems has also been written and results with 100,000 runs are within 1% of the exact probability distribution.

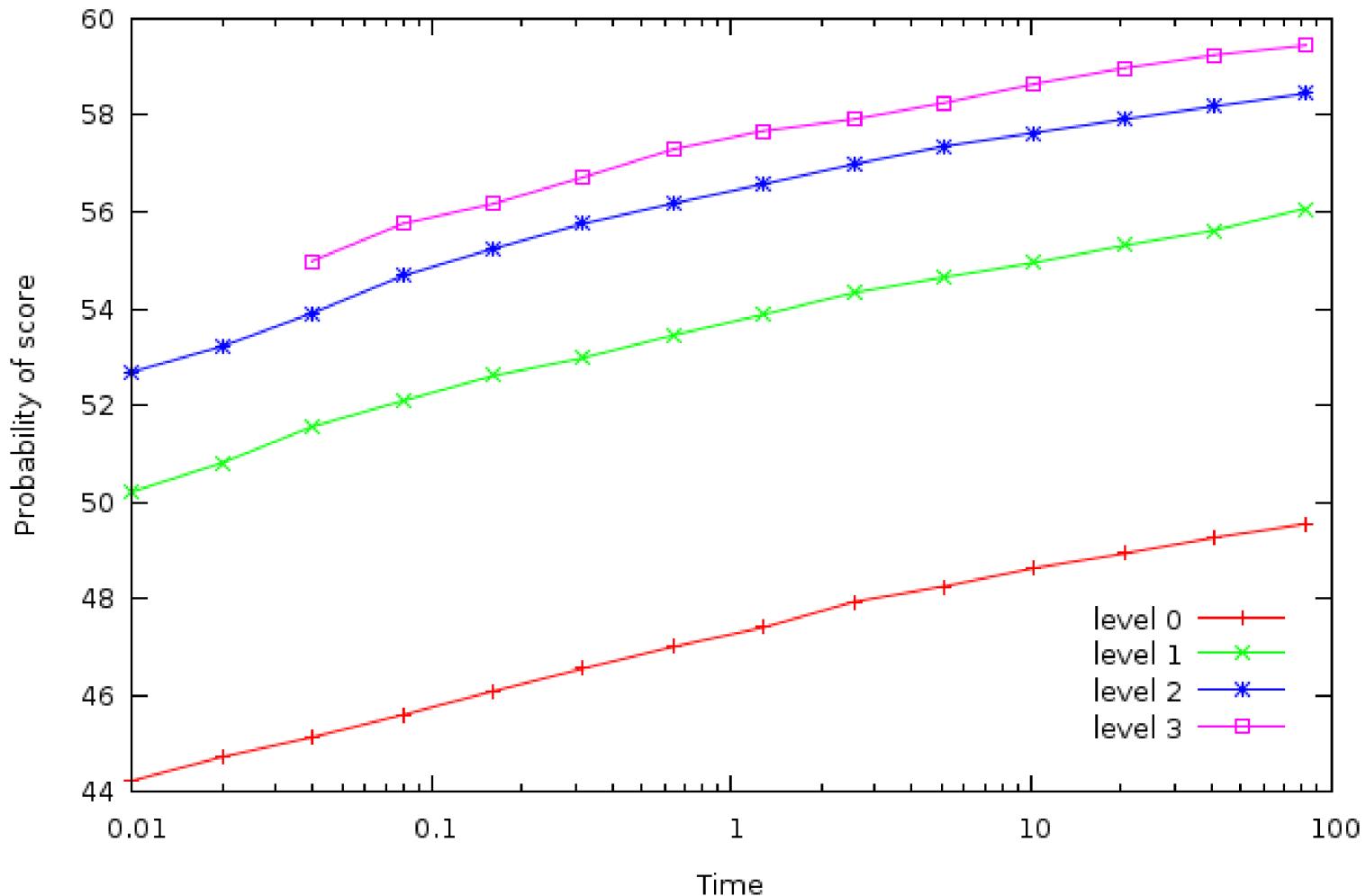
Analysis

- Distributions of the scores for a depth 60 problem.



Analysis

- Mean score in real time

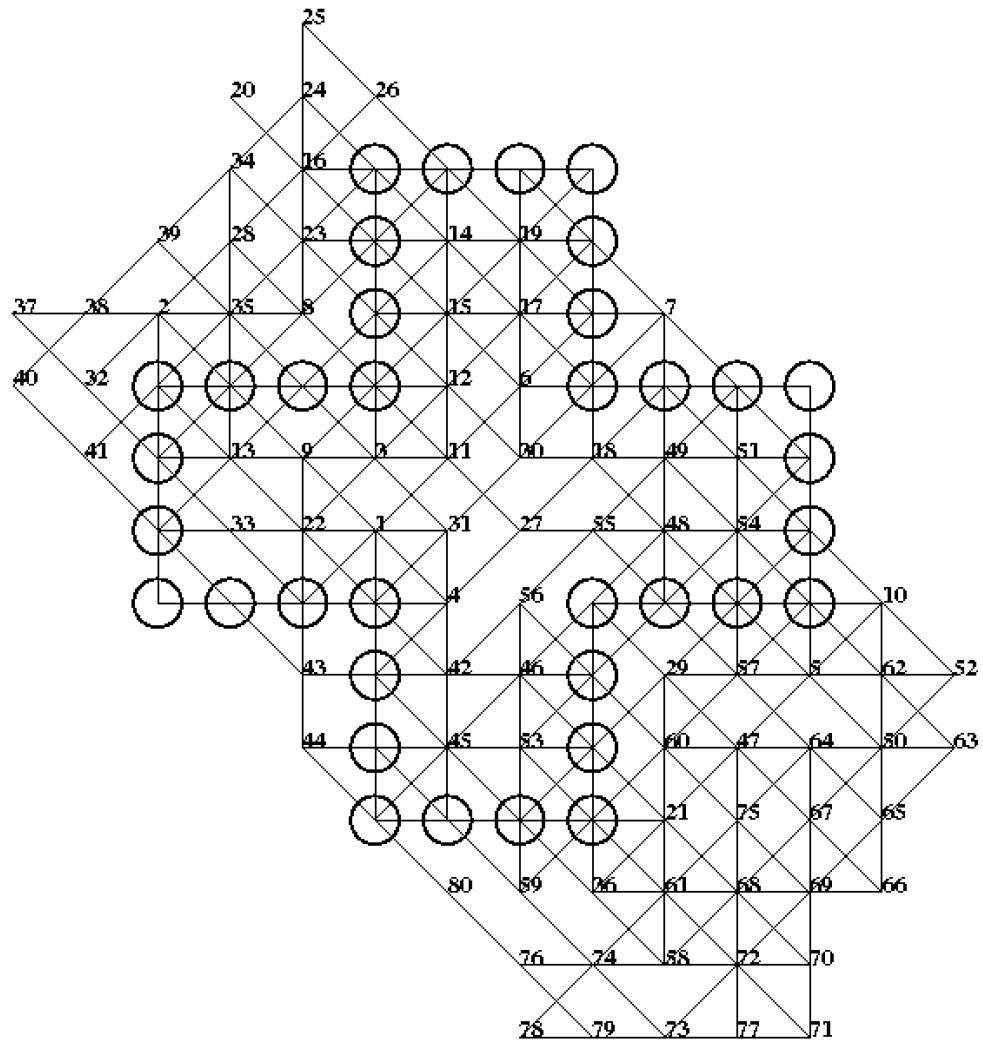


Morpion Solitaire

- Morpion Solitaire is an NP-hard puzzle and the high score is inapproximable within $n^{1-\epsilon}$
- A move consists in adding a circle such that a line containing five circles can be drawn.
- In the disjoint version a circle cannot be a part of two lines that have the same direction.
- Best human score is 68 moves.
- Level 4 Search => 80 moves, after 5 hours of computation on a 64 cores cluster.

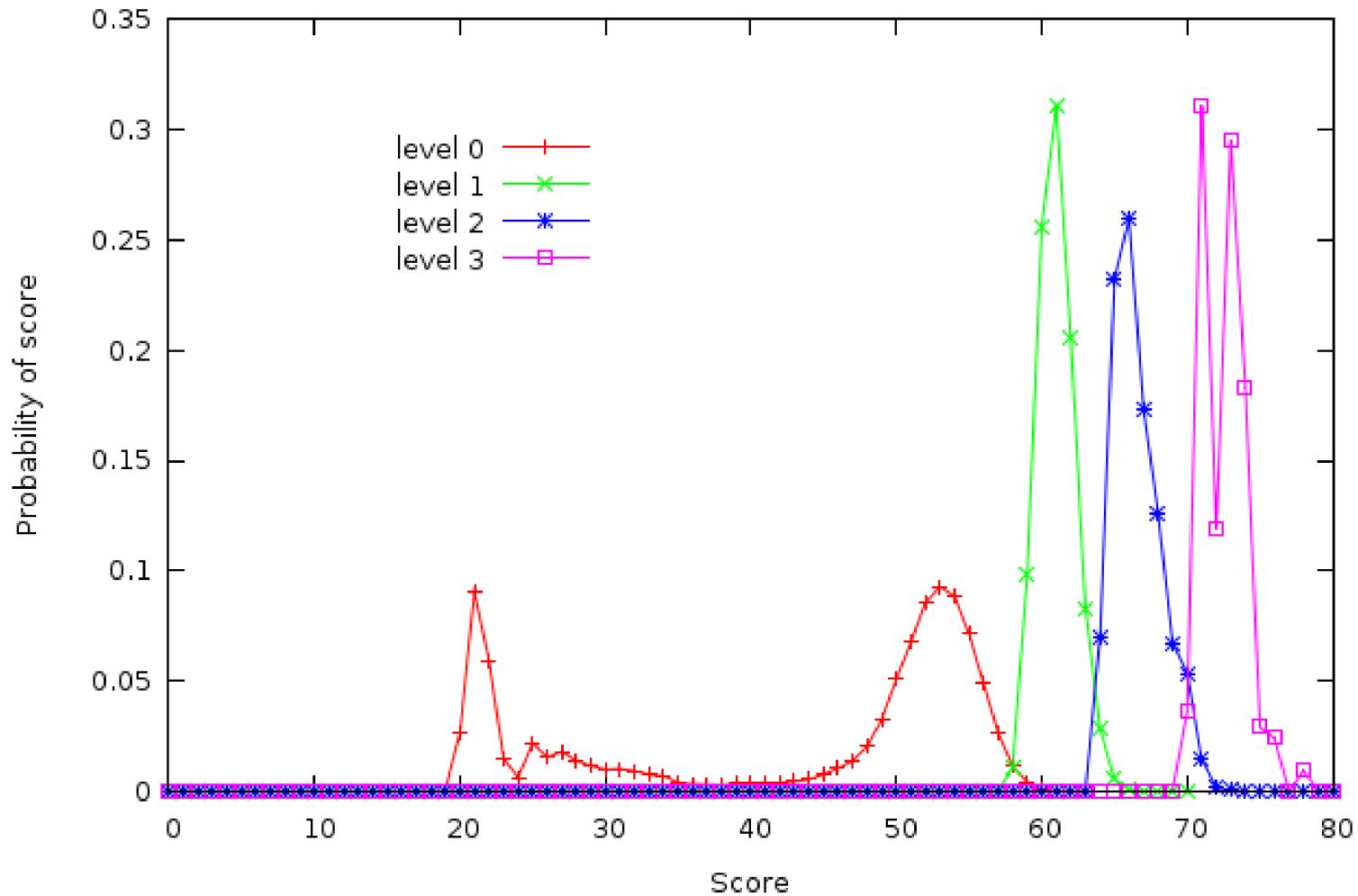
Morpion Solitaire

- 80 moves :



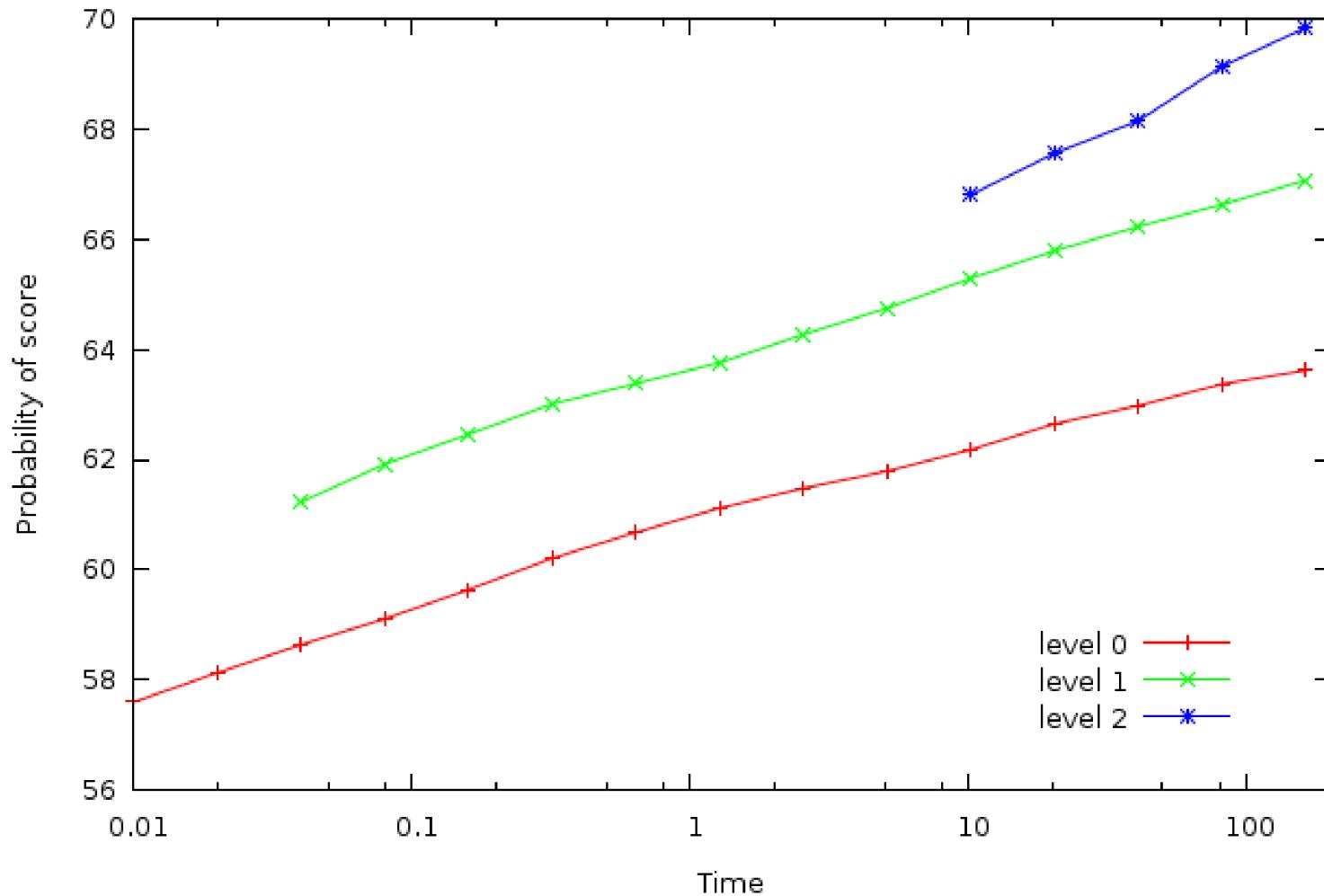
Morpion Solitaire

- Distribution of the scores



Morpion Solitaire

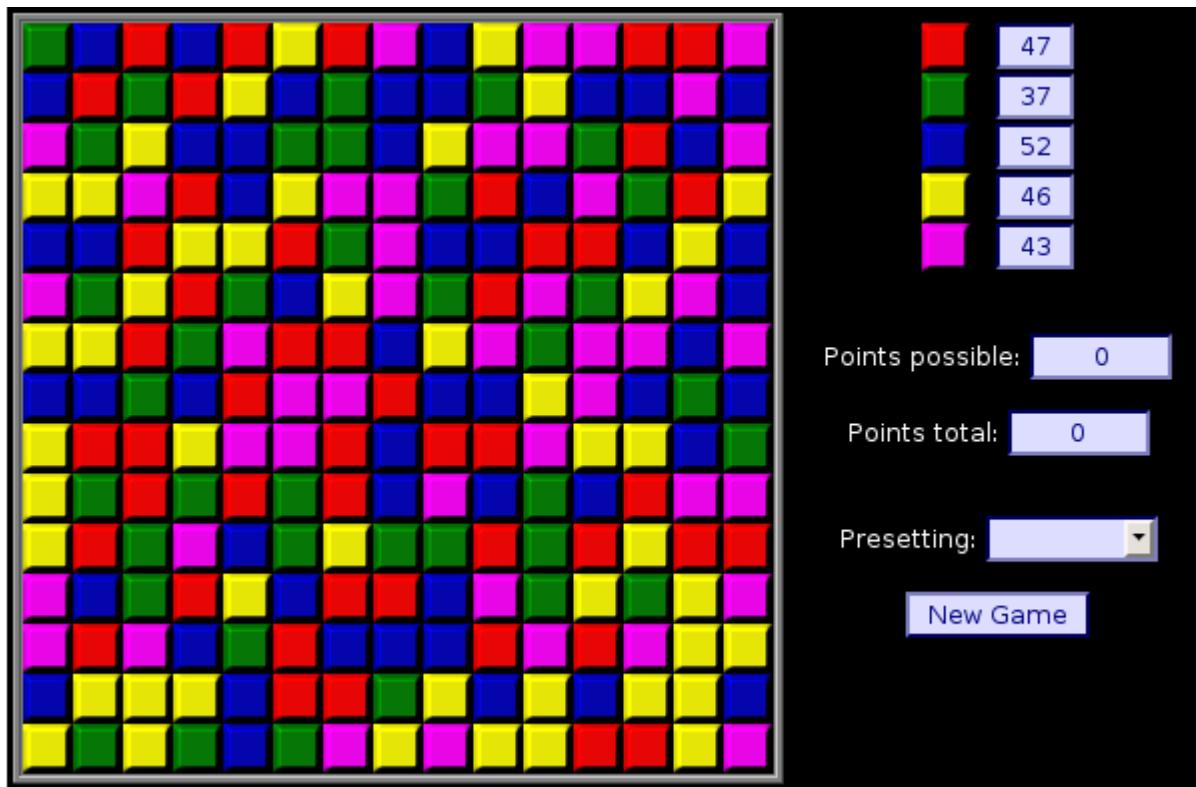
- Mean scores in real-time



SameGame

- NP-complete puzzle.
- It consists in a grid composed of cells of different colors. Adjacent cells of the same color can be removed together, there is a bonus of 1,000 points for removing all the cells.
- TabuColorRandom strategy: the color that has the most cells is set as the tabu color.
- During the playouts, moves of the tabu color are played only if there are no moves of the others colors or it removes all the cells of the tabu color.

Same Game



Same Game

- SP-MCTS = restarts of the UCT algorithm
- SP-MCTS scored 73,998 on a standard test set.
- IDA* : 22,354
- Darse Billings program : 72,816.
- Level 2 without memorization : 44,731
- Nested level 2 with memorization : 65,937
- Nested level 3 : 77,934

Application to Constraint Satisfaction

- A nested search of level 0 is a playout.
- A nested search of level 1 uses a playout to choose a value.
- A nested search of level 2 uses nested search of level 1 to choose a value.
- etc.
- The score is always the number of free variables.

Sudoku

- Sudoku is a popular NP-complete puzzle.
- 16x16 grids with 66% of empty cells.
- Easy-Hard-Easy distribution of problems.
- Forward Checking (FC) is stopped when the search time for a problem exceeds 20,000 s.

Sudoku

- FC : > 446,771.09 s.
- Iterative Sampling : 61.83 s.
- Nested level 1 : 1.34 s.
- Nested level 2 : 1.64 s.

Kakuro

	24	25	20	26	24
18
26
28
26
21

A 5x5 grid

Kakuro

	24	25	20	26	24
18	1	7	5	3	2
26	4	5	3	8	6
28	5	6	7	2	8
26	8	4	1	6	7
21	6	3	4	7	1

Solution

Kakuro

Algorithme	Solved problems	Time
Forward Checking	8/100	92,131.18 s.
Iterative Sampling	10/100	94,605.16 s.
Monte-Carlo level 1	100/100	78.30 s.
Monte-Carlo level 2	100/100	17.85 s.

8x8 Grids, 9 values, stop at 1,000 s.

Bus Regulation

- Goal : minimize passengers waiting times by making buses wait at a stop.
- Evaluation of an algorithm : sum of the waiting times for all passengers.

Regulation Algorithms

- Rule-based regulation: The waiting time depends on the number of stop with the next bus
- Monte-Carlo regulation : Choose the waiting time that has the best mean of random playouts
- Nested Monte-Carlo regulation : Use multiple levels of playouts

Rule-based regulation

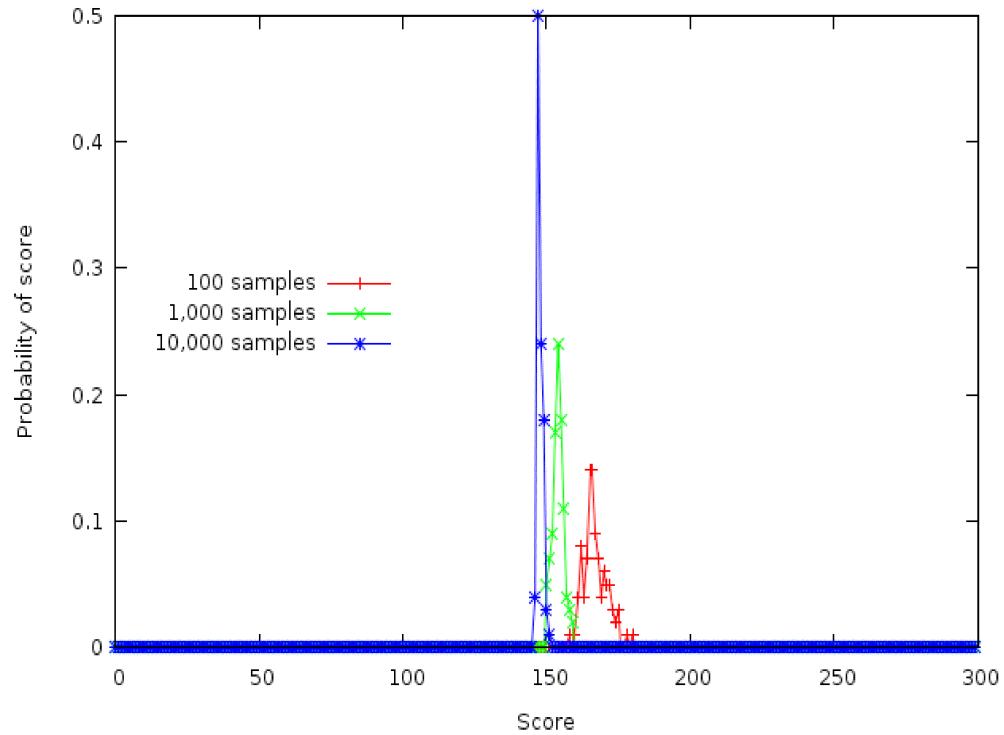
- δ : number of stops before the next bus.
- w : waiting time if the next bus is at more than δ .
- No regulation : 171
- Wait during 4 if more than 7 stops : 164

TABLE I
SCORES FOR DIFFERENT RULES

	w=1	w=2	w=3	w=4
$\delta = 4$	171	192	193	199
$\delta = 5$	171	191	192	195
$\delta = 6$	171	175	176	198
$\delta = 7$	171	169	166	164
$\delta = 8$	171	169	167	165
$\delta = 9$	171	169	167	166
$\delta = 10$	171	170	170	170

Monte-Carlo Regulation

- 165 for $N = 100$
- 154 for $N = 1000$
- 147 for $N = 10000$

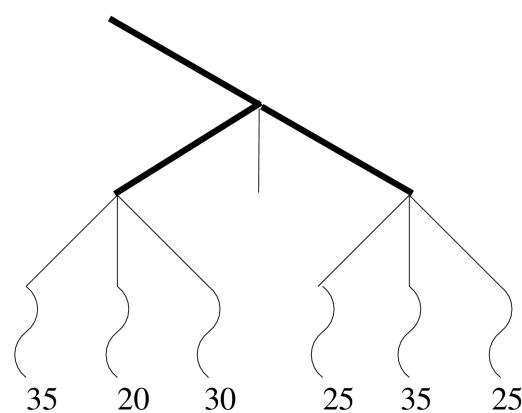
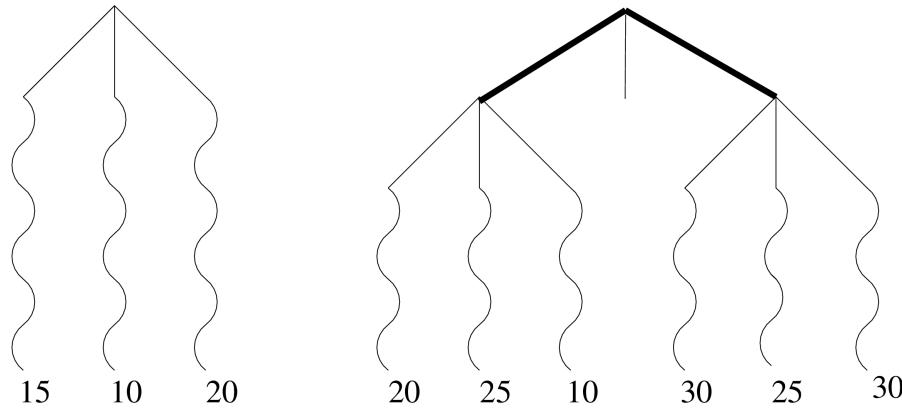


better than rule-based regulation (164).

Parallel Nested Monte-Carlo Search

- Play the highest level sequentially
- Play the lowest levels in parallel
- Speedup = 56 for 64 cores at Morpion Solitaire
- A more simple parallelization : play completely different searches in parallel (i.e. use a different seed for each search).

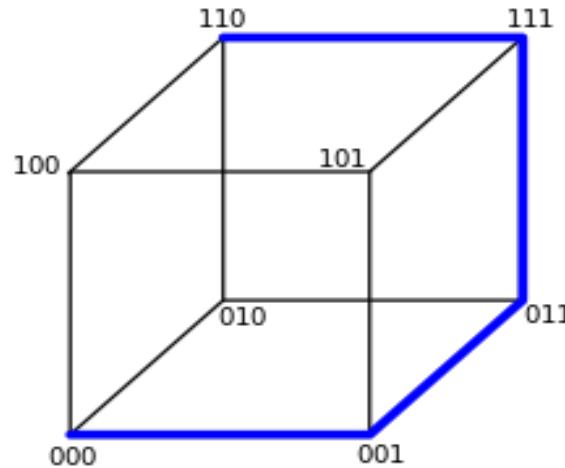
Monte Carlo Beam Search



Single-Agent General Game Playing

- Nested Monte-Carlo search gives better results than UCT on average.
- For some problems UCT is better.
- Ary searches with both UCT and Nested Monte-Carlo search and plays the move that has the best score.

Snake in the box

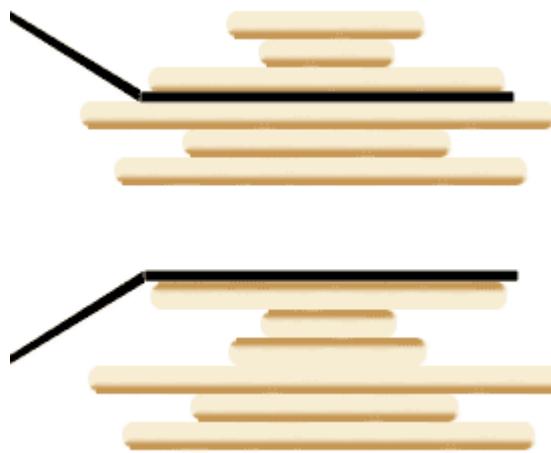


- A path such that for every node only two neighbors are in the path.
- Applications: Electrical engineering, coding theory, computer network topologies.
- World records with NMCS [Kinny 2012].

Multi-agent pathfinding

- Find routes for the agents avoiding collisions.
- Monte Carlo Fork Search enables to branch in the playouts.
- It solves difficult problems faster than other algorithms [Bouzy 2013].

The Pancake Problem

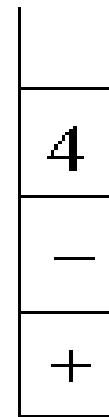
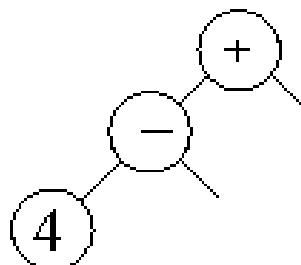


- Nested Monte Carlo Search has beaten world records using specialized playout policies [Bouzy 2015].

Software Engineering

- Search based software testing [Feldt and Poulding 2015].
- Heuristic Model Checking [Poulding and Feldt 2015].
- Generating structured test data with specific properties [Poulding and Feldt 2014].

Monte-Carlo Discovery of Expressions



- Possible moves are pushing atoms.
- Evaluation of a complete expression.
- Better than Genetic Programming for some problems [Cazenave 2010, 2013].

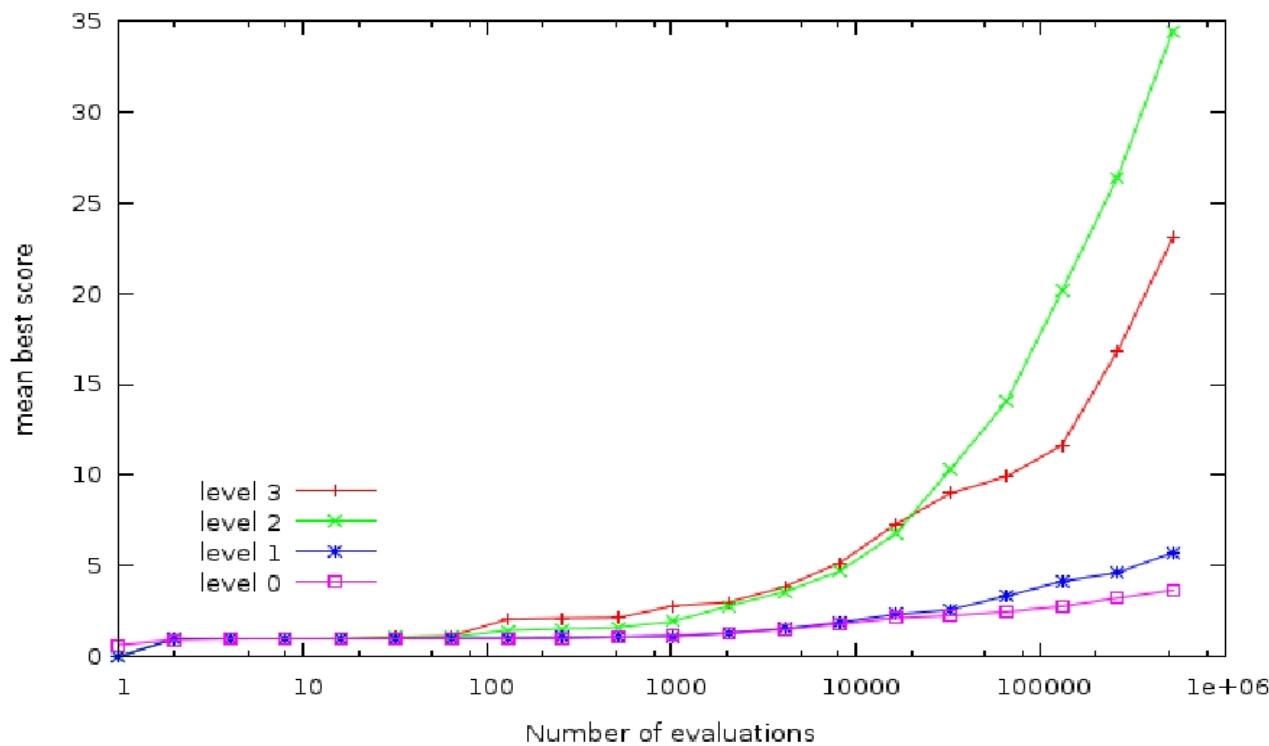
Monte-Carlo Discovery of Expressions

Prime Generating Polynomials:

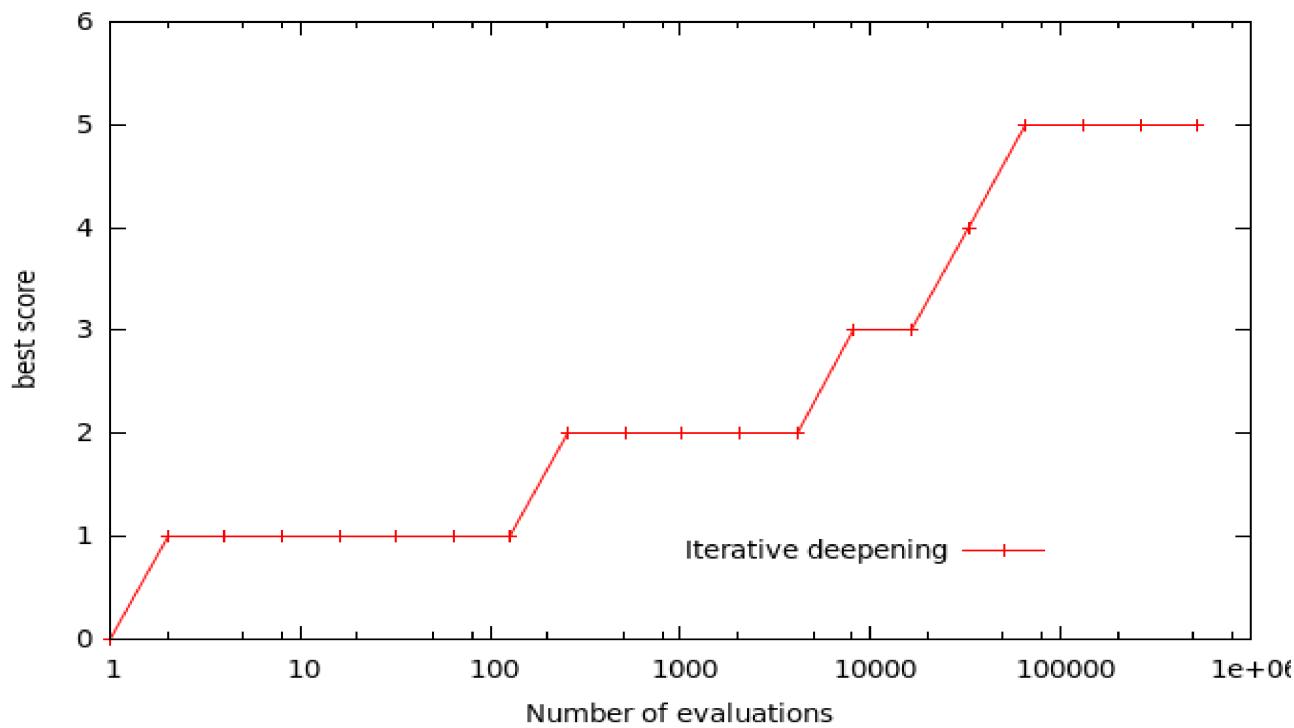
The score of an expression is the number of different primes it generates in a row for integer values of x starting at zero and increasing by one at each step.

Nested Monte-Carlo search is better than UCT and Iterative Deepening search.

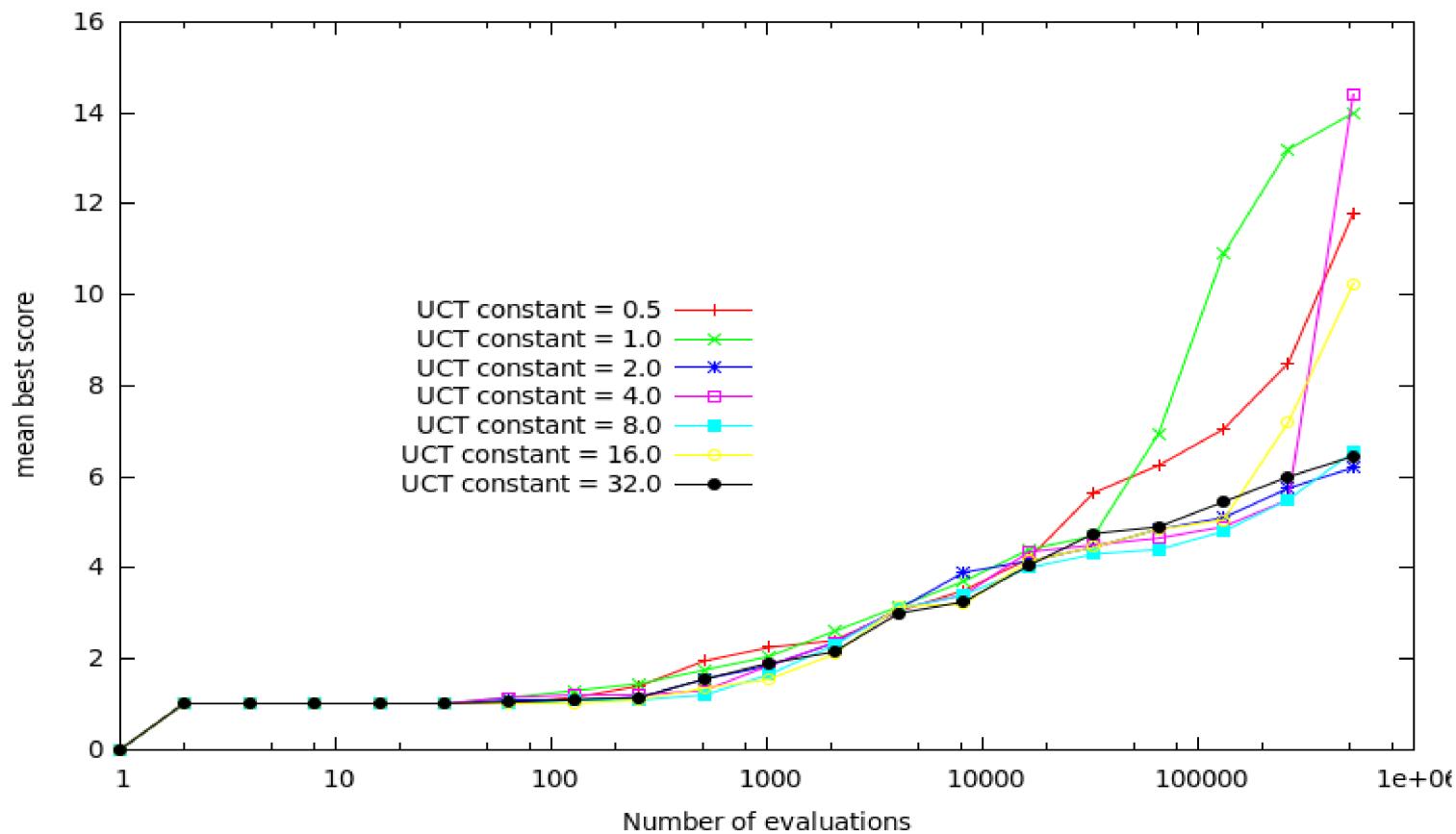
Monte-Carlo Discovery of Expressions



Monte-Carlo Discovery of Expressions



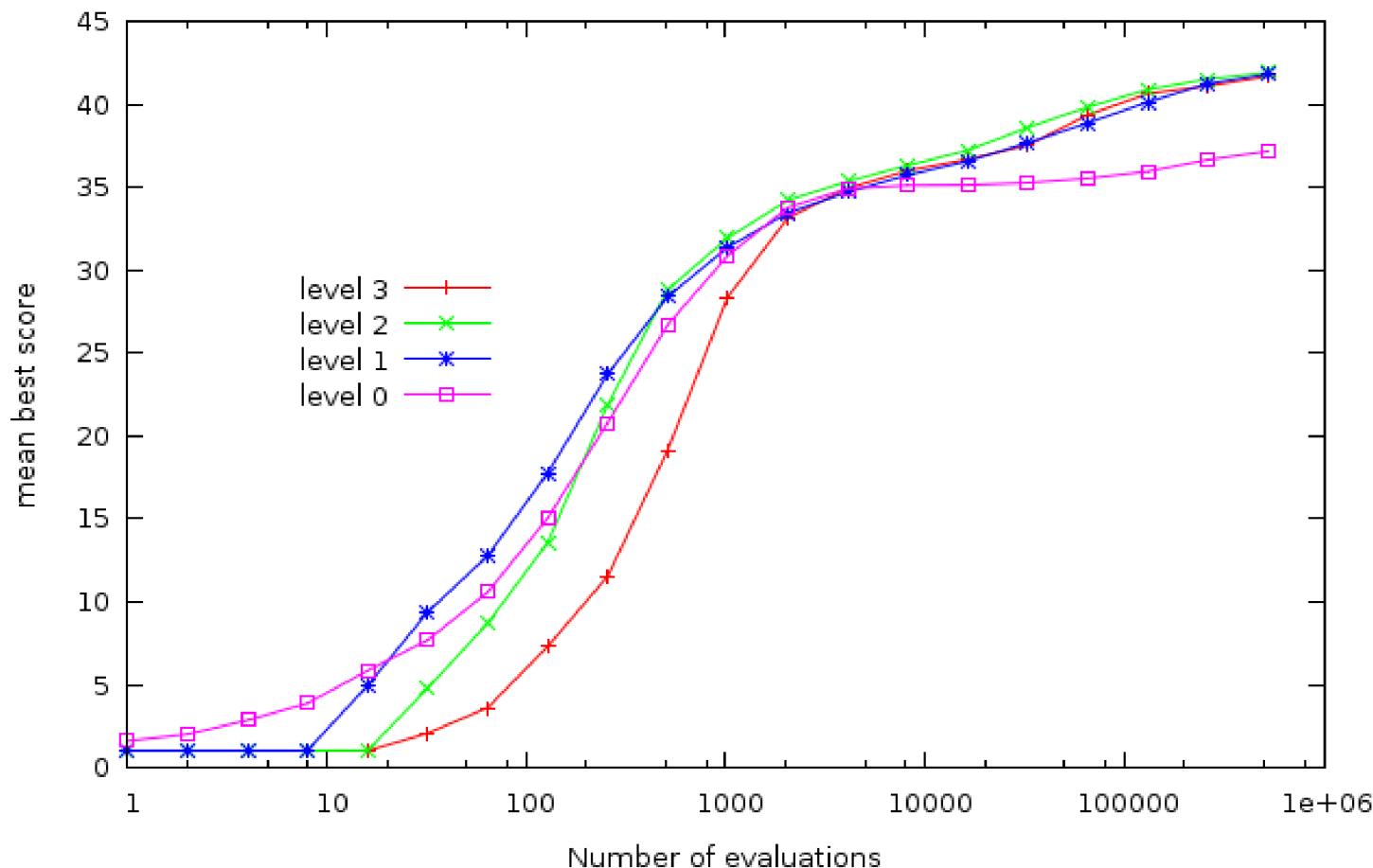
Monte-Carlo Discovery of Expressions



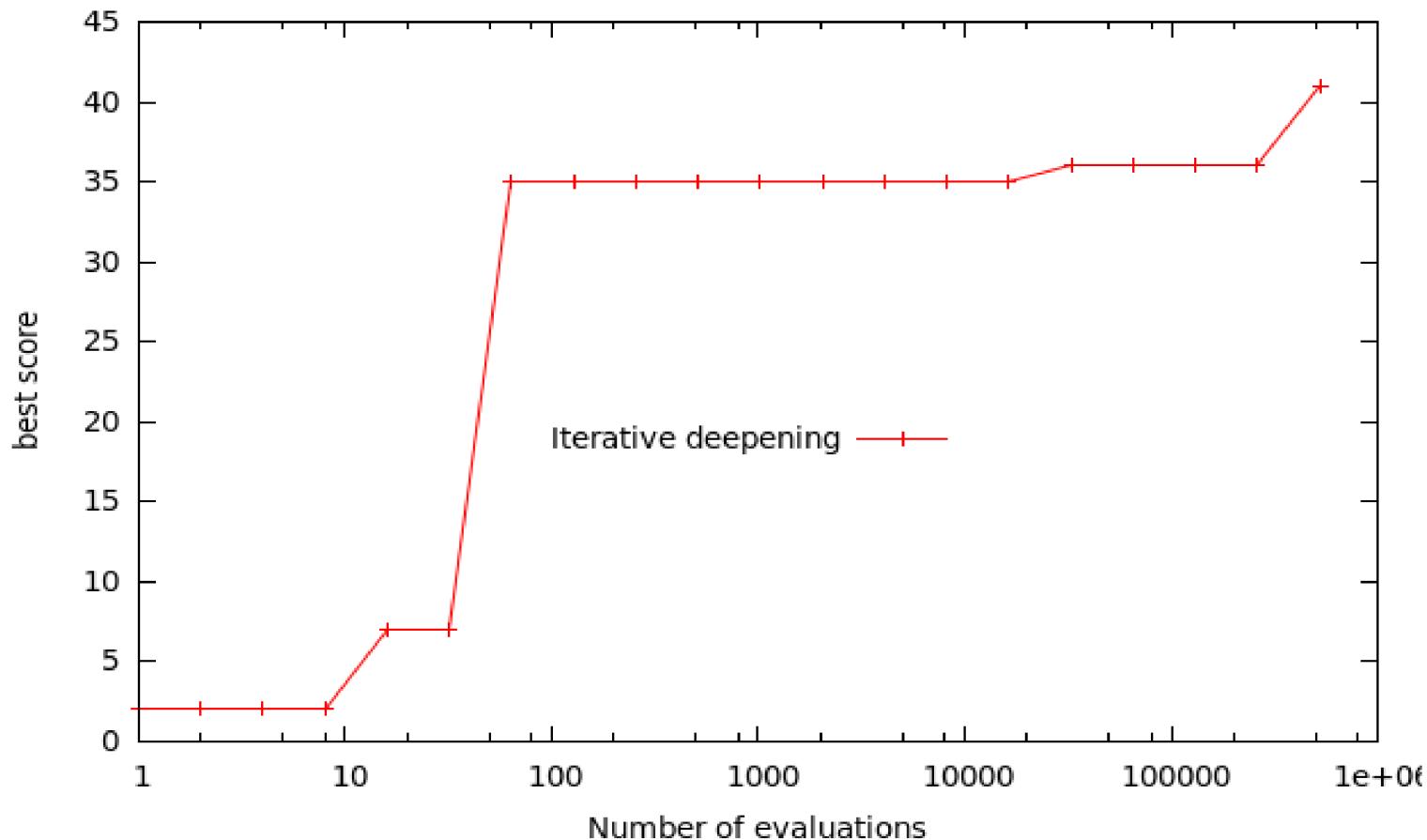
Monte-Carlo Discovery of Expressions

- N prisoners are assigned with either a 0 or a 1.
- A prisoner can see the number assigned to the other prisoners but cannot see his own number.
- Each prisoner is asked independently to guess if he is 0 or 1 or to pass.
- The prisoners can formulate a strategy before beginning the game.
- All the prisoners are free if at least one guesses correctly and none guess incorrectly.
- A possible strategy is for example that one of the prisoners says 1 and the others pass, this strategy has fifty percent chances of winning.

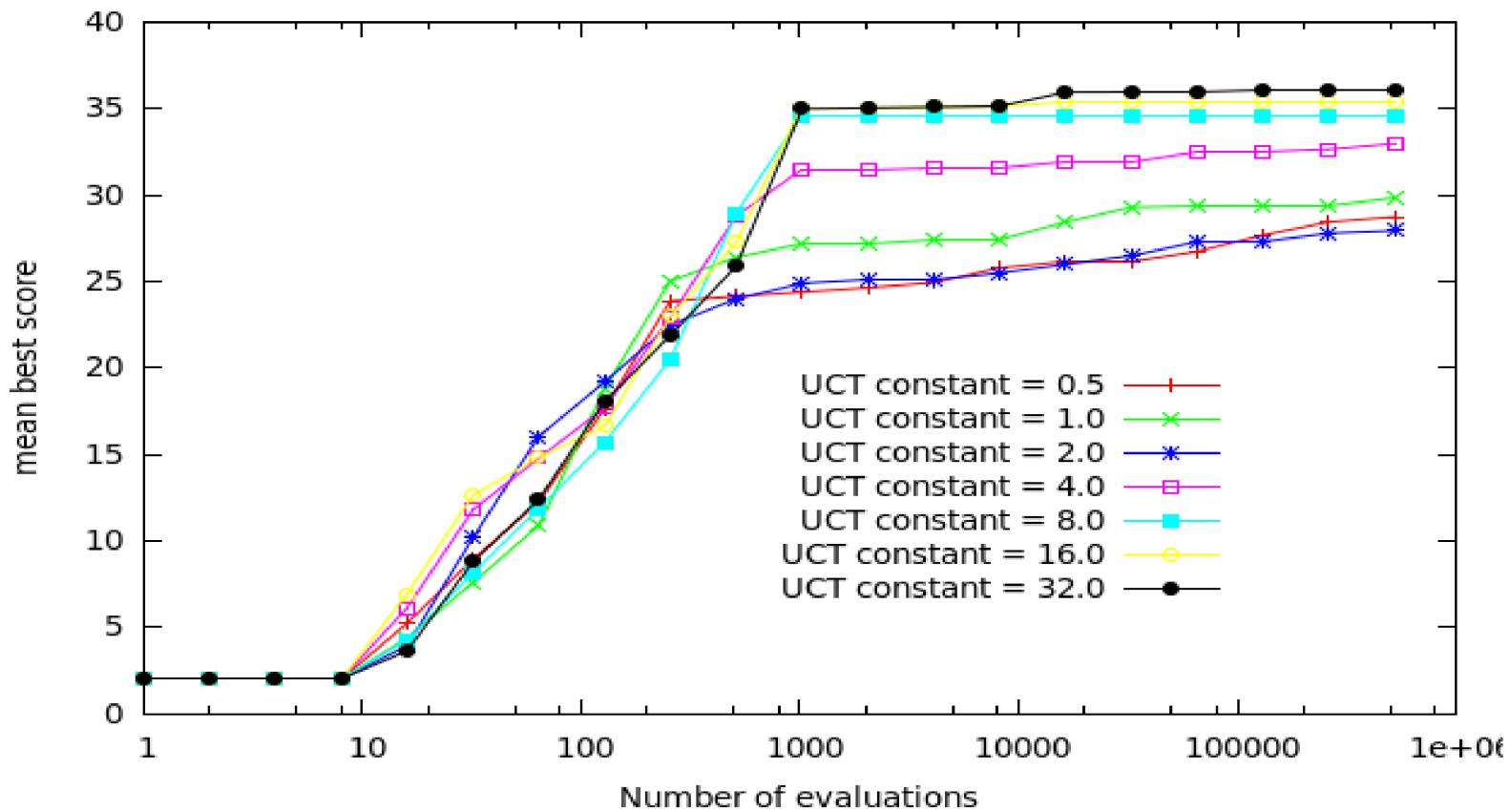
Monte-Carlo Discovery of Expressions



Monte-Carlo Discovery of Expressions



Monte-Carlo Discovery of Expressions



Application to financial data

- Data used to perform our empirical analysis are daily prices of European S&P500 index call options.
- The sample period is from January 02, 2003 to August 29, 2003.
- S&P500 index options are among the most actively traded financial derivatives in the world.

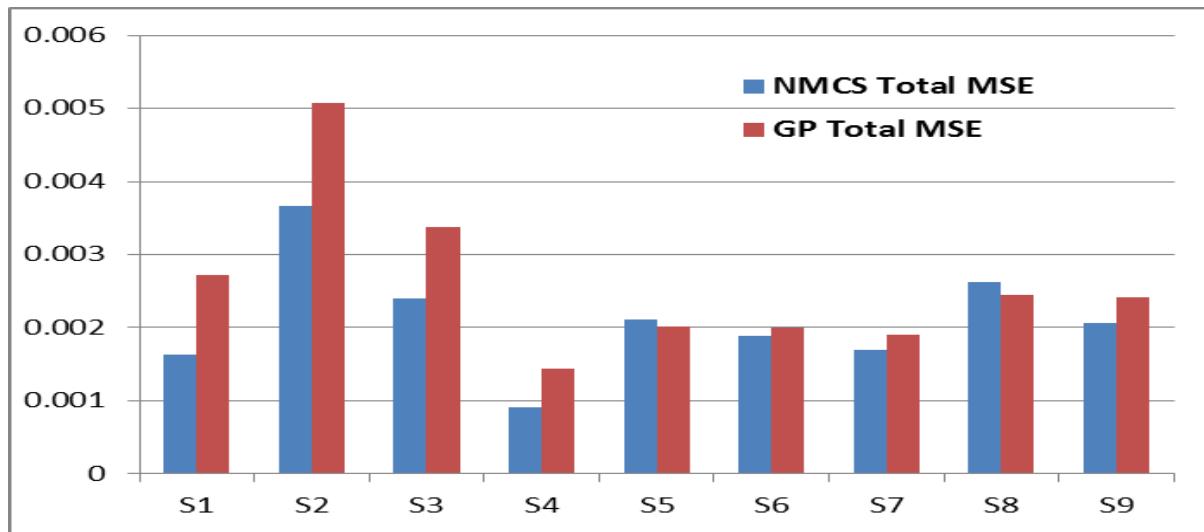
Atom Set

+	Addition	C/K Call Price/Strike Price
-	Subtraction	S/K Index Price/Strike Price
*	Multiplication	tau Time to Maturity
%	Protected Division	
In	Protected Natural Log	
Exp	Exponential function	
Sqrt	Protected Square Root	
cos	Cosinus	
sin	Sinus	
Ncdf	Normal cumulative distribution	

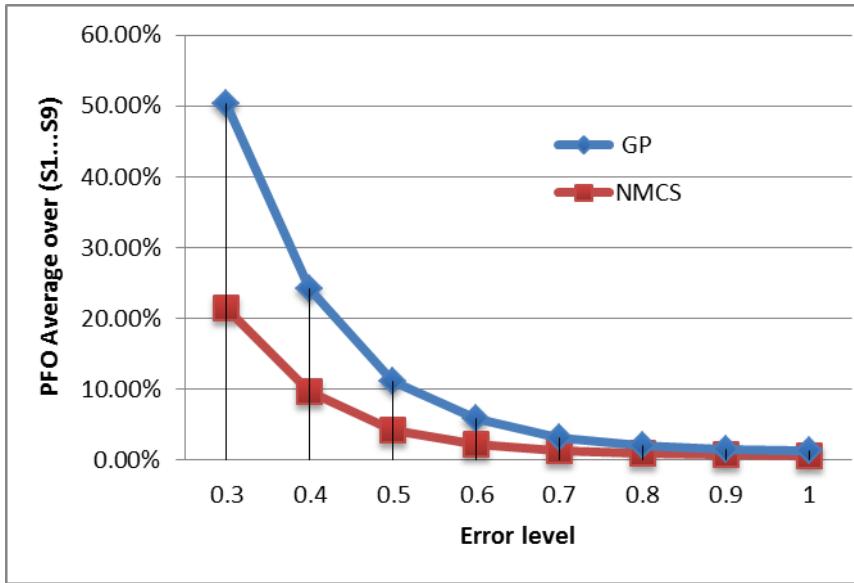
Fitness function

- Each formula found by NMCS or GP is evaluated to test whether it can accurately forecast the implied volatility for all entries in the training set.
- Fitness = Mean Squared Error (MSE) between the estimated volatility and the target volatility.

Mean Square Error



Poor Fitted Observations



RNA Inverse Folding

- Molecule Design as a Search Problem
- Find the sequence of nucleotides that gives a predefined structure.
- A biochemist applied Nested Monte Carlo Search to this problem [Portela 2018].
- Better than the state of the art.

Eterna100 Benchmark

- Molecule Design as Puzzles:



Exercise

- Write a Nested Monte Carlo Search for the left move problem.
- Functions to write :
 - legalMoves (state)
 - play (state, move)
 - terminal (state)
 - score (state)
 - playout (state)
- Then write a Nested Monte Carlo Search using these functions.

Expression Discovery

Exercise :

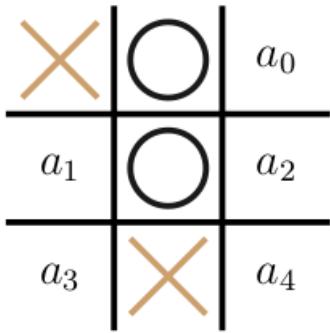
- Possible atoms : 1, 2, 3, +, -
- Goal : find expressions containing less than 10 atoms that have great evaluations.
- Generate random expressions (i.e. list of atoms).
- Evaluate an expression given as a list of atoms.
- Use NMCS to generate expressions

Nested Monte-Carlo Search for Two-player Games

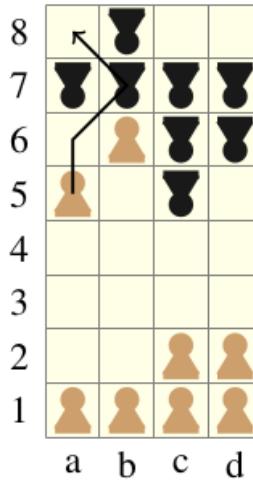
- The quality of information propagated during the search can be increased via a discounting heuristic, leading to a better move selection for the overall algorithm.
- Improving the cost-effectiveness of the algorithm without changing the resulting policy by using safe pruning criteria.
- Long-term convergence to an optimal strategy can be guaranteed by wrapping NMCS inside a UCT-like algorithm.

Nested Monte-Carlo Search for Two-player Games

- The discounting heuristic turns a win/loss game into a game with a wide range of outcomes by having the max player preferring short wins to long wins, and long losses to short losses.
- A playout returns $v(s_t) / (t + 1)$ with $v(s_t)$ in $\{-1, 1\}$



(a) The X player is to play. Any move except a_3 leads to a draw with perfect play.



(b) White's winning move is a_5-a_6 . b_6-a_7 and b_6-c_7 initially seem good but are blunders.

Figure 1: Partially played games of Tic Tac Toe and Breakthrough with a single winning move for the turn player.

Table 1: Effect of discounting on the distribution of nested level 2 policies applied to Figure 1a, across 1000 games.

Move	$\Pi(V(\text{NMC}(2)))$		$\Pi(V_D(\text{NMC}_D(2)))$	
	Value	Frequency	Value	Frequency
a_0	{0}	0	{0}	0
a_1	{0, 1}	176	{0}	0
a_2	{0, 1}	123	{0, $\frac{1}{4}$ }	0
a_3	{1}	575	{ $\frac{1}{2}$ }	1000
a_4	{0, 1}	126	{0, $\frac{1}{4}$ }	0

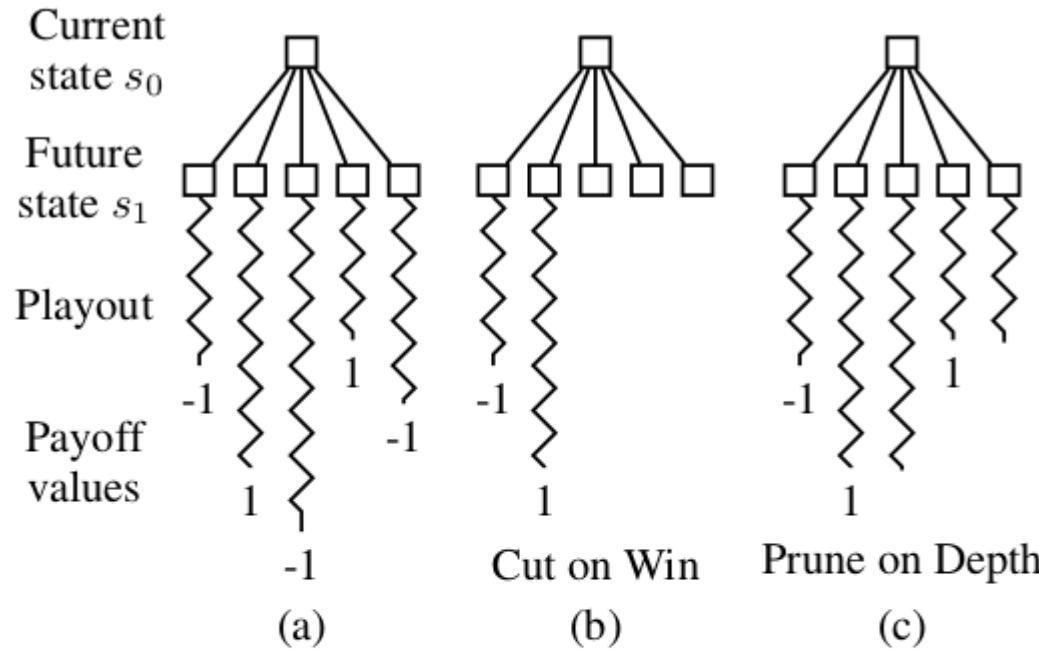


Figure 2: Effect of the pruning strategies on an NMCS run. We assume a max root state and a left-to-right evaluation order. (a) In the standard case, the second and the fourth successor states are equally preferred. With discounting, the fourth successor state is the most-preferred one. (b) This fourth state may fail to be selected when Cut on Win is enabled. (c) With Pruning on Depth and discounting, however, this fourth state would be found and preferred too.

Nested Monte-Carlo Search for Two-player Games

Table 2: Performance of $\text{NMC}(3)$ and $\text{NMC}_D(3)$ starting from Figure 1b, averaged over 900 runs; showing how discounting and pruning affect the number of states visited and the correct move frequency.

Discounting	Pruning	States Visited(k)	Freq(%)
No	None	$4,459 \pm 27$	11.9 ± 2.2
No	COW(≤ 1)	$1,084 \pm 8$	12.3 ± 2.6
No	COW(≤ 2)	214 ± 2	10.9 ± 2.0
No	COW(≤ 3)	25 ± 1	9.8 ± 2.0
Yes	None	$2,775 \pm 26$	64.1 ± 3.4
Yes	POD(≤ 1)	$1,924 \pm 20$	64.7 ± 3.5
Yes	POD(≤ 2)	$1,463 \pm 16$	58.6 ± 3.5
Yes	POD(≤ 3)	627 ± 19	62.4 ± 3.3

Nested Monte-Carlo Search for Two-player Games

Table 3: Winrates (%) of NMCS with discounting vs. NMCS without it for nesting levels 0 to 2 and game engine speed.

Game	Nesting Level			States visited per second (k)
	0	1	2	
Breakthrough	79.6	99.6	99.4	411
misère	42.4	80.8	90.0	409
Knightthrough	78.6	100.0	100.0	264
misère	46.0	83.2	85.8	328
Domineering	71.2	77.0	83.8	550
misère	43.4	63.2	68.4	592
NoGo	62.8	76.4	83.4	357
misère	53.2	65.6	67.2	648
AtariGo	69.6	97.2	100.0	280

Table 4: Win percentages of NMCS against a standard MCTS player for various settings and thinking times.

	Game	n	COW POD	10ms	20ms	40ms	80ms	160ms	320ms
Breakthrough	misère	1	3.2	6.0	12.0	11.6	7.8	6.4	
	misère	1	✓	27.6	22.6	16.8	21.6	15.4	20.4
	misère	1	✓	22.6	25.2	30.4	34.6	35.2	39.6
	misère	2	✓	4.6	2.0	2.4	1.4	2.4	3.8
	misère	1	85.4	83.4	70.2	60.8	57.0	56.4	
	misère	1	✓	91.4	95.6	97.0	97.8	98.8	98.8
	misère	1	✓	95.2	95.2	98.0	99.0	99.8	99.8
	misère	2	✓	1.0	27.6	43.6	87.0	93.2	95.6
Knightthrough	misère	1	42.2	57.2	9.8	49.4	50.2	50.0	
	misère	1	✓	68.6	50.2	42.4	42.4	46.4	44.6
	misère	1	✓	27.2	25.4	28.0	43.4	49.2	49.6
	misère	2	✓	20.0	16.4	5.8	1.8	29.2	38.2
	misère	1	43.0	31.6	20.0	15.4	11.2	12.6	
	misère	1	✓	54.6	72.2	80.6	88.4	94.2	98.4
	misère	1	✓	77.8	82.2	88.8	94.4	98.2	98.6
	misère	2	✓	20.8	18.6	32.2	42.2	54.0	67.0
Domineering	misère	1	13.4	8.6	8.6	6.0	14.2	28.0	
	misère	1	✓	40.8	34.4	37.4	48.4	50.0	50.0
	misère	1	✓	44.4	38.6	40.6	49.4	50.0	50.0
	misère	2	✓	11.2	14.4	20.2	25.2	32.2	45.4
	misère	1	33.4	25.2	20.0	18.8	13.2	12.2	
	misère	1	✓	45.4	47.2	56.8	60.2	62.8	54.2
	misère	1	✓	69.4	66.6	71.6	70.4	68.4	58.6
	misère	2	✓	37.0	45.2	45.6	51.0	57.8	53.6
NoGo	misère	1	5.8	3.0	2.6	3.0	0.6	0.8	
	misère	1	✓	7.2	16.0	31.8	35.2	35.4	40.6
	misère	1	✓	37.6	39.2	38.4	40.8	47.8	48.0
	misère	2	✓	0.4	2.8	5.4	15.0	20.6	17.0
	misère	1	14.6	6.6	5.2	3.0	2.4	1.8	
	misère	1	✓	17.2	25.0	38.8	51.2	48.2	48.8
	misère	1	✓	55.4	56.6	57.0	57.6	54.6	60.8
	misère	2	✓	5.2	10.6	19.4	35.6	37.2	47.8
Atari-Go	1	0.6	2.2	4.6	5.4	6.8	7.6		
	1	✓	0.2	19.2	42.0	42.0	55.4	67.2	
	1	✓	42.0	59.0	60.2	71.0	71.2	77.2	
	2	✓	0.2	0.0	0.6	7.4	8.6	4.8	

Algorithm 1: Two-player two-outcome NMCS.

```
1 nested (nesting n, state s, depth d, bound λ)
2   while  $s \notin T$  do
3      $s^* \leftarrow \text{rand}(\delta(s))$ 
4     if  $\tau(s) = \max$  then  $l^* \leftarrow \frac{-1}{d}$  else  $l^* \leftarrow \frac{1}{d}$ 
5     if d-pruning and  $\tau(s)\{-l^*, \lambda\} = \lambda$  then return  $\lambda$ 
6     if  $n > 0$  then
7       foreach  $s'$  in  $\delta(s)$  do
8          $l \leftarrow \text{nested}(n - 1, s', d + 1, l^*)$ 
9         if  $\tau(s)\{l, l^*\} \neq l^*$  then  $s^* \leftarrow s'; l^* \leftarrow l$ 
10        if cut on win and  $\tau(s)\{l, 0\} \neq 0$  then break
11       $s \leftarrow s^*$ 
12       $d \leftarrow d + 1$ 
13    if discounting then return  $\frac{v(s)}{d}$ 
14    else return  $v(s)$ 
```

Exercise

- Modify Breakthrough to play Misere Breakthrough.
- Modify playouts for discounted rewards.
- Nested playouts.
- UCT with nested discounted playouts.
- Compare to standard UCT.

Nested Rollout Policy Adaptation

Nested Rollout Policy Adaptation

- NRPA is NMCS with policy learning.
- It uses Gibbs Sampling as a playout policy.
- It adapts the weights of the moves according to the best sequence of moves found so far.
- During adaptation each weight of a move of the best sequence is incremented and the other moves in the same state are decreased proportionally to the exponential of their weights.

Nested Rollout Policy Adaptation

- Each move is associated to a weight w_i
- During a playout each move is played with a probability:

$$\exp(w_i) / \sum_k \exp(w_k)$$

Nested Rollout Policy Adaptation

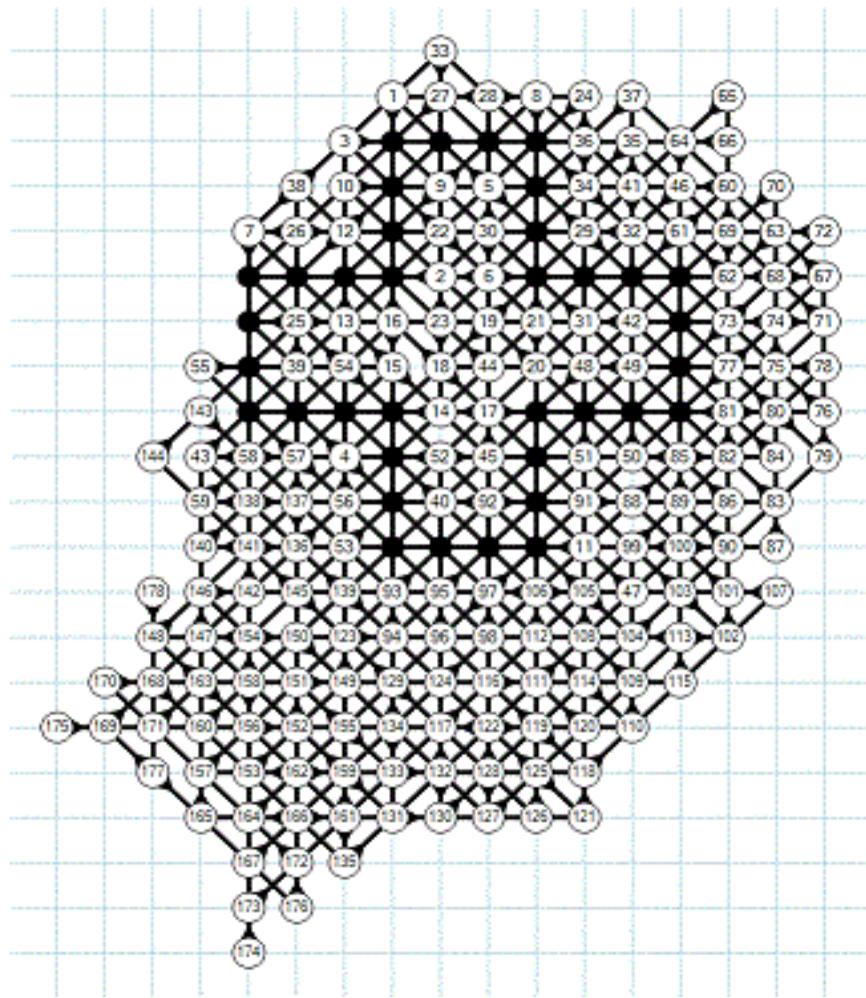
- For each move of the best sequence:

$$w_i = w_i + 1$$

- For each possible move of each state of the best sequence:

$$w_i = w_i - \exp(w_i) / \sum_k \exp(w_k)$$

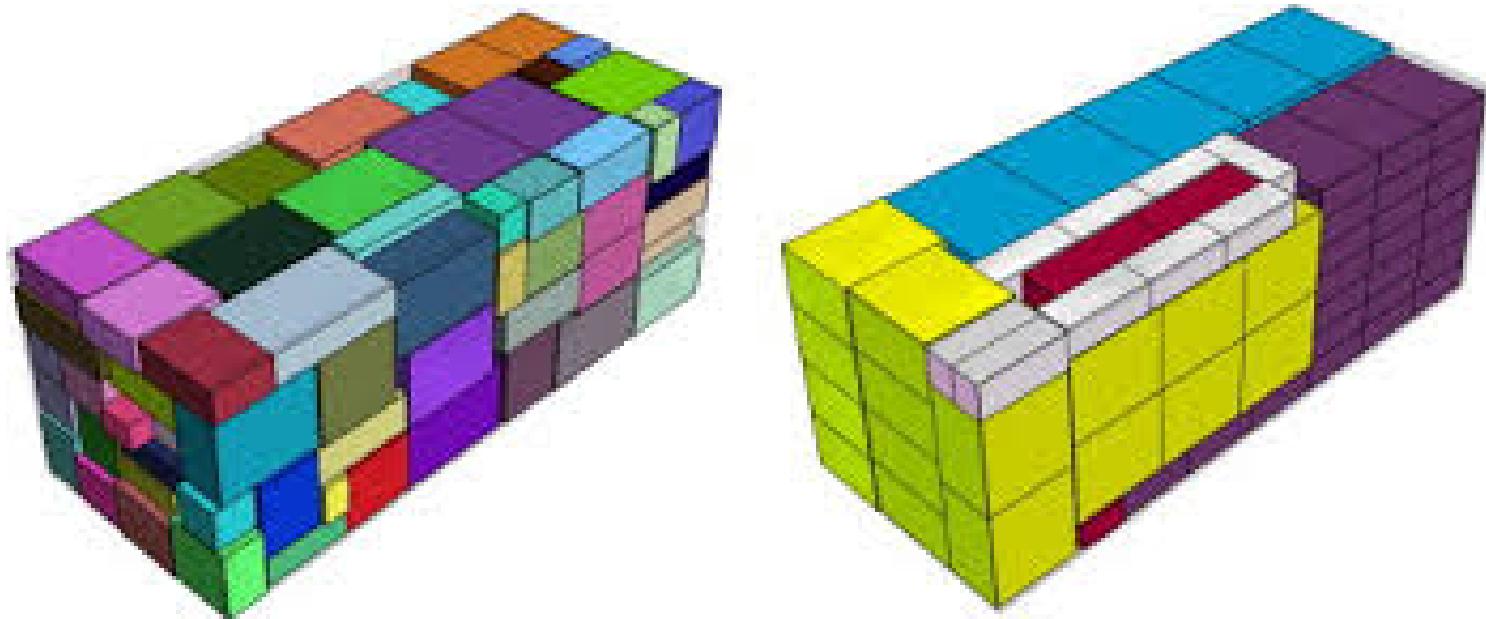
Morpion Solitaire



World record [Rosin 2011]

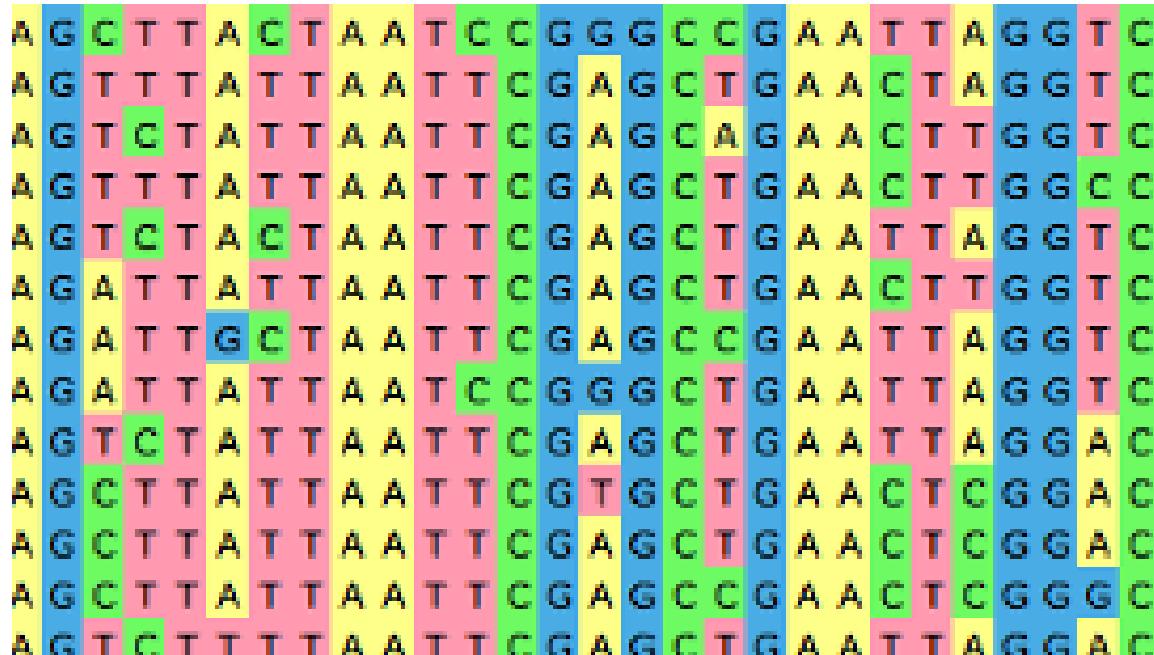
Applications of NRPA

- 3D packing with object orientation.



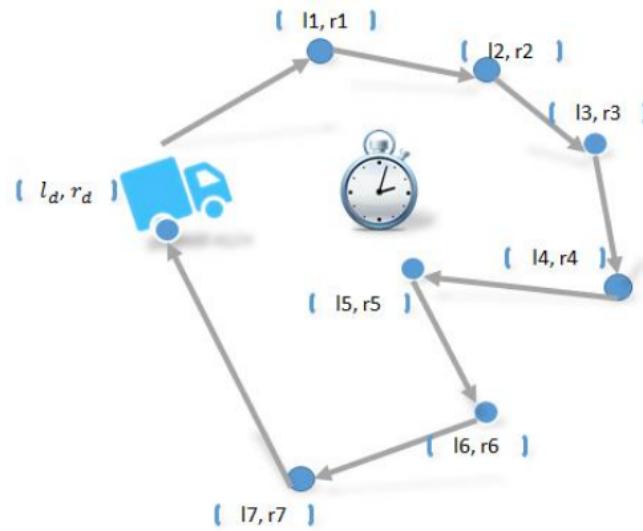
Applications of NRPA

- Improvement of some alignments for Multiple Sequence Alignment [Edelkamp & al 2015].



Applications of NRPA

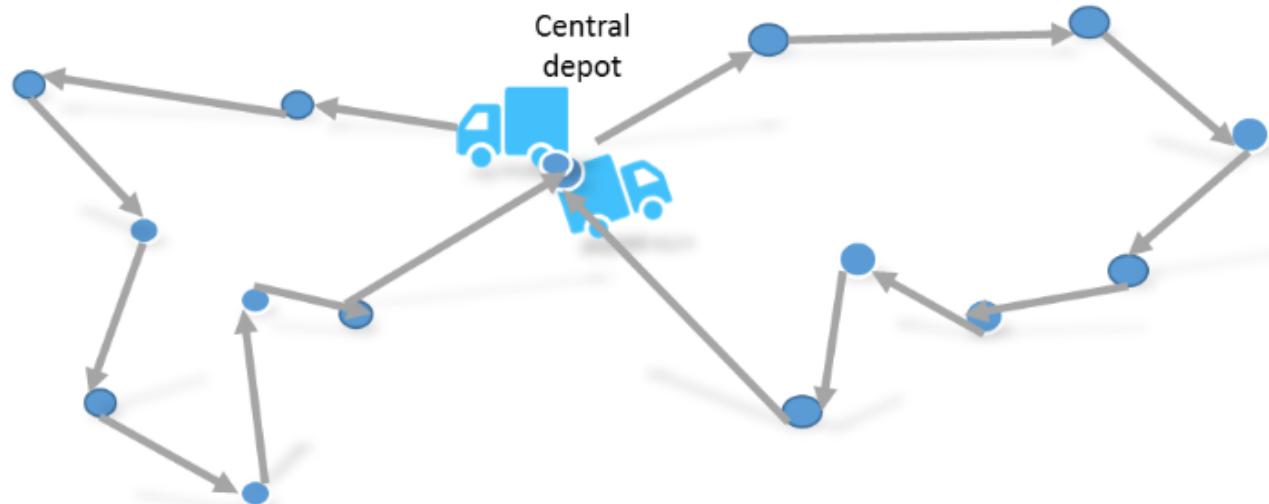
- Traveling Salesman Problem with Time Windows [Cazenave 2012].



- Physical traveling salesman problem.

Applications of NRPA

- State of the art results for Logistics [Edelkamp & al. 2016].



EDF Agents

- EDF fleet of vehicles is one of the largest.
- They plan interventions every day.
- Monte Carlo Search is 5% better than the specialized algorithms they use.
- Millions of kilometers saved each year.
- Hundreds of tons of CO₂ each year.

Nested Rollout Policy Adaptation

- Morpion Solitaire [Rosin 2011]
- CrossWords [Rosin 2011]
- Travelling Salesman Problem with Time Windows [Cazenave et al. 2012]
- 3D Packing with Object Orientation [Edelkamp et al. 2014]
- Multiple Sequence Alignment [Edelkamp et al. 2015]
- SameGame [Cazenave et al. 2016]
- Vehicle Routing Problems [Edelkamp et al. 2016, Cazenave et al. 2020]
- Graph Coloring [Cazenave et al. 2020]
- RNA Inverse Folding [Cazenave & Fournier 2020]
- ...

Selective Policies

- Prune bad moves during playouts.
- Modify the legal moves function.
- Use rules to find bad moves.
- Different domain specific rules for :
 - Bus regulation,
 - SameGame,
 - Weak Schur numbers.

Bus Regulation

- At each stop a regulator can decide to make a bus wait before continuing his route.
- Waiting at a stop can reduce the overall passengers waiting time.
- The score of a simulation is the sum of all the passengers waiting time.
- Optimizing a problem is finding a set of bus stopping times that minimizes the score of the simulation.

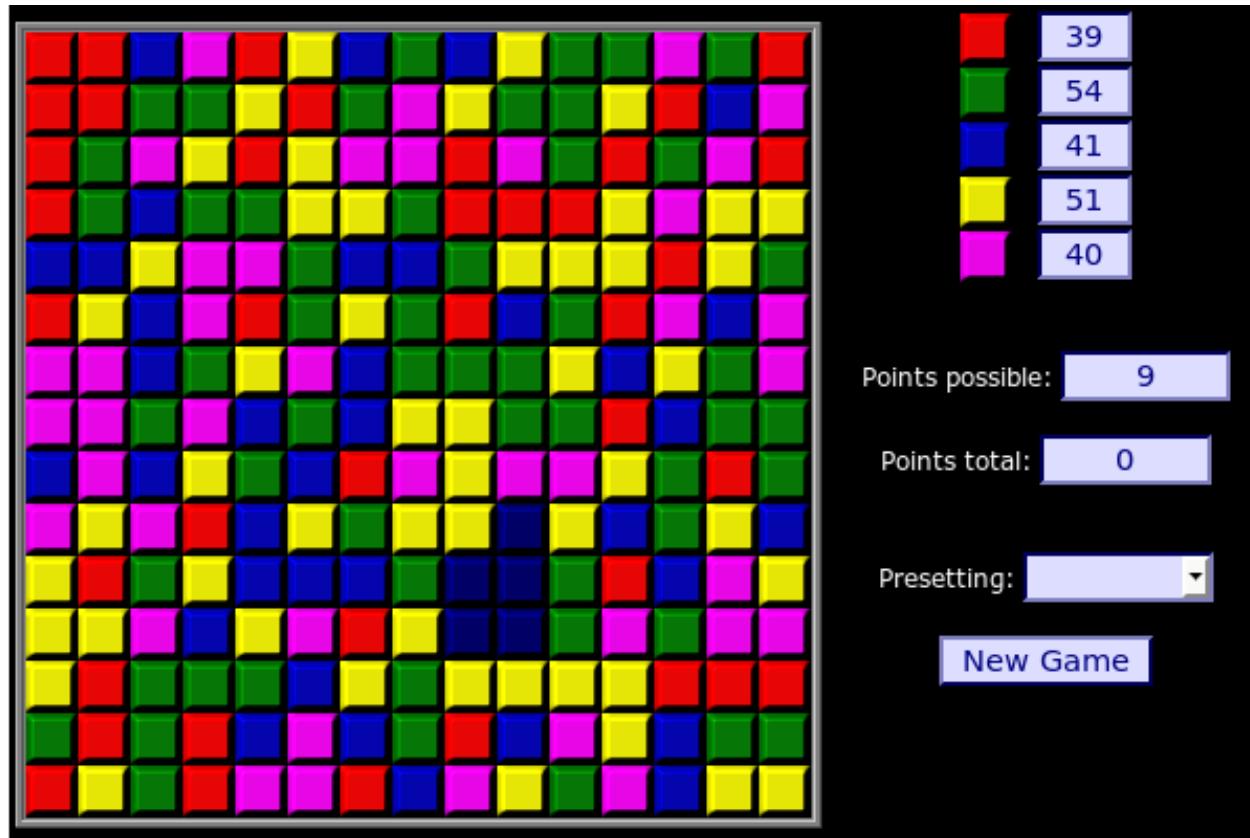
Bus Regulation

- Standard policy: between 1 and 5 minutes
- Selective policy : waiting time of 1 if there are fewer than δ stops before the next bus.
- Code for a move:
 - the bus stop,
 - the time of arrival to the bus stop,
 - the number of minutes to wait before leaving the stop.

Bus Regulation

Time	No δ	$\delta = 3$
0.01	2,620	2,147
0.02	2,441	2,049
0.04	2,329	2,000
0.08	2,242	1,959
0.16	2,157	1,925
0.32	2,107	1,903
0.64	2,046	1,868
1.28	1,974	1,811
2.56	1,892	1,754
5.12	1,802	1,703
10.24	1,737	1,660
20.48	1,698	1,640
40.96	1,682	1,629
81.92	1,660	1,617
163.84	1,632	1,610

SameGame



SameGame

- Code of a move = Zobrist hashing.
- Tabu color strategy = avoid moves of the dominant color until there is only one block of the dominant color.
- Selective policy = allow moves of size two of the tabu color when the number of moves already played is greater than t .

SameGame

Time	No tabu	tabu	t > 10
0.01	155.83	352.19	257.59
0.02	251.28	707.56	505.05
0.04	340.18	927.63	677.57
0.08	404.27	1,080.64	822.44
0.16	466.15	1,252.14	939.30
0.32	545.78	1,375.78	1,058.54
0.64	647.63	1,524.37	1,203.91
1.28	807.20	1,648.16	1,356.81
2.56	1,012.42	1,746.74	1,497.90
5.12	1,184.77	1,819.43	1,605.86
10.24	1,286.25	1,886.48	1,712.17
20.48	1,425.55	1,983.42	1,879.10
40.96	1,579.67	2,115.80	2,100.47
81.92	1,781.40	2,319.44	2,384.24
163.84	2,011.25	2,484.18	2,636.22

SameGame

Standard test set of 20 boards:

NMCS	SP-MCTS	NRPA	web
77,934	78,012	80,030	87,858

Same Game

- Hybrid Parallelization [Negrevergne 2017].
- Root Parallelization for each computer. Leaf Parallelization of the playouts using threads.
- New record of 83 050.
- Parallelization for Morpion Solitaire [Nagorko 2019].

Weak Schur Numbers

- Find a partition of consecutive numbers that contains as many consecutive numbers as possible
- A partition must not contain a number that is the sum of two previous numbers in the same partition.
- Partition of size 3 :

1 2 4 8 11 22

3 5 6 7 19 21 23

9 10 12 13 14 15 16 17 18 20

Weak Schur Numbers

- Often a good move to put the next number in the same partition as the previous number.
- If it is legal to put the next number in the same partition as the previous number then it is the only legal move considered.
- Otherwise all legal moves are considered.
- The code of a move for the Weak Schur problem takes as input the partition of the move, the integer to assign and the previous number in the partition.

Weak Schur Numbers

Time	ws(9)	ws-rule(9)
0.01	199	2,847
0.02	246	3,342
0.04	263	3,717
0.08	273	4,125
0.16	286	4,465
0.32	293	4,757
0.64	303	5,044
1.28	314	5,357
2.56	331	5,679
5.12	362	6,065
10.24	384	6,458
20.48	403	6,805
40.96	422	7,117
81.92	444	7,311
163.84	473	7,538

Selective Policies

- We have applied selective policies to three quite different problems.
- For each problem selective policies improve NRPA.
- We used only simple policy improvements.
- Better performance could be obtained refining the proposed policies.

Algorithm 1 The playout algorithm

```
playout (state, policy)
sequence  $\leftarrow []$ 
while true do
    if state is terminal then
        return (score (state), sequence)
    end if
    z  $\leftarrow 0.0$ 
    for m in possible moves for state do
        z  $\leftarrow z + \exp(\text{policy}[\text{code}(m)])$ 
    end for
    choose a move with probability  $\frac{\exp(\text{policy}[\text{code}(\text{move})])}{z}$ 
    state  $\leftarrow \text{play}(\text{state}, \text{move})$ 
    sequence  $\leftarrow \text{sequence} + \text{move}$ 
end while
```

Algorithm 2 The Adapt algorithm

```
Adapt (policy, sequence)
  polp  $\leftarrow$  policy
  state  $\leftarrow$  root
  for move in sequence do
    polp [code(move)]  $\leftarrow$  polp [code(move)] +  $\alpha$ 
    z  $\leftarrow$  0.0
    for m in possible moves for state do
      z  $\leftarrow$  z + exp (policy [code(m)])
    end for
    for m in possible moves for state do
      polp [code(m)]  $\leftarrow$  polp [code(m)] -  $\alpha * \frac{\exp(\text{policy}[\text{code}(m)])}{z}$ 
    end for
    state  $\leftarrow$  play (state, move)
  end for
  policy  $\leftarrow$  polp
```

Algorithm 3 The NRPA algorithm.

```
NRPA (level, policy)
if level == 0 then
    return playout (root, policy)
end if
bestScore  $\leftarrow -\infty$ 
for N iterations do
    (result,new)  $\leftarrow$  NRPA(level - 1, policy)
    if result  $\geq$  bestScore then
        bestScore  $\leftarrow$  result
        seq  $\leftarrow$  new
    end if
    policy  $\leftarrow$  Adapt (policy, seq)
end for
return (bestScore, seq)
```

Exercise

- Apply NRPA to the Left Move problem.
- Write a function playout (state) that plays a playout using Gibbs sampling.
- The probability of playing a move is proportional to the exponential of the weight of the move.
- weight is a dictionary that contains the weights of the moves.
- Write the Adapt function
- Write the NRPA function

Exercise

- Apply NRPA to the Weak Schur problem.
- Write a function that plays a playout using Gibbs sampling.
- The probability of playing a move is proportional to the exponential of the weight of the move.
- weight is an array that contains the weights of the moves.
- code (move) returns the index associated to the move in the weight array.

Exercise

- Write the adapt function that modifies the weights of the moves according to the best sequence of moves.
- Weights of the moves of the best sequence are incremented.
- For each state of the best sequence, weights of all the moves are reduced proportional to their probabilities.

Exercise

- Write the multi level NRPA code that retains a best sequence per level and recursively calls lower levels.
- Level zero is a playout with Gibbs sampling.

Analysis of Nested Rollout Policy Adaptation

The playouts use Gibbs sampling. Each move m_{ik} is associated to a weight w_{ik} .
The probability p_{ik} of choosing the move m_{ik} in a playout is the softmax function:

$$p_{ik} = \frac{e^{w_{ik}}}{\sum_j e^{w_{ij}}}$$

The cross-entropy loss for learning to play move m_{ib} is $C_i = -\log(p_{ib})$. In order to apply the gradient we calculate the partial derivative of the loss: $\frac{\delta C_i}{\delta p_{ib}} = -\frac{1}{p_{ib}}$. We then calculate the partial derivative of the softmax with respect to the weights:

$$\frac{\delta p_{ib}}{\delta w_{ij}} = p_{ib}(\delta_{bj} - p_{ij})$$

Where $\delta_{bj} = 1$ if $b = j$ and 0 otherwise. Thus the gradient is:

$$\nabla w_{ij} = \frac{\delta C_i}{\delta p_{ib}} \frac{\delta p_{ib}}{\delta w_{ij}} = -\frac{1}{p_{ib}} p_{ib}(\delta_{bj} - p_{ij}) = p_{ij} - \delta_{bj}$$

If we use α as a learning rate we update the weights with:

$$w_{ij} = w_{ij} - \alpha(p_{ij} - \delta_{bj})$$

This is the formula used in the NRPA algorithm to adapt weights.

Generalized Nested Rollout Policy Adaptation

We propose to generalize the NRPA algorithm by generalizing the way the probability is calculated using a temperature τ and a bias β_{ij} :

$$p_{ik} = \frac{e^{\frac{w_{ik}}{\tau} + \beta_{ik}}}{\sum_j e^{\frac{w_{ij}}{\tau} + \beta_{ij}}}$$

Generalized Nested Rollout Policy Adaptation

The formula for the derivative of $f(x) = \frac{g(x)}{h(x)}$ is:

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h^2(x)}$$

So the derivative of p_{ib} relative to w_{ib} is:

$$\frac{\delta p_{ib}}{\delta w_{ib}} = \frac{\frac{1}{\tau} e^{\frac{w_{ib}}{\tau} + \beta_{ib}} \sum_j e^{\frac{w_{ij}}{\tau} + \beta_{ij}} - \frac{1}{\tau} e^{\frac{w_{ib}}{\tau} + \beta_{ib}} e^{\frac{w_{ib}}{\tau} + \beta_{ib}}}{(\sum_j e^{\frac{w_{ij}}{\tau} + \beta_{ij}})^2}$$

$$\frac{\delta p_{ib}}{\delta w_{ib}} = \frac{1}{\tau} \frac{e^{\frac{w_{ib}}{\tau} + \beta_{ib}}}{\sum_j e^{\frac{w_{ij}}{\tau} + \beta_{ij}}} \frac{\sum_j e^{\frac{w_{ij}}{\tau} + \beta_{ij}} - e^{\frac{w_{ib}}{\tau} + \beta_{ib}}}{\sum_j e^{\frac{w_{ij}}{\tau} + \beta_{ij}}}$$

$$\frac{\delta p_{ib}}{\delta w_{ib}} = \frac{1}{\tau} p_{ib}(1 - p_{ib})$$

Generalized Nested Rollout Policy Adaptation

The derivative of p_{ib} relative to w_{ij} with $j \neq b$ is:

$$\frac{\delta p_{ib}}{\delta w_{ij}} = -\frac{1}{\tau} \frac{e^{\frac{w_{ij}}{\tau} + \beta_{ij}} e^{\frac{w_{ib}}{\tau} + \beta_{ib}}}{(\sum_j e^{\frac{w_{ij}}{\tau} + \beta_{ij}})^2}$$
$$\frac{\delta p_{ib}}{\delta w_{ij}} = -\frac{1}{\tau} p_{ij} p_{ib}$$

We then derive the cross-entropy loss and the softmax to calculate the gradient:

$$\nabla w_{ij} = \frac{\delta C_i}{\delta p_{ib}} \frac{\delta p_{ib}}{\delta w_{ij}} = -\frac{1}{\tau} \frac{1}{p_{ib}} p_{ib} (\delta_{bj} - p_{ij}) = \frac{p_{ij} - \delta_{bj}}{\tau}$$

If we use α as a learning rate we update the weights with:

$$w_{ij} = w_{ij} - \alpha \frac{p_{ij} - \delta_{bj}}{\tau}$$

Generalized Nested Rollout Policy Adaptation

Let the weights and probabilities of playing moves be indexed by the iteration of the GNRPA level. Let w_{nij} be the weight w_{ij} at iteration n , p_{nij} be the probability of playing move j at step i at iteration n , δ_{nbj} the δ_{bj} at iteration n .

$$p_{0ij} = \frac{e^{\frac{1}{\tau}w_{0ij} + \beta_{ij}}}{\sum_k e^{\frac{1}{\tau}w_{0ik} + \beta_{ik}}}$$

$$w_{1ij} = w_{0ij} - \frac{\alpha}{\tau}(p_{0ij} - \delta_{0bj})$$

$$p_{1ij} = \frac{e^{\frac{1}{\tau}w_{1ij} + \beta_{ij}}}{\sum_k e^{\frac{1}{\tau}w_{1ik} + \beta_{ik}}} = \frac{e^{\frac{1}{\tau}w_{0ij} - \frac{\alpha}{\tau^2}(p_{0ij} - \delta_{0bj}) + \beta_{ij}}}{\sum_k e^{\frac{1}{\tau}w_{1ik} + \beta_{ik}}}$$

$$w_{2ij} = w_{1ij} - \frac{\alpha}{\tau}(p_{1ij} - \delta_{1bj}) = w_{0ij} - \frac{\alpha}{\tau}(p_{0ij} - \delta_{0bj} + p_{1ij} - \delta_{1bj})$$

By recurrence we get:

$$p_{nij} = \frac{e^{\frac{1}{\tau}w_{nij} + \beta_{ij}}}{\sum_k e^{\frac{1}{\tau}w_{nik} + \beta_{ik}}} = \frac{e^{\frac{w_{0ij}}{\tau} - \frac{\alpha}{\tau^2}(\sum_k p_{kij} - \delta_{kbj}) + \beta_{ij}}}{\sum_k e^{\frac{1}{\tau}w_{nik} + \beta_{ik}}}$$

Generalized Nested Rollout Policy Adaptation

From this equation we can deduce the equivalence between different algorithms. For example GNRPA₁ with $\alpha_1 = (\frac{\tau_1}{\tau_2})^2 \alpha_2$ and τ_1 is equivalent to GNRPA₂ with α_2 and τ_2 provided we set w_{0ij} in GNRPA₁ to $\frac{\tau_1}{\tau_2} w_{0ij}$. It means we can always use $\tau = 1$ provided we correspondingly set α and w_{0ij} .

Another deduction we can make is we can set $\beta_{ij} = 0$ provided we set $w_{0ij} = w_{0ij} + \tau \times \beta_{ij}$. We can also set $w_{0ij} = 0$ and use only β_{ij} which is easier.

The equivalences mean that GNRPA is equivalent to NRPA with the appropriate α and w_{0ij} . However it can be more convenient to use β_{ij} than to initialize the weights w_{0ij} as we will see for SameGame.

SameGame

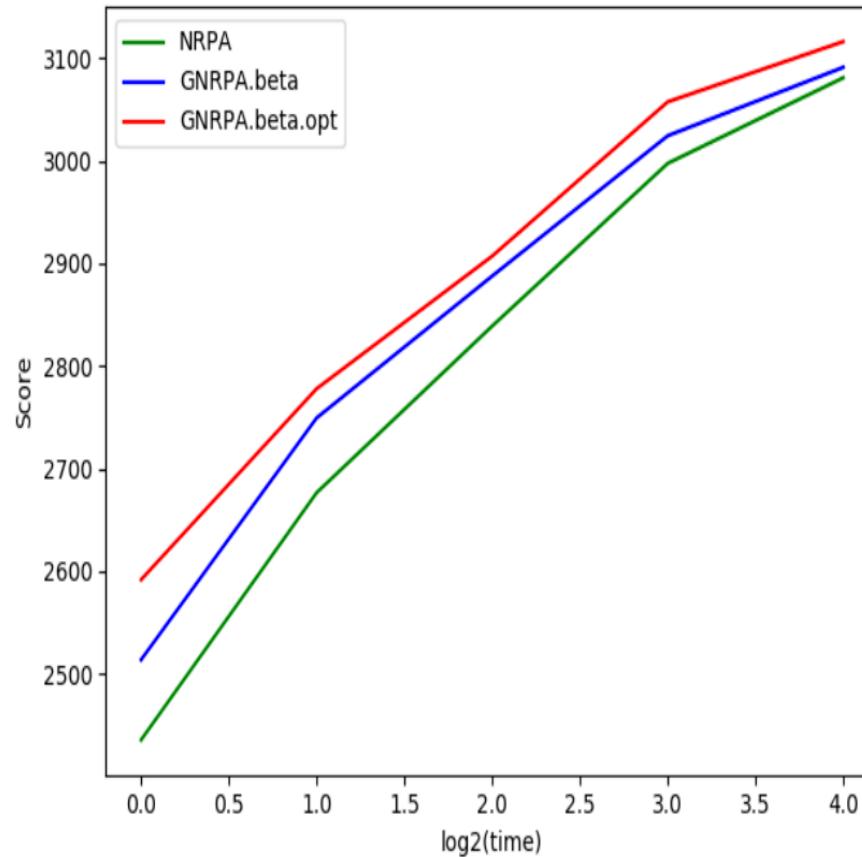


Fig. 1: Evolution of the average scores of the three algorithms at SameGame.

TSPTW

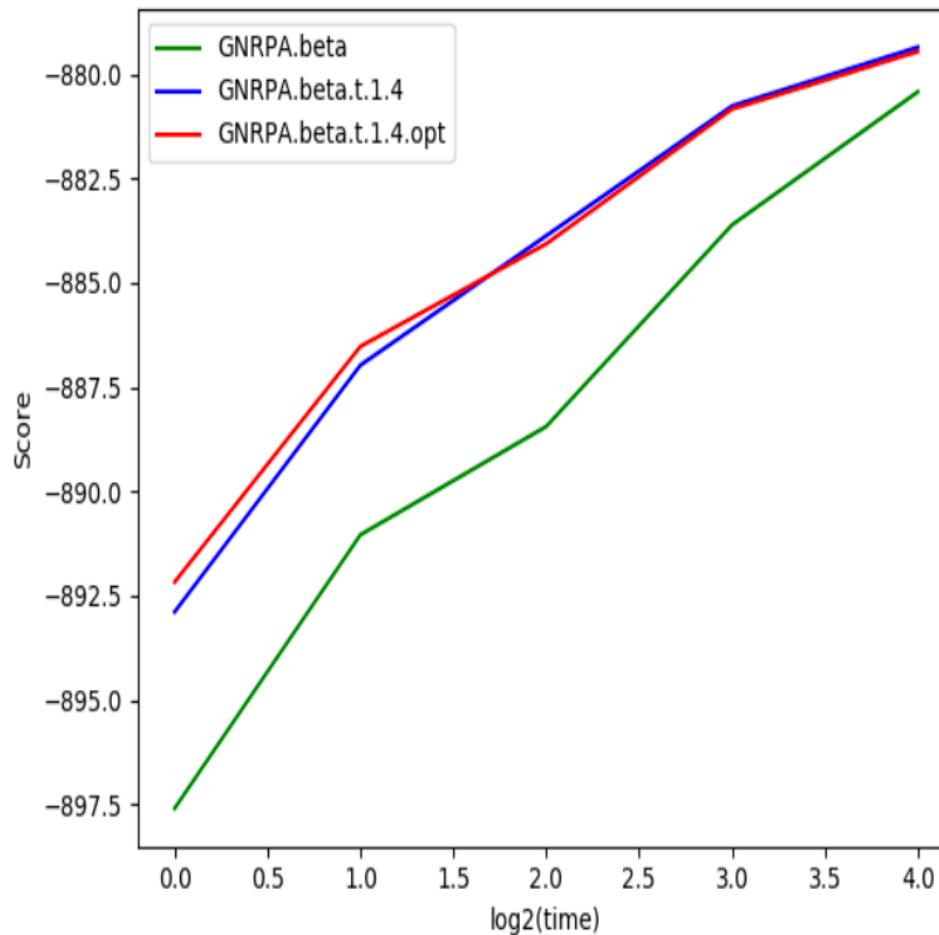


Fig. 2: Evolution of the average scores of the three algorithms for TSPTW.

TSPTW

Table 2: Results for the TSPTW rc204.1 problem

Time	NRPA	GNRPA.beta	GNRPA.beta.t.1.4	GNRPA.beta.t.1.4.opt
40.96	-3745986.46 (245766.53)	-897.60 (1.32)	-892.89 (0.96)	-892.17 (1.04)
81.92	-1750959.11 (243210.68)	-891.04 (1.05)	-886.97 (0.87)	-886.52 (0.83)
163.84	-1030946.86 (212092.35)	-888.44 (0.98)	-883.87 (0.71)	-884.07 (0.70)
327.68	-285933.63 (108975.99)	-883.61 (0.63)	-880.76 (0.40)	-880.83 (0.32)
655.36	-45918.97 (38203.97)	-880.42 (0.30)	-879.35 (0.16)	-879.45 (0.17)

GNRPA

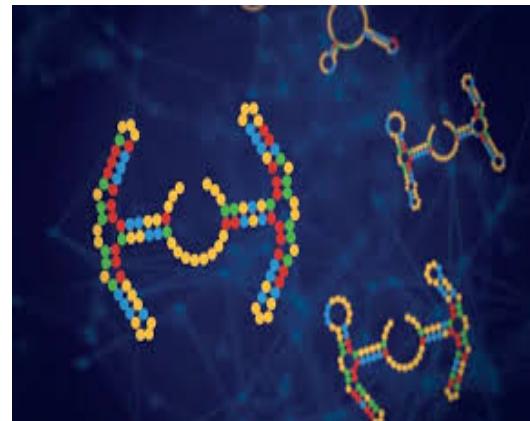
- NRPA with a bias.
- Equivalent to the initialization of the weights.
- More convenient to use a bias.
- We can always set the temperature to 1 without a loss of generality.
- Good results for SameGame and TSPTW.

GNRPA

- Exercise:
- Apply GNRPA to the Weak Schur problem.

Eterna 100

- Find a sequence that has a given folding



Eterna 100

- Human experts have managed to solve the 100 problems of the benchmark
- No program has so far achieved such a score.
- The best score so far is 95/100 by NEMO:
NEsted MOnte Carlo RNA Puzzle Solver

NEMO

- NEMO uses two sets of heuristics
- General ones that give probabilities to pairs of bases.
- More specific ones that are tailored to puzzle solving.

GNRPA

Let w_{ib} be the weight associated to move b at index i in the sequence. In NRPA the probability of choosing move b at index i is:

$$p_{ib} = \frac{e^{w_{ib}}}{\sum_k e^{w_{ik}}}$$

We propose to try Generalized NRPA (GNRPA) [9] for Inverse Folding and to replace it with:

$$p_{ib} = \frac{e^{w_{ib} + \beta_{ib}}}{\sum_k e^{w_{ik} + \beta_{ik}}}$$

where we use for β_{ib} the logarithm of the probabilities used in NEMO.

Other Improvements

- Stabilized GNRPA
- Beam GNRPA
- Zobrist Hashing
- Restarts
- Parallelization

Experimental Results

Level	α	N	β_{ib}	P	Beam	H	Solved
1	1.0	100	no	1	1.1.1	0	3
1	1.0	100	yes	1	1.1.1	0	30
1	1.0	100	yes	1	1.1.1	1	32
1	1.0	100	yes	1	4.1.1	1	42
1	1.0	100	yes	1	8.1.1	1	53
1	1.0	100	yes	1	16.1.1	1	54
1	1.0	100	yes	4	8.1.1	1	69
1	1.0	100	yes	4	8.1.1	2	69
2	1.0	100	no	1	1.1.1	0	49
2	1.0	100	yes	1	1.1.1	0	73
2	1.0	100	yes	1	1.1.1	1	75
2	1.0	100	yes	2	1.1.1	0	73
2	1.0	100	yes	3	1.1.1	0	74
2	1.0	100	yes	4	1.1.1	0	80
2	1.0	100	yes	5	1.1.1	0	77
2	1.0	100	yes	6	1.1.1	0	75
2	1.0	100	yes	7	1.1.1	0	80
2	1.0	100	yes	8	1.1.1	0	79
2	1.0	100	yes	9	1.1.1	0	81
2	1.0	100	yes	10	1.1.1	0	80
2	1.0	100	yes	4	8.1.1	1	85
3	1.0	100	yes	1	1.1.1	0	85

Experimental Results

- Leaf Parallelization

Table 2: Parallelization efficiency.

Algorithm	1	2	4	6	8	12
GNRPA(level=1,N=100,P=4,Beam=8)	11.916	6.889	4.526	3.657	3.169	3.359

Experimental Results

Table 3: Number of problems solved by GNRPA using different parameters and a fixed time limit.

β_{ib}	P	Beam	//	R	N	Start	H	1m	2m	4m	8m	16m	32m	64m
no	1.1	1.1	n	n	100.100	0.0	0	30	33	41	50	61	67	69
yes	1.1	1.1	n	n	100.100	0.0	0	58	64	68	72	74	75	79
yes	4.1	4.1	n	n	100.100	0.0	0	71	75	78	79	81	83	84
yes	4.1	8.1	n	n	100.100	0.0	0	75	75	79	80	81	83	84
yes	4.1	8.1	n	n	100.100	0.0	1	75	78	80	82	83	85	87
yes	4.1	8.1	n	n	100.100	4.4	1	76	80	81	82	84	85	87
yes	4.1	8.1	y	n	100.100	4.4	1	78	84	83	86	87	87	88
yes	4.1	8.1	y	3	$\infty.\infty$	4.4	1	80	84	85	85	88	89	92

Experimental Results

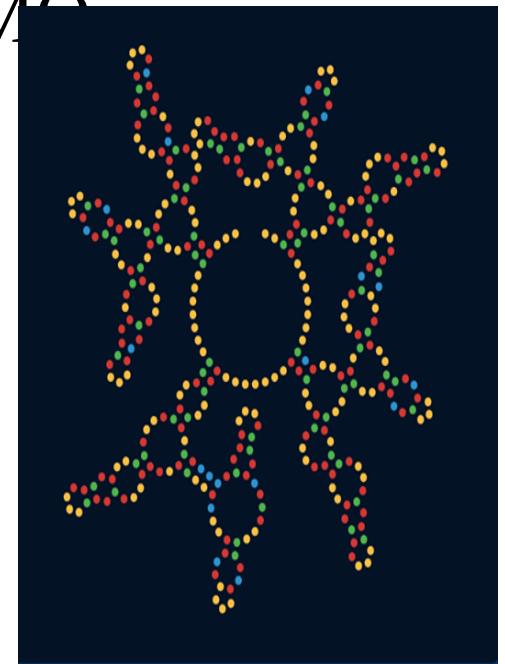
- Root Parallelization

Table 6: Number of problems solved with root parallel GNRPA.

Process	1m	2m	4m	8m	16m	32m	64m
20	82	84	85	86	87	88	89

Conclusion

- 95/100 problems solved, same as NEMO.
- Less domain knowledge.
- Various improvements of NRPA.



Playout Policy Adaptation

Offline learning of a playout policy

- Offline learning of playout policies has given good results in Go [Coulom 2007, Huang 2010] and Hex [Huang 2013], learning fixed pattern weights so as to bias the playouts.
- Patterns are also used to do progressive widening in the UCT tree.

Online learning of a playout policy

- The RAVE algorithm [Gelly 2011] performs online learning of moves values in order to bias the choice of moves in the UCT tree.
- RAVE has been very successful in Go and Hex.
- A development of RAVE is to use the RAVE values to choose moves in the playouts using Pool RAVE [Rimmel 2010].
- Pool RAVE improves slightly on random playouts in Havannah and reaches 62.7% against random playouts in Go.

Online learning of a playout policy

- Move-Average Sampling Technique (MAST) is a technique used in the GGP program Cadia Player so as to bias the layouts with statistics on moves [Finnsson 2010].
- It consists of choosing a move in the playout proportionally to the exponential of its mean.
- MAST keeps the average result of each action over all simulations.

Online learning of a playout policy

- Later improvements of Cadia Player are N-Grams and the last good reply policy [Tak 2012].
- They have been applied to GGP so as to improve playouts by learning move sequences.
- A recent development in GGP is to have multiple playout strategies and to choose the one which is the most adapted to the problem at hand [Swiechowski 2014].

Online learning of a playout policy

- Playout Policy Adaptation (PPA) also uses Gibbs sampling.
- The evaluation of an action for PPA is not its mean over all simulations such as in MAST.
- Instead the value of an action is learned comparing it to the other available actions for the state where it has been played.

Playout Policy learning

- Start with a uniform policy.
- Use the policy for the playouts.
- Adapt the policy for the winner of each playout.

Playout Policy learning

- Each move is associated to a weight w_i .
- During a playout each move is played with a probability :

$$\exp(w_i) / \sum_k \exp(w_k)$$

Playout Policy learning

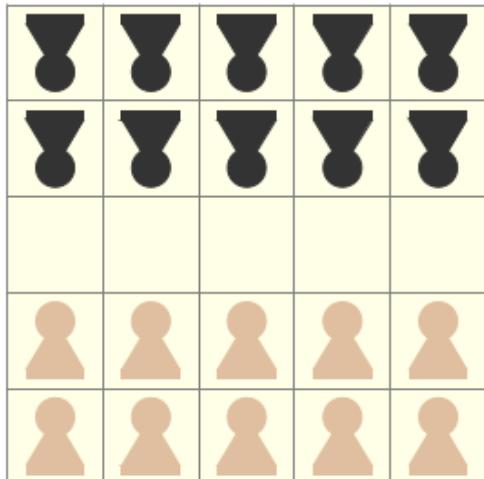
- Online learning :
- For each move of the winner :

$$w_i = w_i + 1$$

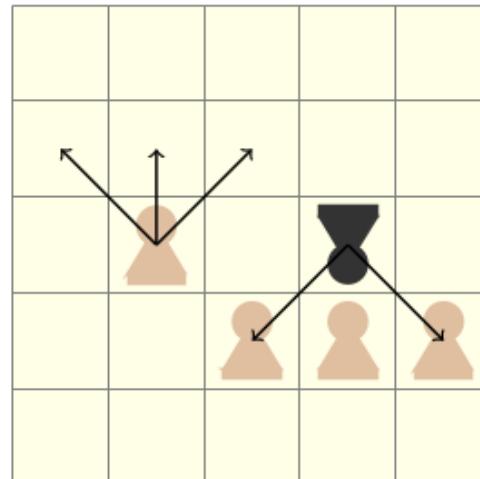
- For each possible move of each state of the winner :

$$w_i = w_i - \exp(w_i) / \sum_k \exp(w_k)$$

Breakthrough



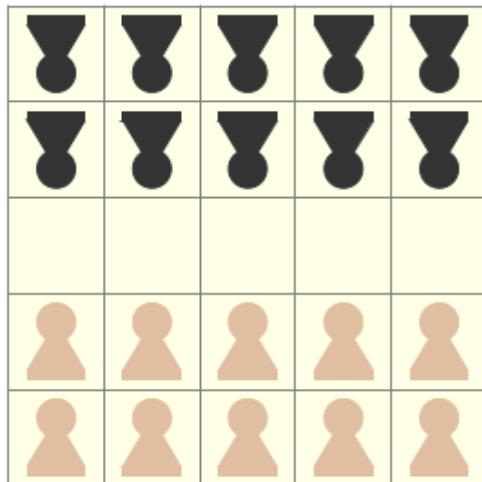
(a) Starting position on size 5×5 .



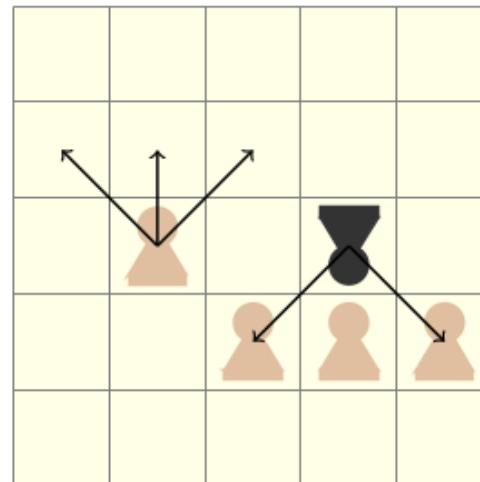
(b) Possible movements.

- The first player to reach the opposite line has won

Misère Breakthrough



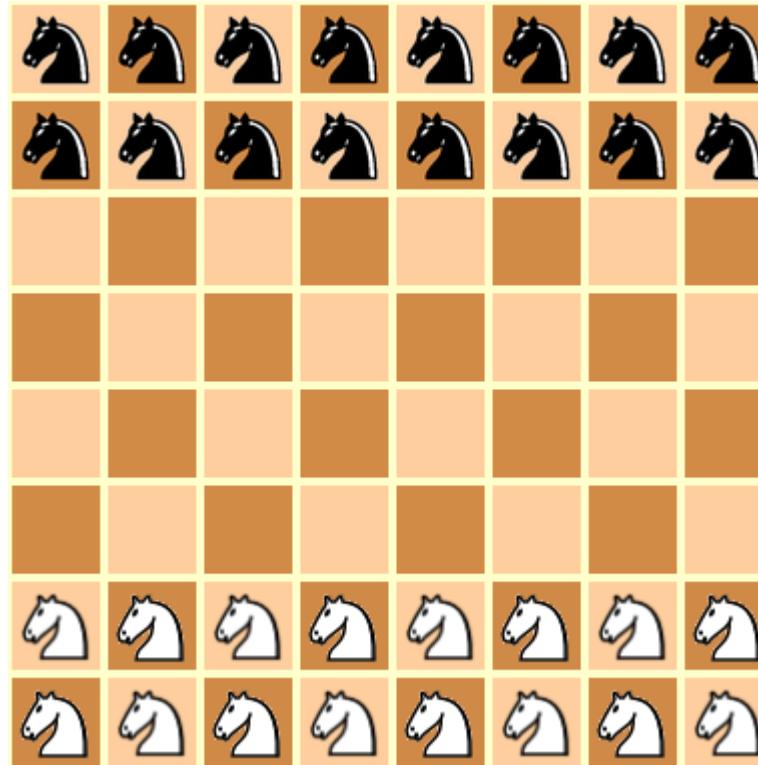
(a) Starting position on size 5×5 .



(b) Possible movements.

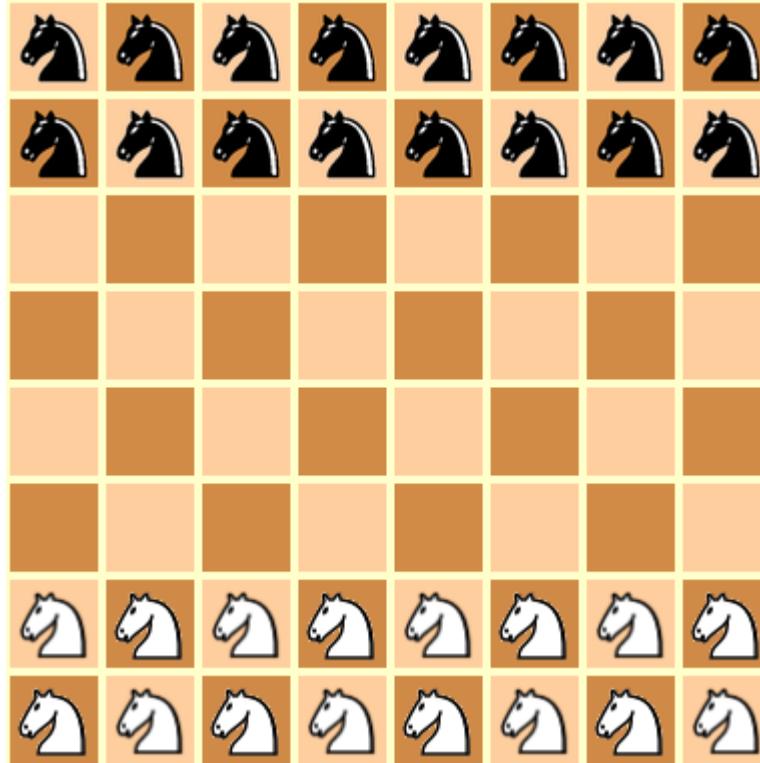
- The first player to reach the opposite line has lost

Knightthrough



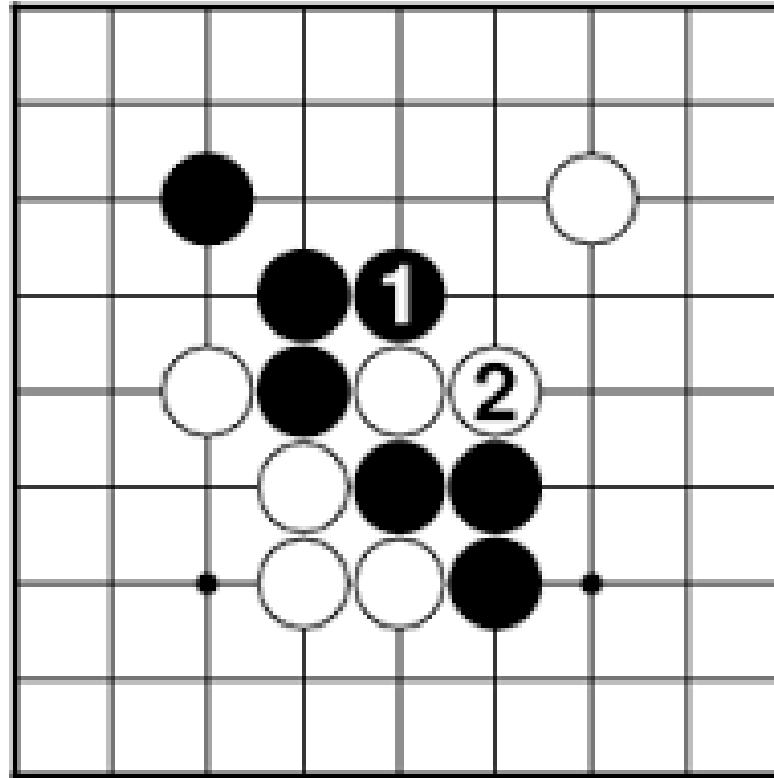
- The first to put a knight on the opposite side has won.

Misère Knightthrough



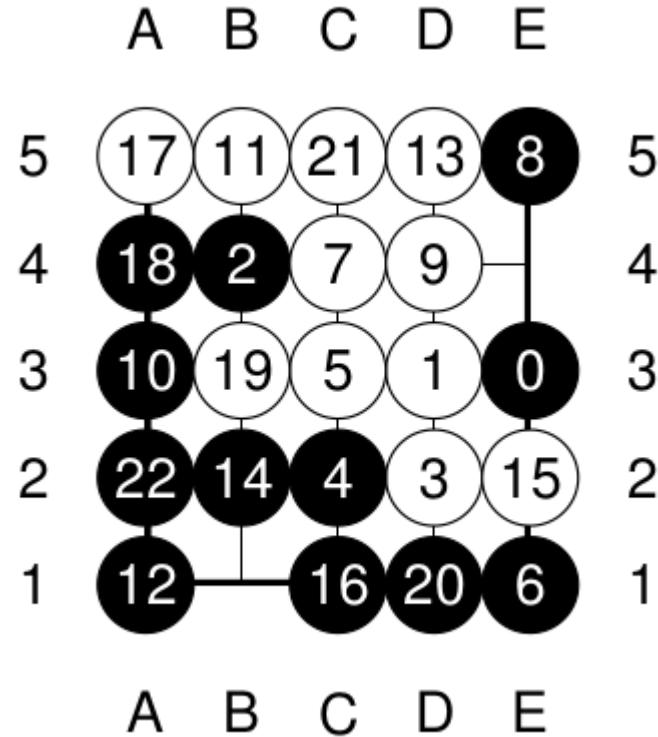
- The first to put a knight on the opposite side has lost.

Atarigo



- The first to capture has won

Nogo

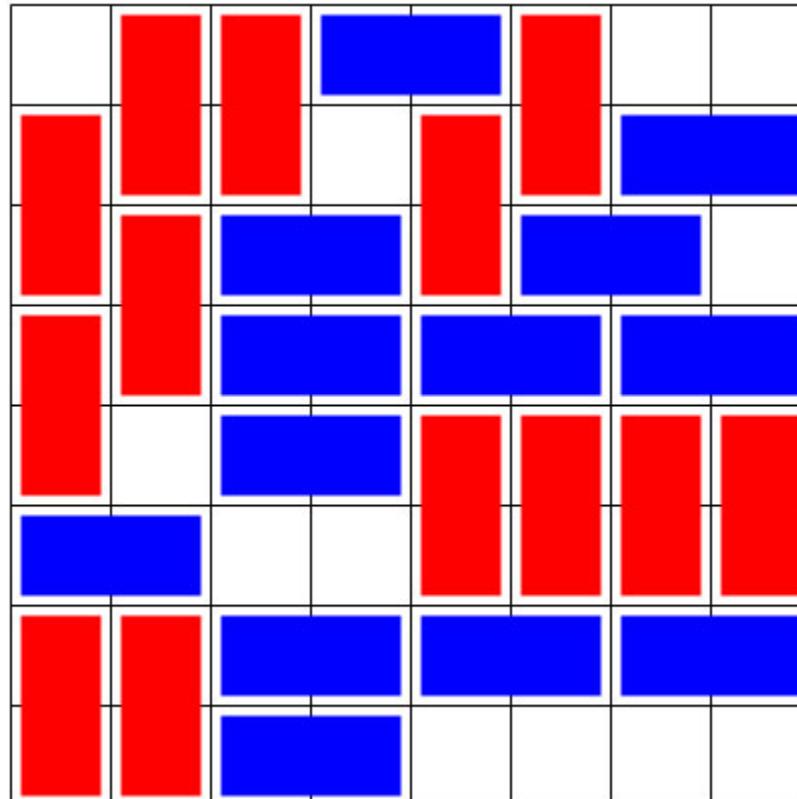


- The first to capture has lost

Domineering

Misère Domineering

- The last to play has won / lost.



Experimental results

	Size	Playouts	
		1,000	10,000
Atarigo	8 x 8	72.2	94.4
Breakthrough	8 x 8	55.2	54.4
Misere Breakthrough	8 x 8	99.2	97.8
Domineering	8 x 8	48.4	58.0
Misere Domineering	8 x 8	76.4	83.4
Go	8 x 8	23.0	1.2
Knightthrough	8 x 8	64.2	64.6
Misere Knightthrough	8 x 8	99.8	100.0
Nogo	8 x 8	64.8	46.4
Misere Nogo	8 x 8	80.6	89.4

Playout Policy learning with Move Features

- Associate features to the move.
- A move and its features are associated to a code.
- The algorithm learns the weights of codes instead of simply the weights of moves.

Playout Policy learning with Move Features

- Atarigo : four adjacent intersections
- Breakthrough : capture in the move code
- Misère Breakthrough : same as Breakthrough
- Domineering : cells next to the domino played
- Misère Domineering : same as Domineering
- Knightthrough : capture in the move code
- Misère Knightthrough : same as Knightthrough
- Nogo : same as Atarigo

Experimental results

- Each result is the outcome of a 500 games match, 250 with White and 250 with Black.
- UCT with an adaptive policy (PPAF) is played against UCT with a random policy.
- Tests are done for 10,000 playouts.
- For each game we test size 8x8.
- We tested 8 different games.

Experimental results

	Size	Winning %
Atarigo	8 x 8	94.4 %
Breakthrough	8 x 8	81.4 %
Misere Breakthrough	8 x 8	100.0 %
Domineering	8 x 8	80.4 %
Misere Domineering	8 x 8	93.0 %
Knightthrough	8 x 8	84.0 %
Misere Knightthrough	8 x 8	100.0 %
Nogo	8 x 8	95.4 %

PPAF and Memorization

- Start a game with an uniform policy.
- Adapt at each move of the game.
- Start at each move with the policy of the previous move.

PPAF and Memorization

- A nice property of PPAF is that the move played after the algorithm has been run is the most simulated move.
- The memorized policy is related to the state after the move played by the algorithm since it is the most simulated move.
- When starting with the memorized policy for the next state, this state has already been partially learned

PPAFM versus PPAF uniform

Game	Score
Atarigo	66.0%
Breakthrough	87.4%
Domineering	58.0%
Knightthrough	84.6%
Misere Breakthrough	97.2%
Misere Domineering	56.8%
Misere Knightthrough	99.2%
Nogo	49.4%

PPAFM versus UCT

Game	Score
Atarigo	95.4%
Breakthrough	94.2%
Domineering	81 .8%
Knightthrough	96.6%
Misere Breakthrough	100.0%
Misere Domineering	95.8%
Misere Knightthrough	100.0%
Nogo	91.6%

PPA Adapt Algorithm

Algorithm 4 The PPA adapt algorithm

```
adapt (winner, board, player, playout, policy)
 $polp \leftarrow policy$ 
for move in playout do
    if winner = player then
         $polp [code(move)] \leftarrow polp [code(move)] + \alpha$ 
         $z \leftarrow 0.0$ 
        for m in possible moves on board do
             $z \leftarrow z + \exp (policy [code(m)])$ 
        end for
        for m in possible moves on board do
             $polp [code(m)] \leftarrow polp [code(m)] - \alpha * \frac{\exp(policy[code(m)])}{z}$ 
        end for
    end if
    play (board, move)
    player  $\leftarrow$  opponent (player)
end for

policy  $\leftarrow polp$


```

Exercise

- Try PPA for Misere Breakthrough.
 - The playout function
 - The adapt function
 - Combination with UCT
- Take capture into account (PPAF).
- Memorize the policy (PPAFM).
- Compare to UCT.

Outline

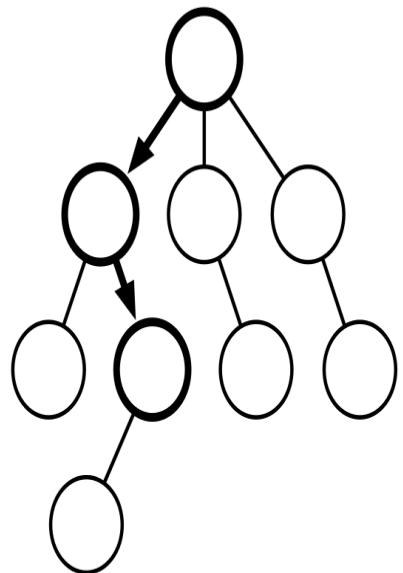
- Algorithm for solving games
- GRAVE and PPAF
- Monte Carlo move ordering
- Experiments
- Conclusion

Solving Games

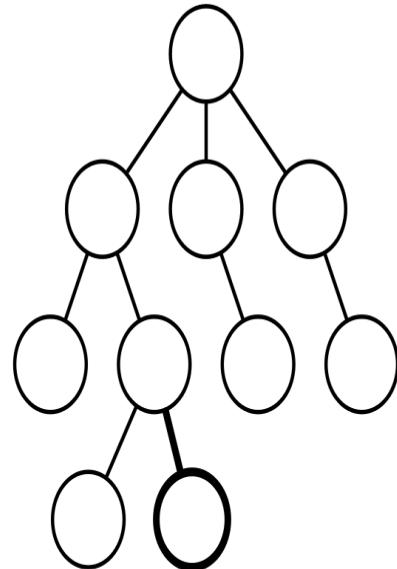
- Proof-Number Search (PN)
- PN²
- Alpha-Beta
- Iterative Deepening Alpha-Beta
- Retrograde Analysis

UCT

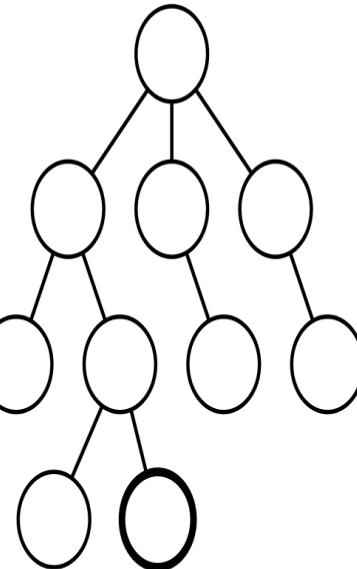
Selection



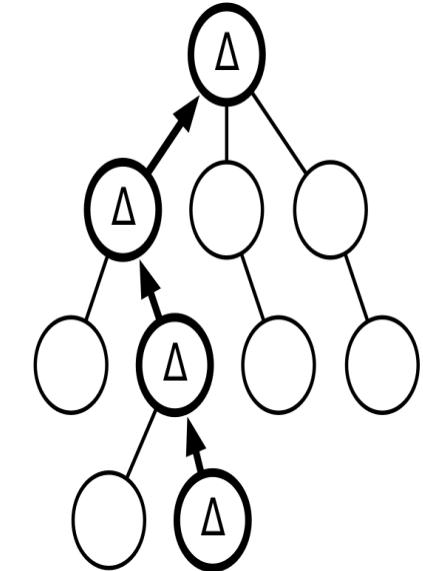
Expansion



Sampling



Backpropagation



Tree Policy

Default Policy



RAVE

- A big improvement for Go, Hex and other games is Rapid Action Value Estimation (RAVE) [Gelly and Silver 2007].
- RAVE combines the mean of the playouts that start with the move and the mean of the playouts that contain the move (AMAF).

RAVE

- Parameter β_m for move m is :

$$\beta_m \leftarrow p\text{AMAF}_m / (p\text{AMAF}_m + p_m + \text{bias} \times p\text{AMAF}_m \times p_m)$$

- β_m starts at 1 when no playouts and decreases as more playouts are played.

- Selection of moves in the tree :

$$\text{argmax}_m((1.0 - \beta_m) \times \text{mean}_m + \beta_m \times \text{AMAF}_m)$$

GRAVE

- Generalized Rapid Action Value Estimation (GRAVE) is a simple modification of RAVE.
- It consists in using the first ancestor node with more than n playouts to compute the RAVE values.
- It is a big improvement over RAVE for Go, Atarigo, Knightthrough and Domineering [Cazenave 2015].

Playout Policy learning

- Start with a uniform policy.
- Use the policy for the playouts.
- Adapt the policy for the winner of each playout.

Playout Policy learning

- Each move is associated to a weight w_i .
- During a playout each move is played with a probability :

$$\exp(w_i) / \sum_k \exp(w_k)$$

Playout Policy learning

- Online learning :
- For each move of the winner :

$$w_i = w_i + 1$$

- For each possible move of each state of the winner :

$$w_i = w_i - \exp(w_i) / \sum_k \exp(w_k)$$

Monte Carlo Game Solver

- Use the order of moves of GRAVE when the state is in the GRAVE tree.
- Use the order of moves of Playout Policy Adaptation when the state is outside the GRAVE tree.

Table 1: Different algorithms for solving Atarigo.

Size	5×5	
Result	Won	
	Move count	Time
PN^2	14 784 088 742	37 901.56 s.
ID $\alpha\beta$ TT	> 35 540 000 000	> 86 400.00 s.
$\alpha\beta$ TT	> 37 660 000 000	> 86 400.00 s.
ID $\alpha\beta$ TT MC	62 800 334	126.84 s.
$\alpha\beta$ TT MC	3 956 049	12.79 s.

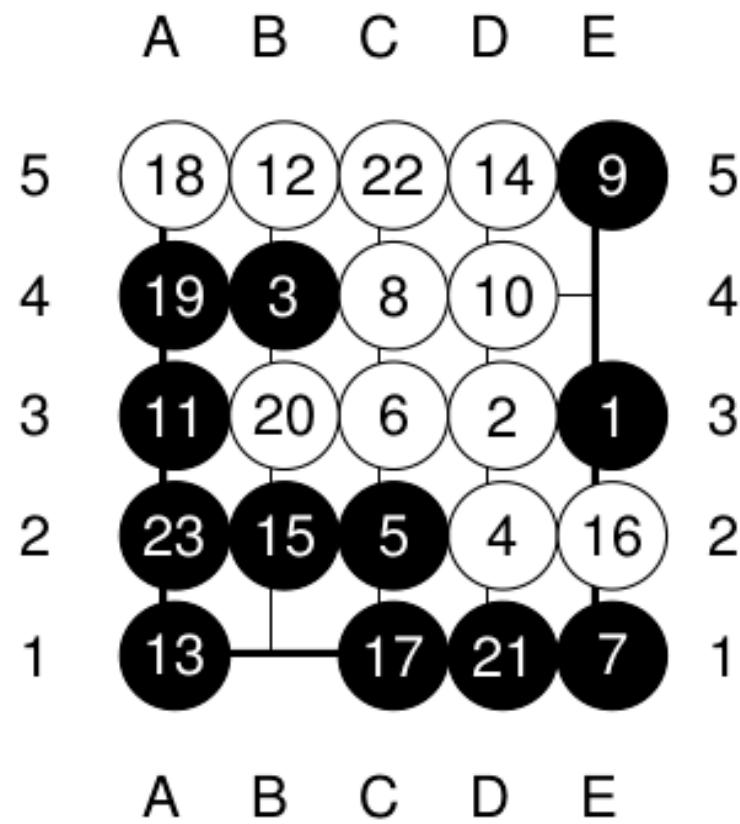
Size	6×5	
Result	Won	
	Move count	Time
PN^2	> 33 150 000 000	> 86 400.00 s.
ID $\alpha\beta$ TT	> 37 190 000 000	> 86 400.00 s.
$\alpha\beta$ TT	> 7 090 000 000	> 44 505.91 s.
ID $\alpha\beta$ TT MC	12 713 931 627	27 298.35 s.
$\alpha\beta$ TT MC	329 780 434	787.17 s.

Table 2: Different algorithms for solving Nogo.

Size	7×3	
Result	Won	
	Move count	Time
PN^2	$> 80\ 390\ 000\ 000$	$> 86\ 400.00$ s.
ID $\alpha\beta$ TT	10 921 978 839	12 261.64 s.
$\alpha\beta$ TT	3 742 927 598	4 412.21 s.
ID $\alpha\beta$ TT MC	1 927 635 856	2 648.91 s.
$\alpha\beta$ TT MC	35 178 886	49.72 s.

Size	5×4	
Result	Won	
	Move count	Time
PN^2	$> 101\ 140\ 000\ 000$	$> 86\ 400.00$ s.
ID $\alpha\beta$ TT	1 394 182 870	1 573.72 s.
$\alpha\beta$ TT	1 446 922 704	1 675.64 s.
ID $\alpha\beta$ TT MC	73 387 083	134.26 s.
$\alpha\beta$ TT MC	33 850 535	74.77 s.

Fig. 1: Solution of Nogo 5×5 .



	1	2	3	4	5	6	7	8	9	10
1	2	1	1	2	1	1	1	1	1	1
2	1	1	2	2	1	1	1	1	2	2
3	1	2	1	2	1	1	1	1		
4	2	2	2	2	1	1				
5	1	1	1	1	1					
6	1	1	1	1						
7	1	1	1							
8	1	1	1							
9	1	2								
10	1	2								

Table 3: Winner for Nogo boards of various sizes

Table 4: Different algorithms for solving Go.

Size	3×3	
Result	Won	
	Move count	Time
PN^2	246 394	3.72 s.
ID $\alpha\beta$ TT	840 707	11.34 s.
$\alpha\beta$ TT	420 265	11.50 s.
ID $\alpha\beta$ TT MC	375 414	5.62 s.
$\alpha\beta$ TT MC	6 104	0.16 s.

Size	4×3	
Result	Won	
	Move count	Time
PN^2	43 202 038	619.98 s.
ID $\alpha\beta$ TT	39 590 950	515.71 s.
$\alpha\beta$ TT	107 815 563	1 977.86 s.
ID $\alpha\beta$ TT MC	22 382 730	348.08 s.
$\alpha\beta$ TT MC	4 296 893	96.63 s.

Table 5: Different algorithms for solving Breakthrough.

Size	5×5	
Result	Lost	
	Move count	Time
PN^2	$> 38\ 780\ 000\ 000$	$> 86\ 400.00$ s.
ID $\alpha\beta$ TT	13 083 392 799	33 590.59 s.
$\alpha\beta$ TT	19 163 127 770	43 406.79 s.
ID $\alpha\beta$ TT MC	3 866 853 361	11 319.39 s.
$\alpha\beta$ TT MC	3 499 173 137	9 243.66 s.

Table 6: Different algorithms for solving Misere Breakthrough.

Size	4 × 5	
Result	Lost	
	Move count	Time
PN^2	> 42 630 000 000	> 86 400 s.
ID $\alpha\beta$ TT	> 43 350 000 000	> 86 400 s.
$\alpha\beta$ TT	> 42 910 000 000	> 86 400 s.
ID $\alpha\beta$ TT MC	1 540 153 635	3 661.50 s.
$\alpha\beta$ TT MC	447 879 697	1 055.32 s.

Table 7: Different algorithms for solving Knightthrough.

Size	6×6
Result	Won
	Move count Time
PN^2	$> 33\,110\,000\,000$ > 86 400 s.
ID $\alpha\beta$ TT	1 153 730 169 4 894.69 s.
$\alpha\beta$ TT	2 284 038 427 6 541.08 s.
ID $\alpha\beta$ TT MC	17 747 503 102.60 s.
$\alpha\beta$ TT MC	528 783 129 1 699.01 s.

Size	7×6
Result	Won
	Move count Time
PN^2	$> 30\,090\,000\,000$ > 86 400 s.
ID $\alpha\beta$ TT	$> 17\,500\,000\,000$ > 86 400 s.
$\alpha\beta$ TT	$> 29\,980\,000\,000$ > 86 400 s.
ID $\alpha\beta$ TT MC	2 540 383 012 13 716.36 s.
$\alpha\beta$ TT MC	6 650 804 159 23 958.04 s.

Table 8: Different algorithms for solving Misere Knightthrough.

Size	5×5		
Result	Lost	Move count	Time
PN^2	> 45 290 000 000	> 86 400 s.	
ID $\alpha\beta$ TT	> 52 640 000 000	> 86 400 s.	
$\alpha\beta$ TT	> 56 230 000 000	> 86 400 s.	
ID $\alpha\beta$ TT MC	> 41 840 000 000	> 86 400 s.	
$\alpha\beta$ TT MC	20 375 687 163 42 425.41	s.	

Table 9: Different algorithms for solving Domineering.

Size	7×7	
Result	Won	
	Move count	Time
PN^2	$> 41\ 270\ 000\ 000$	$> 86\ 400$ s.
ID $\alpha\beta$ TT	18 958 604 687 35	196.62 s.
$\alpha\beta$ TT	197 471 137	376.23 s.
ID $\alpha\beta$ TT MC	2 342 641 133	5 282.06 s.
$\alpha\beta$ TT MC	29 803 373	123.76 s.

Table 10: Different algorithms for solving Misere Domineering.

Size	7×7	
Result	Won	
	Move count	Time
PN^2	$> 44\ 560\ 000\ 000$	$> 86\ 400$ s.
ID $\alpha\beta$ TT	$> 49\ 290\ 000\ 000$	$> 86\ 400$ s.
$\alpha\beta$ TT	$> 49\ 580\ 000\ 000$	$> 86\ 400$ s.
ID $\alpha\beta$ TT MC	7 013 298 932	14 936.03 s.
$\alpha\beta$ TT MC	72 728 678	212.25 s.

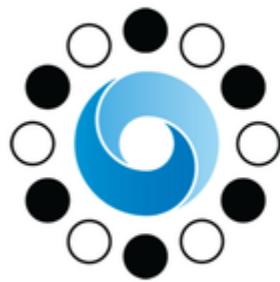
Conclusion

- For the games we solved, Misere Games are more difficult to solve than normal games.
- In Misere Games the player waits and tries to force the opponent to play a losing move.
- This makes the game longer and reduces the number of winning sequences and winning moves.
- Monte Carlo Move Ordering improves much the speed of $\alpha\beta$ with transposition table compare to depth first $\alpha\beta$ and Iterative Deepening $\alpha\beta$ with transposition table but without Monte Carlo Move Ordering.
- The experimental results show significant improvements for nine different games.

Zero Learning

Deep Reinforcement Learning

- AlphaGo
- Golois
- AlphaGo Zero
- Alpha Zero
- Mu Zero
- Polygames



AlphaGo



David Silver



Aja Huang

AlphaGo

Fan Hui is the european Go champion and a 2p professional Go player :

AlphaGo Fan won 5-0
against Fan Hui in
November 2015.

Nature, January 2016.



AlphaGo

Lee Sedol is among the strongest and most famous 9p Go player :



AlphaGo Lee won 4-1 against Lee Sedol in march 2016.

AlphaGo

Ke Jie is the world champion of Go according to Elo ratings :

AlphaGo Master
won 3-0 against
Ke Jie in
may 2017.



AlphaGo

AlphaGo Zero learns to play Go from scratch
playing against itself.

After 40 days of self play it surpasses AlphaGo
Master.

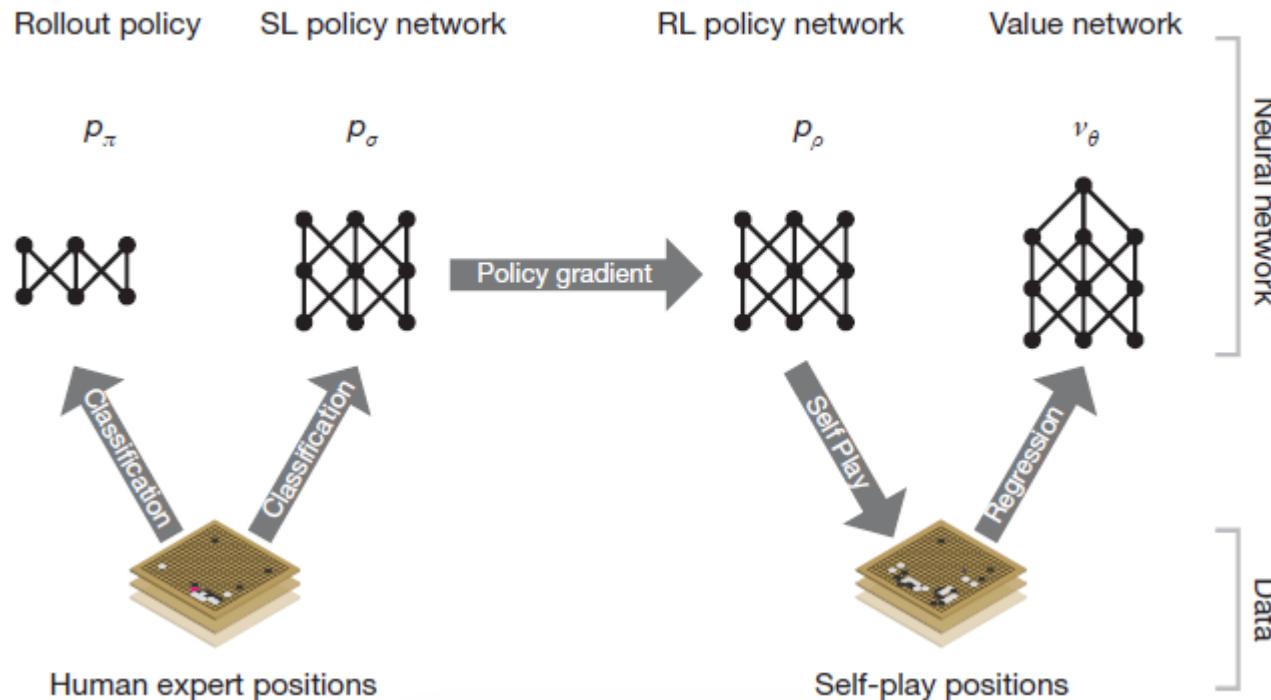
Nature, 18 october 2017.

AlphaGo

- AlphaGo combines MCTS and Deep Learning.
- There are four phases to the development of AlphaGo :
 - Learn strong players moves => policy network.
 - Play against itself and improve the policy network => reinforcement learning.
 - Learn a value network to evaluate states from millions of games played against itself.
 - Combine MCTS, policy and value network.

AlphaGo

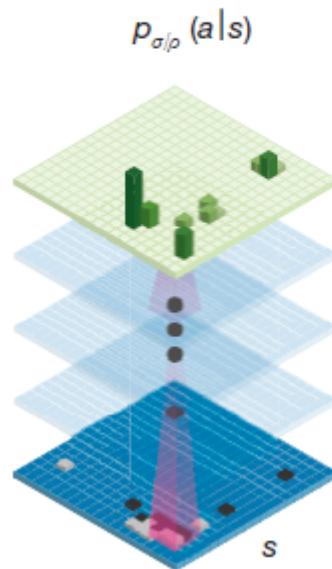
a



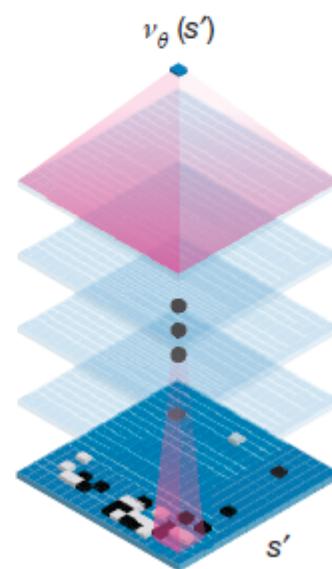
AlphaGo

b

Policy network



Value network



AlphaGo

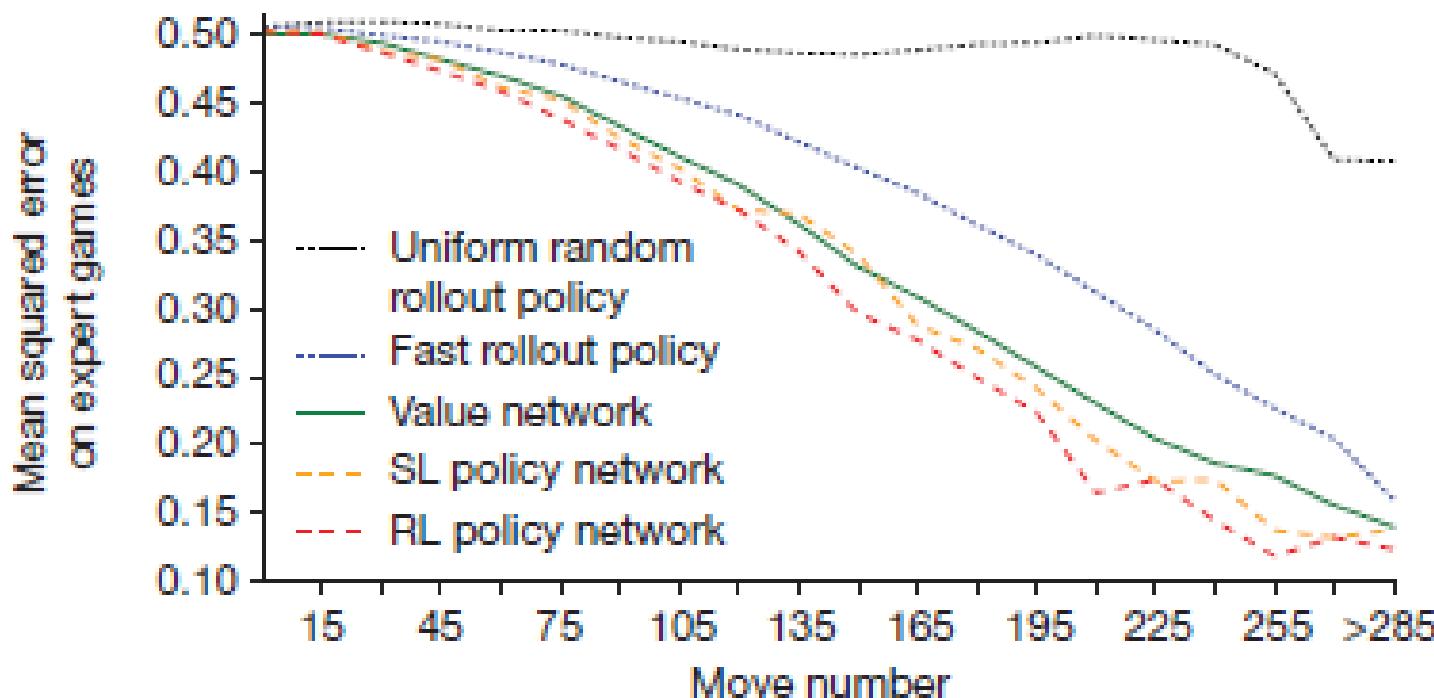
- The policy network is a 13 layers network.
- It uses either 128 or 256 feature planes.
- It is fully convolutional.
- It learns to predict moves from hundreds of thousands of strong players games.
- Once it has learned, it finds the strong player move 57.0 % of the time.
- It takes 3 ms to run.

AlphaGo

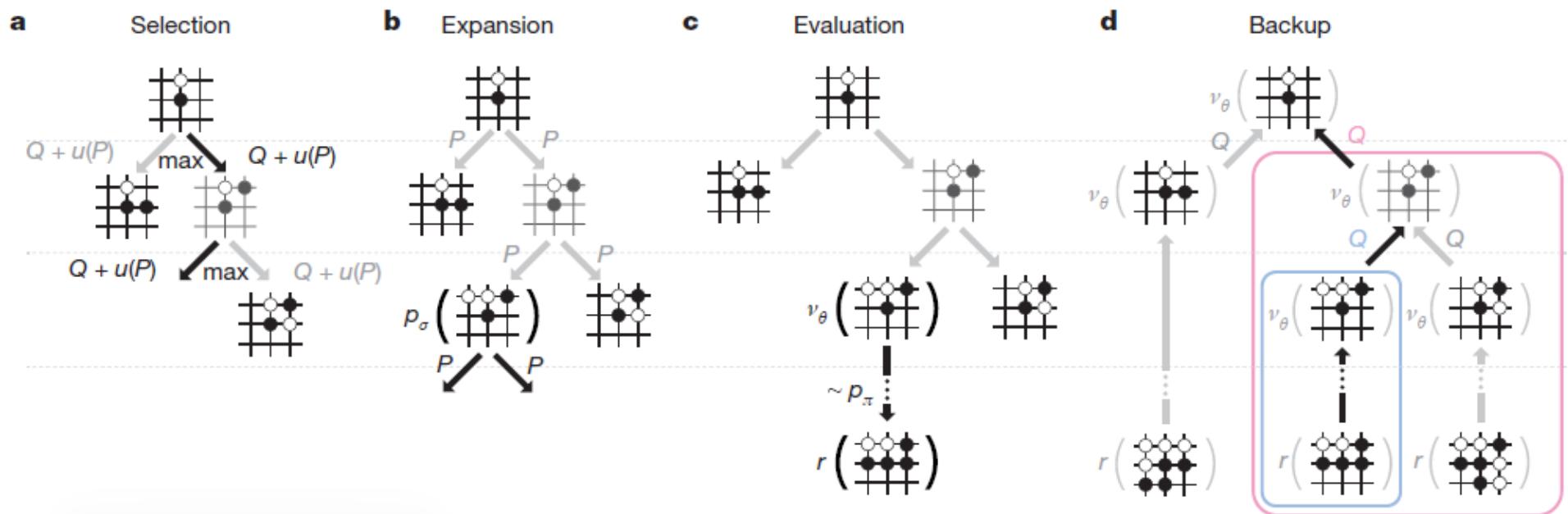
- The value network is also a deep convolutional neural network.
- AlphaGo played a lot of games and kept for each game a state and the corresponding terminal state.
- It learns to evaluate states with the result of the terminal state.
- The value network has learned an evaluation function that gives the probability of winning.

AlphaGo

b



AlphaGo



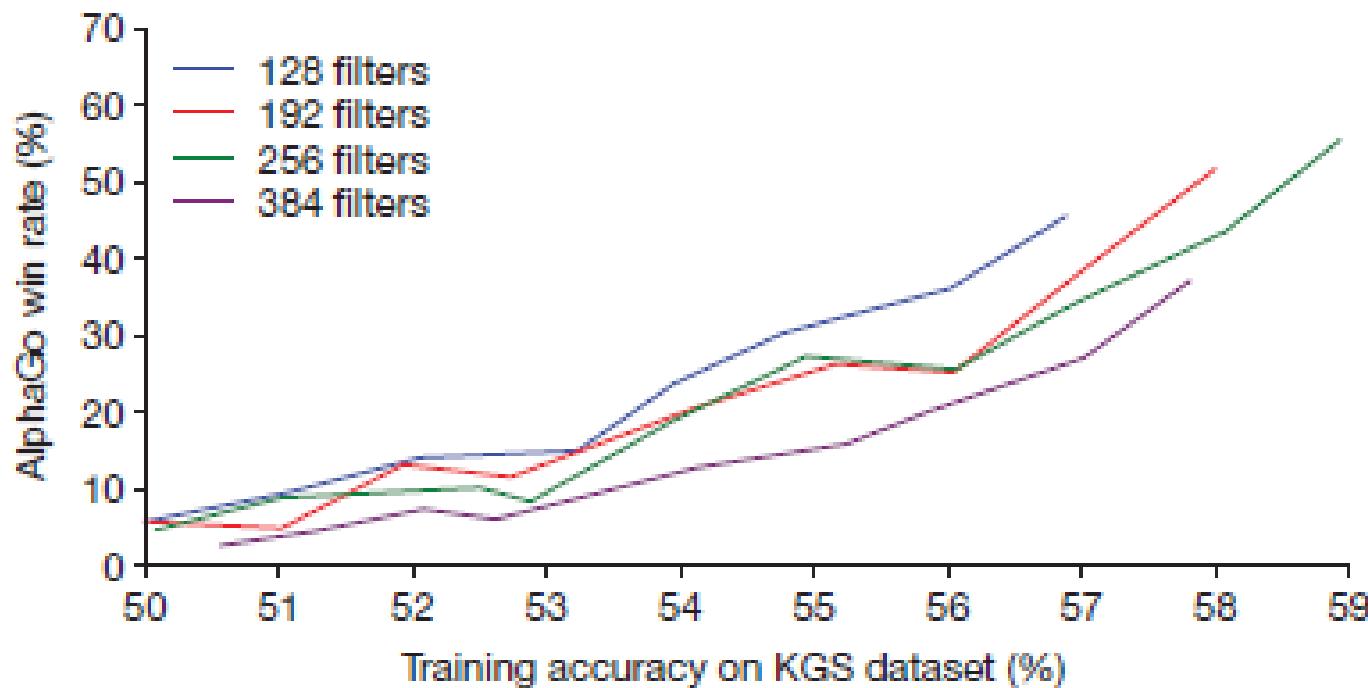
AlphaGo

- The policy network is used as a prior to consider good moves at first.
- Playouts are used to evaluate moves
- The value network is combined with the statistics of the moves coming from the playouts.
- PUCT :

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

AlphaGo

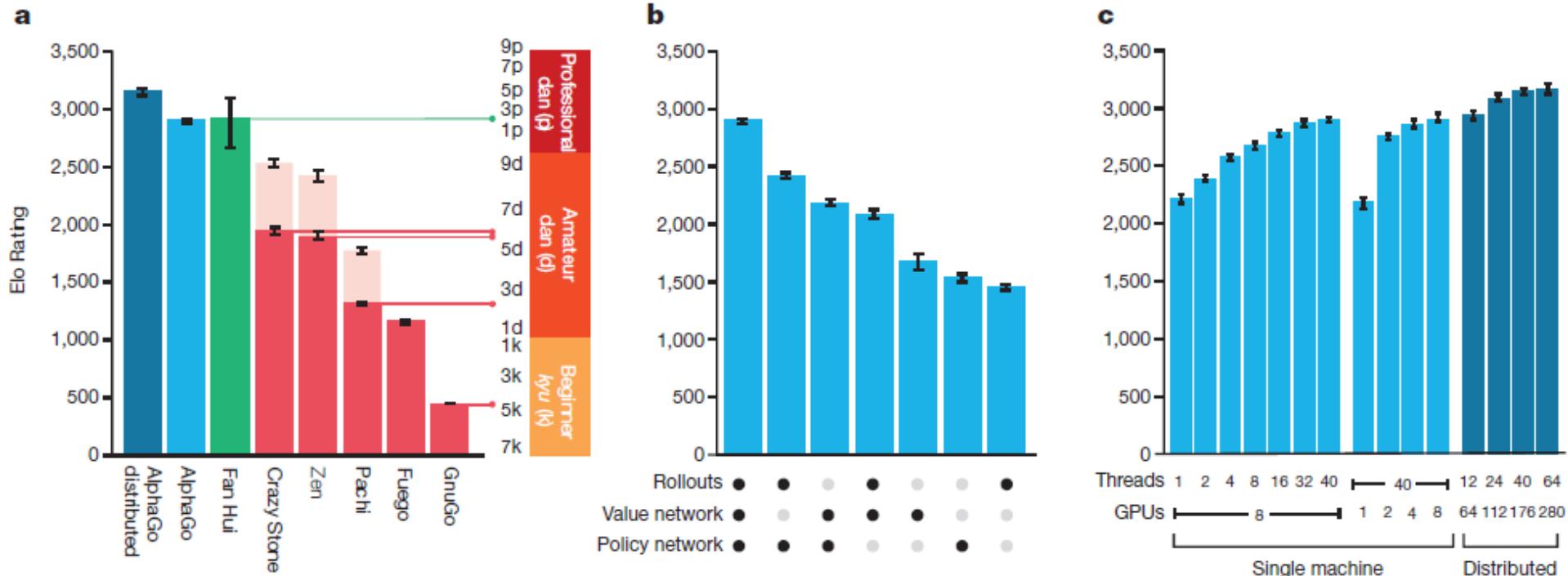
a



AlphaGo

- AlphaGo has been parallelized using a distributed version.
- 40 search threads, 1,202 CPUs and 176 GPU.

AlphaGo



AlphaGo

Extended Data Table 2 | Input features for neural networks

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Feature planes used by the policy network (all but last feature) and value network (all features).

AlphaGo

Extended Data Table 3 | Supervised learning results for the policy network

Architecture			Evaluation				
Filters	Symmetries	Features	Test accuracy %	Train accuracy %	Raw wins %	net AlphaGo wins %	Forward time (ms)
128	1	48	54.6	57.0	36	53	2.8
192	1	48	55.4	58.0	50	50	4.8
256	1	48	55.9	59.1	67	55	7.1
256	2	48	56.5	59.8	67	38	13.9
256	4	48	56.9	60.2	69	14	27.6
256	8	48	57.0	60.4	69	5	55.3
192	1	4	47.6	51.4	25	15	4.8
192	1	12	54.7	57.1	30	34	4.8
192	1	20	54.7	57.2	38	40	4.8
192	8	4	49.2	53.2	24	2	36.8
192	8	12	55.7	58.3	32	3	36.8
192	8	20	55.8	58.4	42	3	36.8

The policy network architecture consists of 128, 192 or 256 filters in convolutional layers; an explicit symmetry ensemble over 2, 4 or 8 symmetries; using only the first 4, 12 or 20 input feature planes listed in Extended Data Table 1. The results consist of the test and train accuracy on the KGS data set; and the percentage of games won by given policy network against AlphaGo's policy network (highlighted row 2): using the policy networks to select moves directly (raw wins); or using AlphaGo's search to select moves (AlphaGo wins); and finally the computation time for a single evaluation of the policy network.

AlphaGo

Extended Data Table 7 | Results of a tournament between different variants of AlphaGo

Short name	Policy network	Value network	Rollouts	Mixing constant	Policy GPUs	Value GPUs	Elo rating
α_{rvp}	p_σ	v_θ	p_π	$\lambda = 0.5$	2	6	2890
α_{vp}	p_σ	v_θ	—	$\lambda = 0$	2	6	2177
α_{rp}	p_σ	—	p_π	$\lambda = 1$	8	0	2416
α_{rv}	$[p_\tau]$	v_θ	p_π	$\lambda = 0.5$	0	8	2077
α_v	$[p_\tau]$	v_θ	—	$\lambda = 0$	0	8	1655
α_r	$[p_\tau]$	—	p_π	$\lambda = 1$	0	0	1457
α_p	p_σ	—	—	—	0	0	1517

Evaluating positions using rollouts only (α_{rp}, α_r), value nets only (α_{vp}, α_v), or mixing both (α_{rv}, α_{rv}); either using the policy network $p_\sigma(\alpha_{np}, \alpha_{vp}, \alpha_{rp})$, or no policy network ($\alpha_{np}, \alpha_{vp}, \alpha_{rp}$), that is, instead using the placeholder probabilities from the tree policy p_τ throughout. Each program used 5 s per move on a single machine with 48 CPUs and 8 GPUs. Elo ratings were computed by BayesElo.

AlphaGo

Extended Data Table 8 | Results of a tournament between AlphaGo and distributed AlphaGo, testing scalability with hardware

<i>AlphaGo</i>	Search threads	CPUs	GPUs	Elo
Asynchronous	1	48	8	2203
Asynchronous	2	48	8	2393
Asynchronous	4	48	8	2564
Asynchronous	8	48	8	2665
Asynchronous	16	48	8	2778
Asynchronous	32	48	8	2867
Asynchronous	40	48	8	2890
Asynchronous	40	48	1	2181
Asynchronous	40	48	2	2738
Asynchronous	40	48	4	2850
Distributed	12	428	64	2937
Distributed	24	764	112	3079
Distributed	40	1202	176	3140
Distributed	64	1920	280	3168

Each program played with a maximum of 2 s thinking time per move. Elo ratings were computed by BayesElo.

Golois

Golois

- I replicated the AlphaGo experiments with the policy and value networks.
- Golois policy network scores 58.54% on the test set (57.0% for AlphaGo).
- Golois plays on the kgs internet Go server.
- It has a strong 4d ranking just with the learned policy network (AlphaGo policy network is 3d).

Data

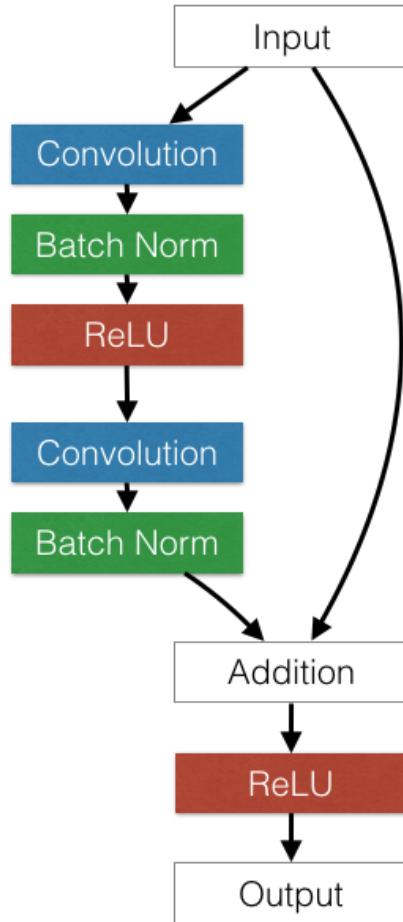
- Learning set = games played on the KGS Go server by players being 6d or more between 2000 and 2014.
- No handicap games.
- Each position is rotated to eight possible symmetric positions.
- 160 000 000 positions in the learning set.
- Test set = games played in 2015.
- 100 000 different positions not mirrored.

Training

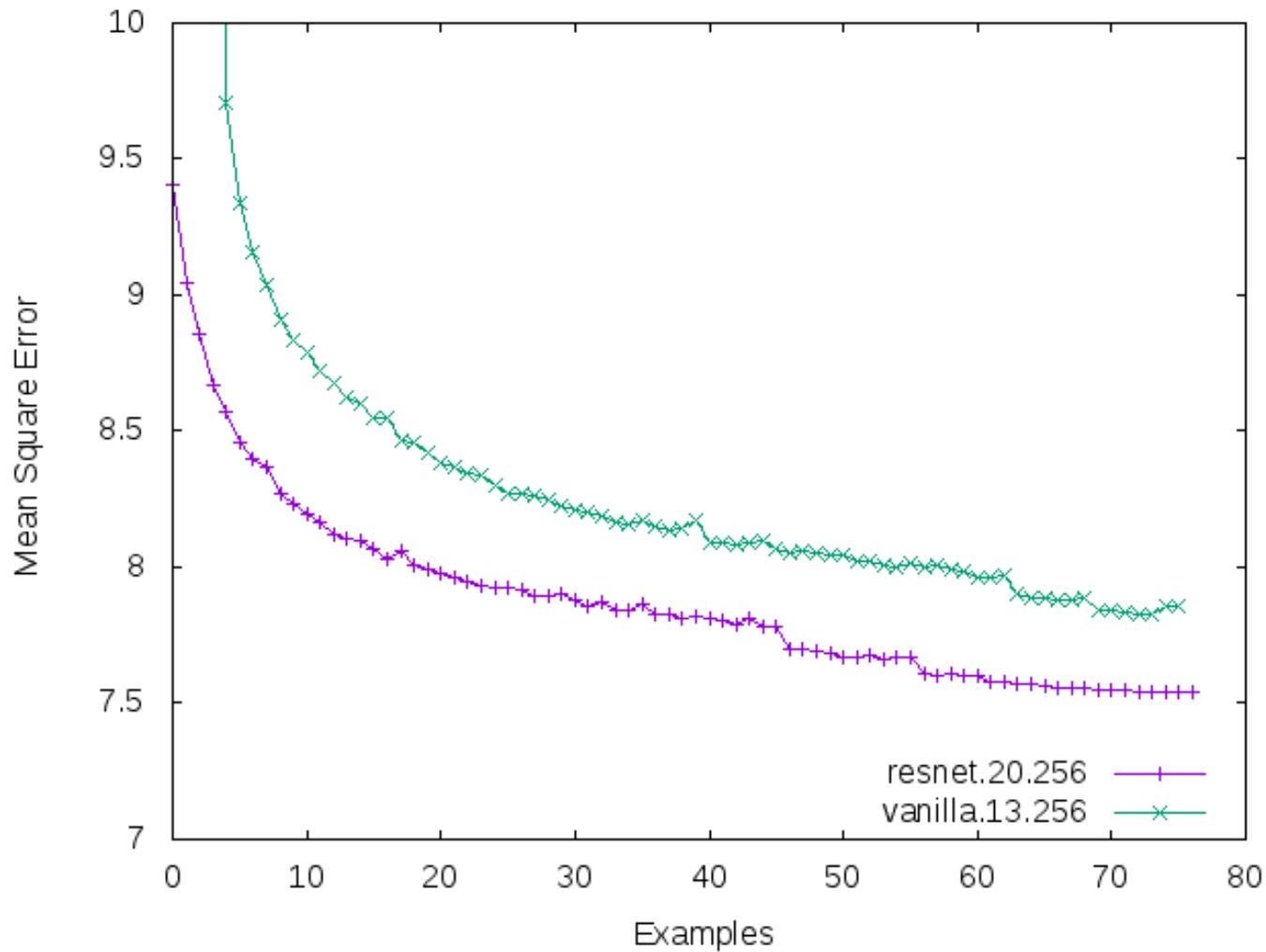
- In order to train the network we build minibatches of size 50 composed of 50 states chosen randomly in the training set.
- Each state is randomly mirrored to one of its eight symmetric states.
- We define an epoch as 5 000 000 training examples.
- The accuracy and the error on the test set are computed every epoch.
- The learning rate is divided by two each time the training error stalls, i.e. is greater than the previous average training error over the last epoch.

Residual Nets

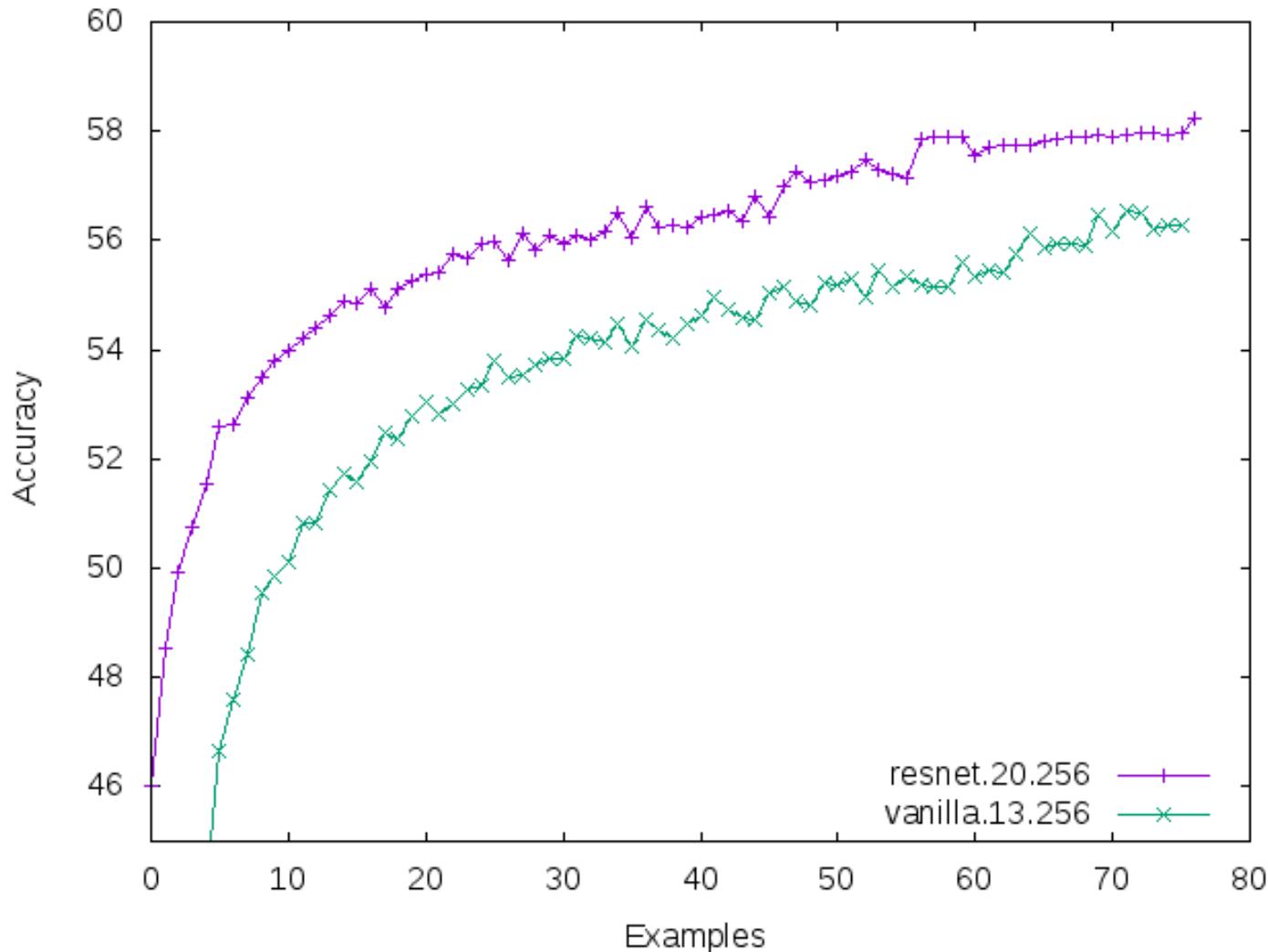
- Residual Nets :



Evolution of the error



Evolution of the accuracy



Golois Policy Network

- Using residual network enables to train deeper network.
- It enables better accuracy than AlphaGo policy network.
- It has a 4 dan level on kgs, playing moves instantly.
- Next: value network and parallel MCTS.

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

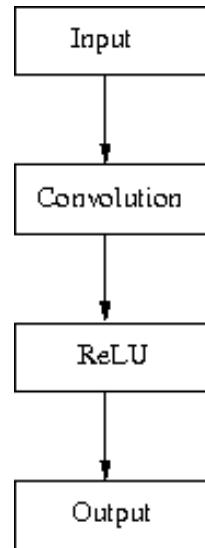
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

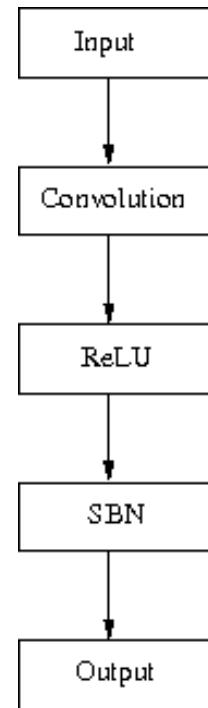
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

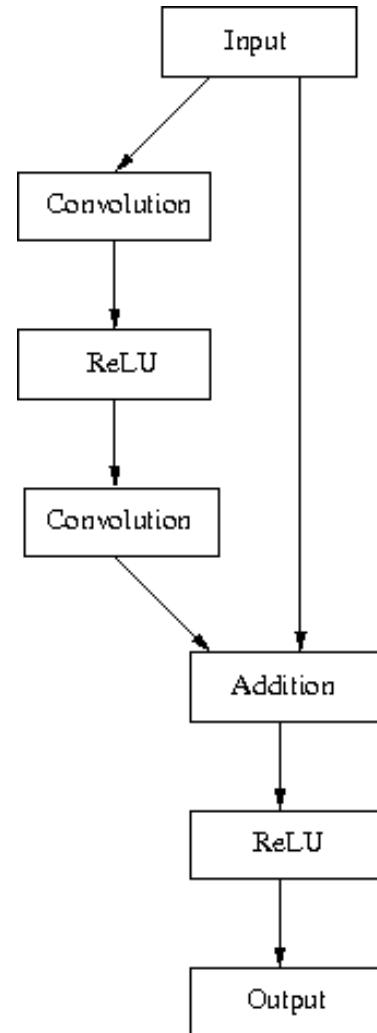
Usual Layer



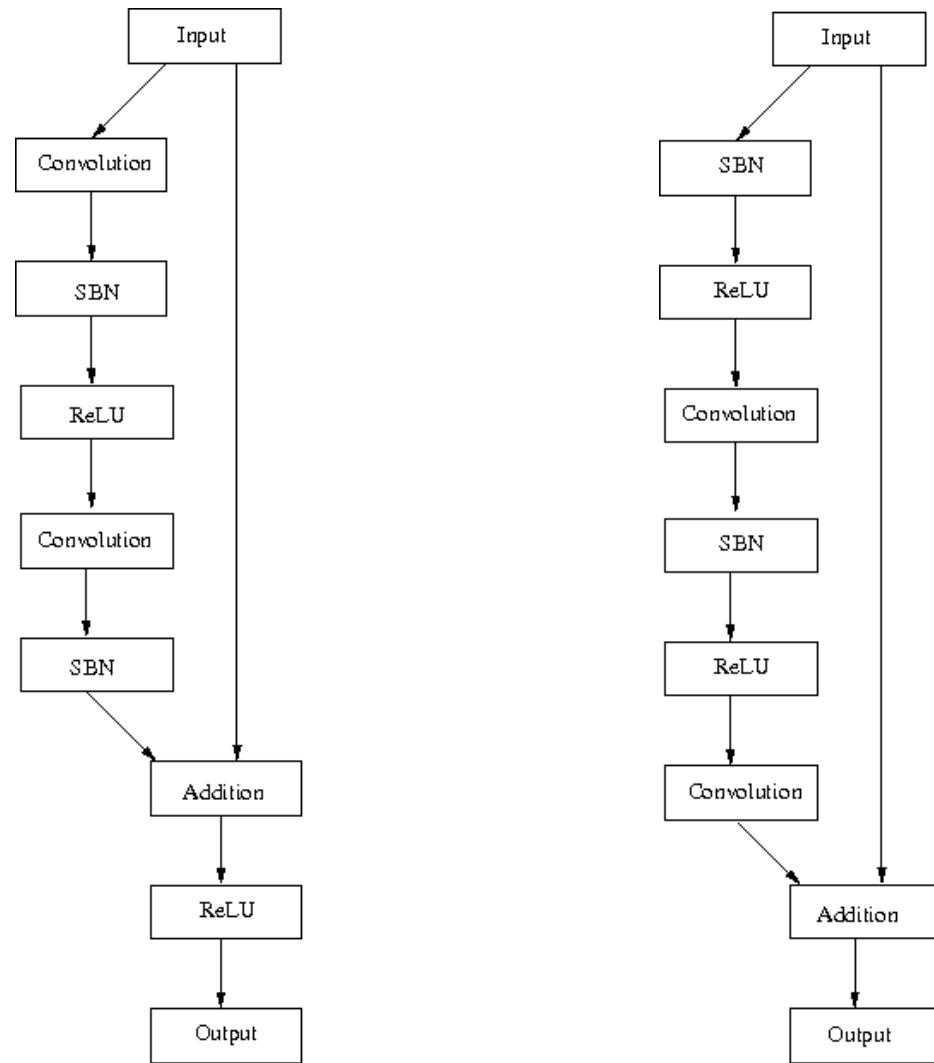
DarkForest Layer



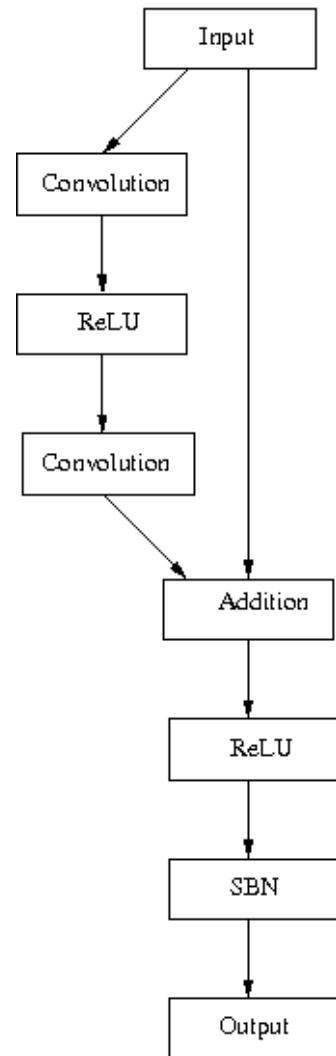
Residual Layer



Residual Layers for Images



Golois Layer



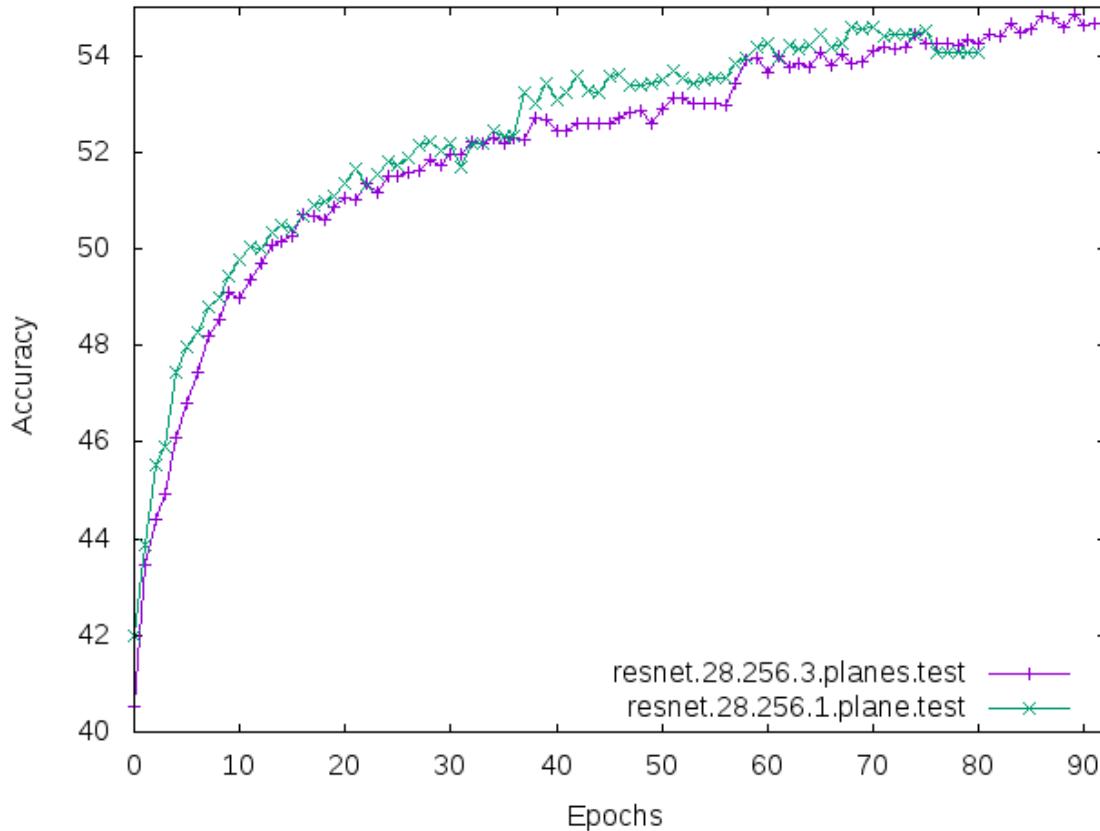
Multiple Output Planes

- In DarkForest it improves the level of play to predict the next three moves instead of only the next move.
- We compared the next move prediction and the next three moves prediction with the Golois layer.

Data

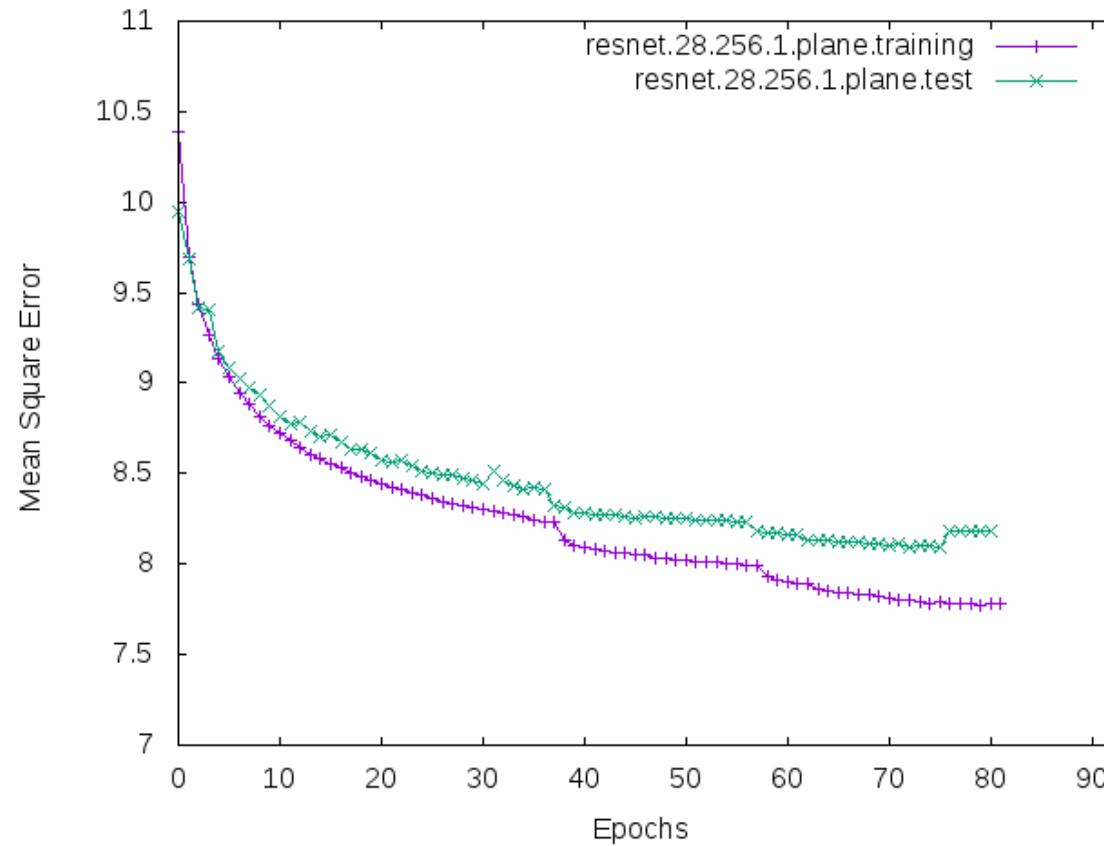
- We use the GoGoD dataset.
- It is composed of many professional games played until today.
- We used the games from 1900 to 2014 for the training set and the games from 2015 and 2016 as the test set.
- The first 500 000 positions of the test set to evaluate the error and the accuracy of the networks.

Multiple Output Planes



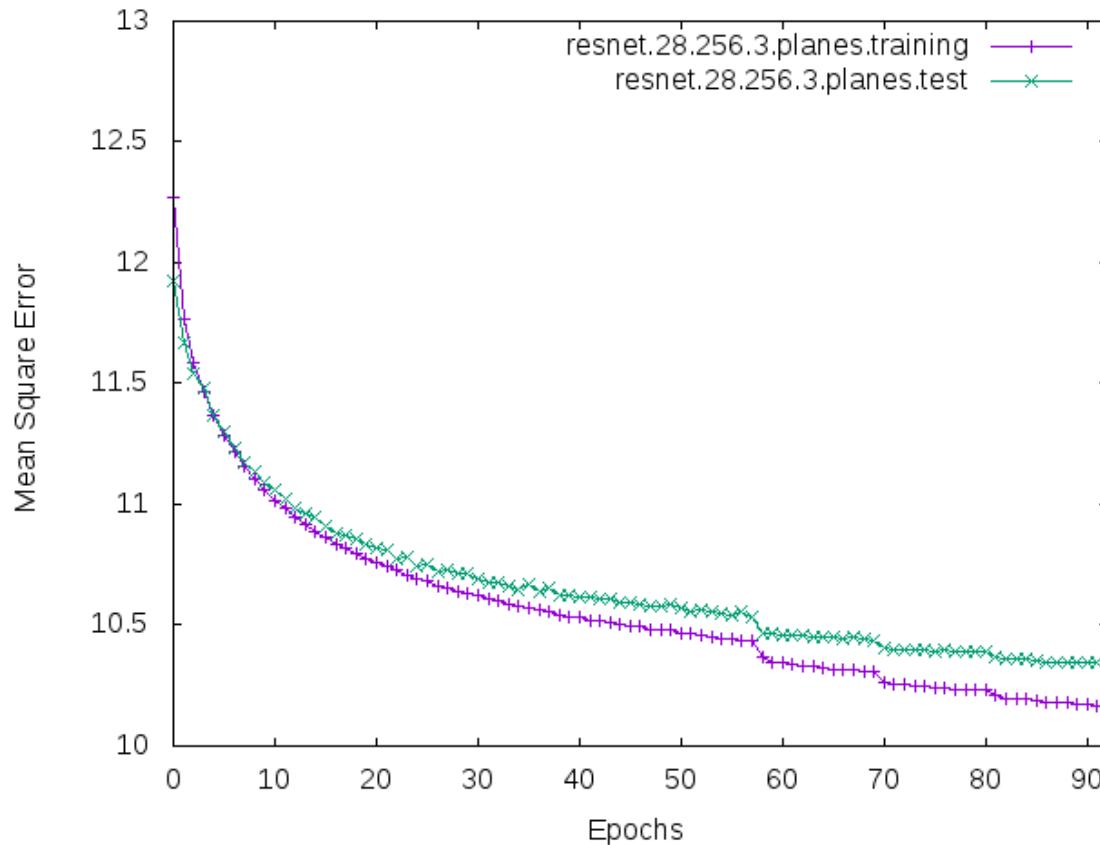
- Evolution of the accuracy for 1 and 3 output planes

Multiple Output Planes



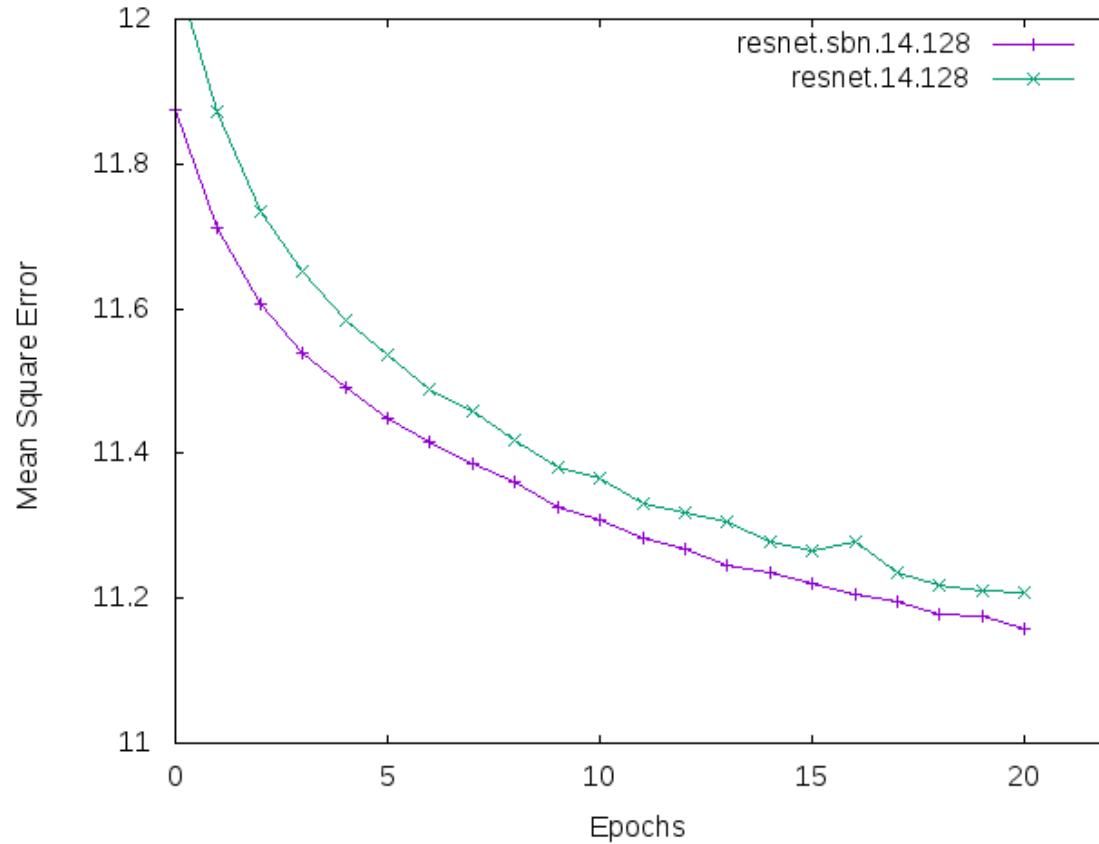
- Evolution of the errors for one output plane

Multiple Output Planes



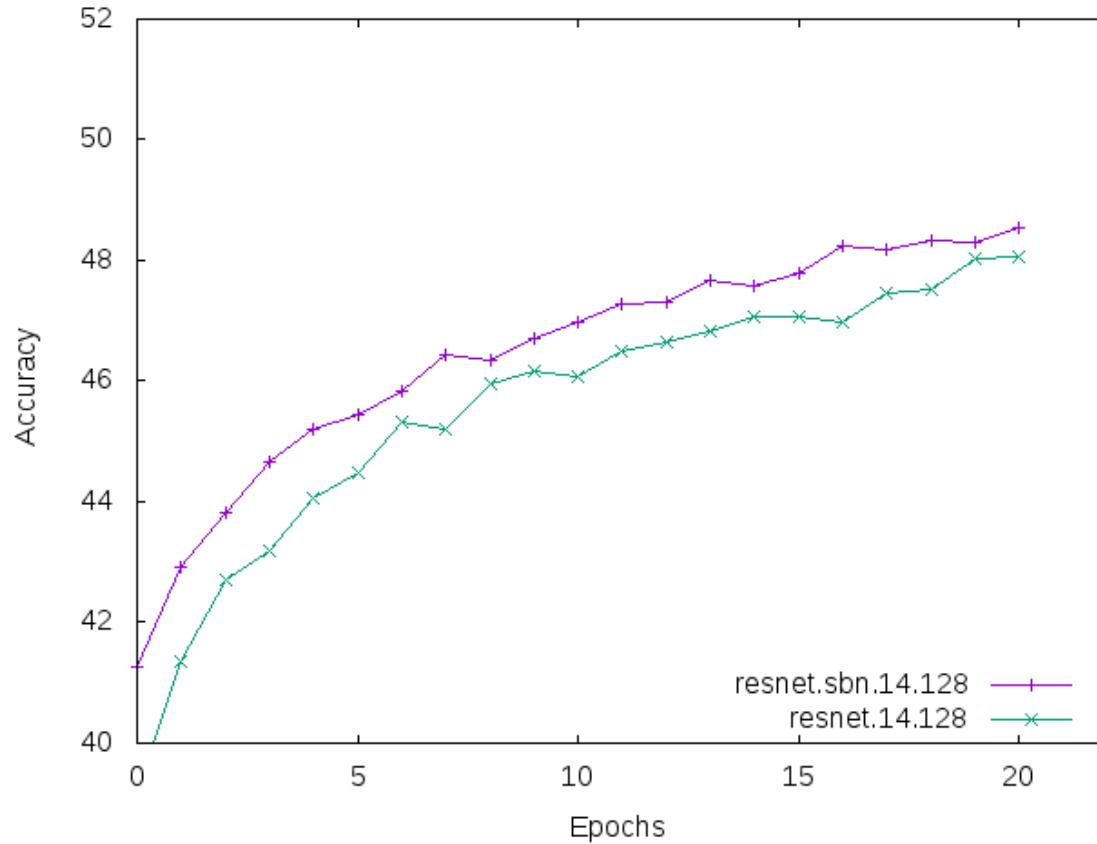
- Evolution of the errors for three output planes

Golois Layer



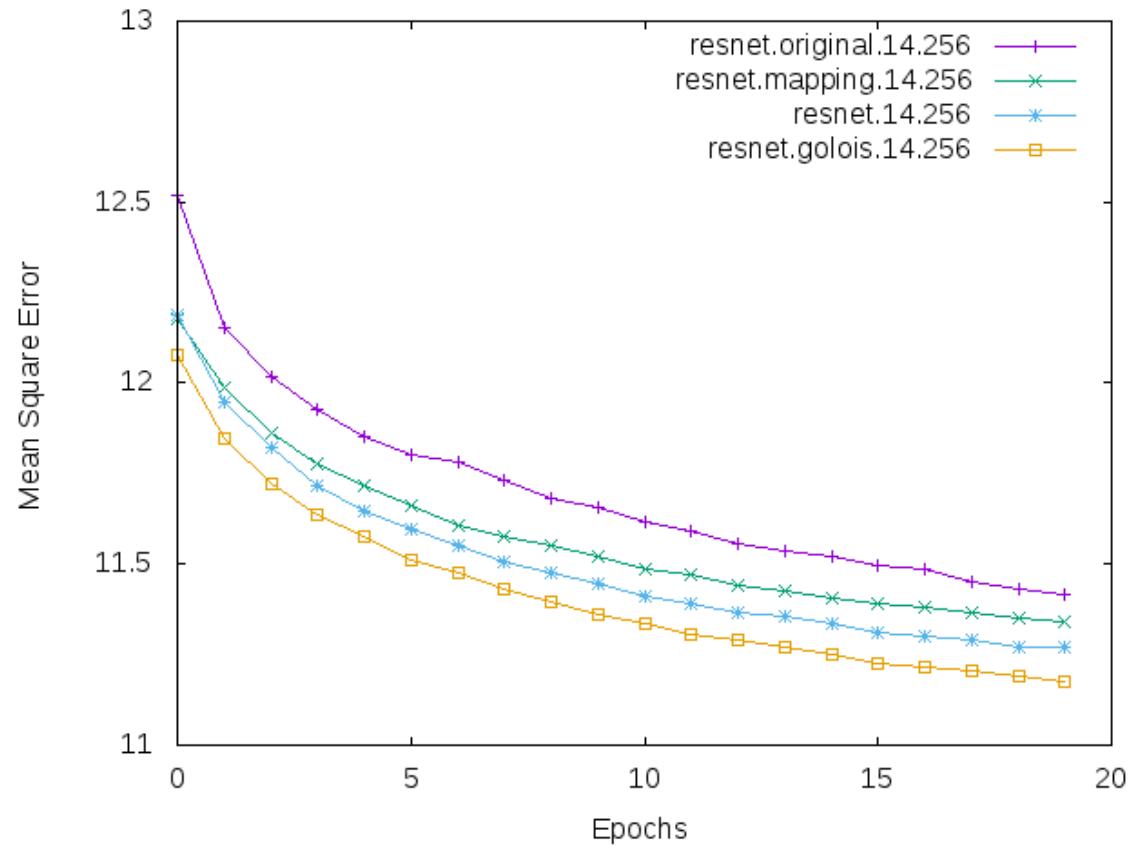
- Evolution of the error for resnet and Golois layers

Golois Layer



- Evolution of the accuracy for resnet and Golois layers

Golois Layer



- Evolution of the error for different layers

Golois Layer

- Multiple Output Planes improves the generalization of residual networks.
- Spatial Batch Normalization helps train the residual network faster and obtain better accuracy.

Value Network

- The Golois policy network played 1,600,000 games against itself.
- 20 layers residual value network trained on these games.
- Parallel PUCT with policy and value networks.
- No playouts.
- Using this value network Golois reached 8d.
- ELF network => 9d.

AlphaGo Zero

AlphaGo Zero

- AlphaGo Zero starts learning from scratch.
- It uses the raw representation of the board as input, even liberties are not used.
- It has 15 input planes, 7 for the previous Black stones, 7 for the previous White Stones and 1 plane for the color to play.

AlphaGo Zero

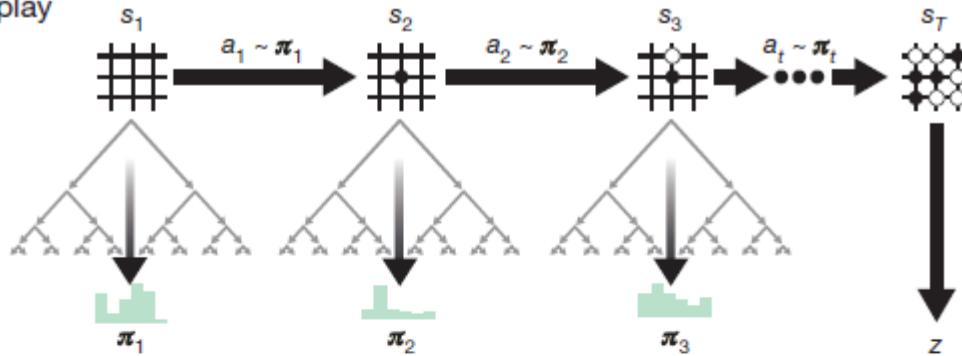
- It plays against itself using PUCT and 1,600 tree descent

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

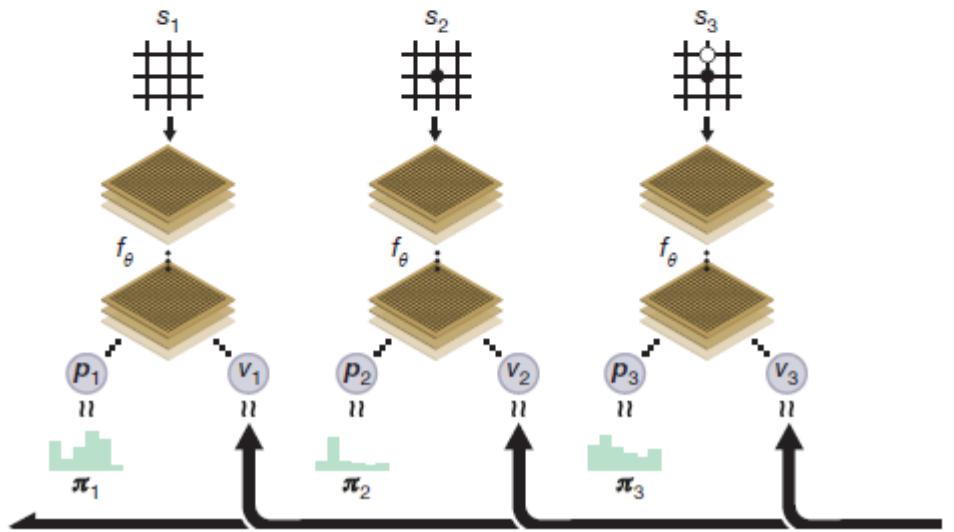
- It uses a residual neural network with two heads.
- One head is the policy, the other head is the value.

AlphaGo Zero

a Self-play



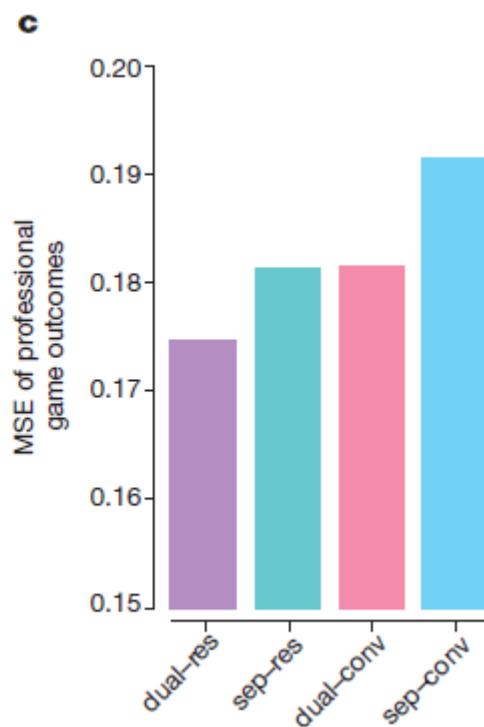
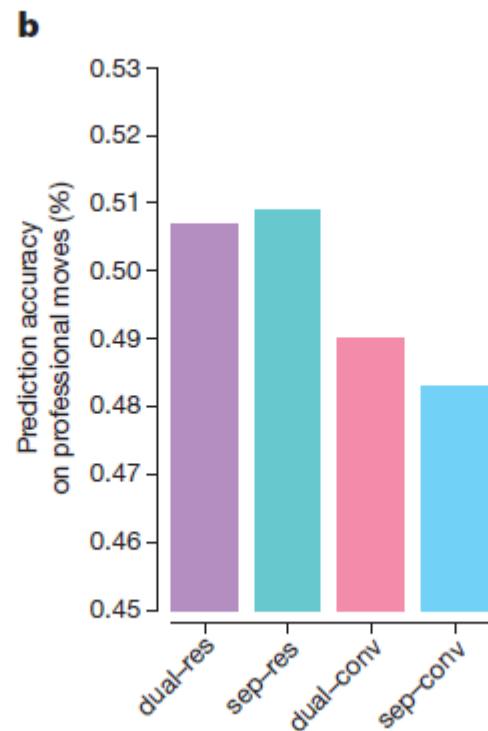
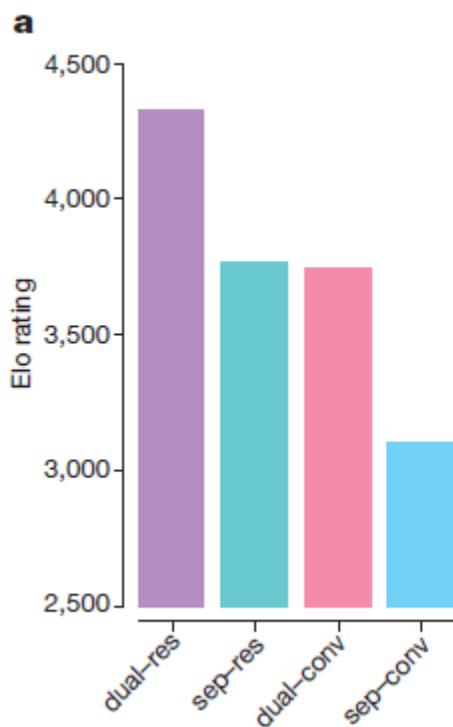
b Neural network training



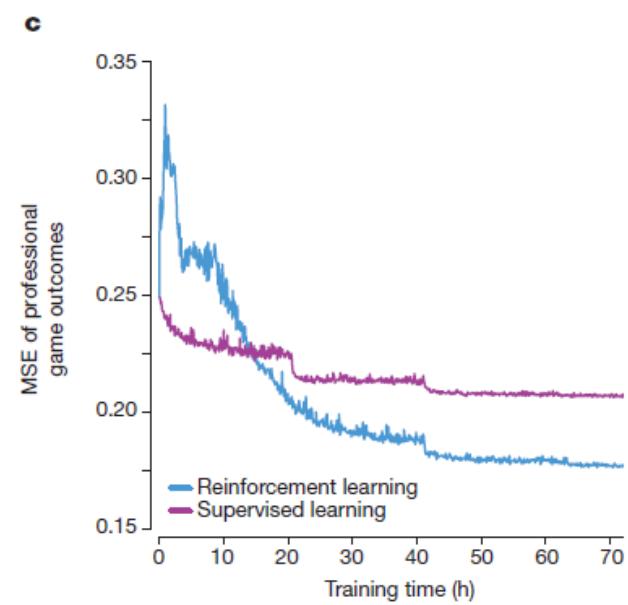
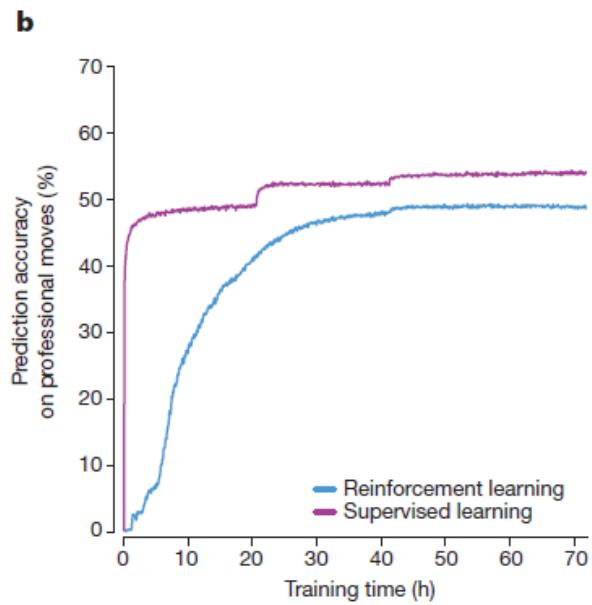
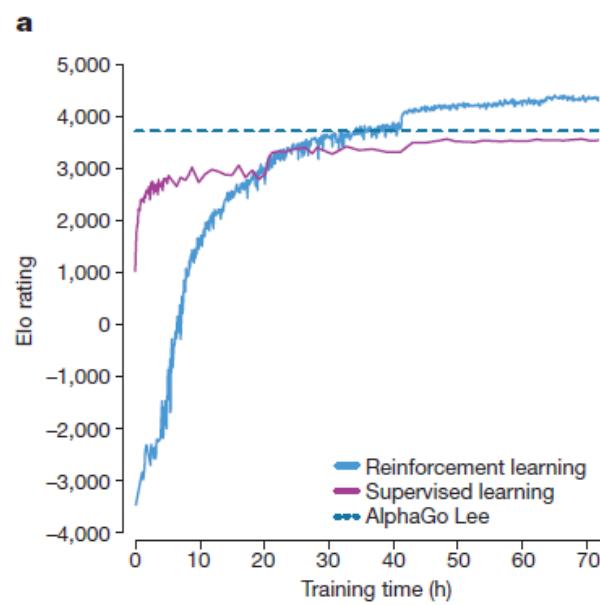
AlphaGo Zero

- After 4.9 million games against itself a 20 residual blocks neural network reaches the level of AlphaGo Lee (100-0).
- 3 days of self play on the machines of DeepMind.
- Comparison : Golois searches 1,600 nodes in 10 seconds on a 4 GPU machine.
- It would take Golois 466 years to play 4.9 million such games.

AlphaGo Zero



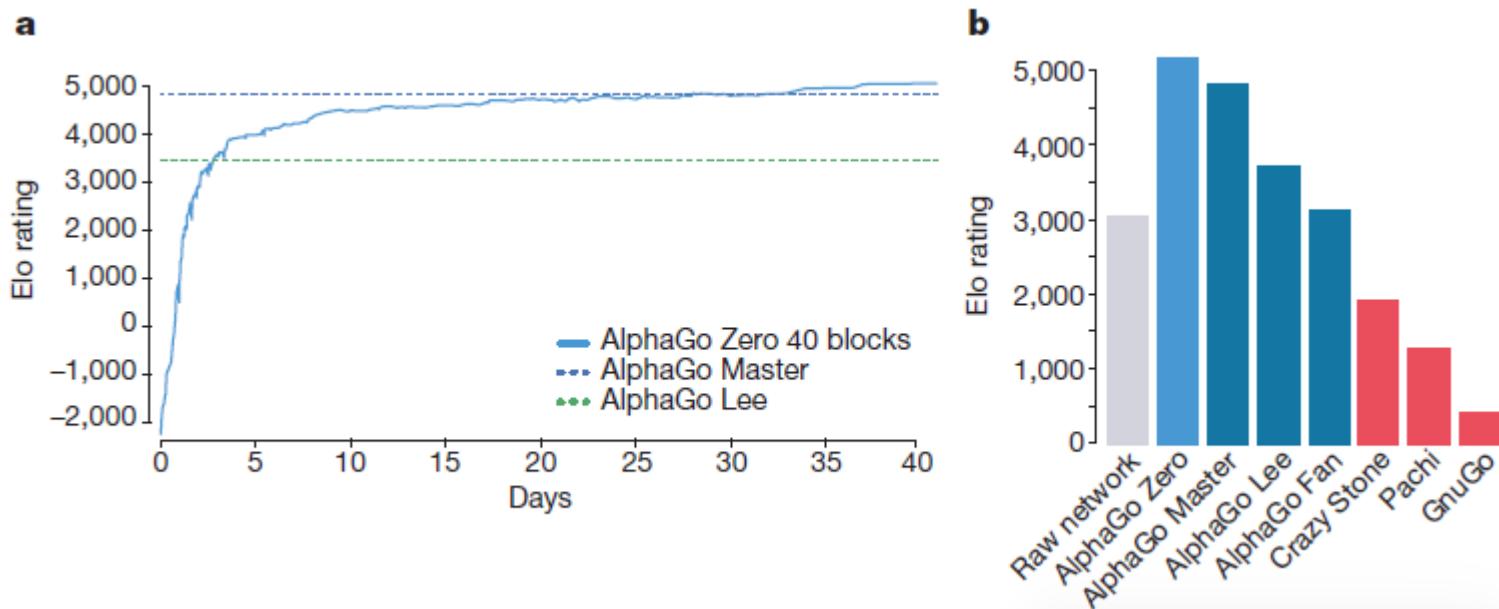
AlphaGo Zero



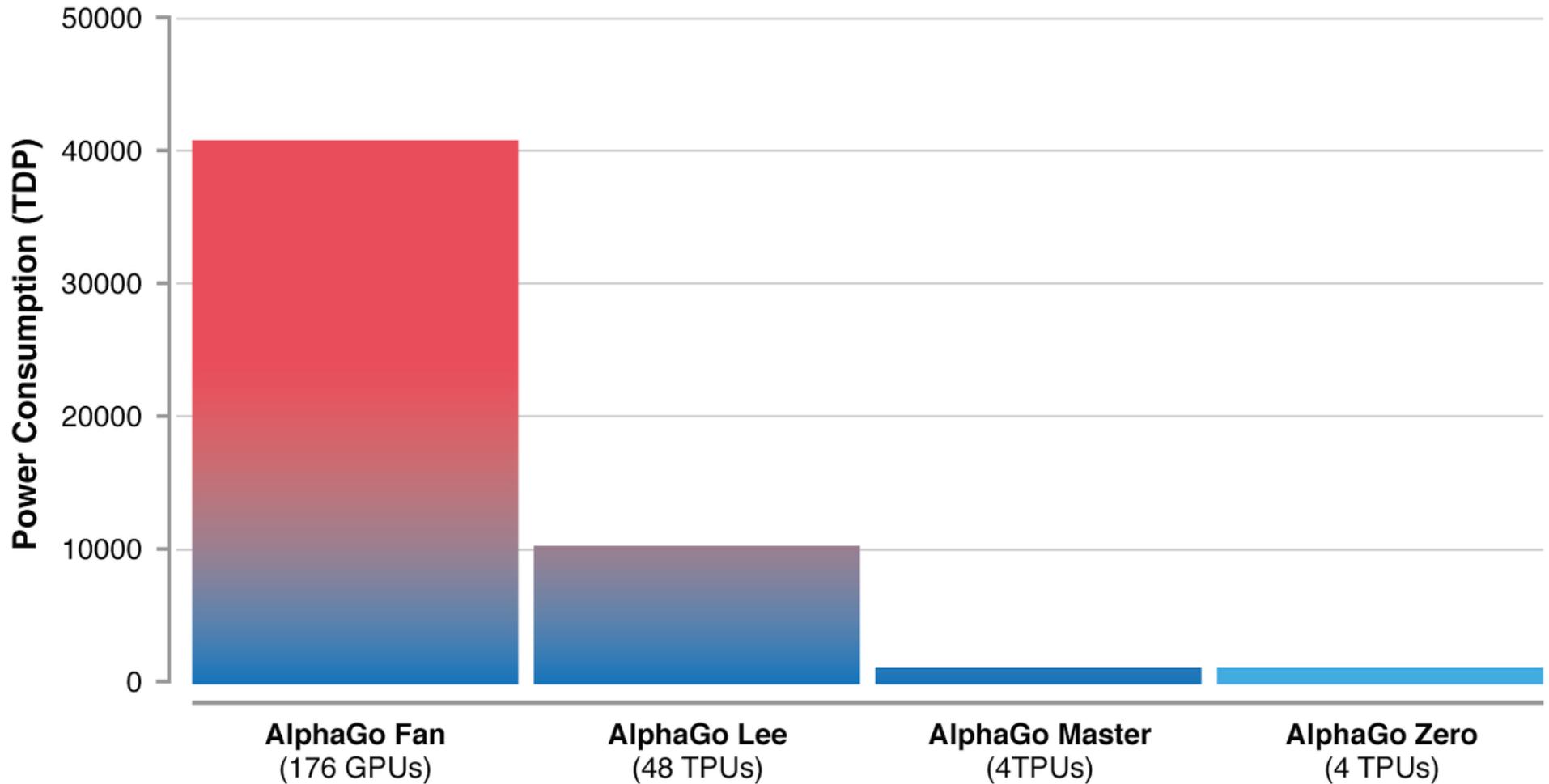
AlphaGo Zero

- They used a longer experiment with a deeper network.
- 40 residual blocks.
- 40 days of self play on the machines of DeepMind.
- In the end it beats Master 89-11.

AlphaGo Zero



AlphaGo Zero



AlphaGo Zero

- AlphaGo Zero uses 40 residual blocks instead of 20 blocks for AlphaGo Master.
- With 20 blocks learning stalls after 3 days.
- Master with 40 blocks better than AlphaGo Zero?

Alpha Zero

Alpha Zero

- Arxiv, 5 december 2017.
- Deep reinforcement learning similar to AlphaGo Zero.
- Same algorithm applied to two other games :
Chess and Shogi.
- Learning from scratch without prior knowledge.

Alpha Zero

- Alpha Zero surpasses Stockfish at Chess after 4 hours of self-play.
- Alpha Zero surpasses Elmo at shogi after 2 hours of self play.

Program	Chess	Shogi	Go
<i>AlphaZero</i>	80k	40k	16k
<i>Stockfish</i>	70,000k		
<i>Elmo</i>		35,000k	

Table S4: Evaluation speed (positions/second) of *AlphaZero*, *Stockfish*, and *Elmo* in chess, shogi and Go.

Alpha Zero

- 5 000 first generation TPU for training.
- 4 TPU for playing.

	Chess	Shogi	Go
Mini-batches	700k	700k	700k
Training Time	9h	12h	34h
Training Games	44 million	24 million	21 million
Thinking Time	800 sims 40 ms	800 sims 80 ms	800 sims 200 ms

Table S3: Selected statistics of *AlphaZero* training in Chess, Shogi and Go.

Mu Zero

Mu Zero

- Arxiv, december 2019.
- Similar to Alpha Zero without knowing the rules of the games.
- Atari, Go, Chess and Shogi.
- Learning from scratch without prior knowledge.

Polygames

Polygames

- Alpha Zero approach for many games.
- A common interface to all the games.
- Fully convolutional network, average pooling...
- Pytorch and C++.
- Open source !

Conclusion

- AlphaGo : supervised learning and self play.
- Golois : residual networks and Spatial Batch Normalization improve learning.
- AlphaGo Zero : reinforcement learning from self play with MCTS. Raw inputs. Residual networks and combined policy and value network. Better than Master.
- Alpha Zero : Go, Chess and Shogi.
- Mu Zero : Atari, Go, Chess and Shogi.
- Polygames : many games.

Alpha Zero

Alpha Zero

- Define a network that takes as input the Breakthrough board and gives as output the policy and the value for the board.
- Bias the MCTS with policy and value using PUCT.
- Make the network play games and record the results of the Monte Carlo and the result of the games.
- Train the network on the results of the games.
- Iterate.

Alpha Zero

- The network takes 41 inputs with values 0 or 1, 20 inputs for black pawns, 20 inputs for white pawns and one input for the color to play.
- Option: also use previous boards as inputs.
- The network has 60 outputs for the policy head (3 possible moves for each cell), and 1 output for the value head.
- The architecture of the network can be completely connected as a starting point.
- Option : convolutional network, residual network.

Alpha Zero

- 1) Define the network
- 2) Implement the PUCT algorithm using the network.
Use the same network for black and white, rotate the board for white so that moves are always forward.
- 3) Make the algorithm play against itself.
- 4) Record the Monte Carlo distributions and the result of self played games.
- 5) Train the network on the recorded data.

Projet Python

- Transformer une position de breakthrough 5x5 en trois matrices 5x5 de 0 et de 1 (Noir/Blanc/Vide).
- Faire deux réseaux convolutifs (blanc et noir) avec 76 sorties (75 coups possibles + évaluation) et ces trois matrices en entrée.
- Utiliser les réseaux dans PUCT pour politique et évaluation.
- Faire jouer à PUCT >100 parties contre lui même.
- Mémoriser pour chaque position un vecteur de 76 réels entre 0 et 1 (une fréquence pour chaque code de coup entre 0 et 75, $\text{code} = 3 * (5 * x + y) + 0, 1 \text{ ou } 2$) et un réel (1.0 si blanc a gagné, 0.0 sinon).
- Entraîner les deux réseaux convolutifs pour retrouver les fréquences et le résultat de la partie en sortie pour chaque position en entrée.
- Itérer.

Monte Carlo Search with Imperfect Information

Information Set MCTS

- Flat Monte Carlo Search gives good results for Phantom Go.
- Information Set MCTS.
- Card games.

Counter Factual Regret Minimization

- Poker : Libratus (CMU), DeepStack (UofA).
- Approximation of the Nash Equilibrium.
- There are about 320 trillion “information sets” in heads-up limit hold’em.
- What the algorithm does is look at all strategies that do not include a move, and count how much we “regret” having excluded the move from our mix.
- Combination with neural networks.
- Better than top professional players.

$\alpha\mu$

- Bridge
- Generate a set of possible worlds.
- Solve each world exactly
- Search multiple moves ahead
- Strategy Fusion => joint search
- Non Locality => Pareto fronts

PIMC

For all possible moves

For all possible worlds

Exactly solve the world

Play the move winning in the most worlds

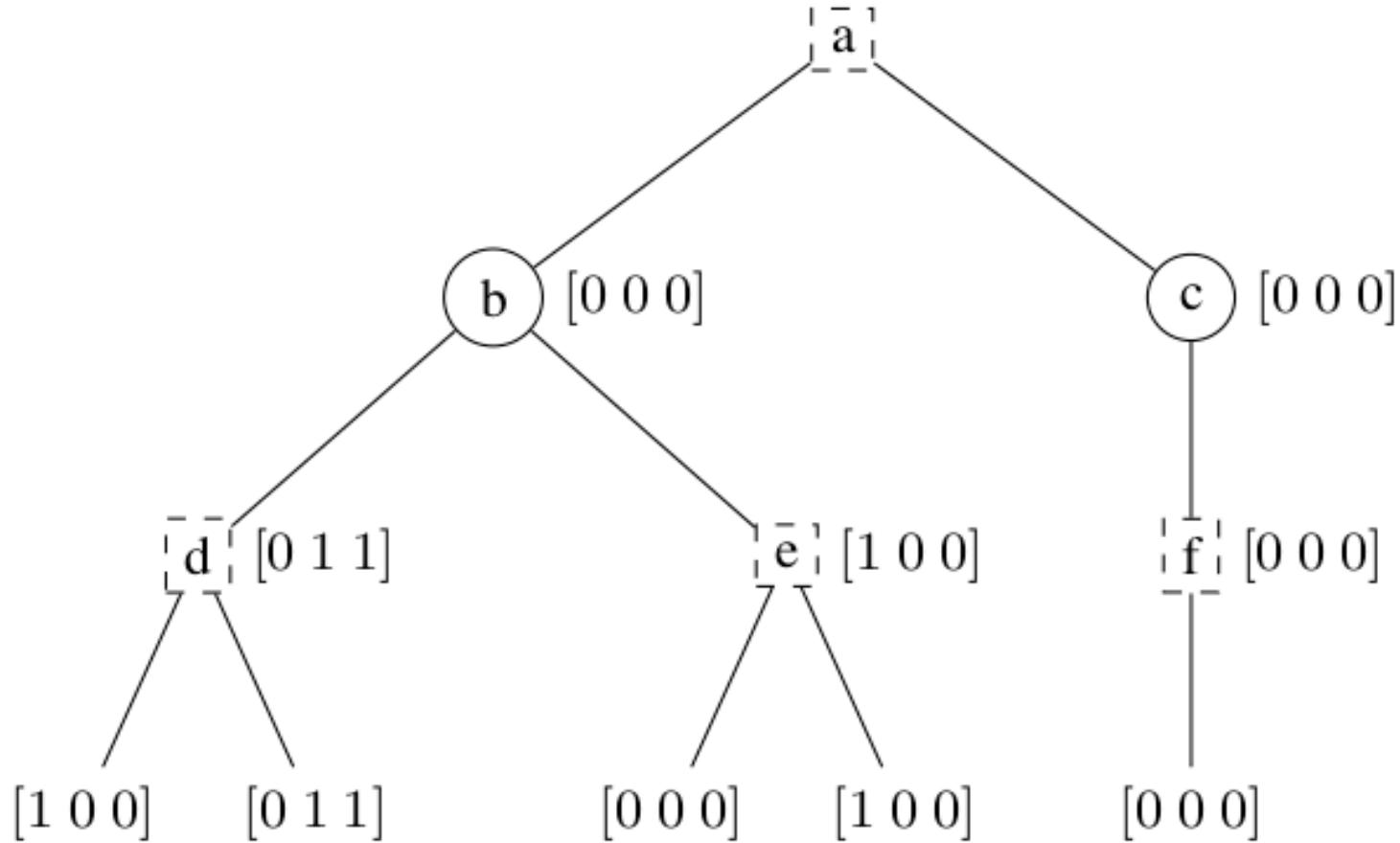
Strategy Fusion

- Problem = PIMC can play different moves in different worlds.
- Whereas the player cannot distinguish between the different worlds.

♠KJT7
♥AKQ
♦AKQ
♣XXX
N
S

♠A986
♥XXX
♦XXX
♣AKQ

Non Locality



Pareto Fronts

- A Pareto Front is a set of vectors.
- It maintains the set of vectors that are not dominated by other vectors.
- Consider the Pareto front $\{[1\ 0\ 0], [0\ 1\ 1]\}$.
- If the vector $[0\ 0\ 1]$ is a candidate for entering the front, then the front stays unchanged since $[0\ 0\ 1]$ is dominated by $[0\ 1\ 1]$.
- If we add the vector $[1\ 1\ 0]$ then the vector $[1\ 0\ 0]$ is removed from the front since it is dominated by $[1\ 1\ 0]$, and then $[1\ 1\ 0]$ is inserted in the front. The new front becomes $\{[1\ 1\ 0], [0\ 1\ 1]\}$.
- It is useful to compare Pareto fronts.
- A Pareto front P_1 dominates or is equal to a Pareto front P_2 iff $\forall v \in P_2, \exists v' \in P_1$ such that (v' dominates v) or $v'=v$.

AlphaMu

- At Max nodes each possible move returns a Pareto front.
- The overall Pareto front is the union of all the Pareto fronts of the moves.
- The idea is to keep all the possible options for Max, i.e. Max has the choice between all the vectors of the overall Pareto front.

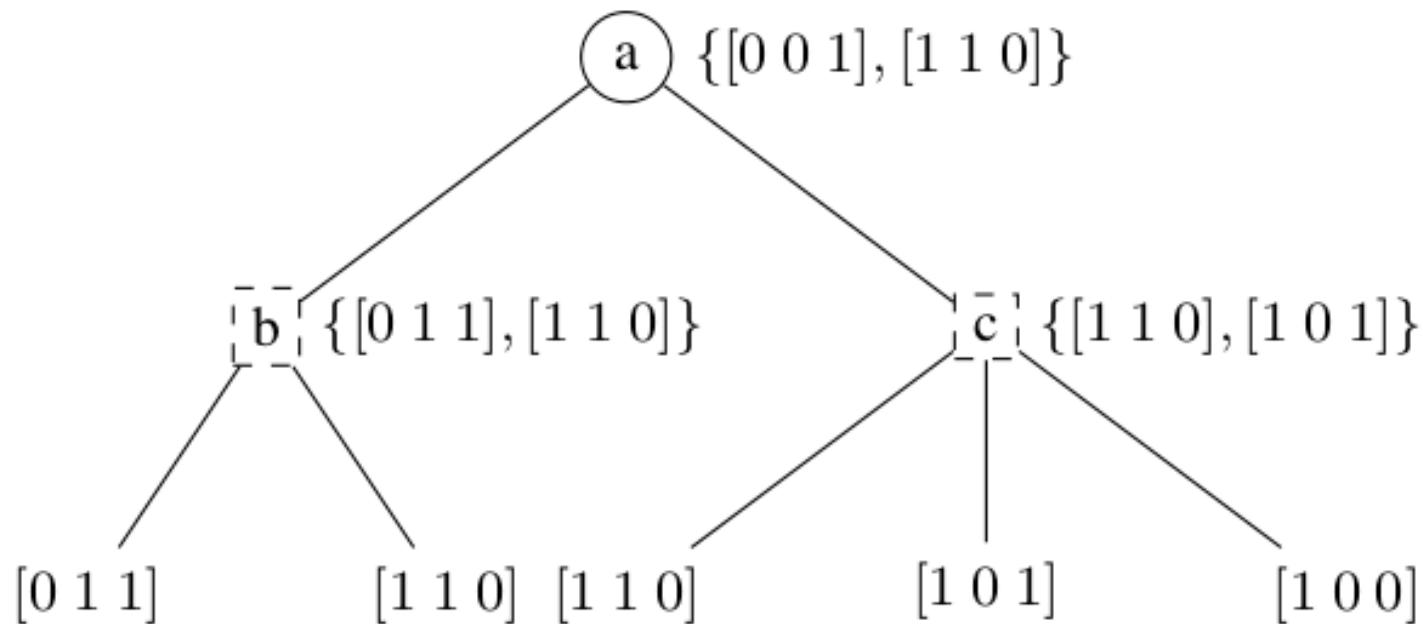
AlphaMu

- At Min nodes, the Min players can choose different moves in different possible worlds.
- They take the minimum outcome over all the possible moves for a possible world.
- When they can choose between two vectors they take for each index the minimum between the two values at this index of the two vectors.

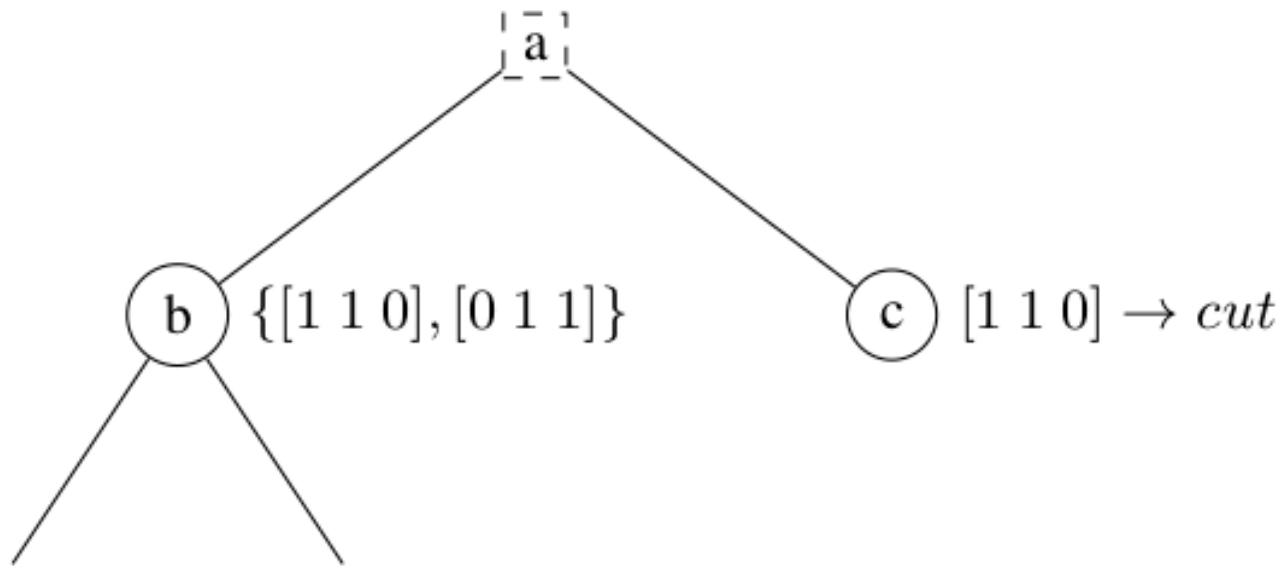
AlphaMu

- When Min moves lead to Pareto fronts, the Max player can choose any member of the Pareto front.
- For two possible moves of Min, the Max player can also choose any combination of a vector in the Pareto front of the first move and of a vector in the Pareto front of the second move.
- Compute all the combinations of the vectors in the Pareto fronts of all the Min moves.
- For each combination the minimum outcome is kept so as to produce a unique vector.
- Then this vector is inserted in the Pareto front of the Min node.

Product of Pareto Fronts at Min nodes



The Early Cut



The Root Cut

- If a move at the root of $\alpha\mu$ for M Max moves gives the same probability of winning than the best move of the previous iteration of iterative deepening for $M-1$ Max moves, the search can safely be stopped since it is not possible to find a better move.
- A deeper search will always return a worse probability than the previous search because of strategy fusion.
- Therefore if the probability is equal to the one of the best move of the previous shallower search the probability cannot be improved and a better move cannot be found so it is safe to cut.

Experimental Results

- Comparison of the average time per move of different configurations of $\alpha\mu$ on deals with 52 cards for the 3NT contract.

Cards	M	Worlds	TT	R	E	Time
52	1	20				0.118
52	2	20	n	n	n	1.054
52	2	20	y	y	n	0.512
52	2	20	y	n	y	0.503
52	2	20	y	y	y	0.433
52	3	20	n	n	n	10.276
52	3	20	y	y	n	3.891
52	3	20	y	n	y	1.950
52	3	20	y	y	y	1.176

Experimental Results

- Comparison of $\alpha\mu$ versus PIMC for the 7NT contract, playing 10 000 games.

Cards	M	Worlds \neq results	Winrate	σ
52	2	20	0.643	0.0285
52	3	20	0.673	0.0257
52	4	20	0.679	0.0241
52	2	40	0.630	0.0268
52	3	40	0.637	0.0258
52	4	40	0.655	0.0248

AlphaMu

- AlphaMu solves de strategy fusion and the non locality problems of PIMC up to a given depth.
- It maintains Pareto Fronts in its search tree.
- It improves on PIMC for the 7NT contract of Bridge.

Conclusion

Monte Carlo Search is a simple algorithm that gives state of the art results for multiple problems:

- Games
- Puzzles
- Discovery of formulas
- RNA Inverse Folding
- Snake in the box
- Pancake
- Logistics
- Multiple Sequence Alignment