Monte-Carlo Game Tree Search

Tsan-sheng Hsu

徐讚昇

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu

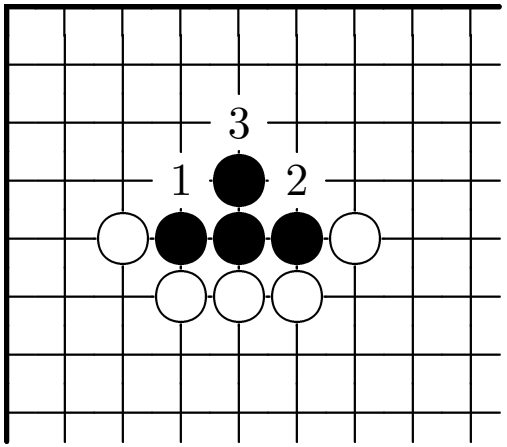
Abstract

- **Introducing the original ideas of using Monte-Carlo simulation in computer Go.**
  - **Sequential Implementation only here.**
  - **From pure Monte-Carlo simulation to a tree based UCT simulation.**
- **Adding new ideas to pure Monte-Carlo approach for computer Go.**
  - **On-line knowledge**
    - ▷ *Progressive pruning*
    - ▷ *All moves as first heuristic*
    - ▷ *Node expansion policy*
    - ▷ *Temperature*
    - ▷ *Depth-2 tree search*
  - **Off-line domain knowledge**
    - ▷ *Node expansion*
    - ▷ *Better simulation policy*
- **Conclusion:**
  - **With the ever-increasing power of computers, we can add more knowledge to the Monte-Carlo approach to get a reasonable solution for computer games.**

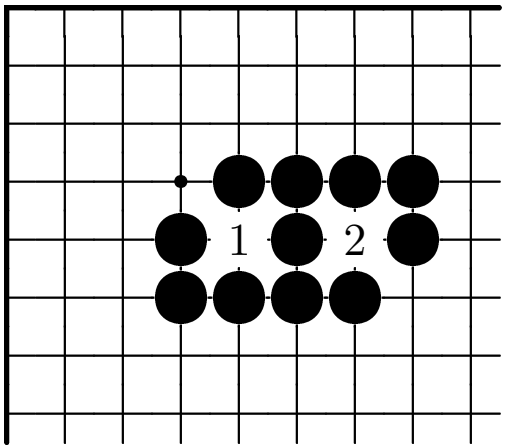
Basics of Go (1/2)

- Black first, a player can pass anytime. It's a draw when both players pass.
- intersection: a cell where a stone can be placed or is placed.
- two intersections are connected: they are either adjacent vertically or horizontally.
- string: a connected, i.e., vertically or horizontally, set of stones of one color.
- liberty: the number of connected empty intersections.
  - Usually we find the amount of liberties for a stone or a string.
  - A string with no liberty is captured.
- eye:
  - Exact definition: very difficult to be understood and implemented.
  - Approximated definition:
    - ▷ *An empty intersection surrounded by stones of one color with two liberties or more.*
    - ▷ *An empty intersection surrounded by stones belonging to the same string.*

Basics of Go (2/2)

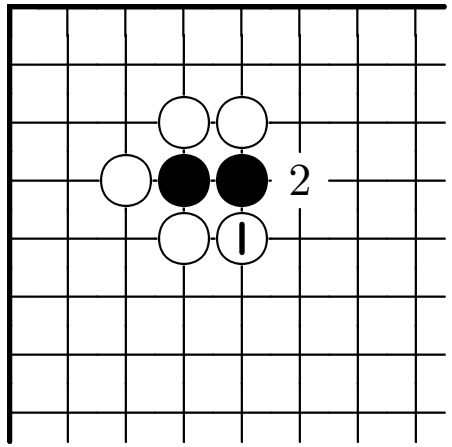


- A black string with 3 liberties.
- A black string with 2 eyes.
  - A string with 2 internal eyes cannot be captured unless you fill in one of the eyes first.



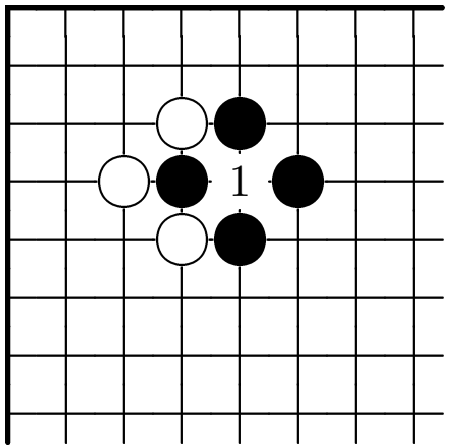
Atari

- **A string with 1 liberty is in danger.**
  - **Placing a white stone at 1 threatens the black string.**
  - **The black string is in danger. The intersection at 2 is now critical.**

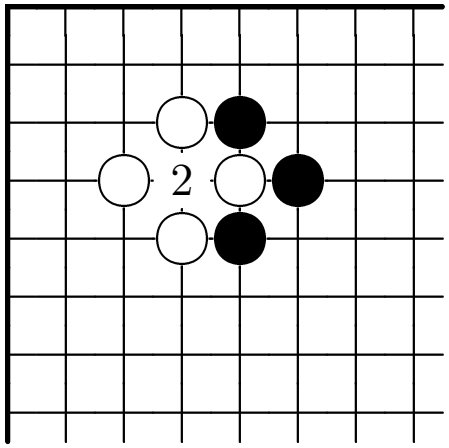


The rule of Ko

- Use the rule of Ko to avoid endless repeated plys.
  - Place a white stone at 1, a black stone is captured.



  - Place a black stone at 2, a white stone is captured.



  - This can go on forever and thus is forbidden.

General rules of Go

- **Black plays first.**
- **A string without liberty is removed.**
- **You cannot place a stone and results in a previous position. after the removing of strings without liberty.**
  - **You cannot create a loop.**
    - ▷ *Note: exact rules for avoiding loops are very complicated and have many different definitions.*
- **You can pass, but cannot play a suicide ply.**
  - **You can place a stone in an intersection without liberty if as a result you can capture opponent's stones.**
- **When both players pass in consecutive plys, the game ends.**
- **The one with more stones and eyes wins at the end of the game.**

Komi

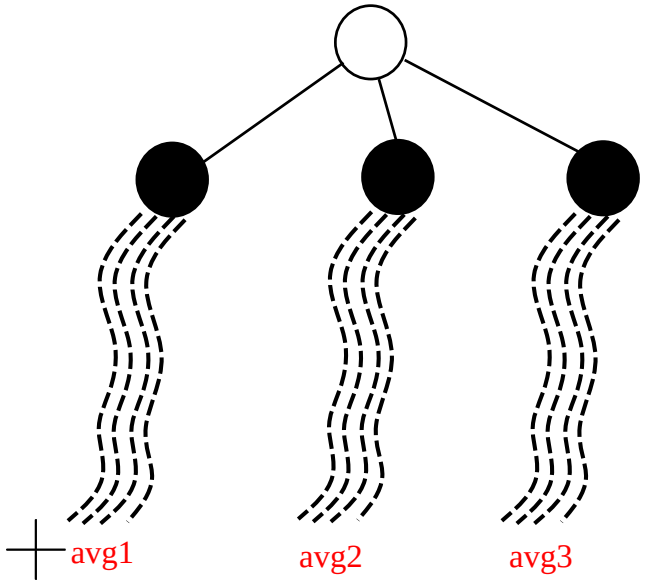
- When calculating the final score, the black side, namely the first player, has a penalty of X stones, which is set by what is called Komi.
  - To offset the initiative.
  - When X is an integer, you can draw a game.
- Go has different very subtle rules which set the value of Komi differently.
  - For 9 by 9 Go, currently it is 7.
  - For 19 by 19 Go, it is either 6.5 or 7.5.

Why Alpha-Beta cut won't work on Go?

- **Alpha-beta based searching has been used since the dawn of CS.**
  - **Effective when a good evaluating function can be computed efficiently.**
  - **Good for games with a not-too-large branching factor, say within 40 and a relative small effective branching factor, say within 5.**
    - ▷ *Effective plys mean those that are not obviously bad plays.*
- **Go has a huge branching and a good evaluating function cannot be easily computed.**
  - **First Go program is probably written by Albert Zobrist around 1968.**
  - **Until 2004, due to a lack of major break through, the performance of computer Go programs is around 5 to 8 kyu for a very long time.**
  - **Need new ideas.**

Monte-Carlo search: original ideas

- **Algorithm MCS$_{pure}$:**
  - **For each possible next move**
    - ▷ ***Play a large number of almost random games from a position to the end, and score them.***
  - **Evaluate a move by computing the average of the scores of the random games in which it had played.**
  - **Play a move with the best score.**



How scores are calculated

- **Score of a game: the difference of the total numbers of stones and eyes for the two sides.**
- **Evaluation of the moves:**
  - **Moves are considered independent of positions.**
  - **Moves were evaluated according to the average scores of the games in which they were played, not only at the beginning but at every stage of the games provided that it was the first time one player had played at the intersection.**

How almost random games are played

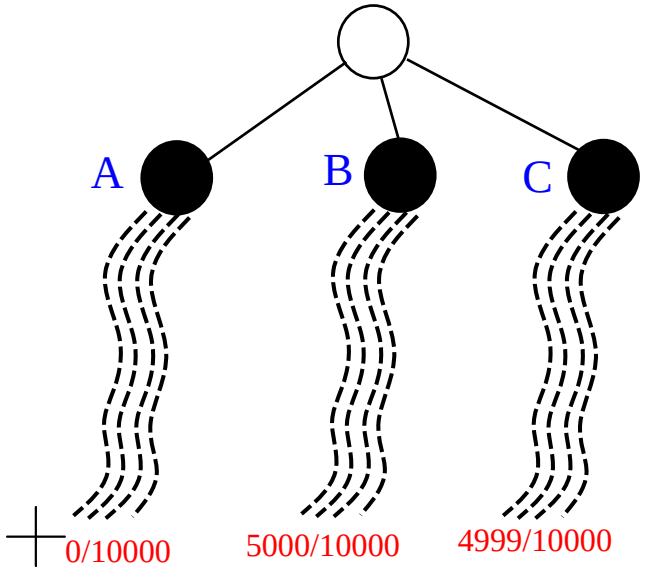
- **No filling of the eyes when doing a random game.**
  - **The only domain-dependent knowledge used in the original version of GOBBLE in 1993.**
- **Moves are ordered according to their current scores.**
- **Ideas from "simulating annealing" were used to control the probability that a move could be played out of order.**
  - **The amount of randomness put in the games was controlled by the controlled by the temperature.**
    - ▷ ***The temperature was set high in the beginning, and then graduately decreased.***
    - ▷ ***For example, the amount of randomness can be a random value drawn from the interval*** $[-v(N)\cdot e^{-c\cdot t(N)}, v(N)\cdot e^{-c\cdot t(N)}]$ ***where*** $v(N)$ ***is the value at the*** N***th iteration,*** c ***is a constant and*** $t(N) = N$ ***is the temperature at the*** N***th iteration.***
    - ▷ ***Simulating annealing is not required, but was used in the original 1993 version.***

Results

- **Original version: GOBBLE 1993.**
  - **Performance is not good compared to other Go programs.**
- **Enhanced versions**
  - **Adding the ideas of new scoring function and a minimax tree search.**
  - **Adding more domain knowledge.**
  - **Adding more techniques.**
    - ▷ *Much more than what are discussed here.*
  - **Building theoretical foundations from statistics, and on-line and off-line learning.**
- **Recent results**
  - **MoGo**
    - ▷ *Won CO champion of the* $19 * 19$ *version in 2007.*
    - ▷ *Beat a professional human 8 dan with a 8-stone handicap at January 2008.*
    - ▷ *Judged to be in a "professional" level for* $9 * 9$ *Go in 2009.*
  - **Zen:**
    - ▷ *Is close to amateur 3-dan in 2011.*
    - ▷ *Beat a 9-dan professional master with handicaps at March 17, 2012.*
      First game: Five stone handicap and won by 11 points.
      Second game: four stones handicap and won by 20 points.

Problems of MCS_{pure}

- **May spend too much time on hopeless branches.**
  - **In the example below, after some trials on A, it can be concluded that this branch is hopeless and this time can be spent on B and C to tell their difference which is currently too close to call.**



† **4999/10000 means winning 4,999 times out of 10,000 simulations.**

First major refinement

- **Efficient sampling:**
  - **Original: equally distributed among all legal moves.**
  - **Biased sampling: sample some moves more often than others.**
- **Observations:**
  - **Some moves are bad and do not need further exploring.**
  - **Should spend some time to verify whether a move that is current good will remain good or not.**
  - **Need to have a mechanism for moves that are bad because of extremely bad luck to have a chance to be reconsidered later.**

Better playout allocation

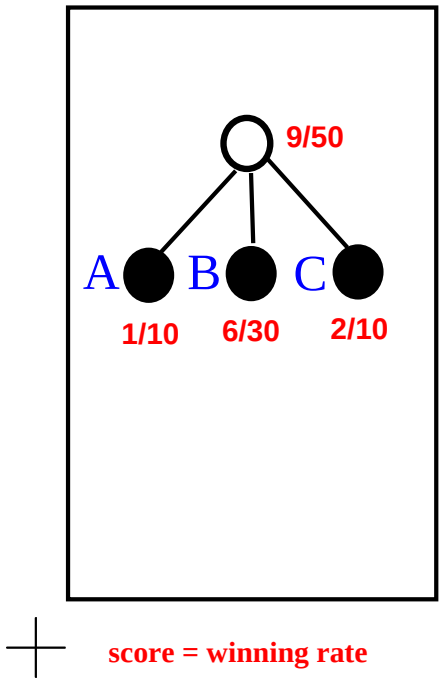
- **K-arm bandit problem:**
  - **Assume you have K slot machines each with a different payoff, i.e., expected value of returns μ_i, and an unknown distribution.**
  - **Assume you can bet on the machines N times, what is the best strategy to get the largest returns?**
- **Ideas:**
  - **Try each machine a few, but enough, times and record their returns.**
  - **For the machines that currently have the best returns, play more often later on.**
  - **For the machines that currently return poorly, give them a chance from time to time just in case their distributions are bad for the runs you tried.**

UCB

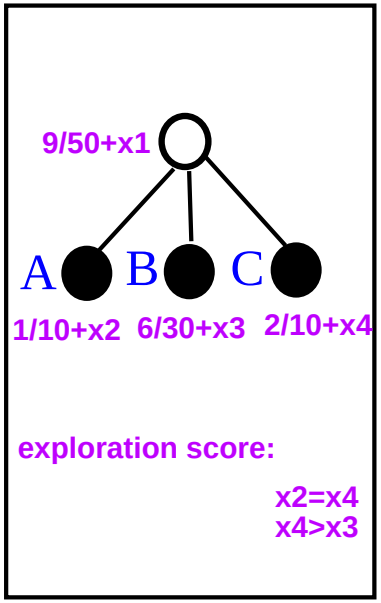
- **UCB: Upper Confidence Bound [Auer et al 2002]**
  - **For each child M_i of a parent node v, compute its**
    $$\text{UCB}_i = \frac{W_i}{N_i} + c\sqrt{\frac{\log N}{N_i}} \text{ where}$$
    - ▷ *W_i is the number of win's for move M_i,*
    - ▷ *N_i is the total number of games played M_i,*
    - ▷ *N is the total number of games played on v, and*
    - ▷ *c is a constant called exploration parameter which controls how often a slightly bad move be tried.*
  - **Expand a new simulated game for the move with the highest UCB value.**
- **Note:**
  - **We only compare UCB scores among children of a node.**
  - **It is meaningless to compare scores of nodes that are not siblings.**
- **Using c to keep a balance between**
  - **Exploitation: exploring the best move so far.**
  - **Exploration: exploring other moves to see if they can be proved to be better.**

Illustration: using UCB scores

- **Using winning rate, B and C are tied.**
- **Using UCB scores, C is better than B because C obtained the score using less trials.**



score = winning rate



UCB score

Other formulas for UCB

- **Other formulas are available from the statistic domain.**
  - **Ease of computing**
  - **Better statistical behaviors**
    - ▷ ***For example, consider the variance of scores in each branch.***
- **Example: consider the games are either win (1) or lose (0), and there is no draw.**
  - **Then $\mu_i = W_i/N_i$ is the expected value of the playouts simulated from this position.**
  - **Let σ_i^2 be the variance of the playouts simulated from this position.**
  - **Define $V_i = \sigma_i^2 + c_1\sqrt{\frac{\log N}{N_i}}$ where c_1 is a constant to be decided by experiments.**
  - **A revised UCB formula is**

$$\mu_i + c\sqrt{\frac{\log N}{N_i}\min\{V_i, c_2\}},$$

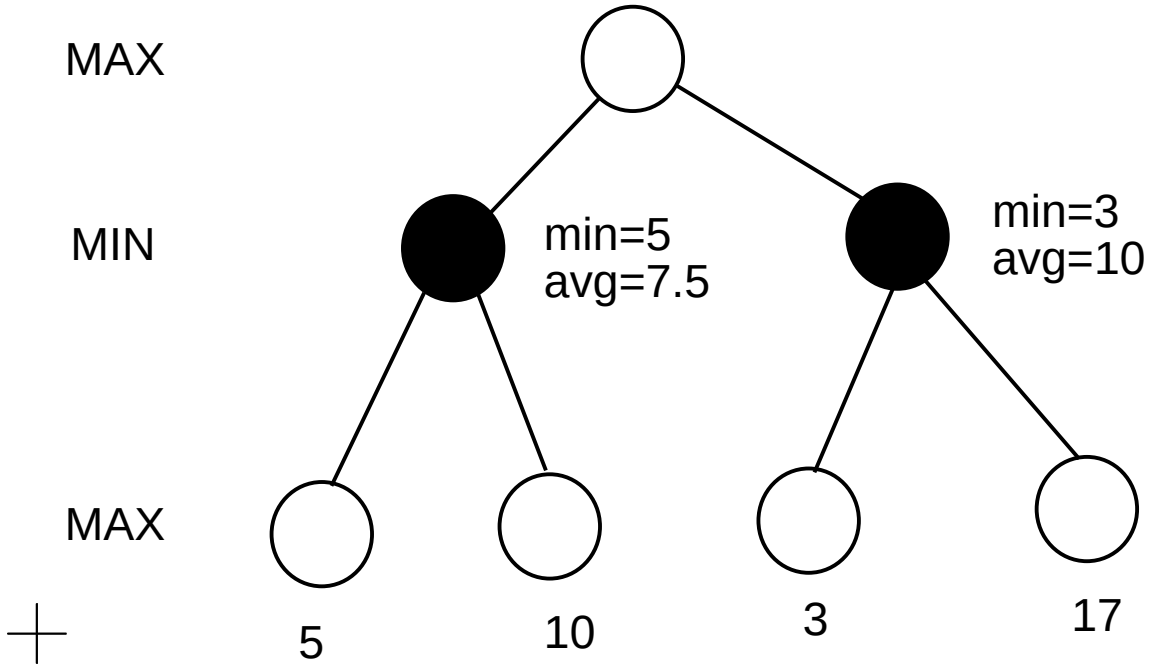
where c and c_2 are both constants to be decided by experiments [Auer et al 2002].

Monte-Carlo search using UCB scores

- **Algorithm MCS$_{UCB}$:**
  - **Generate all possible child positions $r_1, r_2, \ldots, r_b$ of the current position**
  - **Perform x almost random simulations for each child**
  - **Calculate the UCB scores for each child**
  - **While there is still time do**
    - ***Pick a child r_i with the largest UCB score***
    - ***Perform y almost random simulations for r_i***
    - ***Update the UCB score of r_i***
  - **Pick the child with the largest winning rate to play**
- **The values of x and y are parameters.**

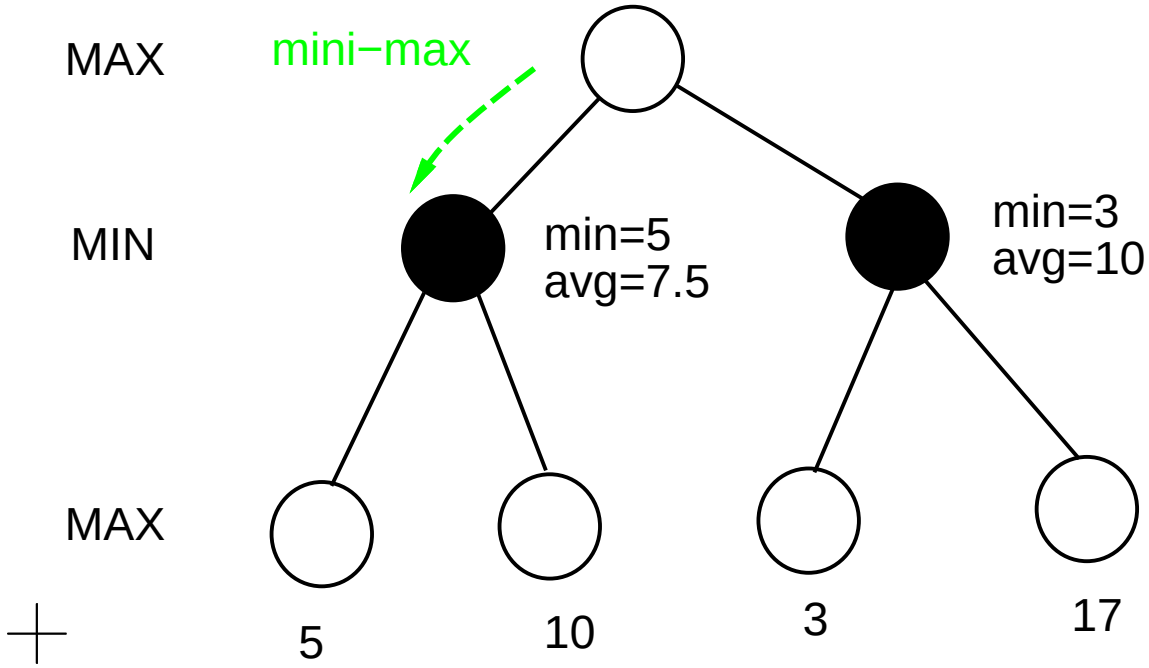
More problems of MCS_{pure}

- **The average score of a branch sometimes does not capture the essential idea of a minimax tree search.**



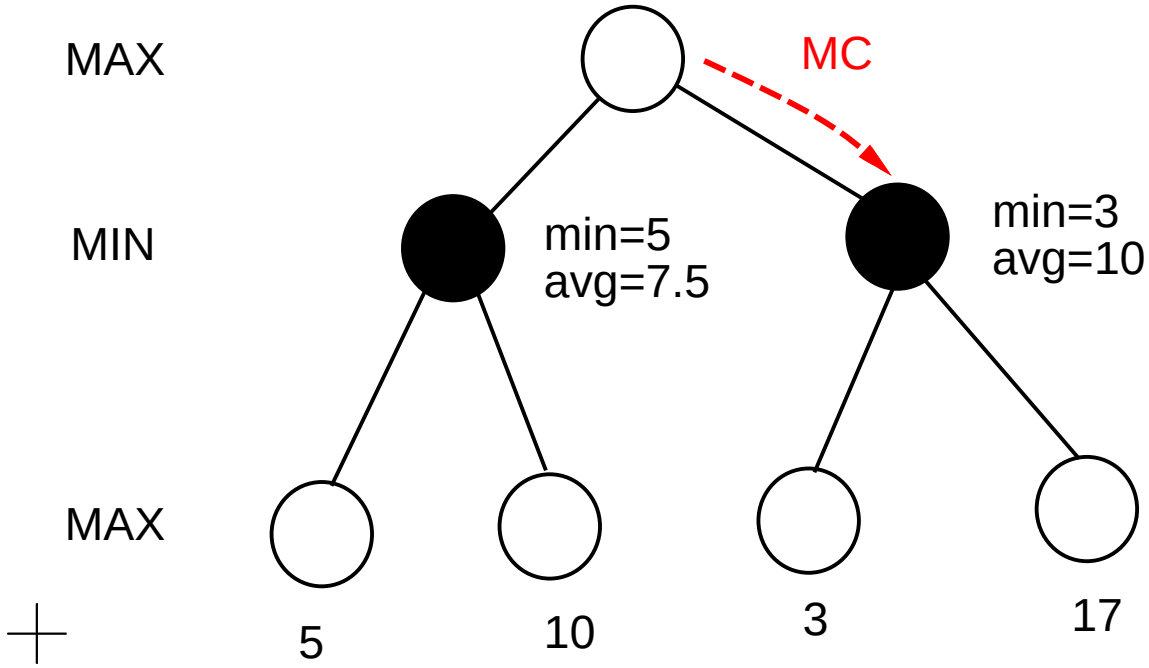
Problems of MCS_{pure}

- **The average score of a branch sometimes does not capture the essential idea of a minimax tree search.**



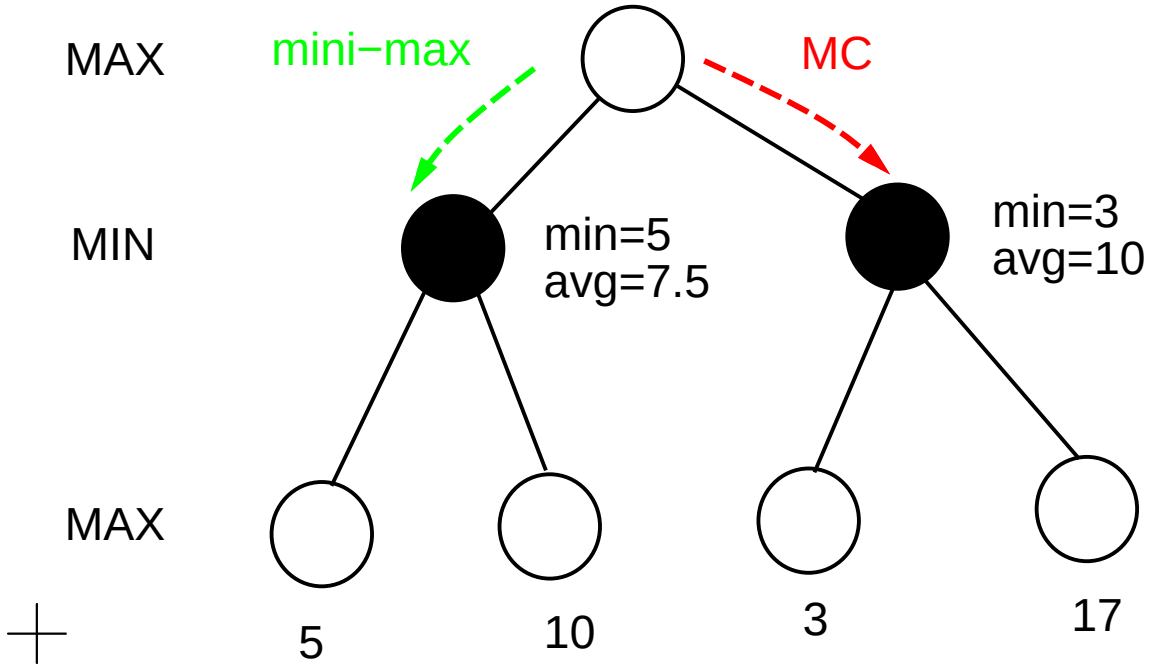
More problems of MCS_{pure}

- **The average score of a branch sometimes does not capture the essential idea of a minimax tree search.**



More problems of MCS_{pure}

- **The average score of a branch sometimes does not capture the essential idea of a minimax tree search.**



- **May spend too much time on hopeless branches.**

Second major refinement

- **Intuition:**
  - **Initially, obtain some candidates for current possible choices that are needed to be further investigated.**
  - **Perform some simulations on the leaf at the PV branch.**
    - ▷ ***A PV path is a path from the root so that each node in this path has a largest score among all of its siblings.***
  - **Update the scores of nodes in the current tree using a mini-max formula.**
  - **Grow a best leaf at the PV one level.**
  - **Repeat the above process until time runs out.**
- **Best first tree growing**

Comment

- **In finding the PV path in a Monte-Carlo tree,**
  - **We do this by a top-down fashion.**
  - **From the root, which is a max node, pick a child p_1 with the largest possible score and then go one step down.**
  - **From p_1, which is a MIN node, pick a child with the smallest score p_2 and then go one more step.**
  - **We keep on doing this until we reach a leaf.**
- **In updating the scores of node in a Monte-Carlo tree when some more simulations are done in a leaf q,**
  - **we do it by a bottom-up fashion.**
  - **We first update the score of q.**
  - **Then we update the score of q's parent q_1 by merging the newly generated statistics of q with the existing statistics of q_1.**
  - **We keep on doing this until the root is reached.**
  - **This is different from the updating operations done in a minimax tree.**
  - **The reasons to merge, not to replace, are**
    - ▷ ***the value is a winning chance from sampling, not really an actual value obtained from an evaluating function;***
    - ▷ ***after merging you get a statistical value that is more trustful since the sample size is increased;***

Best first tree growing: algorithm

- When the number of simulations done on a node is not enough, the mini-max formula of the scores on the children may not be a good approximation of the true value of the node.
  - For example on a MIN node, if not enough children are probed for enough number of times, then you may miss a very bad branch.
- When the number of simulations done on a node is enough, the mini-max value is a good approximation of the true value of the node.
- Use a formula to take into the consideration of node counts so that it will initially act as returning the mean value and then shift to computing the normal mini-max value [Bouzy 2004], [Coulom 2006], [Chaslot et al 2006].

Monte-Carlo based tree search

- **Algorithm MCTS$_{basic}$: // Monte-Carlo mini-max tree search**
- **1: Obtain an initial game tree**
- **2: Repeat the following sequence N_{total} times**
  - **2.1: Selection**
    - ▷ *From the root, pick one path to a leaf with the best "score" using a mini-max formula.*
  - **2.2: Expansion**
    - ▷ *From the chosen leaf with the best "score", expand it by one level using a good node expansion policy.*
  - **2.3: Simulation**
    - ▷ *For the expanded leaves, perform some trials (playouts).*
  - **2.4: Back propagation**
    - ▷ *Update the "scores" for nodes from the selected leaves to the root using a good back propagation policy.*
- **Pick a child of the root with the current best winning rate as your move.**

Illustration: Tree growing

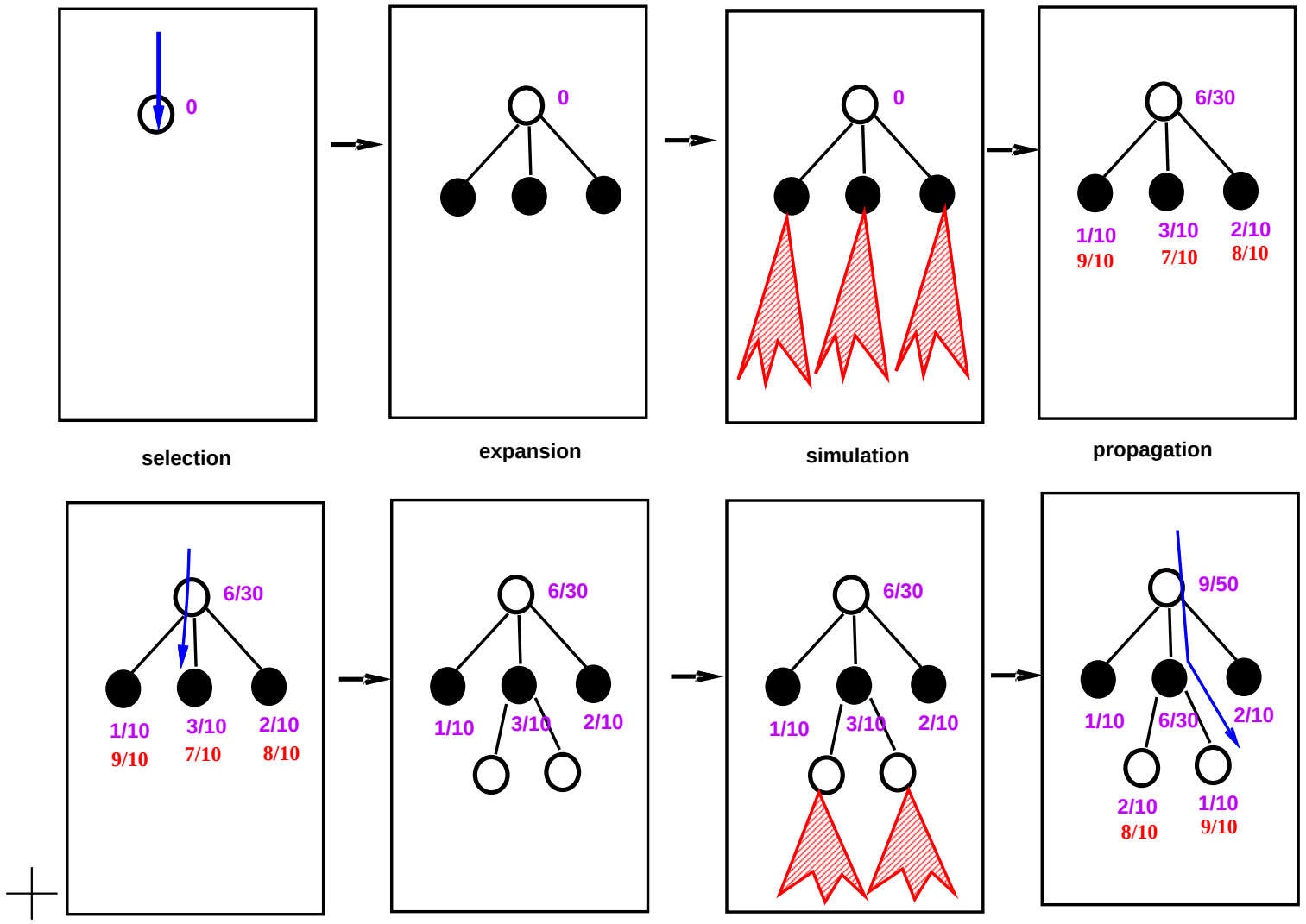
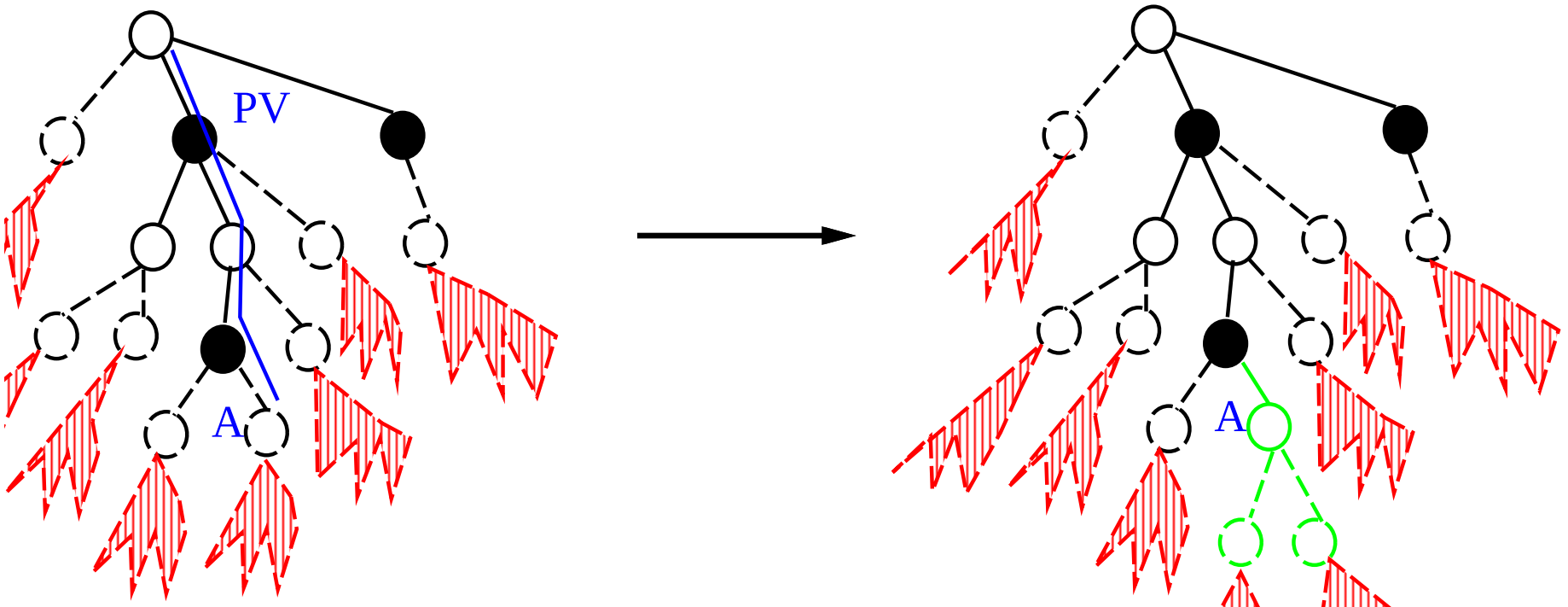


Illustration: Best first tree growing



simulations
and then pick the best leaf to expand

Grow the best leaf

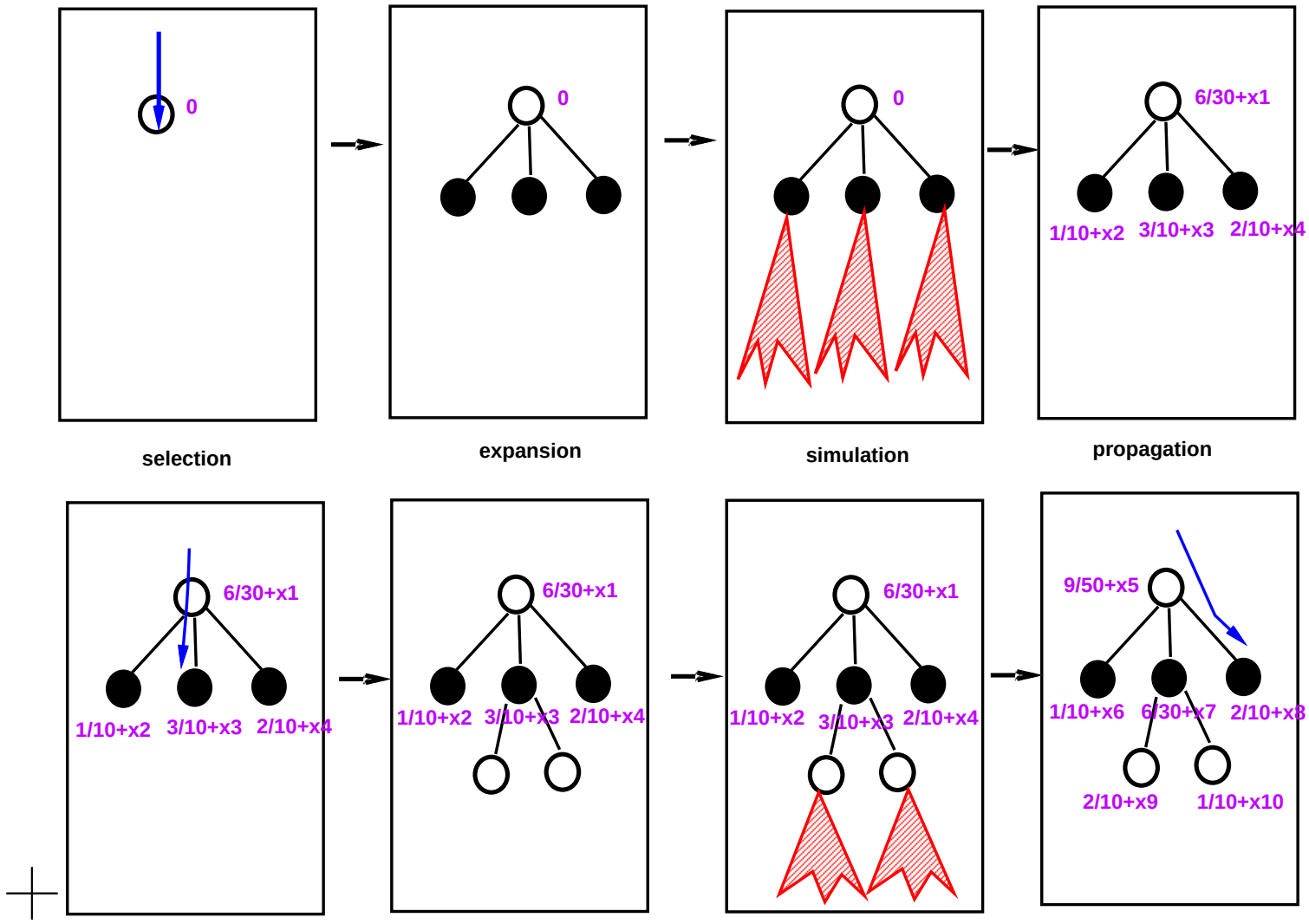
UCT

- **UCT: Upper Confidence Bound for Tree**
  - **Maintain the UCB value for each node in the game tree that is visited so far.**
  - **Best first tree growing:**
    - ▷ ***From the root, pick a PV path such that each node in this path has a largest UCB score among all of its siblings.***
    - ▷ ***Pick the leaf-node in the PV path and has been visited more than a certain amount of times to expand.***
- **UCT approximates mini-max tree search with cuts on proven worst portion of trees.**
- **Usable when the "density of goals" is sufficiently large.**
  - **When there is only a unique goal, Monte-Carlo based simulation may not be useful.**
  - **The "density" and distribution of the goals may be something to consider when picking the threshold for the number of playouts.**

MCTS with UCT

- **Algorithm MCTS:**
- **1: Obtain an initial game tree**
- **2: Repeat the following sequence N_{total} times**
  - **2.1: Selection**
    - ▷ *From the root, pick a PV path to a leaf such that each node has best UCB score among its siblings.*
    - ▷ *May decide to "trust" the score of a node if it is visited more than a threshold number of times.*
    - ▷ *May decide to "prune" a node if its raw score is too bad to save time.*
  - **2.2: Expansion**
    - ▷ *From a leaf with the best UCB score, expand it by one level.*
    - ▷ *Use some node expansion policy to expand.*
  - **2.3: Simulation**
    - ▷ *For the expanded leaves, perform some trials (playouts).*
    - ▷ *May decide to add knowledge into the trials.*
  - **2.4: Back propagation**
    - ▷ *Update the UCB scores for nodes using a good back propagation policy.*
- **Pick a child of the root with the best winning rate as your move.**

Tree growing using UCB scores



Comments about the UCB value

- **For each node M_i, its UCB$_i$ $= \frac{W_i}{N_i} + c\sqrt{\frac{\log N}{N_i}}$.**
- **What does "winning rate" mean:**
  - **For a MAX node, W_i is the number of win's for the MAX player.**
  - **For a MIN node, W_i is the number of win's for the MIN player.**
- **When N_i is approaching $\log N$, then UCB$_i$ is nothing but the current winning rate plus a constant.**
  - **When N is very large, then the current winning rate is approaching the real winning rate for this node.**
  - **If you walk down the tree from the root along the path with the largest UCB values, then it is like walking down the PV.**

Domain independent refinements

- **Main considerations**
  - **Avoid doing un-needed computations**
  - **Increase the speed of convergence**
  - **Avoid early mis-judgement**
  - **Avoid extreme bad cases**
- **Refinements came from on-line knowledge.**
  - **Progressive pruning.**
    - ▷ ***Cut hopeless nodes early.***
  - **All moves at first.**
    - ▷ ***Increase the speed of convergence.***
  - **Node expansion policy.**
    - ▷ ***Grow only nodes with a potential.***
  - **Temperature.**
    - ▷ ***Introduce randomness.***
  - **Depth-i enhancement.**
    - ▷ ***With regard to Line 1, the initial phase, exhaustively enumerate all possibilities.***

Progressive pruning (1/5)

- **Each move has a mean value m and a standard deviation σ.**
  - **Left expected outcome $m_l = m - r_d \cdot \sigma$.**
  - **Right expected outcome $m_r = m + r_d \cdot \sigma$.**
  - **r_d is a ratio fixed up by practical experiments.**
- **A move M_1 is *statistically inferior* to another move M_2 if $M_1.m_r < M_2.m_l$, and $M_1.\sigma < \sigma_e$ and $M_2.\sigma < \sigma_e$.**
  - **σ_e is called *standard deviation for equality*.**
  - **Its value is determined by experiments.**
- **Two moves M_1 and M_2 are *statistically equal* if $M_1.\sigma < \sigma_e$, $M_2.\sigma < \sigma_e$ and no move is statistically inferior to the other.**
- **Remarks:**
  - **We only compare nodes that are of the same parent.**
  - **We usually compare their raw scores not their UCB values.**
  - **If you use UCB scores, then the mean and standard deviation of a move are those calculated only from its un-pruned children.**

Progressive pruning (2/5)

- **After a minimal number of random games, say 100 per move, a move is pruned as soon as it is statistically inferior to another.**
  - **For a pruned move:**
    - ▷ *Not considered as a legal move.*
    - ▷ *No need to maintain its UCB information.*
  - **This process is stopped when**
    - ▷ *this is the only one move left for its parent, or*
    - ▷ *the moves left are statistically equal, or*
    - ▷ *a maximal threshold, say 10,000 multiplied by the number of legal moves, of iterations is reached.*
- **Two different pruning rules.**
  - **Hard: a pruned move cannot be a candidate later on.**
  - **Soft: a move pruned at a given time can be a candidate later on if its value is no longer statistically inferior to a currently active move.**
    - ▷ *The score of an active move may be decreased when more simulations are performed.*
    - ▷ *Periodically check whether to reactive it.*

Progressive pruning (3/5)

- **Experimental setup:**
  - **9 by 9 Go.**
  - **Difference of stones plus eyes after Komi is applied.**
  - **The experiment is terminated if either one of the followings is true.**
    - ▷ *There is only move left for the root.*
    - ▷ *All moves left for the root are statistically equal.*
    - ▷ *A given number of simulations are performed.*

Progressive pruning (4/5)

- **Selection of r_d.**
  - **The greater r_d is,**
    - ▷ ***the less pruned the moves are;***
    - ▷ ***the better the algorithm performs;***
    - ▷ ***the slower the play is.***
  - **Results [Bouzy et al 2004]:**

r_d	1	2	4	8
score	0	+ 5.6	+ 7.3	+9.0
time	10'	35'	90'	150'

- **Selection of σ_e.**
  - **The smaller σ_e is,**
    - ▷ ***the fewer equalities there are;***
    - ▷ ***the better the algorithm performs;***
    - ▷ ***the slower the play is.***
  - **Results [Bouzy et al 2004]:**

σ_e	0.2	0.5	1
score	0	-0.7	-6.7
time	10'	9'	7'

- **Conclusions:**
  - **r_d plays an important role in the move pruning process.**
  - **σ_e is less sensitive.**

Progressive pruning (5/5)

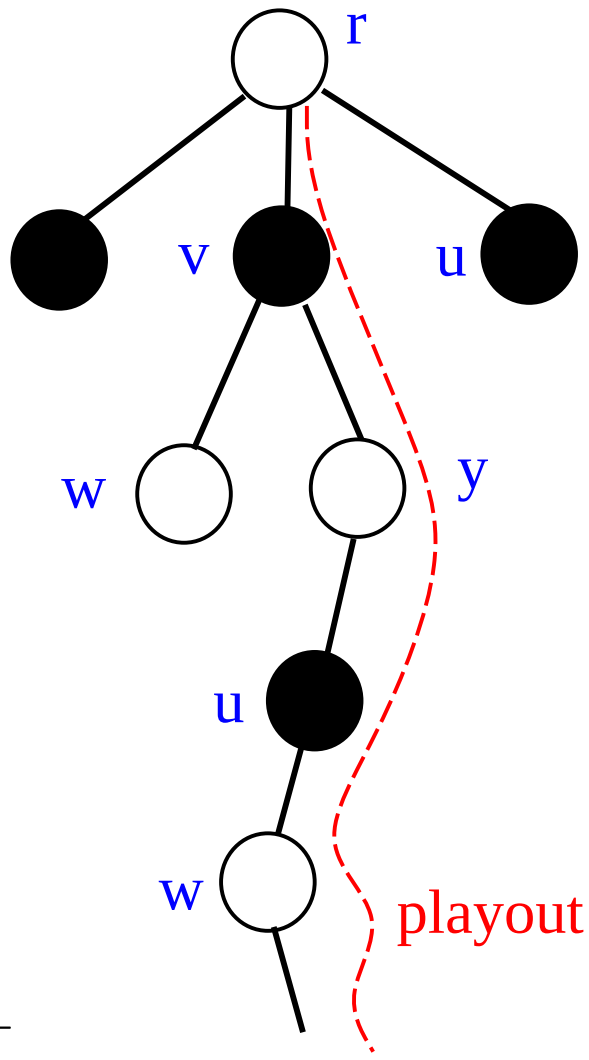
- **Comments:**
  - **It makes little sense to compare nodes that are of different depths or belonged to different players.**
  - **Another trick that may need consideration is progressive widening or progressive un-pruning.**
    - ▷ *A node is effective if enough simulations are done on it and its values are good.*
  - **Note that we can set a threshold on whether to expand or grow the end of the PV path.**
    - ▷ *This threshold can be enough simulations are done and/or the score is good enough.*
    - ▷ *Use this threshold to control the way the underline tree is expanded.*
    - ▷ *If this threshold is high, then it will not expand any node and looks like the original version.*
    - ▷ *If this threshold is low, then we may make not enough simulations for each node in the underline tree.*

All-moves-as-first heuristic (AMAF)

- **How to perform statistics for a completed random game?**
  - **Basic idea: its score is used for the first move of the game only.**
  - **All-moves-as-first AMAF: its score is used for all moves played in the game as if they were the first to be played.**
- **AMAF Updating rules:**
  - **If a playout P, starting from the root following PV towards the best leaf and then appending a simulation run, passes through a node v with a sibling node u, then**
    - ▷ ***the counters at v is updated;***
    - ▷ ***the counters at u is also updated if P later contains the node u.***
  - **Note, we apply this update rule for all nodes in P regardless nodes made by the player that is different from the root player.**

Illustration: AMAF

- **Assume a playout is simulated with the sequence of r, v, y, u, w, $\cdots$.**
- **The statistics of nodes along this path are updated.**
- **The statistics of node u that is a child of r, and node w that is a child of v are also updated.**



AMAF: Pro's and Con's

- **Advantage:**
  - **All-moves-as-first helps speeding up the experiments.**
- **Drawbacks:**
  - **The evaluation of a move from a random game in which it was played at a late stage is less reliable than when it is played at an early stage.**
  - **Recapturing.**
    - ▷ *Order of moves is important for certain games;*
    - ▷ *Modification: if several moves are played at the same place because of captures, modify the statistics only for the player who played first.*
  - **Some move is good only for one player.**
    - ▷ *It does not evaluate the value of an intersection for the player to move, but rather the difference between the values of the intersections when it is played by one player or the other.*

AMAF: results

- **Results [Bouzy et al 2004]:**
  - Basic idea is very slow: 2 hours vs 5 minutes.
  - Relative scores between different heuristics.

AMAF	basic idea	PP
0	+13.7	+ 4.0

  - Number of random games N: relative scores with different values of N using AMAF.

N	1000	10000	100000
scores	-12.7	0	+3.2

    - ▷ *Using the value of 10000 is better.*

- **Comments:**
  - The statistical natural is something very similar to the history heuristic as used in alpha-beta based searching.

AMAF refinement – RAVE

- **Definitions:**
  - Let $v_1(m)$ be the score of a move m without using AMAF.
  - Let $v_2(m)$ be the score of a move m with AMAF.
- **Observations:**
  - $v_1(m)$ is good when sufficient number of trials are performed starting with m.
  - $v_2(m)$ is a good guess for the true score of the move m when
    - ▷ *it is approaching the end of a game;*
    - ▷ *when too few trials are performed starting with m such as when the node for m is first expanded.*
- **Rapid Action Value Estimate (RAVE)**
  - Let revised score $v_3(m) = \alpha \cdot v_1(m) + (1 - \alpha) \cdot v_2(m)$ with a properly chosen value of α.
  - Other formulas for mixing the two scores exist.
  - Can dynamically change α as the game goes.
    - ▷ *For example: $\alpha = \min\{1, N_m/10000\}$, where N_m is the number of playouts done on m.*
    - ▷ *This means when N_m reaches 10000, then no RAVE is used.*
- **Works out better than setting $\alpha = 0$, i.e., pure AMAF.**

RAVE

- **Some other forms of formula for using the RAVE values are know.**
- **Silver in his 2009 Ph.D. thesis.**
  - **Let $\beta = 1 - \alpha$.**
  - **Let $\tilde{N}_m = N_m + N'_m$ where N_m is the number of simulations done at a move m and N'_m is the number of simulations from AMAF at m.**
  - **$\beta = \frac{\tilde{N}_m}{N_m + \tilde{N}_m + 4b^2 N_m \tilde{N}_m}$ where b is a constant to be decided empirically.**
- **Discussion:**
  - $\beta = \frac{1}{\frac{N_m}{\tilde{N}_m} + 1 + 4b^2 N_m}$
  - **We know $\tilde{N}_m \geq N_m$, hence $\frac{1}{2+4b^2 N_m} \leq \beta \leq \frac{1}{1+4b^2 N_m}$.**
  - **During updating, when N'_m increases a lot due to AMAF being applied on many of its children, then β becomes larger.**

Node expansion

- **May decide to expand potentially good nodes judging from the current statistics [Yajima et al 2011].**
  - **All ends: expand all possible children of a newly added node.**
  - **Visit count: delay the expansion of a node until it is visited a certain number of times.**
  - **Transition probability: delay the expansion of a node until its "score" or estimated visit count is high comparing to its siblings.**
    - ▷ *Use the current value, variance and parent's value to derive a good estimation using statistical methods.*
- **Expansion policy with some transition probability is much better than the "all ends" or "pure visit count" policy.**

Temperature

- **Constant temperature: consider all the legal moves and play one of them with a probability proportional to $exp(K \cdot v)$, where**
  - **K is the temperature, and**
  - **v is the current value.**
- **Results [Bouzy et al 2004]:**

K	0	2	5	10	20
score	-8.1	0	+2.6	-4.9	-11.3

- **Simulated annealing:**
  - **increases K from 0 to 5 over time does not enhance the performance.**

Depth-i enhancement

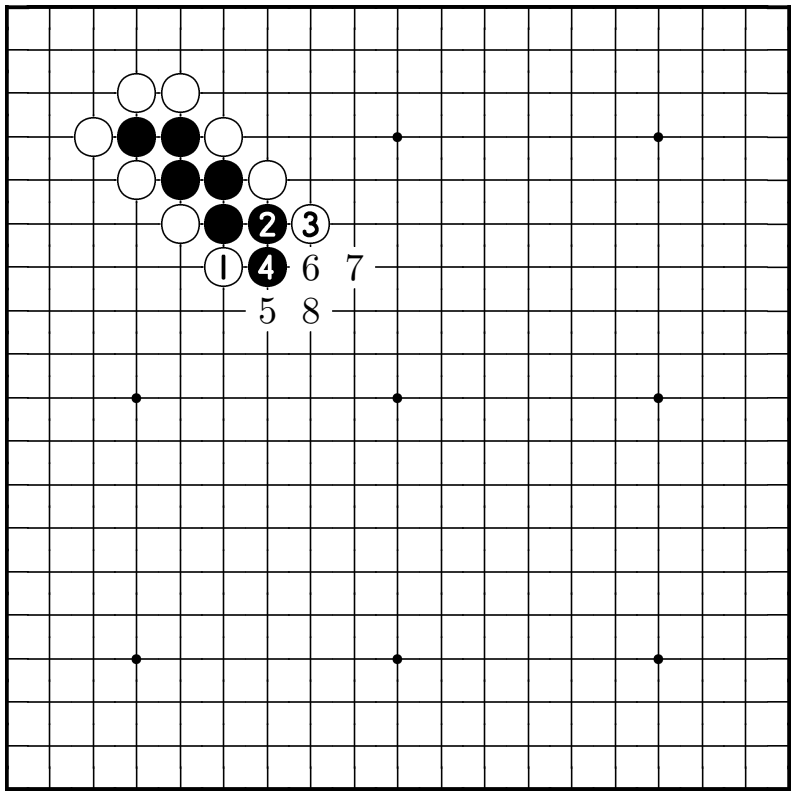
- **Algorithm:**
  - **Enumerate all possible positions from the root after i moves are made.**
  - **For each position, use Monte-Carlo simulation to get an average score.**
  - **Use a minimax formula to compute the best move from the average scores on the leaves.**
- **Result [Bouzy et al 2004]: depth-2 is worse than depth-1 due to oscillating behaviors normally observed in iterative deepening.**
  - **Depth-1 overestimates the root's value.**
  - **Depth-2 underestimates the root's value.**
  - **It is computational difficult to get depth-i results for $i > 2$.**

Putting everything together

- **Two versions [Bouzy et al 2004]:**
  - **Depth = 1, $r_d = 1$, $\sigma_e = 0.2$ with PP, and basic idea.**
  - **$K = 2$, no PP, and all-moves-as-first.**
- **Still worse than GnuGo in 2004, a Go program with lots of domain knowledge, by more than 30 points.**
- **Conclusions:**
  - **Add tactical search: for example, ladders.**
  - **Add more domain knowledge besides no filling of eyes: for example, in Atari, simulate extending plys first.**
    - ▷ ***An extending ply is one which can increase the amount of liberty of some strings.***
  - **As the computer goes faster, more domain knowledge can be added.**
  - **Exploring the locality of Go using statistical methods.**

Ladder

- **White to move next at 1, then black at 2, then white at 3, and then black at 4, ...**



Domain dependent refinements

- **Main consideration**
  - **Adding domain knowledge**
- **Refinements came from off-line training**
  - **During the expansion phase:**
    - ▷ *Special case: open game.*
    - ▷ *General case: use domain knowledge to expand only the nodes that are meaningful in the case of the game considered, e.g., Go.*
  - **During the simulation phase: try to find a better simulation policy.**
    - ▷ *Simulation balancing for getting a better playout policy.*
    - ▷ *Other techniques are also known.*

Simulation balancing (SB) — Go

- **Use ideas from data mining:**
  - **Features: all possible, for example $3 * 3$ patterns, i.e., $3^9 = 19,683$ of them [Huang et al. 2011].**
  - **Training set: known game records.**
  - **Try to find a set of weights for the features to maximize the score.**
- **When doing the simulation, use the sum of weighted feature values to select the next move for each player.**
  - **It is easy to have an efficient implementation.**
  - **Can add some amount of randomness in selecting the moves, such as using the idea of temperature.**
- **Results are very good.**
  - **A very good playout policy may not be good for the purpose of finding out the average behavior.**
    - ▷ *The samplings must be consider the average "real" behavior of a player can make.*
    - ▷ *It is extremely unlikely that a player will make trivially bad moves.*
  - **Need to balance the time used in carrying out the policy found and the number of simulations can be computed.**

Important notes

- We only describe some specific implementations of Monte-Carlo techniques.
  - Other implementations exist for say AMAF and others.
- It is important to know the underling "theory" that make a technique useful, not a particular implementation.
- Depending on the amount of resources you have, you can
  - decide the frequency to update the node information,
  - decide the frequency to re-pick PV,
  - decide the frequency to prune/unprune nodes.
- You also need to know the precision and cost of your floating-point number computation which is the core of calculating UCB scores.

Implementation

- **How to partition stones into strings?**
  - Scan the stones one by one.
  - For each unvisited stone
    - ▷ *Do a DFS to find all stones of the same color that are connected.*
  - Can use a good data structure to maintain this information when a stone is placed.
    - ▷ *Example: disjoint union-find*
- **How to know an empty intersection is an eye?**
  - Check its 4 neighbors
  - Each neighbor must be either
    - ▷ *out of board, or*
    - ▷ *it is in the same string with the other neighbors.*
- **How to find out the amount of liberties of a string?**
  - for each empty intersection, check its 4 neighbors:
    - ▷ *it is a liberty of the string where its neighbor is in;*
    - ▷ *make sure an empty intersection contributes at most 1 in counting the amount of liberties of a string;*

Implementation hints (1/3)

- **Each node m_i maintains 3 counters W_i, L_i and D_i, which are the number of games won, lost, and drawn, respectively, for playouts simulated starting from this position.**
  - **Note that $N_i = W_i + L_i + D_i$.**
  - **For ease of coding, the numbers are from the view point of the root, namely MAX, player.**
- **Assume $m_{i,1}, m_{i,2}, \ldots, m_{i,b}$ are the children of m_i.**
  - $W_i = \sum_{j=1}^{b} W_{i,j}$
  - $L_i = \sum_{j=1}^{b} L_{i,j}$
  - $D_i = \sum_{j=1}^{b} D_{i,j}$
- **"Winning rate"**
  - **For a MAX node, it is W_i/N_i.**
  - **For a MIN node, it is L_i/N_i.**

Implementation hints (2/3)

- **Only nodes in the current "partial" tree maintained the 3 counters.**
- **Assume $m_{i,1}, m_{i,2}, \ldots, m_{i,b}$ are the children of m_i that are currently in the "partial" tree.**
  - **It is better to maintain a "default" node representing the information of playouts simulated when m_i was a leaf.**
- **When any counter of a node v is updated, it is important to update the counters of its ancestors.**
- **Need efficient data structures and algorithms to maintain the UCB value of each node.**
  - **When a simulated playout is completed, the UCB scores of all nodes are updated because the total number of playouts, N, is increased by 1.**
    - ▷ ***The winning rate of all v and v's ancestors are also changed.***

Implementation hints (3/3)

- **How to incrementally update mean and variance of a node?**
  - **Assume the results of the simulation form the sequence** $x_1, x_2, x_3, \ldots, x_i, x_{i+1}, x_{i+2}, \ldots$
  - **Let** $Var(n)$ **be the variance of the first** n **elements. Hence** $Var(n) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu(n))^2$ **where** $\mu(n) = \frac{1}{n}\sum_{i=1}^{n} x_i$**.**
  - **In each node, we maintain the following data:**
    - n
    - $sum2(n) = \sum_{i=1}^{n} x_i^2$
      Hence $sum2(n+1) = sum2(n) + x_{n+1}^2$
    - $sum1(n) = \sum_{i=1}^{n} x_i$
      Hence $sum1(n+1) = sum1(n) + x_{n+1}$
  - $\mu(n) = \frac{1}{n} \cdot sum1(n)$
  - $Var(n) = \frac{1}{n} \cdot (sum2(n) - 2 \cdot \mu(n) \cdot sum1(n)) + \mu(n)^2$
- **Note:**
  - **In general, we do not perform a division operator unless it is really needed to do so.**
  - **If the value of a node can only be 0 or 1, then** $sum1(n) = sum2(n)$**.**
  - **If the value of a node can be something else, then** $sum1(n)$ **and** $sum2(n)$ **may be different.**

Comments (1/2)

- Proven to be successful on a few games.
  - Very successful on Go.
- Not very successful on some games.
  - Not currently very good on Chess or Chess-like games.
- Performance becomes better when the game is going to converge.
- Needs a good random playout strategy that can simulate the average behavior of the current position efficiently.
  - On a bad position, do not try to always get the best play.
  - On a good position, try to usually get the best play.
- It is still an art to find out what coefficients to set.
  - Need a theory to efficiently find out the values of the right coefficients.

Comments (2/2)

- **The "reliability" of a Monte-Carlo simulation depends on the number of trials it performs.**
  - **The rate of convergence is important.**
  - **Do enough number of trials, but not too much for the sake of saving computing time.**
- **Adding more knowledge can slow down each simulation trial.**
  - **There should be a tradeoff between the amount of knowledge added and the number of trials performed.**
  - **Similar situation in searching based approach:**
    - ▷ ***How much time should one spent on computing the evaluating function for the leaf nodes?***
    - ▷ ***How much time should one spent on searching deeper?***
- **Knowledge, or patterns, about Go can be computed off-lined by Monte-Carlo methods.**

References and further readings (1/4)

* B. Bruegmann. Monte Carlo Go. unpublished manuscript, 1993.
- B. Bouzy and B. Helmstetter. Monte-Carlo Go developments. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *Advances in Computer Games, Many Games, Many Challenges, 10th International Conference, ACG 2003, Graz, Austria, November 24-27, 2003, Revised Papers*, volume 263 of *IFIP*, pages 159–174. Kluwer, 2004.
* Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning*, ICML '07, pages 273–280, New York, NY, USA, 2007. ACM.
* P. Auer, N. Cesa-Bianchi, P. Fischer. Finite-time analysis of the multiarmed bandit problem. Machine Learning, pages 235–256, 2002.

References and further readings (2/4)

- **Bruno Bouzy. Associating shallow and selective global tree search with Monte Carlo for 9x9 Go. In *Lecture Notes in Computer Science 3846: Proceedings of the 4th International Conference on Computers and Games*, pages 67–80, 2004.**
- **Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Lecture Notes in Computer Science 4630: Proceedings of the 5th International Conference on Computers and Games*, pages 72–83. Springer-Verlag, 2006.**
- **Hugues Juille. *Methods for Statistical Inference: Extending the Evolutionary Computation Paradigm*. PhD thesis, Department of Computer Science, Brandeis University, May 1999.**

References and further readings (3/4)

- David Silver. Reinforcement Learning and Simulation-Based Search in Computer Go. PhD thesis, University of Alberta, 2009.
- Guillaume Chaslot, Jahn Takeshi Saito, Jos W. H. M. Uiterwijk, Bruno Bouzy, and H. Jaap Herik. Monte-Carlo strategies for computer Go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 83–91, Namur, Belgium, 2006.
- Takayuki Yajima, Tsuyoshi Hashimoto, Toshiki Matsui, Junichi Hashimoto, and Kristian Spoerer. Node-expansion operators for the UCT algorithm. In H. Jaap van den Herik, H. Iida, and A. Plaat, editors, *Lecture Notes in Computer Science 6515: Proceedings of the 7th International Conference on Computers and Games*, pages 116–123. Springer-Verlag, New York, NY, 2011.

References and further readings (4/4)

- Shih-Chieh Huang, Rmi Coulom, and Shun-Shii Lin. Monte-Carlo Simulation Balancing in Practice. In H. Jaap van den Herik, H. Iida, and A. Plaat, editors, *Lecture Notes in Computer Science 6515: Proceedings of the 7th International Conference on Computers and Games*, pages 81–92. Springer-Verlag, New York, NY, 2011.

* Browne, Cameron B., et al. "A survey of monte carlo tree search methods." Computational Intelligence and AI in Games, IEEE Transactions on 4.1 (2012): 1-43.