Monte-Carlo Game Tree Search: Basic Techniques

Tsan-sheng Hsu

徐讚昇

tshsu@iis.sinica.edu.tw

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

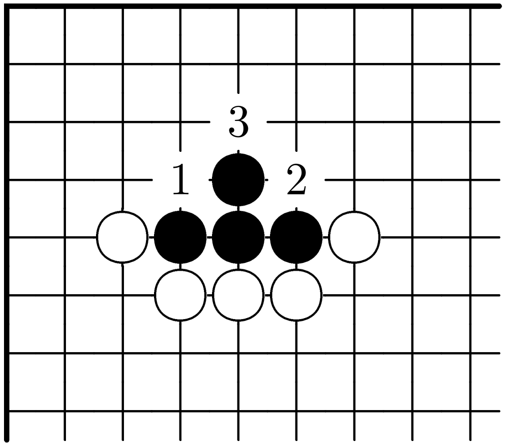
Abstract

- **Introducing the original ideas of using Monte-Carlo simulation in computer Go.**
  - **Pure Monte-Carlo simulation.**
  - **Using UCB scores.**
  - **Incooperate with Mini-Max tree search.**
  - **Using UCT tree expansion.**
    - ▷ ***Best first tree growing.***
- **Only introduce sequential implementation here.**
  - **Parallel implementation will be introduced later.**
- **Conclusion:**
  - **A new search technique that proves to be very useful in solving selective games including computer Go.**

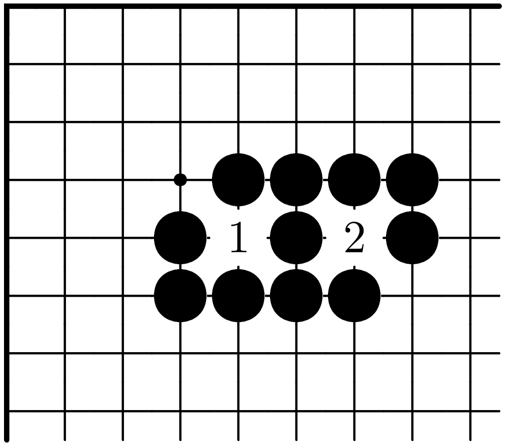
Basics of Go (1/2)

- **Black first, a player can pass anytime.**
- **The game is over when both players pass in consecutive turns.**
- **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)

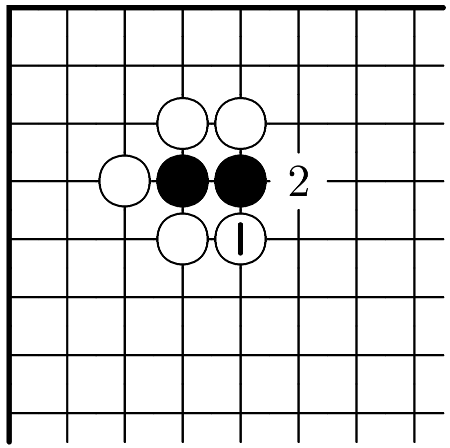


- **A black string with 3 liberties.**
- **A black string with 2 eyes.**
  - **A string with 2 internal eyes cannot be captured by the opponent unless you fill in one of the two eyes yourself first.**



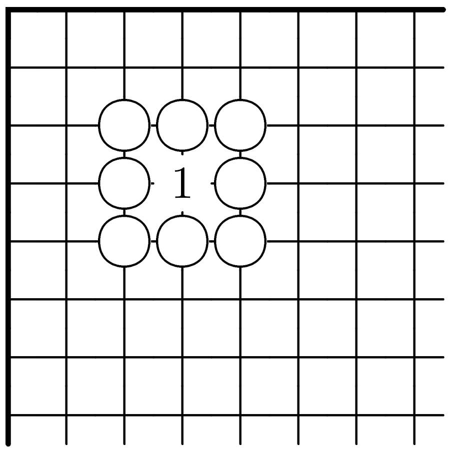
Atari

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



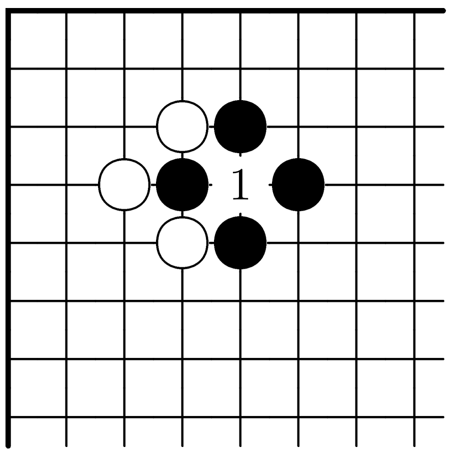
Legal ply

- **Place your stone in an empty intersection and not causing suicide.**
  - **Black cannot place a black stone at the intersection 1.**

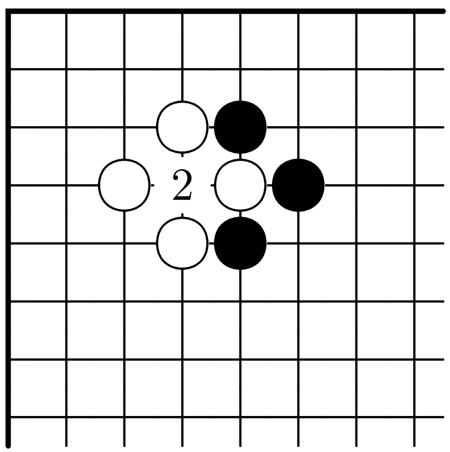


The rule of Ko

- **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.**



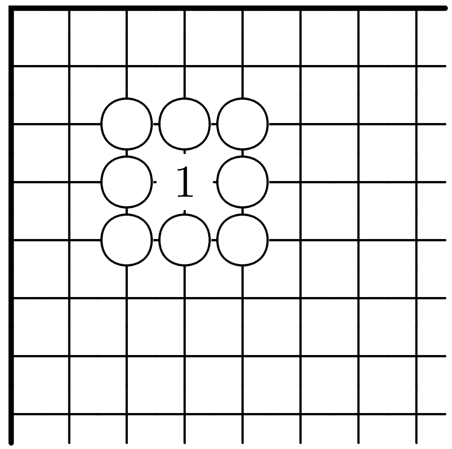
  - **This can go on forever and thus is forbidden (to the black).**

General rules of Go

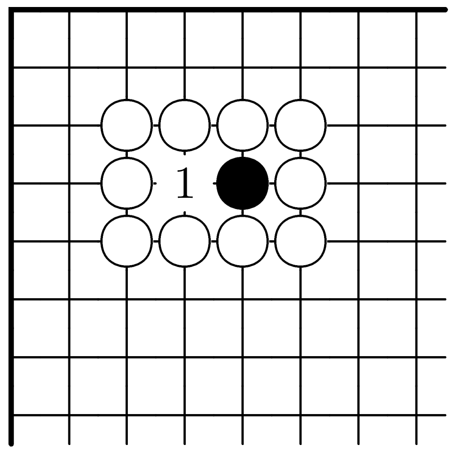
- **Black plays first.**
- **A string without liberty is removed.**
- **You cannot place a stone and results in a position that is 2-plys ago. after the removing of strings without liberty.**
  - **You cannot create a loop.**
    - ▷ *Note: exact rules for avoiding loops are very complicated and have many different definitions.*
- **You can pass, but cannot play a plain suicide ply.**
  - **A suicide ply is one that causes the stone played being removed immediately by itself.**
    - ▷ *You can place a stone to cause more than one of your stones being removed.*
  - **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 after discounting Komi.**

More examples

- **Illegal move at 1 for black.**



- **Legal move at 1 for black.**



Komi

- **When calculating the final score, the black side, namely the first player, has a penalty of K stones, which is set by what is called Komi.**
  - **To offset the initiative.**
  - **When K 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.**
    - ▷ ***It is possible to draw!***
  - **For 19 by 19 Go, it is either 6.5 or 7.5.**
    - ▷ ***No draw!***

Ranking system

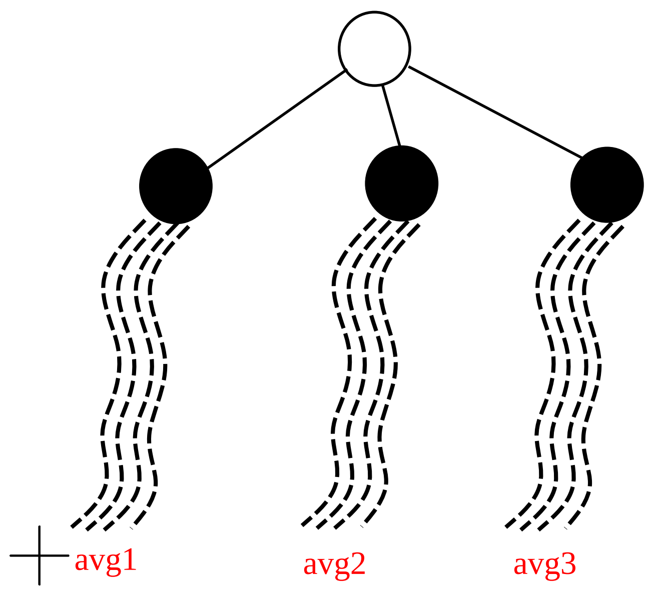
- **Dan-kyu system: from good to bad in the order of**
  - **Professional level: dan.**
    - ▷ ***9, 8, . . ., 2, 1***
  - **Amateur level: dan.**
    - ▷ ***9, 8, . . ., 2, 1***
  - **Kyu.**
    - ▷ ***1, 2, 3, 4, . . .***
- **Elo: assign a numerical score to a player so that the larger the score, the better a player is.**
  - **Usually between 100 to 3000+.**
  - **More details in later lectures.**
  - **Human: www.goratings.org**
    - ▷ ≥ 2940: ***professional 9 dan***
    - ▷ ~ 2820: ***professional 5 dan***
    - ▷ ***Note: human history high is 3692.33 (Nov. 2019; Shin, Jinseo).***
- **A higher ranked player has a better chance of winning, not a sure win, against a lower ranked player.**

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 designed manually by human and computed efficiently by computers.**
    - *Evaluating functions do not need to be designed purely by human any-more.*
    - *One can use machine learning techniques as well.*
    - *Example: the development of GNU Go before 2004 using manually designed heuristics, and the development of Alpha Go after 2016 using deep learning.*
  - **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 designed manually.**
  - **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 child position of a possible next move from the root**
    - ▷ ***Play a large number of almost random games from a position to the end, and score them.***
  - **Evaluate a child position by computing the average of the scores of the random games in which it had played.**
  - **Play a move going to the child position 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 child positions from the possible next moves:**
  - **Child positions are considered independently.**
  - **Child positions 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.**
- **Can use winning rate or non-losing rate as the score.**
  - **For ease of description, we use mostly winning rate in the rest of our slides here.**

How almost random games are played

- **No filling of the eyes when a random game is drawn.**
  - **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 gradually decreased.***
    - ▷ ***For example, the amount of randomness can be a random value drawn from the interval $[-v(i) \cdot e^{-c \cdot t(i)}, v(i) \cdot e^{-c \cdot t(i)}]$ where $v(i)$ is the value at the ith iteration, c is a constant and $t(i)$ is the temperature at the ith iteration.***
    - ▷ ***Simulating annealing is not required, but was used in the original 1993 version.***

Results

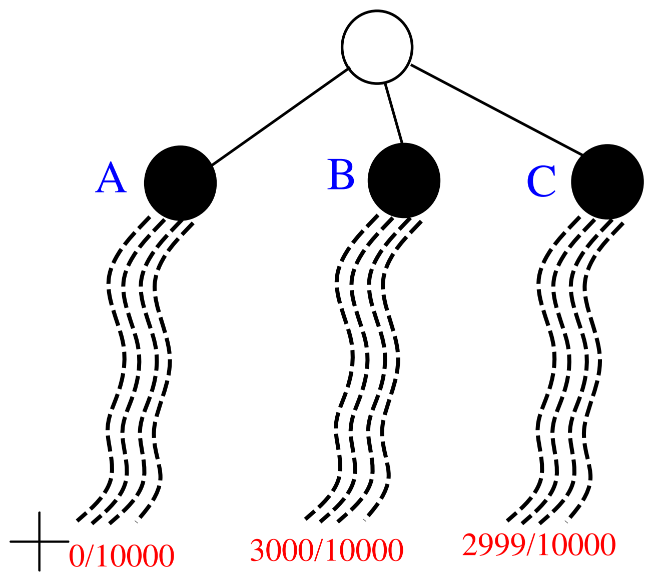
- **Original version: GOBBLE 1993 [Bruegmann'93].**
  - **Performance is not good compared to other Go programs of the same era.**
- **Enhanced versions**
  - **Adding the idea of using new scoring functions.**
  - **Using a mini-max tree search.**
  - **Using a best first tree growing.**
  - **Adding more domain knowledge.**
  - **Adding more techniques.**
    - ▷ ***Much more than what are discussed here.***
    - ▷ ***In practice, works out well when the game is approaching the end or when the state-space complexity is not large.***
  - **Building theoretical foundations from statistics, and on-line and off-line learning.**
  - **Using techniques from deep learning.**

Recent results

- **Recent results**
  - **MoGo (France):**
    - ▷ *Won Computer Olympiad champion of the $19 * 19$ version in 2007.*
    - ▷ *Beat a professional 8 dan with a 8-stone handicap at January 2008.*
    - ▷ *Judged to be in a "professional" level for $9 * 9$ Go in 2009.*
    - ▷ *Very close to professional 1-dan for $19 * 19$ Go.*
  - **Zen (Japan):**
    - ▷ *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.
    - ▷ *Add techniques from machine learning.*
  - **AlphaGo Lee: Beat a professional 9-dan at March 2016 with a record of 4 to 1 !**
    - ▷ *Using deep learning.*
    - ▷ *Elo 3739* $\sim 10dan?$ *[Silver et al 2016]*
  - **AlphaGo Zero: An earlier version beat one of the very top professional players at May 2017 with a record of 3 to 0 !!!**
    - ▷ *Using unsupervised learning.*
    - ▷ *Elo 5185 !!!* $\sim 10 + X dan?$ *[Silver et al 2017]*

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.**
  - **From now on, time should be spent on B and C to tell their difference which is currently too close to call.**



† **2999/10000 means winning 2,999 times out of 10,000 simulations.**

First major refinement

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

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 have tried so far.**

UCB

- **UCB: Upper Confidence Bound [Auer et al'02]**
  - **For each child p_i of a parent node p, 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 the position p_i,*
    - *N_i is the total number of games played p_i,*
    - *$N = \sum N_i$ is the total number of games played on p, and*
    - *c is a positive 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 when later on tree search is incooperated.**

How UCB is derived

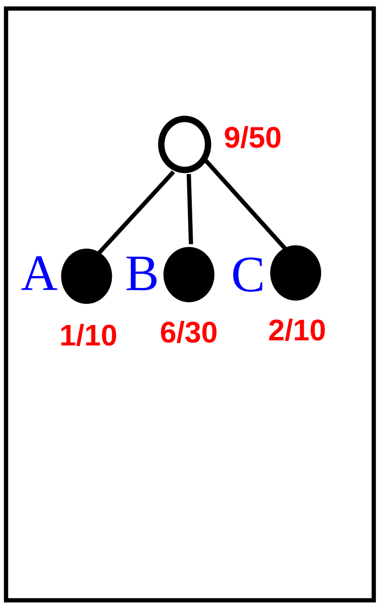
- **Hoeffding inequality [Hoeffding 1963]:**
  - **Assume X is a Bernoulli random variable in the range of $[0,1]$ with an expected value of $E(X)$.**
  - **Let X_i be the ith independent sampling of X.**
  - **Let $\overline{X}_t$ be the average of the first t samplings.**
  - $P(|E(X) - \overline{X}_t| \geq u) \leq e^{-2 \cdot t \cdot u^2}$ **for $u > 0$.**
- **It gives an estimation on the difference between the real value and the observed average value at time t.**
  - **Fixing u, when t increases, the chance of the difference to be $\geq u$ decreases exponentially.**
  - **Fixing the chance of failure, difference to be $\geq u$, to be a small value of N^{-c^2} where $c > 0$. Then $u \leq c\sqrt{\frac{\log N}{t}}$.**
  - **This means the real value has a very small chance to be more than $\overline{X}_t + u$.**
- **In the UCB formula,**
  - **t is N_i and $\overline{X}_t = \frac{W_i}{N_i}$ is the observed value.**
  - $\frac{W_i}{N_i} + c\sqrt{\frac{\log N}{N_i}}$ **is the upper bound of $E(X)$ with a good confidence.**

Exploitation or Exploration

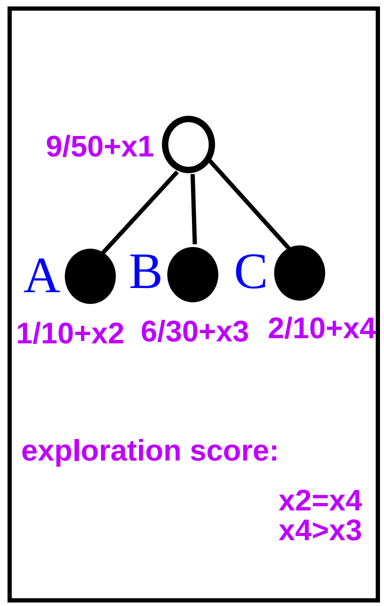
- $$\text{UCB}_i = \frac{W_i}{N_i} + c\sqrt{\frac{\log N}{N_i}}$$
- **Using c to keep a balance between**
  - **Exploitation: exploring the best move so far.**
  - **Exploration: exploring other moves to see if they can be proven to be better.**
- **No N_i should be zero.**
  - **Give each child at least some trials.**
- **The theoretical value for c in [Auer et al'02] is**
  - $\sqrt{2 \cdot \frac{\log 2}{\log e}} \sim 1.18$ **where e is the base of the natural logarithm which is about 2.718.**

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**

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

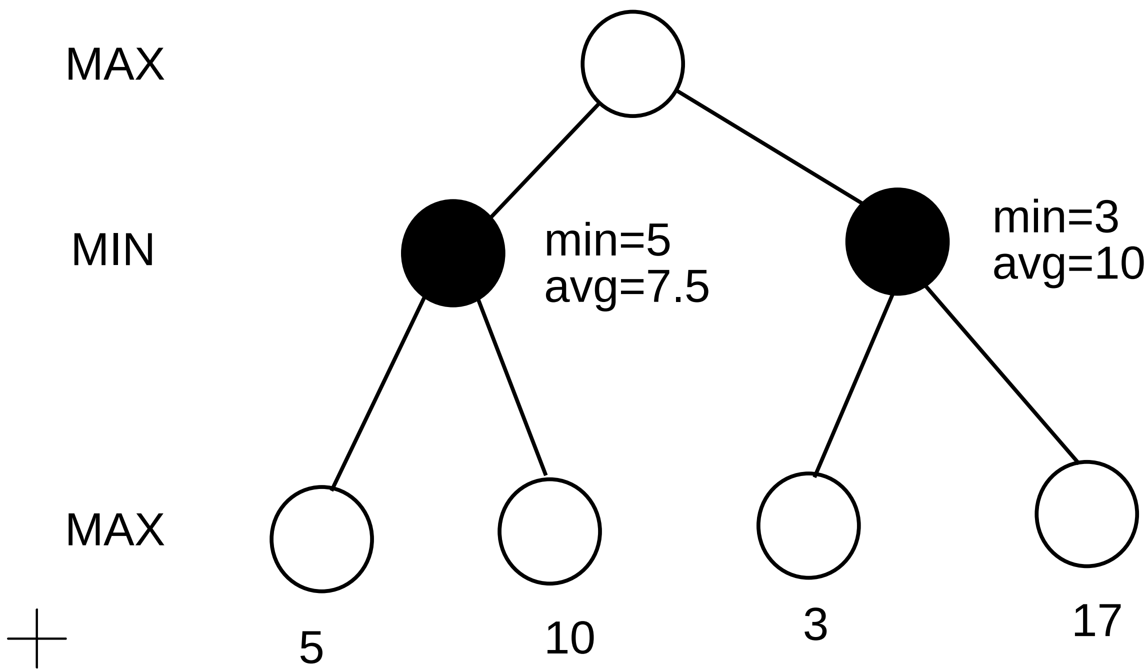
where c and c_2 are both constants to be decided by experiments [Auer et al'02] and c_2 is used to bound the influence of V_i.

Monte-Carlo search using UCB scores

- **Algorithm MCS$_{UCB}$(position p, int x, int y):**
  - **Generate all possible child positions $p_1, p_2, \ldots, p_b$ of the current position p**
  - **for each child p_i do**
    - ▷ ***Perform x almost random simulations for p_i***
    - ▷ ***Calculate the UCB score for p_i***
  - **While there is still time do**
    - ▷ ***Pick a child p^* with the largest UCB score***
    - ▷ ***Perform y almost random simulations for p^****
    - ▷ ***Update the UCB score of p^* as well as other nodes***
  - **Pick a child with the largest winning rate to play**
- **It is usually the case we pick a child with the largest winning rate, not with the largest UCB score to play.**
  - **After enough trials, one with the largest winning rate is usually, but not always, the one with the largest UCB score.**

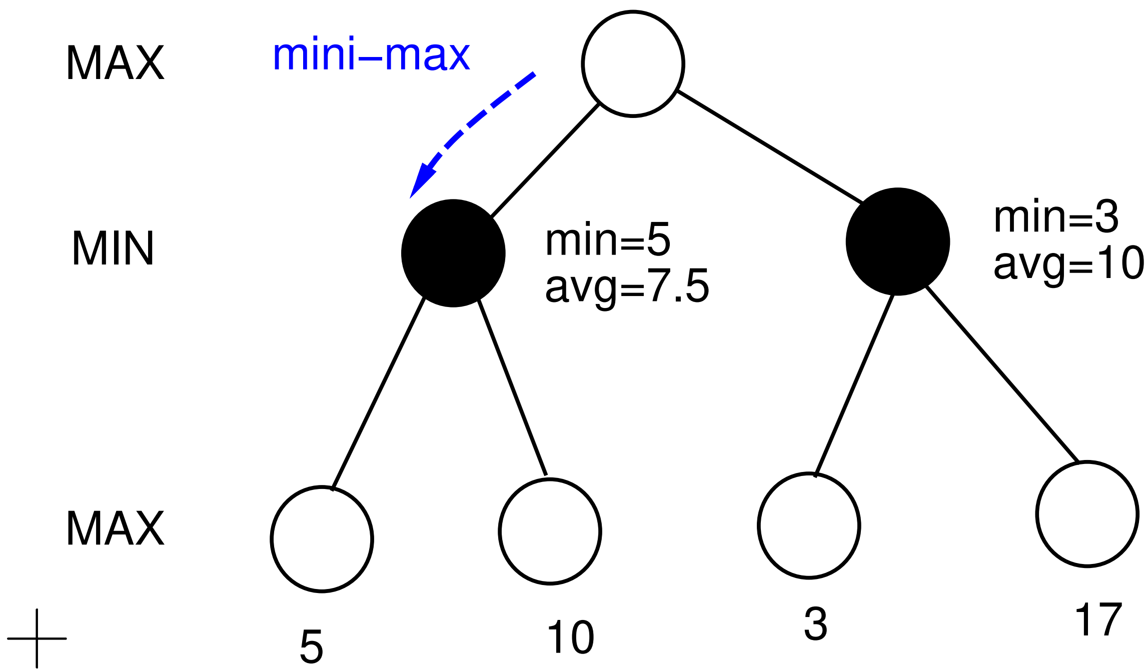
More problem of MCS$_{pure}$

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



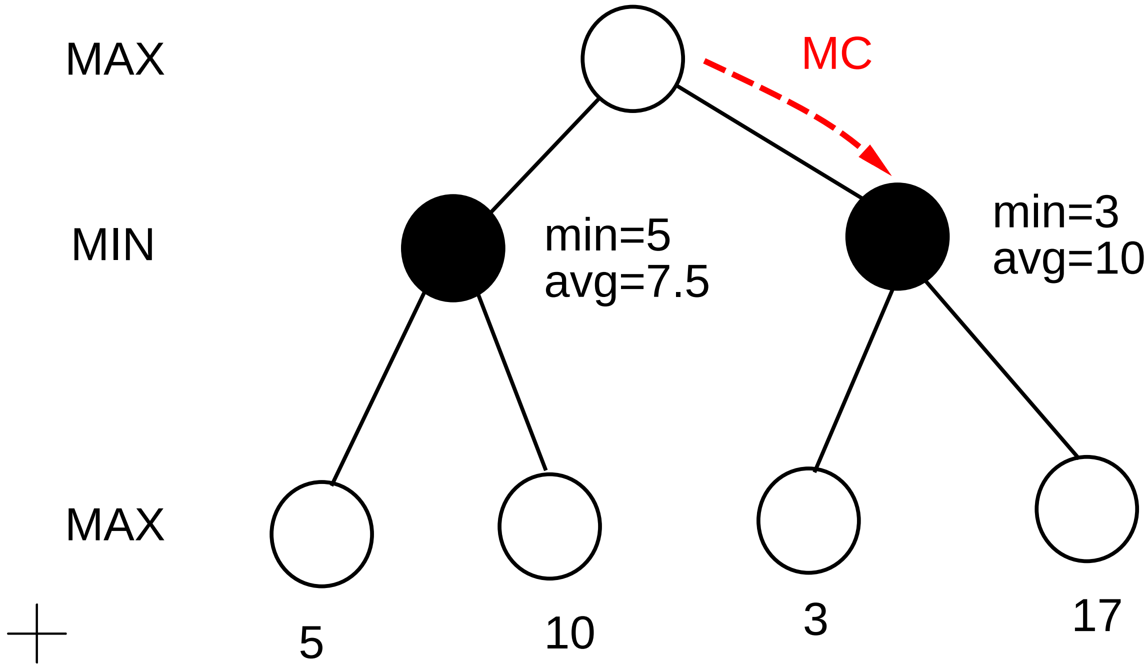
More problem of MCS$_{pure}$

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



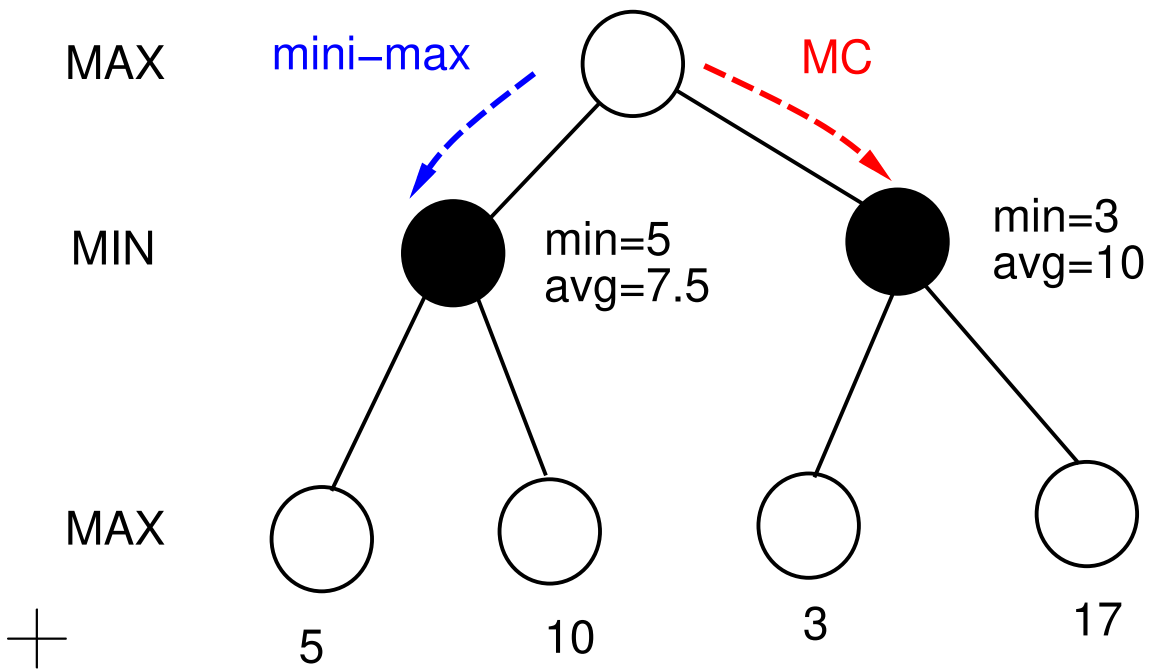
More problem of MCS$_{pure}$

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



More problem of MCS$_{pure}$

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



- **May spend too much time on the wrong branch.**

Second major refinement

- **Intuition:**
  - **Initially, obtain some candidate choices that are needed to be further investigated.**
  - **Perform some simulations on the leaf at a 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.*
    - ▷ *Note: In a mini-max tree, "largest" means different numerical values for min and max nodes.*
  - **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 [Coulum'06].**
  - **Keep a partial game tree and uses the mini-max formula within the partial game tree kept.**
  - **Grow the game tree on demand.**

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

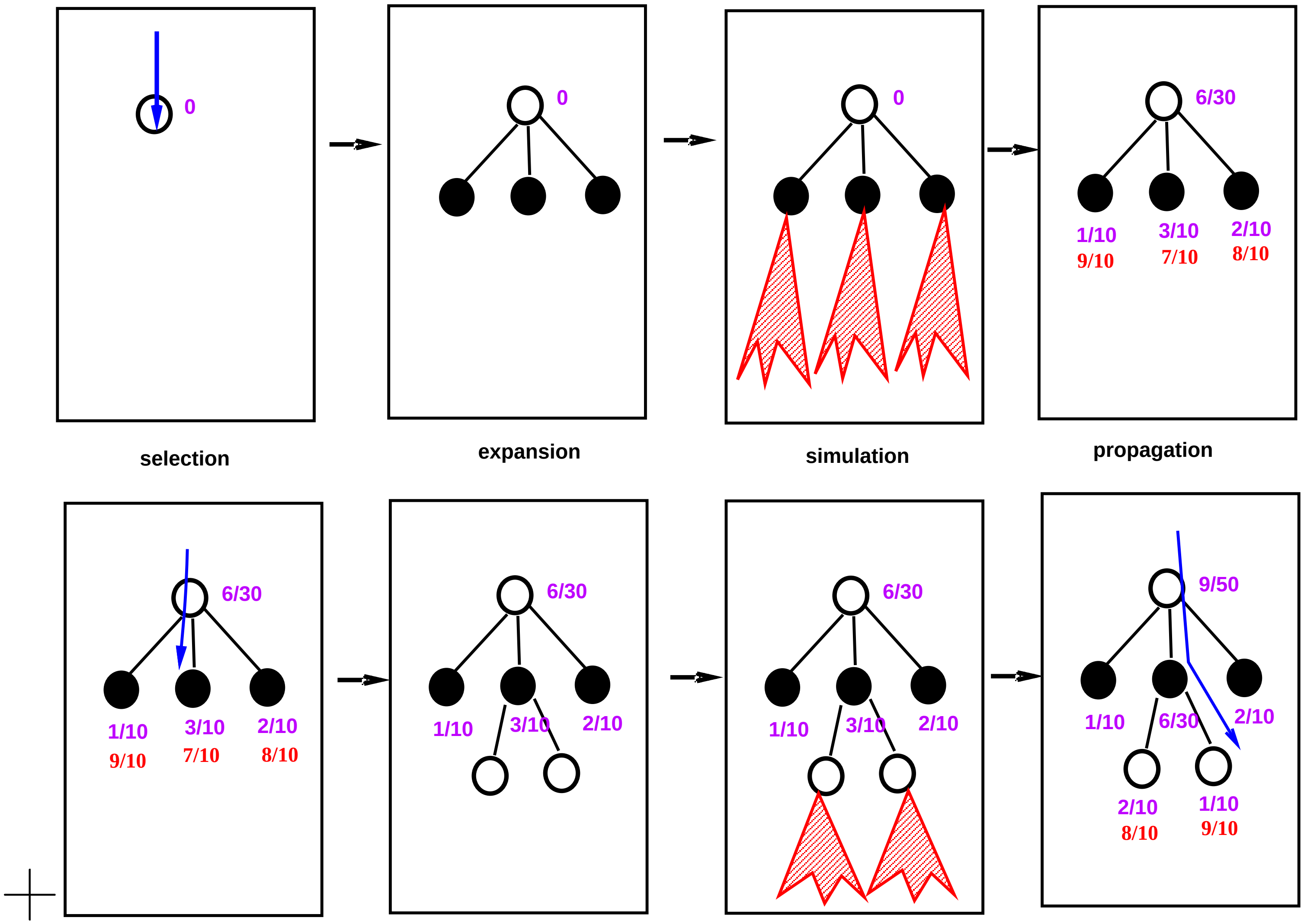
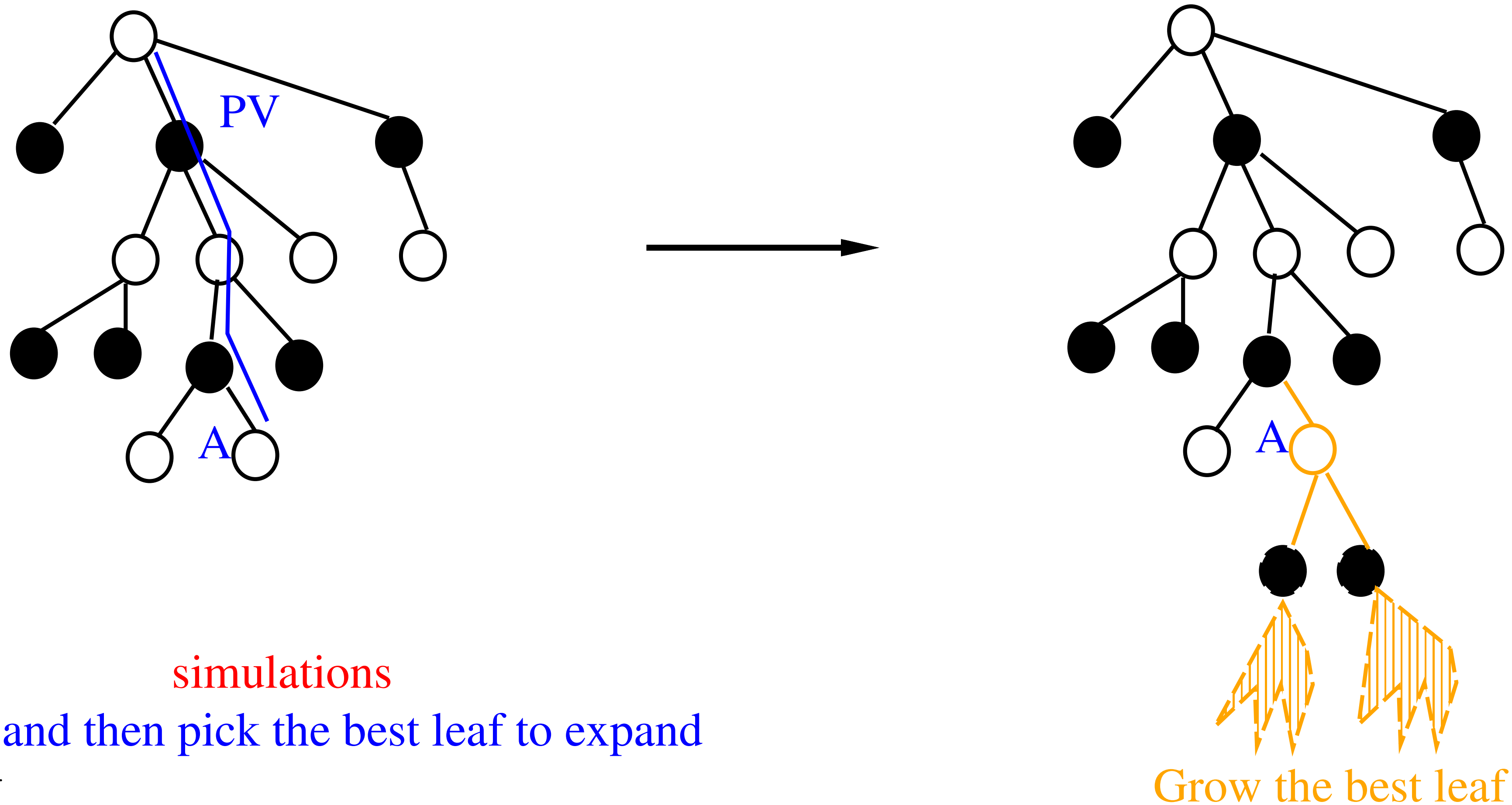


Illustration: Best first tree growing



simulations
and then pick the best leaf to expand

Comments (1/2)

- **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 down.**
  - **We keep on doing this until a leaf is reached.**
- **In updating the scores of nodes 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^* by merging the newly generated statistics of q with the existing statistics of q^*.**
  - **We keep on doing this until the root is reached.**
  - **This is different from the updating operations done in a mini-max 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.*

Comments (2/2)

- **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'04], [Coulom'06], [Chaslot et al'06].**

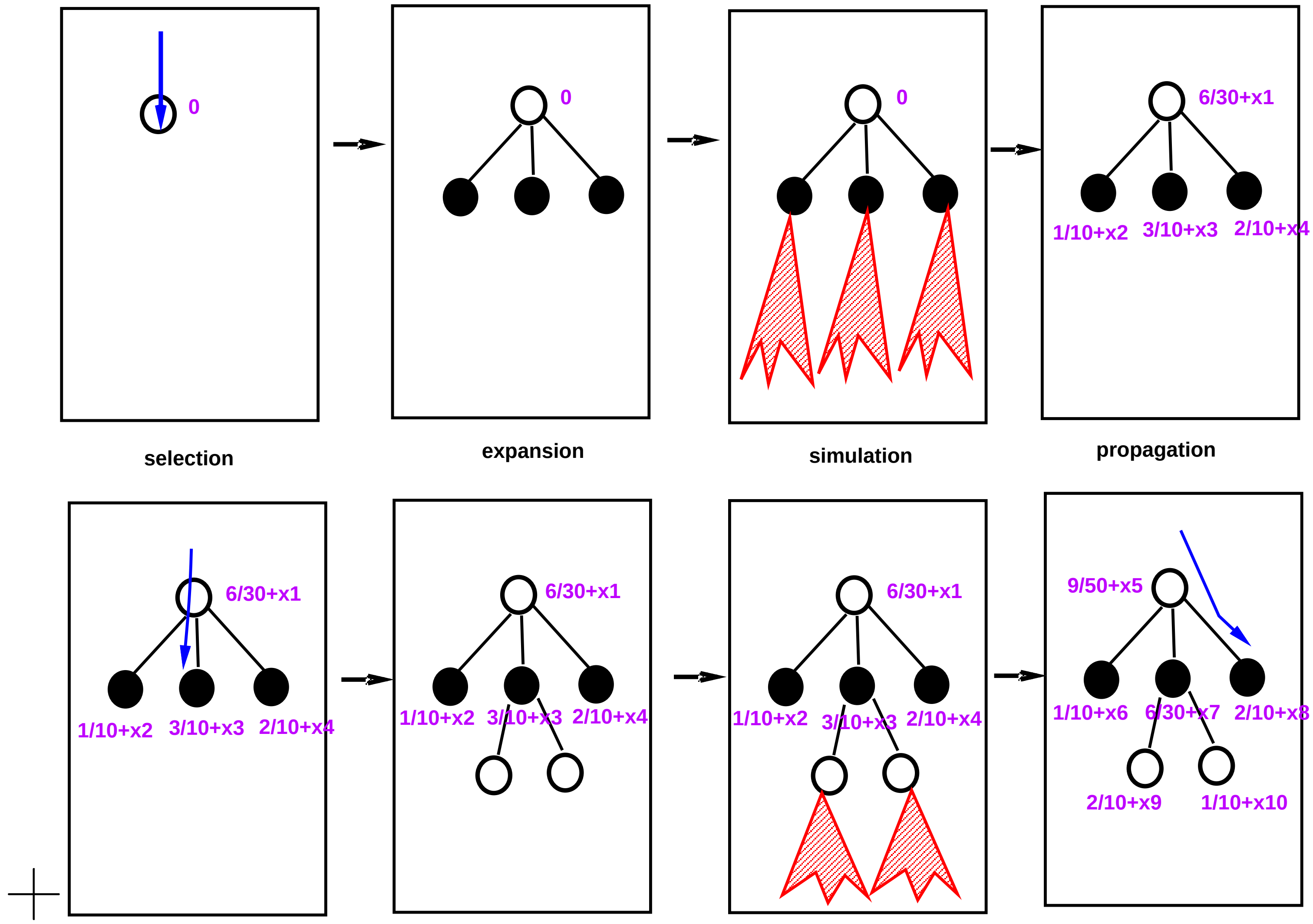
UCT

- **UCT: Upper Confidence Bound for Tree [Chaslot et al '08]**
  - **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_{UCB} 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 efficient.**
  - **The "density" and distribution of the goals may be something to consider when picking the threshold for the number of playouts needed to reach a statistical conclusion.**

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_{UCB} path to a leaf such that each node has a 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 node 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 largest UCB values, namely PV$_{UCB}$, then it is like walking down the traditional mini-max PV.**

Important notes

- **We only describe some specific implementations of Monte-Carlo techniques.**
  - **Other implementations exist for say UCB scores.**
- **It is important to know the underling "theory", not a particular implementation, that makes a technique work.**
- **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,**
- **You also need to know the precision and cost of your floating-point number computation which is the core of calculating UCB scores.**

Implementation for Go

- **How to partition stones into strings?**
  - **Visit 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 a potential 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:**
    - ▷ *check it is a liberty of the string where its neighbors are in;*
    - ▷ *make sure an empty intersection contributes at most 1 in counting the amount of liberties of a string.*

General implementation hints (1/3)

- **Each node p_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 $p_{i,1}, p_{i,2}, \ldots, p_{i,b}$ are the children of p_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.**

General implementation hints (2/3)

- **Only nodes in the current "partial" tree are maintaining the 3 counters.**
- **Assume $p_{i,1}, p_{i,2}, \ldots, p_{i,b}$ are the children of p_i that are currently in the "partial" tree.**
  - **It is better to maintain a "default" node representing the information of playouts simulated when p_i was a leaf.**
- **When any counter of a node v is updated, it is important to update the counters of all of its ancestors.**
  - ▷ ***For example: the winning rates of all v's ancestors are also changed.***
- **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 changed because the total number of playouts, N, is increased by 1.**

General 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.**

Hints on updating UCB scores

- **When x more simulations are done on a node p, then**
  - **the winning rates of p and p's ancestors may change;**
  - **the exploration scores of p and p's ancestors decrease;**
    - ▷ $c\sqrt{\frac{\log N}{N_i}} \rightarrow c\sqrt{\frac{\log(N+x)}{(N_i+x)}}$
  - **the winning rates of the siblings of p and p's ancestors do not change;**
  - **the exploration scores of the siblings of p and p's ancestors increases;**
    - ▷ $c\sqrt{\frac{\log N}{N_i}} \rightarrow c\sqrt{\frac{\log(N+x)}{N_i}}$
- **Calculating $\log$ and division is time consuming, do not do it unless it is necessary.**
  - **Assume you have to find the max UCB value among children of a node with a total of N simulations.**
    - ▷ ***The value $\log N$ needs to be calculated only once among all children.***
    - ▷ ***Save $\frac{W_i}{N_i}$ and reuse it if it is not changed.***

Hints on UCT tree maintaining

- **After a certain rounds of best-first tree growing as used in UCT tree growing, the shape of the tree is critical in getting a fast and correct convergence.**
  - **Shape of the tree can be roughly quantified by**
    - ▷ ***Total number of nodes:*** n
    - ▷ ***Average depth of leaves:*** $avgd$
    - ▷ ***Maximum depth:*** $maxd$
    - ▷ ***Depth of PV:*** pvd
    - ▷ ***Average branching factor:*** $avgb$
  - **If** $avgd$ **and** $maxd$ **are about the same, then you do not have a good direction of searching.**
  - **If** n **is too small, then your code is not efficient.**
  - **If** n **is too large, then your code does not prune enough.**
  - $\cdots$

Slow code

```
double maxV,Ntotal,N[maxS],W[maxS];
int s; // number of children
int i;
...
Ntotal = 0.0;
for (i=0;i<s;i++)
   Ntotal += N[i];
maxV = -99999.9;
for(i=0;i<s;i+)
   if(maxV < W[i]/N[i] + c * sqrt(log(Ntotal)/log(N[i])))
       maxv = W[i]/N[i] + c * sqrt(log(Ntotal)/log(N[i])));
```

Faster code

```
double maxV,sqrtlogNtotal, temp,
       sqrtlogN[maxS], // = sqrt(log((double) N[i]))
       WR[maxS]; // winning rate = (double) W[i]/ (double) N[i]
int Ntotal, // the total value is calculated when it is updated
    N[maxS],W[maxS];
int s; // number of children
int i;
...
sqrtlogNtotal = sqrt(log((double) Ntotal));
maxV = WR[0] + c * sqrtlogNtotal/sqrtlogN[0];
for(i=1;i<s;i+){
   temp = WR[i] + c * sqrtlogNtotal/sqrtlogN[i];
   if(maxV < temp)
       maxv = temp;
}
```

Data structure for an UCB-tree

```
struct NODE {
            int ply; // the ply from parent to here
            int p_id; // parent id, root's parent is the root
            int c_id[MaxChild]; // children id
            int depth; // depth, 0 for the root
            int Nchild; // number of children
            int Ntotal; // total # of simulations
            double sqrtlogNtotal;
            int Wins; // number of wins
            double WR; // win rate
            } nodes[MaxNodes];

#define parent(ptr) (nodes[ptr].p_id) // id of your parent
#define child(ptr,i) (nodes[ptr].c_id[i]) // the ith child
```

Updating from leaf to root

```
...
// add deltaN simulations with deltaW wins
void update(int id, int deltaW, int deltaN)
{
     nodes[id].Ntotal += deltaN; // additional # of trials
     nodes[id].sqrtlogNtotal = sqrt(log((double) nodes[id].Ntotal));
     nodes[id].Wins += deltaW; // additional # of wins in trials
     nodes[id].WR = (double) nodes[id].Wins
                     / (double) nodes[id].Ntotal;
}

...

do{
     update(ptr,deltaW,deltaN);
     ptr = parent(ptr);
}until(ptr == root);
update(root);
```

Finding PV

```
// compute the UCB score of nodes[id]
double UCB(int id)
{
      return (nodes[id].depth%2) ? (1.0-nodes[id].WR) : (nodes[id].WR)
             c * nodes[parent(id)].sqrtNtotal / nodes[id].sqrtNtotal;
}
...
PV[0] = ptr = root;
while(nodes[ptr].Nchild > 0){
     maxchild = child(ptr,0);
     maxV = UCB(maxchild);
     for(i=1;i<nodes[ptr].Nchild;i++){
      ctemp =  child(ptr,i);
      temp = UCB(ctemp);
      if(maxV < temp){ maxV = temp; maxchild = ctemp; }
     PV[nodes[ptr].depth] = ptr = maxchild;
}
```

Advanced data structure

```
struct NODE {
            int ply; // the ply from parent to here
            int p_id; // parent id, root's parent is the root
            int c_id[MaxChild]; // children id
            int depth; // depth, 0 for the root
            int Nchild; // number of children
            int Ntotal; // total # of simulations
            double sqrtlogNtotal;
            int Scores; // sum1: sum of scores
            int sum2; // sum2: sum of each square of score
            double Average; // average score
            double Variance; // variance of score
            } nodes[MaxNodes];
```

Advanced UCB routine

```
// compute the UCB score of nodes[id]
// for a range of scores [MinS,MaxS]
double UCB(int id)
{
      double range = MaxS - MinS;
      // normalized the average score to be between 0 and 1
      double SR = (nodes[id].Average - MinS) / range;
      return (nodes[id].depth%2) ? (1.0-SR) : (SR)) +
             c * nodes[parent(id)].sqrtNtotal / nodes[id].sqrtNtotal;
}
```

Advanced updating routine

```
...
// add deltaN simulations with deltaS additional scores
// and sum of square of scores deltaS2
void update1(int id, int deltaS, int deltaS2, int deltaN)
{
     nodes[id].Ntotal += deltaN; // additional # of trials
     nodes[id].sqrtlogNtotal = sqrt(log((double) nodes[id].Ntotal));
     nodes[id].Scores += deltaS; // additional scores in trials
     nodes[id].Sum2 += deltaS2;
     nodes[id].Average = (double) nodes[id].Scores
                     / (double) nodes[id].Ntotal;
     nodes[id].Variance = ((double) nodes[id].sum2 -
        2 * nodes[id].Average * nodes[id].Scores)
        / (double) nodes[id].Ntotal;
}
```

Comments (1/2)

- Using the idea of sampling to evaluate a position was used previously for other games such as 6x6 Othello [Abramson'90].
- Proven to be successful on a few games.
  - Very successful on computer Go.
- Not very successful on some games.
  - Not currently outperform alpha-beta based programs on Chess or Chess-like games.
- Performance becomes better when the game is going to converge, namely the endgame phase.
- Need 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.
  - It also depends on the speed of your simulation.

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 using statistical learning or deep learning.**

References and further readings (1/3)

* B. Bruegmann. Monte Carlo Go. unpublished manuscript, 1993.
* 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.
* P. Auer, N. Cesa-Bianchi, P. Fischer. Finite-time analysis of the multiarmed bandit problem. Machine Learning, pages 235–256, 2002.
* 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.

References and further readings (2/3)

- **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.**
- **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.**
- **B. Abramson. Expected-outcome: a general model of static evaluation. IEEE Transactions on Pattern Analysis and Machine Intelligence archive Volume 12 Issue 2, February 1990, Pages 182-193.**

References and further readings (3/3)

- Chaslot, Guillaume and Bakkes, Sander and Szita, Istvan and Spronck, Pieter Monte-Carlo Tree Search: A New Framework for Game AI. Proceedings of the BNAIC 2008, the twentieth Belgian-Dutch Artificial Intelligence Conference, pages 389–390, 2008

* Silver, David, Huang, Aja, Maddison, Chris J, Guez, Arthur, Sifre, Laurent, Van Den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, et al. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587):484-489, 2016.

* Silver, David, Schrittwieser, Julian, Simonyan, Karen, Antonoglou, Ioannis, Huang, Aja, Guez, Arthur, Hubert, Thomas, Baker, Lucas, Lai, Matthew, Bolton, Adrian, et al. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676):354359, 2017