



MONTE-CARLO TREE SEARCH

Olivier Teytaud, Hassen Doghmen
Tao, INRIA Saclay-IDF

ABSTRACT HERE.....

Minimax Algorithm: the brute force solution

- For the MAX player
1. Generate the game to terminal states
 2. Apply the utility function to the terminal states
 3. Back-up values
 - At MIN play assign
 - At MAX play assign
 4. At root, MAX chooses the highest p_i

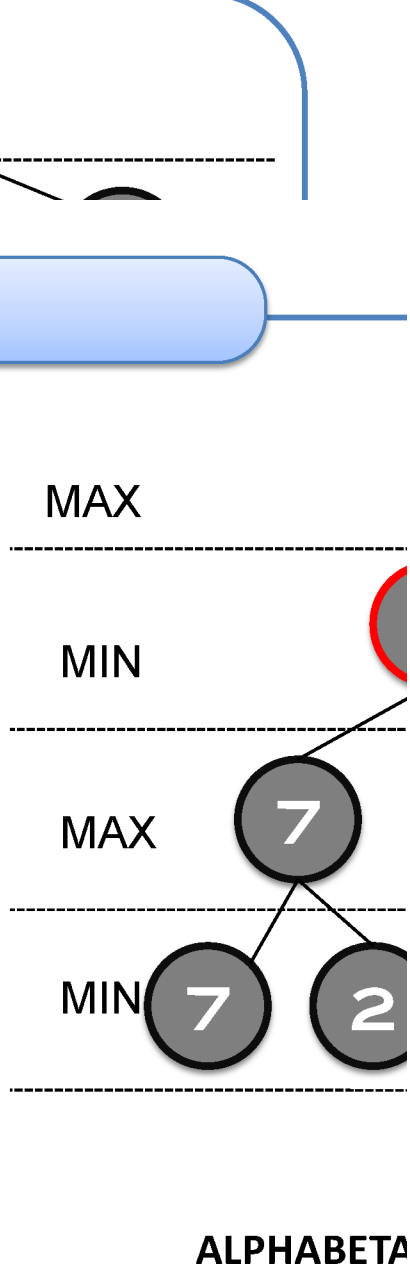
- Perfect play for deterministic games
- The program must assume opponent will select it to make no mistakes

- Totally impractical since whole tree
 - Time complexity is
 - Space complexity is

AlphaBeta: an improvement of Minimax

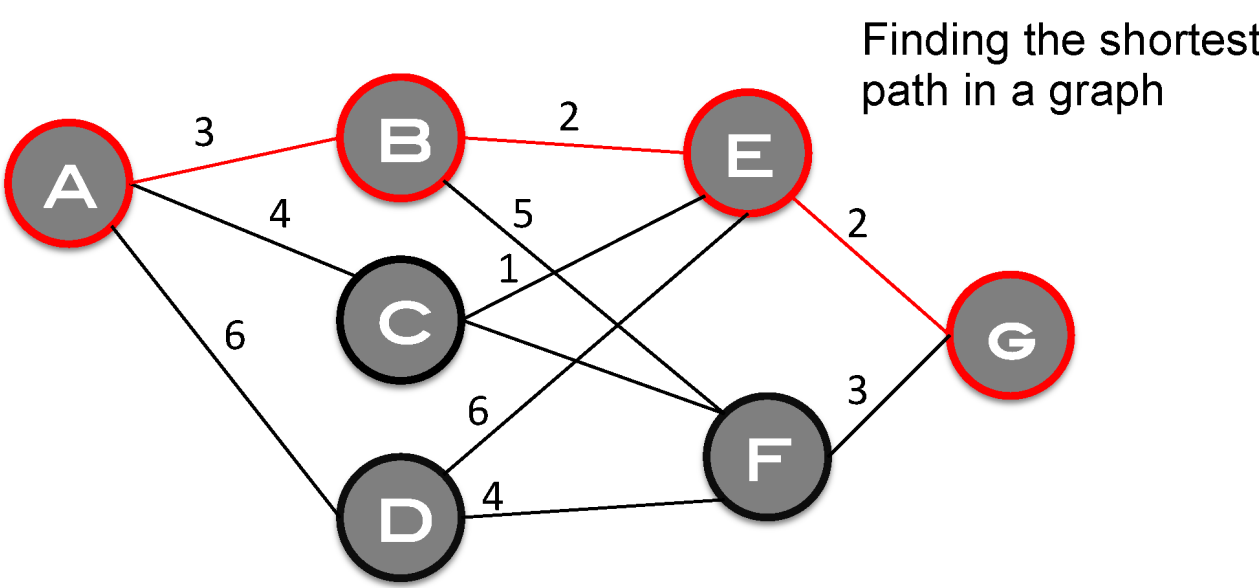
Algorithm: a MiniMax with pruning.

- ALPHABETA takes advantage of the calculated states when determining whether to continue searching a particular subtree or prune it.
- Pruning does not affect final result.
- Good move ordering improves effectiveness of pruning.
- Time complexity
 - $O(b^{d/2})$
 - means we go from an effective branching factor of b to \sqrt{b} (e.g. 35 \rightarrow 6 for chess).



Dynamic programming:

- Dynamic Programming is a method used to optimize the objective function in a planning problem.
- Recursively **decompose** the problem into subproblems, solve subproblems and construct the global solution.
- However, the algorithm is **not anytime**; if stopped before then, there is no result.
- Can be seen as one player game against a **random** player.



Monte-Carlo Tree search: Algorithm sketch

Input: a situation s
Output: a (hopefully good) move m for situation s
Memory = empty; statistics = empty mapping

```
While (time left > 0)
  s' = s, game = {}
  While (s' in memory, not terminal)
    s' = reachable situation such that nbWins(s') / nbSims(s') max
  endwhile
  s'' = s'
  While (s' not terminal)
    s' = random reachable situation from s'
  endwhile
  result = result(s')
  memory = memory + s''
  for all states in the simulation
    statistics(s) = statistics(s) + statistics(game, result)
  endfor
endwhile
```

Restart games

Choose move to be explored Using statistics

Choose move to be explored

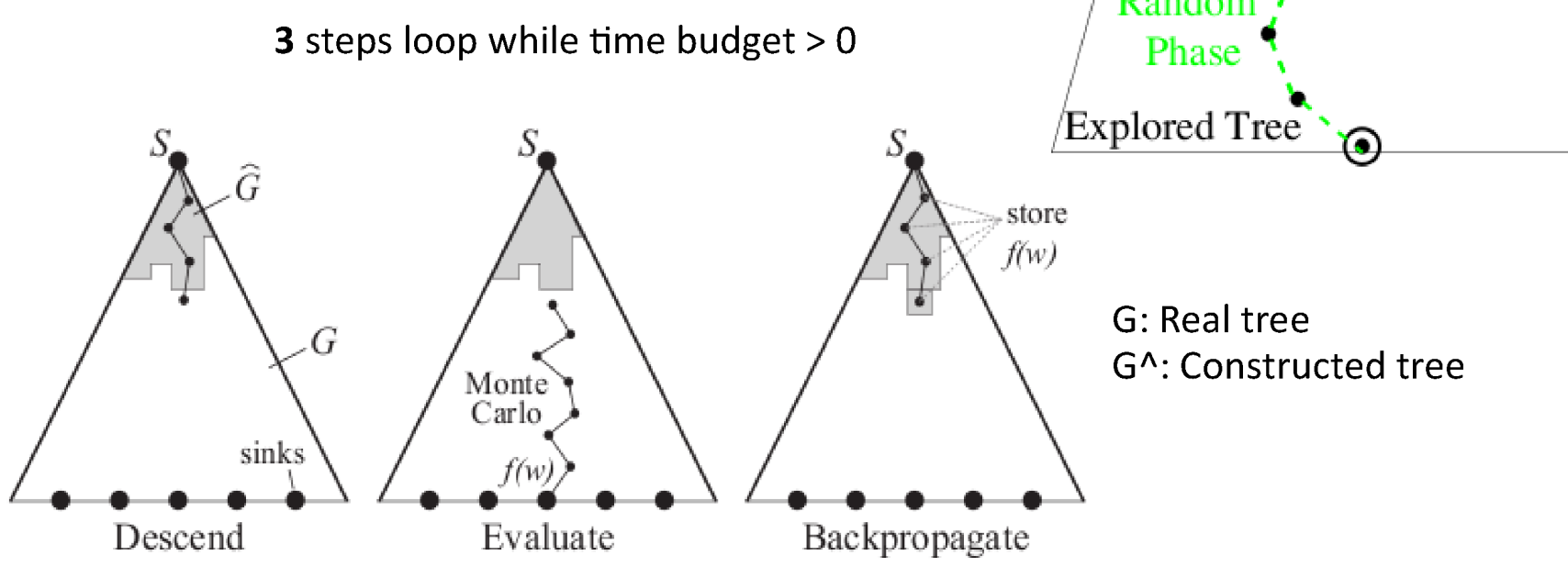
Update statistics

Monte-Carlo Tree search: how it works ?

Algorithms for taking decisions in uncertain environments.

Principles:

- Iteratively growing **unbalanced** tree (choosing a decision is seen as bandit problems – UCT).
- Random simulations (Monte-Carlo simulations as an **evaluation** function).



K-one armed bandit problem

- Problem Description :
 - p_1, \dots, p_N unknown probabilities $\in [0,1]$
 - At each time step $t \in \{1, \dots, N\}$
 - Choose arm $a_t \in \{1, \dots, N\}$ (as a function of a_j and r_t $j < t$)
 - With probability $p_{j,t}$
 - win ($r_t = 1$)
 - loose ($r_t = 0$)

Goal: minimize a Regret: $R_N = N \max\{p_i\} - \sum r_t$ ($t < N$)

Classical solution:
UCB: Choose arm j maximizing the compromise:

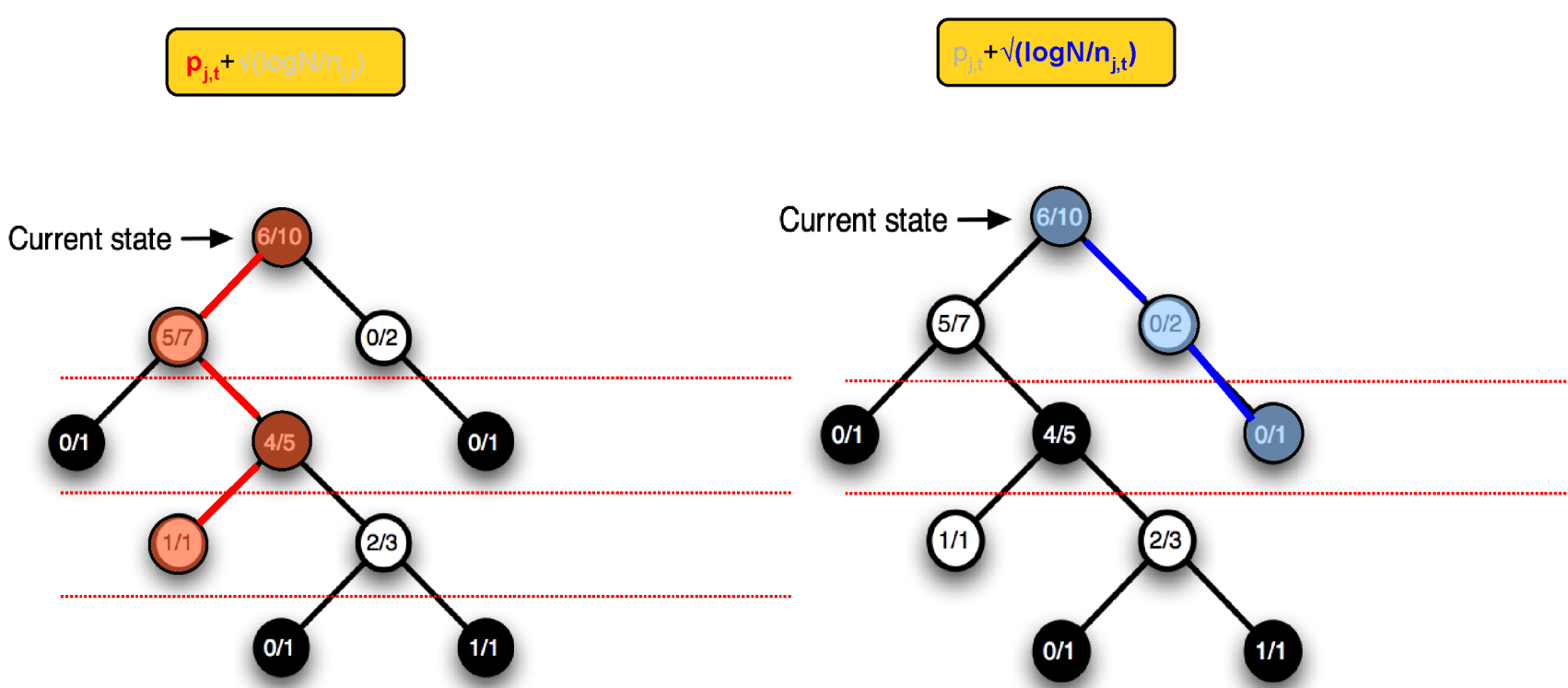
$$p_{j,t} + \sqrt{\frac{\log N}{n_{j,t}}}$$

(Lai et al; Auer et al)

Optimal regret $O(\log(n))$

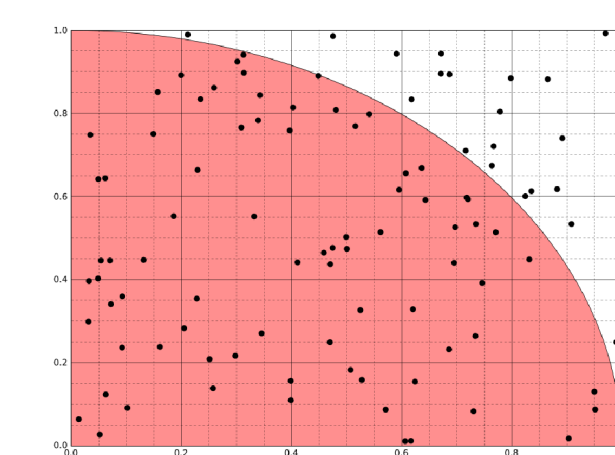
$p_{j,t}$: Empirical average for decision j
 N : Total number of trials
 $n_{j,t}$: number of trials with decision j

Exploitation vs Exploration dilemma



Monte-Carlo simulations

- simulations to approximate π
- Simulations to find the optimal move

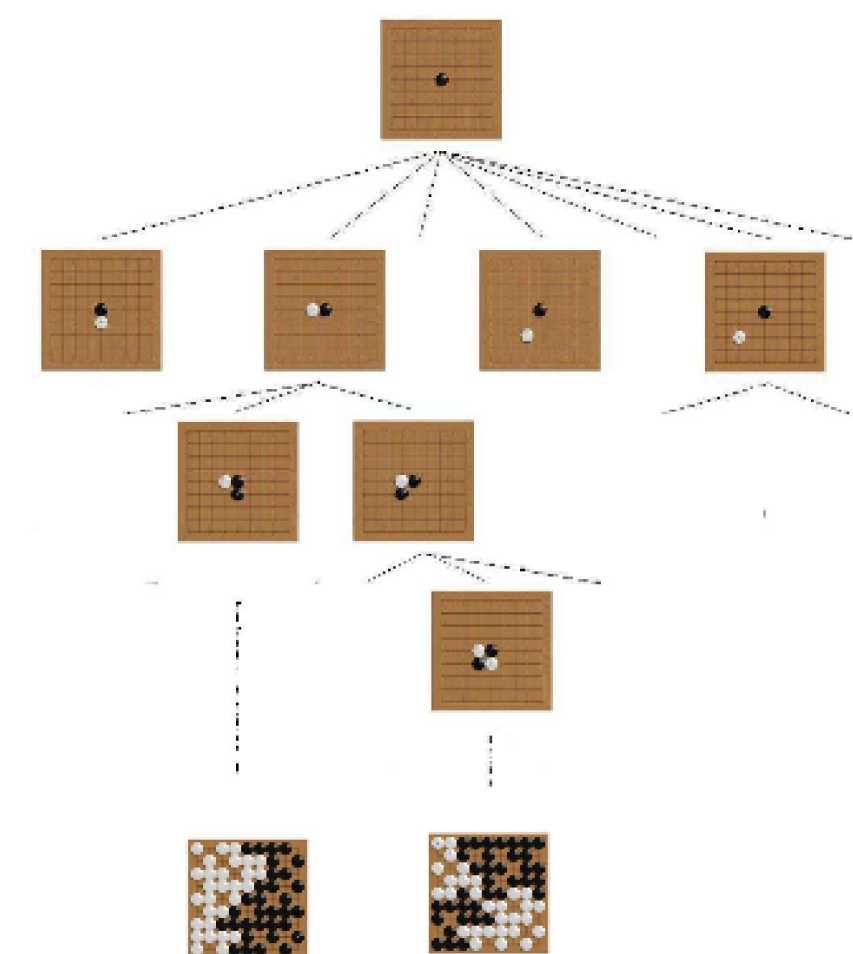


```
double nbTotalPoints = 0;
double nbPointsSquare = 0;
double PI;
int timeLeft = 10000;
double x, y;

srand(time(NULL));
cout << "Estimating PI by Monte-Carlo Simulations" << endl;

while (timeLeft > 0) {
  x = (double)rand() / 10;
  y = (double)rand() / 10;
  nbTotalPoints++;
  if ((sqrt(x*x + y*y) < 1) {
    nbPointsSquare++;
  }
  timeLeft--;
}

PI = 4 * (nbPointsSquare / nbTotalPoints);
```



Monte-Carlo evaluation function consists in starting from the position and playing with a *simulation policy* until the end of the game. Then the game is scored exactly simply using the rules.