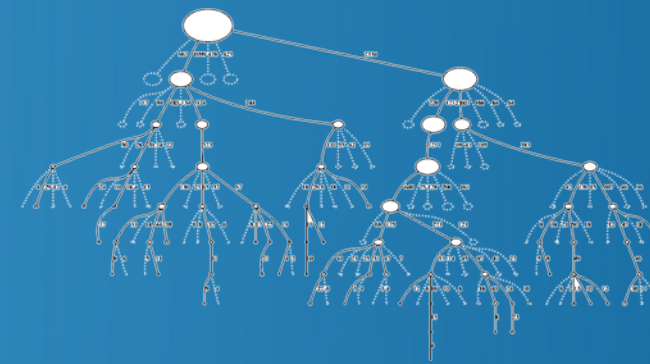


An Introduction to Monte Carlo Tree Search

Manlio Morini, *software developer*

morini@eosdev.it

Italian C++ Community

www.italiancpp.org

C++ Day – Parma 2019

The algorithm for General Game Playing (?)

Google DeepMind

2016

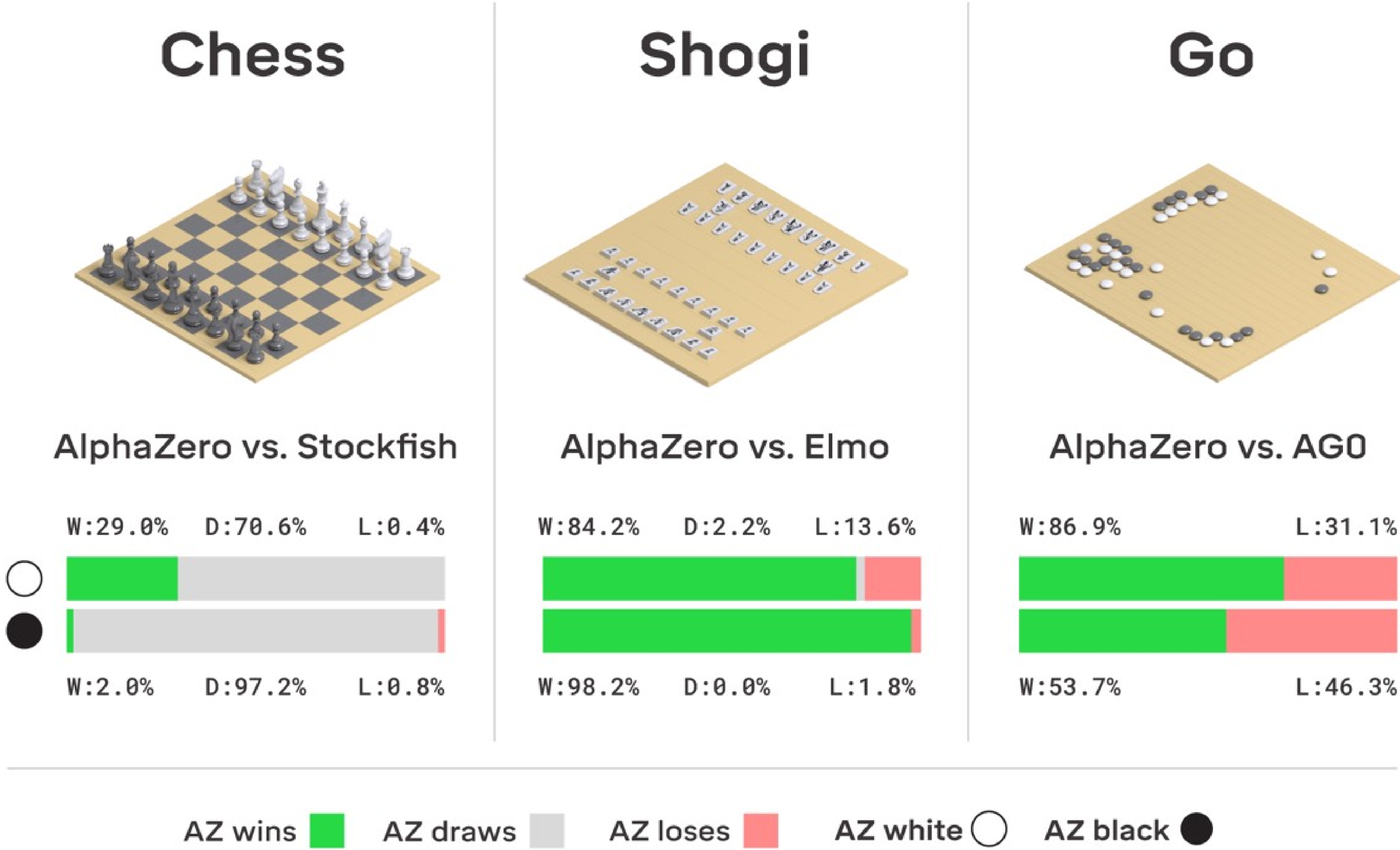
- AlphaGo became the first computer program to beat the world champion in a game of Go.

The algorithm for General Game Playing (?)

2017 – 2018

- Starting from random play and given no domain knowledge except the game rules, the AlphaZero program taught itself to play chess, shogi, and Go, defeating a world champion program in each game.

The algorithm for General Game Playing (?)



Foreward

- MCTS is a heuristic search algorithm inspired from multi-armed bandits to efficiently **explore a tree of possible action sequences**.
- Mostly employed in game play.
- Likely **birthday**: 2006. Rémi Coulom describes the application of Monte Carlo methods to tree-search / Kocsis and Szepesvári develop the UCT (*Upper Confidence bounds applied to Trees*) algorithm.

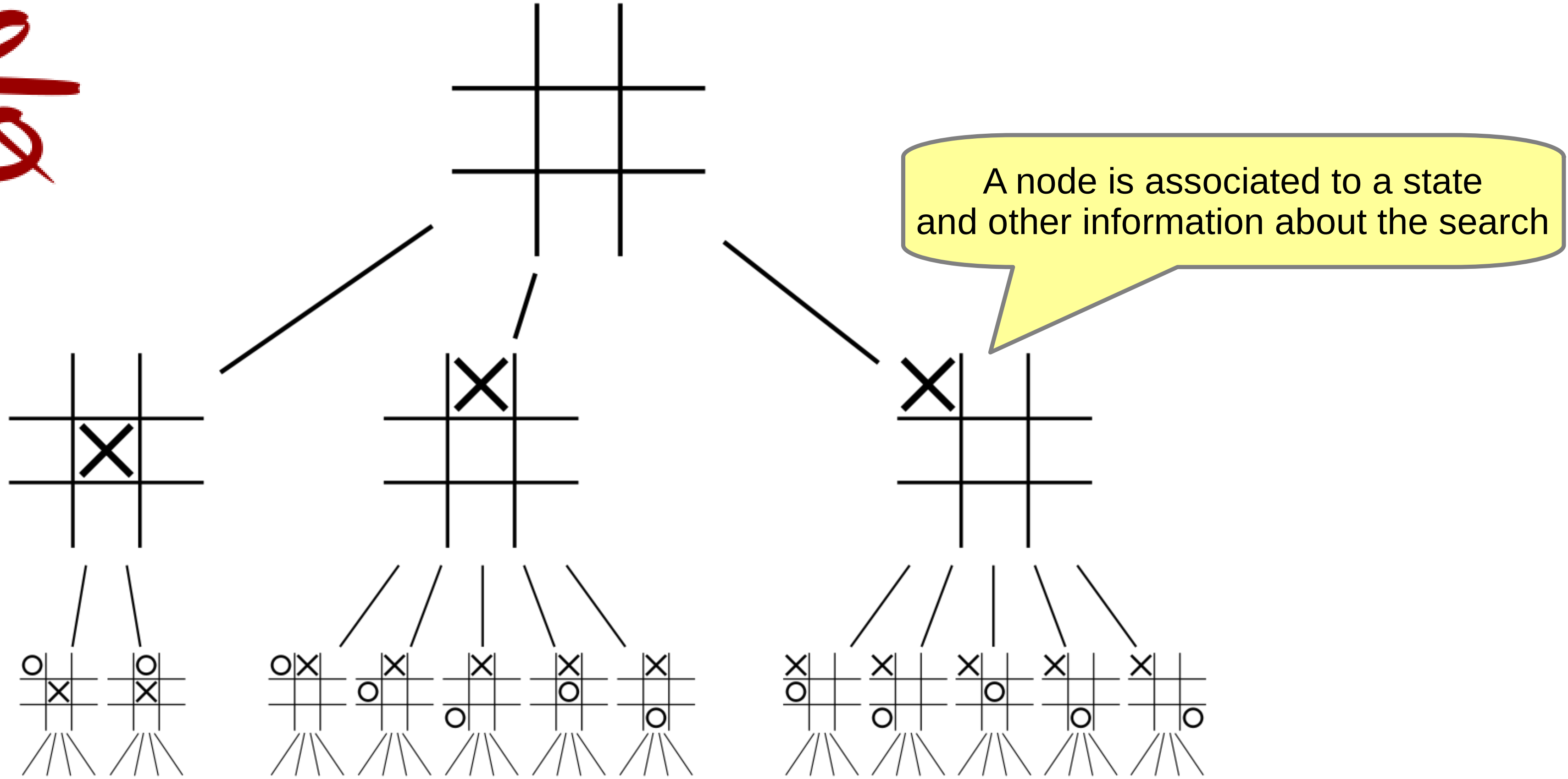
Scope

MCTS application extends beyond games, it can theoretically be applied to any domain that can be described in terms of `{state, action}` pairs and simulation used to forecast outcomes

MCTS keywords tag cloud (first 100 most relevant papers about MCTS according to https://ieeexplore.ieee.org)

Game Tree

Partial game tree for the first two plies of tic-tac-toe



MCT(ree)S

MCTS builds a search/game tree (or graph) from scratch, by simulations, storing statistical information needed to choose good moves.

The `state` class - adapter pattern

```
class state
{
public:
  using action = /* the representation of an action/move */;

  std::vector<action> actions() const;  // set of available actions in this
                                        // state. MUST return an empty
                                        // container for final states

  void take_action(action);             // performs the required action
                                        // changing the current state

  unsigned agent_id() const;            // active agent

  std::vector<double> eval() const;     // how good is this state from each
                                        // agent's POV

  bool is_final() const;                // returns `true` if the state is
                                        // final

private:
  // ...Hic sunt leones...
};
```

Legal actions only

Standard MCTS requires meaningful values
ONLY WHEN REACHING A FINAL STATE

Basic algorithm

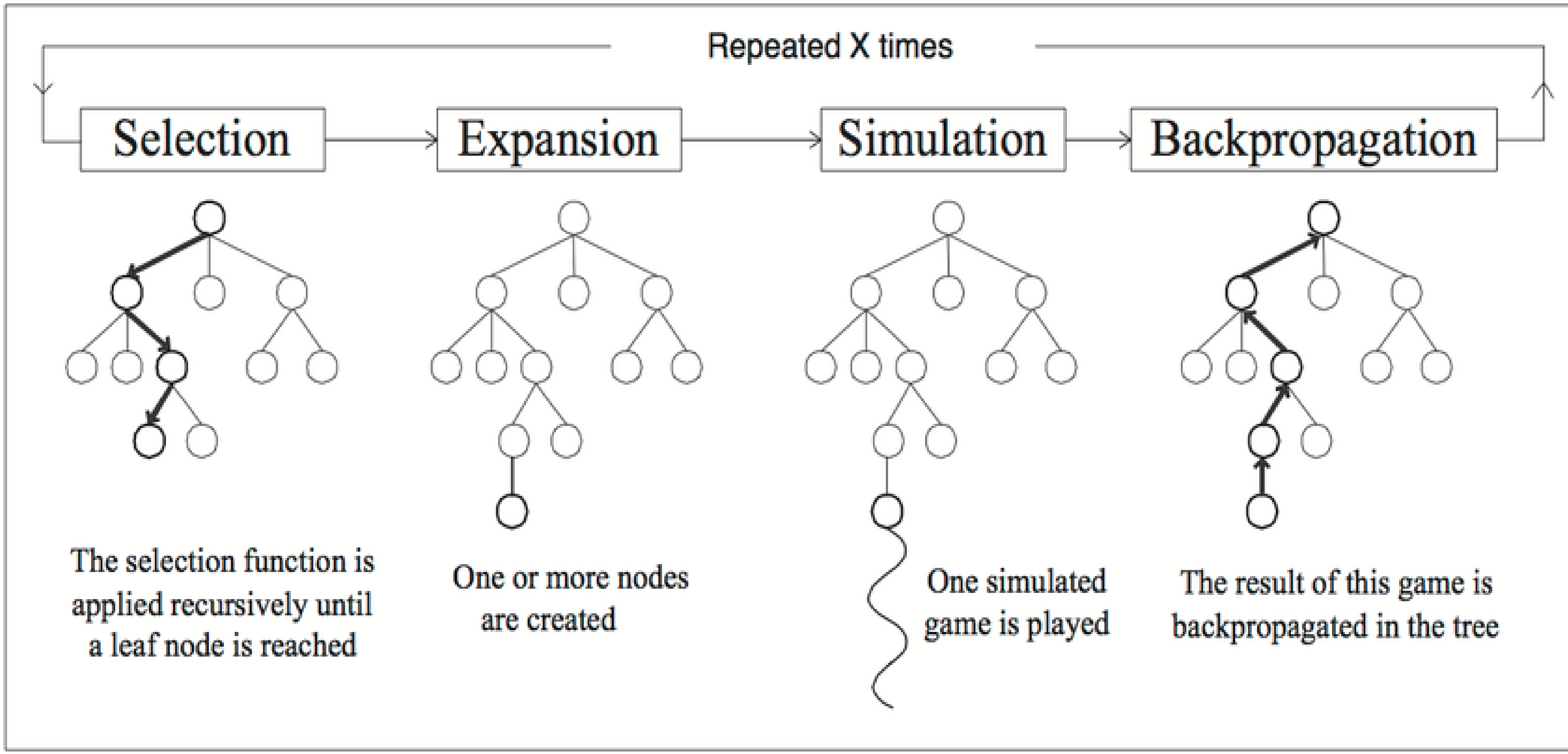
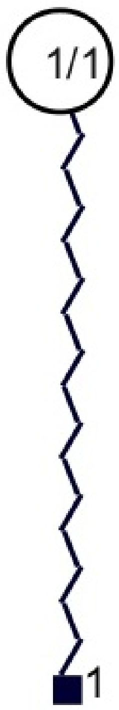


Figure from Chaslot (2006)

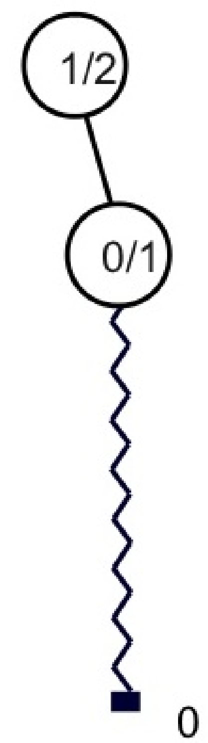
MCTS – Iterations

1 iteration



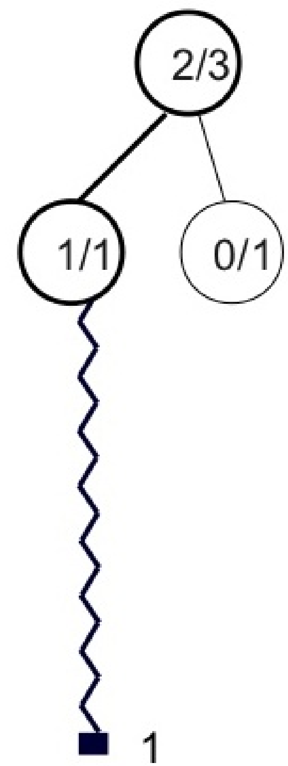
MCTS – Iterations

2 iterations



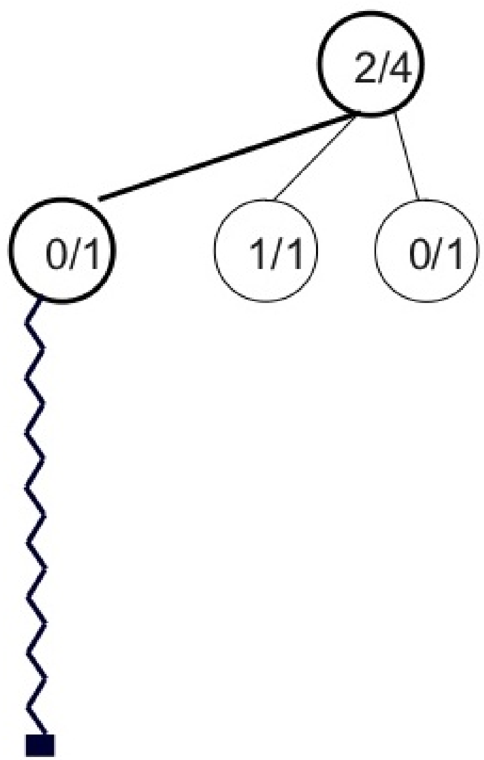
MCTS – Iterations

3 iterations



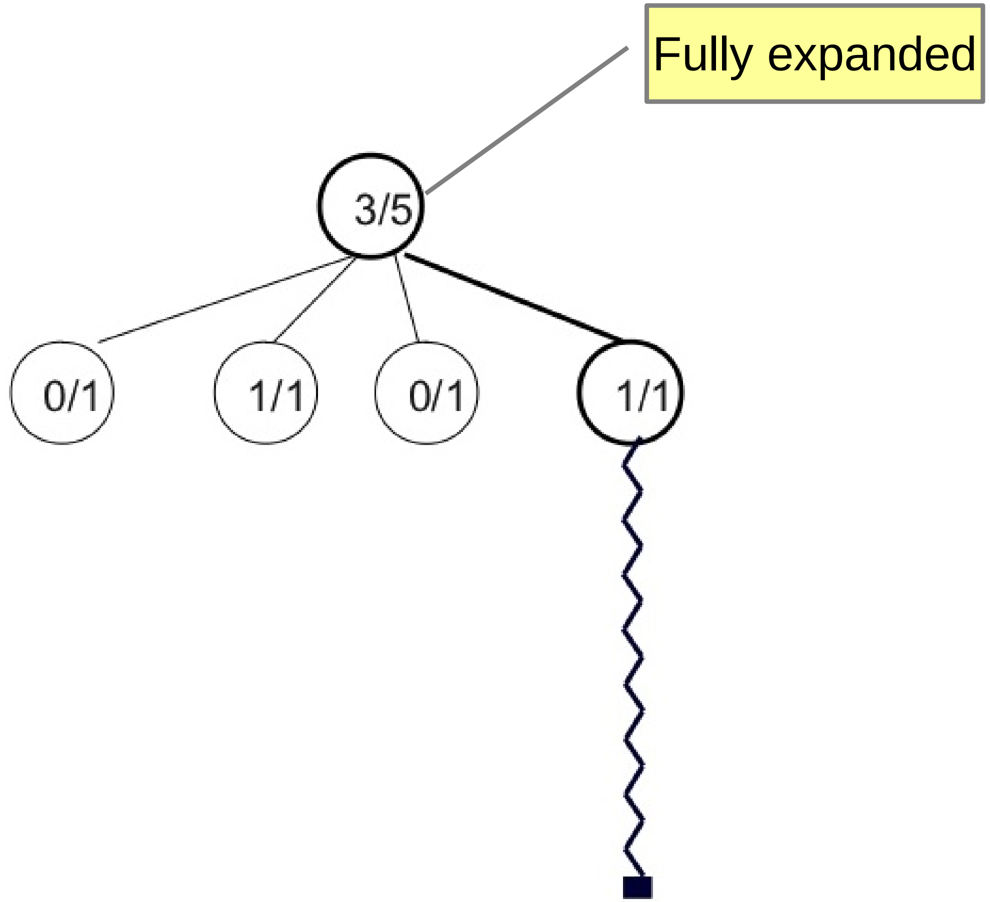
MCTS – Iterations

4 iterations



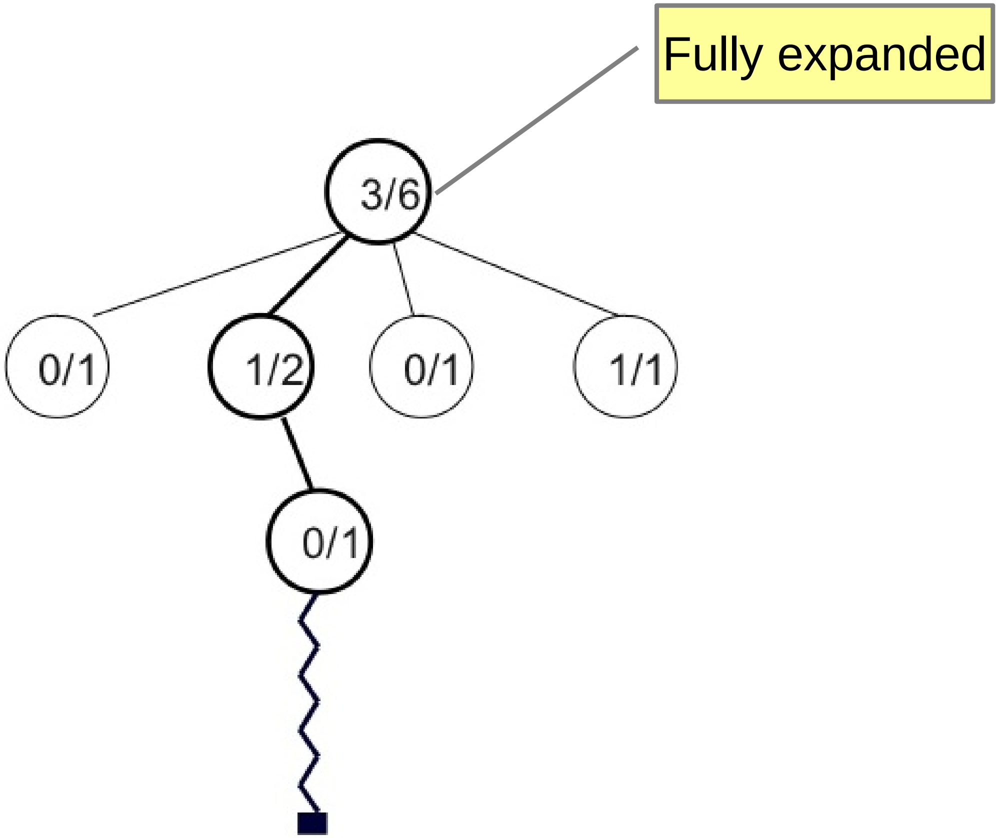
MCTS – Iterations

5 iterations



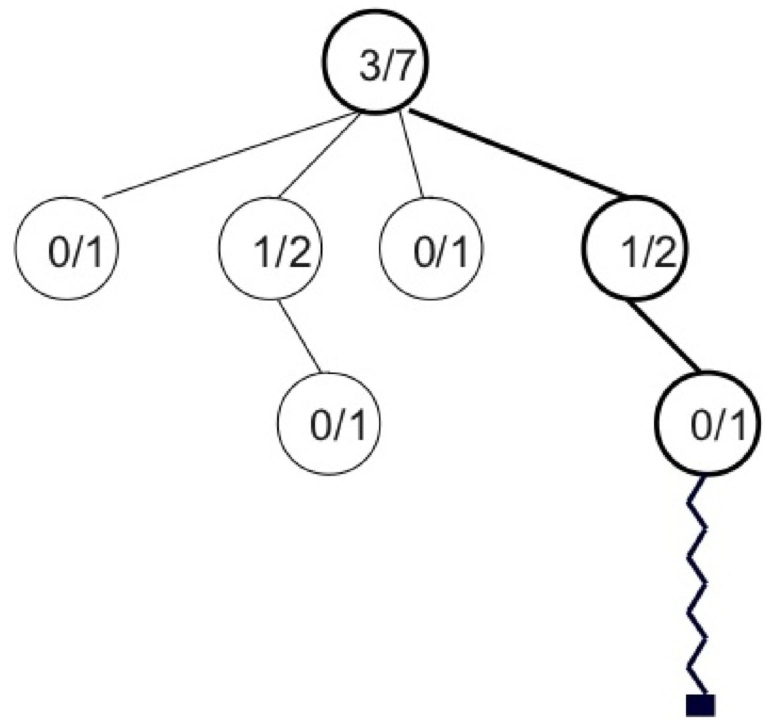
MCTS – Iterations

6 iterations



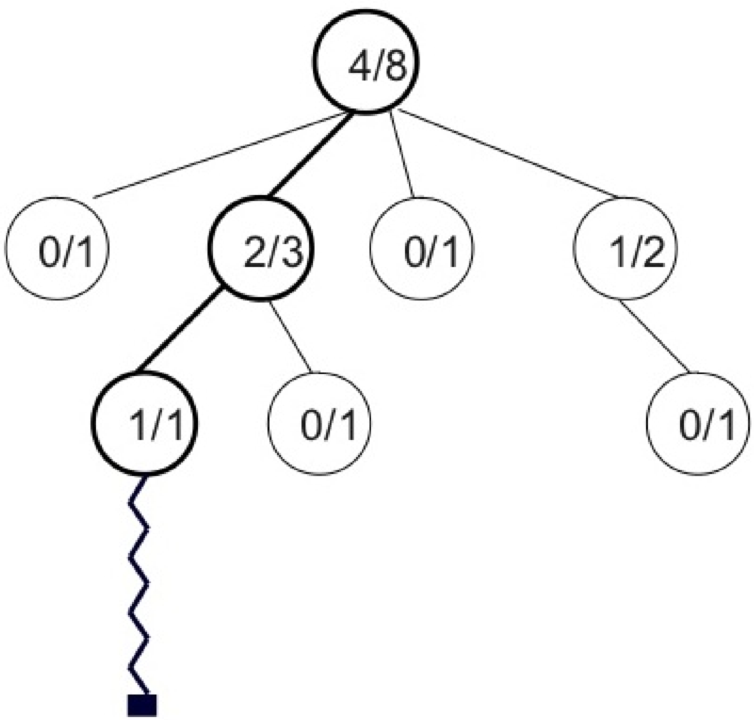
MCTS – Iterations

7 iterations



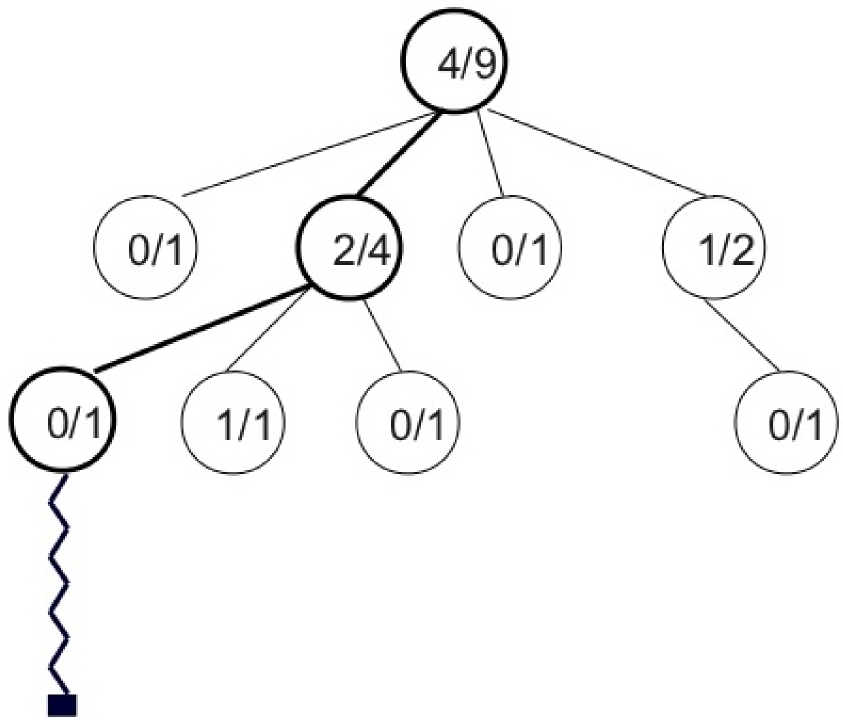
MCTS – Iterations

8 iterations



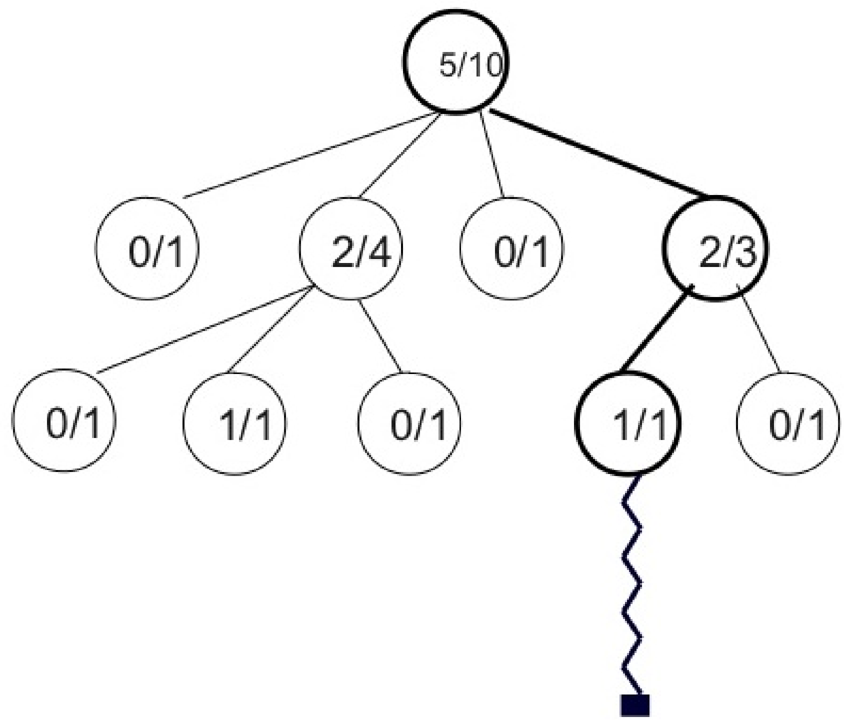
MCTS – Iterations

9 iterations



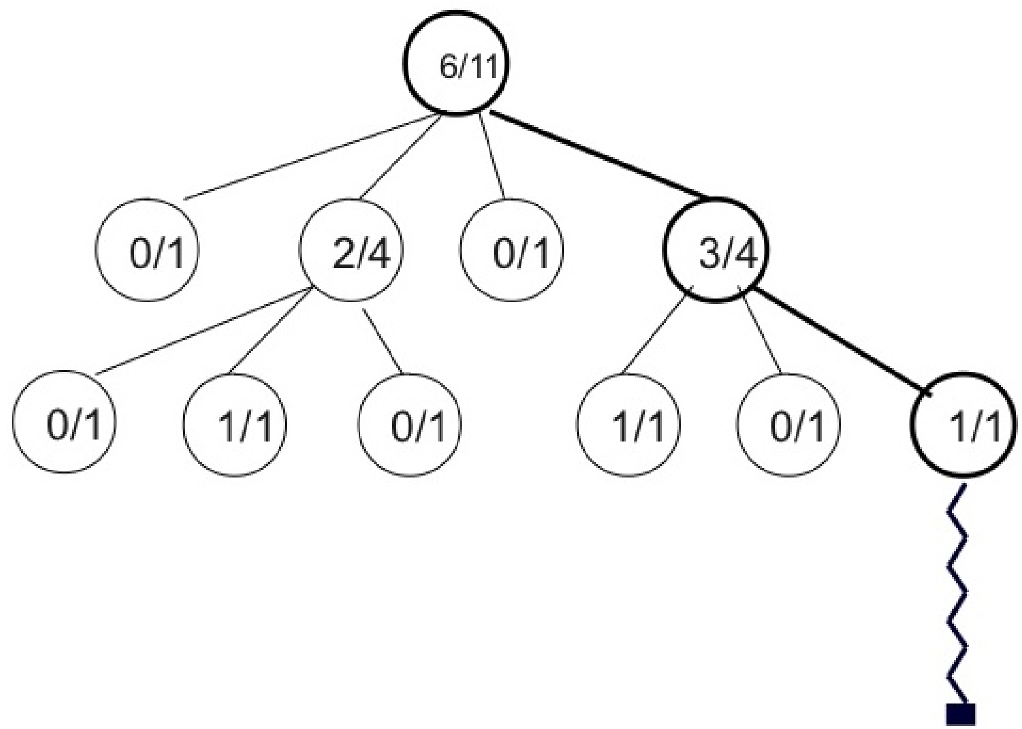
MCTS – Iterations

10 iterations



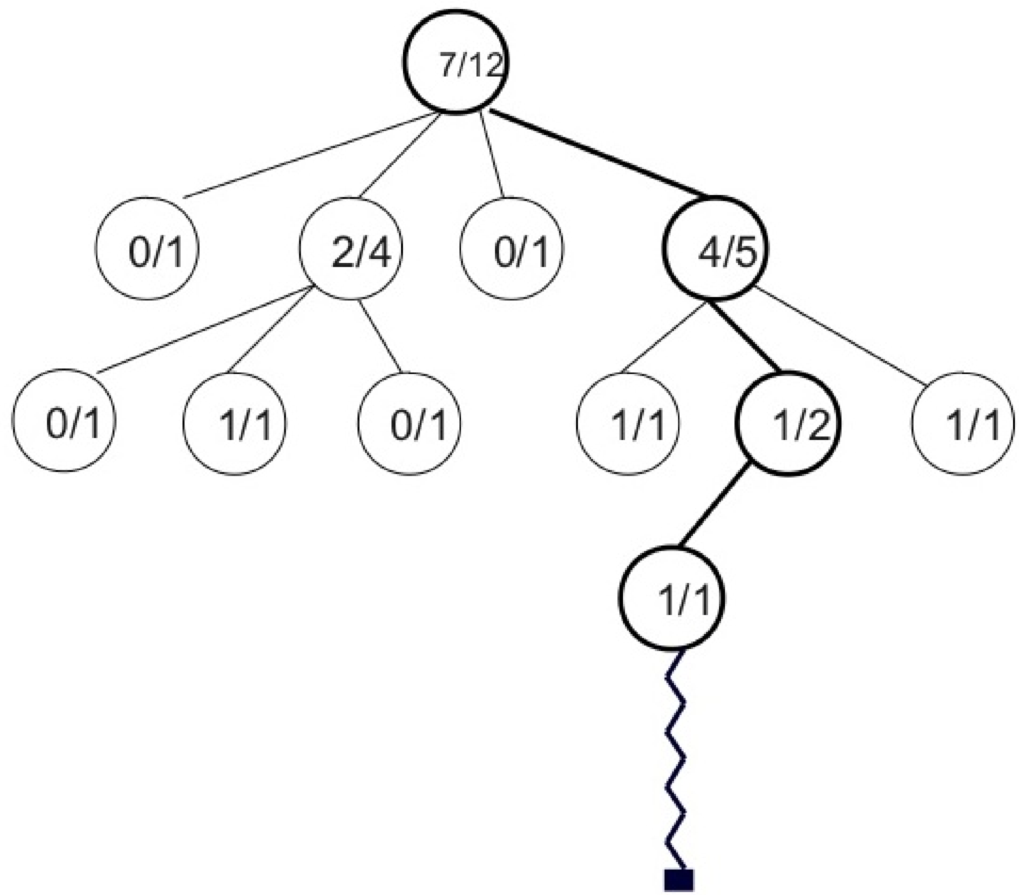
MCTS – Iterations

11 iterations



MCTS – Iterations

12 iterations



MCTS Tree

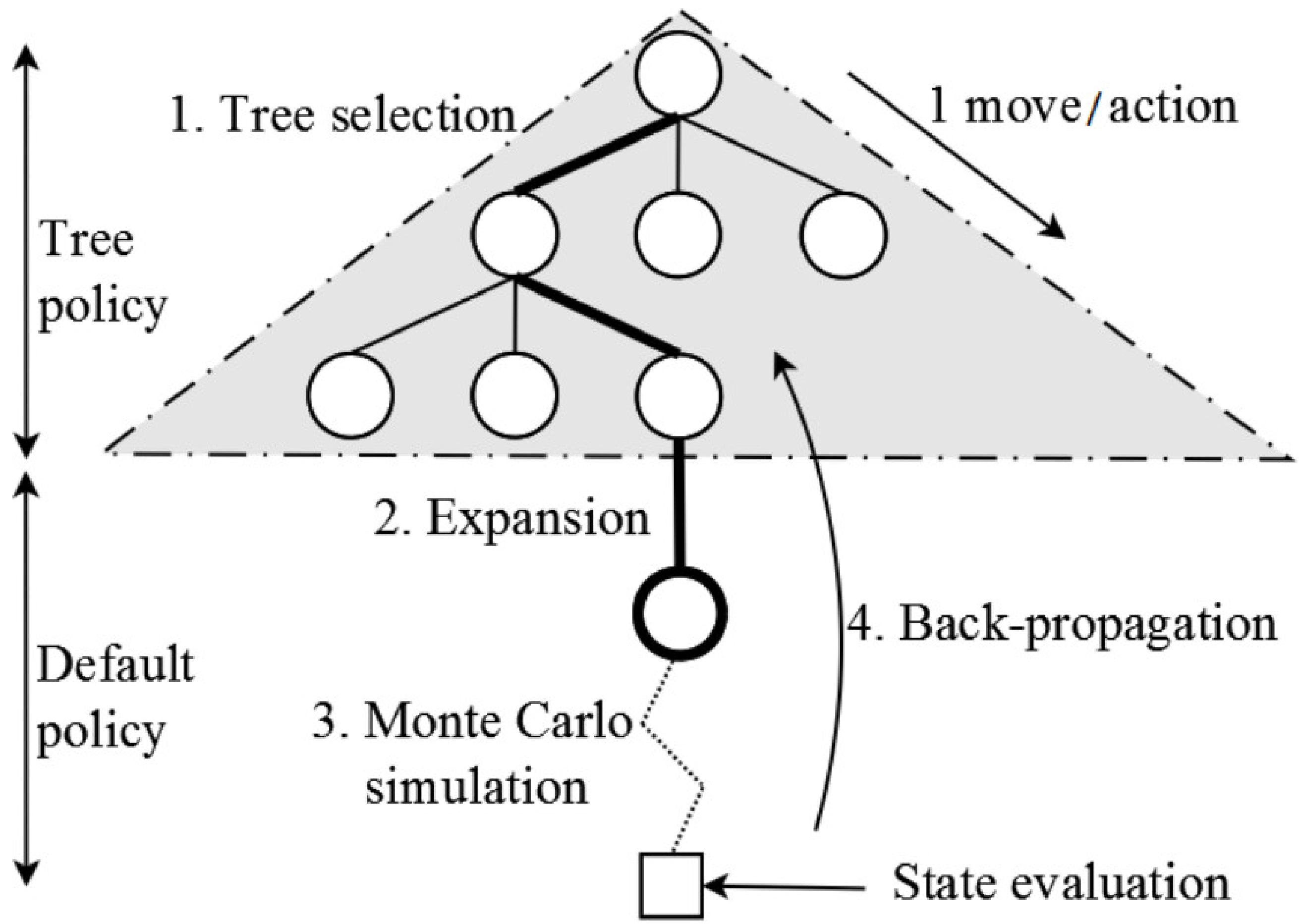
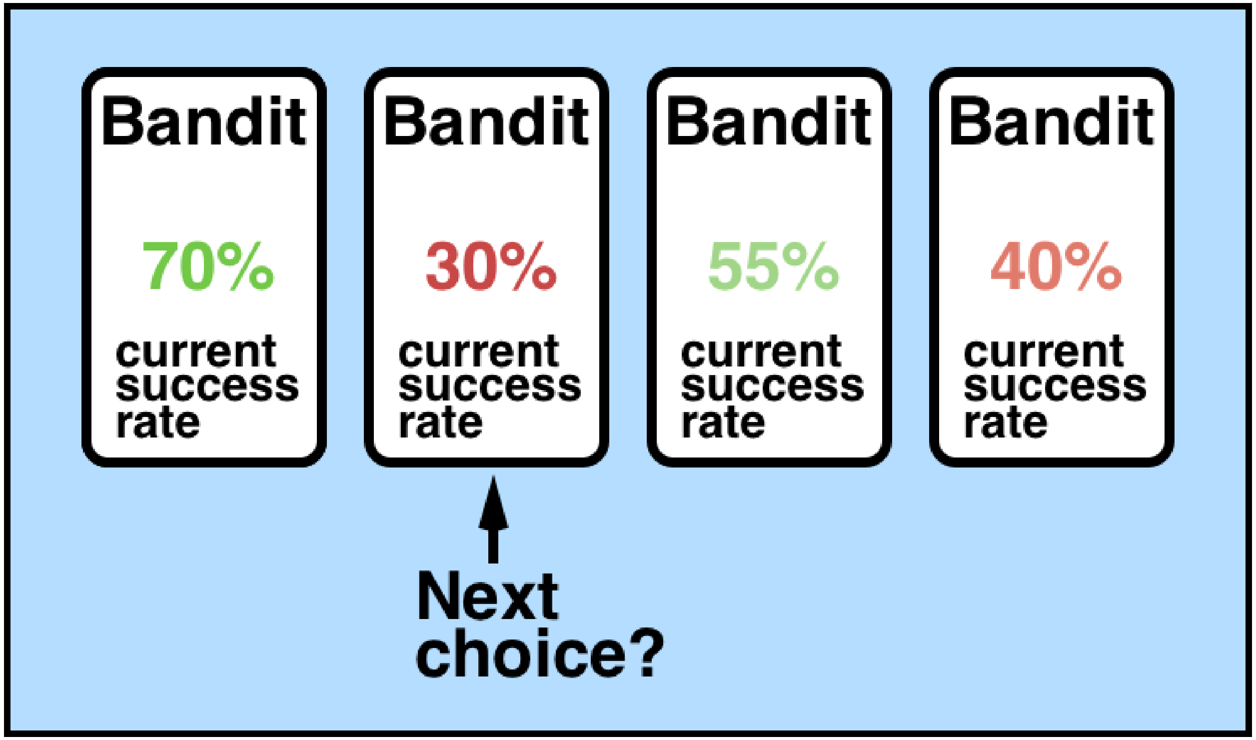


Figure from Chaslot (2006)

Selection – UCB and multiarmed bandit problem

- Node selection during tree descent is achieved by choosing the node that maximizes some quantity.
- An ***Upper Confidence Bounds*** (UCB) formula is typically used.
- Analogous to the *multiarmed bandit problem*: a player must choose the slot machine (bandit) that maximizes the estimated reward each turn.



Selection – UCB (Upper Confidence Bounds)

$$v_i + C \cdot \sqrt{\frac{\ln(N)}{n_i}}$$

- v_i is the estimated (average) value of the node;
- n_i is the number of times the node has been visited;
- N is the number of times its parent has been visited;
- C is a tunable bias parameter.

Selection - Exploitation vs Exploration

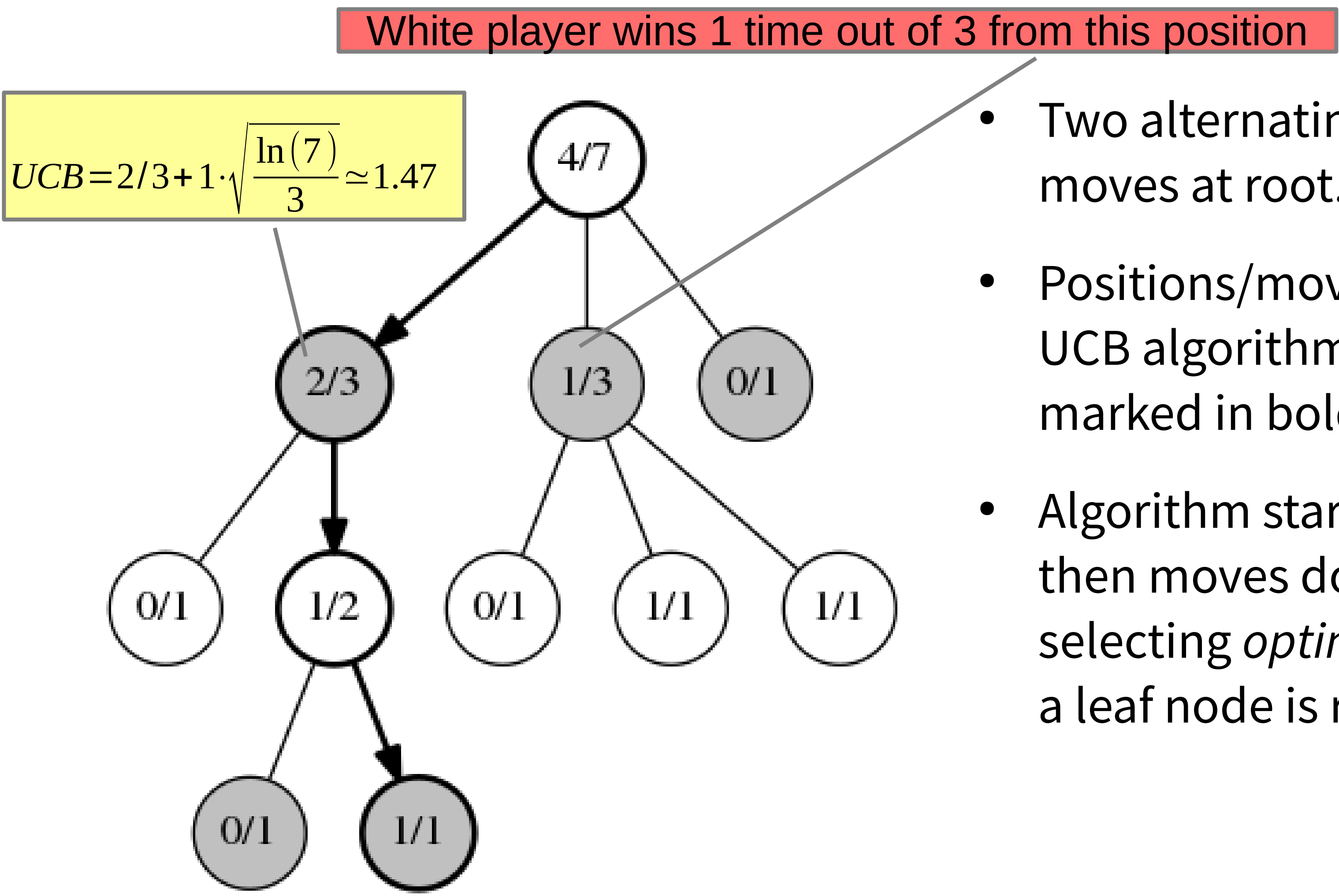
Exploration

Exploitation

$$v_i + C \cdot \sqrt{\frac{\ln(N)}{n_i}}$$

- UCB formula balances the *exploitation* of known rewards with the *exploration* of relatively unvisited nodes.
- Reward estimates are based on random simulations, so nodes must be visited a number of times before these estimates become reliable.
- MCTS estimates will typically be unreliable at the start of a search but converge to more reliable estimates given sufficient time and perfect estimates given infinite time.

Selection – Example (two players game)



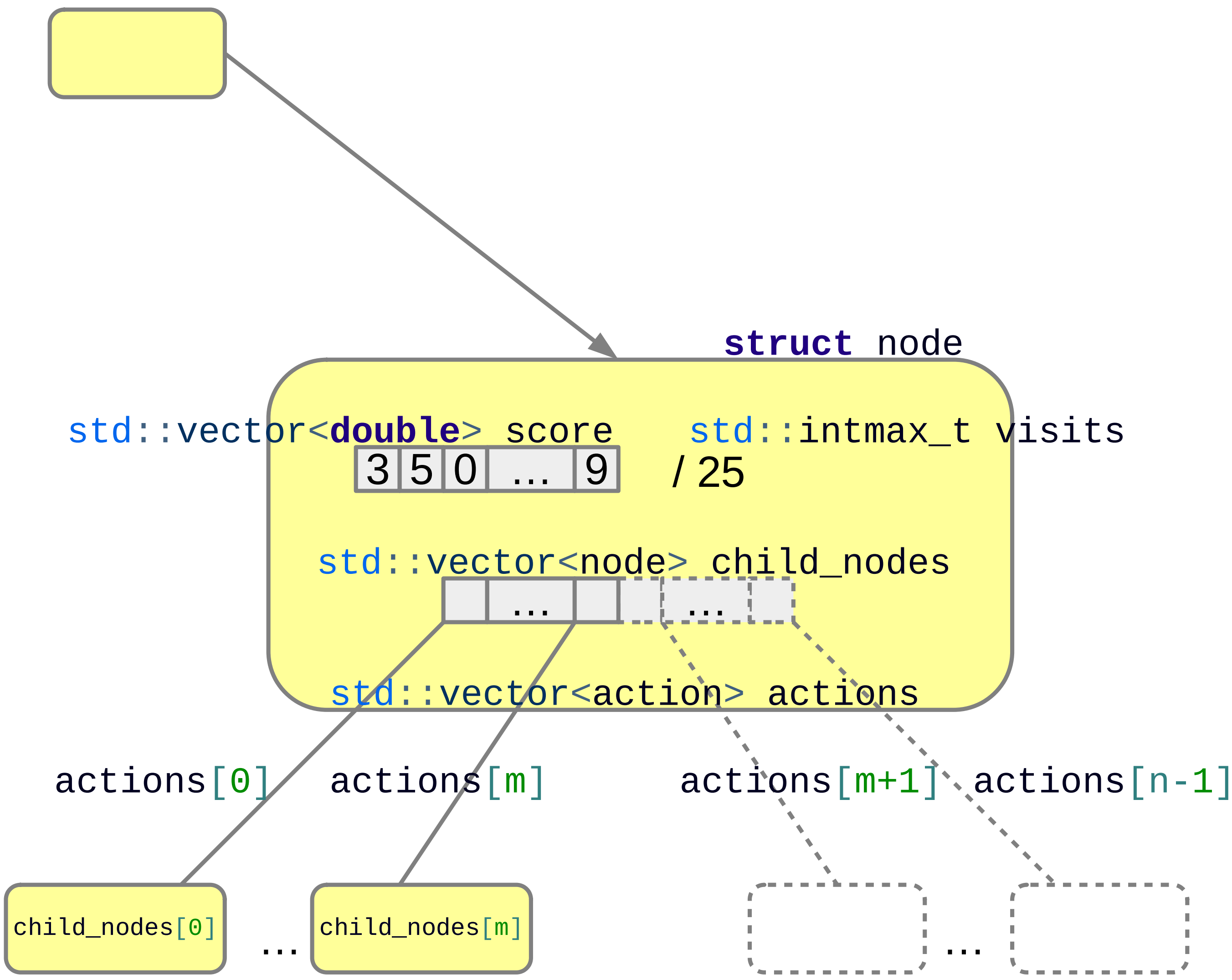
- Two alternating players. White moves at root.
- Positions/moves selected by the UCB algorithm at each step are marked in bold.
- Algorithm starts at root node, then moves down the tree by selecting *optimal* child node until a leaf node is reached.

\`node\` - Data members (partial list 1)

```
struct node
{
  explicit node(const STATE &);

  // ...

  // *** DATA MEMBERS ***

  // ...

  const std::vector<action> actions;
  std::vector<node>      child_nodes;

  // Score of the associated state from multiple POVs. It’s an estimated
  // value based on simulation results.
  std::vector<double> score;

  std::intmax_t visits;  // number of times this node has been visited

  agent_id_t agent_id;   // id of the active agent
};
```

E.g. `{100, 20}` means
first player has a good position
(5 times better than the second player)

`node` - Graphical representation



`node` - Constructor (partial)

```
node::node(const STATE &state /* ... */)
  : actions(state.actions()), child_nodes(),
    score(), visits(0), agent_id(state.agent_id())  // ...
{
  child_nodes.reserve(actions.size());  // to avoid reallocation
}
```

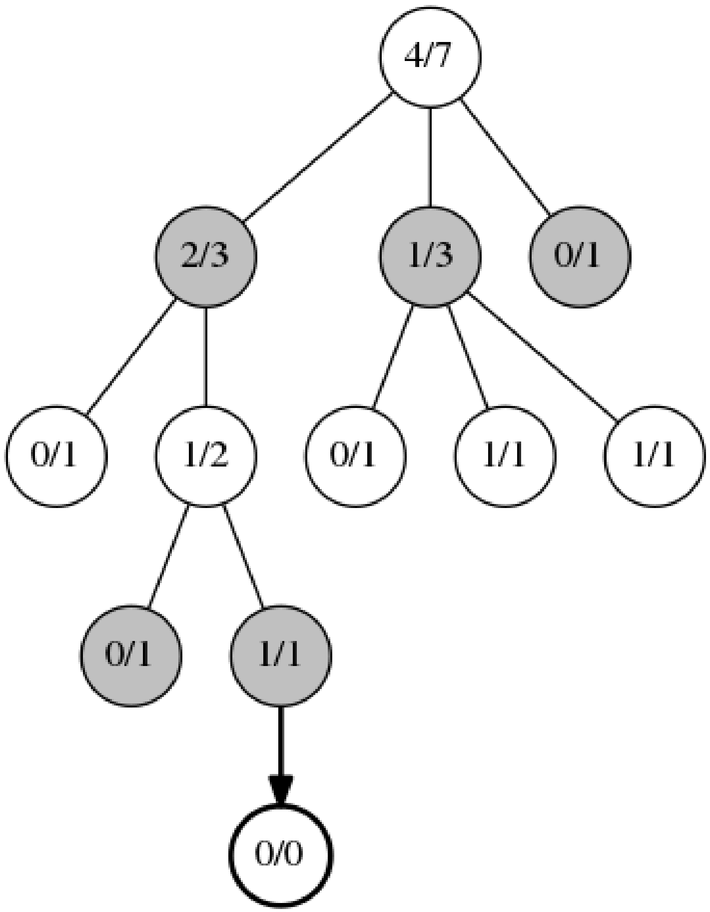
`node` - Selection

```
std::pair<action, node> *node::select_child()
{
  const auto ucb =  // UCB score of a child node
    [this](const node &child)
    {
      if (!child.visits)
        return std::numeric_limits<double>::max();

      // Agent-just-moved point of view for the score.
      return child.score[agent_id] / child.visits
             + uct_k * std::sqrt(std::log(visits) / child.visits);
    };

  const auto child(std::max_element(child_nodes.begin(), child_nodes.end(),
                                    [ucb](const node &lhs, const node &rhs)
                                    {
                                      return ucb(lhs) < ucb(rhs);
                                    }));
  const auto pos(std::distance(child_nodes.begin(), child));


  return {actions[pos], &*child};
}
```

MCTS – Skeleton of algorithm (1)

```
while (!stop_request)
{
  node *n(&root_node);
  STATE state(root_state_);

  // Selection.
  while (n->fully_expanded_branch())
  {
    const auto [best_action, best_child] = n->select_child();

    n = best_child;
    state.take_action(best_action);
  }

  // Expansion...

  // Simulation (aka Playout / Rollout)...

  // Backpropagation...

  // Polling for stop conditions...
}
```

Expansion



- Expansion, occurs when you can no longer apply UCB. An unvisited child position is randomly chosen and a new node is added to the tree.
- The position marked 1/1 at the bottom of the tree has no further statistics records under it, so we choose a random move and add a new record for it (bold).

`node` - Data members (partial list 2)

```
struct node
{
  explicit node(const STATE &, node * = nullptr);

  std::pair<action, node *> select_child();
  node *add_child(const STATE &);
  // ...

  // *** DATA MEMBERS ***
  const std::vector<action> actions;
  std::vector<node>     child_nodes;

  node *const parent_node;  // used during backpropagation

  // Score of the associated state from multiple POVs. It's an estimated
  // value based on simulation results.
  std::vector<double> score;

  std::intmax_t visits;  // number of times this node has been visited

  agent_id_t agent_id;   // id of the active agent
};
```

`node` - Expansion

```
node *node::add_child(const STATE &s)
{
  assert(untried_action());                                // not fully expanded
  assert(child_nodes.size() < child_nodes.capacity());  // no reallocation

  child_nodes.emplace_back(s, this);

  return &child_nodes.back();
}
```

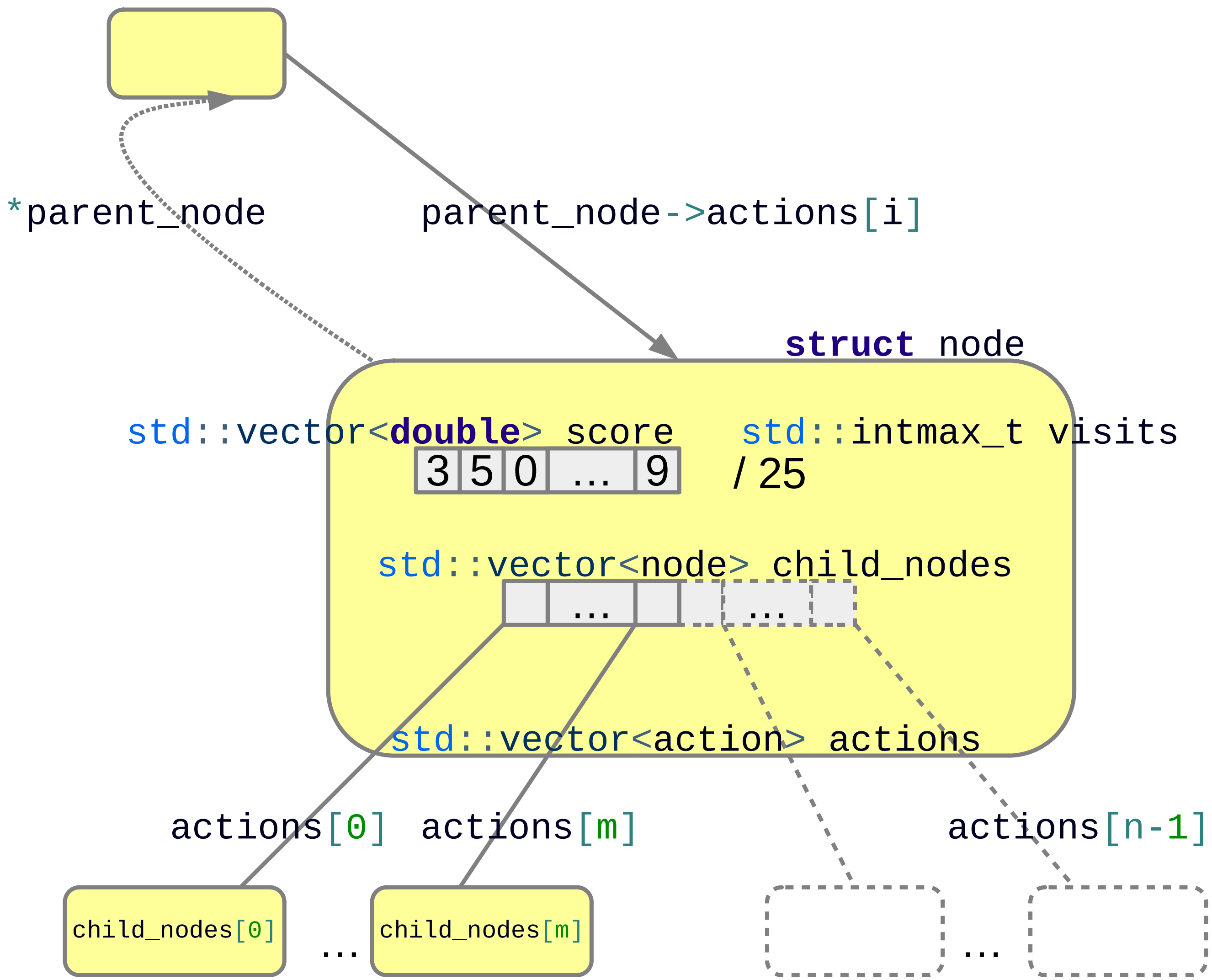
```
node::node(const STATE &state, node *parent_n)
  : actions(state.actions()), child_nodes(), parent_node(parent_n),
    score(), visits(0), agent_id(state.agent_id())
{
  child_nodes.reserve(actions.capacity());  // to avoid reallocation
}
```

MCTS – Skeleton of algorithm (Tree Policy)

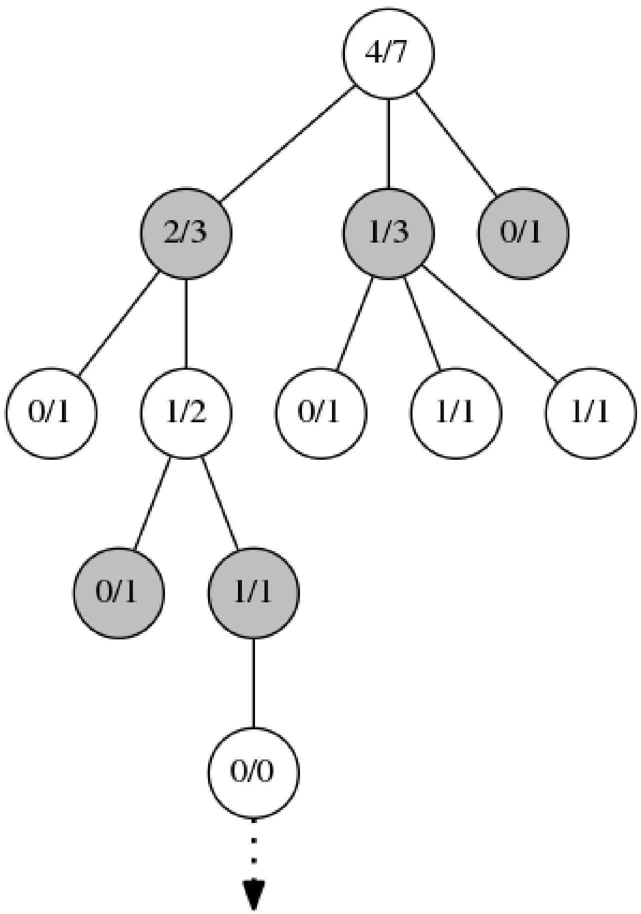
```
while (!stop_request)
{
  node *n(&root_node);
  STATE state(root_state_);

  // Selection.
  while (n->fully_expanded_branch())
  {
    const auto [best_action, best_child] = n->select_child();
    n = best_child;
    state.take_action(best_action);
  }

  // Expansion.
  if (n->untried_action())  // node can be expanded
  {
    state.take_action(*n->untried_action());
    n = n->add_child(state);
  }

  // Simulation (aka Playout / Rollout)...
  // Backpropagation...
  // Polling for stop conditions...
}
```

TREE POLICY

`node` - Graphical representation



Simulation (Monte Carlo)

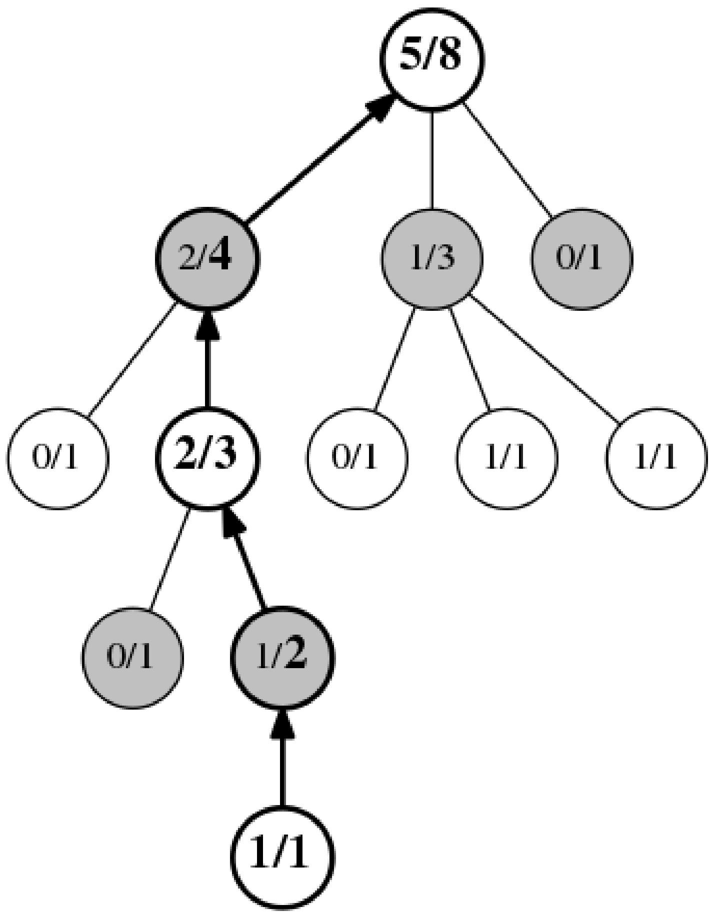


- Run a simulated *rollout* from the new node until a terminal state is found. The terminal state contains a result that will be returned to upwards in the *backpropagation* phase.
- Typical Monte Carlo simulation:
  - either purely random
  - with some simple weighting heuristics if a **light playout** is desired
  - by using some computationally expensive heuristics and evaluations for a **heavy playout**.
- With a lower branching factor, a light playout can give good results.

MCTS – Skeleton of algorithm (3)

```
while (!stop_request)
{
  node *n(&root_node);
  STATE state(root_state_);

  while (n->fully_expanded_branch())    // Selection
  {
    const auto [best_action, best_child] = n->select_child();
    n = best_child;
    state.take_action(best_action);
  }

  if (n->untried_action())              // Expansion
  {
    state.take_action(*n->untried_action());
    n = n->add_child(state);
  }

  // Simulation (aka Playout / Rollout).
  for (auto actions=n->actions; !state.is_final(); actions=state.actions())
    state.take_action(random_element(actions));

  // ...
}
```

Backpropagation



- Occurs when the simulation/playout reaches the end of the game.
- All of the positions visited have their counter incremented and the score updated according to the result of the simulation.

`node` - Data members

```
struct node
{
  explicit node(const STATE &, node * = nullptr);

  std::pair<action, node *> select_child();
  node *add_child(const STATE &);
  void update(const std::vector<double> &);

  bool fully_expanded_branch() const;

  // *** DATA MEMBERS ***
  const std::vector<action> actions;
  std::vector<node>     child_nodes;

  node *const parent_node;  // used during backpropagation

  // Score of the associated state from multiple POVs. It’s an estimated
  // value based on simulation results.
  std::vector<double> score;

  std::intmax_t visits;  // number of times this node has been visited

  agent_id_t agent_id;   // id of the active agent
};
```

`node` - Backpropagation

```
void node::update(const std::vector<double> &sv)
{
  ++visits;

  if (score.empty())
    score = sv;
  else
    std::transform(score.begin(), score.end(), sv.begin(), score.begin(),
                   std::plus());
}
```

MCTS – Skeleton of algorithm (4)

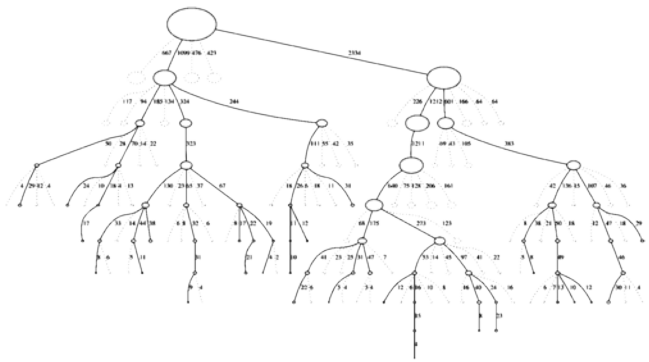
```
while (!stop_request)
{
  node *n(&root_node);
  STATE state(root_state_);

  while (n->fully_expanded_branch())    // Selection
  {
    const auto [best_action, best_child] = n->select_child();
    n = best_child;
    state.take_action(best_action);
  }

  if (n->untried_action())              // Expansion
  {
    state.take_action(*n->untried_action());
    n = n->add_child(state);
  }

  // Simulation (aka Playout / Rollout).
  for (auto actions=n->actions; !state.is_final(); actions=state.actions())
    state.take_action(random_element(actions));

  const auto scores(state.eval());
  for (; n; n = n->parent_node)  // Backpropagation
    n->update(scores);

  // Polling for stop conditions...
}
```

Benefits - Aheuristic

MCTS doesn't require any knowledge about the given domain to make reasonable decisions.

The algorithm can function effectively with no knowledge of a game apart from its legal moves and end conditions; this makes MCTS a potential boon for general game playing.

Benefits - Asymmetric



MCTS performs asymmetric tree growth that adapts to the topology of the search space.

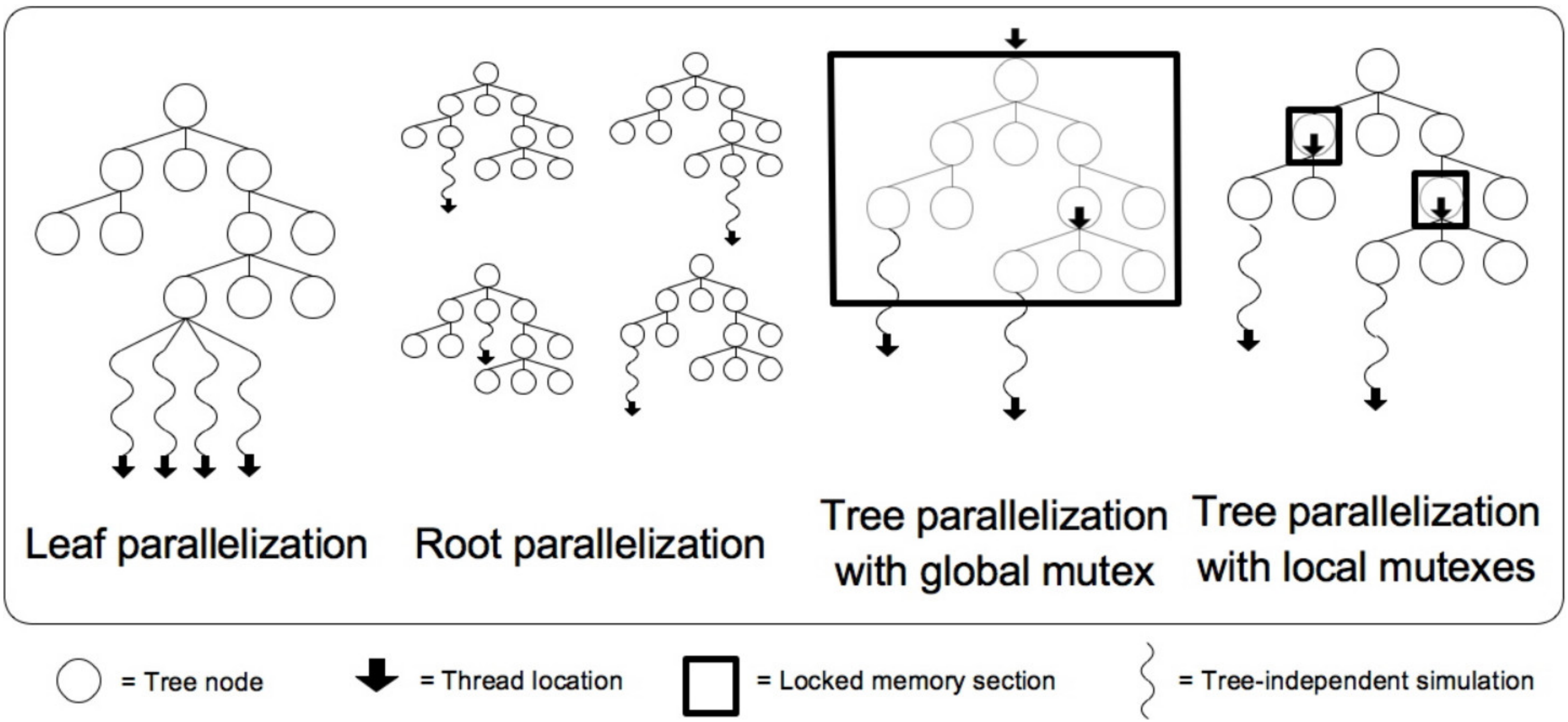
The algorithm visits more interesting nodes more often and focusses its search time in more relevant parts of the tree.

Benefits - AnyTime

The algorithm **can be halted at any time** to return the current best estimate.

The search tree built thus far may be discarded or preserved for future reuse.

Benefits - Simple parallelization



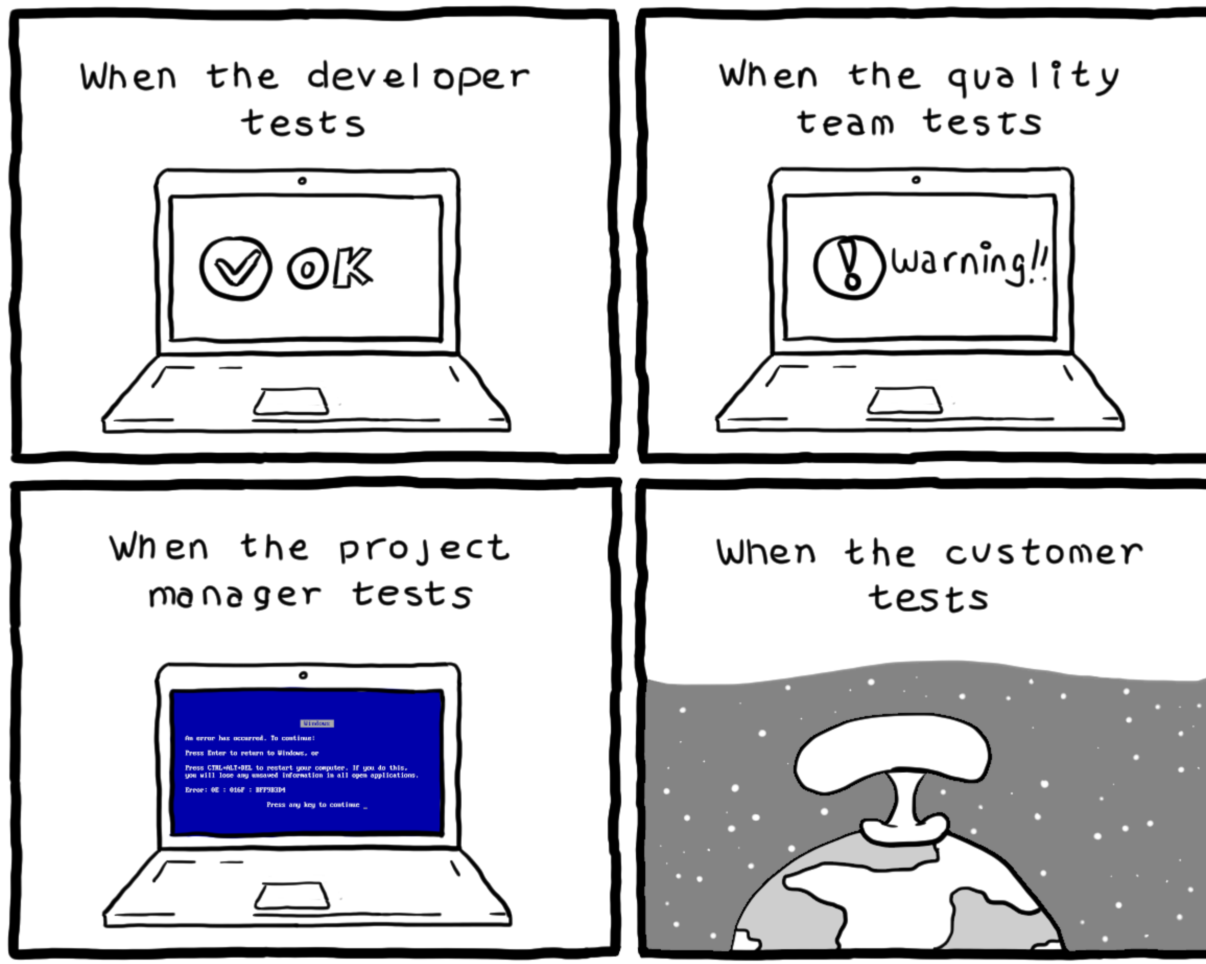
From Parallel Monte-Carlo Tree Search (Chaslot, Winands, van den Herik)

Drawbacks – Slow convergence

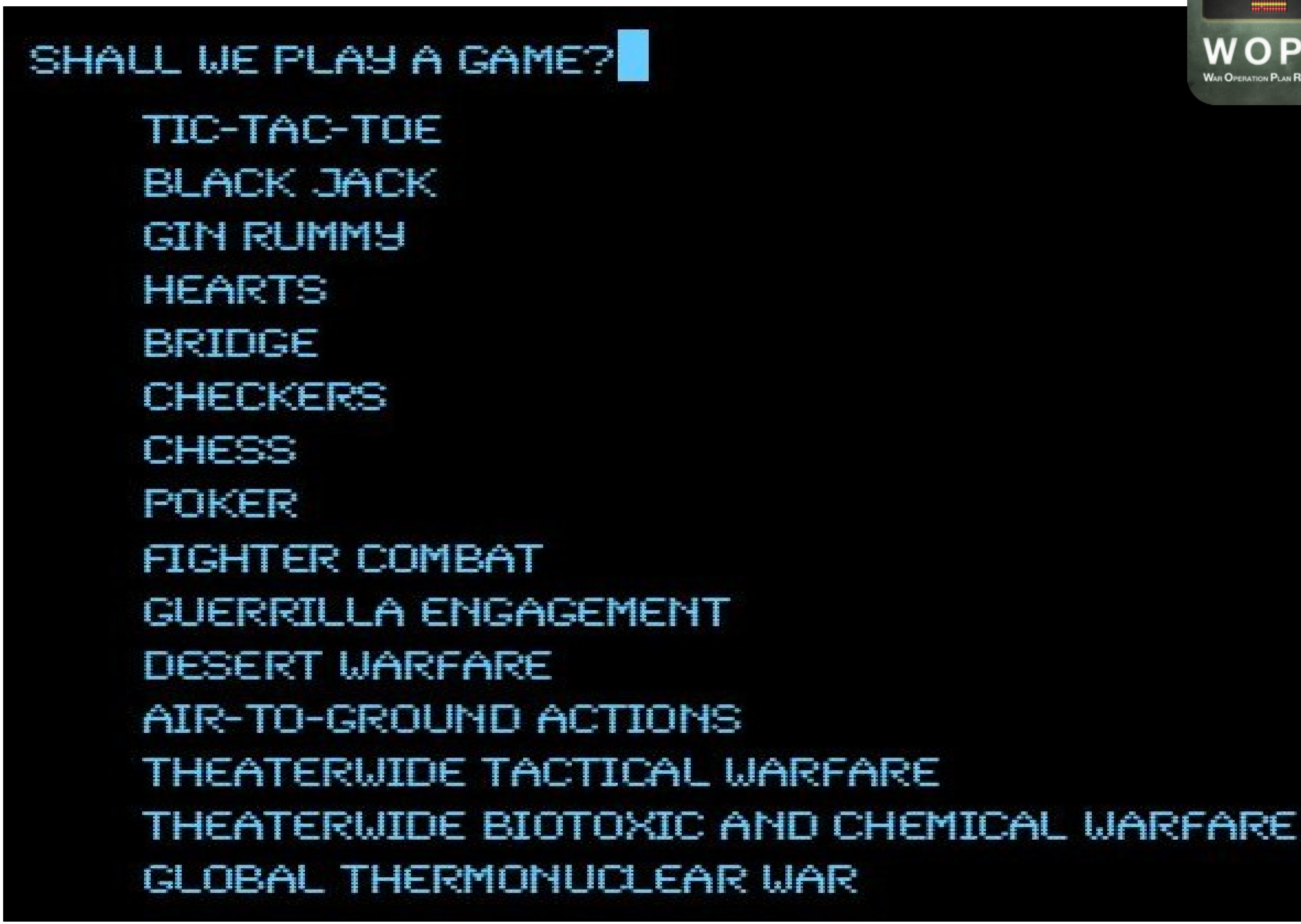
Can fail to find reasonable moves for even games of medium complexity within a reasonable amount of time.
This is mostly due to the sheer size of the combinatorial move space and the fact that key nodes may not be visited enough times to give reliable estimates.

Simple implementation in C++17

https://github.com/morinim/pocket_mcts



QUESTIONS?


SHALL WE PLAY A GAME?
TIC-TAC-TOE
BLACK JACK
GIN RUMMY
HEARTS
BRIDGE
CHECKERS
CHESS
POKER
FIGHTER COMBAT
GUERRILLA ENGAGEMENT
DESERT WARFARE
AIR-TO-GROUND ACTIONS
THEATERWIDE TACTICAL WARFARE
THEATERWIDE BIOTOXIC AND CHEMICAL WARFARE
GLOBAL THERMONUCLEAR WAR

References - Algorithm

- **A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games (volume 4, issue 1, march 2012). DOI: 10.1109/TCIAIG.2012.2186810**
- **Bandit based Monte-Carlo Planning**. Kocsis Levente, Szepesvári Csaba. Machine Learning: ECML 2006, 17th European Conference on Machine Learning. CiteSeerX 10.1.1.102.1296. doi:10.1007/11871842_29. ISBN 3-540-45375-X
- **Finite-time Analysis of the Multiarmed Bandit Problem**. Peter Auer, Nicolò Cesa-Bianchi, Paul Fischer. Machine Learning (volume 47, pp 235-256, 2002)
- **Parallel Monte-Carlo Tree Search**. Guillaume M. J-b. Chaslot , Mark H. M. Win , H. Jaap Van Den Herik (Computers and Games. CG 2008. Lecture Notes in Computer Science, vol 5131).

References – Videogames

- **Monte Mario: platforming with MCTS**. Emil Juul Jacobsen, Rasmus Greve, Julian Togelius (GECCO '14 Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, pp 293-300). DOI: 10.1145/2576768.2598392. Also see https://youtu.be/Xj7-QA-aCus.
- **Curiosity-driven Exploration by Self-supervised Prediction**. Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, Trevor Darrell (Proceedings of the 34th International Conference on Machine Learning, PMLR 70:2778-2787, 2017).
- **Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI**. Alex J. Champandard (2014).

References – Chess, Go, Shogi...

- **A General reinforcement learning algorithm that masters chess, shogi, and Go through self-play**. Science 07 Dec 2018: Vol. 362, Issue 6419, pp. 1140-1144. DOI: 10.1126/science.aar6404
- **Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm**. ArXiv:1712.01815v1 [cs.AI] 5 Dec 2017
- **https://www.chessprogramming.org/Monte-Carlo_Tree_Search**.