# General Game Playing

# Statistical Methods

# Statistical Methods

General Heuristics examined thus far

- Based on properties of small portions of game tree
- e.g. mobility, focus, goal proximity

Statistical Methods

- Based on Statistical analysis
- Many samples of larger portions of game trees

Methods:

- Monte Carlo Search
- Monte Carlo Tree Search (in particular UCT)



In our last lesson, we saw various approaches to incomplete search of game trees. In each approach, the evaluation of states is based on local properties of those states (i.e. properties that do not depend on the game tree as a whole). In many games, there is no correlation between these local properties and the likelihood of success in completing a game successfully. In this lesson, we look at some alternative methods based on statistical analysis of game trees. We first examine a simple approach based on Monte Carlo game simulation; and we then we look at a more sophisticated variation called Monte Carlo Tree Search.

# Monte Carlo Search

Basic Idea

(1) Explore game graph to some level storing states

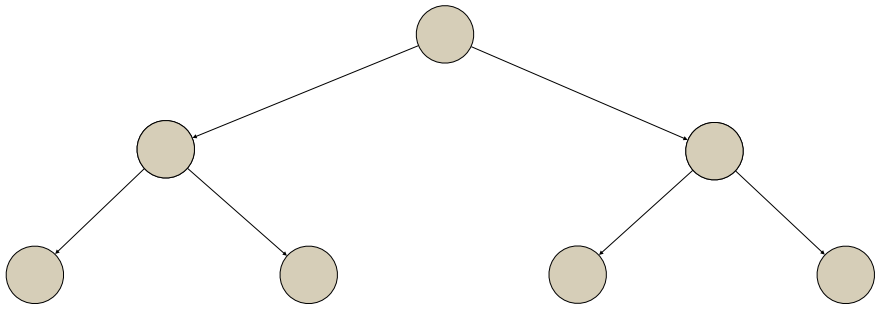
(2) Beyond this, explore to end of game
making random choices for moves of all players,
not storing states (to limit space growth)

(3) Assign expected utilities to states by
summing utilities and dividing by number of trials



The basic idea of Monte Carlo Search (MCS) is simple. As with depth-limited search, we explore the game tree to some fixed depth. In order to estimate the value of a non-terminal state at this depth, we make some probes from that state to the end of the game by selecting random moves for the players. We sum up the total reward for all such probes and divide by the number of probes to obtain an estimated utility for that state. We can then use these expected utilities in comparing states and selecting actions.

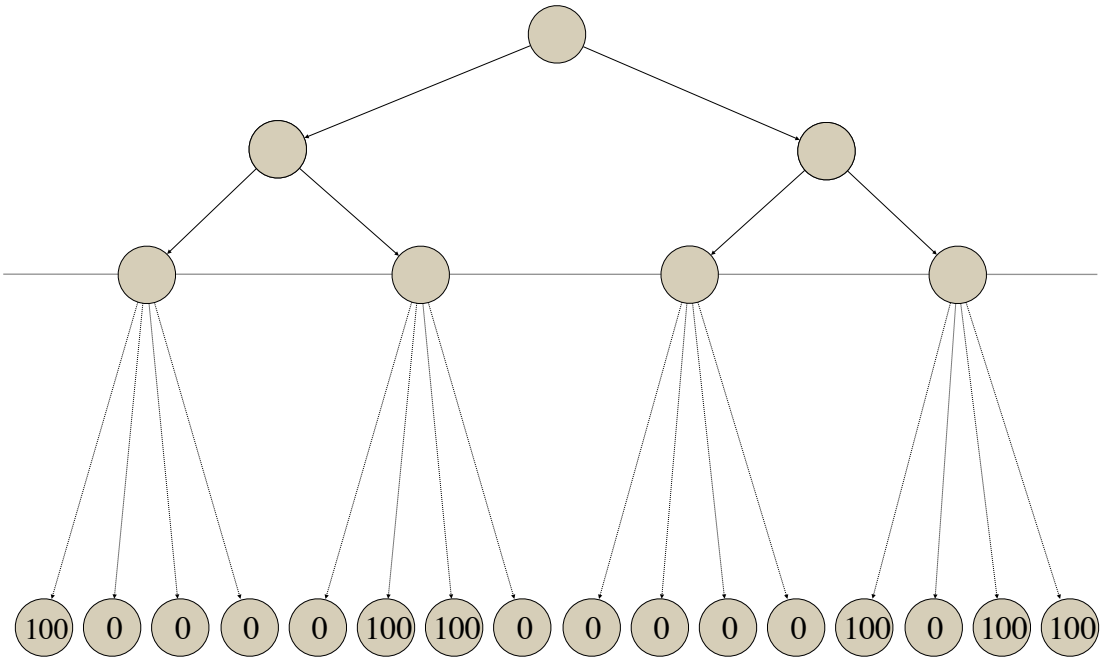
# Example





As just mentione, the expansion phase of Monte Carlo is the same as depth-limited search. The tree is explored until some fixed depth is reached.

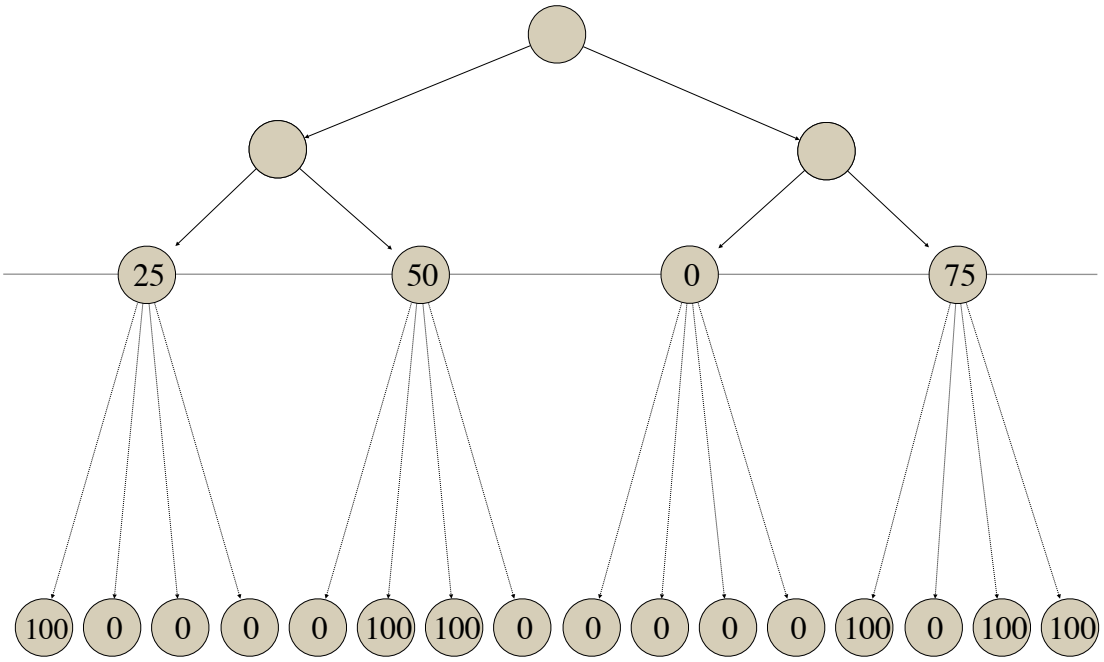
# Example






The probe phase of Monte Carlo takes the form of exploration from each of the fringe states reached in the expansion phase, for each making random probes from there to a terminal state.

# Example






The values produced by each probe are added up and divided by the number of probes for each state to obtain an expected utility for that state. These expected utilities are the compared to determine the relative utilities of the fringe states produced at the end of the expansion phase.

# Implementation

```
function maxscore (state,level)
 {if (findterminalp(state,ruleset))
     {return findreward(role,state,ruleset)};
  if (level>levels) {return montecarlo(state)};
  var actions = findlegals(role,state,ruleset);
  var score = 0;
  for (var i=0; i<actions.length; i++)
      {var result = minscore(actions[i],state,level);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

A simple implementation of maxscore for Monte Carlo Search is shown here. The method is the same as ordinary fixed-depth heuristic search except that the player uses the montecarlo routine to evaluate states.

# Implementation

```
function montecarlo (state)
 {var total = 0;
  for (var i=0; i<count; i++)
      {total = total + depthcharge(state)};
  return total/count}

function depthcharge (state)
 {if (findterminalp(state,ruleset))
     {return findreward(role,state,ruleset)};
  var move = seq();
  for (var i=0; i<roles.length; i++)
      {var options = findlegals(roles[i],state,library);
       var best = randomindex(options.length);
       move[i] = options[best]};
  var newstate = findnexts(move,state,library);
  return depthcharge(newstate)}
```

The montecarlo routine takes a state as argument and returns the average utility obtained from a set of n probes (here called depth charges) where n is the value of the global parameter count. The depthcharge subroutine first checks if a state is terminal. If so, it returns that value. Otherwise, it forms a joint move by taking random legal actions of all of the players. It then simulates the state and calls itself recursively and returns the result. The recursion terminates when the player encounters a terminal state.

# Problems and Features

Problems

- Optimistic - opponent might not respect probabilities



One downside on the Monte Carlo method is that it can be optimistic. It assumes all players are playing randomly when in fact it is possible that they know exactly what they are doing. It does not help if most of the probes from a position in Chess lead to success if one leads to a state in which one's player is checkmated and the the other player sees this. This issue is addressed to some extent in the UCT method described below.

# Problems and Features

Problems

- Optimistic - opponent might not respect probabilities
- Does not utilize game structure in any useful way



Another drawback of the Monte Carlo method is that it does not take into account the structure of a game. For example, it may not recognize symmetries or independences that could substantially decrease the cost of search. For that matter, it does not even recognize boards or pieces or piece count or any other features that might form the basis of game-specific heuristics.

# Problems and Features

Problems

- Optimistic - opponent might not respect probabilities
- Does not utilize game structure in any useful way

Features

- Fast because no search
- Small space because nothing stored in probes



Even with these drawbacks, the Monte Carlo method is quite powerful. It is fast and consumes little space. And it is surprisingly effective. Prior to its use, general game players were at best interesting novelties. Once players started using Monte Carlo, the improvement in game play was dramatic. Suddenly, automated general game players began to perform at a high level. Using a variation of this technique, CadiaPlayer won the International General Game Playing competition 3 times. Almost every general game playing program today includes some version of Monte Carlo Search.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

# Monte Carlo Tree Search

# Basic Idea

Monte Carlo Tree Search (MCTS) is a search method that relies on random probes to estimate state values. Similar to MCS but different in an important way.

Similarity:

- Both build up game tree incrementally
- Both use random probes to estimate state values

Main Difference:

- MCS expands tree uniformly
- MCTS expands nonuniformly based on statistics



Monte Carlo Tree Search (MCTS) is a more sophisticated variation of Monte Carlo Search that tackles some of the weaknesses of the simpler method. Both methods build up a game tree incrementally and both rely on random simulation of games; but they differ on the way the tree is expanded. MCS uniformly expands the partial game tree during its expansion phase and then simulates games starting at states on the fringe of the expanded tree. MCTS uses a more sophisticated approach in which the processes of expansion and simulation are interleaved.

# Monte Carlo Tree Search

**Selection**

Player selects an unexpanded node of the tree
chooses based on visit counts and stored utilities

**Expansion**

successors of the selected state added to the tree

**Simulation**

players simulates the game from selected node to end
chooses actions at random at each node along the way

**Backpropagation**

Value propagated back to the root node
visit counts and utilities are updated accordingly

MCTS processes the game tree in cycles of four steps each. After each cycle is complete, it repeats these steps so long as there is time remaining, at which point it selects an action based on the statistics it has accumulated to that point.
On the selection step, the player traverses the tree produced thus far to select an unexpanded node of the tree, making choices based on visit counts and utilities stored on nodes in the tree.
During expansion, the successors of the state chosen during the selection phase are added to the tree.
The player simulates the game starting at the node chosen during the selection phase. In so doing, it chooses actions at random until a terminal state is encountered.
Finally, the value of the terminal state is propagated back along the path to the root node and the visit counts and utilities are updated accordingly.

# Selection

```
function select (node)
 {if (node.visits==0) {return node};
  for (var i=0; i<node.children.length; i++)
      {if (node.children[i].visits==0)
          {return node.children[i]}};
  score = 0;
  result = node;
  for (var i=0; i<node.children.length; i++)
      {var newscore = selectfn(node.children[i]);
       if (newscore>score)
          {score = newscore; result=node.children[i]}};
  return select(result)}
```

An implementation of the MCTS selection procedure is shown below. If the initial state has not been seen (i.e. it has 0 visits), then it is selected. Otherwise, the procedure searches the successors of the node. If any have not been seen, then one of the unseen nodes is selected. If all of the successors have been seen before, then the procedure uses the selectfn subroutine to find values for those nodes and chooses the one that maximizes this value.

# Selection Function

Upper Confidence Bounds on Trees (UCT)

$$v_i + sqrt(log(n_p) / n_i)$$

Implementation

```
function selectfn(node)
 {var vi = node.utility;
  var np = node.visits;
  var ni = node.parent.visits;
  return vi + Math.sqrt(Math.log(np)/ni)}
```

One of the most common ways of implementing selectfn is UCT (Upper Confidence bounds applied to Trees). A typical UCT formula is vi + sqrt(log np / ni). vi here is the average reward for that state. np is the total number of times the state's parent was picked. ni is the number of times this particular state was picked.  Of course, there are other ways that one can evaluate states. The formula here is based on a combination of exploitation and exploration. Exploitation here means the use of results on previously explored states (the first term). Exploration means expansion of as-yet unexplored states (the second term).

# Expansion

```
function expand (node)
 {var actions = findlegals(role,node.state,ruleset);
  for (var i=0; i<actions.length; i++)
      {var newstate = findnexts([actions[i]],state,ruleset);
       var newnode = makenode(newstate,0,0,node,[]);
       node.children[node.children.length]=newnode};
  return true}
```

Expansion in MCTS is basically the same as that for MCS. An implementation for a single player game is shown below. On large games with large time bounds, it is possible that the space consumed in this process could exceed the memory available to a player. In such cases, it is common to use a variation of the selection procedure in which no additional states are added to the tree. Instead, the player continues doing simulations and updating its numbers for already-known states.

# Simulation

```
function simulate (state)
 {if (findterminalp(state,ruleset))
     {return findreward(role,state,ruleset)};
  var move = seq();
  for (var i=0; i<roles.length; i++)
      {var options = findlegals(roles[i],state,library);
       var best = randomindex(options.length);
       move[i] = options[best]};
  var newstate = findnexts(move,state,library);
  return depthcharge(newstate)}
```

Simulation for MCTS is essentially the same as simulation for MCS. So the same procedure can be used for both methods.

# Backpropagation

```
function backpropagate (node,score)
 {node.visits = node.visits+1;
  node.utility = node.utility+score;
  if (node.parent) {backpropagate(node.parent,score)};
  return true}
```

Backpropagation is easy. At the selected node, the method records a visit count and a utility. The visit count in this case is 1 since it was a newly processed state. The utility is the result of the simulation. The procedure then propagates to ancestors of this node. In the case of a single player game, the procedure adds 1 to the visit count of each ancestor and augments its total utility by the utility obtained on the latest simulation. In the case of a multiple player game, the propagated value is the minimum of the values for all opponent actions.


GENERAL
GAME
PLAYING


Celebrating AAAI's 25th
Anniversary