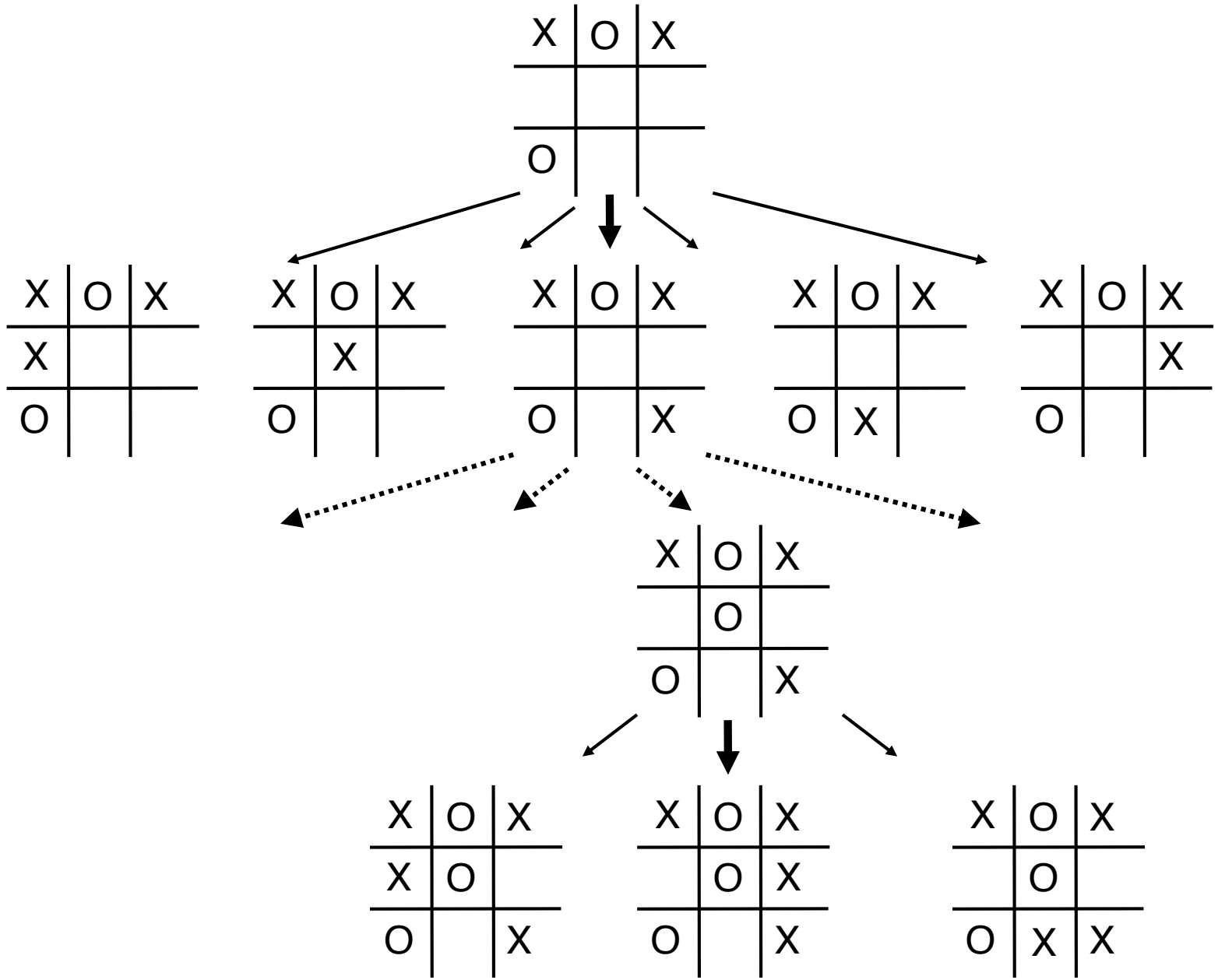
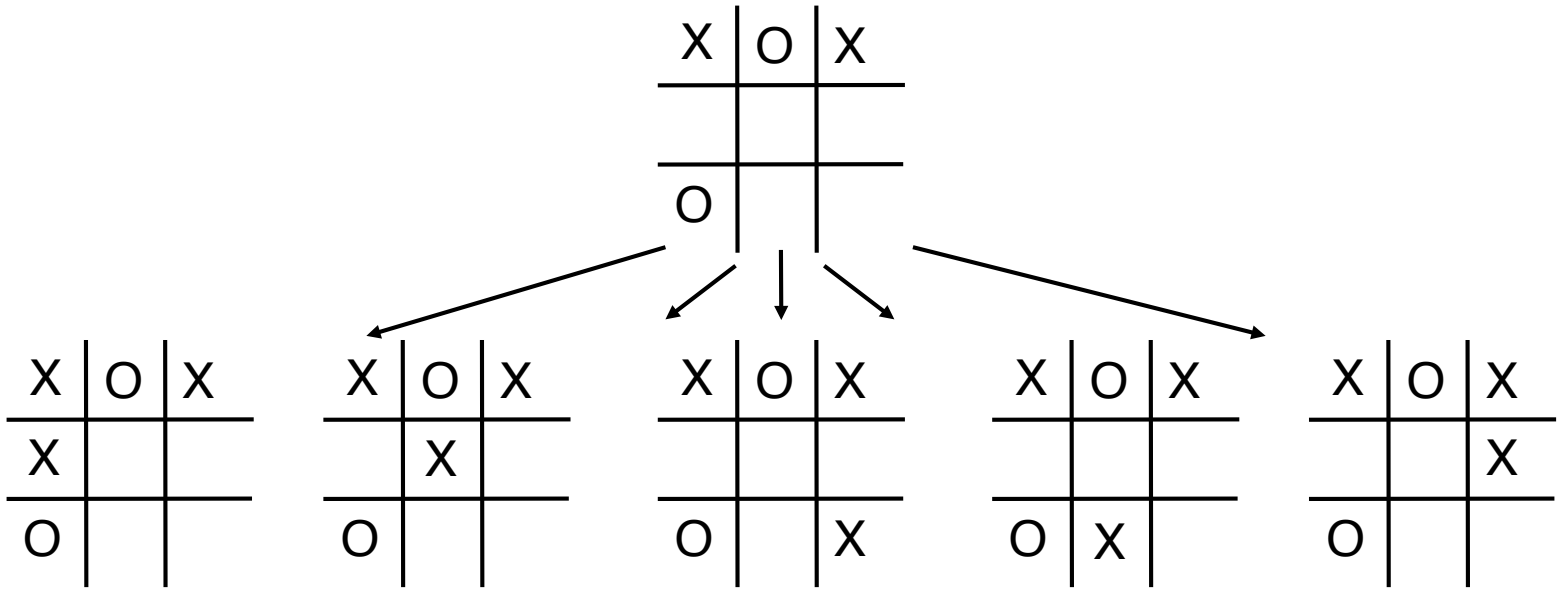
General Game Playing
Incomplete Search

Michael Genesereth
Computer Science Department
Stanford University

Complete Game Graph Search



Incomplete Search



Bounded Depth Minimax

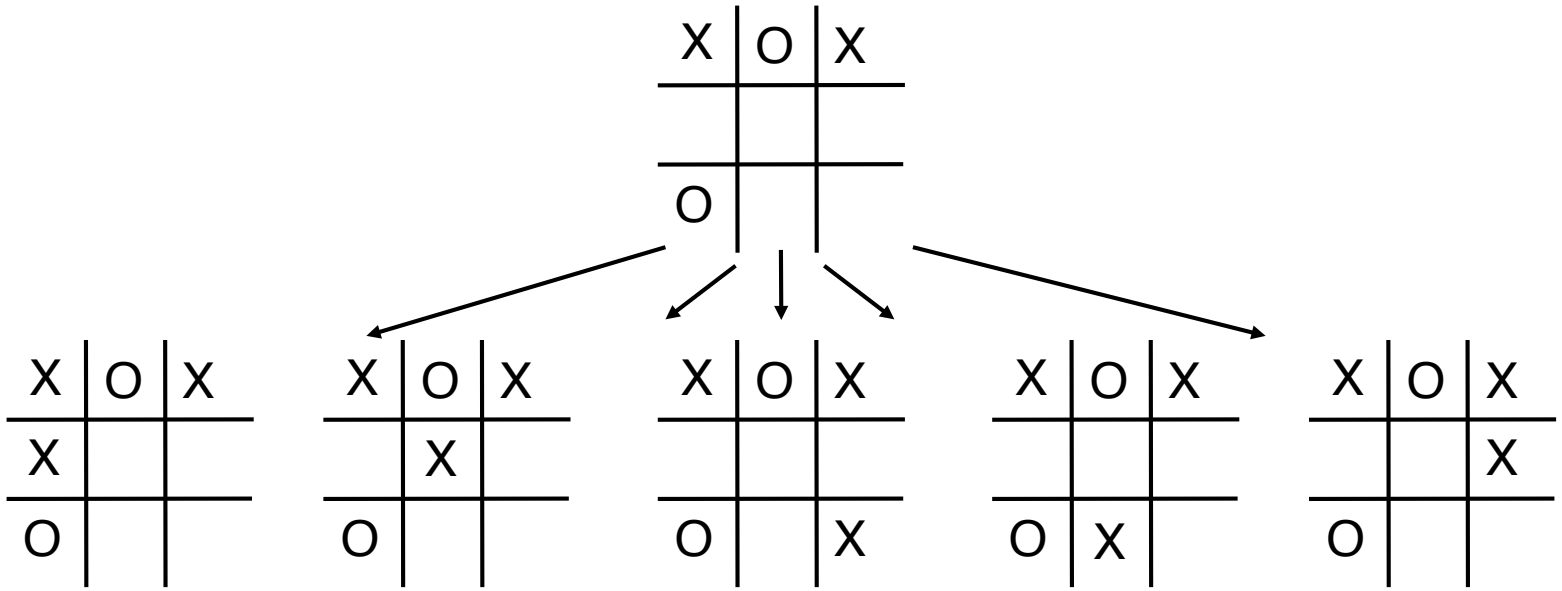
Minimax:

```
function minimax (state)
 {if (findterminalp(state,library))
     {return findreward(role,state,library)*1};
  var active = findcontrol(state,library);
  if (active===role) {return maximize(state)};
  return minimize(state)}
```

Bounded Depth Minimax

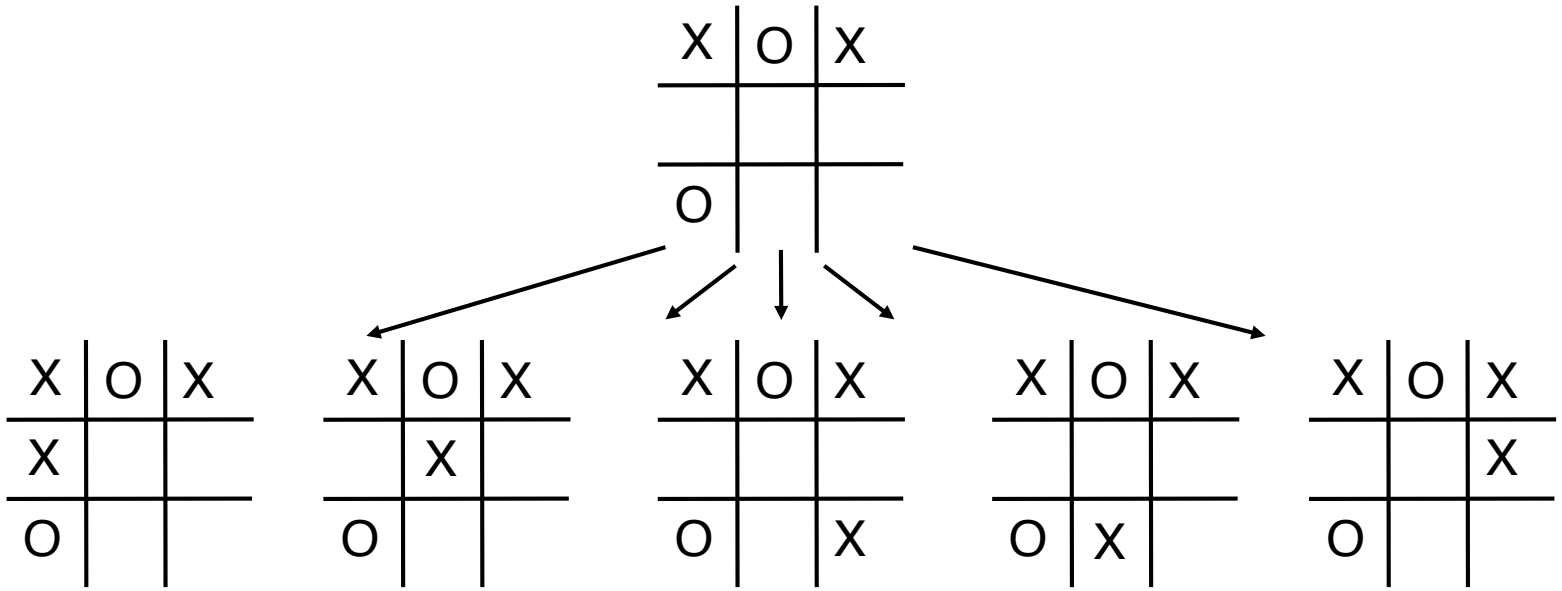
```
function minimaxdepth (state,depth)
 {if (findterminalp(state,library))
     {return findreward(role,state,library)*1};
  if (depth<=0) {return evalfun(state,library)*1};
  var active = findcontrol(state,library);
  if (active===role) {return maxscore(state,depth)};
  return minscore(state,depth)}
```

Evaluation of Non-Terminal States



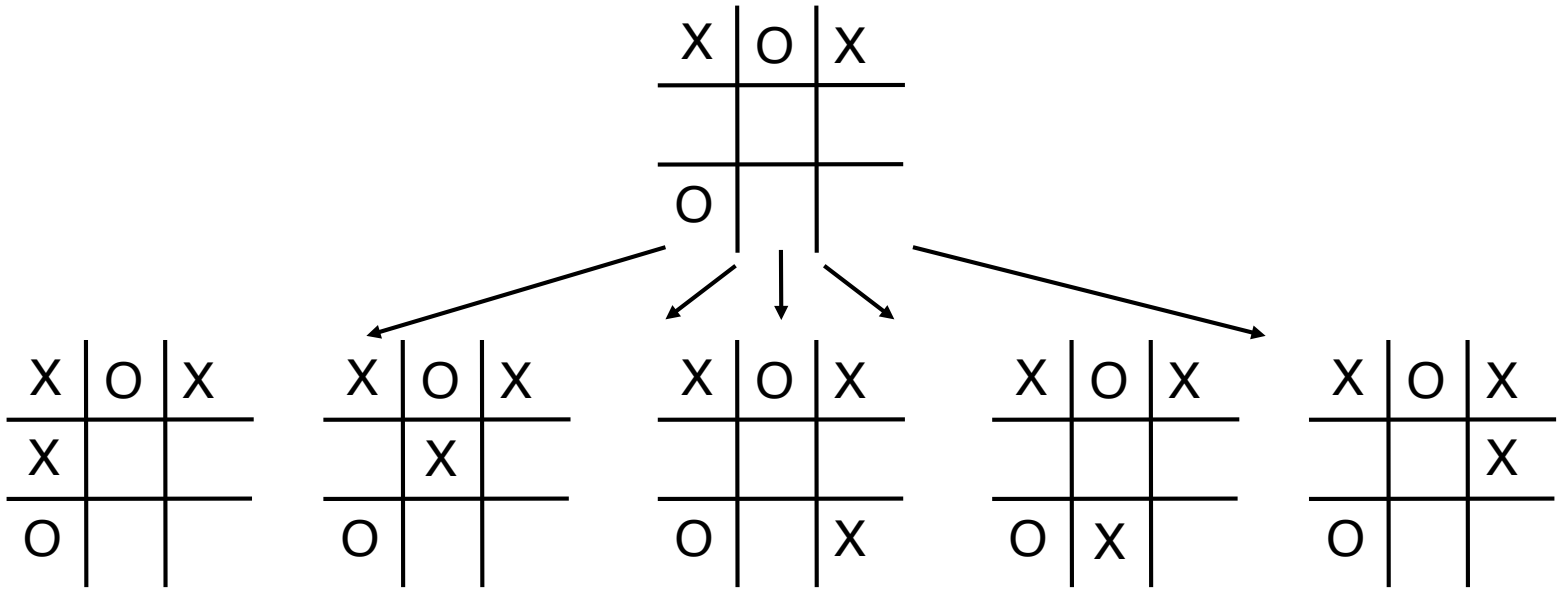
How do we evaluate non-terminal states?

Choice of Depth



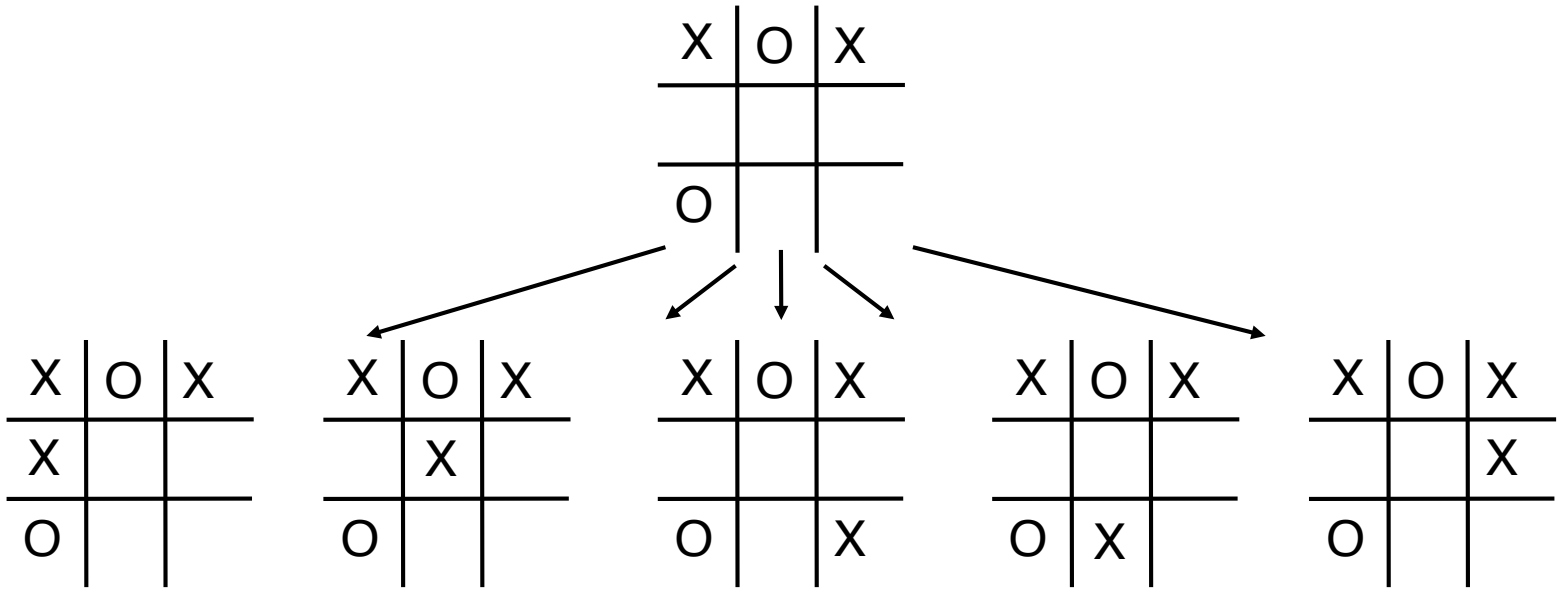
To what depth should we search?

Variable Depth Search



Should we search different branches to different depths?

Persistence



Can we preserve results across moves?

Evaluation Functions

Evaluation Functions

Chess examples:

- Piece count
- Board control

Comments

- Not *necessarily* successful
- Game-specific but this is *general* game playing

Heuristic #1 - Mobility / Focus

Mobility is a measure of the number of things a player can do. *Focus* is a measure of the narrowness of the search space. It is the inverse of mobility.

Basis - number of actions in a state or number of states reachable from that state. Horizon - current state or *n* moves away.

Sometimes it is good to focus to cut down on search space. Often better to restrict opponents' moves while keeping one's own options open.

Heuristic #1 - Mobility / Focus

Mobility is a measure of the number of things a player can do. *Focus* is a measure of the narrowness of the search space. It is the opposite of mobility.

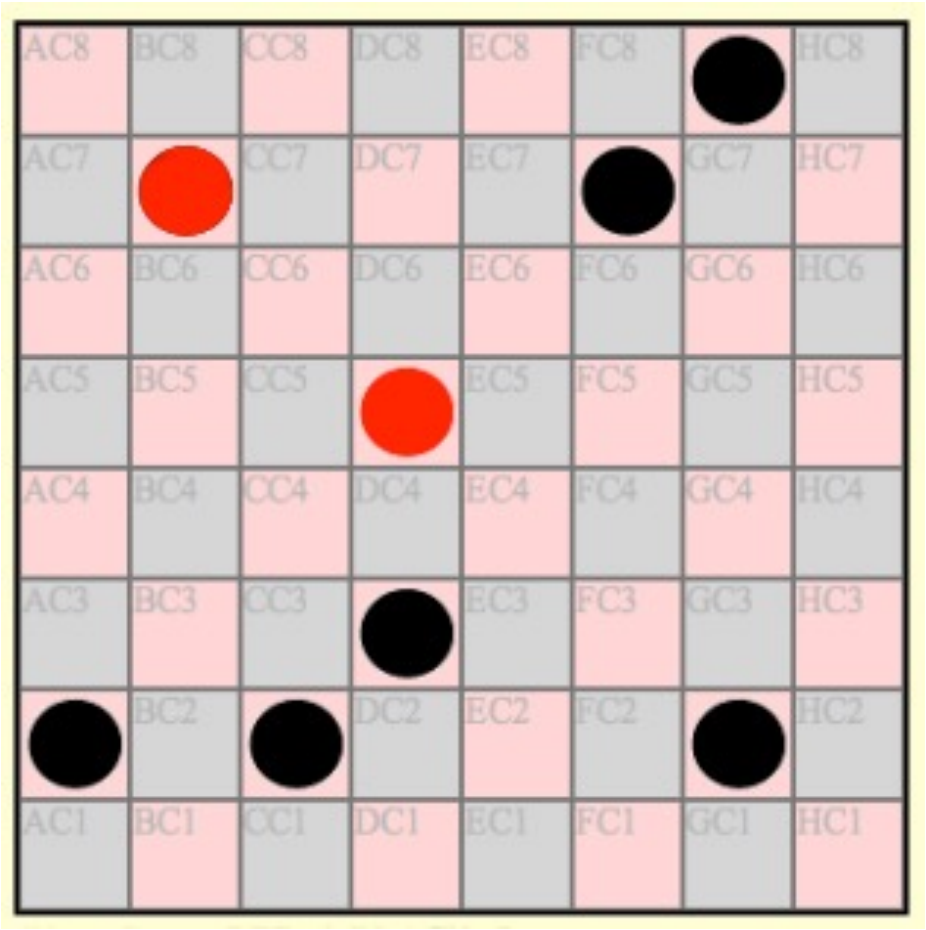
Basis - number of actions in a state or number of states reachable from that state. Horizon - current state or *n* moves away.

Sometimes it is good to focus to cut down on search space. Often better to restrict opponents' moves while keeping one's own options open.

Implementation

```
function mobility (state)
 {var actions = findlegals(state,library);
  var feasibles = findactions(library);
  return (actions.length/feasibles.length * 100)}

function focus (state)
 {var actions = findlegals(state,library);
  var feasibles = findactions(library);
  return (100 - actions.length/feasibles.length * 100)}
```

GGP-06 Final - Cylinder Checkers

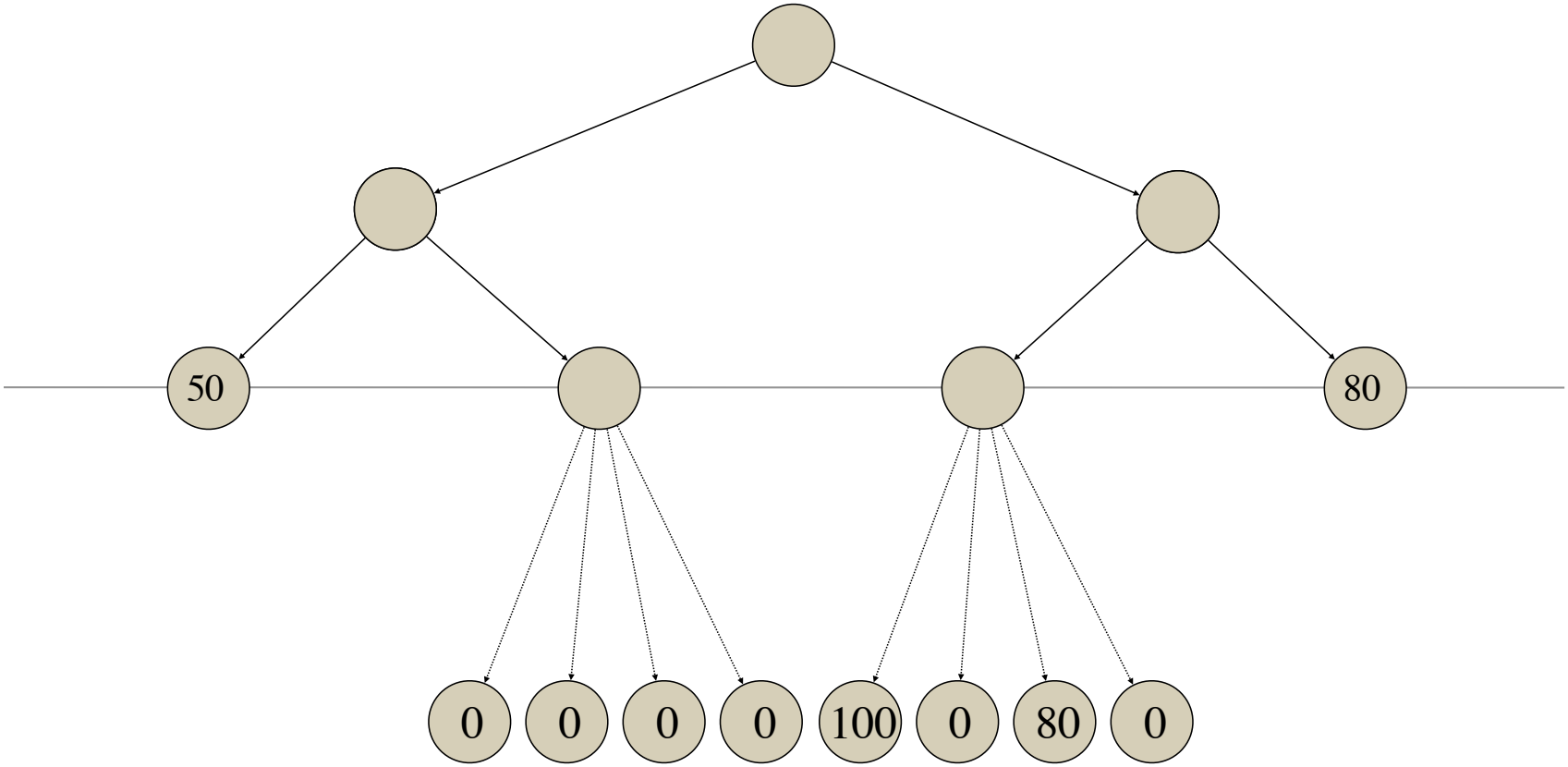


Heuristic #2 - Pessimism

Assume value of 0 for non-terminal states.

$$value(state) = goal(role,state) \text{ if } terminal(state)$$
$$value(state) = 0 \quad \text{otherwise}$$

Example



Heuristic #3 - Intermediate Values

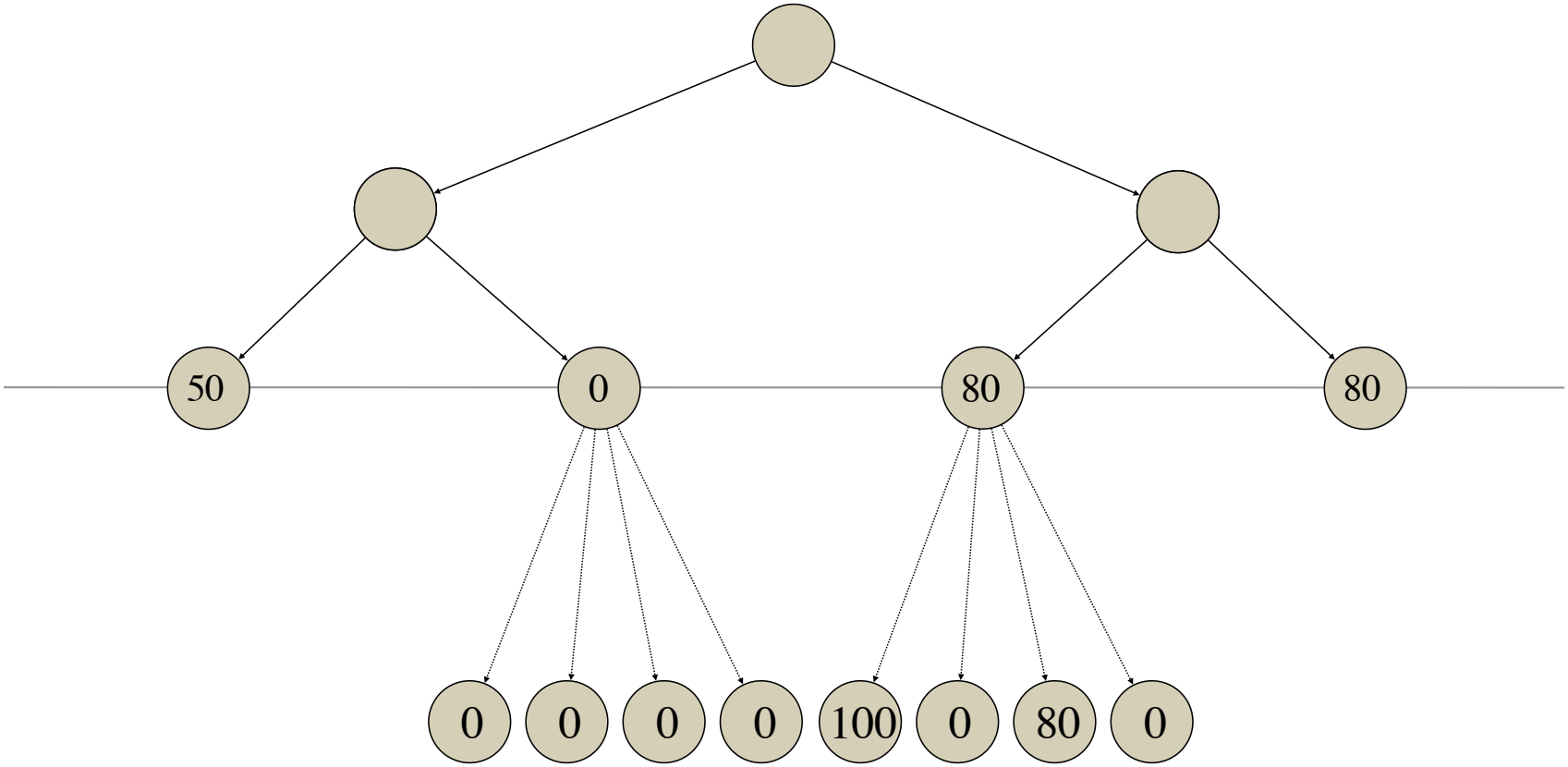
Assume reward for non-terminal states.

$$value(state) = goal(role,state)$$

Good on monotonic games (where utility accumulates as the game progresses), e.g. alquerque.

Not so good on nonmonotonic games. Susceptible to "false summits".

Example



Heuristic #4 - Statistics

Sample a few branches of the game tree and use results to estimate values.

More on this next time.

Weighted Linear Combinations

Definition

$$f(s) = w_1 \times f_1(s) + \ldots + w_n \times f_n(s)$$

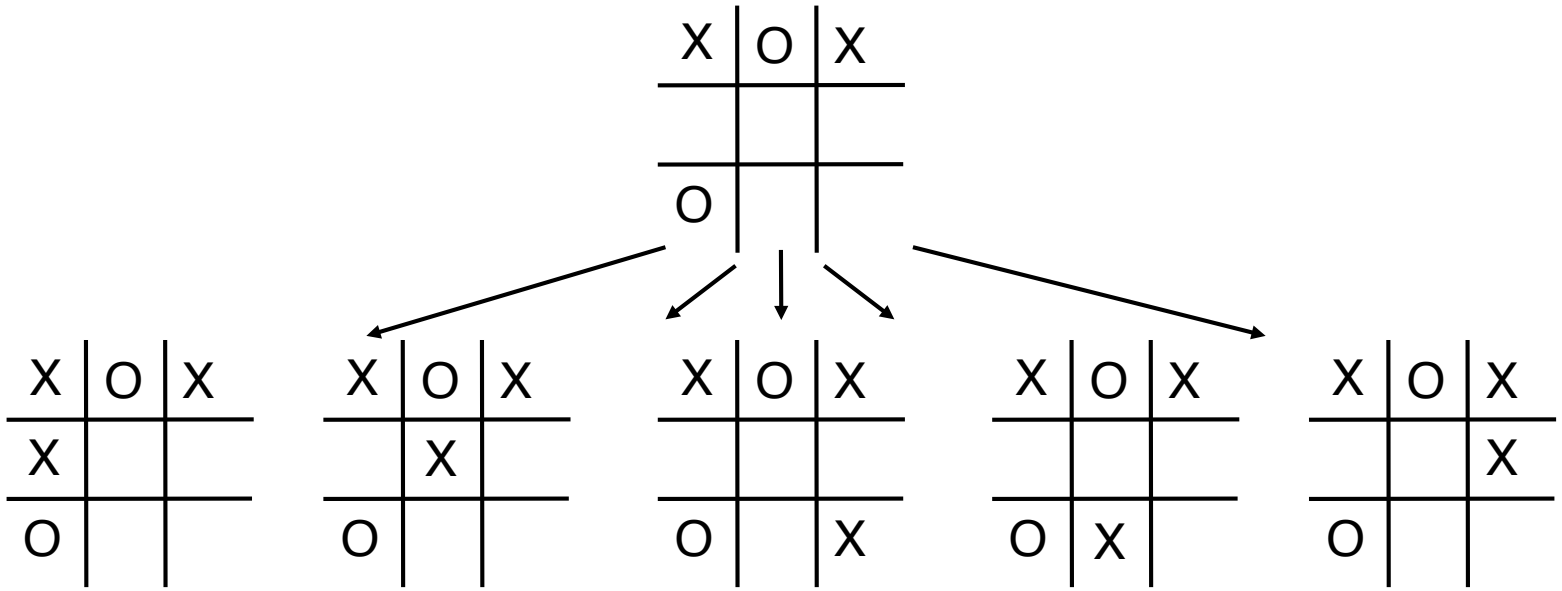
Examples:

- Mobility / Focus
- Intermediate State Values
- Statistics

Some players estimate weights by experimentation during the start clock. *More on this later.*

Basic Search Strategies

Incomplete Search



Technique #1 - Depth-Limited Search

Idea - search tree to some depth-limit

Legal and random players are degenerate depth-limited search procedures with depth 0.

Works well for ragged game trees (games with differing length branches).

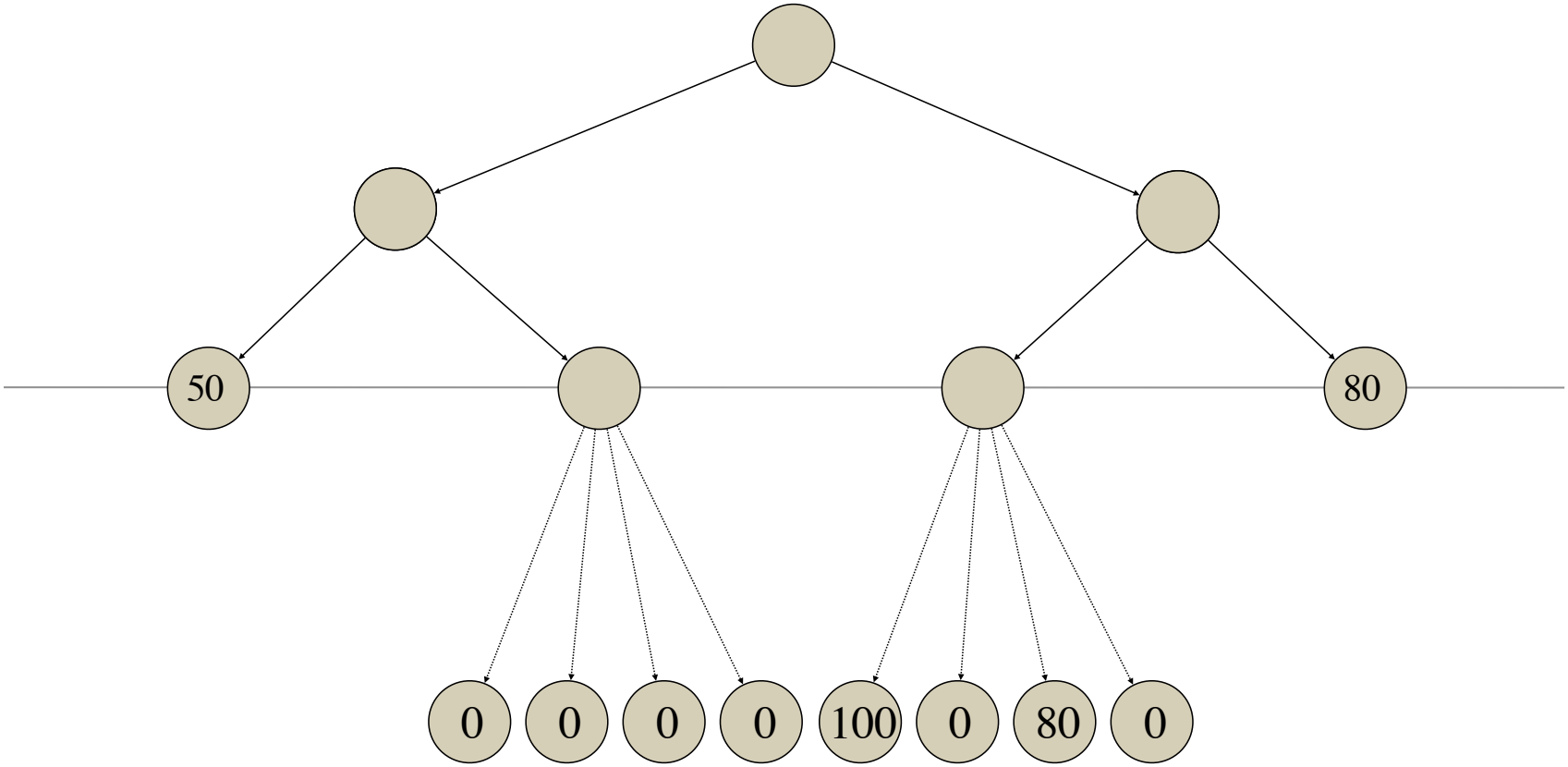
Implementation

```
function playminimaxdepth (depth)
 {var actions = shuffle(findlegals(state,library));
  var action = actions[0];
  var score = 0;
  for (var i=0; i<actions.length; i++)
      {var newstate = simulate(actions[i],state,library);
       var newscore = minimaxdepth(newstate,depth);
       if (newscore===100) {return actions[i]};
       if (newscore>score)
          {action = actions[i]; score = newscore}};
  return action}

function minimaxdepth (state,depth)
 {if (findterminalp(state,library))
     {return findreward(role,state,library)*1};
  if (depth<=0) {return findreward(role,state,library)*1};
  if (findcontrol(state,library)===role)
     {return maxscore(state,depth)};
  return minscore(state,depth)}
```

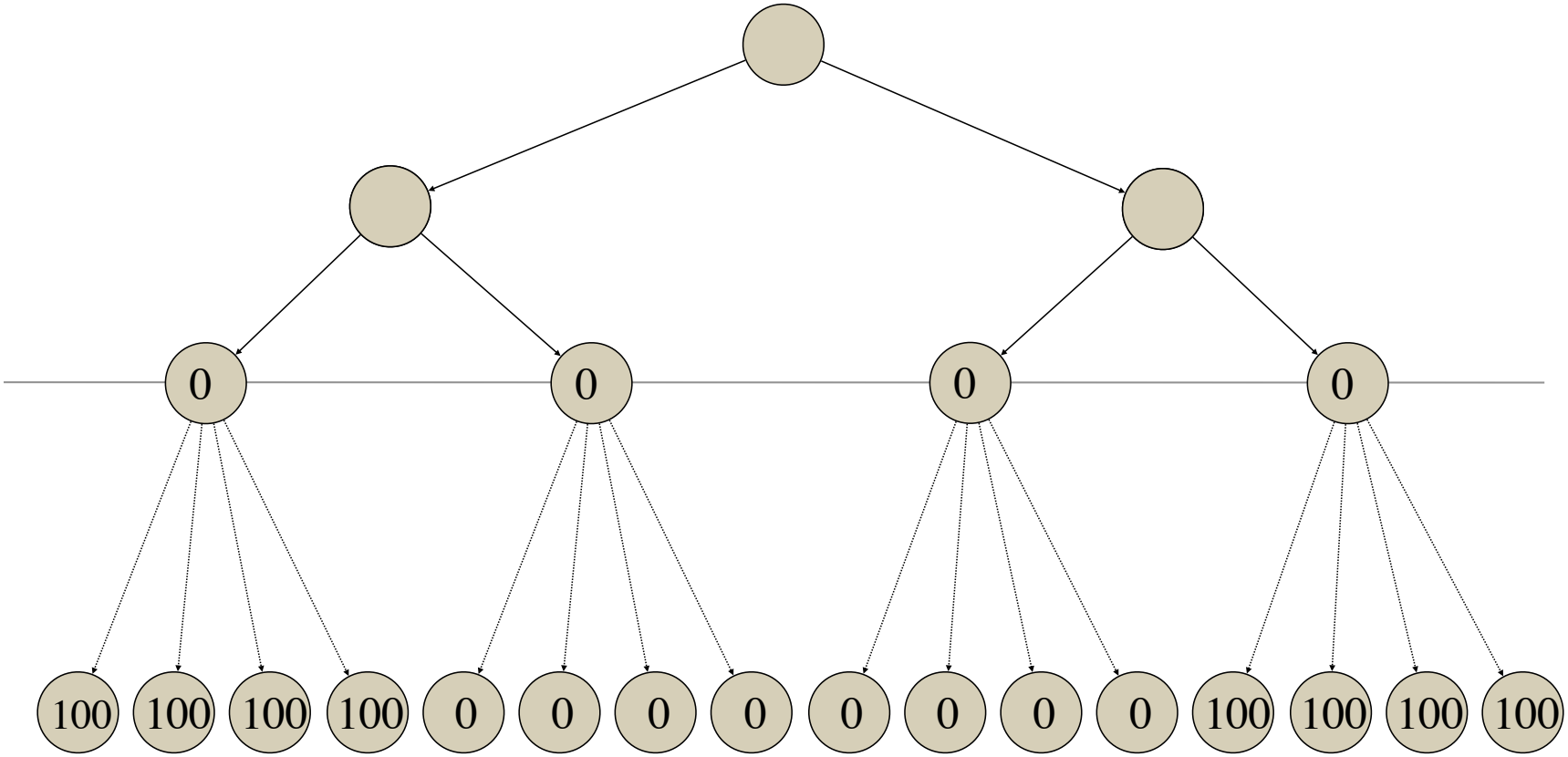
maxscore and minscore

```
function maxscore (state,depth)
 {var actions = findlegals(state,library);
  if (actions.length===0) {return 0};
  var score = 0;
  for (var i=0; i<actions.length; i++)
      {var newstate = simulate(actions[i],state,library);
       var newscore = minimaxdepth(newstate,depth-1);
       if (newscore===100) {return 100};
       if (newscore>score) {score = newscore}};
  return score}

function minscore (state,depth)
 {var actions = findlegals(state,library);
  if (actions.length===0) {return 0};
  var score = 100;
  for (var i=0; i<actions.length; i++)
      {var newstate = simulate(actions[i],state,library);
       var newscore = minimaxdepth(role,newstate,depth-1);
       if (newscore===0) {return 0};
       if (newscore<score) {score = newscore}};
  return score}
```

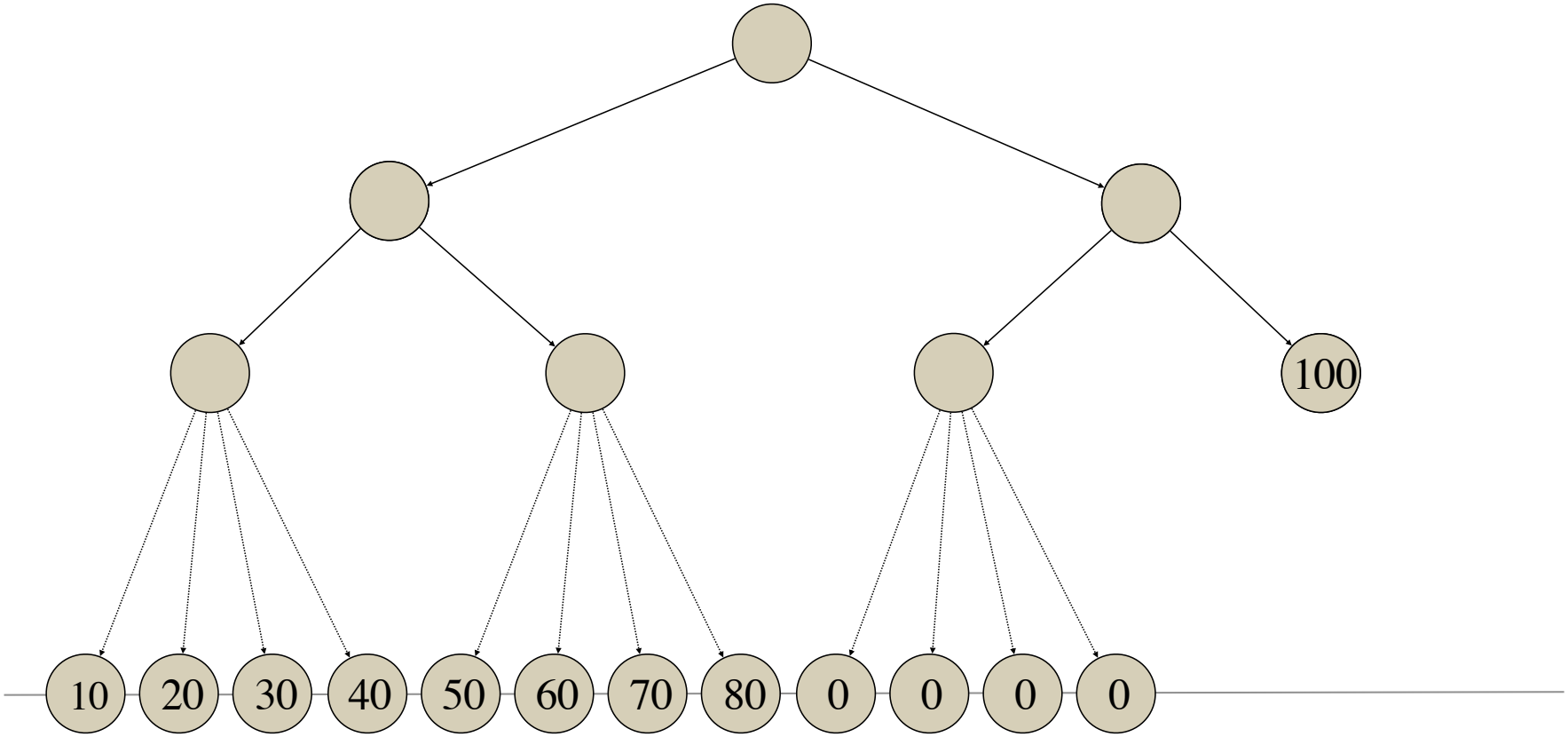
Example



Problem - Insufficient Depth



Problem - Excessive Depth



Time Management

If we expand to some arbitrary fixed depth, may run out of time or may not utilize available time.

Technique #2 - Iterative Deepening

Use depth-limited search to explore entire tree to level 1
Use depth-limited search to explore entire tree to level 2
Use depth-limited search to explore entire tree to level 3
And so forth

Continue till time runs out
Choose action that gives maximal value

Implementation

```
function playminimaxid ()
 {var deadline = Date.now()+(playclock-2)*1000;
  var best = findlegalx(state,library);
  for (var depth=1; depth<25; depth++)
      {var action = simpleminimaxid(state,depth,deadline);
       if (action===false) {return best};
       best = action};
  return best}
```

Implementation

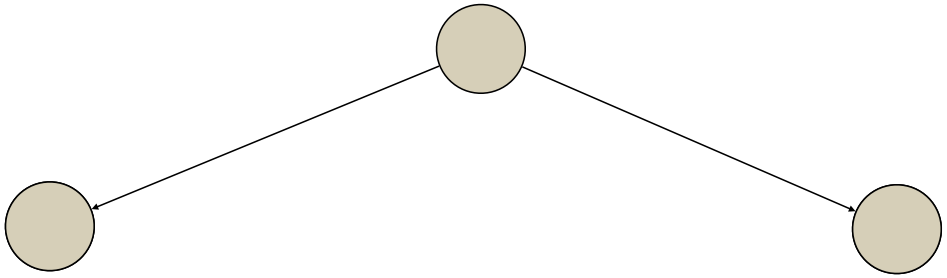
```
function simpleminimaxid (state,depth,deadline)
 {var actions = shuffle(findlegals(state,library));
  var best = actions[0];
  var score = 0;
  for (var i=0; i<actions.length; i++)
      {var newstate = simulate(actions[i],state,library);
       var newscore = minimaxid(newstate,depth,deadline);
       if (newscore===false) {return false};
       if (newscore===100) {return actions[i]};
       if (newscore>score) {best = actions[i]; score=newscore}};
  return best}
```

Implementation

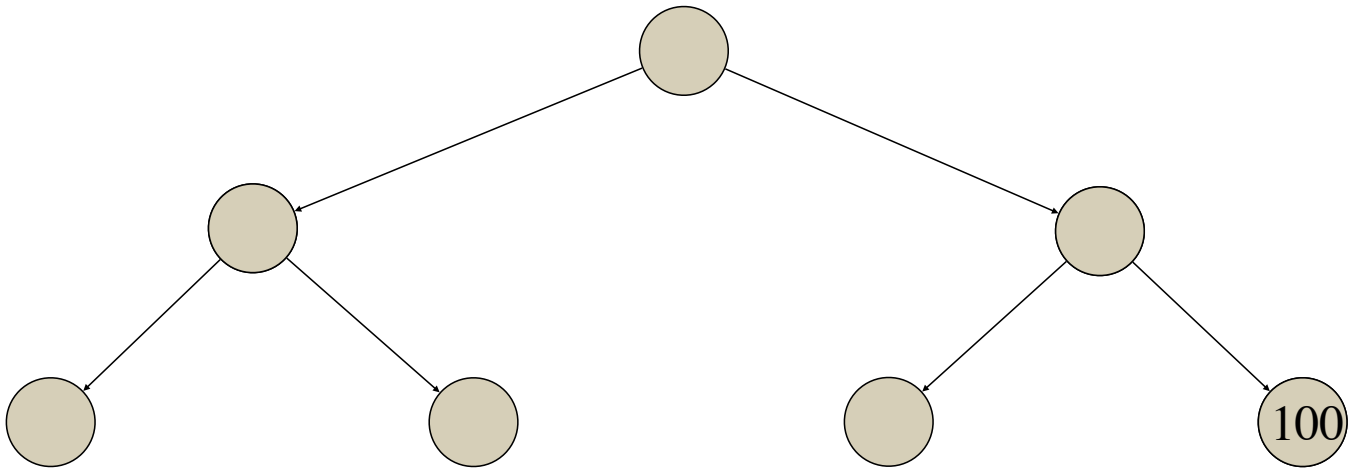
```
function minimaxid (state,depth,deadline)
 {if (findterminalp(state,library))
     {return findreward(role,state,library)*1};
  if (depth<=0) {return evalfun(state,library)*1};
  if (Date.now()>deadline) {return false};
  if (findcontrol(state,library)===role)
     {return maxscoreid(state,depth,deadline)};
  return minscoreid(state,depth,deadline)}
```

Level 1

Level 2



Level 3



Advantages and Disadvantages

Advantages

requires storage linear in depth

still finds shortest path to an optimal solution

Disadvantages (?)

Repeated work

but

Cost only a constant factor more than depth-first search

Why? Tree is growing exponentially, so fringe of tree and size of tree above fringe are approximately same

More Information

https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Advanced Search Strategies

Problems

Horizon Problem

white gains a rook but loses queen or loses game

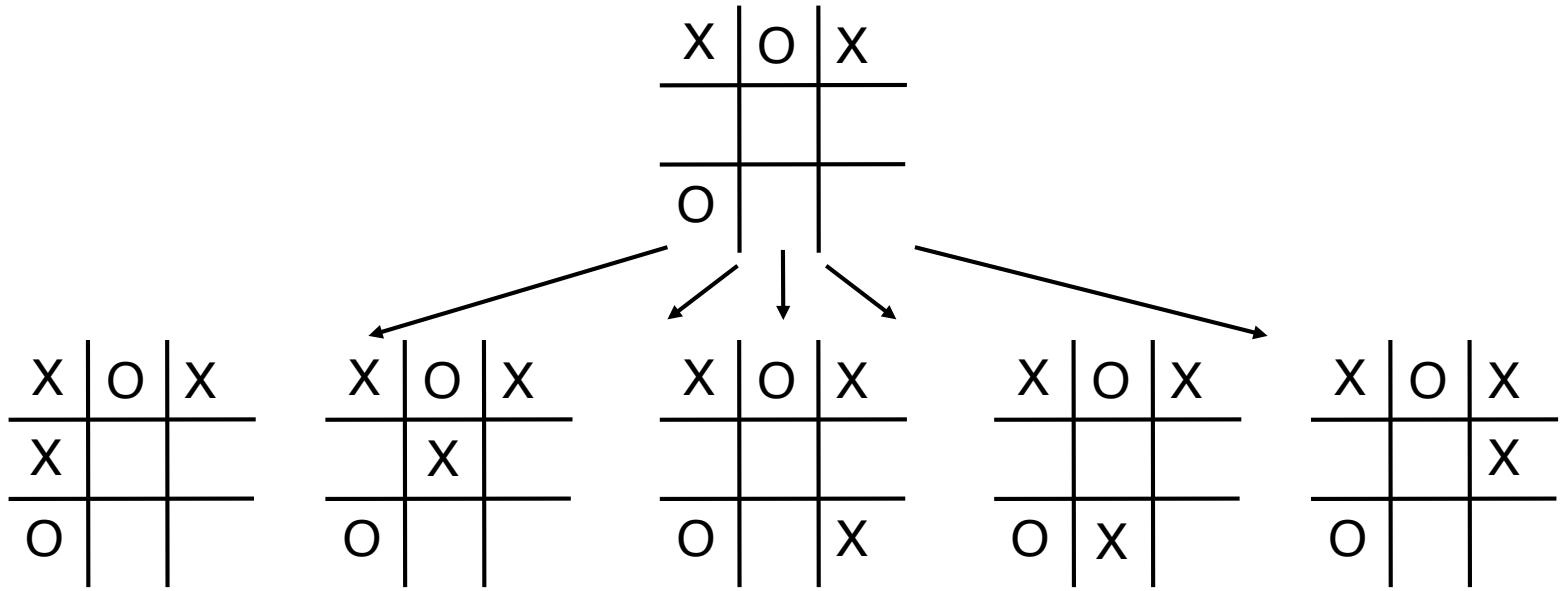
example - sequence of captures in chess

Wasted work

search results not preserved between moves

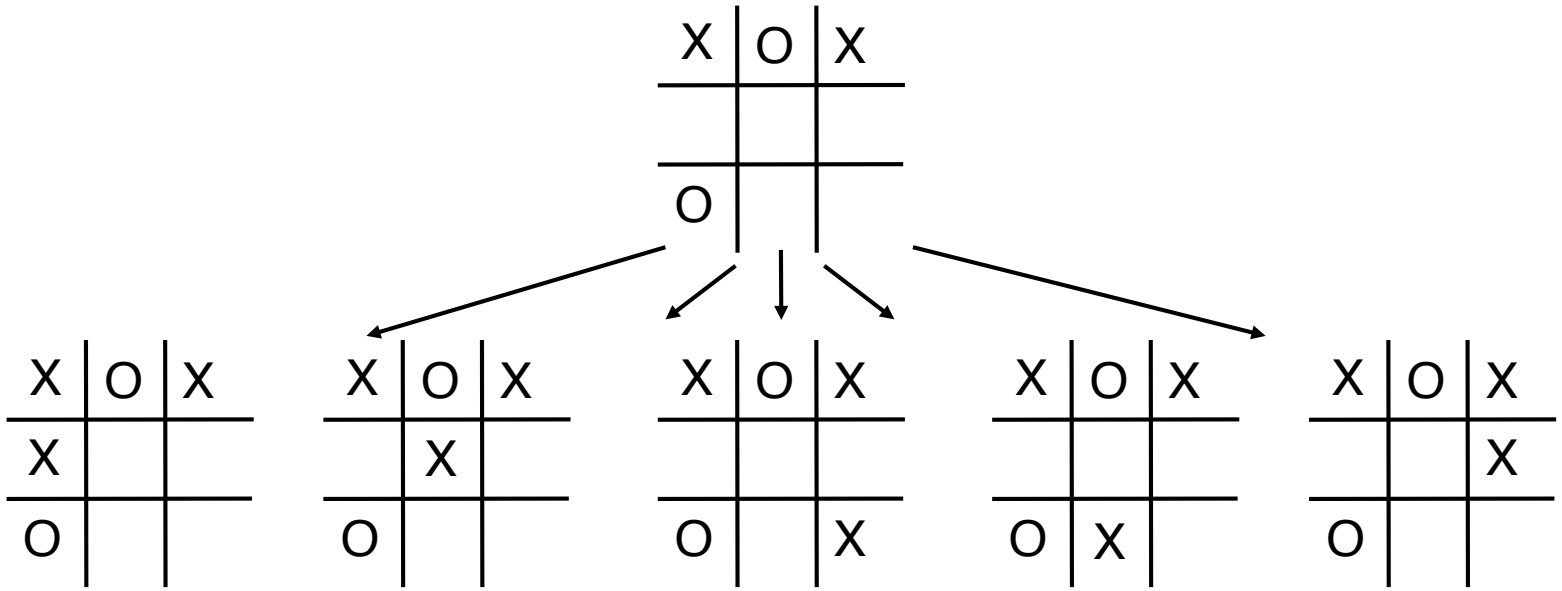
in most cases necessitating repeated computation

Variable Depth Search



Can we search different branches to different depths?

Persistence



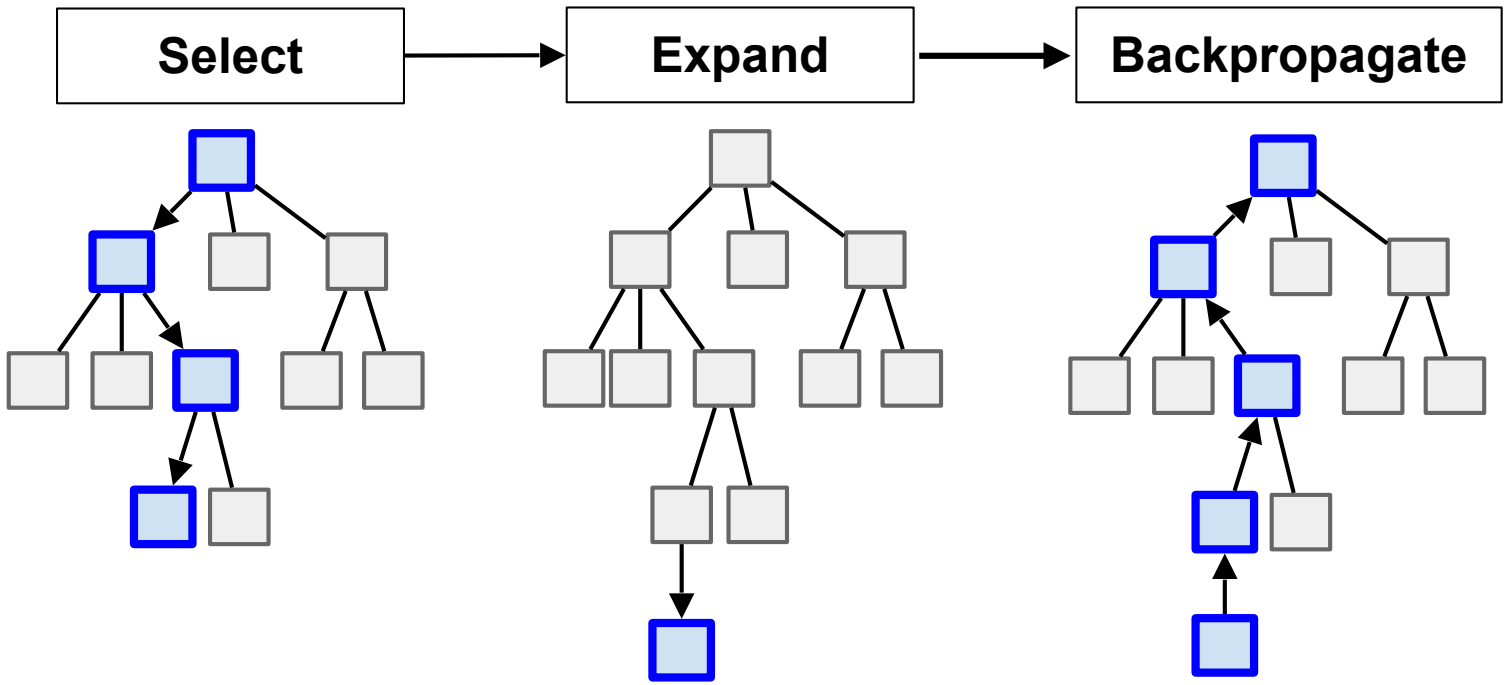
How do we preserve tree across moves?

Technique #3 - Greedy

On each cycle, *select* best node to *expand* based on
- estimated utility
- number of visits
- standard deviation of successors
- etc.

Store tree
Update incrementally while time allows
Replace with subtree after each move made

Overview



start

```
var role, roles, state, library, startclock, playclock;
var tree = {};

function start (r,rs,sc,pc)
 {role = r;
  library = definemorerules([],rs.slice(1));
  roles = findroles(library);
  state = findinits(library);
  startclock = sc;
  playclock = pc;
  var newreward = parseInt(findreward(role,state,library));
  tree = makenode(state,newreward);
  return 'ready'}

function makenode (state,reward)
 {return {state:state,
          actions:[],
          children:[],
          visits:0,
          utility:reward}}
```

play

```
function play (move)
 {if (move!==nil) {updatetree(move)};
  if (findcontrol(state,library)!==role) {return false};
  return playgreedy(role)}
```

playgreedy

```
function playgreedy (role)
 {var deadline = Date.now()+(playclock-2)*1000;
  while (Date.now()<deadline) {process(tree)};
  return selectaction(tree)}

function selectaction (node)
 {var action = node.actions[0];
  var score = node.children[0].utility;
  for (var i=1; i<node.children.length; i++)
      {var newscore = node.children[i].utility;
      if (newscore>score)
         {action = node.actions[i]; score = newscore}};
  return action}
```

process

```
function process (role,node)
 {var newscore = 0;
  if (findterminalp(node.state,library))
     {newscore = node.utility}
     else if (node.children.length===0)
             {newscore = expand(role,node)}
     else {newscore = process(role,selectstate(node))};
  node.utility = Math.max(newscore,node.utility);
  node.visits = node.visits+1;
  return node.utility}
```

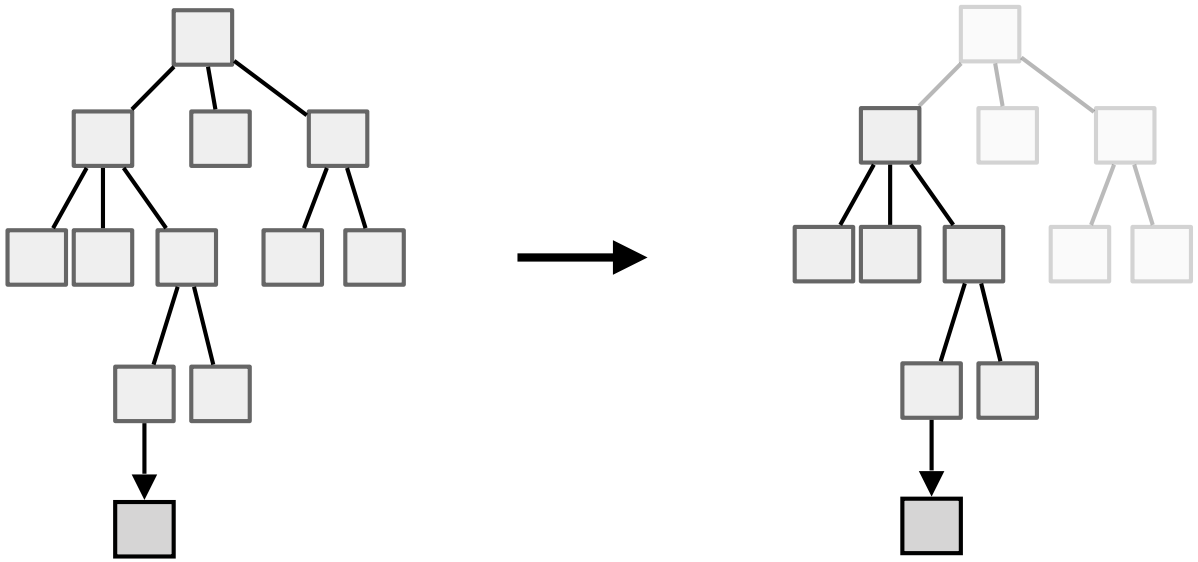
selectstate

```
function selectstate (node)
 {var best = node.children[0];
  var score = best.utility -
                Math.floor(best.visits*100/node.visits);
  for (var i=1; i<node.children.length; i++)
      {var exploitation = node.children[i].utility;
       var exploration =
           Math.floor(node.children[i].visits*100/node.visits);
       var newscore = exploitation - exploration;
       if (newscore>score)
          {best = node.children[i]; score = newscore}};
  return best}
```

expand

```
function expand (role,node)
 {node.actions = findlegals(node.state,library);
  var score = 0;
  for (var i=0; i<node.actions.length; i++)
      {var newstate = simulate(node.actions[i],node.state,library);
       var newscore = parseInt(findreward(role,newstate,library));
       var newnode = makenode(newstate,newscore);
       node.children[i]=newnode;
       if (newscore>score) {score = newscore}};
  return score}
```

Updating the Tree



update

```
function play (move)
 {if (move!==nil) {updatetree(move)};
  if (findcontrol(state,library)!==role) {return false};
  return playgreedy(role)}

function updatetree (move)
 {if (tree.children.length===0)
     {var newstate = simulate(move,tree.state,library);
      var newscore = findreward(role,newstate,library)*1);
      return makenode(newstate,newscore)};
  for (var i=0; i<tree.actions.length; i++)
      {if (equalp(move,tree.actions[i]))
          {return tree.children[i]}}
  return tree}
```

Problem

This implementation does not distinguish between values that are *guaranteed* (because the player searched to the end of the tree) from those that are *estimated* (based on heuristic evaluation function).

Fix by adding additional information to each node.


GENERAL
GAME
PLAYING