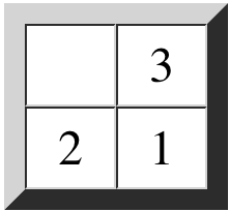
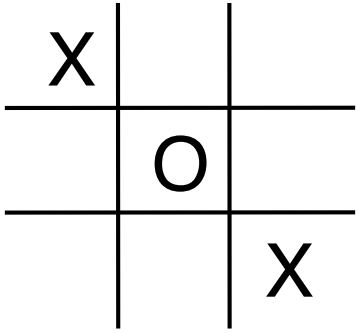
General Game Playing
Incomplete Game Tree Search

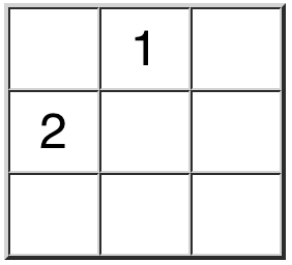
Michael Genesereth
Computer Science Department
Stanford University

Game Variety

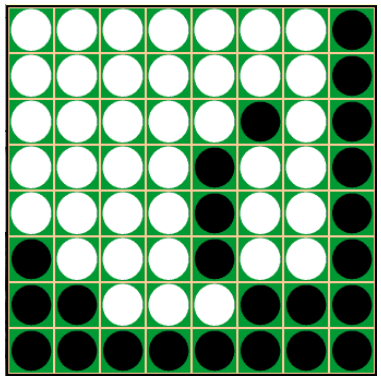
Small Games

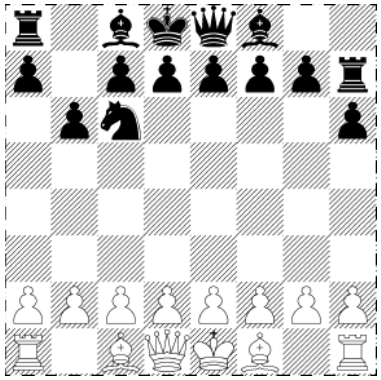


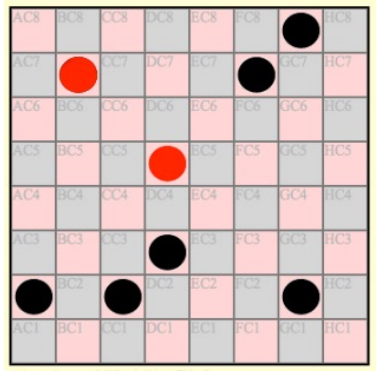




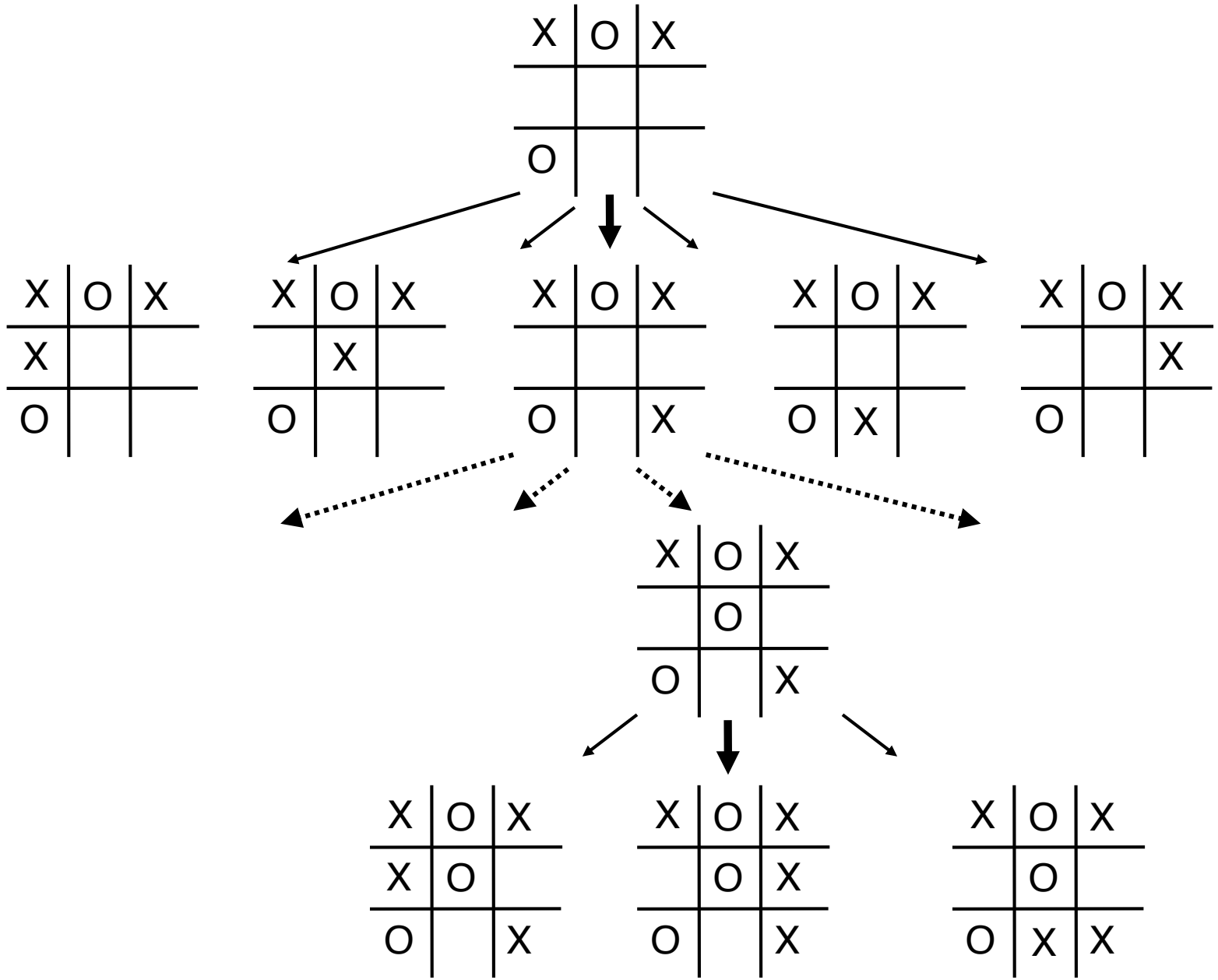
Large Games



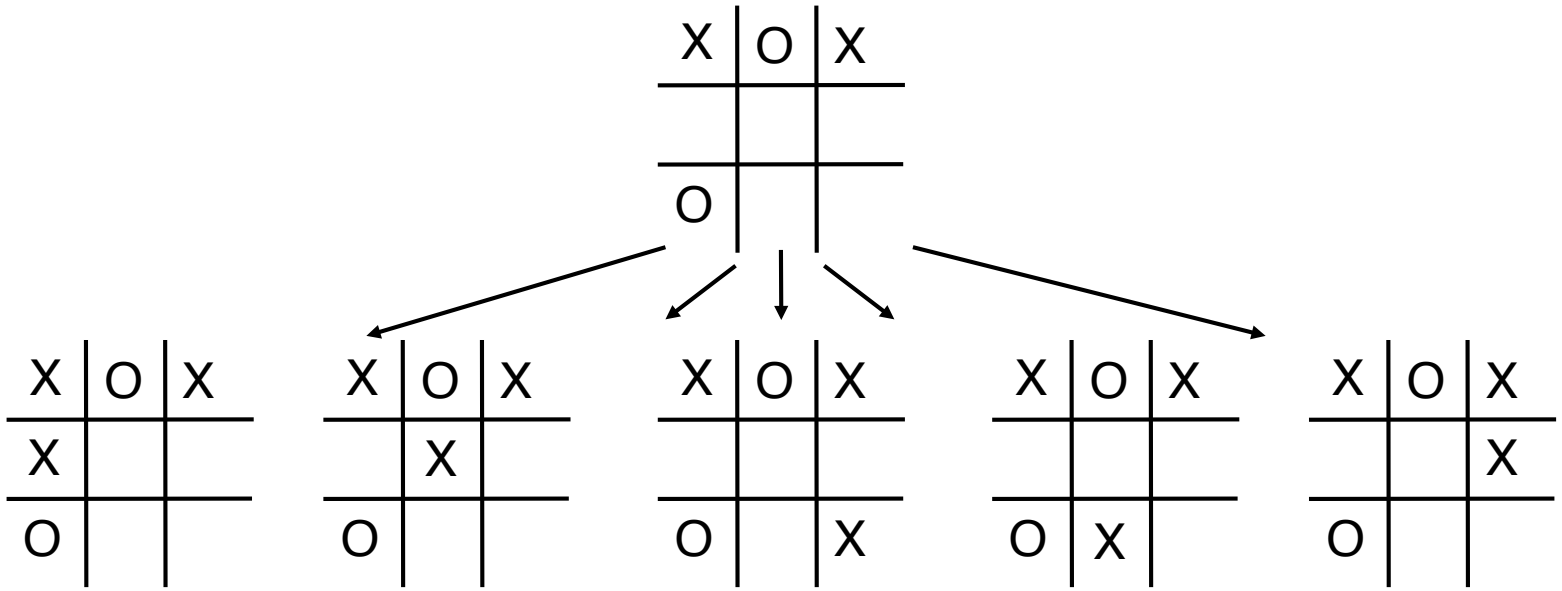




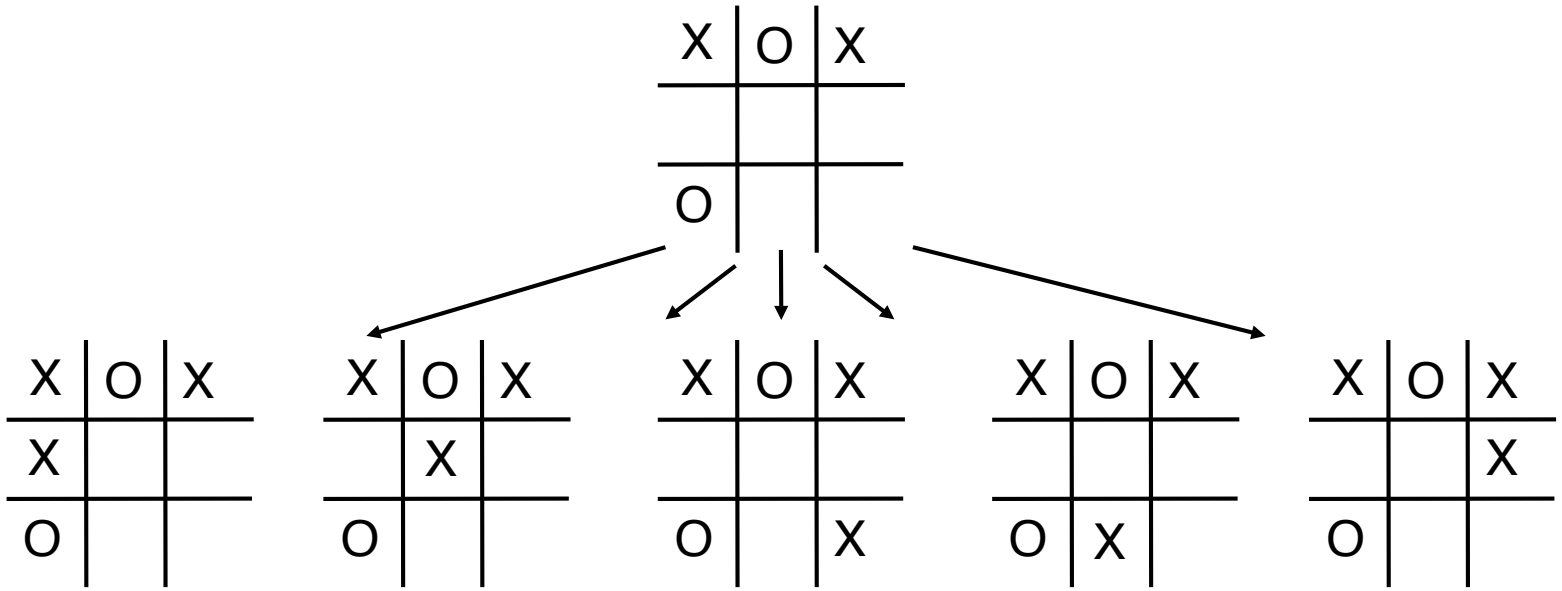
Complete Game Graph Search



Incomplete Search

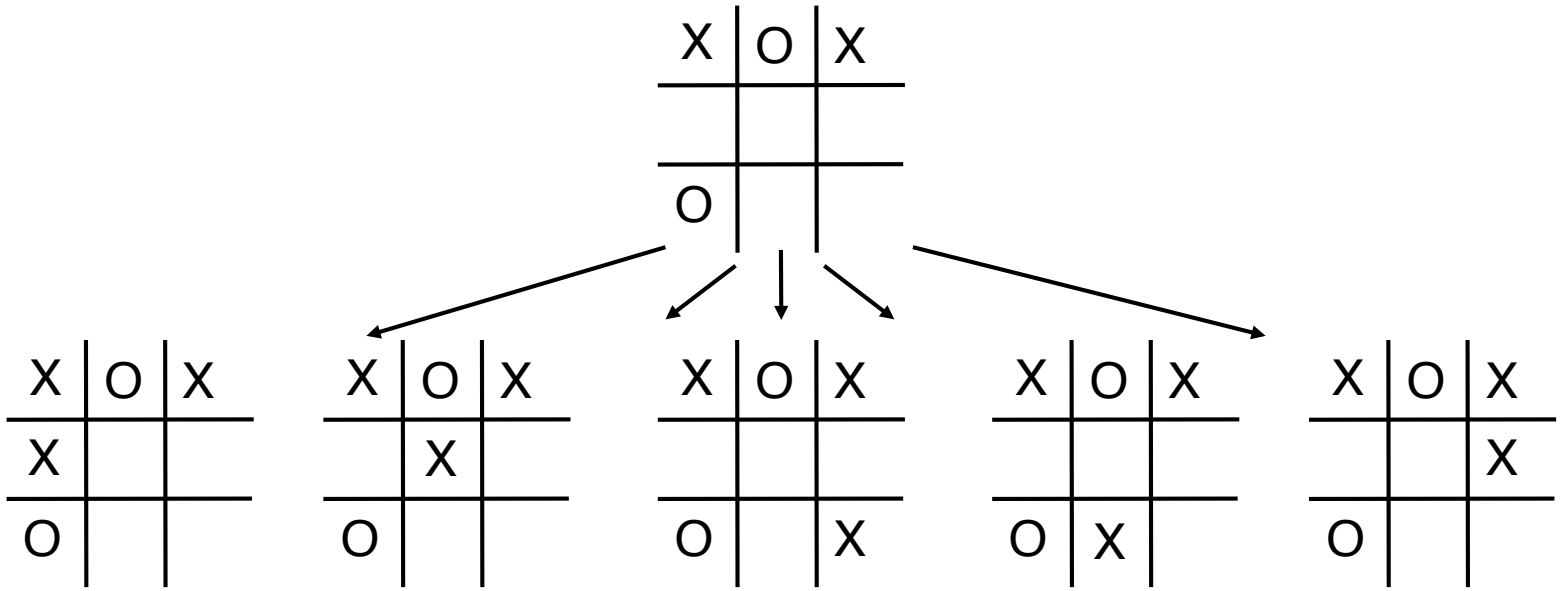


Evaluation of Non-Terminal States



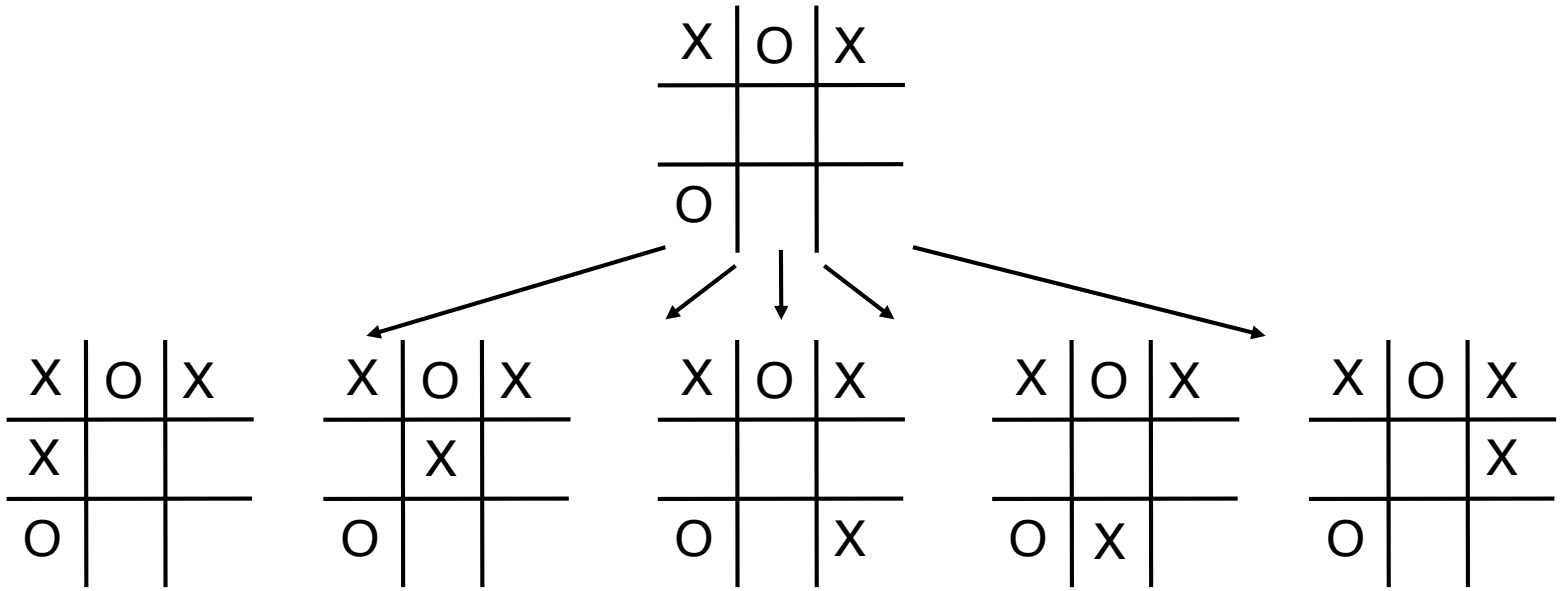
How do we evaluate non-terminal states?

Choice of Depth



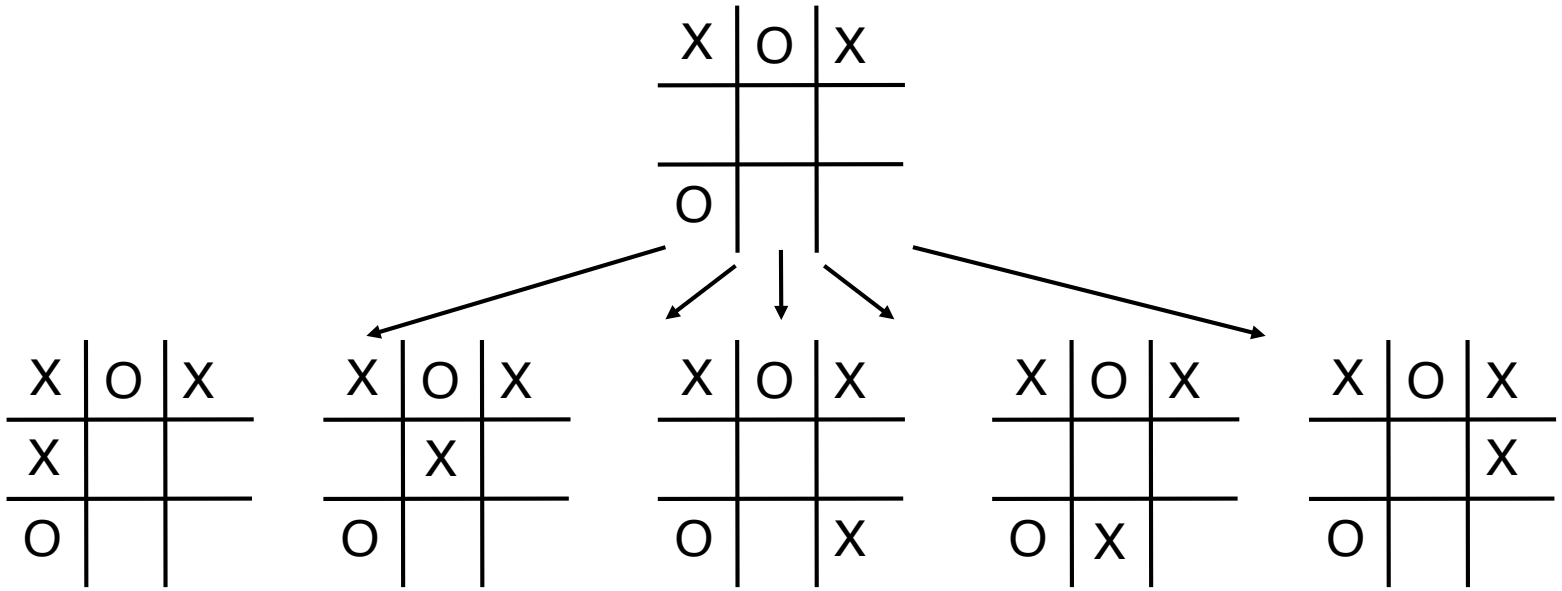
To what depth should we search?

Variable Depth Search



Should we search different branches to different depths?

Persistence



Can we preserve results across moves?

Evaluation Functions

How do we evaluate non-terminal states?

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 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.

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.

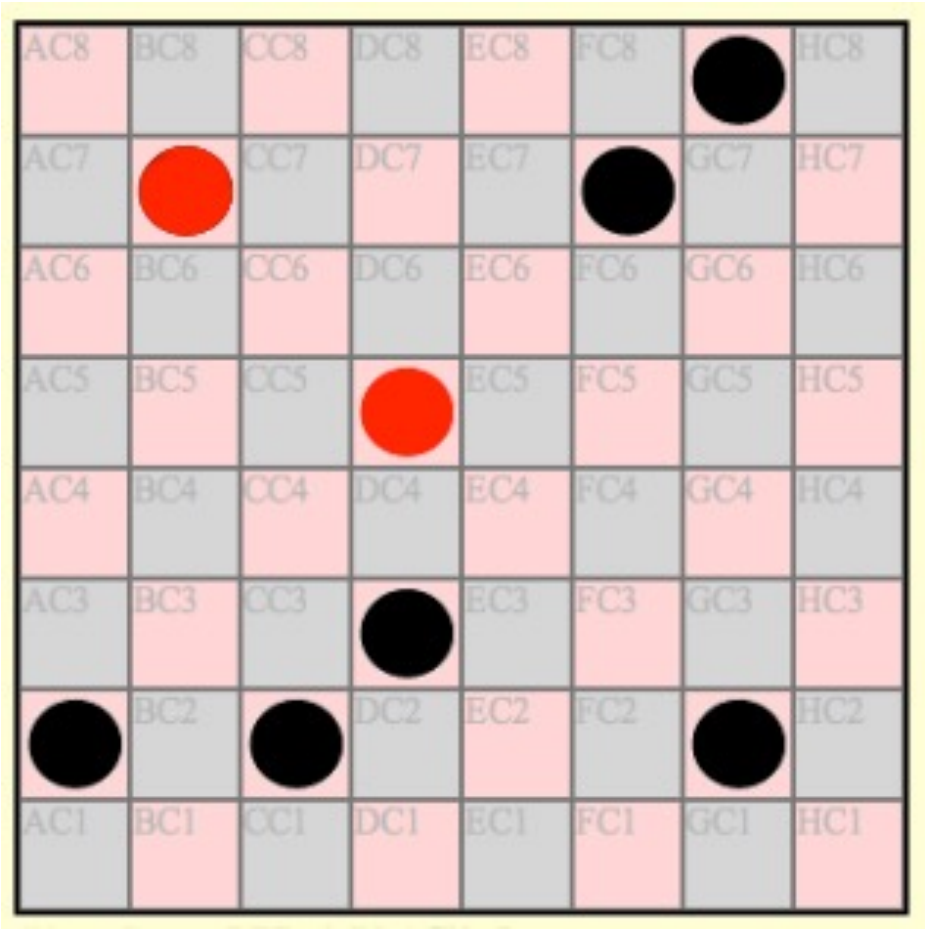
Basis - number of actions in a state or number of states reachable from that state. Horizon - current state or *n* moves away.

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

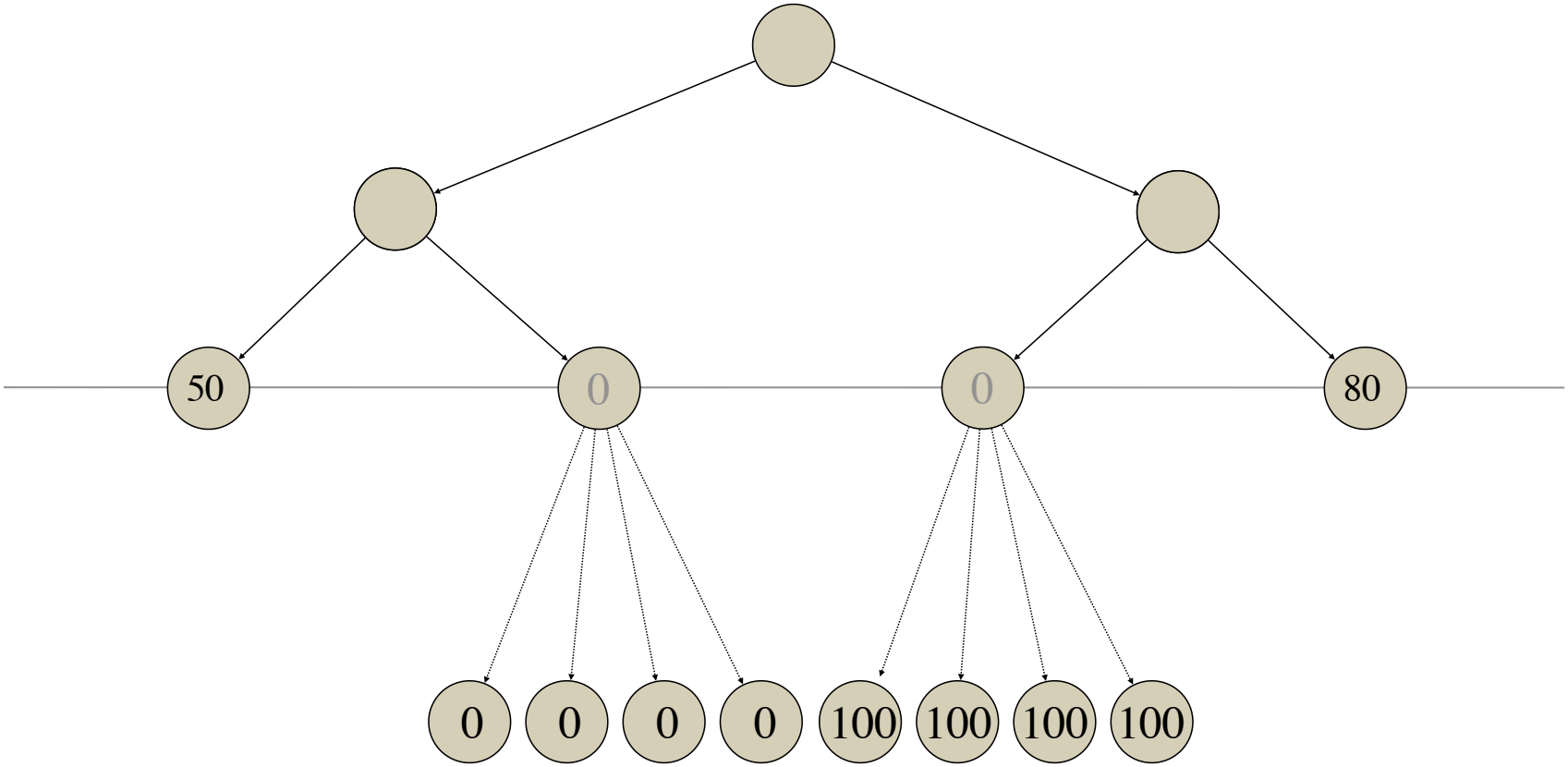


Heuristic #2 - Pessimism

Assume value of 0 for non-terminal states.

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

Example



Grey - estimates of rewards in non-terminal states - here 0.
Black - rewards in terminal states.

Heuristic #3 - Intermediate Values

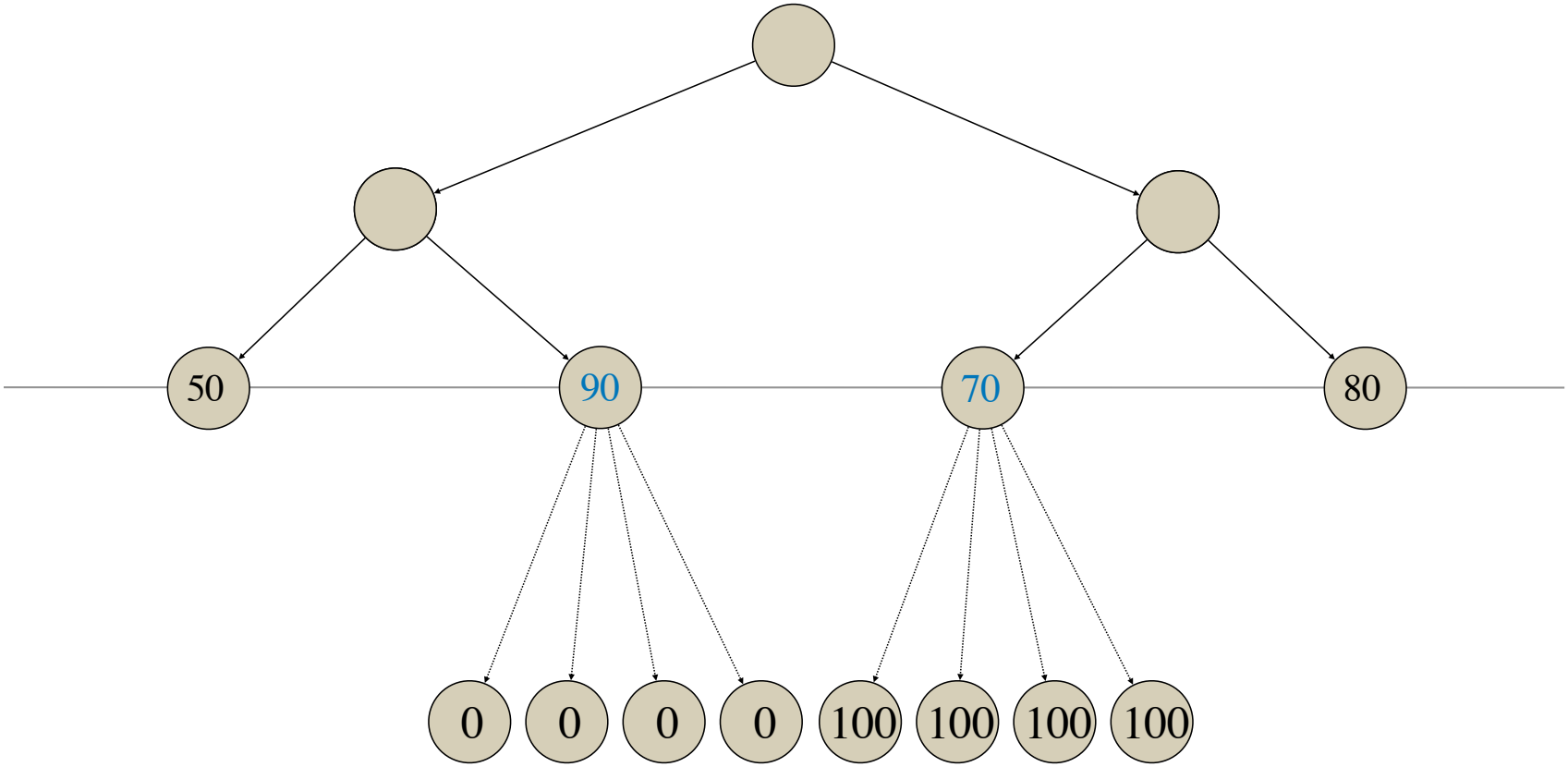
Assume reward for non-terminal states.

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

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



Blue - rewards in non-terminal states.

Black - rewards in terminal states.

Weighted Linear Combinations

Definition

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

Examples:

- Final State Value when known
- Mobility / Focus
- Intermediate State Values
- Other

Some players estimate weights by experimentation during the start clock. *More on this in a few weeks*.

Weighted Linear Combinations

Definition

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

Examples:

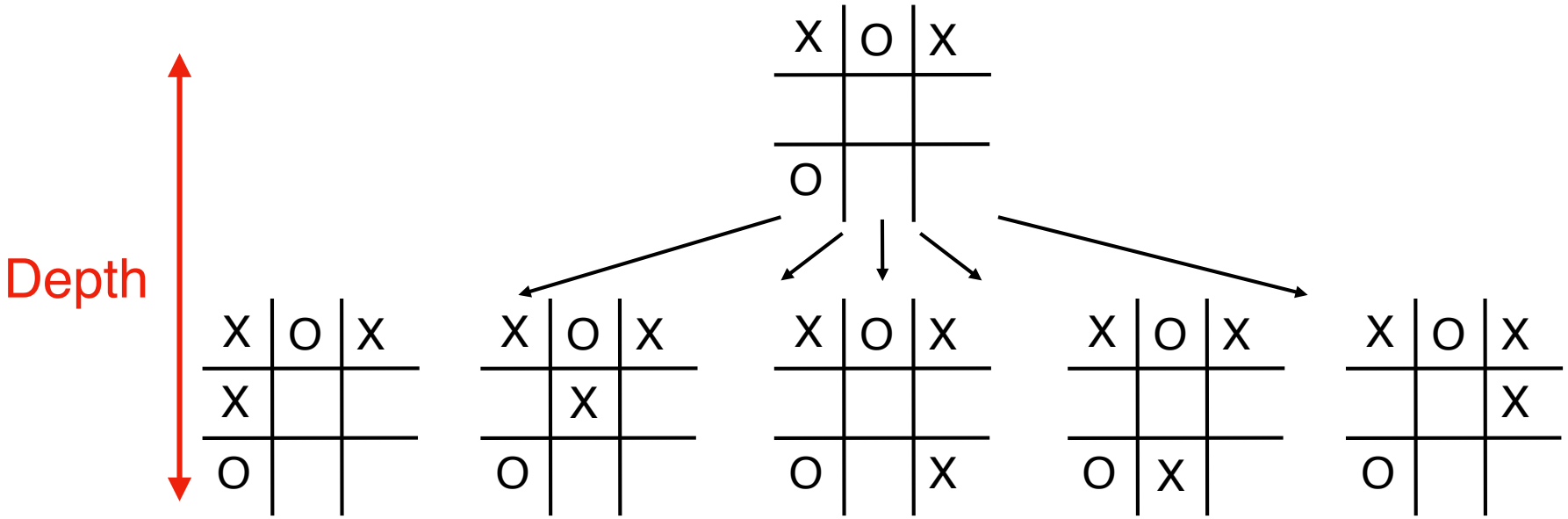
- Final State Value when known
- Mobility / Focus
- Intermediate State Values
- Other

Some players estimate weights by experimentation during the start clock. *More on this in a few weeks*.

Depth-Limited Search

To what depth should we search?

Depth-Limited Search



Depth-Limited 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)}
```

Depth-Limited Minimax

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

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);
       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);
       if (newscore===0) {return 0};
       if (newscore<score) {score = newscore}};
  return score}
```

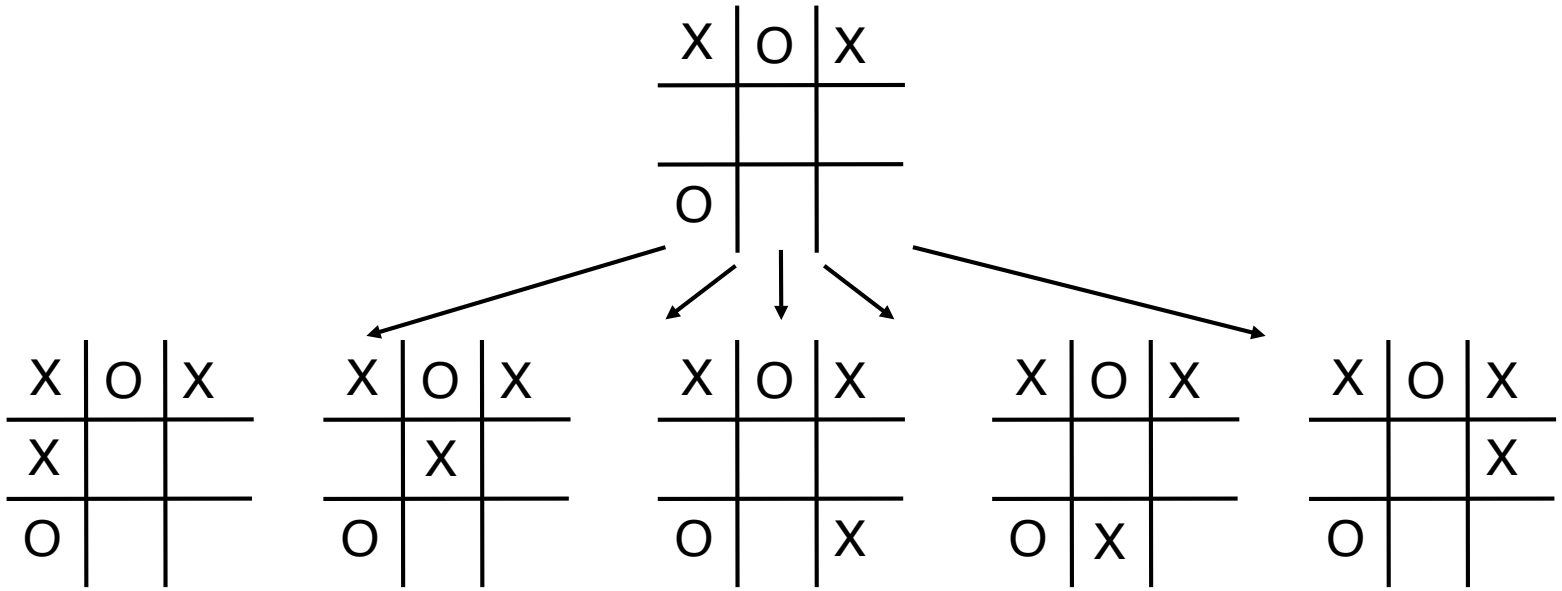
Remarks

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

Onestep and Twostep are degenerate depth-limited search with depths 1 and 2.

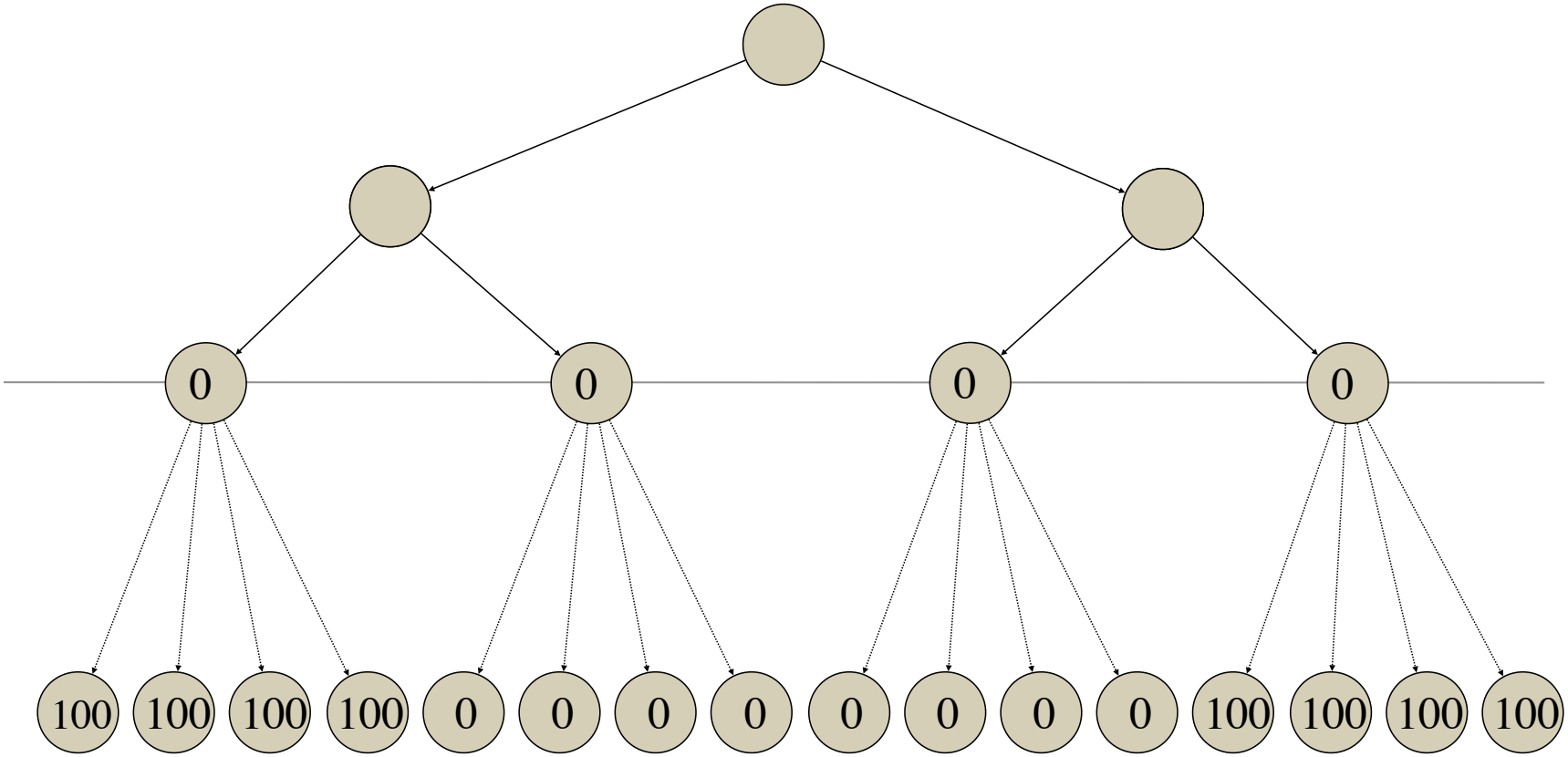
In general, we would like to allow greater depths.

Problem

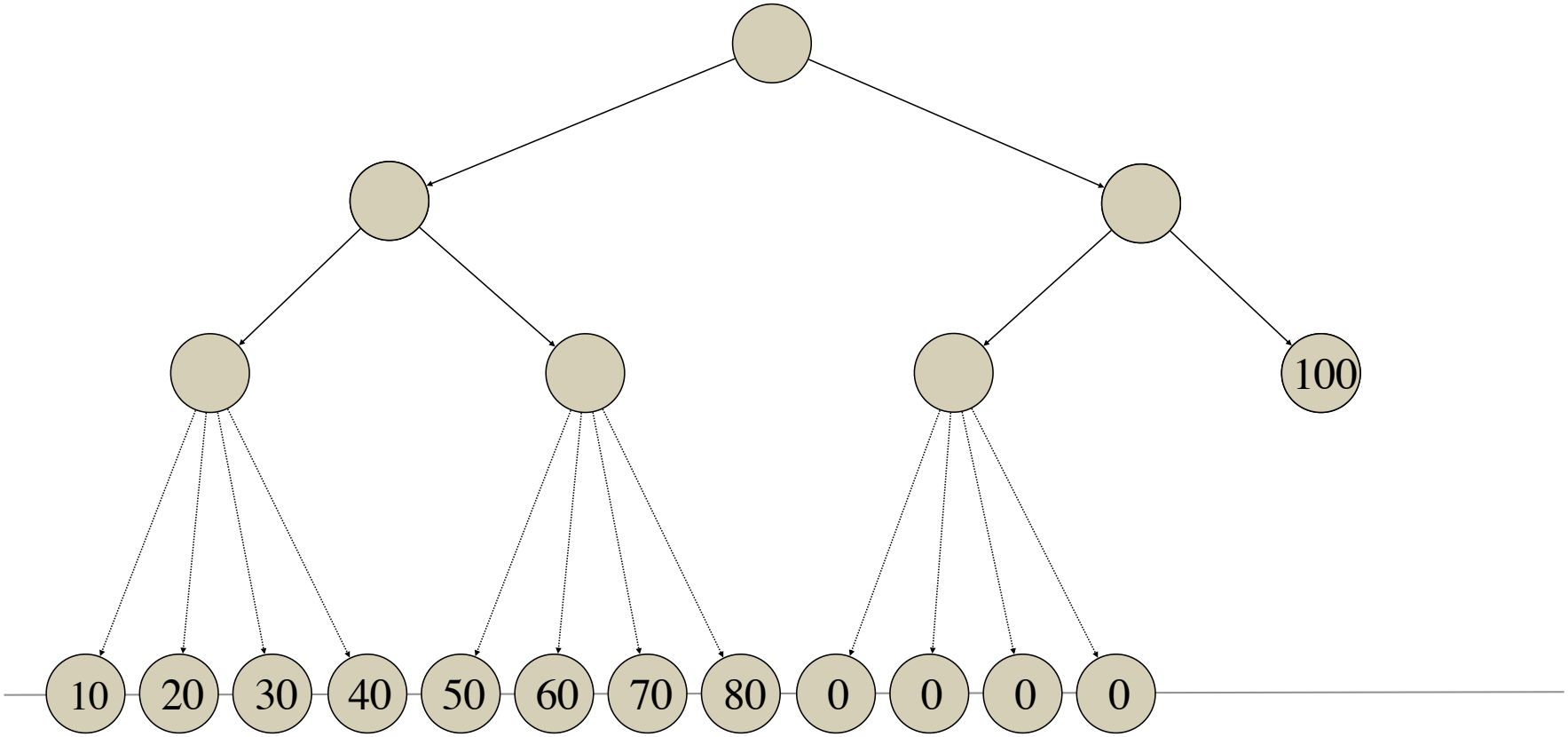


To what depth should we search?

Problem - Insufficient Depth



Problem - Excessive Depth



Iterative Deepening

To what depth should we search?

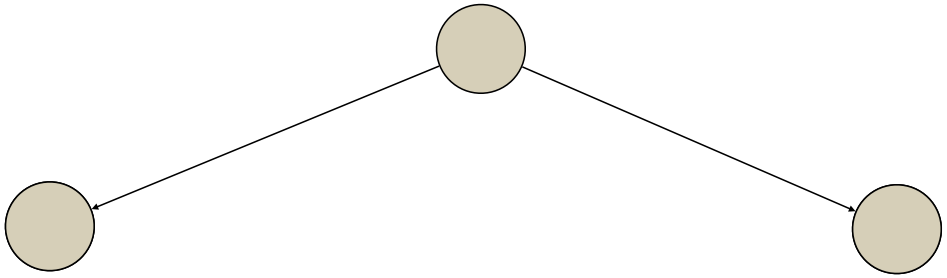
Iterative Deepening

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

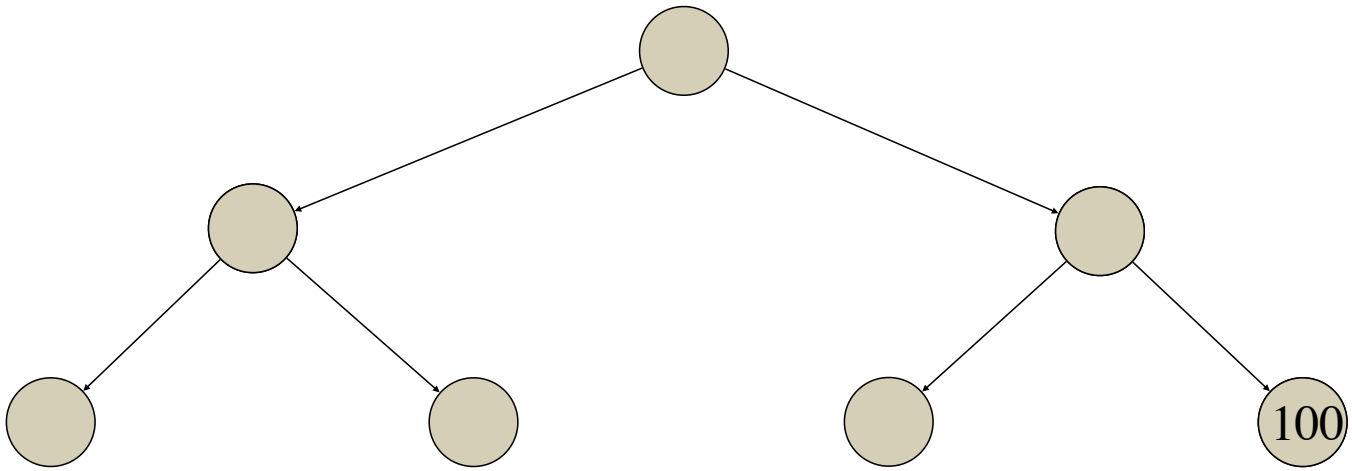
Continue till time runs out
Choose action that gives maximal value

Level 1

Level 2



Level 3



Naive Implementation

```
function playminimaxid ()
 {var best = findlegalx(state,library);
  for (var depth=1; depth<10; depth++)
      {var action = minimaxdepth(state,depth);
       best = action};
  return best}
```

At what depth do we stop?

Implementation

```
function playminimaxid ()
 {var deadline = Date.now()+(playclock-1)*1000;
  var best = findlegalx(state,library);
  for (var depth=1; depth<10; depth++)
      {var action = playminimaxidinner(state,depth,deadline);
       if (action===false) {return best};
       best = action};
  return best}
```

Implementation

```
function playminimaxidinner (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)}
```

maxscore and minscore

```
function maxscore (state,depth,deadline)
 {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 = minimaxid(newstate,depth,deadline);
       if (newscore===false) {return false};
       if (newscore===100) {return 100};
       if (newscore>score) {score = newscore}};
  return score}

function minscore (state,depth,deadline)
 {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 = minimaxid(newstate,depth,deadline);
       if (newscore===false) {return false};
       if (newscore===0) {return 0};
       if (newscore<score) {score = newscore}};
  return score}
```

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

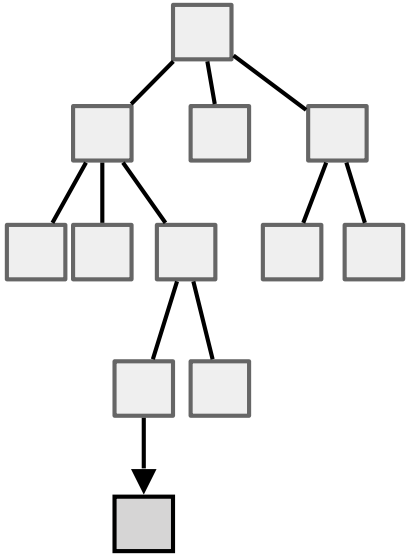
Why? *Tree is growing exponentially, so fringe of tree and size of tree above fringe are approximately same*

More Information

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

Persistent Tree Search (PTS)

Tree data structure



Overview

Create a game tree data structure.

Select branch of tree, sub-branch, etc. till reach fringe.
Expand fringe node.
Propagate values back to the root.

Repeat until time runs out.
Then select move w/ best estimated value.

Once a move is made,
Replace game tree with corresponding subtree.

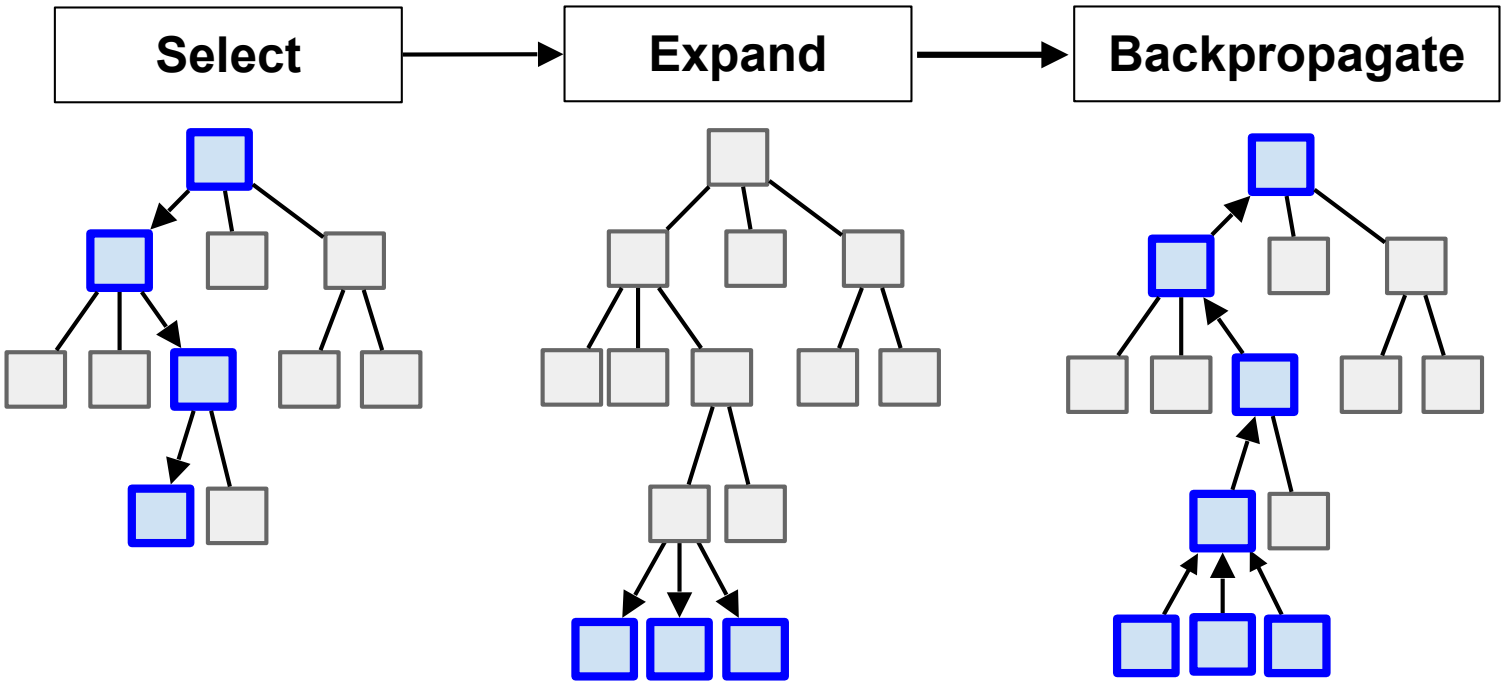
start

```
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 = parseInt(sc);
  playclock = parseInt(pc);
  var reward = parseInt(findreward(role,state,library));
  tree = makenode(state,findcontrol(state,library),reward);
  return 'ready'}

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

play

```
function play (move)
 {if (move!==nil)
     {tree = subtree(move,tree); state = tree.state};
  if (findcontrol(state,library)!==role) {return false};
  var deadline = Date.now()+(playclock-2)*1000;
  while (Date.now()<deadline) {process(tree)};
  return selectaction(tree)}
```

Processing the Tree



process

```
function process (node)
 {if (findterminalp(node.state,library)) {return true};
  if (node.children.length===0) {expand(node)}
     else {process(select(node))};
  update(node);
  return true}
```

(1) *Selection*

(2) *Expansion*

(3) *Backpropagation*

select

Return the subnode with the highest utility:

```
function select (node)
 {var total = node.visits;
  var child = node.children[0];
  var score = child.utility;
  for (var i=1; i<node.children.length; i++)
      {var newchild = node.children[i];
       var newscore = newchild.utility;
       if (newscore>score)
          {child = newchild; score = newscore}};
  return child}
```

expand

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

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

update

```
function update (node)
 {if (node.mover===role) {node.utility = scoremax(node)}
     else {node.utility = scoremin(node)};
 return true}

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

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

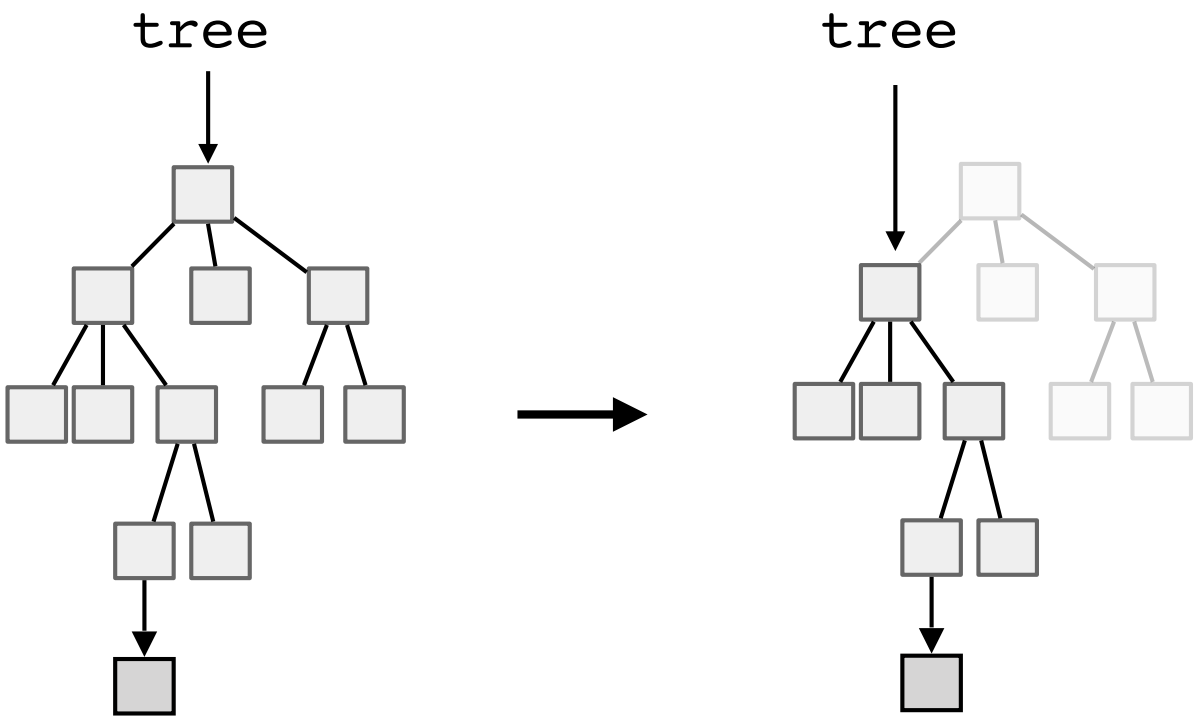
play

```
function play (move)
 {if (move!==nil)
     {tree = subtree(move,tree); state = tree.state};
  if (findcontrol(state,library)!==role) {return false};
  var deadline = Date.now()+(playclock-2)*1000;
  while (Date.now()<deadline) {process(tree)};
  return selectaction(tree)}
```

selectaction

```
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}
```

Updating the Tree After Move



play

```
function play (move)
 {if (move!==nil)
     {tree = subtree(move,tree); state = tree.state};
  if (findcontrol(state,library)!==role) {return false};
  var deadline = Date.now()+(playclock-2)*1000;
  while (Date.now()<deadline) {process(tree)};
  return selectaction(tree)}
```

subtree

```
function subtree (move,node)
 {if (node.children.length===0) {expand(node)};
  for (var i=0; i<node.actions.length; i++)
      {if (equalp(move,node.actions[i]))
          {return node.children[i]}}
  return node}
```

subtree

```
function subtree (move,node)
 {if (node.children.length===0) {expand(node)};
  for (var i=0; i<node.actions.length; i++)
      {if (equalp(move,node.actions[i]))
          {return node.children[i]}}
  return node}
```

Selection Functions

Utility

```
function select (node)
 {var total = node.visits;
  var child = node.children[0];
  var score = child.utility;
  for (var i=1; i<node.children.length; i++)
      {var newchild = node.children[i];
       var newscore = newchild.utility;
       if (newscore>score)
          {child = newchild; score = newscore}};
  return child}
```

Visits

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

function update (node)
 {if (node.mover===role) {node.utility = scoremax(node)}
     else {node.utility = scoremin(node)};
  node.visits = node.visits+1;
  return true}

function select (node)
 {var child = node.children[0];
  var visits = node.children[0].visits;
  for (var i=1; i<node.children.length; i++)
      {var newvisits = node.children[i].visits;
       if (newvisits<visits)
          {child = node.children[i]; visits = newvisits}};
  return child}
```

Exploitation + Exploration

```
function select (node)
 {var total = node.visits;
  var child = node.children[0];
  var score = value(child.utility,child.visits,total);
  for (var i=1; i<node.children.length; i++)
      {var newchild = node.children[i];
       var newvalue = newchild.utility;
       var newvisits = newchild.visits;
       var newscore = value(newvalue,newvisits,total);
       if (newscore>score)
          {child = newchild; score = newscore}};
  return child}

function value (utility,visits,total)
 {var score = (utility + Math.round((1 - visits/total)*100));
  return score}
```

Exploitation

Exploration

Ideas

Some players use different formulas for combining Exploitation and Exploration.

$$\text{selectValue} = \frac{\text{node.utility}}{\text{node.visits}} + C \times \sqrt{\frac{\ln(\text{node.parent.visits})}{\text{node.visits}}}$$

Some players find value in recording the **standard deviation** of scores at each node or where the **change in estimated values** and favor expanding those nodes with the highest value to clarify the uncertainty.

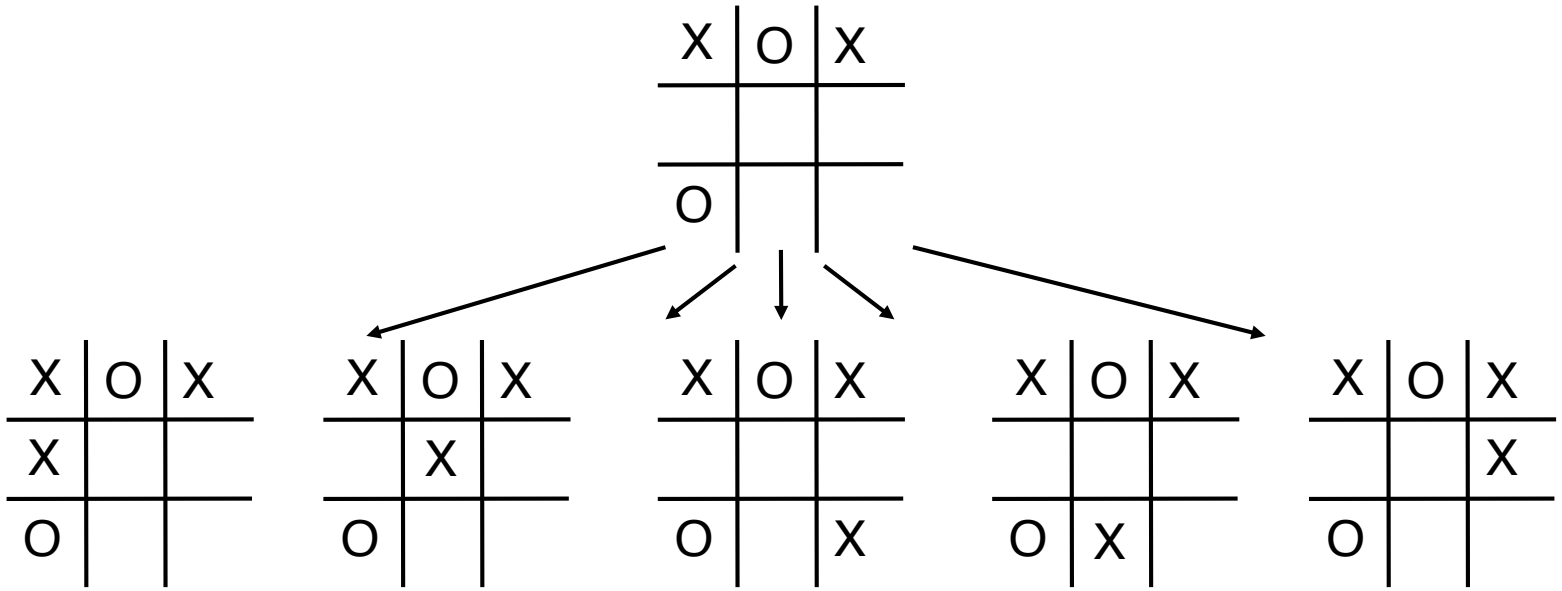
Problem

The preceding 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).

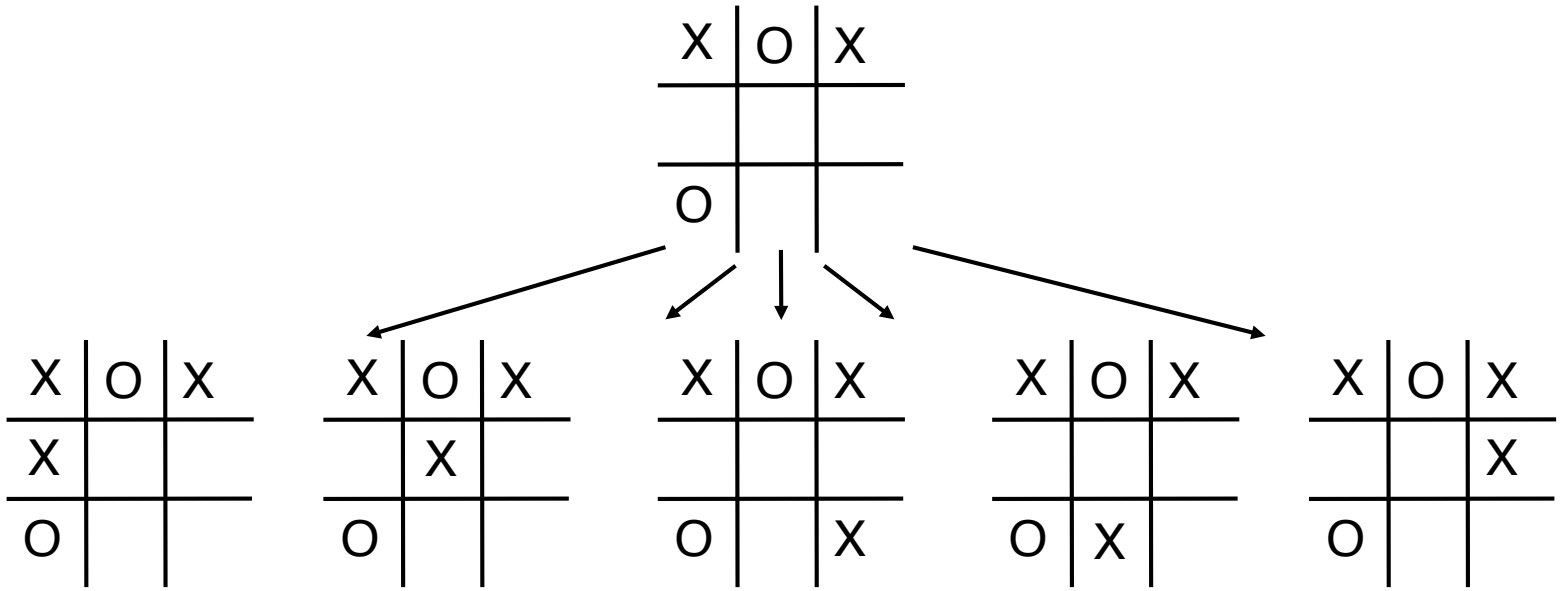
Fix by adding additional information to each node. (Code on the website does not do this. This is **your** job.)

Issues

Incomplete Search

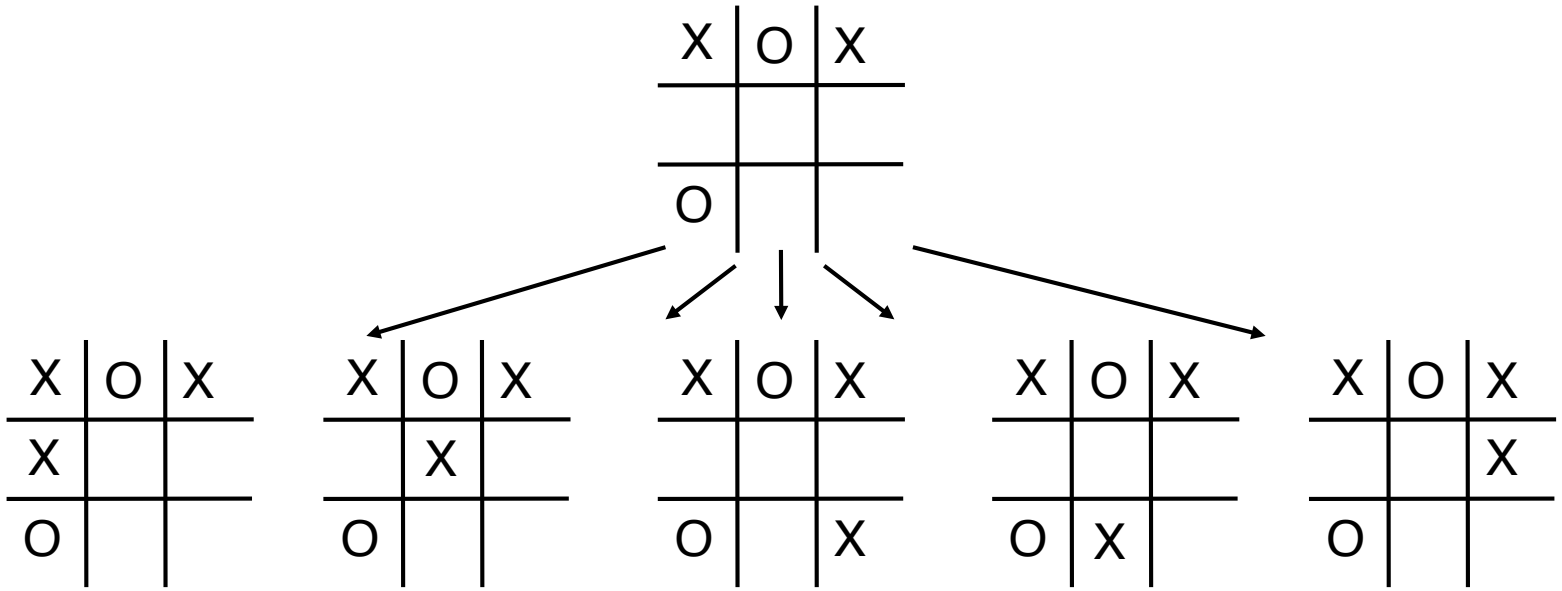


Evaluation of Non-Terminal States



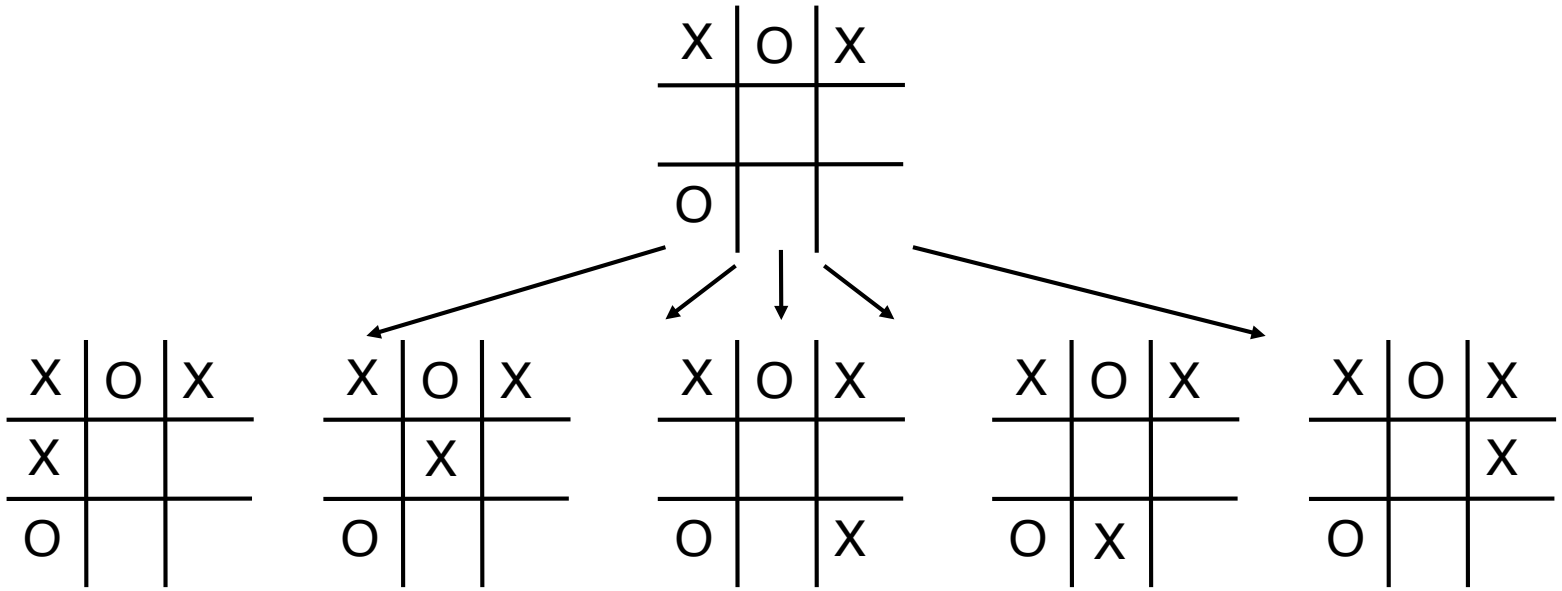
How do we evaluate non-terminal states?

Choice of Depth



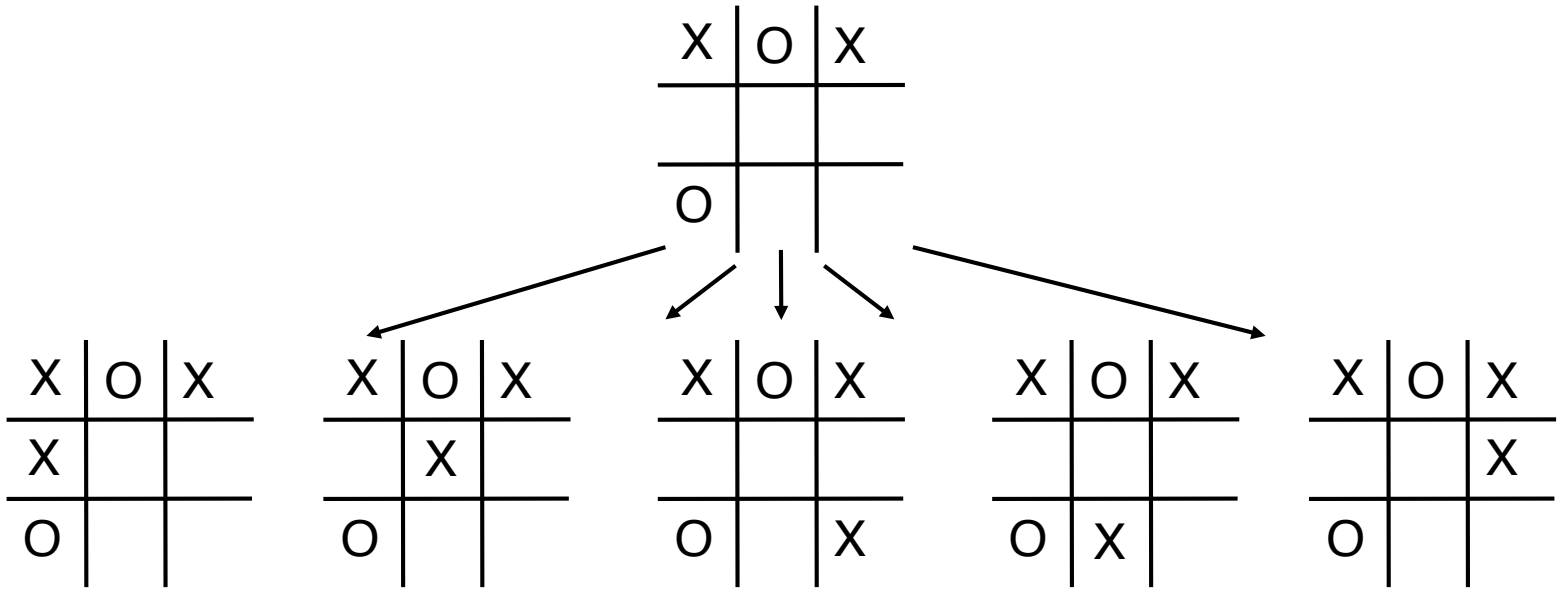
To what depth should we search?

Variable Depth Search



Can we search different branches to different depths? ✔

Persistence



Can we preserve tree across moves? ✔

GENERAL
GAME
PLAYING

GENERAL
GAME
PLAYING