Complete Search

Game Description

init(\text{cell}(1,1,b))
init(\text{cell}(1,2,b))
init(\text{cell}(1,3,b))
init(\text{cell}(2,1,b))
init(\text{cell}(2,2,b))
init(\text{cell}(2,3,b))
init(\text{cell}(3,1,b))
init(\text{cell}(3,2,b))
init(\text{cell}(3,3,b))
init(\text{control}(x))
init(\text{control}(o))

legal(\text{mark}(X,Y)) :-
  \text{true}(\text{cell}(X,Y,b)) & \text{control}(P)
legal(\text{noop}) :-
  \text{true}(\text{control}(x))
legal(\text{noop}) :-
  \text{true}(\text{control}(o))

game(\text{goal}(x,100)) :- \text{line}(x)
game(\text{goal}(x,50)) :- \text{draw}
game(\text{goal}(x,0)) :- \text{line}(o)
game(\text{goal}(o,100)) :- \text{line}(o)
game(\text{goal}(o,50)) :- \text{draw}
game(\text{goal}(o,0)) :- \text{line}(x)
Game Playing Protocol

Start - manager sends Start message to players
\[
\text{start}(id, \text{role}, \text{game}, \text{startclock}, \text{playclock})
\]
\[
\text{ready}
\]

Play - manager sends Play messages to players
\[
\text{play}(id, [\text{action}, ..., \text{action}])
\]
\[
\text{action}
\]

Stop - manager sends Stop message to players
\[
\text{stop}(id, [\text{action}, ..., \text{action}])
\]
\[
\text{done}
\]
Your Mission

Your mission is to write definitions for the three basic event handlers (\texttt{start, play, stop}).

To help you get started, we provide subroutines for accessing the implicit game graph instead of the explicit description.

NB: The game graph is virtual - it is computed on the fly from the game description. Hence, our subroutines are not cheap. Best to cache results rather than call repeatedly.

Basic Player Subroutines

\begin{align*}
\text{findroles}(game) & \rightarrow [role, \ldots, role] \\
\text{findbases}(game) & \rightarrow \{\text{proposition}, \ldots, \text{proposition}\} \\
\text{findinputs}(game) & \rightarrow \{\text{action}, \ldots, \text{action}\} \\
\text{findinitial}(game) & \rightarrow \text{state} \\
\text{findterminal}(state, game) & \rightarrow \text{boolean} \\
\text{findgoal}(role, state, game) & \rightarrow \text{number} \\
\text{findlegals}(role, state, game) & \rightarrow \{\text{action}, \ldots, \text{action}\} \\
\text{findnext}([\text{action}, \ldots, \text{action}], state, game) & \rightarrow \text{state}
\end{align*}
Legal Play

function start (id, role, rules, start, play)
{game = rules;
 player = role;
 state = null;
 return 'ready'}
Playing

function play (id, move)
    {if (move.length == 0)
        {state=findinitial(game)}
    else {state=findnext(move, state, game)};
    var actions=findlegals(role, state, game);
    return actions[0]}

Stopping

function stop (id, move)
    {return 'done'}
Single Player Games

Game Graph

![Game Graph Diagram]
Depth First Search

\[ a \ b \ e \ f \ c \ g \ h \ d \ i \ j \]

Advantage: Small intermediate storage
Disadvantage: Susceptible to garden paths
Disadvantage: Susceptible to infinite loops

Breadth First Search

\[ a \ b \ c \ d \ e \ f \ g \ h \ i \ j \]

Advantage: Finds shortest path
Disadvantage: Consumes large amount of space
Time Comparison

Branching 2 and depth $d$ and solution at depth $k$

<table>
<thead>
<tr>
<th>Time</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth</td>
<td>$k$</td>
<td>$2^d - 2^{d-k}$</td>
</tr>
<tr>
<td>Breadth</td>
<td>$2^{k-1}$</td>
<td>$2^k - 1$</td>
</tr>
</tbody>
</table>

Time Comparison

Analysis for branching $b$ and depth $d$ and solution at depth $k$.

<table>
<thead>
<tr>
<th>Time</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth</td>
<td>$k$</td>
<td>$\frac{b^d - b^{d-k}}{b - 1}$</td>
</tr>
<tr>
<td>Breadth</td>
<td>$\frac{b^{k-1} - 1}{b - 1} + 1$</td>
<td>$\frac{b^k - 1}{b - 1}$</td>
</tr>
</tbody>
</table>
Space Comparison

Total depth $d$ and solution depth $k$.

<table>
<thead>
<tr>
<th></th>
<th>Space</th>
<th>Binary</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth</td>
<td>$d$</td>
<td>$(b - 1) \times (d - 1) + 1$</td>
<td></td>
</tr>
<tr>
<td>Breadth</td>
<td>$2^{k-1}$</td>
<td>$b^{k-1}$</td>
<td></td>
</tr>
</tbody>
</table>

Iterative Deepening

Run depth-limited search repeatedly,

starting with a small initial depth,

incrementing on each iteration,

until success or run out of alternatives.
Example

a
a b c d
a b e f c g h d i j

Advantage: Small intermediate storage
Advantage: Finds shortest path
Advantage: Not susceptible to garden paths
Advantage: Not susceptible to infinite loops

Time Comparison

<table>
<thead>
<tr>
<th>Depth</th>
<th>Iterative</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>57</td>
<td>31</td>
</tr>
<tr>
<td>n</td>
<td>(2^{n+1} - n - 2)</td>
<td>(2^n - 1)</td>
</tr>
</tbody>
</table>
General Results

Theorem [Korf]: The cost of iterative deepening search is $b/(b-1)$ times the cost of depth-first search (where $b$ is the branching factor).

Theorem: The space cost of iterative deepening is no greater than the space cost for depth-first search.
Minimax

Basic Idea

Intuition - Select a move that is guaranteed to produce the highest possible return no matter what the opponents do.

In the case of a one move game, a player should choose an action such that the value of the resulting state for any opponent action is greater than or equal to the value of the resulting state for any other action and opponent action.

In the case of a multi-move game, minimax goes to the end of the game and “backs up” values.
Bipartite Game Graph/Tree

State Value

The value of a max node for player $p$ is either the utility of that state if it is terminal or the maximum of all values for the min nodes that result from its legal actions.

$$value(p,x) = \begin{cases} \text{goal}(p,x) & \text{if terminal}(x) \\ \max\{\text{value}(p,\text{minnode}(p,a,x)) \mid \text{legal}(p,a,x)\} & \end{cases}$$

The value of a min node is the minimum value that results from any legal opponent action.

$$value(p,n) = \min\{\text{value}(p,\text{maxnode}(P-p,b,n)) \mid \text{legal}(P-p,b,n)\}$$
Bipartite Game Tree

Best Action

A player should choose an action \( a \) that leads to a \( \min \) node with maximal value, i.e. the player should prefer action \( a \) to action \( a' \) if and only if the following holds.

\[
\text{value}(p, \text{minnode}(p, a, x)) > \text{value}(p, \text{minnode}(p, a', x))
\]
Best Action

```javascript
function bestaction (role, state)
{var value = [];
 var acts = legals(role, state);
 for (act in acts)
   {value[act]=minscore(role,act,state)};
 return act that maximizes value[act]}
```

Scores

```javascript
function maxscore (role, state)
{if (terminalp(state)) return goal(state);
 var value = [];
 for (act in legals(role, state))
   {value[act] = minscore(role,act,state);
    return max(value)}

function minscore (role, action, state)
{var value = [];
 for (move in moves(role,action,state))
   {value[move] =
    maxscore(role,next(move,state))};
 return min(value)}
```
Bounded Minimax

If the minvalue for an action is determined to be 100, then there is no need to consider other actions.

In computing the minvalue for a state if the value to the first player of an opponent’s action is 0, then there is no need to consider other possibilities.
Alpha Beta Pruning

Alpha Beta Pruning

Alpha-Beta Search - Same as Bounded Minimax except that bounds are computed dynamically and passed along as parameters. See Russell and Norvig for details.
Benefits

Best case of alpha-beta pruning can reduce search space to square root of the unpruned search space, thereby dramatically increasing depth searchable within given time bound.
Caching

Virtual Game Graph

The game graph is virtual - it is computed on the fly from the game description. Hence, our subroutines are not cheap.

May be good to cache results (build an explicit game graph) rather than call repeatedly.
Benefit - Avoiding Duplicate Work

Consider a game in which the actions allow one to explore an 8x8 board by making moves left, right, up, and down.

Game tree has branching factor 4 and depth 16. Fringe of tree has $4^{16} = 4,294,967,296$ nodes.

Only 64 states. Space searched much faster if duplicate states detected.

Disadvantage - Extra Work

Finding duplicate states takes time proportional to the log of the number of states and takes space linear in the number of states.

Work wasted when duplicate states rare. (We shall see two common examples of this next week.)

Memory might be exhausted. Need to throw away old states.