General Game Playing

Game Playing

Michael Genesereth
Computer Science Department
Stanford University
Programme

Game Management

Game Players

Complete Search Techniques for Single Player Games
Complete Search Techniques for Multiple Player Games
Game Management
Communication by message passing between manager and players using "mailboxes" associated with their identifiers.
Message Passing

Manager

Browser

Server

message

message

Player

Player
Message Passing

Manager

reply, reply

Player

... Player

Browser

Server

Browser
Manager requests:

- ping($n$)
- start($n$, role, ruleset($r_1, ..., r_k$), startclock, playclock)
- play($n$, $a$)
- stop($n$, $a$)
- abort($n$)

Player responses:

- reply($n$, value)
A *ping* message is used to determine if a player is running, communicating, and ready to play a match.

General Form:

```
ping(messageid)
```

Replies:

```
reply(messageid, ready) - ready to play
reply(messageid, busy) - not ready
```

No reply after 5 seconds interpreted as not ready.
A start message initiates a match.

General Form:
\[
\text{start}(\text{messageid, role, ruleset}(r_1, \ldots, r_k), \text{start, play})
\]

Reply:
\[
\text{reply}(\text{messageid, ready}) - \text{ready to begin play}
\]

NB: Match begins as soon as all players have replied or when \textit{startclock} seconds have elapsed whichever comes first.
The play Message

A play message is an update and a request for an action.

General Form:

\[
\text{play}(\text{messageid}, \text{nil}) \\
\text{play}(\text{messageid}, \text{action})
\]

Reply:

\[
\text{reply}(\text{messageid}, \text{action})
\]

NB: nil / [] is argument on first step, action thereafter, where action is the action performed on preceding step.

NB: If player returns an illegal action or playclock has elapsed, the Manager substitutes a random legal action.
A stop message is used to inform players that the match has terminated successfully.

General Form:
```
stop(messageid, action)
```

Reply:
```
optional
```
An abort message is used to tell players that a match has terminated abnormally.

**General Form:**
```
abort(messageid)
```

**Reply:**
```
optional
```
Match Management Procedure

Begins when manager receives a request to run a match of a given game with given players and given start clock and play clock.

(1) optionally sends ping messages to players.
(2) sends start message with appropriate parameters.
(3) sends play messages to receive plays
(4) sends stop message when game terminates
(5) sends abort for abnormal termination
Tic-Tac-Toe Example

Manager to Indy: ping(p1)
Manager to Lara: ping(p1)
Indy to Manager: reply(p1,ready)
Lara to Manager: reply(p1,ready)
Manager to Indy: start(s1,x,ruleset(...),10,10)
Manager to Lara: start(s1,o,ruleset(...),10,10)
Indy to Manager: reply(s1,ready)
Lara to Manager: reply(s1,ready)
Manager to Indy: start(s1,x,ruleSet(...),10,10)
Manager to Lara: start(s1,o,ruleSet(...),10,10)
Indy to Manager: reply(s1,ready)
Lara to Manager: reply(s1,ready)

Manager to Indy: play(1,nil)
Manager to Lara: play(1,nil)
Indy to Manager: reply(1,mark(1,1))
Manager to Indy: start(s1,x,ruleset(...),10,10)
Manager to Lara: start(s1,o,ruleset(...),10,10)
Indy to Manager: reply(s1,ready)
Lara to Manager: reply(s1,ready)

Manager to Indy: play(1,nil)
Manager to Lara: play(1,nil)
Indy to Manager: reply(1,mark(1,1))

Manager to Indy: play(2,mark(1,1))
Manager to Lara: play(2,mark(1,1))
Lara to Manager: reply(2,mark(1,2))

Tic-Tac-Toe Example
Manager to Indy: start(s1,x,ruleSet(...),10,10)
Manager to Lara: start(s1,o,ruleSet(...),10,10)
Indy to Manager: reply(s1,ready)
Lara to Manager: reply(s1,ready)

Manager to Indy: play(1,nil)
Manager to Lara: play(1,nil)
Indy to Manager: reply(1,mark(1,1))

Manager to Indy: play(2,mark(1,1))
Manager to Lara: play(2,mark(1,1))
Lara to Manager: reply(2,mark(1,2))

Manager to Indy: play(3,mark(1,2))
Manager to Lara: play(3,mark(1,2))
Indy to Manager: reply(3,mark(2,2))
Manager to Indy: start(s1,x,ruleset(...),10,10)
Manager to Lara: start(s1,o,ruleset(...),10,10)
Indy to Manager: reply(s1,ready)
Lara to Manager: reply(s1,ready)

Manager to Indy: play(1,nil)
Manager to Lara: play(1,nil)
Indy to Manager: reply(1,mark(1,1))

Manager to Indy: play(2,mark(1,1))
Manager to Lara: play(2,mark(1,1))
Lara to Manager: reply(2,mark(1,2))

Manager to Indy: play(3,mark(1,2))
Manager to Lara: play(3,mark(1,2))
Indy to Manager: reply(3,mark(2,2))

Manager to Indy: play(4,mark(2,2))
Manager to Lara: play(4,mark(2,2))
Lara to Manager: reply(4,mark(1,3))
Manager to Indy: start(s1,x,ruleset(...),10,10)
Manager to Lara: start(s1,o,ruleset(...),10,10)
Indy to Manager: reply(s1,ready)
Lara to Manager: reply(s1,ready)

Manager to Indy: play(1,nil)
Manager to Lara: play(1,nil)
Indy to Manager: reply(1,mark(1,1))

Manager to Indy: play(2,mark(1,1))
Manager to Lara: play(2,mark(1,1))
Lara to Manager: reply(2,mark(1,2))

Manager to Indy: play(3,mark(1,2))
Manager to Lara: play(3,mark(1,2))
Indy to Manager: reply(3,mark(2,2))

Manager to Indy: play(4,mark(2,2))
Manager to Lara: play(4,mark(2,2))
Lara to Manager: reply(4,mark(1,3))

Manager to Indy: play(5,mark(1,3))
Manager to Lara: play(5,mark(1,3))
Indy to Manager: reply(5,mark(3,3))
Manager to Indy: start(s1,x,ruleset(...),10,10)
Manager to Lara: start(s1,o,ruleset(...),10,10)
Indy to Manager: reply(s1,ready)
Lara to Manager: reply(s1,ready)

Manager to Indy: play(1,nil)
Manager to Lara: play(1,nil)
Indy to Manager: reply(1,mark(1,1))

Manager to Indy: play(2,mark(1,1))
Manager to Lara: play(2,mark(1,1))
Lara to Manager: reply(2,mark(1,2))

Manager to Indy: play(3,mark(1,2))
Manager to Lara: play(3,mark(1,2))
Indy to Manager: reply(3,mark(2,2))

Manager to Indy: play(4,mark(2,2))
Manager to Lara: play(4,mark(2,2))
Lara to Manager: reply(4,mark(1,3))

Manager to Indy: play(5,mark(1,3))
Manager to Lara: play(5,mark(1,3))
Indy to Manager: reply(5,mark(3,3))

Manager to Indy: stop(6,mark(3,3))
Manager to Lara: stop(6,mark(3,3))
Manager to Indy: abort(a1)
Manager to Lara: abort(a1)
Gamemaster

About  Resources  Profile
Games  Players  Basics
Matches  Leaderboard
 Resources

 Game Processing Utilities

Sierra - interactive development environment for Epilog rulesets and datasets. Click [here](#) for documentation.

Check Rules - allows users to test (Sierra configuration)

Check Style - allows users to test Javascript stylesheets on sample states.

Game Checker - allows users to test games and stylesheets.

Optimize - allows users to perform various optimizations on game descriptions.

Materialize - allows users to materialize relations used in game descriptions.

Ground - allows users to convert game descriptions with variables to fully grounded versions. Useful for game analysis, such as finding factors, latches, dead states, and so forth.

Simplify - allows users to simplify games to include only potentially relevant actions based on dependency analysis.

Conjoin - allows users to factor games into conjunctive subgames based on dependency analysis.

 Game Management Applications (Server-Based Games)

Standalone - human player interface and game manager in single web page

Competitor - human player interface for competitions

Autoplayer - automated player interface for competitions

Manager - game manager for competitions

 Game Management Applications (Local Games)

Standaloneopen - human player interface and game manager in single web page

Competitoropen - human player interface for competitions
Gamemaster

Protocol: manager  
Identifier: manager

Game: buttonsandlights
Startclock: Infinity
Playclock: Infinity

Step: 1

Roles  |  robot
---|---
Players | anonymous
Score  | 0
Errors  | 0

Clear  Begin  Pause  Resume  End
Gamemaster

Protocol: manager
Identifier: manager

Game: tictactoe
Startclock: 10
Playclock: 10

Control: x

Roles | x | o
--- | --- | ---
Players | indy | lara
Score | 50 | 50
Errors | 0 | 0

Clear | Begin | Pause | Resume | End
Game Management Applications (Local Games)

- Standaloneopen - human player interface and game manager in single web page
- Competitoropen - human player interface for competitions
- Autoplayeropen - automated player interface for competitions
- Manageropen - game manager for competitions

---

Browser-Based Players

- Legal - Legal player
- Random - Random player
- Onestep - One Step player
- Minimax - Full Minimax player
- Minimaxdepth - Minimax player with fixed depth
- Minimaxid - Minimax player with iterative deepening
- Greedy - Greedy player
- MCS - Monte Carlo Search player

---

JavascriptCode

- EpilogJS - Logic Programming Interpreter
- ParametricJS - Code for basic players
- PlayerJS - Player set-up for NodeJS

---

Older General Game Playing Websites

- Tiltyard (web site) - allows users to register players for automatic round robin competition against other general game playing programs. Warning: Games on this website do not necessarily comply with the current General Game Playing standard.

- GGPorg (web page) - General website on GGP. Contains information on how to develop software for GGP. Warning: Games on this website do not necessarily comply with the current General Game Playing standard.

---

Questions and Comments
Gamemaster

Protocol: autoplayer
Identifier: legal 🕊️
Gamemaster

Protocol: autoplayer
Identifier: indy 🍀
Gamemaster

Protocol: autoplayer
Identifier: random ✧

Clear  Connect  Disconnect
Gamemaster

Protocol: autoplayer
Identifier: lara  

Clear  Connect  Disconnect
Game Players
(1) Download an existing player, modify, load in browser.

(2) Create player file, go to Autoplayeropen, load file.

(3) Create standalone process
   e.g. download player.js, modify, load into NodeJS
Game Management Applications (Local Games)

- Standaloneopen - human player interface and game manager in single web page
- Competitoropen - human player interface for competitions
- Autoplayeropen - automated player interface for competitions
- Manageropen - game manager for competitions

Browser-Based Players

- Legal - Legal player
- Random - Random player
- OneStep - One Step player
- Minimax - Full Minimax player
- MinimaxDepth - Minimax player with fixed depth
- MinimaxId - Minimax player with iterative deepening
- Greedy - Greedy player
- MCS - Monte Carlo Search player

JavascriptCode

- EpilogJS - Logic Programming Interpreter
- ParametricJS - Code for basic players
- PlayerJS - Player setup for NodeJS

Older General Game Playing Websites

- Tiltyard (web site) - allows users to register players for automatic round robin competition against other general game playing programs. Warning: Games on this website do not necessarily comply with the current General Game Playing standard.
- GGPorg (web page) - General website on GGP. Contains information on how to develop software for GGP. Warning: Games on this website do not necessarily comply with the current General Game Playing standard.
Gamemaster

Protocol: autoplayer
Identifier: legal 🌻

Clear  Connect  Disconnect
Gamemaster

Protocol: autoplayer
Identifier: egghead 🍼
<!DOCTYPE html>
<html>
<head>
    <title>Legal</title>
    <script type="text/javascript" src="http://epilog.stanford.edu/javascript/epilog.js"></script>
    <script type="text/javascript" src="http://gamemaster.stanford.edu/javascript/autoplayer.js"></script>
</head>
<body>

// Player parameters
var player = 'legal';
var manager = 'manager';

function player() {
    (player = readprompt("What is your player's identifier?")));
    document.getElementById('player').innerHTML = player;
    return true;
}

function manager() {
    (manager = readprompt("What is your manager's identifier?")));
    document.getElementById('manager').innerHTML = manager;
    return true;
}

// End of player parameters

function ping () {
    (return 'ready')
}

function start (r,s,sc,pc)
    (role = r);
    library = firstrules(r,s.slice(0));
    roles = findroles(library);
    state = findinitials(library);
    startclock = sc;
    pclock = pc;
    return 'ready';
}

function play (move)
    (if (move=='null') (compexecutemove(state,library));
    return playlegal(role))
}

function stop (move)
    (return false)

function abort ()
    (return false)

// legal

function playlegal (role)
    (return findlegalx(state,library))

// Basics

function findroles (rules)
    (return compfinds('R', seq(role), 'R'), seq(), rules)
Listen Loop
  Receives messages from Game Manager
  Calls appropriate event handler
  Sends result back to Game Manager

Event handlers
  each handles a different type of message
  responds with time bounds
function doreceive ()
{var messages = receive(player); // receive messages
messages = readdata(messages); // parse messages
for (var i=0; i<messages.length; i++)
{var timestamp = messages[i][1]; // timestamp
var sender = messages[i][2]; // sender
var receiver = messages[i][3]; // receiver
var message = messages[i][4]; // content
var response = ggpeval(message);
if (response) // reply to message
{var msg = seq('reply',message[1],response);
var record = send(player,sender,grind(msg))};
return true}

function ggpeval (msg)
{if (symbolp(msg)) {return false};
if (msg[0]==='ping') {return ping();};
if (msg[0]==='start') {return start(msg[2],msg[3],msg[4],msg[5])};
if (msg[0]==='play') {return play(msg[2])};
if (msg[0]==='stop') {return stop(msg[2])};
if (msg[0]==='abort') {return abort()};
return false}
function ping()
    { ...code that calls predefined subroutines... }

function start(r, rs, sc, pc)
    { ...code that calls predefined subroutines... }

function play(move)
    { ...code that calls predefined subroutines... }

function stop(move)
    { ...code that calls predefined subroutines... }

function abort()
    { ...code that calls predefined subroutines... }
findroles(rule)
findbases(rule)
findactions(rule)
findinits(rule)
findcontrol(state, rule)
findlegalp(action, state, rule)
findlegalx(state, rule)
findlegals(state, rule)
findreward(role, state, rule)
findterminalp(state, rule)
simulate(action, state, rule)
Simple behavior
  Maintains current state of the game.
  On each step, selects first legal action it finds.

NB: Selects same action every time in same state.
Implementation

```javascript
var role, roles, state, library, startclock, playclock;

function ping ()
  {return 'ready'}

function start (r, rs, sc, pc)
  {role = r;
   library = rs;  // definerules([], rs.slice(1));
   roles = findroles(library);
   state = findinits(library);
   startclock = sc;
   playclock = pc;
   return 'ready'}

function play (move)
  {if (move!==nil) {state = simulate(move, state, library)};
   if (findcontrol(state, library) !== role) {return false};
   return findlegalx(state, library)}

function stop (move)
  {return false}

function abort ()
  {return false}
```
Simple behavior
  Maintains current state of the game.
  On each step, selects random legal action.

NB: May take different action each time in a state.
var role, roles, state, library, startclock, playclock;

function ping ()
    {return 'ready'}

function start (r,rs,sc,pc)
    {role = r;
     library = rs;  // definerules([],rs.slice(1));
     roles = findroles(library);
     state = findinits(library);
     startclock = sc;
     playclock = pc;
     return 'ready'}

function play (move)
    {if (move!==nil) {state = simulate(move,state,library)};
     if (findcontrol(state,library) !== role) {return false};
     var actions=findlegals(state,library);
     return actions[randomindex(actions.length)]}

function stop (move)
    {return false}

function abort ()
    {return false}
Legal Player:

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

Random Player:

```javascript
function play (move)
    {if (move!==nil) {state = simulate(move,state,library)};
        if (findcontrol(state,library)!==role) {return false};
        var actions=findlegals(state,library);
        return actions[randomindex(actions.length)]}
```
Random Players are no “smarter” than legal players. Appear more interesting because unpredictable.

Sometimes avoid traps that befall consistent players like Legal.

Often used as a comparison to show that a player or method performs statistically better than chance.

Random players take slightly more time than legal. Compute all legal actions rather than just one Noticeable only on games with many legal actions.
Maximizer
Single Player Games

Example:


t

Terminology:

Single Player Games = Puzzles
Playing Single Player Games = Problem Solving

Easier than multiple player games
World static (except when single player acts)
Changes determined entirely by player’s actions
Complete Information

Game Description tells player:
- Initial state
- Legal actions in every state
- Results of performing every action in every state
- Reward for every state
- Whether or not a state is terminal
Small Games

Resources
  Sufficient time and space
to search the entire game tree

Results
  Players can find optimal actions on each time step
  NB: Sometimes possible without searching entire tree

This rules out puzzles like Rubik’s Cube, which requires more complicated techniques, some of which are discussed in subsequent lessons.
Implementation

```javascript
var role, roles, state, library, startclock, playclock;

function ping ()
    {return 'ready'}

function start (r,rs,sc,pc)
    {role = r;
    library = rs;   // definerules([],rs.slice(1));
    roles = findroles(library);
    state = findinits(library);
    startclock = sc;
    playclock = pc;
    return 'ready'}

function play (move)
    {if (move!==nil) {state = simulate(move,state,library)};
    if (findcontrol(state,library)!==role) {return false};
    return bestmove(state)}

function stop (move)
    {return false}

function abort ()
    {return false}
```
Legal Player:

function play (move)
  {if (move!==nil) {state = simulate(move,state,library)};
  if (findcontrol(state,library) !== role) {return false};
  return findlegalx(state,library)}

Maximizer:

function play (id,move)
  {if (move!='nil') {state=simulate(move,state,ruleset)};
  if (findcontrol(state,library) !== role) {return false};
  return bestmove(state)}
function bestmove (state)
    {var actions = 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 result = maxscore(newstate);
         if (result==100) {return actions[i]};
         if (result>score)
             {score = result; action = actions[i]};
     return action}
function maxscore (state)
    {if (findterminalp(state,ruleset))
        {return findreward(role,state,ruleset)};
    var actions = findlegals(role,state,ruleset);
    var score = 0;
    for (var i=0; i<actions.length; i++)
        {var newstate = findnexts(actions[i],state,rules);
            var result = maxscore(newstate);
            if (result==100) {return 100};
            if (result>score) {score = result};
        }
    return score}
Minimax
Multiple Player Games

Example:

More complicated than single-player games
Changes depend on actions of others
and those actions cannot be controlled
So player must consider all possible actions of others
Fixed Sum Games
Total reward in all states is the same
For one player to get more, the others must get less

Also called zero sum in cases where sum is 0
Note that, in GGP, all rewards are non-negative.
However, can be scaled - to 50, for example.

Many common games are zero sum
e.g. Chess - winner and loser

Variable Sum Games
Possible for one player to get more
without other players getting less
Some games are even cooperative
Resources

Sufficient time and space
to search the entire game tree

Results

Players can find optimal *strategy*
Not necessarily *sequential*, as with single player games
Plans may be *conditional* on the actions others
NB: *Sometimes* possible without searching entire tree

Small size rules out games like Chess and Othello, which require more complicated techniques, some of which are discussed in subsequent lessons.
Opponent Modeling
May assume opponent will play rationally

but

Assumption may be wrong

Also, players do not know identities of other players

Common Alternative - pessimistic / conservative
Assume the other player will do the worst possible thing

*Not maximizing its score, minimizing your score*
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 Tree

Max node

Min node
Bipartite Game Tree
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.

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

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

$$value(p, s) = \min(\{value(p, simulate(b, s)) \mid legal(b, s)\})$$
Bipartite Game Tree
Implementation

var role, roles, state, library, startclock, playclock;

function ping ()
    {return 'ready'}

function start (r,rs,sc,pc)
    {role = r;
     library = rs;   // definerules([],rs.slice(1));
     roles = findroles(library);
     state = findinits(library);
     startclock = sc;
     playclock = pc;
     return 'ready'}

function play (move)
    {if (move!==nil) {state = simulate(move,state,library)};
     if (findcontrol(state,library)!==role) {return false};
     return bestmove(state)}

function stop (move)
    {return false}

function abort ()
    {return false}
function bestmove (state)
    {var actions = findlegals(role,state,ruleset);
        var action = actions[0];
        var score = 0;
        for (var i=0; i<actions.length; i++)
            {var newstate = simulate(actions[i],state,library);
                var newscore = minimax(newstate);
                if (result>score)
                    {score = result; action = actions[i]};
        return action}
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)}
function maximize (state)
{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 = minimax(newstate);
    if (newscore>score) {score = newscore};
  }
  return score}

function minimize (state)
{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 = minimax(newstate);
    if (newscore<score) {score = newscore};
  }
  return score}
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.
Bounded Minimax Example
As stated
  100 is the limiting case for maxscore
  0 is the limiting case for minscore

Other possibilities
  Satisficing - fixed minimal score all that is needed
  Fixed sum game - 51 is sufficient
Alpha-Beta Search - Same as Bounded Minimax except that bounds are computed dynamically and passed along as parameters.

If partial result of min node less than alpha, can only decrease score and player need not consider.

If partial result of max node greater than beta, can only increase score and opponent will not allow.
Alpha-Beta Example
Alpha-Beta Example
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.

For example, it could in some cases reduce a tree with branching factor of 25 to branching factor of 5.