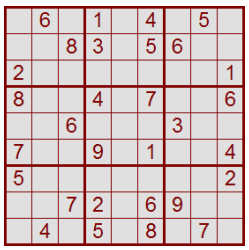
General Game Playing

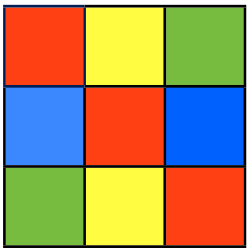
Small Single Player Games

Single Player Games

Example:



	1	3
4	2	5
7	8	6



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

We start our in-depth tour of general game playing by looking at single player games, such as Sudoku, Sliding Tile Puzzles, and Rubik's Cube. In the game-playing community, these are often called puzzles rather than games; and the process of solving such puzzles is often called problem-solving rather than game-playing. Puzzles are simpler than multiple player games because everything is under the control of the single player. The world is static, except when the player acts; and changes to the world are determined entirely by the current state and the actions of the player.

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

In this lesson, as in most of the course, we assume that the player has complete information about the puzzle. We assume that it knows the initial state; it knows all of its legal actions in every state; it knows the effects of its actions in every state; for every state, it knows its reward; and, for every state, it knows whether or not the state is terminal.

Small Games

Resources

Sufficient time and space
to search the entire game tree

Results

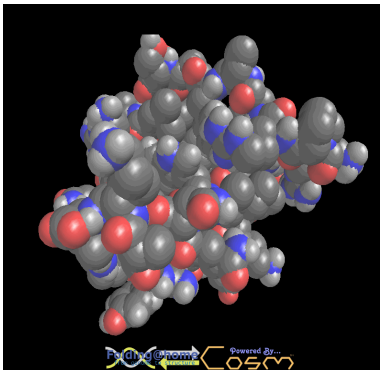
Players can find optimal actions on each time step
NB: *Sometimes* possible without searching entire tree

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

In this lesson, we also assume the games are small, i.e. the player has sufficient space and time to search the entire game tree. This guarantees that the player can find optimal actions to perform. That said, as we shall see, it is sometimes possible to find optimal actions even without searching the entire game tree.

Still Interesting

Many real world problems can be viewed as puzzles



Techniques for more complicated settings are variations

- Multiple player games
- Large game trees
- Games with incomplete information

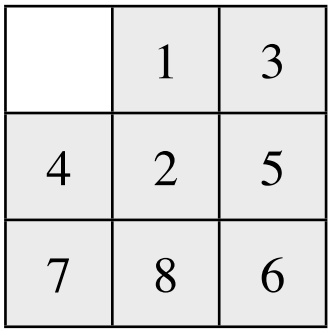
Despite these strong assumptions (just one player, complete information, and the availability of adequate time to search the game tree), the study of single player games is a good place to start our look at general game playing. First of all, many real world problems can be cast as single player games with these same restrictions, such as finding possible protein foldings as suggested by the illustration here. More importantly for us, as we shall see, the techniques we examine later can be viewed as more elaborate versions of the basic techniques introduced here.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

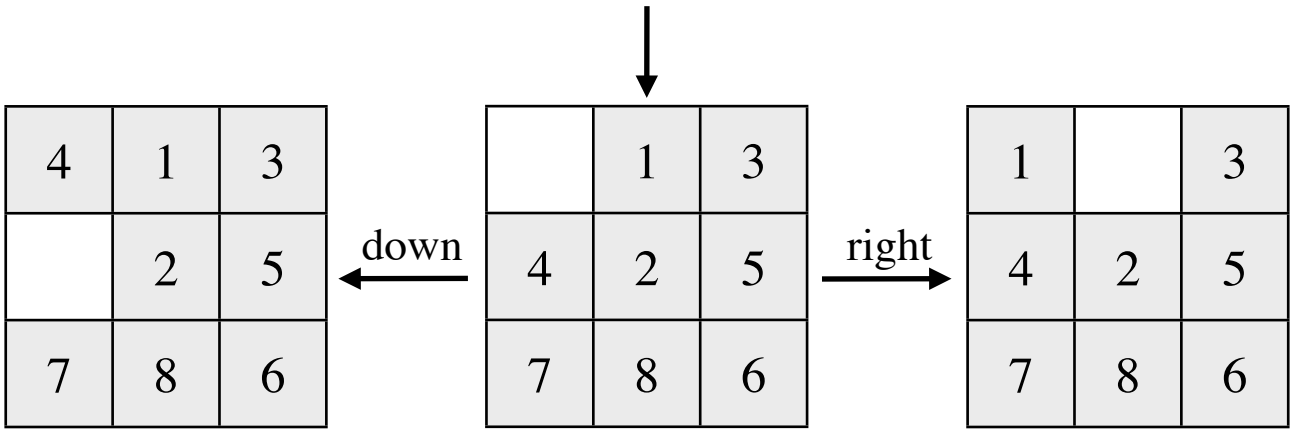
Eight Puzzle

Eight Puzzle



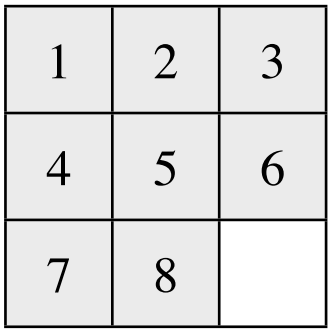
Let's begin this lesson with a look at a common single player game called 8-puzzle. 8-puzzle is a sliding tile game. The game board is a 3x3 square with numbered tiles in all but one of the cells.

Legal Moves



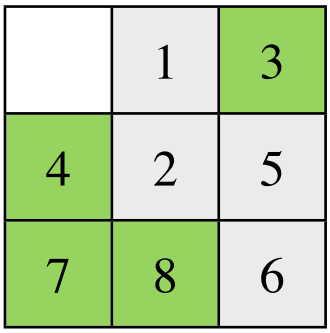
The state of the game is modified by sliding numbered tiles into the empty space from adjacent cells, thus moving the empty space to a new location. There are four possible moves - moving the empty space up, down, left, or right. Obviously, not all moves are possible in all states. The states shown on the left and right here illustrate the possible moves from the state shown in the center.

Ultimate Objective

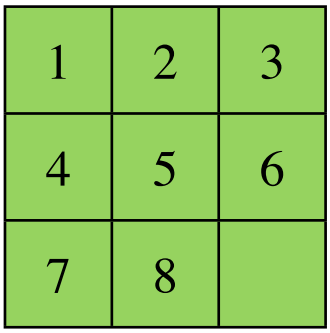


The ultimate object of the game is to place the tiles in order and position the empty square in the lower right cell, as shown here. The game terminates after 8 moves or when all of the tiles are in the right positions, whichever comes first.

Partial Credit



Reward: 40 points



Reward: 100 points

Partial credit is given for states that approximate the ultimate goal, with 10 points being allocated for each numbered tile in the correct position and 20 points being allocated for having the empty tile in the correct position. For example, the first state shown here is worth 40 points; and the goal state is worth 100 points.

GDL for Three Puzzle

```
role(robot)

base(cell(M,N,T)) :-
  index(M) &
  index(N) &
  tile(T)

base(step(1))

base(step(N)) :-
  successor(M,N)

input(robot,up)
input(robot,down)
input(robot,left)
input(robot,right)

init(cell(1,1,b))
init(cell(1,2,3))
init(cell(2,1,2))
init(cell(2,2,1))
init(step(1))

legal(robot,left) :-
  true(cell(M,2,b))

legal(robot,right) :-
  true(cell(M,1,b))

legal(robot,up) :-
  true(cell(2,N,b))

legal(robot,down) :-
  true(cell(1,N,b))
```

```
next(cell(1,N,b)) :-
  does(robot,up) &
  true(cell(2,N,b))

next(cell(2,N,b)) :-
  does(robot,down) &
  true(cell(1,N,b))

next(cell(M,1,b)) :-
  does(robot,left) &
  true(cell(M,2,b))

next(cell(M,2,b)) :-
  does(robot,right) &
  true(cell(M,1,b))

next(cell(2,N,X)) :-
  does(robot,up) &
  true(cell(2,N,b)) &
  true(cell(1,N,X))

next(cell(1,N,X)) :-
  does(robot,down) &
  true(cell(1,N,b)) &
  true(cell(2,N,X))

next(cell(M,2,X)) :-
  does(robot,left) &
  true(cell(M,2,b)) &
  true(cell(M,1,X))

next(cell(M,1,X)) :-
  does(robot,right) &
  true(cell(M,1,b)) &
  true(cell(M,2,X))
```

```
next(cell(M,N,W)) :-
  does(robot,up) &
  true(cell(X,Y,b)) &
  true(cell(M,N,W)) &
  distinct(Y,N)

next(cell(M,N,W)) :-
  does(robot,down) &
  true(cell(X,Y,b)) &
  true(cell(M,N,W)) &
  distinct(Y,N)

next(cell(M,N,W)) :-
  does(robot,left) &
  true(cell(X,Y,b)) &
  true(cell(M,N,W)) &
  distinct(X,M)

next(cell(M,N,W)) :-
  does(robot,right) &
  true(cell(X,Y,b)) &
  true(cell(M,N,W)) &
  distinct(X,M)

next(step(N)) :-
  true(step(M)) &
  successor(M,N)
```

```
goal(robot,100) :-
  true(cell(1,1,1)) &
  true(cell(1,2,2)) &
  true(cell(2,1,3))

goal(robot,0) :-
  ~true(cell(1,1,1))

goal(robot,0) :-
  ~true(cell(1,2,2))

goal(robot,0) :-
  ~true(cell(2,1,3))

terminal :- true(step(7))

index(1)
index(2)

tile(1)
tile(2)
tile(3)
tile(b)

successor(1,2)
successor(2,3)
successor(3,4)
successor(4,5)
successor(5,6)
successor(6,7)
```


As with other games, it is possible to describe Eight Puzzle in GDL. Here are the rules for Three Puzzle (a version of Eight Puzzle on a 2x2 board) with no partial credit. The rules for Eight Puzzle *with* partial credit are analogous but a little verbose. We won't go through the details of either rule set in this lesson, but it may be worth your while to pause and look over the rules to be sure you understand them.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Compulsive Deliberation

Compulsive Deliberation

Play Method
- Compute best move
- Execute
- Update state

Pure compulsive deliberation
- State is updated and move is computed but
- No information preserved from one step to next

Obviously wasteful but ...
- Does not hurt if sufficient resources available
- Simple and template for more sophisticated methods

Compulsive Deliberation is a particularly simple approach to game playing. On each step, the player examines the then-current game tree to determine its best move for that step; and it then makes this move. It repeats this process on the next step and so forth until the end of the game.

In pure compulsive deliberation, each step of the computation is independent of every other step. No data computed during any step is accessible on subsequent steps. The player treats each step as if it were a new game.

This is obviously wasteful, but it does not hurt so long as there is enough time to do the repeated calculations. We start with this method because it is simple to understand and at the same time serves as a template for the more sophisticated, less wasteful methods to come.

Implementation

```
var ruleset;
var role;
var roles;
var state;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  return 'ready'}

function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!=‘nil’) {state=findnexts(move,state,ruleset)};
  return undoesify(bestmove(state))}

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

function abort (id)
  {return 'done'}
```

Using the basic subroutines provided in the GGP starter pack, building a compulsive deliberation player is not difficult. The implementation looks like this. As shown, it is almost identical to our implementation of legal and random players.

Play Handler

Legal Player:

```
function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!='nil') {state=findnexts(move,state,ruleset)};
  return undoesify(legalx(role,state,ruleset))}
```

Compulsive Deliberation Player:

```
function play (id,actions)
 {var move = doesify(roles,actions);
  if (move!='nil') {state=findnexts(move,state,ruleset)};
  return undoesify(bestmove(state))}
```

The only difference lies in the play handler. In selecting an action, a legal player uses legalx to compute a legal move for the given state. In compulsive deliberation, the play handler instead uses a subroutine called bestmove that does a more sophisticated computation.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

Before looking at bestmove, let's look at a slightly simpler subroutine called maxscore. maxscore takes a state as argument and returns the best score that the player can obtain by any sequence of actions in the specified state. Let's see how it works.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

As its first step, the procedure checks whether the given state is terminal. If so, then the best possible score is the reward for the specified state. It computes this by calling the predefined findreward subroutine on the role, the state, and the rule set.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

If the state is not terminal, then it tries each of the actions legal in that state, computes the maximum score for the state that results from executing that action, and returns the best score it finds. The first step in doing this is to compute a list of all legal actions.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

It initializes its score to 0.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

It then loops over the possible actions.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

Since, in general, there can be multiple players, findnexts takes as arguments a list of actions of all players. In this case, we have a single player game, so the player creates a list of just one element.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

The player then uses findnexts to compute the next state resulting from this move in the current state.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

It then finds the maxscore for the successor state.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

If the result is 100, then it simply returns that value, as there is no way to get more than 100 points. Otherwise, if the result is greater than the current score, it updates the score and goes on.

maxscore

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return 100};
       if (result>score) {score = result}};
  return score}
```

Finally, if it has not encountered a 100 value in this process, it returns the current score, which is by construction the maximum score for all possible actions.

bestmove

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return actions[i]};
       if (result>score)
          {score = result; action = actions[i]}};
  return action}
```

Okay, now that we have maxscore, let's return to bestmove. The definition uses maxscore and is actually quite similar to maxscore.

bestmove

```
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 move = [actions[i]];
       var newstate = findnexts(move,state,rules);
       var result = maxscore(newstate);
       if (result==100) {return actions[i]};
       if (result>score)
          {score = result; action = actions[i]}};
  return action}
```

There are just a few differences. First of all, bestmove is not itself recursive, though it calls maxscore, which *is* recursive. bestmove does not need to check whether the state it is given is terminal, because it would not be called if the game were in a terminal state. It initializes a variable called action to the first legal action. And it then behaves like maxscore, trying each possible action to see if it can fine one with a higher score than any previous action it has seen. If it ever encounters a maxscore of 100, it simply returns the corresponding action. Otherwise, it proceed until it had tried all actions, at which point it returns the action with the highest maxscore that it has seen.


GENERAL
GAME
PLAYING


GENERAL
GAME
PLAYING

Sequential Planning

Problems with Compulsive Deliberation

Problems

- Wasteful - work repeated on every step
- Computation done during play clock

Compulsive deliberation is wasteful in that computations are repeated unnecessarily. Once a player is able to find a path to a terminal state with maximal reward, it should not have to repeat that computation on every step. Sequential planning is the antithesis of compulsive deliberation in which no work is repeated. Once a sequential planner finds a good path, it simply saves the sequence of actions along that path and then executes the actions step by step until the game is done without any further deliberation.

Sequential Planning

Start Method - Compute optimal plan
Play Method - Execute action *n* on the *n*-th step

Comments
- Plan computed just once (during start clock)
- Benefits from long start clocks
- Great for short play clocks

A sequential planning player is one that produces an optimal sequential plan (usually during the start clock) and then executes the steps of that plan during game play.

Sequential planning has multiple benefits relative to compulsive deliberation. First and most obvious is that it is not as wasteful, since it searches the game tree just once. If the start clock is sufficiently long, teh planning can be done entirely during the startclock. After that the execution time is very low, since all the player needs on a step to do is to look up the action for that step without doing any search whatsoever.

Note that, although the planning is usually done during the start-up period of a game, it can also be done during regular game play. It is also possible to mix sequential planning with other techniques. For example, in the case of large games, a player might play randomly during the initial part of a game and then switch to sequential planning once the game tree becomes small enough. Of course, in this last case, the player's ability to succeed depends on the strategy used before sequential planning commences.

Sequential Plans

Sequential Plan - sequence of action that leads from initial state to terminal state such that (1) every action is legal in the state in which that action is performed and (2) none of intermediate states is terminal.

Optimal - no other sequential plan has greater reward

Let's start our look at sequential planning with a couple of definitions. A sequential plan for a single player game is a sequence of actions that leads from the initial state of the game to a terminal state such that (1) every action in the sequence in legal in the state in which that action is performed and (2) none of the intermediate states produced during the execution is terminal. A sequential plan is optimal if and only if no other sequential plan produces a greater final reward.

Examples

Eight Puzzle

	1	3
4	2	5
7	8	6

Legal, Minimal, Optimal Plans

```
[right,down,right,down]                          100
[right,down,left,right,right,down]               100
[right,down,left,right,left,right,right,down]    100
```

Legal, Minimal, Non-Optimal Plan

```
[right,left,right,left,right,left,right,left]     40
```

Here are some example of sequential plans for Eight Puzzle. The first plan prescribes a move to the right followed by a move down followed by another move to the right and another move down. This leads to a state in which all tiles are in their goal positions and the empty cell is on the lower right; and the value for this state is 100. Of course, this is not the only plan that works. The player could also move right, down, left, right, right, and down and arrive at the same state. Or it could move right, down. left, right, left, right, right, and down to get there as well. These latter two plans are longer but they are both optimal in that they produce a terminal state with maximal value. By contrast, the sequential plan right, left, right, left, right, left, right, left. This plan leads to a terminal state since any plan with eight steps is terminal. However, it is not an optimal plan because the resulting reward is only 40 points and there are other plans that produce higher value.

Implementation

```
var ruleset;
var role;
var roles;
var state;
var plan;
var step;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  plan = reverse(bestplan(state));
  step = 0;
  return 'ready'}

function play (id,move)
 {var action = plan[step]; step++; return undoesify(action)}

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

function abort (id)
  {return 'done'}
```

The implementation of a sequential planner is very similar to that for compulsive deliberation.

Implementation

```
var ruleset;
var role;
var roles;
var state;
var plan;
var step;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  plan = reverse(bestplan(state));
  step = 0;
  return 'ready'}

function play (id,move)
 {var action = plan[step]; step++; return undoesify(action)}

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

function abort (id)
  {return 'done'}
```

We set up a couple of additional global variables = one to hold the plan and the other to keep track of the current step.

Implementation

```
var ruleset;
var role;
var roles;
var state;
var plan;
var step;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  roles = findroles(rules);
  state = findinits(rules);
  plan = reverse(bestplan(state));
  step = 0;
  return 'ready'}

function play (id,move)
 {var action = plan[step]; step++; return undoesify(action)}

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

function abort (id)
  {return 'done'}
```

During the startclock, the player uses the bestplan subroutine to produce a sequential plan. (It has to reverse the plan, since, as we shall see, bestplan builds the plan backward. The plan is then stored in the plan variable, and the step counter is initialized to 0.

Implementation

```
var ruleset;
var role;
var state;
var plan;
var step;

function info (id)
  {return 'ready'}

function start (id,player,rules,sc,pc)
 {ruleset = rules;
  role = player;
  state = inits(rules);
  plan = reverse(bestplan(role,state,ruleset)[1]);
  step = 0;
  return 'ready'}

function play (id,move)
 {var action = plan[step]; step++; return undoesify(action)}

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

function abort (id)
  {return 'done'}
```

The play handler on each step simply reads off the action corresponding to that step, updates the step counter, and returns the action for that step.

bestplan

```
function bestplan (state)
 {if (findterminalp(state,ruleset))
     {return [findreward(role,state,description),[]]};
  var actions = findlegals(role,state,rules);
  var newstate = findnexts([actions[0]],state,ruleset);
  var result = bestplan(newstate);
  var score = result[0];
  var plan = result[1];
  plan[plan.length] = actions[0];
  for (var i=1; i<actions.length; i++)
      {newstate = findnexts([actions[i]],state,ruleset);
       var result = bestplan(newstate);
       if (result[0]>score}
          {score = result[0];
           plan = result[1];
           plan[plan.length] = actions[i]};
  return [score,plan]}
```

The bestplan subroutine is analogous to maxscore. It takes a state as argument; but, instead of returning a simple score, it returns a pair consisting of a score and a plan to achieve that score. Let's see how this works.

bestplan

```
function bestplan (state)
 {if (findterminalp(state,ruleset))
     {return [findreward(role,state,description),[]]};
  var actions = findlegals(role,state,rules);
  var newstate = findnexts([actions[0]],state,ruleset);
  var result = bestplan(newstate);
  var score = result[0];
  var plan = result[1];
  plan[plan.length] = actions[0];
  for (var i=1; i<actions.length; i++)
      {newstate = findnexts([actions[i]],state,ruleset);
       var result = bestplan(newstate);
       if (result[0]>score}
          {score = result[0];
           plan = result[1];
           plan[plan.length] = actions[i]};
  return [score,plan]}
```

As in maxscore, the first step is to check whether the state is terminal. If so, then the procedure simply computes the player's reward for that state and returns that score paired with an empty plan, i.e. an empty list of actions.

bestplan

```
function bestplan (state)
 {if (findterminalp(state,ruleset))
     {return [findreward(role,state,description),[]]};
  var actions = findlegals(role,state,rules);
  var newstate = findnexts([actions[0]],state,ruleset);
  var result = bestplan(newstate);
  var score = result[0];
  var plan = result[1];
  plan[plan.length] = actions[0];
  for (var i=1; i<actions.length; i++)
      {newstate = findnexts([actions[i]],state,ruleset);
       var result = bestplan(newstate);
       if (result[0]>score}
          {score = result[0];
           plan = result[1];
           plan[plan.length] = actions[i]};
  return [score,plan]}
```

Otherwise, it computes all legal actions for the specified state.

bestplan

```
function bestplan (state)
 {if (findterminalp(state,ruleset))
     {return [findreward(role,state,description),[]]};
  var actions = findlegals(role,state,rules);
  var newstate = findnexts([actions[0]],state,ruleset);
  var result = bestplan(newstate);
  var score = result[0];
  var plan = result[1];
  plan[plan.length] = actions[0];
  for (var i=1; i<actions.length; i++)
      {newstate = findnexts([actions[i]],state,ruleset);
       var result = bestplan(newstate);
       if (result[0]>score}
          {score = result[0];
           plan = result[1];
           plan[plan.length] = actions[i]};
  return [score,plan]}
```

It computes the nextstate corresponding to the first of these actions and computes the best score and bestplan for that state.

bestplan

```
function bestplan (state)
 {if (findterminalp(state,ruleset))
     {return [findreward(role,state,description),[]]};
  var actions = findlegals(role,state,rules);
  var newstate = findnexts([actions[0]],state,ruleset);
  var result = bestplan(newstate);
  var score = result[0];
  var plan = result[1];
  plan[plan.length] = actions[0];
  for (var i=1; i<actions.length; i++)
      {newstate = findnexts([actions[i]],state,ruleset);
       var result = bestplan(newstate);
       if (result[0]>score}
          {score = result[0];
           plan = result[1];
           plan[plan.length] = actions[i]};
  return [score,plan]}
```

It then searches the remaining possible actions.

bestplan

```
function bestplan (state)
 {if (findterminalp(state,ruleset))
     {return [findreward(role,state,description),[]]};
  var actions = findlegals(role,state,rules);
  var newstate = findnexts([actions[0]],state,ruleset);
  var result = bestplan(newstate);
  var score = result[0];
  var plan = result[1];
  plan[plan.length] = actions[0];
  for (var i=1; i<actions.length; i++)
      {newstate = findnexts([actions[i]],state,ruleset);
       var result = bestplan(newstate);
       if (result[0]>score}
          {score = result[0];
           plan = result[1];
           plan[plan.length] = actions[i]};
  return [score,plan]}
```

For each, it computes the nextstate, gets the best score and best plan for that state, and compares the score to the best score seen so far. If the score is better, then it saves that score and the corresponding plan (with the action appended to the end).

bestplan

```
function bestplan (state)
 {if (findterminalp(state,ruleset))
     {return [findreward(role,state,description),[]]};
  var actions = findlegals(role,state,rules);
  var newstate = findnexts([actions[0]],state,ruleset);
  var result = bestplan(newstate);
  var score = result[0];
  var plan = result[1];
  plan[plan.length] = actions[0];
  for (var i=1; i<actions.length; i++)
      {newstate = findnexts([actions[i]],state,ruleset);
       var result = bestplan(newstate);
       if (result[0]>score}
          {score = result[0];
           plan = result[1];
           plan[plan.length] = actions[i]};
  return [score,plan]}
```

After all actions are executed, bestplan returns a pair of the best score and the best plan.


GENERAL
GAME
PLAYING