6.034 Artificial Intelligence, Fall 2006
Prof. Patrick H.Winston

Problem Set 3

This problem set is due Wednesday, October 18th at 11:59 PM. If you have questions about it, ask the TA email list. Your response will probably come from a TA.

To work on this problem set, you will need to get the code, much like you did for earlier problem sets.

- The code can be downloaded from the assignments section.

Your answers for the problem set belong in the main file `ps3.scm`.

Things to remember

- Avoid using DrScheme's graphical comment boxes. If you do, take them out or use *Save definitions as text...* so that you submit a Scheme file in plain text.
- If you are going to submit your pset late, see the problem set grading policy.

1. Constraint propagation

1.1. Sudoku

In this problem you will write a program to solve Sudoku puzzles. This kind of puzzle has 81 cells arranged in a square grid which is subdivided into nine smaller squares. Each row, column, and small square must contain the digits 1 – 9.

There is a graphic that goes with this part of the problem. To ensure that it runs correctly, do the following in DrScheme:

- Open the **Language** menu.
- Select **Choose Language...**
- Under **PLT**, choose the language **Graphical (MrEd, includes MzScheme)**.
- Click OK.

Run the tester and make sure that the Sudoku window appears.

In general, Sudoku puzzles may require search in addition to constraint propagation. In this problem, however, you will never encounter a Sudoku board which cannot be solved with constraint propagation alone.

There are several methods defined in sudoku.scm that you may find useful. Here are the most relevant ones:

- `(get-cells board)` - get all the cell data structures from the board
- `(cells-in-same-row cell board)`, as well as `cells-in-same-col` and `cells-in-same-square` - given a cell data structure, returns the cells in the same row, column, or square
- `(cell-values cell)` -- a list of values this cell can have
- `(set-values! cell new-values)` -- set the possible values for a cell. `new-values` should be a list containing integers from 1 to 9.
- `(get-cell-by-number board n)`: looks up a cell by its unique index number, from 0 to 80. You can also use `get-row-by-number`, `get-col-by-number`, and `get-square-by-number` to get at the row, column, and square data structures. Look in `sudoku.scm` for more accessors like this if you need them.

There is also a function to draw the board in the graphics window:

- `(draw-board board)` - draw a Sudoku board on the screen.

You'll be working with just one board in this problem, and changing it as you go. To get a fresh copy of the board, call the procedure `(testing-board)`, which takes no arguments.

You may find this helpful to visualize the board as you work on the problem, because the scheme-list representation of the board is long. The following code snippet, for example, sets the first cell to value 7 and redraws the board:

```
(define theboard (testing-board))
(set-values! (get-cell-by-number theboard 0) '(7))
(draw-board theboard)
```

1.2. Identifying constraints

- Write (can-propagate? cell). cell is a cell data structure. This function should return whether a cell in the Sudoku puzzle can propagate constriants onto other cells. (There are many tactics for solving Sudoku puzzles that involve looking at constraints in multiple cells, but in this problem, you should use the only constraint you can propagate by examining a single cell.)

- Write (propagate-one-cell cell others). cell is a cell data sructure. Others is a list of cell data structures that are either in the same row, column, or square as cell. cell does not appear anywhere in others. Your function should modify each cell in others which can be modified (i.e. constraints can propagate), and return #t if cells were modified, #f if no cells were modified.

1.3. Solving the puzzle

- Write (propagate-one-round board). board is a board data structure. Your function should apply (propagate-one-cell) to each of the cells in the board, and the groups

of corresponding cells in the same column, row, and square. Usefull tip: use cells-in-same-row, etc. These methods will return a list of cells sharing the same row, column or square as a cell. The returned list will not contain the cell argument.

- Write (constraint-propagate board count). This will be called initially with count=0. It should return the number of steps (propagate-one-rounds) that it took to resolve all of the cells in the puzzle to single values. The return value of this function ought to be small (less than 10) if your constraint propagation is working properly.

2. Game search

In this section, you'll be working with a game AI that uses minimax search, then alpha-beta search, to play a simple game.

2.1. Multiple choice

Answer the questions in ps3.scm. The questions are there to check your understanding, so you can verify your answers with the tester.

2.2. The eight-pawn game

This section of the problem set involves a chess-like game that uses only pawns on a 4x4 board. The trivially-solvable 3x3 version of this game is called "Hexapawn"; it was invented by Martin Gardner to demonstrate AI learning on a very small game tree.

A *pawn* in chess is a piece that can only move in one direction across the board: white pawns only move up, and black pawns only move down. Ordinarily, they move in straight lines, remaining in the same column. A pawn can't move forward in this way if there's already another pawn, of either color, occupying the space in front of it.

In addition to these straight moves, though, pawns can *capture* diagonally. If there is a piece of the opposite color on one of the two squares that are diagonally in front of a pawn, it can capture the other pawn by moving onto its square, which removes the captured pawn from the board. Diagonal moves can only be made when capturing, and this is the only way that a pawn can change columns.

The pawns in this game are not allowed to move two spaces forward on their first move, like they can in chess.

In the eight-pawn game, two players begin with 4 pawns each on opposite sides of the board. There are two ways to win: to move one of your pawns to the opposite side of the board, or to leave your opponent in a situation where he can't move.

Here are some examples. These boards look the same as they will look in the Scheme program. `W` represents a white pawn, `B` represents a black pawn, and `-` indicates an empty space.

```
(B B B B)
(- - - -)  This is the starting position.
(- - - -)
(W W W W)

(B B B B)
(- - - -)  White makes the first move, advancing his pawn
(- - W -)  in the third column.
(W W - W)

(B - B B)
(- B - -)  Black advances her pawn in the second column.
(- - W -)
(W W - W)

(B - B B)
(- W - -)  White captures by moving the pawn diagonally forward
(- - - -)  and to the left. Black has two pieces that can capture
(W W - W)  this piece in response.
(- - B -)
(- B W B)  This situation might arise late in the game. None
(B W - W)  of these pawns can move. Whoever's turn it is loses.
(W - - -)
(B - - B)
(- - B W)  Another situation that can occur after several moves.
(B W - -)  It's black's turn to move.
(- - W -)

(B - - B)
(- - B W)  Black moves a pawn into the bottom row and wins.
(- W - -)  Pawned!
(B - W -)
```

The code in `8pawns.scm` enforces the rules of the game and lets you know what moves are valid from a given position.

2.2.1. Playing the game

You can get a feel for how the game works by playing it against the computer. For example, by uncommenting this line in ps3.scm, you can play white, while a computer player that does minimax search to depth 5 plays black.

```
(play-game startpos (human-player) (chatty-minimax-player 5 simple-evaluate))
```

For each move, the program will prompt you to make a choice, by choosing what location to move a piece to. (In a few cases, this will be ambiguous. These cases aren't

common enough to matter, though. If it helps, the pieces you've moved more recently tend to be near the top of the list.)

Your choices may look like this:

```
(B B - B)
(- - B -)
(- - - W)
(W W W -)

1: (W 4 3)
2: (W 3 3)
3: (W 1 2)
4: (W 2 2)
5: (W 3 2)

Which move do you want to make?
```

Choice 1, for example, represents moving a White pawn to column 4, row 3 (the third from the **bottom** of the board). This means moving the advanced pawn forward again. Choice 2 -- moving to 3, 3 -- indicates moving that same pawn diagonally to capture. The other three choices advance one of the other pawns from the first row to the second.

You'll get the hang of it. The point of this isn't to be a good interface to a game, it's to let you interact with the game at all without large amounts of support code.

The computer, meanwhile, is making the best move it can while looking ahead to depth 5 (three moves for itself and two for you). At each move, it outputs the value it gets from minimax. If this value is 0, it considers the game even so far. A value of 1000 means it's certain it will win, and -1000 means it's certain it will lose (assuming you play optimally).

2.3. The code

Here's an overview of the code in `8pawns.scm` that you may need to work with.

2.3.1. Abstractions

A **position** indicates the state of the board and whose turn it is. It's represented as a list containing three things:

1. the tag `'position`
2. the color that moves next, expressed as `'W` or `'B`
3. the **board**

A **board** is simply a list of all the **pieces** in play.

A **piece** represents the location and color of one piece. It is represented as a list of three items: the piece's color (W or B), the column the piece is in (1-4), and the row the piece is in (1-4).

This same structure is used to represent spaces on the board, whether or not there is actually a piece there. So the bottom left corner of the board is passed to functions as `(W 1 1)`, or maybe as `(B 1 1)`. The piece's color is disregarded if all that matters is the location.

Given these abstractions, the starting position of the game looks like this:

```
'(position W ((W 1 1) (W 2 1) (W 3 1) (W 4 1) (B 1 4) (B 2 4) (B 3 4)
(B 4 4)))
```

Don't confuse *positions* with *locations on the board*. A position represents the entire state of the game, while a location is a small part of the game.

2.3.2. Procedures

This is an overview of the most useful procedures in `8pawns.scm`. You may want to look at the code for a better understanding.

- `(valid-moves position)`: return a list of positions, representing all positions that the current player could put the game in after moving. If the current position has White to move, then all the returned positions will have Black to move, and vice versa.
- `(get-color position)`: which color has the next move in this position?
- `(get-board position)`: extract the board (list of piece locations) from a position.
- `(minimax-search position depth player max? evaluation-function)`: the main function that finds the minimax value of the game tree.
    - *position* is the position it should analyze.
    - *depth* is how many *more* levels it should descend in the game tree before using a static evaluation function. If *depth* is 0, then it will not explore any more moves and simply run the static evaluation function on the given position.
    - *player* is W or B, depending on whose point of view the board should be evaluated from. A position that's worth +3 for White is worth -3 for Black. *player* is not necessarily the player that the position says has the next move, because you need to explore your opponent's moves as well as your own.
    - *max?* is #t or #f. If it's #t, then the procedure will return the move that maximizes the heuristic value; if #f, it will return the move that minimizes the heuristic value.
    - *evaluation-function* is a procedure to use for static evaluation. You can change the behavior of minimax-search by giving it a different evaluation-

function. The provided procedure `simple-evaluate` is an example of an evaluation-function.

`minimax-search` returns a list of two things: the move to make (represented as the position that results from the move), and the heuristic value it calculated.

- `(simple-evaluate pos player depth)`: a simple static evaluation function. It takes in a position *pos*, the *player* whose point of view to use, and the *depth* remaining to search (information which this function doesn't use).
    - If the position is a loss for the player to move, the position is worth -1000 for that player.
    - Otherwise, a position is considered better for the current player if that player has more possible moves. The position is worth the number of possible moves, minus 3 (so that it is roughly centered around 0).
    - If the position is for the opposite player, `simple-evaluate` will negate the value it returns. If it's evaluating positions from White's point of view, for example, then a loss for Black should be worth 1000, not -1000.
- `(maximize-position p1 p2)`: p1 and p2 should be the kind of two-element `(position value)` lists that are returned from *minimax-search*. This function compares the two results and returns the position with the higher heuristic value.
- `(minimize-position p1 p2)`: Same as above, except it returns the position with the lower heuristic value.
- `(check-loss position)`: is this position a loss for the player to move? (There is no `check-win`, so just use `check-loss` on the other player's turn.)
- `(minimax-player depth evaluation-function)` (and related procedures)

  Several of these "player factory" procedure are provided. These procedures take in a depth to search to and an evaluation function to use, and return a "player" function, which will take in a position and return the position it wants to move to.

- `(human-player)`: Another "player factory", but here the "player" will stop and ask you what move to make.
- `(play-game position player1 player2)`: takes in a starting *position*, and two *players* that are returned from player factories. `play-game` plays out the eight-pawn game until one player wins, and returns the color of the winning player.

2.4. Just win already!

Okay, we're about to ask you to actually code stuff. You should stop skimming now.

If you end up in a losing position when playing against the computer, it may seem that the computer is toying with you: instead of making a single move that lets it win, it may prolong your misery by moving other pieces in such a way that it can still be guaranteed to win.

Similarly, if the computer sees that you are in a winning position, the computer may make an apparently stupid move that lets you win faster. It's not stupid from the computer's point of view, though - all moves are equally doomed, so why does it matter which one it makes?

This isn't how people generally play games. People want to win as quickly as possible, and lose slowly so that their opponent has several opportunities to mess up. A small change to the static evaluation function will make the computer play this way too.

- In `ps3.scm`, write a new evaluation function, `focused-evaluate`, which prefers winning positions when they happen sooner and losing positions when they happen later.

It will help to follow these guidelines:

- Leave the "normal" values (the ones that are like 1 or -2, not 1000 or -1000) alone. You don't need to change how the procedure evaluates positions that aren't guaranteed wins or losses.
- Don't return values less than -2000 or greater than 2000 -- we'll be using those numbers to mean "infinity" in a bit.
- Indicate a certain win with a value that is greater than or equal to 1000, and a certain loss with a value that is less than or equal to -1000. The "chatty" versions of the computerized players will comment when they see these kinds of values.

2.5. Alpha-beta search

The computerized players you've used so far would fit in well in the British Museum - they're evaluating all the positions down to a certain depth, even the unhelpful ones. You can make them search much more efficiently by using alpha-beta search.

It may help to read Chapter 6 of the textbook: Winston, Patrick H. *Artificial Intelligence*. 3rd ed. Reading, MA: Addison-Wellsley, 1992, ISBN: 0201533774 if you're unclear on the details of alpha-beta search.

- Write a procedure `alpha-beta-search`, which works like `minimax-search`, except it takes two additional arguments (`alpha` and `beta`) and does alpha-beta pruning to reduce the search space.
- You should always search your possible moves from left to right, in the order they are returned from `valid-moves`, and return a result for a particular level as soon as alpha is greater than or equal to beta.

This procedure is called by the player procedure `(alpha-beta-player depth evaluation-function)`. Initially, `alpha` is -2000 and `beta` is 2000 -- consider these to represent "negative infinity" and "positive infinity".

2.6. A better evaluation function

This problem is going to be a bit different. There's no single right way to do it - it will take a bit of creativity and thought about the game.

Your goal is to write a new procedure for static evaluation that outperforms the one we gave you. It will be tested against `simple-evaluate` on 10 different games, with search depths from 3 to 7. You get a point for each game you win. So a well-formed solution to this problem should easily get 5 points, and you can get a maximum of 10.

You might not get all 10 points - it might not even be feasible to get all 10 points - but we expect that the more thought you put into it, the higher a score you can get.

These points come entirely from the public tester. If there are hidden tests involving this problem, they won't hinge on whether you win; they may, for example, make sure your function doesn't produce errors or evaluate more positions within the static evaluator. (If you did, that wouldn't be very static.)

- Write a new procedure for static evaluation, `(better-evaluate pos player depth)`, which wins against `simple-evaluate` when searching to the same depth.

3. Survey

Please answer these questions at the bottom of your `ps3.scm` file:

- How many hours did this problem set take?
- Which parts of this problem set, if any, did you find interesting?
- Which parts of this problem set, if any, did you find boring or tedious?

(We'd ask which parts you find confusing, but if you're confused you should really ask a TA.)

When you're done, **run the tester** and submit your .scm files to your `6.034-psets/ps3` directory on Athena. If the tester dies with an error when it's run on your code, or if it stops and waits for user input, your submission will not count.

4. Problems?

If the tester dies immediately, complaining that `instantiate` is an unbound variable, you forgot to load the **Graphical** version of the MzScheme language, as described in section 1.1.