Programming Assignment 2 – Adversarial Search (Connect 4)

Sean G Meyer

Problem Description

The goal is to write a program to play Connect Four. The program will read, from standard input, a description of the current game state in JSON format and write, to standard outputs, its move in JSON format. The program must play a valid game of connect four, in particular it must make valid moves in response to the current game state. It needs to consistently beat a naive (random move) implementation of connect four, barring exceedingly rare cases.

Player Program Design

Two main algorithms are needed to solve this problem: an implementation of adversarial search (e.g. minimax), and a utility function that can estimate the value of various game states (since search will not be able to reach the end game, as it requires too many plies). I used an implementation of minimax called negamax, with alpha beta pruning. This was taken from Wikipedia, which has the following psuedo code for the algorithm:

```
function negamax(node, depth, Îś, Îš, color)
    if depth = 0 or node is a terminal node
        return color * the heuristic value of node

    childNodes := GenerateMoves(node)
    childNodes := OrderMoves(childNodes)
    bestValue := âĹŠâĹđ
    foreach child in childNodes
        v := âĹŠnegamax(child, depth âĹŠ 1, âĹŠÎš, âĹŠÎś, âĹŠcolor)
        bestValue := max( bestValue, v )
        Îś := max( Îś, v )
        if Îś âĽě Îš
            break
    return bestValue
```

I used this same algorithm for my implementation, adding parameters for player and opponent in order to do static evaluations for each player, switching back and forth as negamax runs. Beyond the search function, I had to come up with some way of evaluation different states of connect four, and assigning each one scores. This turned out to be the most difficult of the problem, by far,

and required quite a bit of code. Psuedo code for the algorithm at a very high level is shown below with the full implementation details saved for the next section.

```
function eval-by-positions-needed(node, positions)
    // First, a set is generated which contains lists of moves that
     ↪  lead to 4 in a row
    // These sets exclude currently placed pieces, so these sets
     ↪  represent future moves.
    // For example:
    //  if there is a single tile in the middle of the board then 7
     ↪  sets would
    //  be generated which represet all possible 4 in a rows from
     ↪  that spot
    win-sets = sets-of-winning-moves(grid positions)

    // The previous function can return sets which are supersets of
     ↪  each other
    // So we remove those (supersets must include a shorter win path
     ↪  by definition)
    win-sets = remove-supersets(win-sets)

    // Map the win-sets to a function that values the wins based on
     ↪  length
    // For example, a win requiring 1 piece is worth much more than
     ↪  a win of length 3
    win-values = map(value-wins(x), win-sets)

    return sum(win-values)
```

This algorithm is used to value various game boards, by seeing what positions need to be filled in to create 4 in a row. The reasoning for doing it this way is that the algorithm can value certain desirable board states very highly, for example any state where there are 2 different places that 1 piece will lead to a win is valued extremely well.

Player Program Implementation

First is the main portion of the program, which is what runs when the program is called. This handles receiving the input, processing the json into a valid game state, and calling the negamax function to determine the next move. It then sends out the next move as a string and loops back around.

```
(defn -main
  [x]
  (stderr "Connect Four")
  (loop [input-line (read-line)
         move# 1]
    (if-not (or (nil? input-line) (= input-line ":exit"))
      (do
        (stderr input-line)
        (let [precept (json/read-str input-line :key-fn keyword)
              moves (valid-moves (:grid precept))
              player (:player precept)
              opponent (case player
                         1 2
                         2 1)
              move (case move#
                     1 (int (/ (count (:grid precept)) 2))
                     (parallel-negamax (:grid precept) plies player
                      ↪  opponent))
              move-str (json/write-str {:move move})]
          (stderr move-str)
          (println move-str))
        (recur (read-line) (inc move#)))
      (shutdown-agents))))
```

Next is the code to setup the grid, along with some functions for determining valid moves and performing moves. A sample grid structure is included, this is what the main function generates each time it loops through.

```
(def precept
  {:height 6
   :player 1
   :grid [[0 0 0 0 0 0][0 0 0 0 0 2][0 0 0 0 0 0][0 0 0 1 2 1][0 0 0
    ↪  0 0 0][0 0 0 0 0 0][0 0 0 0 0 0]]
   :width 7})

(defn indices
  "Sequences of indicies for which a predicate is satisfied"
  [pred coll]
  (keep-indexed #(when (pred %2) %1) coll))

(defn indices-of-zeroes
  [coll]
  (indices zero? coll))
```

```clojure
(defn valid-moves
  [grid]
  (indices-of-zeroes (map first grid)))

(defn perform-move
  [move grid player]
  (let [grid-col (nth grid move)
        new-col (assoc grid-col (last (indices-of-zeroes grid-col))
         ↪  player)]
    (assoc grid move new-col)))
```

Then some processing of the grid is done, indexing the grid into a more usable form, retrieving the positions that a player currently occupies, and determining if any given position is available to that player. These functions are used in the evaluation functions, and make dealing with the grid a bit simpler.

```clojure
(defn index-columns
  "Takes grid and returns list-grid with values replaced by list:
   ↪  (y-pos value)"
  [grid]
  (map #(reverse (map-indexed list (reverse %))) grid))

(defn index-rows
  "Takes grid and returns list-grid with each column replaced by
   ↪  list: (x-pos (column))"
  [grid]
  (map-indexed list grid))

(defn index-grid
  "Given a grid, returns list-grid with values replaced by list:
   ↪  (x-pos y-pos value)"
  [grid]
  (let [indexed-columns (index-columns grid)
        all-indexed (index-rows indexed-columns)]
    (map (fn [[x-idx col]]
           (map #(conj % x-idx) col))
         all-indexed)))

(defn get-positions
  "Returns set of current occupied positions by player, of form
   ↪  (x-pos y-pos)"
  [grid player]
```

```
  (let [flat-grid (apply concat (index-grid grid))]
    (into #{} (map butlast (filter #(= (last %) player)
     ↪  flat-grid)))))

(defn available?
  [grid height width positions [x y]]
  (cond
    (contains? positions (list x y)) true
    (or (< x 0) (> x (dec width)) (< y 0) (> y (dec height))) false
    (= 0 ((grid x) (- (dec height) y))) true
    :else false))
```

The most arduous code is the evaluation code, which is over 100 lines. It mostly involves generating all of the possible winning rows of 4, combining everything together, creating sets to remove duplicates, and removing supersets afterwards. The final output is a succinct list of positions needed for a win, and the lengths of those lists is used to determine an overall score for the board.

```
(defn static-eval
  [node player opponent]
  (if (= inf (eval-by-positions-needed node (get-positions node
   ↪  opponent)))
    -inf
    (eval-by-positions-needed node (get-positions node player))))

(defn eval-by-positions-needed
  [grid positions]
  (let [win-sets (remove-supersets (win-move-sets grid positions))
        win-values (map #(value-wins (count (first grid)) %)
         ↪  win-sets)
        counts-of-wins (frequencies (map count win-sets))]
    (apply + win-values)))

(defn value-wins
  [board-height win-set]
  (let [win-len (count win-set)
        heights (map inc (map second win-set))
        avg-height (if (zero? win-len) 1 (/ (apply + heights)
         ↪  win-len))
        height-value (/ board-height avg-height)]
    (case win-len
      0 inf
      1 (* height-value 50)
```

```
      2 (* height-value 10)
      3 (* height-value 1))))

(defn remove-supersets
  [win-sets]
  (to-subsets (reverse (sort-by count win-sets))))

(defn to-subsets
  "Takes a sequence of sets ordered from largest to smallest"
  [ coll]
  (loop [result () coll coll]
    (if (empty? coll) result
                      (let  [x  (first coll)
                             xs (rest coll)]
                        (if (some #(clojure.set/superset? x %) xs)
                          (recur result xs)
                          (recur (cons x result) xs))))))

(defn win-move-sets
  [grid positions]
  (reduce
    (fn [wins position]
      (let [f-rows (get-rows
                     position
                     (all-rows-accumulator grid positions)
                     :forwards)
            b-rows (get-rows
                     position
                     (all-rows-accumulator grid positions)
                     :backwards)
            fb-rows (merge-with conj
                                {:h '() :v '() :lr '() :rl '()}
                                f-rows
                                b-rows)
            position-wins (flatten
                            (remove nil?
                                  (map #(reduce-to-wins positions %)
                                       (vals fb-rows))))]
        (into wins position-wins)))
    #{} positions))

(defn reduce-to-wins
  [positions fb-row]
  (let [wins (filter #(= (count %) 4) fb-row)
        sorted-wins (sort-by count
```

```
                          (map #(clojure.set/difference % positions)
                                 wins))]
    (cond
      (empty? sorted-wins) nil
      (= 1 (count sorted-wins)) sorted-wins
      :else (if (clojure.set/subset? (first sorted-wins) (second
      ↪  sorted-wins))
              (list (first sorted-wins))
              sorted-wins))))

(defn get-rows
  [position accumulator order]
  (let [inc (if (= order :backwards) dec inc)
        dec (if (= order :backwards) #(+ 1 %) dec)
        horizontal+ (reduce accumulator #{}
                            (iterate (fn [[x y]] (list (inc x) y))
                             ↪  position))
        horizontal (reduce accumulator horizontal+
                           (iterate (fn [[x y]] (list (dec x) y))
                            ↪  position))
        vertical+ (reduce accumulator #{}
                          (iterate (fn [[x y]] (list x (inc y)))
                           ↪  position))
        vertical (reduce accumulator vertical+
                         (iterate (fn [[x y]] (list x (dec y)))
                          ↪  position))
        lr-diag+ (reduce accumulator #{}
                         (iterate (fn [[x y]] (list (inc x) (inc y)))
                          ↪  position))
        lr-diag (reduce accumulator lr-diag+
                        (iterate (fn [[x y]] (list (dec x) (dec y)))
                         ↪  position))
        rl-diag+ (reduce accumulator #{}
                         (iterate (fn [[x y]] (list (dec x) (inc y)))
                          ↪  position))
        rl-diag (reduce accumulator rl-diag+
                        (iterate (fn [[x y]] (list (inc x) (dec y)))
                         ↪  position))]
    {:h horizontal :v vertical :lr  lr-diag :rl rl-diag}))

(defn all-rows-accumulator
  [grid positions]
  (fn [acc pos]
    (if (= (count acc) 4)
      (reduced acc)
```

```
      (if (available? grid 6 7 positions pos)
        (conj acc pos)
        (reduced acc)))))
```

Finally, the negamax code is relatively simple. It's essentially the same as the psuedo code listed above, used straight from Wikipedia, just modified for Clojure. In addition, a function is used for the base case which also allows negamax to be run in parallel, increasing the number of plies that can be calculated (in testing, this let me use one extra ply).

```
(defn negamax
  [node depth alpha beta color player opponent]
  (if (or (zero? depth) (p ::game-over-check (game-over? node)))
    {:value (* color (static-eval node player opponent)) :move 0}
    (reduce (fn [maxResult move]
              (let [result (negamax-ab-recur
                            (perform-move move node (if (= color 1)
                            ↪  player opponent))
                            (dec depth)
                            (- beta)
                            (- alpha)
                            (- color)
                            player
                            opponent)
                    newMax (max-key #(:value %)
                            maxResult
                            {:value (- (:value result))
                             :move move})
                    newAlpha (max alpha (- (:value result)))]
                (if (>= newAlpha beta)
                  (reduced newMax)
                  newMax)))
            {:value -inf :move nil}
            (shuffle (valid-moves node)))))

(defn parallel-negamax
  [node depth player opponent]
  (let [moves (p ::get-valid-moves (valid-moves node))
        successors (map #(perform-move % node player)
                        moves)
        best-result (apply min-key #(:value %)
                        (map-indexed #(assoc %2 :idx %1)
                          (map #(negamax % (dec depth) -inf inf -1
                          ↪  player opponent)
```

```
                                successors)))]
    (nth moves (:idx best-result))))
```

Expected Results

The expectation is that my program will easily beat the naive implementation, with near 100 percent win rate. Beyond that, I believe that this program should be able to beat many other student's programs due to the relatively involved static evaluation that I'm performing. Playing against itself or other student's will always have the same results, since there is no randomness involved, so I expect that the outcome will always be the same when not playing against the naive player.

Results

My program beat the naive implementation in all tests. A run of 50 was done and my program won 25 times going second and 25 times going first. This matches expectations.

Playing against itself, the program wins when going second. There is no randomness so it's just a single test run, but this doesn't really match expectations since first player has an advantage. The board does get almost entirely filled though, so there are no obviously huge errors being made.

I played my program against three other student's programs, and mine always won when going first (again, no random). Going second mine won against all but one program, and once again almost the entire board was filled up. Overall the results are pretty good, although play is definitely not optimal.

Conclusions

My biggest takeaway from this project is that heuristics are extremely difficult. Initially I decided that my effort was best spent in trying to make the most thorough and accurate evaluation of the board state as possible, but I now believe that similar (if not better) results could have been achieved by simply performing more plies of minimax. Tweaking the evaluation function often produced wildly different results, and it was an exercise in frustration trying to figure out the best values and method for scoring various board states. It's more difficult trying to "solve" the game somewhat intelligently than it would

be to use more computing power and brute force to reach deeper states that can give optimal answers.

This leads into my second takeaway, which is that minimax is quite simple and very effective. It took much less time to implement minimax than my evaluation function, and it worked quite well whenever it was able to reach a win state through the plies I was searching. It seems to be a very effective algorithm, and I feel that I have a better understanding of how game playing search is done now.