COSI 101a
Machine Problem 2
NANNON: Nano-Backgammon

Due Thurs 4/08 11:55 PM. 5% late fee per day.

Nannon is a parametric extract of the complex game of backgammon, for which the first machine-learning champion emerged from IBM Yorktown in 1995. TD-gammon used a neural network with reinforcement learning which played against itself and updated the weights. Additionally, Tesauro used multi-ply expectimax search to beat the best human backgammon players.

Nannon is parametrized by 3 variables Boardsize, Checkers, and Die which you can use as global variables. The standard game uses a board of 6 positions, 3 checkers per player, and a normal 6-sided die. Other variations include {8,4,6} and {12,5,6} Based on the rules, the number of checkers-per-player must be less than the maximum of the die to avoid stalemate.

The rules are as follows: Players start checkers at opposite ends of the board, take turns rolling one die, and, if possible, MUST move one checker forward the number of steps given, or off the board. To determine who goes first, both players roll their dice, and the winner gets the difference between the rolls (in case of tie, roll again). A checker cannot legally land on another checker of the same color, nor on any of the opponent which are protected by an adjacent checker on the board. Landing on a singleton opponent checker hits it back off the board. Full rules are printed on the back of your board, and you can play online against an expert at http://nannon.net. There is a Nannon in the Android Play store with which you can play against a friend. There is also a windows executable at http://nannon.com/freenannon.exe.

The Nannon$^{\{6,3,6\}}$ game has about the same complexity as TicTacToe, but because of the dice rolling, it is more complex to play, and so far nobody has come up with a heuristic which works as well as an experienced human player.

Your goal in this homework is to build a framework for experimenting with Nannon and to attempt to build a set of rules for moving, testing against some standard players.

First you need to represent a position in a data structure, it could be as simple as a list of numbers between 0 and *boardsize*+1. Depending on how you write your functions, you might want a more refined datastructure, and might want to store a dice roll or a nil (if the dice hasn't been rolled yet) as part of your representation.

Basic functions and game logic (35%)

(ROLL) which returns a uniform random number between 1 and *DIE*

(FIRSTROLL) which rolls 2 dice and returns the difference, unless 0, in which case it tries again.

(DICESTREAM n) creates a list of n dicerolls so a game can be repeated for deterministic players.

(SWAPPLAYER pos) flips a board representation so most functions only have to be written from a perspective of player 1.

(WHOWON pos) should check if either player has all pieces off the board and return +1 or -1 or 1 or 2 (or 0 or nil depending on your application requirements)

(PRINT_BOARD POS) should print a position in a human-readable form. A basic version is included which works on a 6-tuple of player 1 and player2 positions. You can convert your representation into a sixtuple to use this PRINT_BOARD or write your own which would scale up to games of 8 or 12 boardsizes. A python version is also included which uses a list of two lists of 0-indexed positions. .Similarly, you can convert your representation to use this version.

(LEGALMOVES POS ROLL) returns the position that player1 can move from with the current roll. This may be a list of 0(nil), 1, 2 or 3 values indicating the board positions of player1's checkers. You might make a module first which determines where player1 can land on the board (i.e. not on player 1 checkers or player 2 checkers next to each other)

(MAKEMOVE POS CHECKER ROLL) this executes a move for player1 returning a new board position, including checking if an opponent was hit and moving that checker back off the board (how, depends on your board representation).

PART 2: GENERALIZE THE GAME (25%)

DEFUN (EXPLORE) using your board representation, and a hash table (Python Dict), use depth or breadth-first search to find all the legal positions of the game. HINT: When you "expand" a position, you need to check all 6 rolls of the die, and both the position and the dual (swapplayer). There are about 2530 in the {6,3,6} game. How many are there in the {8,4,6} and the {10,5,6} game?

Demonstrate that your game logic works on two different size Nannon games.

PART 3: Tournaments! (25%)

(PLAYGAME #'PLAY1 #'PLAY2 &OPTIONAL DICESTREAM) plays one game between two funarg players. By passing in a DICESTREAM, you can get repeated play, especially good for minimizing effects of randomness over skill. PLAYGAME can return which player won the game, or symbol, or the score (1, -1) for player 1. Be especially careful how you handled forced moves and forfeited turns.

Run (PLAYGAME 'HUMAN 'RANDPLAY) to play against a random player until you are sure you have it working. Does it work for {12,5,6}?

```
(defun human (pos roll &aux move)
  (printboard pos)
  (princ "you rolled: ")(princ roll)(princ " move? ")(terpri)
  (setf move (read))
```

```
  (if (member move (legalmoves pos roll))
      (makemove pos move roll)
      (progn (princ "try again")(terpri)(human pos roll))))


(defun randplay (board roll)
   (makemove board (pick (legalmoves board roll)) roll)
```

(PLAYTOURN #'PLAY1 #'PLAY2 &OPTIONAL (NPAIRS 1000)) calls PLAYGAME 2 x NPAIRS times and calculate the advantage or disadvantage of PLAY1. For example if you ask for 50 pairs, it will call PLAYGAME 100 times and then sum the payoffs and divide by 100 to show the advantage of strategy PLAY1 over PLAY2

FIRSTPLAY picks furthest forward piece for moving (to maximize chances of getting all checkers off)

LASTPLAY picks furthest back piece

SCOREPLAY takes a "count" of player1 and player2 positions (eg between 0 and 21) and picks the move with maximum Myscore-yourscore, favoring hitting and running from rear.

Play tournaments of these strategies against one another and report the results

(15%) Part 4: A Knowledge-based PLAYER (Creative)

Having played a number of games against your code, a classmate, or against the CGI-BIN player at nannon.net, you have a sense of strategy. While you have no choice if you have no legal moves or 1 legal move, the choices you make when you have 2 or 3 legal moves will certainly affect the outcome. Do you always hit if possible? Do you keep a prime if you have a choice? Hit or run or cover? Codify your strategy using a set of detection primitives and some kind of decision tree, case structure, or if-then-else trees. Your expert player should slide right into work in your PLAYGAME and TOURNAMENT function as a funarg which takes a position and roll, and returns the new position. Here are some ideas:

- Checkers off the board are good
- Opponent checkers at home are good
- Singleton pieces of mine are bad
- Protected pieces are good

Adjust parameters and combine detection primitives with logic and/or numbers to make the best knowledge-based player you can, and demonstrate it in round-robin tournaments with the stock players.

Play it against your other stock players and report the results.

Hand in, A) a 1-2 page discussion of your work including choice of representations, places you got stuck, pairwise tournament results, and the final results of a round robin tournament. B) the documented source code and C) an edited trace showing that you have met the objectives of the

assignment. The goal is to allow the TA to grade your assignment without running everything themselves.

Appendix: Here is a LISP "Print_board" which only works on {6,3,6} using a six-tuple in which (0 1 2 5 6 7) represents the initial position of the game. You can hack on this to make it work for your representation and expand it (using hex?) up to size 12 boards.

```
(defun spaces (n) (loop for i from 1 to n do (princ " ")))

(defun print_board (board &optional (position 0))
  (cond ((<= position 7)
       (if (= position 1) (princ "<")
         (if (= position 7)(princ ">")))
       (if (> (count position board) 0)
           (loop for piece from 0 to 5 do
              (if (=  (nth piece board) position)
                  (if (> piece 2) (princ "B")
                 (princ "A"))))
         (princ "-"))
       (print_board board (1+ position)))
      (T (terpri)
         (spaces (max 2 (+ 1 (count 0 board))))
         (princ "123456")(terpri))))
```

And here is a printboard in python, which uses two tuples such as [[0,1,2],[0,1,2]] as its input representation:

```
def princ(chr):
    print(chr,sep='',end='')

def printboard(pos):
    bflip=[BOARD+1-x for x in pos[1]]
    for i in range(CHEX-len(pos[0])):
        princ(' ')
    for x in pos[0][::-1]:
        if x==0:
            princ('o')
        else:
            princ(' ')
    princ('|| ')
    for i in range(BOARD):
        if pos[0].count(i+1):
            princ('o ')
        elif bflip.count(i+1):
            princ('* ')
```

```
        else:
            princ('  ')
    princ('||')
    for x in pos[1]:
        if x==0:
            princ ('*')
        else:
            princ (' ')
    print()
    for x in range(CHEX):
        princ(' ')
    princ('||')
    for i in range(BOARD):
        if (i +1) < 10:
            princ(' ')
        princ(i+1)
    if BOARD < 10:
        princ(' ')
    princ('||')
    for x in range(CHEX):
        princ(' ')
    print()
```

Here is a board and rules to cut out. If you don't have checkers or poker chips to use, use 3 quarters and 3 nickels. If you don't have a dicecube, flip a quarter, nickel, and penny and interpret heads as 1 and tails as 0 for a binary number, discarding 0 and 7, e.g. a quarter on heads and a penny on heads would be a roll of 5.