

# MINIMAXING FOR TIC-TAC-TOE

- **Defining "global" symbols**

setflag(Name, X) :-

nonvar(Name), % only constants allowed

retract(flag(Name, Val)), % kill if defined,

!, asserta(flag(Name, X)). % freeze & redefine

setflag(Name, X) :-

nonvar(Name), % only constants allowed

asserta(flag(Name, X)). % initial value

To store the initial board, solve the goal:

"setflag(theboard, [[b,b,b],[b,b,b],[b,b,b]]).

{ adds to the current rules the assertion:

"flag(theboard, [[b,b,b],[b,b,b],[b,b,b]]).

To reset the board, solve the goal:

"setflag(theboard, [[o,b,b],[b,x,b],[b,b,b]]).

{ removes from the current rules the assertion:

"flag(theboard, [[b,b,b],[b,b,b],[b,b,b]]).

and adds the assertion:

"flag(theboard, [[o,b,b],[b,x,b],[b,b,b]]).

To bind variable B to the globally stored board,

solve the goal: "flag(theboard, B)"

- **Starting the game**

```
startplay :- setflag(theboard, [[b,b,b], [b,b,b], [b,b,b]]),  
    % startplay is the initialization rule  
    % Initialize the board to all blanks  
write('The board is initialized'), nl,  
write('How many moves ahead should I consider?'),  
tab(2), read(Searchdepth), (Searchdepth > 0), nl,  
setflag(sdepth, Searchdepth), % Global searchdepth  
write('I am x, you are o; who goes first?'), tab(2),  
read(XorO), nl, % XorO indicates who goes first  
flag(theboard, B), % Retrieve the blank board  
begin(XorO, B, Searchdepth).
```

```
/* This kicks off the game; fairly self-explanatory  
except for "setflag" and "flag"; see below. */
```

```
begin(x, B, Searchdepth) :- % matches if x goes first  
    computermoves(B, Searchdepth).
```

```
begin(o, B, Searchdepth) :- % matches if o goes first  
    write('enter your move as "enter(row#,col#)." '), nl.
```

- **User interface rule**

```
enter(Row, Col) :- % User submits moves through this rule.
    flag(sdepth, SD), flag(theboard, B), % recover board
    install(o, [Row, Col], B, NewB), % place "o"
    write('after your move:'), nl, pprint(NewB),
    endgamecheck(NewB), % did he win?
    computermoves(NewB, SD).
```

```
computermoves(Board, Searchdepth) :- % x's turn now
    choosemove(x, Board, Searchdepth, Movechoice, Maxvalue),
    % find best move; Maxvalue is the largest of choices
    install(x, Movechoice, Board, NewB), % compute board
    setflag(theboard, NewB), % store new board
    write('Here is my move'), nl,
    pprint(NewB), % show this board
    endgamecheck(NewB), % did I win?
    write('enter your move as "enter(row#,col#)." ').
```

- **Choosing the best move**

choosemove(XorO, B, SD, [], 10000) :- terminalwin(B).

choosemove(XorO, B, SD, [], -10000) :- terminalloss(B).

choosemove(XorO, B, SD, Posn, Best) :-

```
/* XorO is a flag indicating which player is making the
choice. B and SD act as input to this rule; we know
the board and search depth. Posn and Max are
determined in solving the choosemove goal; Posn is a
2-element list describing the square in which the
"x" goes, and Max is the value of that move
resulting from minimaxing. In the initial
invocation of choosemove, it's Posn, the actual move
that we need. In the recursive invocations, it's
Max, the maximum value of the positions resulting
from possible moves that we need. This rule always
supplies both. */
```

```
NewSD is SD-1,      % 1 step deeper
```

```
genposns(B, Posnlist), % all possible moves from here
```

```
evalmovelist(XorO, B, NewSD, Posnlist, Valuelist),
```

```
    % a value for each
```

```
bestmove(XorO, Posnlist, Valuelist, Posn, Best).
```

```
    % pick the best
```

- **Where can we move?**

```
genposns(B, Posnlist) :- setof(P, openposn(B, P), Posnlist), !.  
genposns(B, []). % if "setof" fails, Posnlist should be empty  
  
openposn(B, [R,C]) :- row(R), column(C), install(x,[R,C],B,B1).  
  
row(1).  
row(2).  
row(3).  
column(1).  
column(2).  
column(3).
```

- **Putting an x or o on the board**

```
install(XorO, [1,Col], [BdRow | Rest], [New | Rest]) :-  
    !, installinrow(XorO, Col, BdRow, New). % move into row  
  
install(XorO, [Row,Col], [BdRow | Rest], [BdRow | New]) :-  
    Next is Row-1, install(XorO, [Next,Col], Rest, New).  
  
installinrow(XorO, 1, [b | Rest], [XorO | Rest]) :- !.  
  
installinrow(XorO, Col, [FirstCol | Rest], [FirstCol | New]) :-  
    Next is Col-1, installinrow(XorO, Next, Rest, New).
```

- **Evaluate a list of moves**

```
evalmovelist(XorO, B, SD, [], []). % Empty base case
```

```
evalmovelist(XorO, B, SD, [Pos], [Val]) :- % 1 element list  
    evalmove(XorO, B, SD, Pos, Val).
```

```
evalmovelist(XorO, B, SD, [Pos | P], [Val | V]) :-  
    evalmovelist(XorO, B, SD, P, V), % do the rest of it,  
    evalmove(XorO, B, SD, Pos, Val). % now the 1st element
```

```
evalmove(XorO, B, 0, Pos, Val) :- % Base case; at search limit.  
    !, install(XorO, Pos, B, B1), % Freeze choice, place the x  
    staticvalue(B1, Val). % or o and compute static value
```

```
evalmove(XorO, B, SD, Pos, Val) :-  
    install(XorO, Pos, B, B1), % place the x or o, find best  
    toggle(XorO, OorX), % move from the resulting choices  
    choosemove(OorX, B1, SD, Ignore, Val).
```

```
toggle(x, o) :- !. % Given x or o in one argument, it binds  
toggle(o, x) :- !. % the other argument to the opposite token
```

- **Selecting the best from the list**

```
bestmove(XorO, [], [], [], 0) :- !.
```

```
    % none possible; give it a zero
```

```
bestmove(XorO, [Move], [Value], Move, Value) :- !.
```

```
    % only one, it's the Best
```

```
bestmove(x, [M1 | Othermoves], [V1 | Othervalues], M, V) :-
```

```
    % Best = Best of rest if
```

```
    bestmove(x, Othermoves, Othervalues, M, V),
```

```
    V >= V1, !.    % the best of the rest is > 1st; freeze.
```

```
bestmove(x, [M1 | Othermoves], [V1 | Othervalues], M1, V1).
```

```
    % else best is 1st
```

```
bestmove(o, [M1 | Othermoves], [V1 | Othervalues], M, V) :-
```

```
    % Best = Best of rest if
```

```
    bestmove(o, Othermoves, Othervalues, M, V),
```

```
    V =<= V1, !.    % the best of the rest is =<= 1st; freeze.
```

```
bestmove(o, [M1 | Othermoves], [V1 | Othervalues], M1, V1).
```

- **Evaluate the current board**

```
staticvalue(B, 10000) :- terminalwin(B), !.
```

```
staticvalue(B, -10000) :- terminalloss(B), !.
```

```
staticvalue(B, V) :-
```

```
    extractrcd(B, RCDlist), evallist(RCDlist, V).
```

```
extractrcd( [ [SQ11, SQ12, SQ13],
```

```
            [SQ21, SQ22, SQ23],
```

```
            [SQ31, SQ32, SQ33] ],
```

```
[ [SQ11,SQ12,SQ13], [SQ21,SQ22,SQ23], % 1st 2 rows,
```

```
  [SQ31,SQ32,SQ33], [SQ11,SQ21,SQ31], % 3rd row, 1st col,
```

```
  [SQ12,SQ22,SQ32], [SQ13,SQ23,SQ33], % last 2 cols,
```

```
  [SQ11,SQ22,SQ33], [SQ13,SQ22,SQ31] ] ). % 2 diagonals
```

```
evallist([L], V) :- !, eval(L, V). % only 1 list; evaluate it
```

```
evallist([L1 | Lr], V) :- evallist(Lr, V1), % > 1 list; do the rest
```

```
    eval(L1, V2), % then do this one
```

```
    V is V1+V2. % add, get result
```

```
eval([b,b,b], 0) :- !.
```

```
eval(L, V) :- numberof(x, L, Nx), numberof(o, L, No),
```

```
    val(Nx, No, V).
```

```
numberof(Char, [], 0).
```

```
numberof(Char, [Char | Y], N) :- numberof(Char, Y, Nrest),
```

```
    N is Nrest+1, !.
```

```
numberof(Char, [X | Y], N) :- numberof(Char, Y, N).
```

```
val(Nx, No, 0) :- Nx>0, No>0, !.
```

```
val(Nx, 0, V) :- !, V is (Nx ** 3).
```

```
val(0, No, V) :- !, V is -(No ** 3).
```

- **Recognizing the end**

```
endgamecheck(B) :- terminalwin(B), !,  
    write('Eureka!! I WIN'), nl, abort.  
endgamecheck(B) :- terminalloss(B), !,  
    write('Damn!! You must have been lucky'), nl, abort.  
endgamecheck(B) :- fullboard(B), !,  
    write('Nobody wins; tie game'), nl, abort.  
endgamecheck(B).    % If the 1st 3 fail, do nothing and succeed.  
  
terminalwin(B) :- extractrcd(B, RCDlist),  
    x3(RCDlist).  
  
x3([[x,x,x] | Rest]) :- !.  
x3([Head | Rest]) :- x3(Rest).  
  
terminalloss(B) :- extractrcd(B, RCDlist),  
    o3(RCDlist).  
  
o3([[o,o,o] | Rest]) :- !.  
o3([Head | Rest]) :- o3(Rest).  
  
fullboard([Row1,Row2,Row3]) :- not member(b, Row1),  
    not member(b, Row2),  
    not member(b, Row3).
```

- **Printing out the board**

```
pprint([[S11,S12,S13],[S21,S22,S23],[S31,S32,S33]]) :-  
    nl, tab(5), type1,  
    nl, tab(5), pprintrow(S11,S12,S13),  
    nl, tab(5), type1,  
    nl, tab(5), type2,  
    nl, tab(5), type1,  
    nl, tab(5), pprintrow(S21,S22,S23),  
    nl, tab(5), type1,  
    nl, tab(5), type2,  
    nl, tab(5), type1,  
    nl, tab(5), pprintrow(S31,S32,S33),  
    nl, tab(5), type1, nl, nl.
```

```
type1 :- tab(5), write('|'), tab(5), write('|'), tab(5).
```

```
type2 :- write('-----').
```

```
pprintrow(C1, C2, C3) :- write(' '),  
    pprintchar(C1),  
    write(' | '),  
    pprintchar(C2),  
    write(' | '),  
    pprintchar(C3),  
    write(' ').
```

```
pprintchar(b) :- write(' '), !.
```

```
pprintchar(Char) :- write(Char).
```