Extended Example: Tic Tac Toe

week	date	Monday	Tuesday	Thursday
1	Jan. 9	**Introduction**	Haskell Start-Up	Haskell Start-Up
2	Jan 16	Haskell Start-Up	Recursion	Lists and Tuples *(assn 1 due)*
3	Jan 23	More About Lists	More About Lists	Proofs
4	Jan 30	Proofs/review	**Quiz 1**	I/O *(assn 2 due)*
5	Feb 6	Algebraic Types	Algebraic Types	Generalization
6	Feb 13	Functions As Values	Type Classes	Type Checking & Inference
7	Feb 27	Haskell review	**Quiz 2**	Lazy Programming
8	Mar 6	take up quiz 2 *(assn 3 due)*	Prolog Intro	Prolog Intro
9	Mar 13	Prolog Intro	Prolog Intro	Lists *(assn 4 due Friday)*
10	Mar 20	review / arithmetic	**Quiz 3**	arithmetic
11	Mar 27	cuts & negation	depth-first search *(assn 5 due)*	I/O
12	Apr 3	**example**	**example**	review *(assn 6 due Friday)*

Goals

1. Practice Prolog & Haskell on a non-trivial problem.
2. Learn about minimax searching.
3. Learn one more Prolog feature: **`assert`**

`assert` & minimax won't be on the final.

Prolog's Database

Prolog maintains a database of facts and rules.
Normally we add to the database by "consulting" a file.

Normal way things happen:

- facts and rules stated in file (statically)
- query those rules (from SWI-Prolog or other rules)

Sometimes it's convenient to assert a fact dynamically –
"Discover" a fact while working,
Record it for later use.

How To Use `assert`

`assert(predicate(arg1,arg2))`:

adds `predicate(arg1,arg2)` to Prolog's database

Try in SWI-Prolog....

Static vs. Dynamic Predicates

A *static* predicate is defined by rules or facts stated in a file.
A *dynamic* predicate is defined using **`assert`**.
demo...

If you're going to assert facts inside rules, good idea to declare the predicate as dynamic:

```
:- dynamic(wearing/2).
```

Means:

- **`wearing`** is a predicate with 2 parameters
- we will be defining some or all facts about wearing dynamically (with **`assert`**)

Reading

`assert` described in Section 7.4 of Prolog text:

"Database Manipulation"

includes more than I've told you:

- you can assert rules as well as facts
- you can "retract" facts & rules

Reading is optional.

Caution

`assert` can be useful.

In tic tac toe program, we will use it to remember moves we've already found: major speed gain.

BUT:

`assert` can also be used to write programs that are impossible to understand/debug/modify!

Use with care!

Tic Tac Toe in Prolog

Want a tic tac toe program where the computer plays the user.
Simplify: computer is always X, user is O.
Either player can go first.
demo...

Convention in Describing Prolog Predicates

Borrowed from SWI-Prolog manual:

put a character before each parameter name in comment

+: input parameter (should be bound before call)

-: output parameter (should not be bound – predicate will set)

?: can be used either way

Example:

```
% flatten(+List1, -List2)
```

The following is OK:

```
flatten([[1,2],[3,4]],L)
```

The following doesn't work

```
flatten(L,[1,2,3,4]).
```

Preliminary Decision: I/O

Ideal: use a Prolog GUI library or link with foreign I/O code.
No time!

Displaying board with text is easy:

```
x | x | o
---------
o | o | x
---------
x | x | o
```

Show numbers on unoccupied spaces:

```
1 | 2 | 3
---------
4 | 5 | 6
---------
7 | 8 | 9
```

User can choose a move by typing a number.

Step 1: Decide on a Representation

State of tic tac toe game must specify:

- whose turn it is
- what squares have Xs
- what squares have Os

Ideas for representation?

My decision: use structure with three parts:

- whose turn it is (x or o)
- list of X squares
- list of O squares

Example of Representation

`state(x, [1,3],[2,9])`

Means it's x's turn, board looks like this:

```
x | o | x
---------
4 | 5 | 6
---------
7 | 8 | o
```

Two Helper Predicates

```
opponent(x,o).
opponent(o,x).

player(state(x,_,_),x).
player(state(o,_,_),o).
```

List Order

Later on, we'll want to store and recognise states we've already visited.

Useful for lists of Xs and Os to be in order.

Two more helper predicates to create & use ordered lists of numbers:

`addToList(+List,+Item,-NewList):` adds **`Item`** to **`List`**, assuming **`List`** is an ordered list of numbers. **`NewList`** is the result, also ordered.

`ordSubset(+Subset,+List)`: like subset, but assuming both lists are ordered.

Step 2: Predicate For Displaying Board

`displayState(+State):` displays the board as a 3x3 grid

Main idea:

    for N = 1 to 9:
        if N is in X list, write 'X'
        else if N is in O list, write 'O'
        else write N
    ... plus line breaks and grid lines

Details in posted code.

Step 3: Detect Win, Lose, Draw

For minimax algorithm, want to give each state a rating:

2 = X wins

1 = draw

0 = O wins

X wants to maximize, O wants to minimize.

This step, just consider "terminal" states – game is over.
Simple criteria for a draw: all squares taken, nobody has won.

`terminal(+State,-Value)` : means **`State`** is a terminal state and its value is **`Value`**. Fails if **`State`** is not terminal.

`terminal` predicate

Player has won if all spaces on some row, column or diagonal contain that player's mark.

Could attempt general solution – not worth it!

Only 8 winning combinations. Use facts to enumerate them.

```
winningComb([1,2,3]). % first row
winningComb([4,5,6]). % second row
winningComb([7,8,9]). % third row
winningComb([1,4,7]). % first column
winningComb([2,5,8]). % second column
winningComb([3,6,9]). % third column
winningComb([1,5,9]). % \ diagonal
winningComb([3,5,7]). % / diagonal
```

terminal rules (1)

```
% rule 1: X has won:
terminal(state(Player,Xlist,Olist),2) :-
  winningComb(Comb),
  ordSubset(Comb,Xlist),
  !.
```

terminal rules (2)

```
% rule 2: O has won:
terminal(state(Player,Xlist,Olist),0) :-
  winningComb(Comb),
  ordSubset(Comb,Olist),
  !.
```

terminal rules (3)

```
% rule 3: a draw
terminal(state(Player,Xlist,Olist),1) :-
  length(Xlist,Xlength),
  length(Olist,Olength),
  Xlength+Olength =:= 9.
```

Step 4: Specify Legal Moves

For depth first search:

`move(A,B)` means you can move from A to B

For minimax, more useful to have a predicate to list all possible moves.

`moves(+State,-NewStates)` : means **`NewStates`** is a list of all possible moves from **`State`**.

Legal Moves

Suppose current state is **`state(x,Xlist,Olist)`**.

Legal next states will have:

- player = o
- same Olist
- Xlist has one extra number – which was not taken before

try example....

Minimax Searching

Applies to 2-person "full information" games:

- no randomness (toss a die, shuffle a deck of cards)
- nothing hidden (hands of cards)

From a given state, searches "game tree" to find optimal move.

Rating States

Recall ratings of terminal states:

2 = X wins

1 = draw

0 = O wins

X will always act to maximize the final state of the game.

O will always act to minimize the final state of the game

Rating Non-Terminal States

X's turn:

look at all possible moves

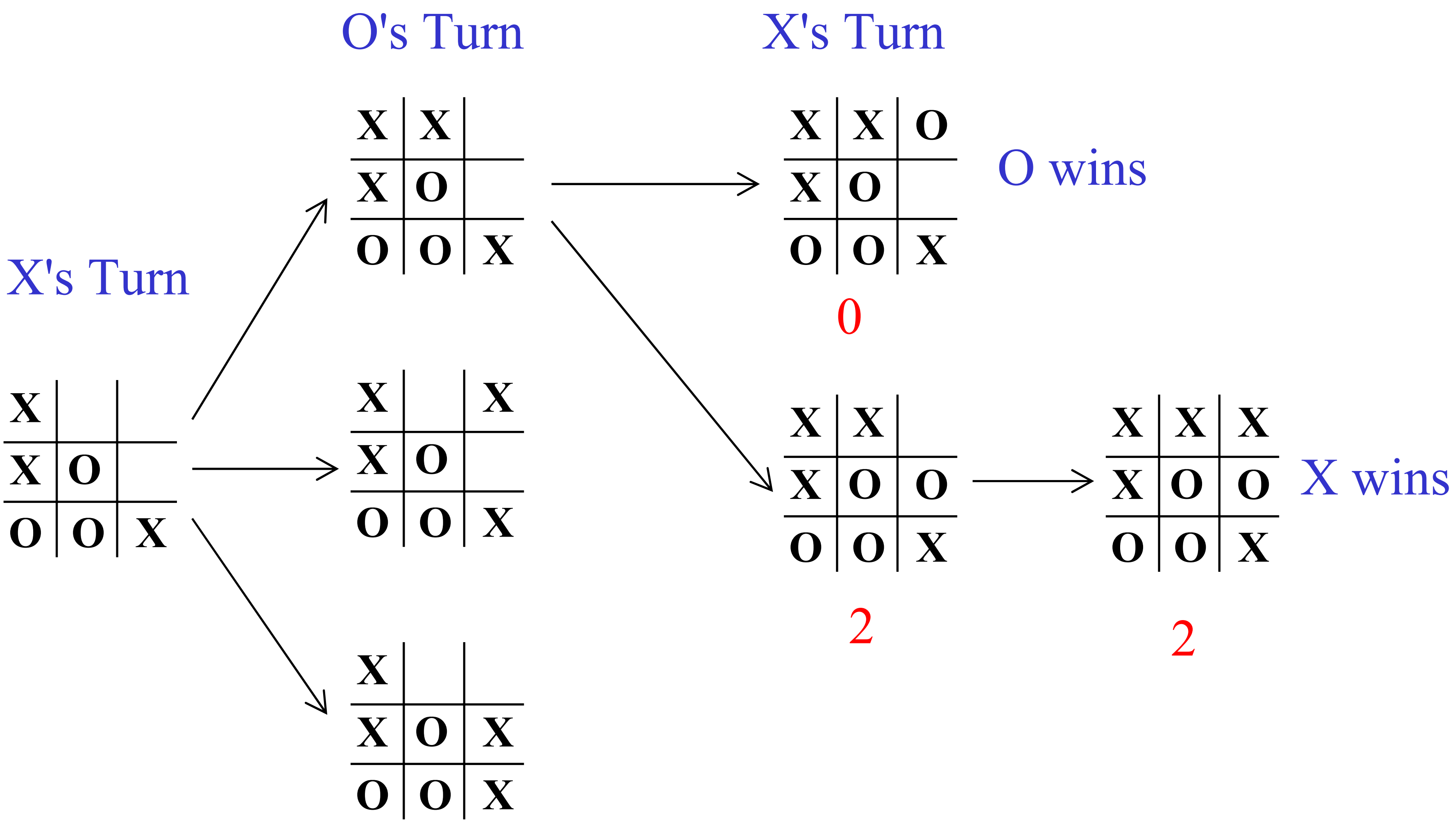
pick the one with the maximum rating

O's turn:

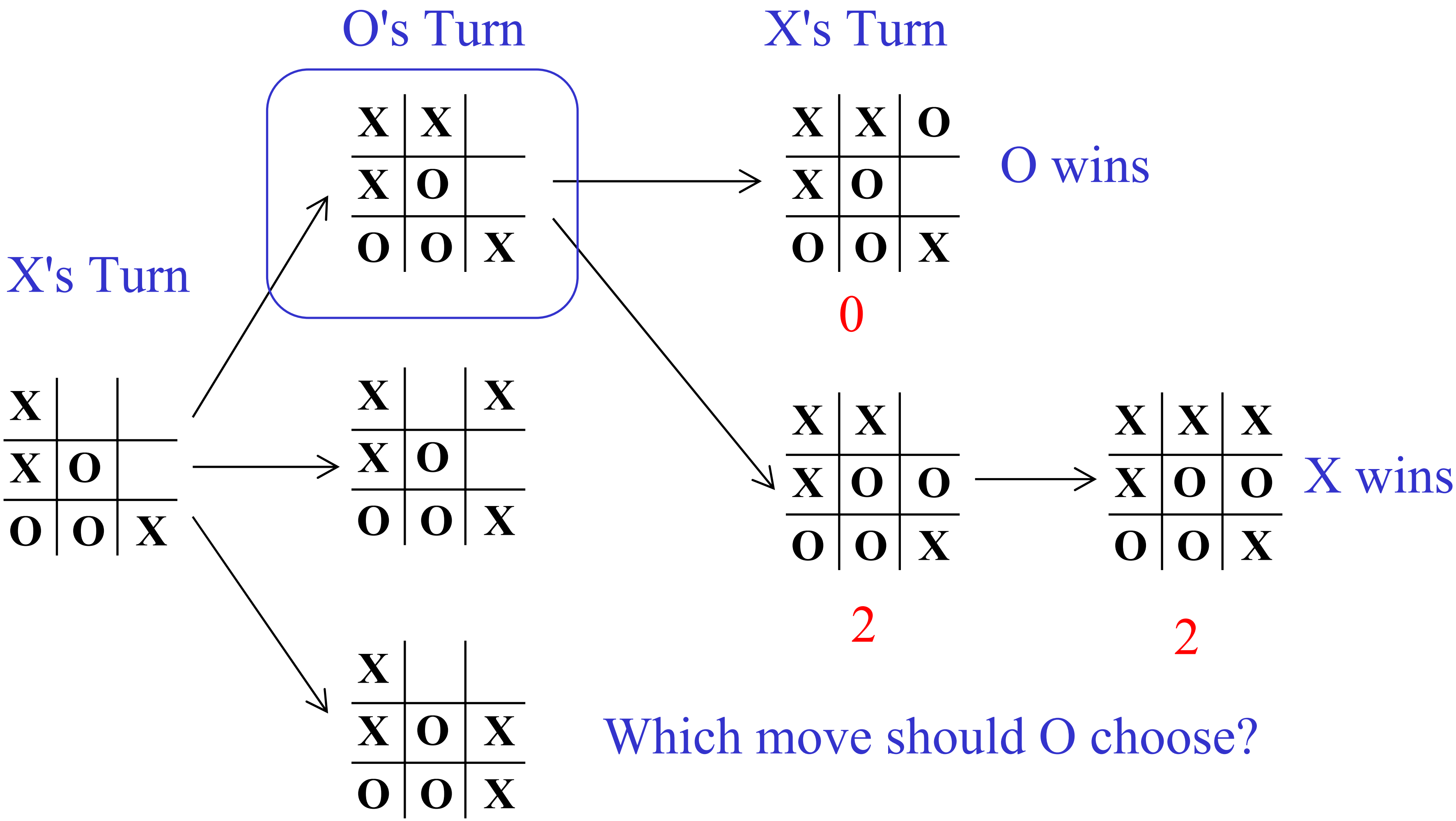
look at all possible moves

pick the one with the minimum rating

Sample Game Tree

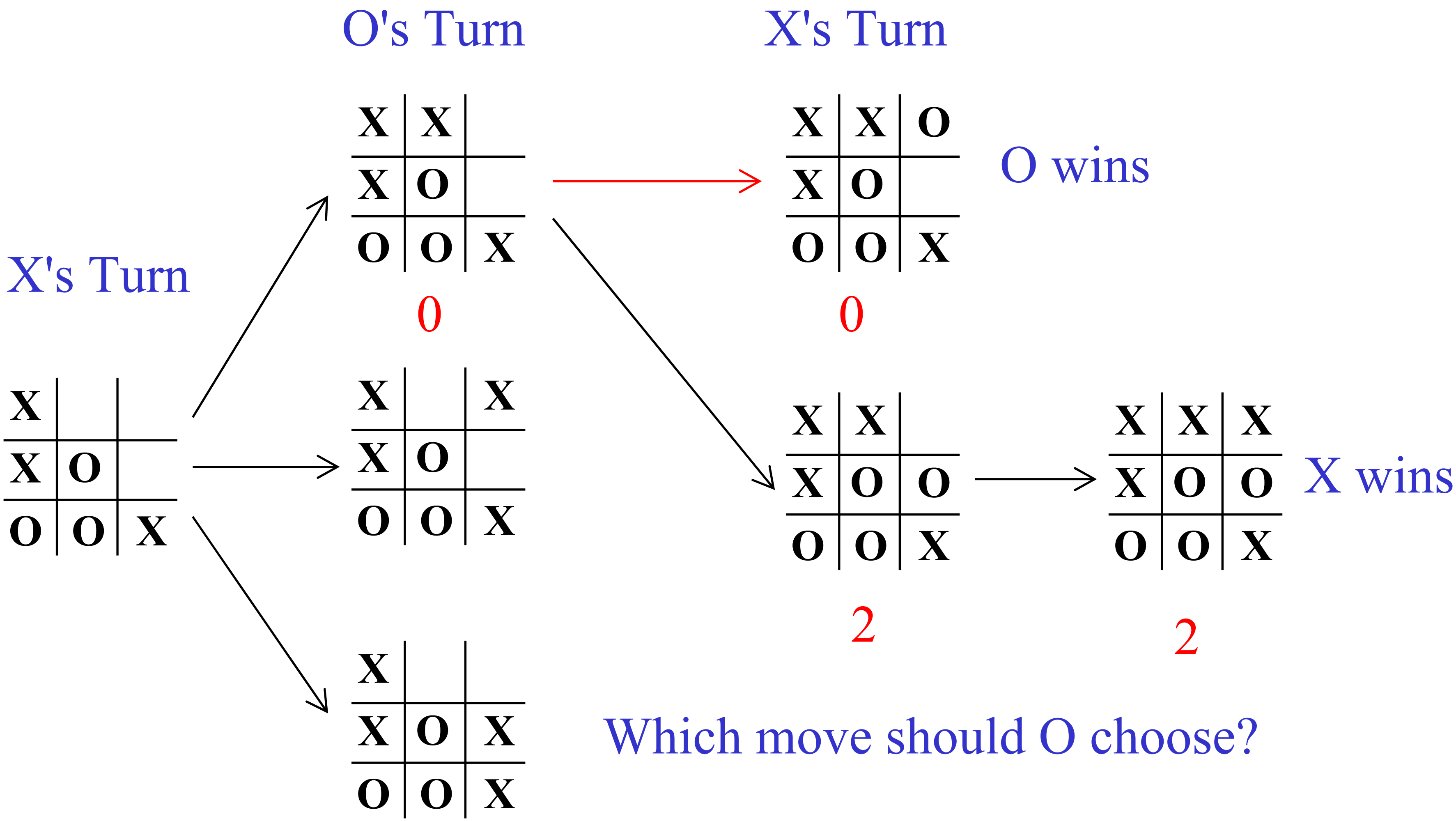


Sample Game Tree



Which move should O choose?

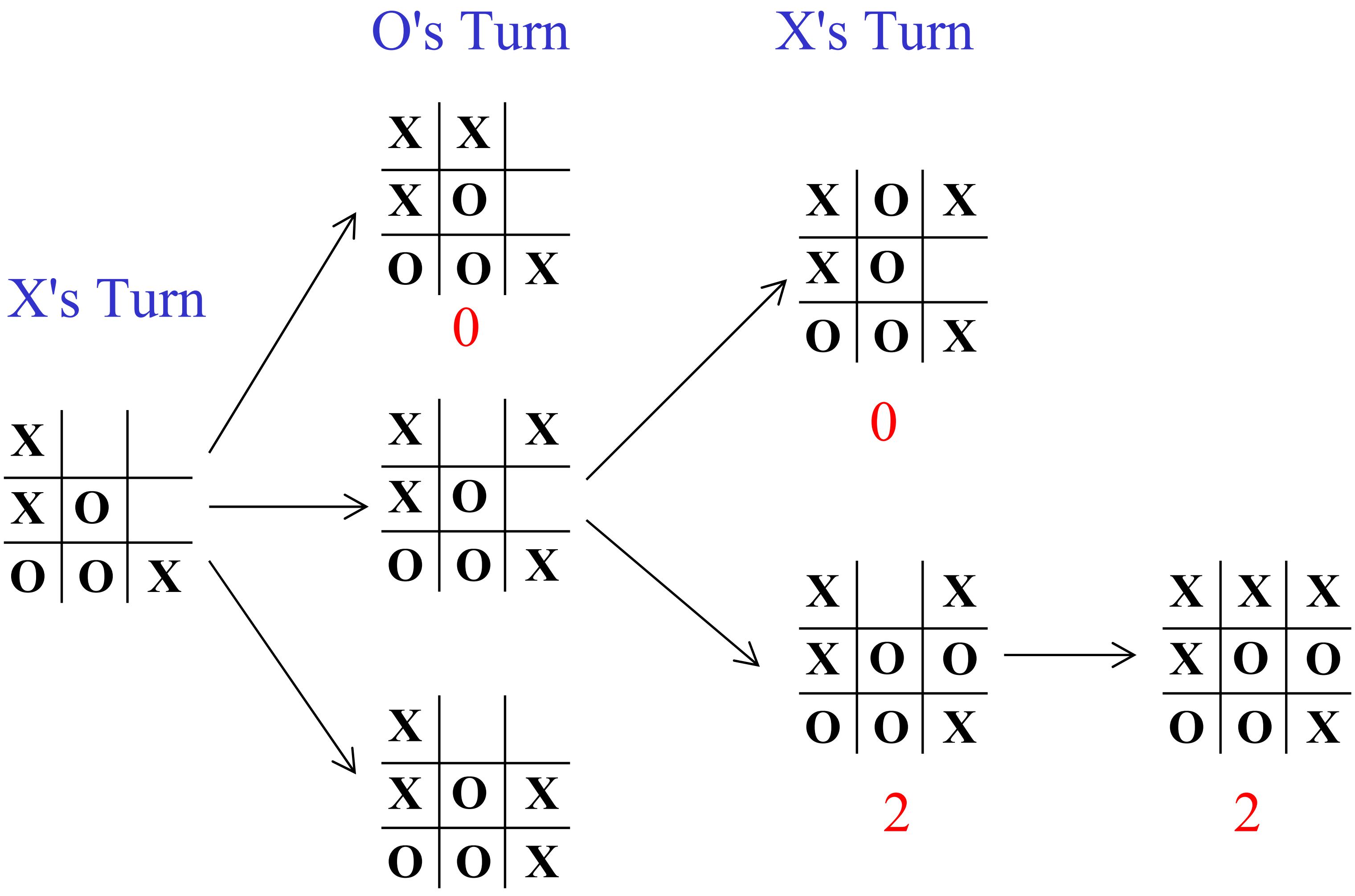
Sample Game Tree



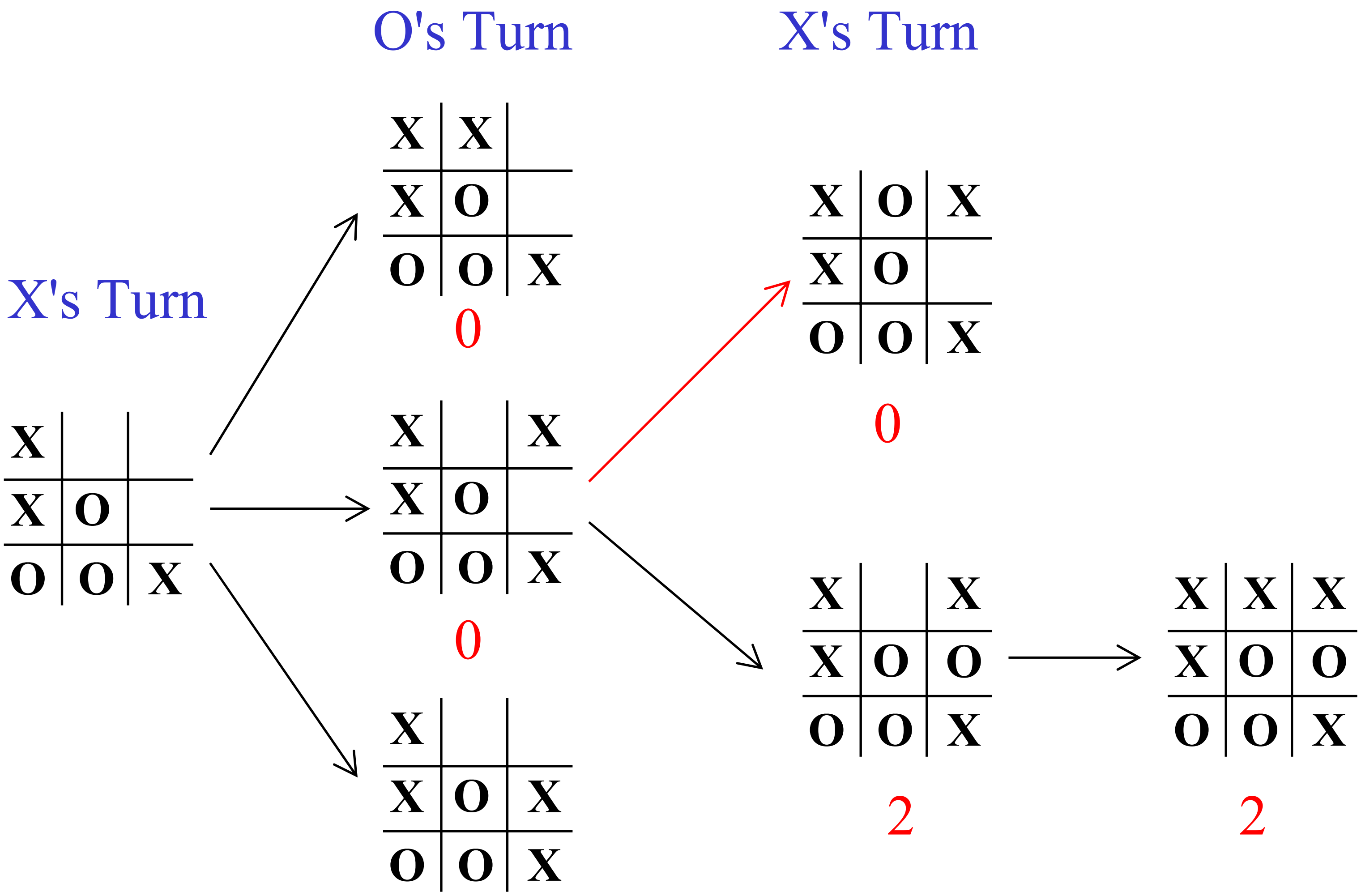
Which move should O choose?

O chooses the move that leads to the minimum score.

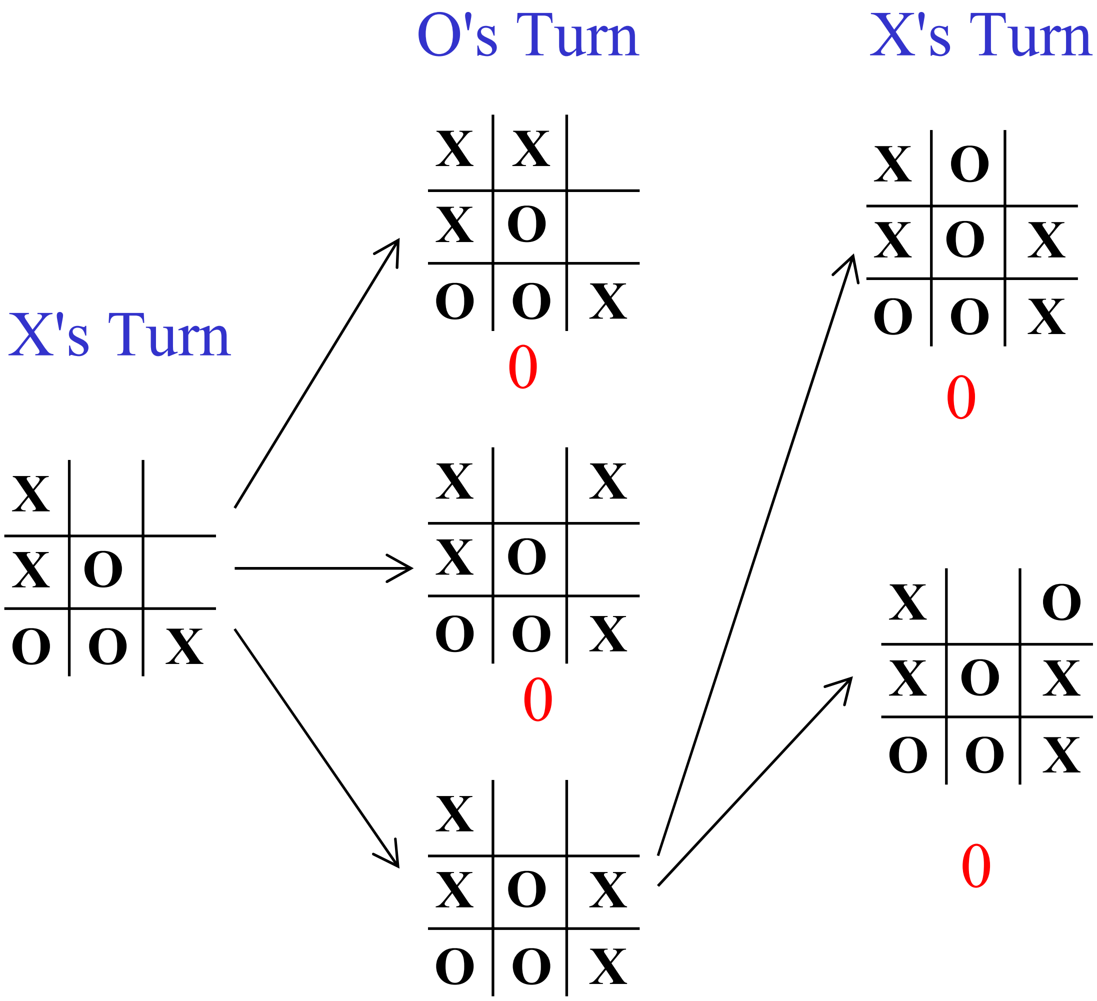
Sample Game Tree



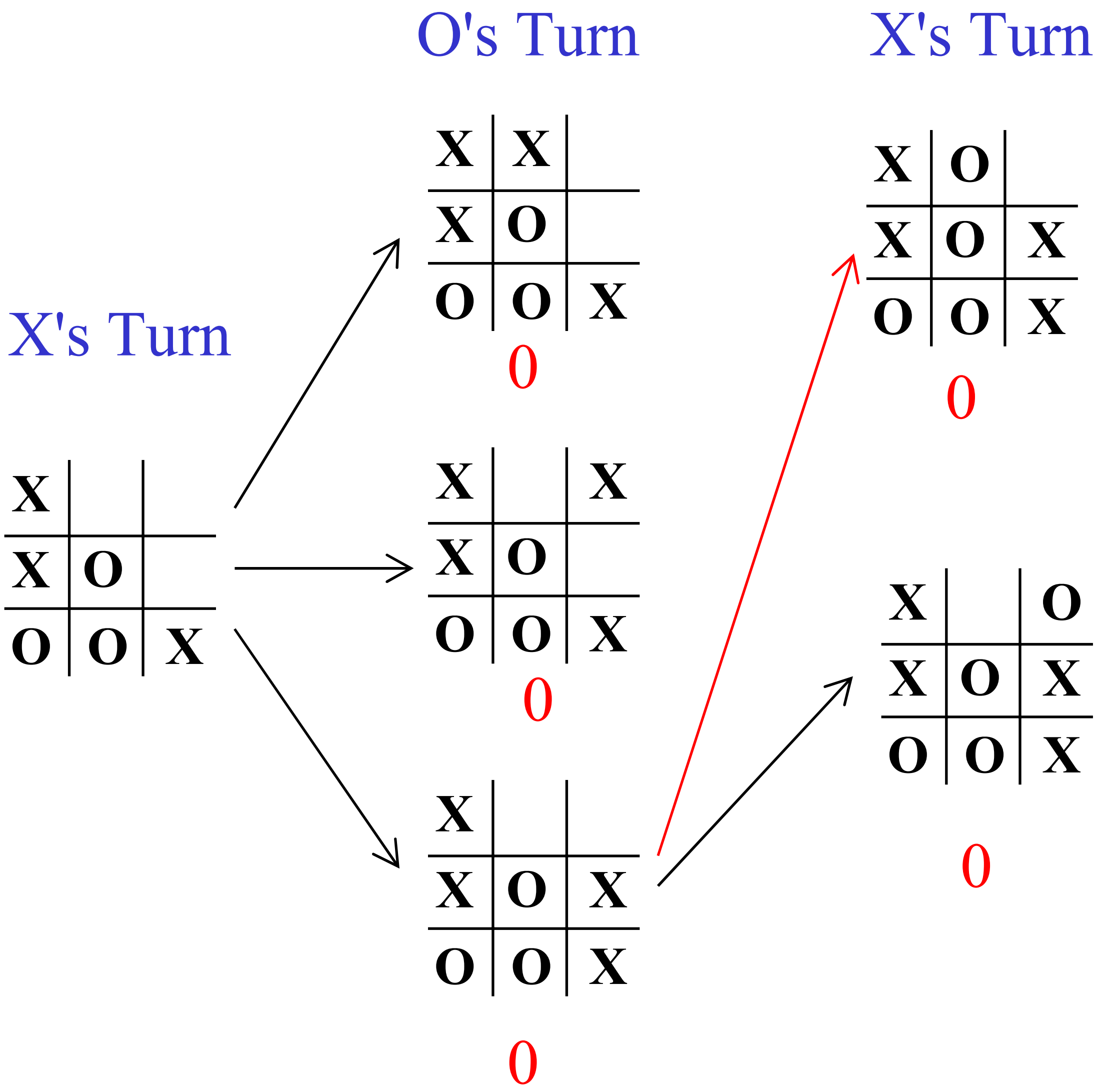
Sample Game Tree



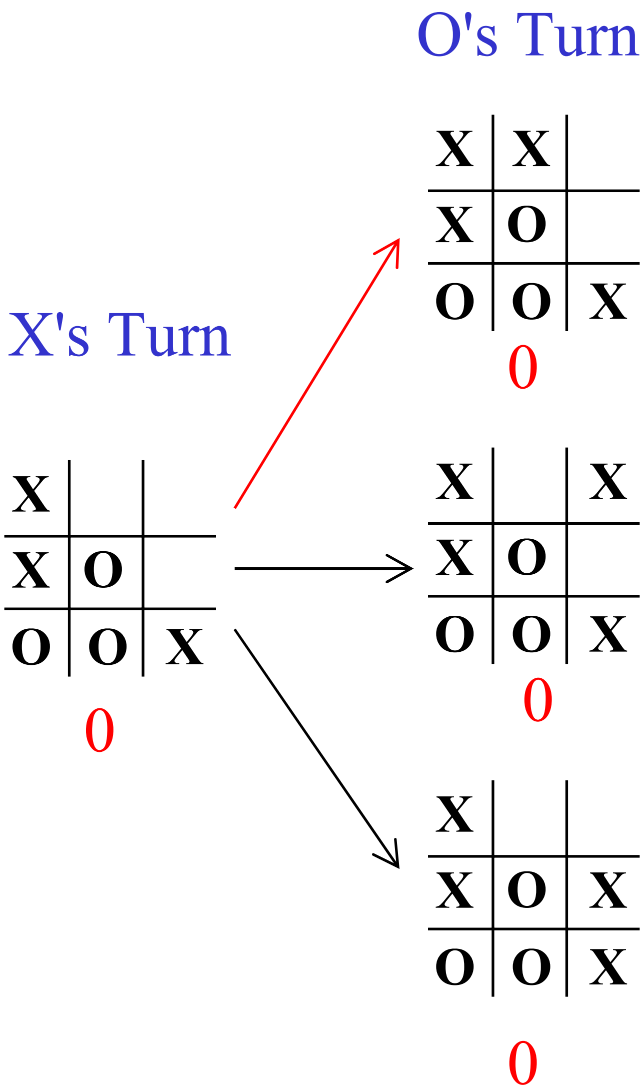
Sample Game Tree



Sample Game Tree



Sample Game Tree



X will pick the move that gets the maximum rating.

In this case, all moves lead to O winning, so makes an arbitrary choice.

Minimax Algorithm

Base Case:

- if X has won, rating = 2
- if O has won, rating = 0
- if all squares are taken and no one has won, rating = 1

X's turn:

- generate list of possible moves from current state
- find rating of each
- pick move with maximum rating

O's turn:

- generate list of possible moves from current state
- find rating of each
- pick move with minimum rating

minimax predicate (1)

`minimax(+State,-BestMove,-Value):`

from State, generates best move & value (rating) of that move

```
% base case: game is over
minimax(State, _, Value) :-
   terminal(State, Value),
   !.
```

minimax predicate (2)

```
minimax(State, BestMove, Value) :-
  moves(State, Moves),
  player(State, Player),
  bestMove(Player, Moves, BestMove, Value).
```

bestMove predicate (1)

`bestMove(+Player, +Moves, -BestMove, -Value):`
`Moves` = list of moves
examines list of moves and choose the best move for **`Player`**.
`BestMove` gets the best move from the list
`Value` gets the value of that move

```
% Base case: only one move in list
bestMove(_,[OneMove],OneMove,Value) :-
  !,
  minimax(OneMove, _, Value).
```

bestMove predicate (2)

```
bestMove(Player, [FirstMove|OtherMoves],
         BestMove, BestValue) :-
  minimax(FirstMove, _, ValueFromFirst),
  bestMove(Player,OtherMoves, MoveFromTail,
           ValueFromTail),
  choose(Player, FirstMove, ValueFromFirst,
         MoveFromTail, ValueFromTail,
         BestMove, BestValue).
```

choose predicate

```
choose(+Player, +Move1, +Val1, +Move2, +Val2,
       -BestMove, -BestVal)
```

Chooses between two moves.

```
choose(x, Move1, Val1, Move2, Val2, Move1, Val1) :-
  % x chooses the maximum value
  Val1 >= Val2,
  !.
choose(x, _, _, Move2, Val2, Move2, Val2).
```

corresponding pair of rules for o.

Last Step: I/O Predicates

Left for you to read if you're interested.
`ttt`: computer goes first
`meFirst`: user goes first

Problem with initial version: slow, ran out of stack space.

Optimization: Remember Previous Results

Search will encounter same intermediate states many times.
Remember these states & their best moves.
Use `assert`.

Problem: If we store too many facts, clogs up memory and database searches will be slow.

Solution: Store facts from one "level" of the tree only.

Decision: Store facts from level 5 of the tree: 5 squares already taken
Don't include terminal states.

new version of minimax rule

```
minimax(State, BestMove, Value) :-
   moves(State, Moves),
   player(State, Player),
   bestMove(Player, Moves, BestMove, Value),
   remember(State, BestMove, Value).

remember(State, BestMove, Value) :-
   numMarks(State, Count),
   Count = 5,
   !,
   assert(foundBest(State, BestMove, Value)).

remember(_,_,_). % why?
```

new base case for minimax

```
minimax(State, BestMove, Value) :-
    foundBest(State, BestMove, Value),
    !.
```

One More Optimzation

Still slow & needs large stack.
Another idea: symmetry.
General case = difficult.

Simple observation:
From an empty tic tac toe board, there are only three different first moves: corner, side and middle.

Improvement To `moves` Predicate

```
% new initial rules:
moves(state(x,[],[]),
               [state(o,[1],[]),
                state(o,[2],[]),
                state(o,[5],[])]) :- !.
moves(state(o,[],[]),
               [state(x,[],[1]),
                state(x,[],[2]),
                state(x,[],[5])]) :- !.
```

Plan For Today

1. More about minimax & game searching
2. Use Haskell version of tic tac toe to compare Haskell & Prolog

Reminder: Assignment 6 due in boxes&WebCT at 2 p.m. tomorrow.

Review Sessions ("group office hours"): tentative times

Thursday, Apr. 13, 1-3

Monday, Apr. 17, 1-3

regular office hours Apr. 11, 12, morning of 18th.

Will post on WebCT & web page when this is definite.

Minimax Alone

Useful only for fairly short games like tic tac toe

- maximum 8 moves at any time
- maximum tree depth = 9

Using minimax with no optimizations, how many nodes in game tree?

1 + 8 + 8*7 + 8 *7*6 + ... + 8! = approx. 109,000

Optimizations are possible:

- remembering game states
- stopping early: detecting win or draw situations
- exploiting symmetry
- more sophisticated algorithms to "prune" game tree

Chess

Theoretically, minimax can "solve" chess.

- What's the best opening move?
- From any chess board, how can I win?

Ball-park assumptions:

average of 15 moves possible from any board

average games takes 50 moves

How many nodes in game tree?

$1 + 15 + 15^2 + 15^3 + ... + 15^{50}$ == approx 7×10^{58}

Suppose a program takes one microsecond to examine each node

total time = 7×10^{52} seconds = approx. 1×10^{45} years.

Even with careful optimization, tree searching alone is not useful!

Deep Blue

1997: Computer program beat world chess champion

How Did Deep Blue Work?

Speeded up search with:

- parallel processors with specialized hardware
- many software optimizations

Still, searching to end of game isn't practical.

Heuristics & Evaluation Function

Heuristic = rule of thumb, works most of the time.
Evaluation function: gives a score to a chess board
 high score means black is likely to win
 low score means white is likely to win

Evaluation function doesn't use searching.
Based on wisdom of chess experts:

- what pieces each player has left
- who is controlling the center
- etc. etc.

Deep Blue team analyzed library of thousands of master games to help develop rules

Deep Blue Algorithm

- From any position, search to a depth of 12
- Most "leaves" of this tree won't be wins or losses
- Use evaluation function to assign a score to each leaf.
- Use minimax or variant to decide on move that results in best score for the current player.

Back To Tic-Tac-Toe and Haskell

Haskell version of tic-tac-toe:

- some parts are a fairly straightforward translation of the Prolog version
- some differences; useful to note as a way to compare the two languages / paradigms

`assert` optimization

Prolog version: optimizes by keeping database of positions and best moves

Easy because of **`assert`** facility.

Haskell: no side effects. Not possible to use/modify a central database.

Would be very difficult to implement this sort of "memory" in a Haskell program.

Multiple Results in Prolog

Prolog very useful for situation in which there are more than one possible "answer".

```
terminal(state(_,Xlist,_),2) :-
   winningComb(Comb),
   ordSubset(Comb,Xlist),
   !.
winningComb([1,2,3]). % first row
winningComb([4,5,6]). % second row
winningComb([7,8,9]). % third row
winningComb([1,4,7]). % first column
winningComb([2,5,8]). % second column
winningComb([3,6,9]). % third column
winningComb([1,5,9]). % \ diagonal
winningComb([3,5,7]). % / diagonal
```

Checking For Win in Haskell

Had to build a list of winning combinations,
loop through list

```
winningCombs =
   [[1,2,3],[4,5,6],[7,8,9],[1,4,7],[2,5,8],
     [3,6,9],[1,5,9],[3,5,7]]
xWins :: State -> Bool
xWins (State _ xlist _) = winner xlist
oWins :: State -> Bool
oWins (State _ _ olist) = winner olist
winner :: [Int] -> Bool
winner list = or
  [ordSubset comb list | comb <- winningCombs]
```

Cuts in Prolog

Many other situations: multiple possibilities, mutually exclusive. Had to use lots of cuts to avoid reduncant/inefficient tests

```
minimax(State, _, Value) :-
   terminal(State, Value),
   !.

minimax(State, BestMove, Value) :-
   moves(State, Moves),
   player(State, Player),
   bestMove(Player, Moves, BestMove, Value),
   remember(State, BestMove, Value).
```

Haskell Version

```
minimax :: State -> (State, Int)
minimax state
  | xWins state = (arbState, 2)
  | oWins state = (arbState, 0)
  | draw state = (arbState, 1)
  | otherwise = bestMove (player state)
                         (moves state)
```

List Comprehensions

Very useful in Haskell! Avoided some extra recursive functions.

```
moves (State True xlist olist) = -- X's turn
  [(State False (addToList xlist square) olist) |
    square <- [1..9], not (elem square xlist),
    not (elem square olist)]
```

Prolog version: needed recursive helper to do the equivalent of looping through `[1..9]`.

Also very convenient to be able to call a function inside an expression.

Pattern Matching & Failures in Prolog

```
ordSubset([],_).
ordSubset([X|Tail1],[X|Tail2]) :-
  !,
  ordSubset(Tail1,Tail2).
ordSubset([X|Tail1],[Y|Tail2]) :-
  X > Y,
  ordSubset([X|Tail1], Tail2).
```

Note: fails if second parameter is `[]` or `X < Y`.

Pattern Matching & Failures in Haskell

```
ordSubset :: [Int] -> [Int] -> Bool
ordSubset [] _ = True
ordSubset _ [] = False
ordSubset (x:xs) (y:ys)
  | x == y = ordSubset xs ys
  | x > y = ordSubset (x:xs) ys
  | otherwise = False
```

Note: **`False`** cases had to be included.

Challenge (1)

Practice problem in both languages: Modify ttt programs to take more advantage of symmetry.

General idea: Two states are equivalent if you can make one out of the other by "flip" or rotation.

X		
	O	
		X

&

		X
	O	
X		

X	O	

&

	O	X

or:

		O
		X

or:

	O	X

Challenge (2)

Modify move predicate/function.
Examine list of possible moves.
If any 2 are equivalent, remove one of them.

Suggestion: Start with just one kind of transformation and get the logic working, then add more

Experiment to see if time spent comparing for symmetry is worth it.