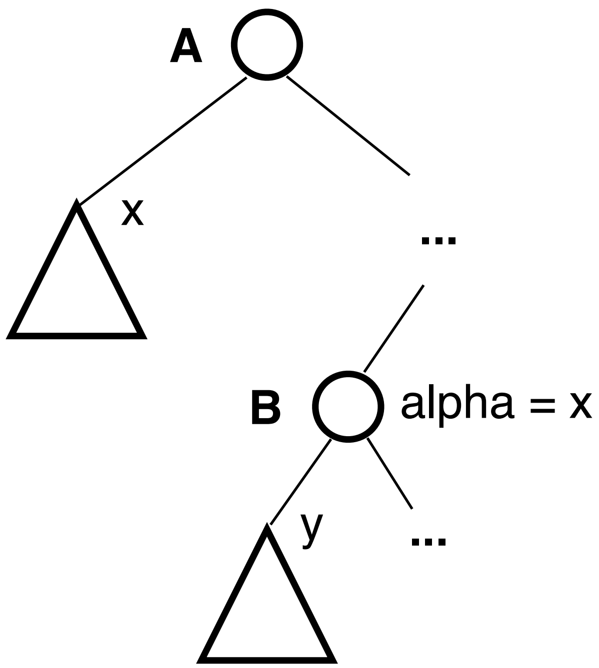
Alpha-beta pruning

```
def minimax(game, player=A, alpha=-inf, beta=inf):
    if terminal(state):
        return utility(state)
    best = -inf if player==A else inf
    for action, next_state in successors(state):
        if player == A:
            if best >= beta: return best
            util = minimax(next_state, player=B,
                           alpha=best, beta=beta)
            best = max(best, util)
        if player == B:
            if best <= alpha: return best
            util = minimax(next_state, player=A,
                           alpha=alpha, beta=best)
            best = min(best, util)
    return best
```

Midterm review:

- Cheat sheet

Alpha-beta pruning

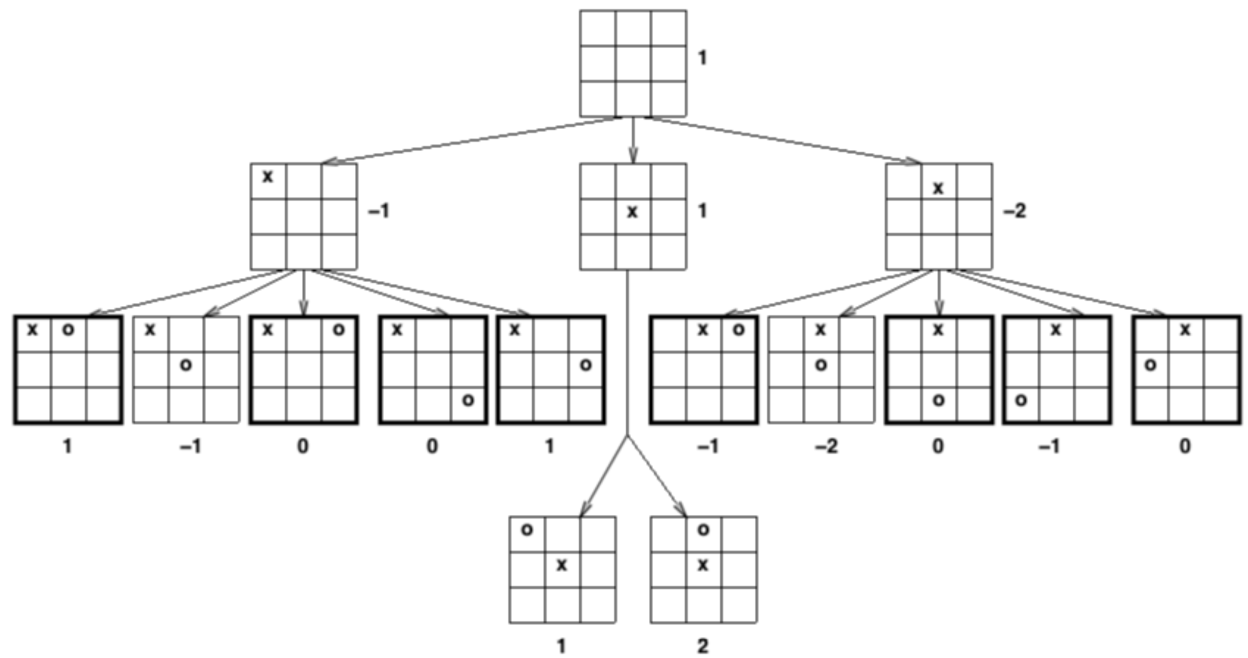


- Node 1: **>= x**
- Node 2: **<= y**.
- Do we need to search the right side of B node?
    - Suppose y <= x
    - Two options: 1) We find more positive -- don't care, B will choose y. 2) We find more negative -- A will make a different choice at node 1 anyway.

Example

Properties?
- Pruning does not affect result! Still optimal. You always want to use it.
- Effectiveness depends on move ordering.
- With perfect ordering, time complexity = O(b^(m/2)). (Note: Not (b^m)/2 !).
- Can usually get good ordering. In chess, consider capturing pieces, putting in check etc.

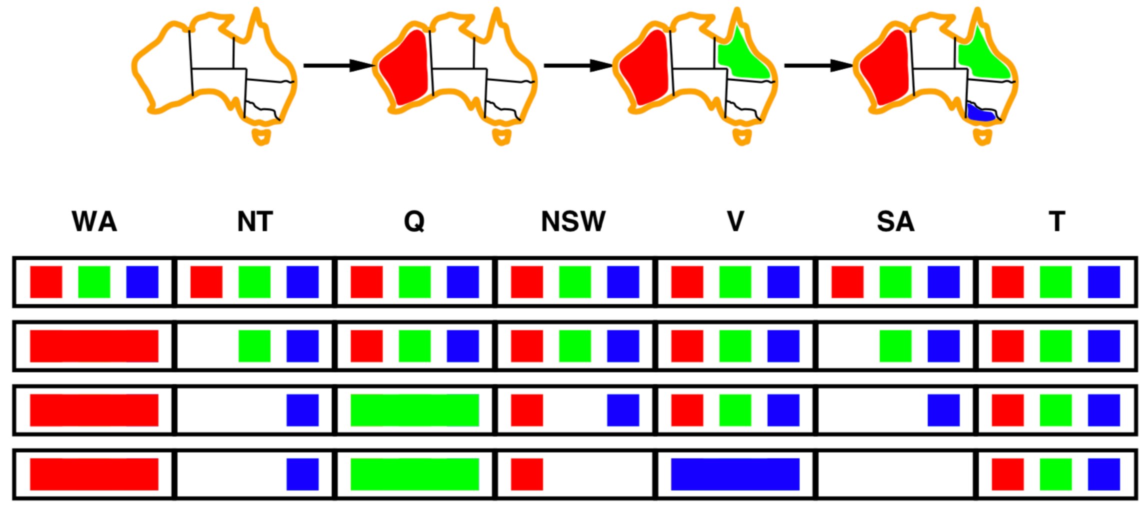
Example: tic-tac-toe



- Xn as the number of rows, columns, or diagonals with exactly n X's and no O's. Similarly, On is the number of rows, columns, or diagonals with just n O's. The utility function assigns +1000 to any position with X3=1 and -1000 to any position with O3=1.
- Eval(s)=3X2(s)+X1(s)-(3O2(s)+O1(s)).
- Bold = do not need to be evaluated.

Forward checking

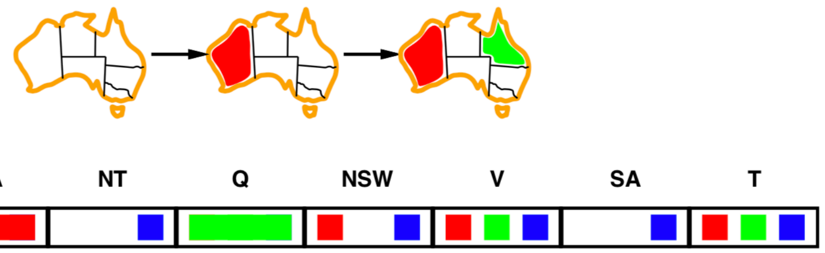
Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



Arc consistency

$X \to Y$ is consistent iff

for **every** value x of X there is **some** allowed y



- Run arc consistency on X (with neighbors of X) every time X loses a legal value. When assigning to Y, will run on all of its neighbors.
- Just assigned green to Q. SA loses a legal value, so run arc consistency on that.
    - SA -> NSW: No problem, blue is okay.
    - NSW -> SA: Blue doesn't work, no value for SA. NSW just lost a value, now need to run with its neighbors again.
    - NSW -> V: All good
    - V -> NSW: Red doesn't work.
    - SA -> NT: Blue doesn't work. This solution is inconsistent.
- This is expensive: O(n^2 d^2). It's still helpful because it reduces the search depth of backtracking.

-

Backtracking search

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- Forward checking: write pencil marks

Hill climbing search

```
def hill_climbing(csp, max_steps):
    current = choose_assignment(csp)
    for i in 1 .. max_steps:
        if csp.satisfies(current):
            return current
        var = choose_variable(csp, assignment)
        val = min_conflicts(csp, assignment, var)
        current[var] = val
    return "failure"
```

- We have choices in choose_variable,
- choose_variable: Should be a variable that violates some constraints.
- min_conflicts: whichever value violates the fewest assignments

Hill climbing search

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- Fill in all squares with pencil
- Count number of contraints violated
- Pick square to update; calculate constraints violated by each value

Converting to CNF

(A or B) => (B and C)
-(A or B) or (B and C)
(-A and -B) or (B and C)
(-A or B) and (-B or B) and (-A or C) and (-B or C)

Admissible and consistent heuristics

Probability

Suppose we generate a random bit string of length 4. Is whether or not the string has an even number of 1s independent from whether the string ends in a 1?

- Conditional probability
- Marginalization

Vars: B1, B2, B3, B4
P(even number of 1s) = 1/2
P(even number of 1s | last digit is 1) = 1/2