Games and CSP search

1. Games:

A) General Minimax search:

```
function max-value(state, depth)
    1. if state is an end-state (end of game) or depth is 0
           return SV(state)
    2. v = -infinity
    3. for s in get-all-next-moves(state)
          v = max(v, min-value(state, depth-1))
    4. return v

function min-value(state, depth)
    1. if state is an end-state (end of game) or depth is 0
           return SV(state)
    2. v = +infinity
    3. for s in get-all-next-moves(state)
          v = min(v, max-value(state, depth-1))
    4. return v
```

SV(state) returns the static value of the game state. These are the numbers we give you as the leaf nodes in the game search tree. They are generally in the perspective of the current player.

At the very **top level** your max-value function should return the best move (as well as the best value), so a slight modification is needed here:

```
function top-level-max-value(state, depth)
    1. if state is an end-state (end of game) or depth is 0
           return SV(state)
    2. [v, move] = [-infinity, nil]
    3. for s, move in get-all-next-moves(state)
          [v, move] = max( [v, move],  [min-value(state, depth-1), move] )
    4. return [v, move]
```

For example, calling top-level-max-value(start-state, 10) will search the tree up to depth of 10. and return the best move and its SV.

B) Minimax Search As One Function:

The functions max-value and min-value in minimax search can be expressed as a single function;

The single function version works for zero-sum games, which assumes that:

SV(A, state) = -SV(B, state)

You can find the static value of a particular game board from A's perspective by negating the static value from B's perspective.

```
function minimax(state, depth, player)
  1. if state is an end-state (end of game) or depth <= 0
        return SV(player, state)
  2. v = -infinity
  3. for s in get-all-next-moves(state)
        v = max(v, -1 * minimax(state, depth-1, next(player))
  4. return v
```

Here player = {0, 1} (A or B)
next(player) returns the next player to play; so next(0) = 1 and next(1) = 0.

C) General N-player Minimax:

For a game with N-players taking turns in sequential order A, B, C, D...n :
You can use a general version of the one-function minimax to search for the best move.

```
function n-player-minimax(state, depth, player)
  1. if state is an end-state (end of game) or depth is 0 then
        return SV-vector(state)

  2. v = (-infinity, ..., -infinity)    # v is a vector of n static values.
  3. for s in get-all-next-moves(state)
       v = vector-max[player](v, nplayer(state, depth-1, next-player(player)))
  4. return v
```

The leaves of the n-player game tree now have a vector of static value scores:

SV-vector(state) => (SV(player_1), SV(player_2), SV(player_n))

vector-max[player] is a function that will find the max vector wrt to the player-th element in the vectors.

Example:

vector-max[0]((**1**, 2, 3), (**2**, 3, 1)) returns (2, 3, 1) since wrt to the 0th element 2 > 1
vector-max[2]((1, 2, **3**), (2, 3, **1**)) returns (1, 2, 3) since wrt to the 2nd element 3 > 1

Ties are broken with preference for the left most vector (or the one found earlier.)

D) Minimax with Alpha Beta Pruning:

```
function max-value(state, depth, alpha, beta):
      1. if state is an end-state (end of game) or depth is 0 then
```

```
        return SV(state)
    2. v = -infinity
    3. for s in get-all-next-moves(state)
        v = max(v, min-value(state, depth-1, alpha, beta))
        alpha = max(alpha, v)
        if alpha >= beta then
            prune!
            return alpha
    4. return v

function min-value(state, depth, alpha, beta):
    1. if state is an end-state (end of game) or depth is 0
        return SV(state)
    2. v = +infinity
    3. for s in get-all-next-moves(state)
        v = min(v, max-value(state, depth-1, alpha, beta))
        beta = min(beta, v)
        if alpha >= beta then
            prune!
            return beta
    4. return v
```

For example, calling:

max-value(start-state, 10, -infinity, +infinity)

Will do minimax-with-alpha-beta-search for up to depth 10.
Note: like in minimax, you will need to implement a **top-level-max-value** to get return the best next move (rather than the best value).

E: Theoretical Pruning Bounds for Alpha Beta Search

Let s be the number of static evaluations done.
For a complete tree of depth d with a branching factor of b.

If d is even:

$$s = 2b^{d/2} - 1$$

If d is odd

$$s = b^{(d+1)/2} + b^{(d-1)/2} - 1$$

Number of nodes statically-evaluated for a maximally-pruned tree is $O(b^{d/2})$

F: Tree reordering to maximizing pruning

To ensure maximum amount of pruning under alpha-beta-search (as expressed in the equations above), we can apply the following tree-rotation algorithm.
This rotation algorithm is often used when alpha-beta-search is used in conjunction with progressive deepening:

```
function tree-rotation-optimizer:
    From level in [1 ... n]
        if level is MAX then
            sort nodes at this level from smallest to largest
        else if level is MIN then
            sort nodes at this level from largest to smallest
```

NOTE: On Quizzes we usually ask you to fill in the static values at inner nodes by running minimax from the leaf up to root.

G: alpha-beta-search + progressive deepening + tree-rotation-optimizer

Here is a rough sketch implementation of how these 3 elements work together as a speed up for game search.

Step 1. **Run alpha-beta-search up to depth d**.
This will yield some static values for (un-pruned) leaf nodes at depth d.

Step 2. Using these computed static values compute the values associated with intermediate nodes by **running minimax from leaf up to root.**

Step 3. **Run the tree-rotation-optimizer** on this (possibly-pruned) tree .

Step 4. **Record** at each node, the ordering of its children.

Step 5. **Run alpha beta search at depth d + 1.**
But when computing the next-moves for any node, lookup the node-ordering from step 4. **Evaluate alpha beta with nodes sorted using this ordering.** NOTE: Any nodes not in the ordering will automatically receive the lowest priority; such nodes come from having been pruned in alpha beta search.

Note:
The results of the tree rotation optimizer only *influences* alpha beta search evaluation order. It will not actually **guarantee** maximum pruning at every stage. This is because:

1. We are using the rotation algorithm with static values at d to approximate the values of the tree at d+1.
2. Pruned nodes from alpha-beta-search will not be used by the rotation algorithm.

While the tree-rotation doesn't yield maximum pruning, it does **enhance** pruning, so we still get a worthwhile speed up.

2. Constraint Satisfaction Problems

Constraint Types

- Unary Constraints - constraint on single variables (often used to reduce the initial domain)
- Binary Constraints - constraints on two variables C(X, Y)
- n-ary Constraints constraints involving n variables. Any n-ary variables n > 2
    - Can always be expressed in terms of Binary constraints + Auxiliary variables

Search Algorithm types:

1. DFS w/ BT + basic constraint checking (*)
    - Check current partial solution see if you violated any constraint.
2. DFS w/ BT + forward checking (*)
    - Assume the current partial assignment, apply constraints and reduce domain of other variables.
3. DFS w/ BT + forward checking + propagation through singleton domains (*)
    - If during forward checking, you reduce a domain to size 1 (singleton domain)
    - Then assume the assignment of singleton domain, repeat forward checking from singleton variable.
4. DFS w/ BT with propagation through reduced domains (AC-2)

You can replace DFS with Best-first Search or Beam-search if variable assignments have "scores", or if you are interested in the *best* assignment.

Variable domain (VD) table - book keeping for what values are allowed for each variable.

Variable Selection Strategies:

1. **Minimum Remaining Values** Heuristic (MRV)
    - Pick the variable with the smallest domain to start.
2. **Degree Heuristic** - usually the tie breaker after running MRV
    - When domains sizes are same (for all variables), choose the variable with the most constraints on other variables (greatest number of edges in the constraint graph)

Forward Checking Pseudo Code:

Assume all constraints are Binary on variables X, and Y.
constraint.get_X gives the X variable
constraint.get_Y gives the Y variable
X.domain gives the current domain of variable X.

forward_checking(state)

1. run basic constraint checking fail if basic check fails
2. Let X be the current variable being assigned,
3. Let x be the current value being assigned to X.
4. **check_and_reduce**(state, X=x)

function check_and_reduce(state, X=x)

1. neighbor_constraints = get_constraints(X)
2. foreach constraint in neighbor_constraints:
3. Y = constraint.get_Y()
4. skip if Y is already assigned
5. foreach y in Y.get_domain()
6. if check_constraint(constraint, X=x, Y=y) fails
7. reduce Y's domain by removing y.
8. if Y's domain is empty return fail
9. return success

forward_checking_with_prop_thru_singletons(state)

1. run forward checking
2. queue = find all unassigned variables that have domain size = 1
3. while queue is not empty

4. X = pop off first variable in queue
5. **check_and_reduce**(state, X = x.DOMAIN[0])
6. singletons' = find all new *unvisited* *unqueued* singletons
7. add singletons' to queue

forward_checking_with_prop_thru_reduced_domains(state)

1. run arc-consistency(state)

"arc-consistency" is also known as "pure-consistency":

arc-consistency(state):

1. queue = all (X, Y) variable pairs from binary constraints
2. while queue not empty
3. (X, Y) get the first pair in queue
4. if **remove-inconsistent-values**(X, Y) then
5. for each Y' neighbors(X) do
6. add (Y', X) to queue if not already there

remove-inconsistent-values(X, Y)

1. reduced = false
2. for each x in X.domain
3. if **no** y in Y.domain satisfies constraints(X=x, Y=y)
4. remove x from X.domain
5. reduced = true
6. return reduced
7.

Definitions:

- k-consistency. If you set values for k-1 variables, then the kth variable can still have some non-zero domain.
- 2-consistency = arc-consistency.
  - For all variable pairs (X, Y)
    - for any settings of X there is an assignment available for Y
- 1-consistency = node-consistency
  - For all variables there is some assignment available.

- Forward Checking is only checking consistency of the current variable with all neighbors.
- Arc-consistency = propagation through reduced domains
- Arc-consistency ensures consistency is maintained for all **pairs** of variables

6.034 Artificial Intelligence

Fall 2010