Minimax and Alpha-Beta Implementation Details, Guidelines, and Tips

Here is some information that should help you implement the three algorithms for Project 2. Luckily, there is less bookkeeping than in Project 1.

- *How do I implement a state and an action?*

  These concepts are almost identical to their counterparts in state-space search. The only difference is that in adversarial search, we usually **do** want a way of representing an action, because we will want to store those actions in our transposition table for when our agents are playing the game.

- *How do I implement a node?*

  In minimax and alpha-beta, implementing a node is different than in state-space search (A*) because we do not need to store the cost of an action nor parent pointers. In fact, when using a transposition table (as we are in this project), we often don't need to make a data structure for the game tree anyway. It's usually too big to hold in memory all at once, so what we will do is use a very small class or struct (or even a 2-element list or tuple), that will simply hold the minimax value of a state and the action representing the best move from the state. In the pseudocode, this class is called MinimaxInfo.

- *How do I implement the functions in the book like TO-MOVE(s), ACTIONS(s), RESULT(s, a), IS-TERMINAL(s), UTILITY(s), IS-CUTOFF(s), and EVAL(s)?*

  Like in project 1, some of these are optional and might be more trouble than their worth to write, however some of them **definitely** be written explicitly.

  The functions you should absolutely write are:

  - UTILITY(s): This function is needed to calculate the utility of a terminal state (whether the state is a win, loss, or draw). It will also consider the number of moves the game has lasted, so it almost certainly should be its own function in your code.

  - EVAL(s): Like UTILITY(s), this function is needed for alpha-beta heuristic search when we can't search all the way to the leaf (terminal) nodes of the game tree. It should almost certainly be its own function in your code.

  The functions you may or may not want to write explicitly are:

  - ACTIONS(s) and RESULT(s, a), like in Project 1, are only used when "expanding" a node of the game tree to look for successor nodes (though the pseudocode doesn't explicitly use the term "expanding.") However, unlike in Project 1, we always need to explicitly represent the concept of an action in adversarial search, so I recommend writing these two functions, or at least noting how code you've already written emulates them. In other words, you may not have a function called RESULT, but you might have already written a function that does something similar, so you can just use that.

  - TO-MOVE and IS-TERMINAL you can write if you want, but they will be very simple, so you may not want to bother. It's up to you.

  - IS-CUTOFF, depending on how complicated you want to make this for Part C, may or may not need its own function.

- *How do I implement the transposition table?*

  A transposition table is always a map from states to pairs of minimax values and actions. These pairs are called "MinimaxInfo" objects in the pseudocode. The book's pseudocode doesn't use transposition tables, but you'll notice that it does use pairs of minimax values and actions frequently (e.g., "v2, a2" or "return v, move"). So a MinimaxInfo object just holds those pairs explicitly, and the transposition table allows us to look up a state and get back the minimax value and the corresponding "best move."

Pseudocode details:

MinimaxInfo is a struct or class that stores the minimax value of a state *and* an action representing the best move from that state. In the code below, we refer to these as VALUE and ACTION.

table (the transposition table) is a map that stores a mapping between game states and MinimaxInfo objects.

Is-Terminal(state) is a function that returns true if state is a terminal state (meaning the game has been won, lost, or drawn).

Utility(state) is a function that determines the numerical worth of a *terminal* state. Typically these values are positive for a MAX win and negative for a MIN win, with 0 meaning a draw.

Is-Cutoff(state, depth) is a Boolean function that returns true if the search should be cut off at this state. Usually this is based on the depth of the state in the search tree; for instance, we could choose to cut off all searches after looking a fixed number of moves ahead. This function can be based on other factors of the state, however, if desired, leading to some branches of the search tree searching deeper than other branches.

Eval(state) is a function that uses a heuristic to determine the numerical worth of a *non-terminal* state. Like the **Utility** function, these values should be positive for states that will likely lead to MAX wins, and negative for states that will likely lead to MIN wins. For MAX, higher values (closer to positive infinity) should correlate with a higher probability of winning. For MIN, lower values (closer to negative infinity) should correlate with a higher probability of winning. Important: In order to prioritize "guaranteed wins" over "heuristic wins" (which may or may not be guaranteed), the absolute value of a number returned from **Eval** should be less than the absolute value of a number returned from **Utility**.

Actions(state) is a function that returns all *legal* actions from a state.

Result(state, action) is a function that takes a state and an action and returns a new state (the successor state, or child state) that results from taking the action in the original state. This function assumes the action is a legal action from the state.

To-Move(state) is a function that takes a state and returns the player who moves next from that state. (This might just be a variable stored in the state object itself.)

Important Notes:

- My versions of the pseudocode combine the MAX-VALUE and MIN-VALUE functions together since there is so much common code between them.

- For Part A (regular minimax), the code should be run before the game is played, and then the transposition table should be used to look up all moves in the game the computer makes. This will always work because minimax searches the entire game tree.

- For Part B (minimax with alpha-beta), the code should be run before the game is played, and then the transposition table should be used to look up all moves in the game the computer makes. However, if a state is encountered that is **not** in the transposition table (this is possible if the human player chooses a sub-optimal move that was pruned), then the alpha-beta algorithm should be re-run from this state to determine the optimal move from this point in the game. This may happen multiple times, if the human player continues to choose moves that were pruned. ☺

- For Part C (minimax with alpha-beta and heuristics), the code should **not** be run before the game is played. Because Part C only searches to a limited depth, the code should be run each time --- from the current state, not the initial state --- whenever the computer needs to choose a move. The transposition table should be cleared before each new search, because the tree will be searched deeper each time (to be clear, the depth parameter is not changing, but because we will be beginning the search deeper in the tree each time, we will be able to look deeper and deeper into the tree).

- States that are pruned are not stored in the transposition table. They correspond to states that will never be chosen during optimal play, so there's no reason to store them. The code reflects this.

Pseudocode for minimax with a transposition table (Part A):

Initial call to this function should be MINIMAX-SEARCH(*initial_state*, *an_empty_map*).

```
function MINIMAX-SEARCH(state, table) returns a MinimaxInfo object
    if state in table then
        return table[state]

    else if IS-TERMINAL(state) then
        util ← UTILITY(state)
        info ← a new MinimaxInfo object with VALUE = util, ACTION = null
        // action is null because this is a terminal state
        table[state] ← info     // add this state to the transposition table
        return info

    else if TO-MOVE(state) == MAX then
        v ← -∞
        best_move = null
        for each action in ACTIONS(state):
            child_state ← RESULT(state, action)
            child_info = MINIMAX-SEARCH(child_state, table)
            v2 = child_info.VALUE
            if v2 > v then
                v = v2
                best_move = action
        info ← a new MinimaxInfo object with VALUE = v, ACTION = best_move
        table[state] ← info     // add this state to the transposition table
        return info

    else // TO-MOVE(state) == MIN
        v ← +∞
        best_move = null
        for each action in ACTIONS(state):
            child_state ← RESULT(state, action)
            child_info = MINIMAX-SEARCH(child_state, table)
            v2 = child_info.VALUE
            if v2 < v then
                v = v2
                best_move = action
        info ← a new MinimaxInfo object with VALUE = v, ACTION = best_move
        table[state] ← info     // add this state to the transposition table
        return info
```

Pseudocode for alpha-beta search with a transposition table (Part B):

Initial call to this function should be ALPHA-BETA-SEARCH(*initial_state*, -∞, +∞, *an_empty_map*).

```
function ALPHA-BETA-SEARCH(state, alpha, beta, table) returns a MinimaxInfo object
    if state in table then
        return table[state]

    else if IS-TERMINAL(state) then
        util ← UTILITY(state)
        info ← a new MinimaxInfo object with VALUE = util, ACTION = null
        // action is null because this is a terminal state
        table[state] ← info     // add this state to the transposition table
        return info

    else if TO-MOVE(state) == MAX then
        v ← -∞
        best_move = null
        for each action in ACTIONS(state):
            child_state ← RESULT(state, action)
            child_info = ALPHA-BETA-SEARCH(child_state, alpha, beta, table)
            v2 = child_info.VALUE
            if v2 > v then
                v = v2
                best_move = action
                alpha = MAX(alpha, v)
            if v >= beta then                          // prune tree, don't store state in TT
                return a new MinimaxInfo object with VALUE = v, ACTION = best_move

        info ← a new MinimaxInfo object with VALUE = v, ACTION = best_move
        table[state] ← info     // add this state to the transposition table
        return info

    else // TO-MOVE(state) == MIN
        v ← +∞
        best_move = null
        for each action in ACTIONS(state):
            child_state ← RESULT(state, action)
            child_info = ALPHA-BETA-SEARCH(child_state, alpha, beta, table)
            v2 = child_info.VALUE
            if v2 < v then
                v = v2
                best_move = action
                beta = MIN(beta, v)
            if v <= alpha then                         // prune tree, don't store state in TT
                return a new MinimaxInfo object with VALUE = v, ACTION = best_move

        info ← a new MinimaxInfo object with VALUE = v, ACTION = best_move
        table[state] ← info     // add this state to the transposition table
        return info
```

Pseudocode for alpha-beta search with heuristics and a transposition table (Part C):

Initial call to this function should be
ALPHA-BETA-HEURISTIC-SEARCH(*initial_state*, -∞, +∞, 0, *an_empty_map*).

```
function ALPHA-BETA-HEURISTIC-SEARCH(state, alpha, beta, depth, table) returns a MinimaxInfo object
    if state in table then
        return table[state]

    else if IS-TERMINAL(state) then
        util ← UTILITY(state)
        info ← a new MinimaxInfo object with VALUE = util, ACTION = null
        // action is null because this is a terminal state
        table[state] ← info        // add this state to the transposition table
        return info

    else if IS-CUTOFF(state, depth) then
        heuristic ← EVAL(state)
        info ← a new MinimaxInfo object with VALUE = heuristic, ACTION = null
        // action is null because we are cutting off the search here
        table[state] ← info        // add this state to the transposition table
        return info

    else if TO-MOVE(state) == MAX then
        v ← -∞
        best_move = null
        for each action in ACTIONS(state):
            child_state ← RESULT(state, action)
            child_info = ALPHA-BETA-HEURISTIC-SEARCH(child_state, alpha, beta, depth+1, table)
            v2 = child_info.VALUE
            if v2 > v then
                v = v2
                best_move = action
                alpha = MAX(alpha, v)
            if v >= beta then                          // prune tree, don't store state in TT
                return a new MinimaxInfo object with VALUE = v, ACTION = best_move

        info ← a new MinimaxInfo object with VALUE = v, ACTION = best_move
        table[state] ← info        // add this state to the transposition table
        return info

    else // TO-MOVE(state) == MIN
        v ← +∞
        best_move = null
        for each action in ACTIONS(state):
            child_state ← RESULT(state, action)
            child_info = ALPHA-BETA-HEURISTIC-SEARCH(child_state, alpha, beta, depth+1, table)
            v2 = child_info.VALUE
            if v2 < v then
                v = v2
                best_move = action
                beta = MIN(beta, v)
            if v <= alpha then                         // prune tree, don't store state in TT
                return a new MinimaxInfo object with VALUE = v, ACTION = best_move

        info ← a new MinimaxInfo object with VALUE = v, ACTION = best_move
        table[state] ← info        // add this state to the transposition table
        return info
```