AI Game Engine

m by n Tic Tac Toe with *k* strikes and *s* obstacles

Abhishek Mishra

May 18th, 2014

1 Introduction

In this project, we desgin the A.I. for a computer agent that play a modified version of the Tic Tac Toe.The board is *m by n* where either agent needs to get a certain *k* number of strikes to win the game. Additionally, there can also be obstacles and/or X's and/or O's already set up on the board before the start of the game such that agents will be required to heuristically compute the best move to go for.

2 Theory

The following are some of the techniques that have been used in designing the game A.I.

1. Min-max Search
2. Alpha-Beta Pruning
3. Zobrist Hashing
4. Static Evaluation
5. Dynamic Evaluation

The A.I. relies on key idea that it is possible to lookahead a few moves down the decision tree and compute a *dynamic evaluation* that suggests the cost of making that move compared to other moves. Since the board size can vary from very small with no obstacles to very large with high density of obstacles, approaches are taken to optimize the lookahead computation by using *alpha/beta pruning* and *zobrist hashing*.

Alph/beta pruning allows us to discard a subtree when it seems that further exploration will only minimimze (in case of X) or maximize(in case of O) our computed heuristic and as such will work against us. This is because either

agent needs to compute the alpha (for X) and beta (for O) at alternating depths and attempt to reach an interior node where successive exploration will work against them or a leaf node beyond which no more computation is possible. Any computation at the root or the leaf or at the maximum depth set is known as *static evaluation*. Hence, a node's static evaluation is the cost of being in that state. A node's *dynamic evaluation* is the *net static evaluation* computed for a certain depth of the subtree of that node and is the cost of choosing that node such that this cost will favor successive moves for this agent after the next move.

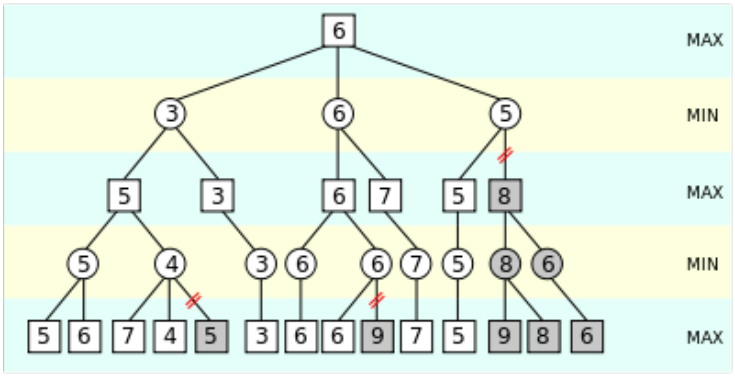

Figure 1: An illustration of alphabeta pruning. The grayed-out subtrees need not be explored (when moves are evaluated from left to right), since we know the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. The max and min levels represent the turn of the player and the adversary, respectively.

Other technique used here is Zobrist Hashing. Since computng dynamic evaluations takes large amount of recursions, it is not optimal to go through the minmax search especially because the state of the board changes by only 1 row/column with every move. Hence we hash every new static evaluation we compute. This works because each state of the board is unique, i.e., there are no two or more states that look the same per game, which allows us to map these states to their static evaluation.

Static Evaluation is simply the sum of a polynomial derived from any given state. It is the cost of achieving the next state and is used to evaluate the best move across different prospective next states. The polynomial used here is given by

$$y = c_0 + c_1 B + c_2 B^2 + c_3 B^3 + c_4 B^4 + \ldots c_n B^n$$

In the above equation each c_i represents the cost coefficient of the i_{th} row or column. In our case, the cost coefficient is evaluated by taking into account

- the maximum allowable opponent symbols.
- the maximum allowable obstacles
- the minimum number of empty spaces

Hence, if all the conditions are met, the coefficient for a given row is incremented by 1 and the same procedure is followed n times for the given state to construct n coefficients. Note that the *row* is only a term here since we do the same for column by transposing the state of the board and calculating static eval for that state too. We calculate this in favor of both X's and O's and the *net cost heuristic* is given by

$$\Big(staticEval_{mySymbol} + staticEval^{T}{}_{mySymbol}\Big) - \Big(staticEval_{opponentSymbol} + staticEval^{T}{}_{opponentSymbol}\Big)$$

where *mySymbol* stands for the symbol you are playing for and *opponentSymbol* is the one your opponent plays as. The T superscript stands for the transposed state. Recall, that we need to calculate static eval both for the regular and the transposed state and add them up.

3 Implementation

We use Python 3.2 to implement the algorithms described in Section 2. The simplicity of the language allows for quick results and changes to be incorporated very easily. Following are some of the key algorithms implemented in the project.

3.1 Min-max Search with Alpha-beta pruning

```
# Returns the static eval by computing static and dynamic
    evaluations by looking ahead down the tree.
# Uses alpha-beta pruning to discard any path of lower (higher)
    interest
def minmaxAlphaBeta(state, moveToAttainState, depth, alpha, beta):
    global stats
    if depth != 0:
        stats[3]+=1                                          #
            Increment the number of dynamic evaluation since we're
            at an interior node
    if  depth == 0 or (isLeaf(state)):
        stats[2]+=1
            # Increment the number of static evaluations since we're
            at a leaf or at the final depth
        return hashAndGetStaticEval(state, depth) #static evaluation
            (base case)
    successors, moves = successors_and_moves(state)
    if state[1] == 'X':
        i = 0
        while i < len(successors):
            alpha = max(alpha, minmaxAlphaBeta(successors[i],
                moves[i], depth - 1, alpha, beta))
            if beta <= alpha:
                stats[1]+=1              #Increment the number of
                    beta cutoffs.
```

```
            break                           # (* Beta cut-off *)
        i+=1
    if stats[4] == depth:           #If we are at the depth
        where we want to return the successor and the move
        required to attain it
        return [successors[i-1], moves[i-1], alpha]
    else:
        return alpha
else:
    i = 0
    while i < len(successors):
        beta = min(beta, minmaxAlphaBeta(successors[i], moves[i],
            depth - 1, alpha, beta))
        if beta <= alpha:
            stats[0]+=1             # Increment the number of
                alpha cutoffs.
            break                       #  (* Alpha cut-off *)
        i+=1
    if stats[4] == depth:                       #If we are at
        the depth where we want to return the successor and the
        move required to attain it
        return [successors[i-1], moves[i-1], beta]
    else:
        return beta
```

3.2 Zobrist Hashing

```
# Initialize zobrist hashtable with values
def iniZobristHash(rows, cols):
    global zobristHash
    zobristHash = [[[0 for i in range(4)] for j in range(n)] for
        z in range(m)]
    for i in range(rows):
        for j in range(cols):
            zobristHash[i][j][0] = random.randint(0, 4294967296) #
                for 'X'
            zobristHash[i][j][1] = random.randint(0, 4294967296) #
                for 'O'
            zobristHash[i][j][2] = random.randint(0, 4294967296) #
                for '-'
            zobristHash[i][j][3] = random.randint(0, 4294967296) #
                for ' '

# Returns the key corresponding to this state
def getStateKey(state):
    global zobristHash
    key = 0
    for i in range(len(state[0])):
```

```
    for j in range(len(state[0][0])):
        whatIsAtIJ = state[0][i][j]
        key^=zobristHash[i][j][getIndexForSymbol(whatIsAtIJ)]
return key
```

3.3 Static evaluation of a given game state

```
 # Retuns the static evaluation of a state with m Rows and n
     Columns
# Used as a HELPER FUNCTION for staticEval(state)
def static_eval_rows(gamestate, forSymbol):
    global n
    global this_k
    polynomial = 0
    for i in range(n):
        coeffCtr = 0
        for row in gamestate:
            #if (number_of_forSymbols_in_this_row == i) and (
                AgainstSymbol and '-' are not in row)
            numEmptySpaces = row.count(' ')
            numForSymbol = row.count(forSymbol)
            if (row.count(forSymbol) == i) and (numEmptySpaces
                >= this_k):
                coeffCtr+=1

        polynomial+= (10**i) * (coeffCtr)
    return polynomial
```