Assignment2 report

March 29, 2019

1 break through game

1.1 assignment 2 report

1.1.1 Melika Adabinezhad 810195536

1.1.2 Evaluation Function

My evaluation function is a linear combination of the agent's score and the opponent's score for non-terminal nodes: $myScore - opponentScore$ Mathematically, this kind of evaluation function is called a weighted linear function and it is constructed by giving proper weights to the features of game(like number of threats, scores, value of pawns, and so many other features and possibilities). Agent's score is the number of players it has eliminated. For terminal nodes the value is the maximum score available: 12(for Max nodes) and -12(for Min nodes) and 0(for draw).

1.1.3 Improvements

1.Add height effect so that we can reach the goal faster(by choosing the shallower goal. 2.Transposition Table. 3.Ordering nodes before expanding.This can improve execution time of alphabeta to O($b^{(d/2)}$) (at most!).

1.1.4 AlphaBeta pruning

For satisfying time constraints I use the iterative deepening algorithm. By calculating alphabeta on each height I always have a solution but if I have more time ,I can find better solutions. I have my own classes: 1.MyTree : has an additional function that adds another level to the tree at each iteration. 2.Mynode : didn't need some things like RandomEvalFunction but needed other things like color, opponentColor and height. 3.Result: I implement deadline by using signal handler so to preserve result i have to use an object instead of returning. 4.AlphaBeta(static class): root of the tree is always Max node. In function $computeAlphabeta$ I iterate on roots children which are min nodes and tell them to compute $findBestChildMin$. This function finds their minimum children by alphaBeta pruning. I also have a $findBestChildMax$ for Max nodes(except root). These two recent functions recursively call each other until the whole tree is processed. But how do they prune? Alpha–beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path: = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX. = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Alpha–beta search updates the values of and as it goes along and prunes the remaining

branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current or value for MAX or MIN, respectively. The general principle is this: consider a node n somewhere in the tree , such that Player has a choice of moving to that node. If Player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play. So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

```
In [ ]: class Agent:
            def __init__(self, color, opponentColor, time=None):
                self.color = color
                self.opponentColor = opponentColor
                self.timeToMove = time

            def move(self, board):
                # todo: you have to implement this agent.
                if(self.timeToMove == None):
                    height = 4
                    gameTree = MyTree(board, self.color, self.opponentColor, height)
                    AlphaBeta.calLeafValues(gameTree, height)
                    return AlphaBeta.computeAlphabeta(gameTree)
                else:
                    height = 1
                    gameTree = MyTree(board, self.color, self.opponentColor, height)
                    signal.signal(signal.SIGALRM, handler)
                    signal.alarm(self.timeToMove)
                    try:
                        result = Result()
                        AlphaBeta.calNextMove(
                            gameTree, height, self.timeToMove, result)
                    except TimeoutError as e:
                        return result.from_cell, result.to_cell


        class AlphaBeta:
            @staticmethod
            def calNextMove(tree, height, timeToMove, result):
                # iterative deepening
                for h in range(height, 12):
                    AlphaBeta.calLeafValues(tree, h)
                    result.from_cell, result.to_cell = AlphaBeta.computeAlphabeta(
                        tree)  # best_state
                    tree.addOneLevelAtBottom()

            @staticmethod
            def calLeafValues(tree, height):
                # evaluation function
                if(height % 2 == 0):
                    color = tree.color
```

```
            opponentColor = tree.opponentColor
            isMax = True

        else:
            color = tree.opponentColor
            opponentColor = tree.color
            isMax = False

        for leaf in tree.nodes[height]:
            AlphaBeta.calculateEvaluationFunc(leaf, color, opponentColor)
            if(isMax):
                leaf.setUtility(leaf.utility + leaf.height)
            else:
                leaf.setUtility(leaf.utility - leaf.height)

    @staticmethod
    def computeAlphabeta(tree):
        bestVal = -INFINITY
        beta = INFINITY
        bestNode = None
        children = tree.root.children
        for node in children:
            childVal = AlphaBeta.findBestChildMin(tree, node, bestVal, beta)
            if childVal > bestVal:
                bestVal = childVal
                bestNode = node

        return bestNode.getFromCell(), bestNode.getToCell()

    @staticmethod
    def findBestChildMin(tree, node, alpha, beta):
        if node.board.win(tree.color) and node.board.win(tree.opponentColor):
            return 0-node.height
        if node.board.win(tree.color):
            return 12+node.height
        elif node.board.win(tree.opponentColor):
            return -12-node.height

        value = INFINITY
        if(len(node.children) != 0):
            node.children.sort(key=lambda x: x.utility)
        for child in node.children:
            value = min(value, AlphaBeta.findBestChildMax(
                tree, child, alpha, beta))

            if value <= alpha:
                return value
```

```
            beta = min(value, beta)

        if(value == INFINITY):
            AlphaBeta.calculateEvaluationFunc(
                node, node.color, node.opponentColor)
        node.setUtility(node.utility - node.height)
        return node.utility

    @staticmethod
    def findBestChildMax(tree, node, alpha, beta):
        if node.board.win(tree.color) and node.board.win(tree.opponentColor):
            return 0+node.height
        if node.board.win(tree.color):
            return 12+node.height
        elif node.board.win(tree.opponentColor):
            return -12-node.height

        value = -INFINITY
        if(len(node.children) != 0):
            node.children.sort(key=lambda x: x.utility, reverse=True)
        for child in node.children:
            value = max(value, AlphaBeta.findBestChildMin(
                tree, child, alpha, beta))

            if value >= beta:
                return value

            alpha = max(value, alpha)

        if(value == -INFINITY):
            AlphaBeta.calculateEvaluationFunc(
                node, node.color, node.opponentColor)
        node.setUtility(node.utility + node.height)
        return node.utility

    @staticmethod
    def calculateEvaluationFunc(node, color, opponentColor):
        colorScore = 12 - node.board.getNumberOfArmy(opponentColor)
        opponentColorScore = 12 - node.board.getNumberOfArmy(color)
        node.setUtility(colorScore - opponentColorScore)
```

1.1.5 Advantages of AlphaBeta pruning

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half. The trick is that it is possible to compute the correct minimax

decision without looking at every node in the game tree. With proper ordering, AlphaBeta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax.