Christopher Anderson
Bronco ID: 014105585
Artificial Intelligence
Lab 2

MCTS for Black Jack

Implementation:

For my MCTS, I made a tree class that incorporated a node class that I also built.

```
"""
Create Tree Structure
"""
class MCTSTree:
    def __init__(self):
        self.hasRoot = False
        self.rootNode = None
        self.currentNode = None
        self.hasSplitNode = False
        self.inSplit = False
        self.isTraversing = False
"""
```

```
"""
Create Node Structure to use with tree
"""
class MCTSNode: #current game state
    def __init__(self, cards, dealer_cards):
        """
        Cards of Node
        """
        self.pcards = cards #our card value in current state
        self.dcards = dealer_cards
        self.handValue = get_value(cards)
        """
        Node properties
        """

        self.name = None
        self.parent = None
        self.isLeaf = False

        self.hitNode = None
        self.hitNodeExpected = 0

        self.standNode = None
        self.standNodeExpected = 0

        self.ddNode = None
        self.ddNodeExpected = 0

        self.splitNode = None
        self.splitNodeExpected = 0

        self.timesVisited = 0
        self.values = []
```

The tree was part of the roll out player constructor and was initialized after rolloutplayer was made. Each node was made of the current game state based off player hands and values.

Steps:

1) Exploring

The first time through a priority system was used, it would first check if it was the root, if so then see if we can split. Otherwise, Stand and double down nodes were then processed. Afterward the hit node was processed, if they busted then it would become a leaf(note we would check the top card of the deck for bust, the card is not actually added to the hand in this case so further simulation can be done), otherwise it would continue to add stand, double down and hits(not split since that's only off the root) until hit eventually turned into a leaf by busting. This method gave a fully built tree for each hand that was fully explored in every possible direction while not busting.

2) Selection

Once the tree was fully made and the final hit path turned into a leaf a selection process was done via UCB. The program would check to see if a split was current happening or not to include splitUCB. From here the best UCB action would be added to the action list. If this best action happened to be a non-leaf node such as a mid tree hit, the selection process would loop while not a leaf and continue to add in actions.

3) Back propagation and storing values

```
p = RolloutPlayer("Rollout", deck)# need to make a tree here and it needs to know if it needs to expand nodes or not
p.initTree(cards,dealer_cards)
# We create a new game object with the reduced deck, played by our rollout player
gl = Game(deck, p, verbose=False)
results = {}

for i in range(MCTS_N):#note this is looped 100 times
    p.reset()
    res = gl.continue_round(cards, dealer_cards, self.bet)
    #if p.myTree.hasSplitNode:
        #print("Made it back!")
    #print(p.myTree.currentNode.values)
    #print(p.myTree.currentNode.values)
    p.myTree.currentNode.values.append(res)
    #print(p.myTree.currentNode.values,p.myTree.currentNode.timesVisited)
    #print()
    temp = p.myTree.currentNode.parent
    """
    Back propogate results to master nodes off root and increment
    """
    while temp is not p.myTree.rootNode and temp is not None:
        temp.values.append(res)##will have to back propogate here
        temp.timesVisited = temp.timesVisited + 1
        temp = temp.parent
"""
```

In this step we now have a list of actions and our current node is our last node measured by our UCB. From here we go through a pseudo do-while loop. The results are stored in the current node and current node is then equal to current nodes parent through our tree and node structure. This will continue to loop, storing the result in all parents until our current node is our root node.

4) Action Selection

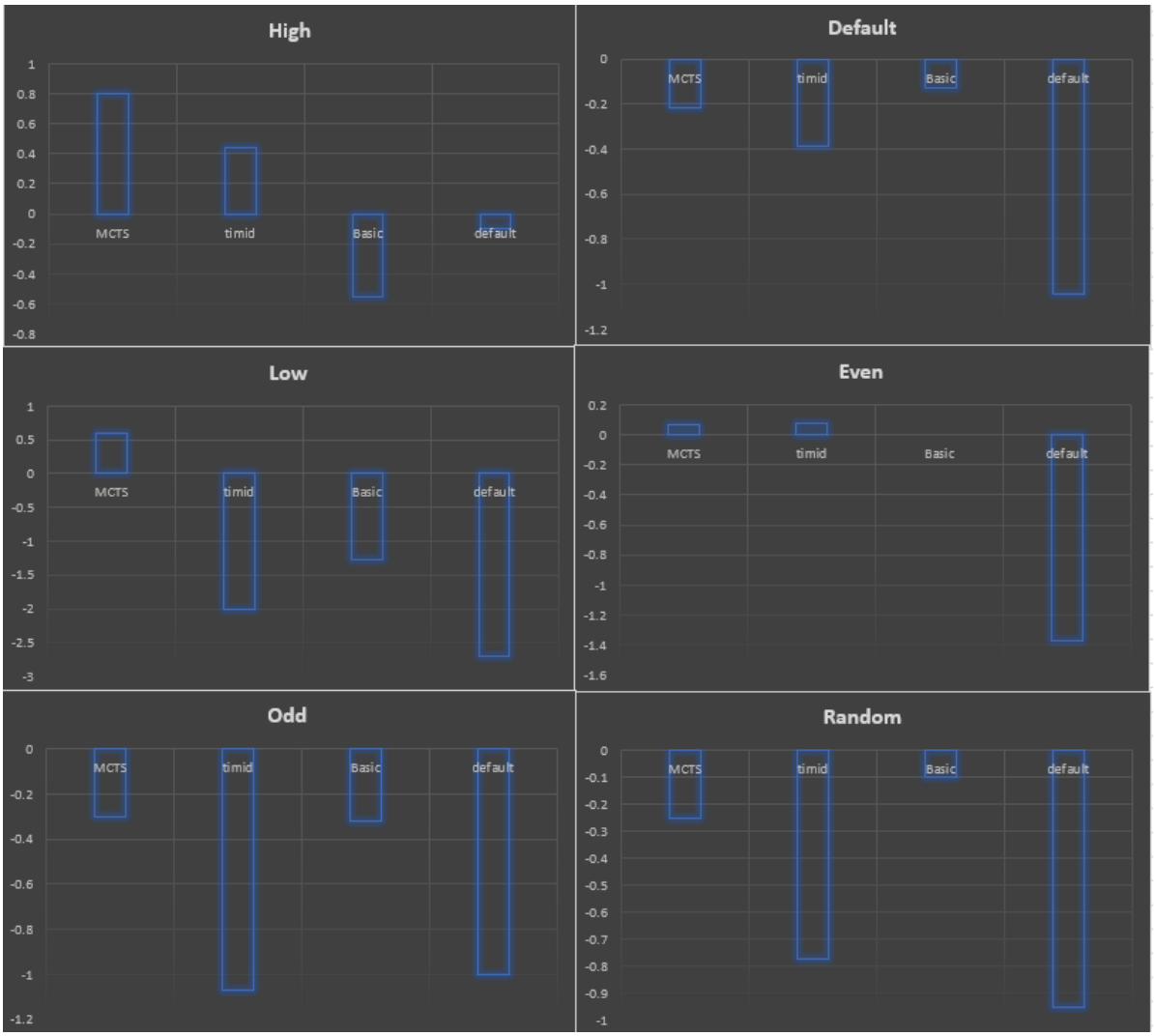
With all the values properly back propagated taking the value list from each of the root nodes children and dividing it by it's size gave the expected value. From here the max was chosen and that action taken.

Christopher Anderson
Bronco ID: 014105585
Artificial Intelligence
Lab 2

Part II.

Comparisons

Overall, MCTS generally performed either the best or very similar to the best in all deck cases.



Christopher Anderson
Bronco ID: 014105585
Artificial Intelligence
Lab 2

	MCTS	timid	Basic	default
high	0.8	0.44	-0.55	-0.1
default	-0.22	-0.39	-0.13	-1.04
low c = 30	0.6	-2	-1.28	-2.7
even	0.068	0.08	0	-1.37
odd	-0.3	-1.07	-0.32	-1
random	-0.25	-0.77	-0.1	-0.95

Part III

Parameters

For MCTS_N, increases this value seemed to have a decent effect up until the range of 1000 or so.

MCTS_N	expected
10000	0.69
1000	0.852
100	0.758
10	0.4

During the test increasing this value seemed to not effect the output as much- the above table is an example of 1000 test cases with some variance. Once the “enough” point was reached further increasing this value seemed to only increase runtime. Furthermore, there was a print function for UCB values, these values were already fairly set after a few hundred runs unless the C value was drastically increased to allow more fluctuation in choices.

For C value in UCB:

C	expected
10000	0.77
25	0.85
2	0.78
1	0.75

Values from “high”.

Generally a lower value worked for most tests, as seen in the comparisons in part II, these were done with a C = 2 value. However, the “low” deck actually drastically improved when raising C from 2 to 30, this was the only deck it made a big difference in. However, raising the C value made the default much worse, it went from averaging -0.2ish to anywhere from -0.2 to -0.5.