Hunter Leise

CSCI 3202

Problem Set 2

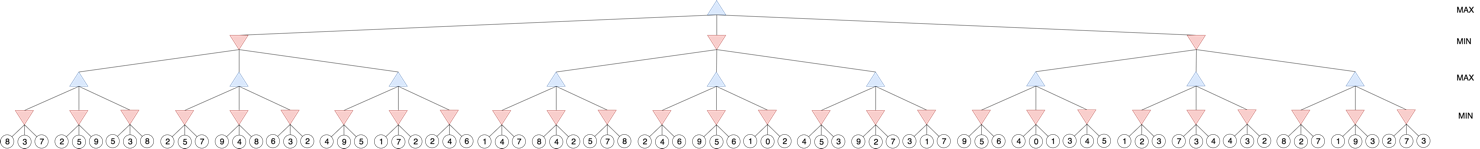
**Problem 2.1**

def ai\_move(self, color):  
 if color == **"red"**:  
 cumulative\_list = {key: self.graph.red\_counter[key] +   
 len(self.graph.graph[key])  
 for key in self.graph.graph.keys()}  
 else:  
 cumulative\_list = {key: self.graph.blue\_counter[key] +   
 len(self.graph.graph[key])  
 for key in self.graph.graph.keys()}  
  
 min\_list = sorted(cumulative\_list, key=cumulative\_list.get)  
 failing\_move = **""** for node1 in min\_list:  
 for node2 in min\_list:  
 if self.is\_valid\_move(node1, node2):  
 new\_graph = SimGraph(copy.deepcopy(self.graph.graph))  
 new\_graph.add\_edge(node1, node2, color)  
 if not new\_graph.triangle\_exists():  
 self.graph.add\_edge(node1, node2, color)  
 print(**"AI went from %s to %s"** % (node1, node2))  
 return  
 else:  
 failing\_move = node1 + node2  
  
 *# Only make a losing move if you absolutely have to* self.graph.add\_edge(failing\_move[0], failing\_move[1], color)  
 print(**"AI went from %s to %s"** % (failing\_move[0], failing\_move[1]))

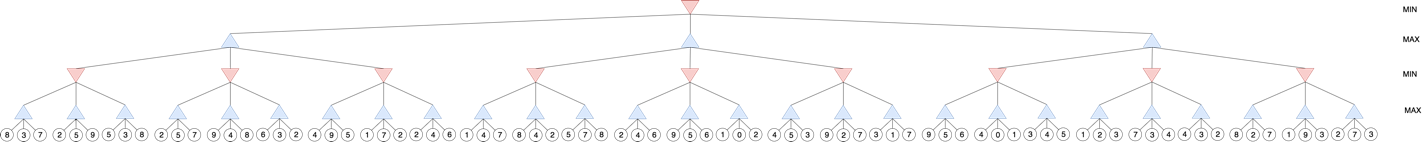
Output Example:

I modeled my program based on tactics that I noticed while I played. The first step was to make my program only lose if it absolutely had to. This can usually beat a bad player. Next, I noticed that better moves were generally when you try to minimize the number of edges of your color per node. This made the AI much more difficult, but I found

**Problem 2.2**

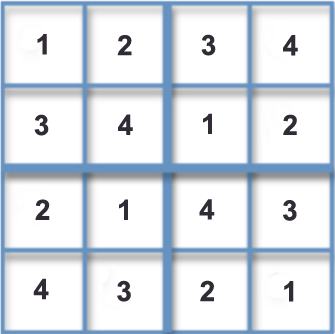
****

1. The overall value of this tree for MAX is 3
2. If the possible moves are examined from left to right, the alpha-beta game search algorithm will have to evaluate 42 of the 81 leaves
3. If the possible moves are examined from right to left, the alpha-beta game search algorithm will have to evaluate 46 of the 81 leaves
4. If the root is now MIN and all other levels are inverted as well.



If the possible moves are examined from left to right, the alpha-beta game search algorithm will have to evaluate 44 of the 81 leaves

**Problem 3**



1. Ensure there is exactly one “3” in the fourth column of the puzzle:

S143 OR S243 OR S343 OR S443

NOT(S143 AND S243)

NOT(S143 AND S343)

NOT(S143 AND S443)

NOT(S243 AND S343)

NOT(S243 AND S443)

NOT(S343 AND S443)

Ensure that S343 is true:

# Starting Constraints

S111 AND S144 AND S242 AND S312 AND S423 AND S441

# Rules to ensure exactly one “3” is in the fourth column

S143 OR S243 OR S343 OR S443

NOT(S143 AND S243) AND NOT(S143 AND S343) AND NOT(S143 AND S443)

NOT(S243 AND S343) AND NOT(S243 AND S443)

NOT(S343 AND S443)

# Rules to ensure that only one number is in each row of the fourth column

S141 OR S142 OR S143 OR S144

NOT(S141 AND S142) AND NOT(S141 AND S143) AND NOT(S141 AND S144)

NOT(S142 AND S143) AND NOT(S142 AND S144)

NOT(S143 AND S144)

S241 OR S242 OR S243 OR S244

NOT(S241 AND S242) AND NOT(S241 AND S243) AND NOT(S241 AND S244)

NOT(S242 AND S243) AND NOT(S242 AND S244)

NOT(S243 AND S244)

S341 OR S342 OR S343 OR S344

NOT(S341 AND S342) AND NOT(S341 AND S343) AND NOT(S341 AND S344)

NOT(S342 AND S343) AND NOT(S342 AND S344)

NOT(S343 AND S344)

S441 OR S442 OR S443 OR S444

NOT(S441 AND S442) AND NOT(S441 AND S443) AND NOT(S441 AND S444)

NOT(S442 AND S443) AND NOT(S442 AND S444)

NOT(S443 AND S444)

Resolution from the rules above:

S144 🡪 NOT(S143) # Based on the rules ensuring only one number in a spot

S242 🡪 NOT(S243) # Based on the rules ensuring only one number in a spot

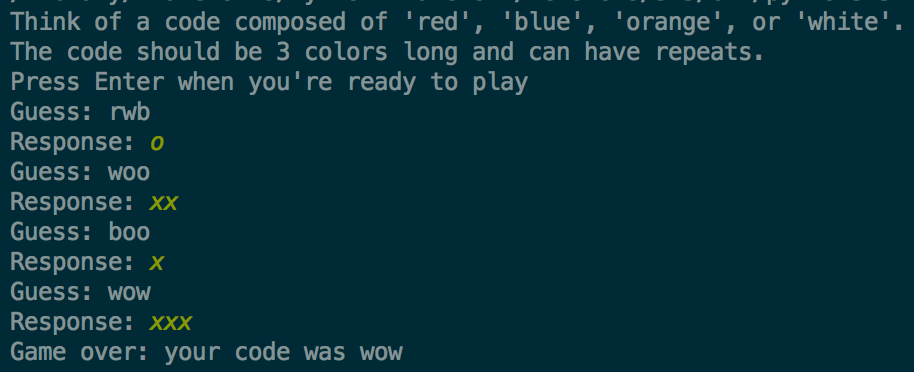
S441 🡪 NOT(S443) # Based on the rules ensuring only one number in a spot

🡪 S343 # Based on the rules ensuring exactly one “3” in the 4th column

**Problem 4**

import sys  
from random import choice  
  
  
class MastermindGame:  
 def \_\_init\_\_(self):  
 *# Doesn't change, therefore it's better to hard code than use resources  
 # and time to calculate it every single time* self.options = [**'rrr'**, **'rrb'**, **'rro'**, **'rrw'**, **'rbr'**, **'rbb'**, **'rbo'**, **'rbw'**, **'ror'**,  
 **'rob'**, **'roo'**, **'row'**, **'rwr'**, **'rwb'**, **'rwo'**, **'rww'**, **'brr'**, **'brb'**,  
 **'bro'**, **'brw'**, **'bbr'**, **'bbb'**, **'bbo'**, **'bbw'**, **'bor'**, **'bob'**, **'boo'**,  
 **'bow'**, **'bwr'**, **'bwb'**, **'bwo'**, **'bww'**, **'orr'**, **'orb'**, **'oro'**, **'orw'**,  
 **'obr'**, **'obb'**, **'obo'**, **'obw'**, **'oor'**, **'oob'**, **'ooo'**, **'oow'**, **'owr'**,  
 **'owb'**, **'owo'**, **'oww'**, **'wrr'**, **'wrb'**, **'wro'**, **'wrw'**, **'wbr'**, **'wbb'**,  
 **'wbo'**, **'wbw'**, **'wor'**, **'wob'**, **'woo'**, **'wow'**, **'wwr'**, **'wwb'**, **'wwo'**,  
 **'www'**]  
  
 *# Engine to run through the game until the code is guessed* def run\_game(self):  
 print(**"Think of a code composed of 'red', 'blue', 'orange', or 'white'."**)  
 print(**"The code should be 3 colors long and can have repeats."**)  
 input(**"Press Enter when you're ready to play"**)  
  
 game\_over = False  
  
 while not game\_over:  
 guess = self.guess\_code()  
  
 response = self.get\_user\_response().lower()  
  
 if response == **"xxx"**:  
 game\_over = True  
 print(**"Game over: your code was "** + guess)  
 else:  
 self.options.remove(guess)  
 self.update\_options(guess, response)  
  
 *# Randomly guess code from the remaining options* def guess\_code(self):  
 if len(self.options) == 0:  
 print(**"Game over: No valid guesses, maybe you gave me a wrong response?"**)  
 sys.exit()  
 else:  
 guess = choice(self.options)  
 print(**"Guess: "** + guess)  
 return guess  
  
 *# Query the user for their response to the AI's guess  
 # Xs should come before Os* def get\_user\_response(self):  
 response = input(**"Response: "**).lower()  
 if response != **""** and **'x'** not in response and **'o'** not in response:  
 print(**"Invalid response, try again"**)  
 self.get\_user\_response()  
 return response  
  
 *# Return a response given a guess and a code* def get\_response(self, code, guess):  
 response = **""** code\_incorrect = **""** guess\_incorrect = **""** for i in range(0, 3): *# Add Xs* if code[i] == guess[i]:  
 response += **"x"** else:  
 code\_incorrect += code[i]  
 guess\_incorrect += guess[i]  
  
 for color in guess\_incorrect: *# Add Os* if color in code\_incorrect:  
 response += **"o"** code\_incorrect = code\_incorrect.replace(  
 code\_incorrect[code\_incorrect.index(color)], **""**)  
  
 return response  
  
 *# Remove options that would not return the given response given a guess* def update\_options(self, guess, response):  
 self.options = [code for code in self.options  
 if self.get\_response(code, guess) == response]  
  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 new\_game = MastermindGame()  
 new\_game.run\_game()

Output Example:



This strategy could technically work for a large-scale Mastermind game, but it would take drastically more computing power, in which case, it would probably be more optimal to use another strategy. With 30 colors and 25 positions, there would be 3025 (which is 8.4728861 x 1036) “open” possibilities at the beginning of the game. This is way too many to consider this strategy practical.