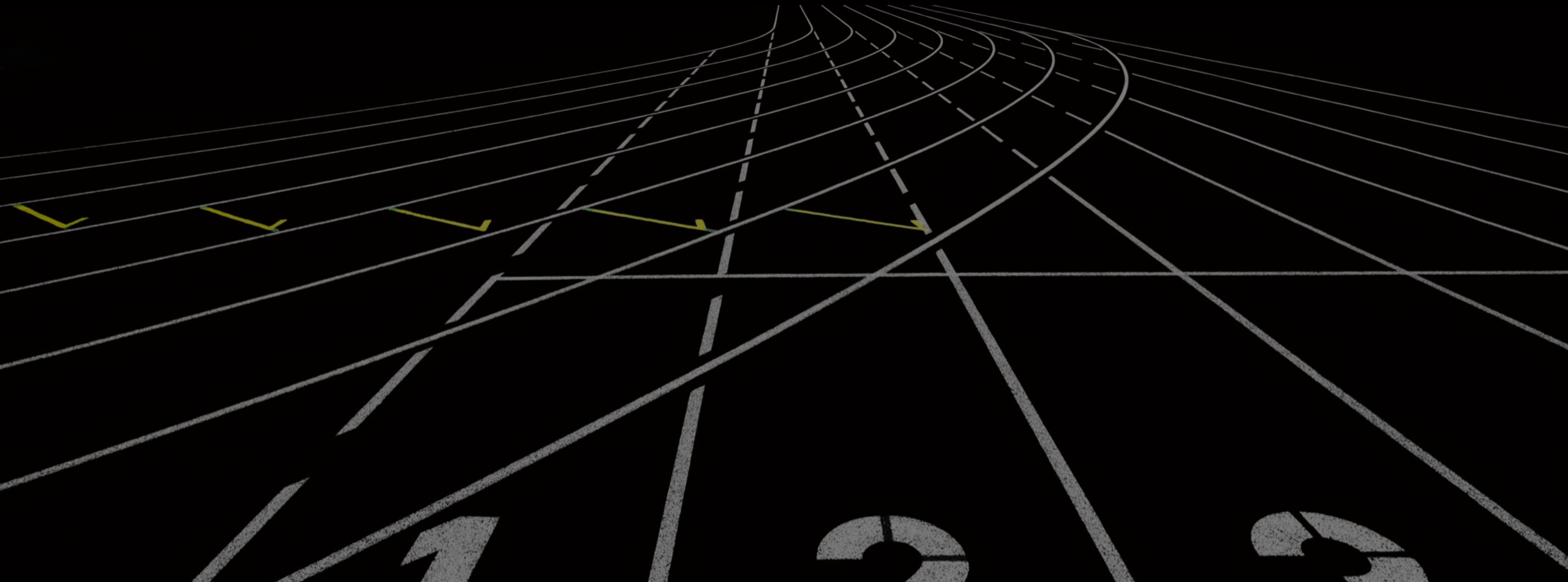
CMSE 202 Final Presentation

Team Catan

CMSE 202 section 2

Aman, Kollin, Mark, Phillip, Trevor



Introduction

- The Game of Catan
  - Strategy Based Game
  - Multiplayer
  - Some actions are
    - Build roads
    - Build settlements
    - Build cities
    - Buying development cards
  - 10 Victory Points to win

Credit: Victoria Dye

What is Genetic Programing

- Inspired from nature
- Population of bots
- Evolving random mutations
- Best selections

Credit: Ann Johnson Prum

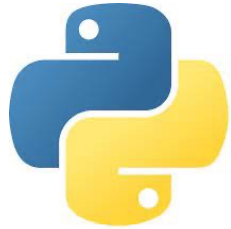
Credit: https://www.genetic-programming.com/

Tools that are used!

- Catanatron -@bcollazo
- Python and Python Notebook
- Matplotlib
- Numpy

Tools that we created:

- Genetic Program Mutator



Source: Respective Packages

What is has been implemented?

Player	% of wins in 1v1 games	num games used for result
AlphaBeta(n=2)	80% vs ValueFunction	25
ValueFunction	90% vs GreedyPlayouts(n=25)	25
GreedyPlayouts(n=25)	100% vs MCTS(n=100)	25
MCTS(n=100)	60% vs WeightedRandom	15
WeightedRandom	53% vs WeightedRandom	1000
VictoryPoint	60% vs Random	1000
Random	-	-

Current bot implementations by Catanatron

How our approach is different?

- Stages
- Using Genetic Programing for optimization

```
WEIGHTS_BY_ACTION_TYPE = {
    ActionType.BUILD_CITY: 10000,
    ActionType.BUILD_SETTLEMENT: 1000,
    ActionType.BUY_DEVELOPMENT_CARD: 100,
}
```

Weights of *Weighted Random* from Catanatron

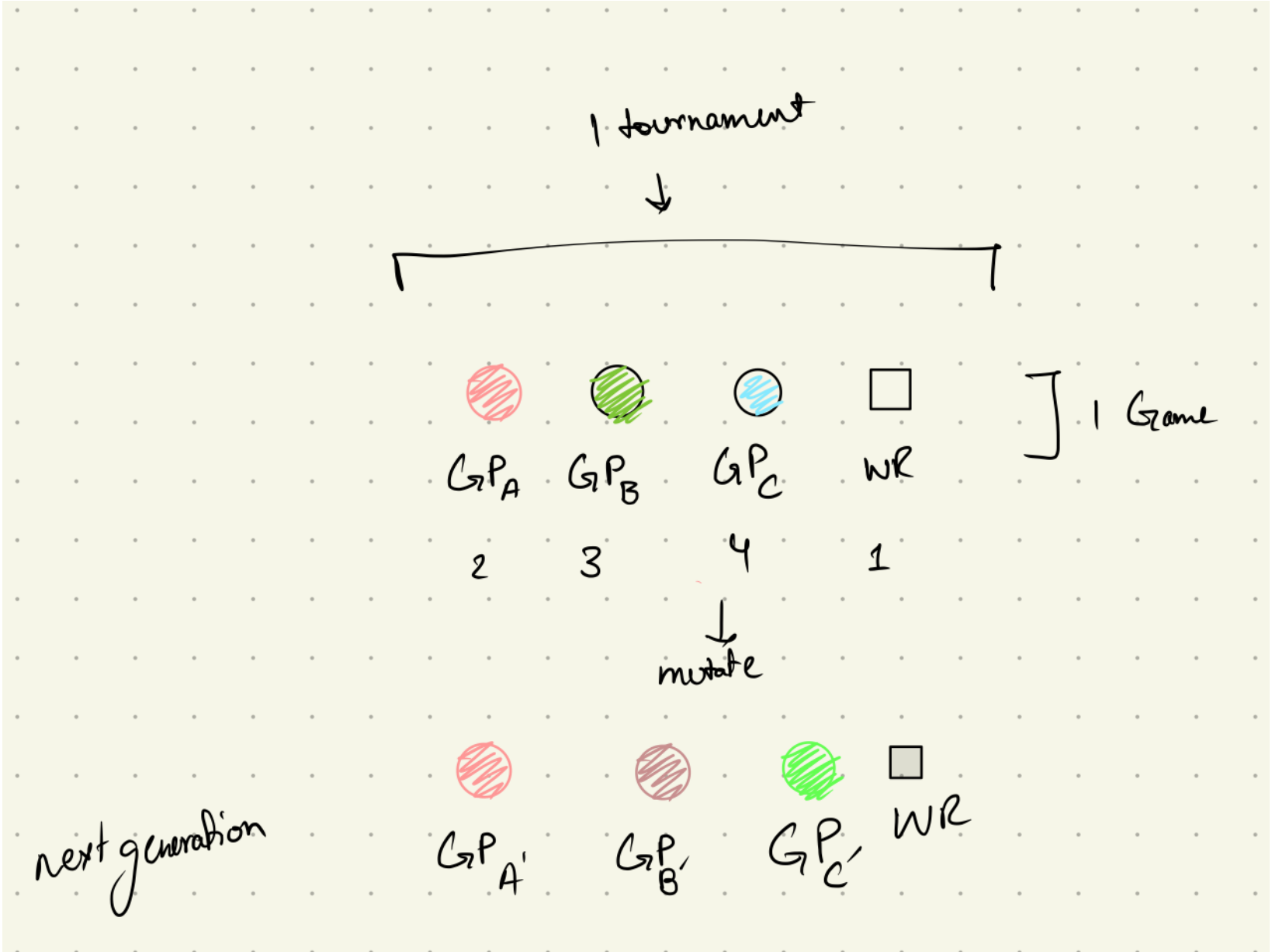
The Goal

Questions that we are trying to answer

- Find optimal weights for actions
- Which actions are good/bad
- When to take such actions in the game

Source: iStock

Approach #1

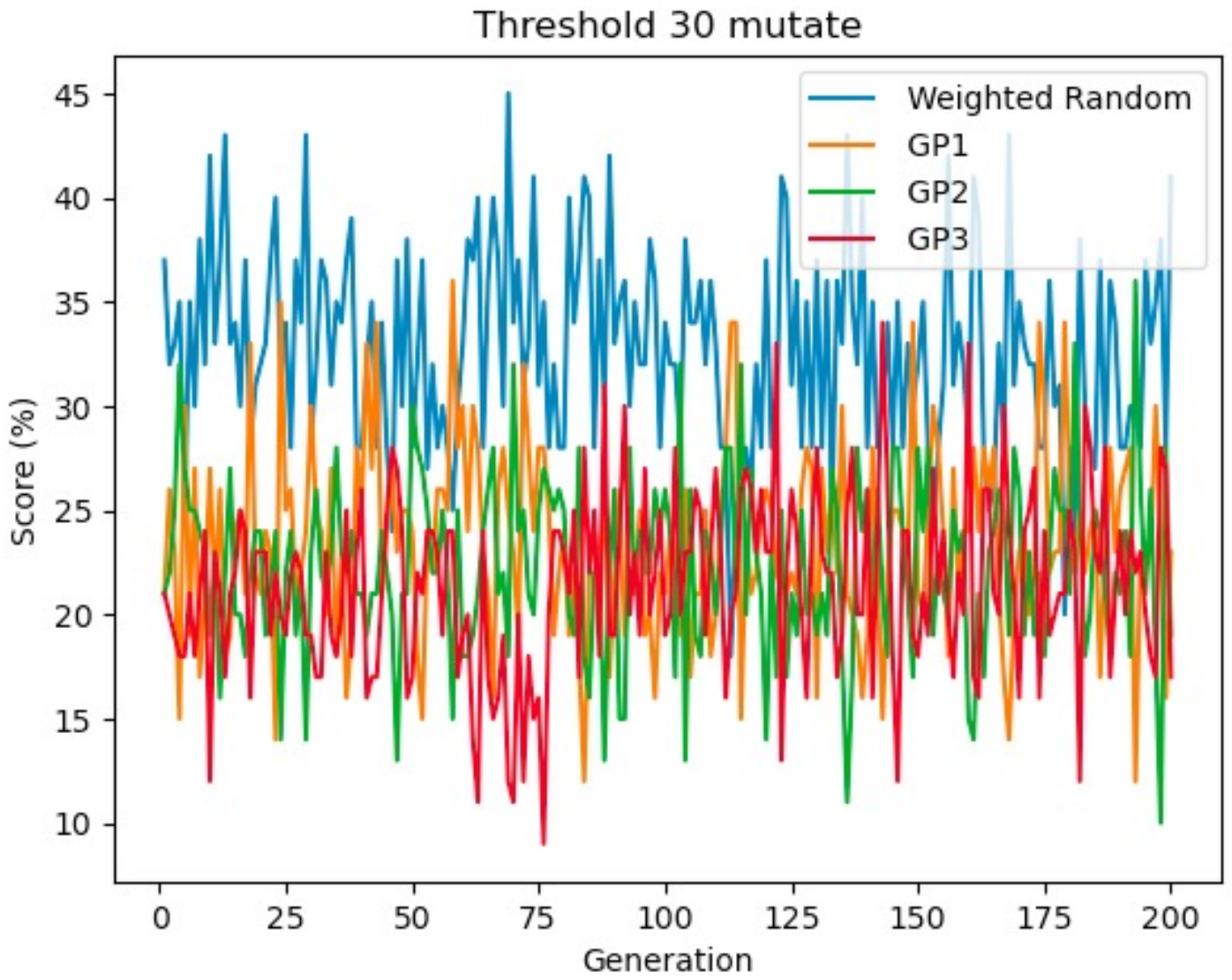

1 tournament
GP_A
GP_B
GP_C
WR
1 Game
2
3
4
1
mutate
next generation
GP_A'
GP_B'
GP_C'
WR

Approach #1 a

- Mutate just the bad population

```
for key in c:
    if c[key] < 0.3 and key != Color.RED and key:
        try:
            players[playerDict[key]].mutate()
        except:
            print("Error: ", key)
```

Approach #1
a


Threshold 30 mutate
Weighted Random
GP1
GP2
GP3
Score (%)
Generation
45
40
35
30
25
20
15
10
0
25
50
75
100
125
150
175
200

Approach #1 a

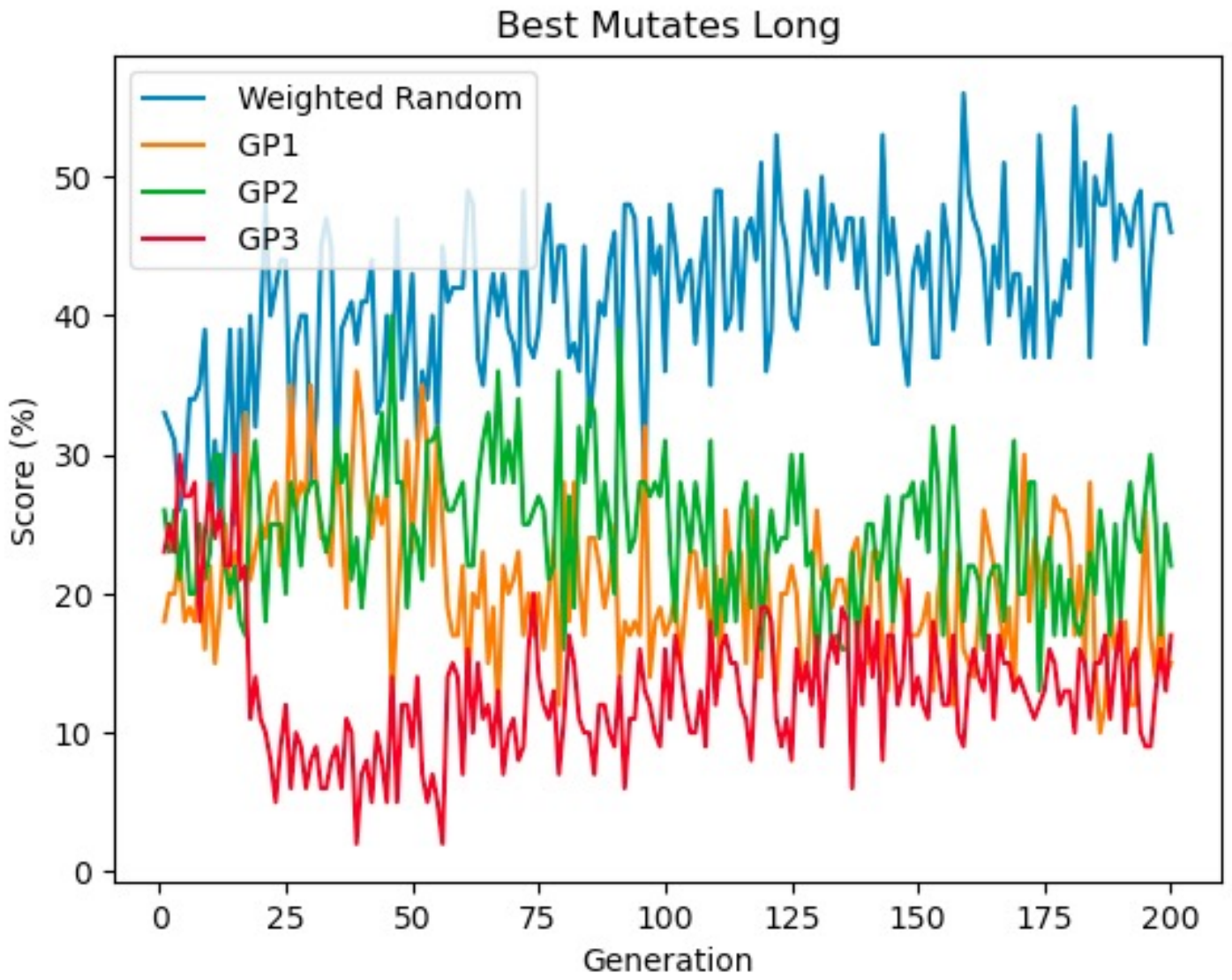
- Problems with this approach!
  - Often all players get mutated
  - No survival pressure
  - Was not giving results better than
  - Mutating Low Quality Individuals

Approach #1 b

- Applying selection pressure

```
# mutate the best player
extend = []
for i in c:
    if i == Color.RED or not i:
        continue
    extend = extend + [i] * c[i]

# make a random pick from the list extend
pick = random.choice(extend)
players[playerDict[pick]].mutate()
```

Approach #1 b

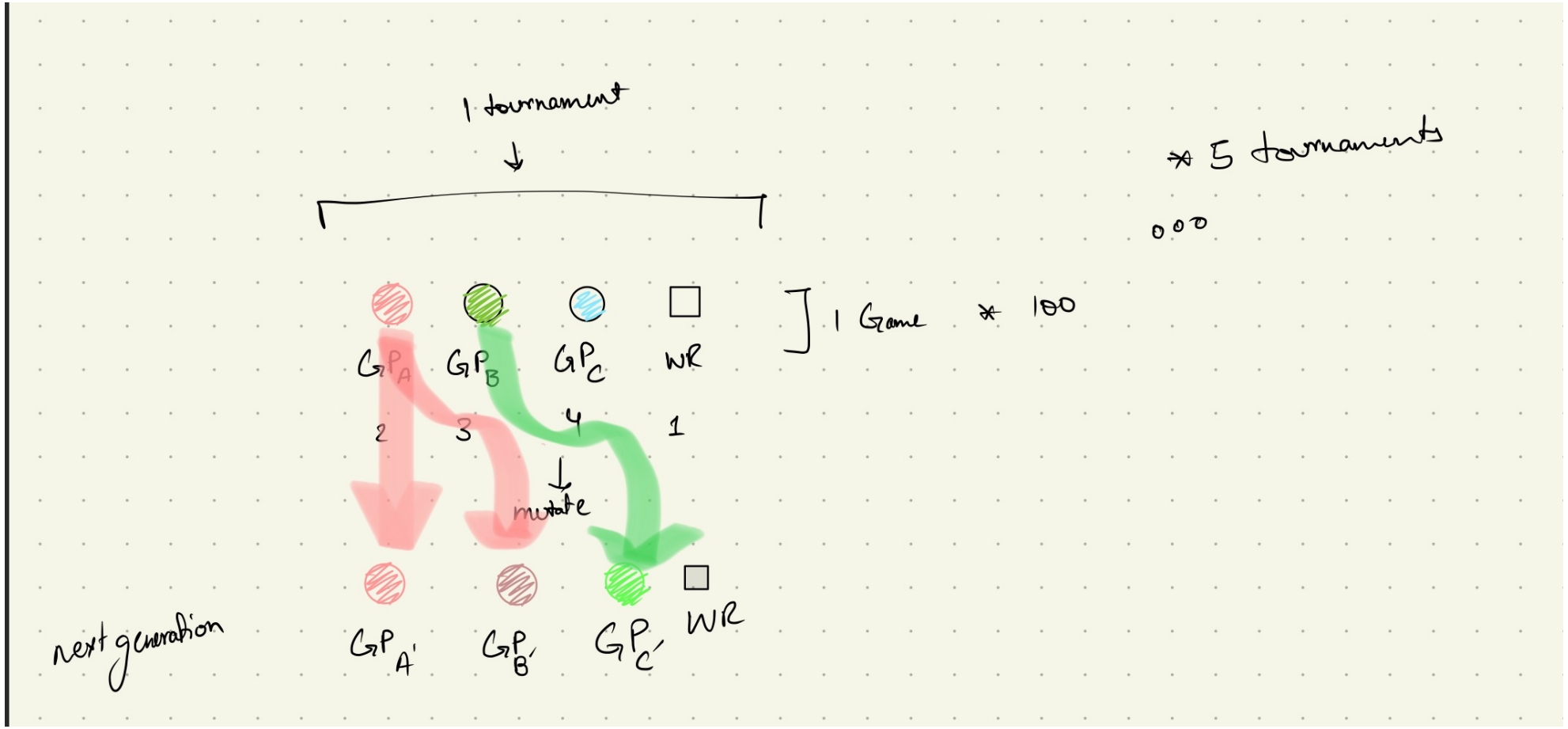

Best Mutates Long
Weighted Random
GP1
GP2
GP3
Score (%)
0
10
20
30
40
50
0
25
50
75
100
125
150
175
200
Generation

Approach #1 b

Problems with this approach

- Only mutation the best
- The rest of the population still continues without change

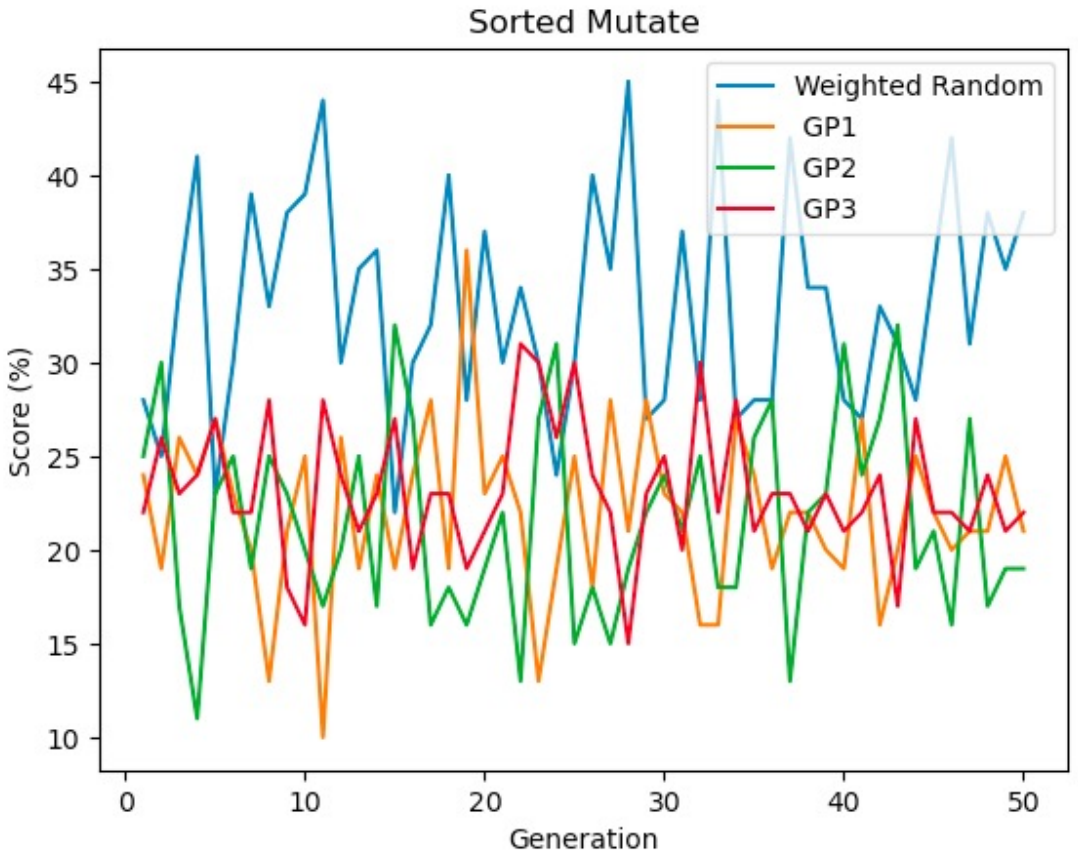
Approach #1 c

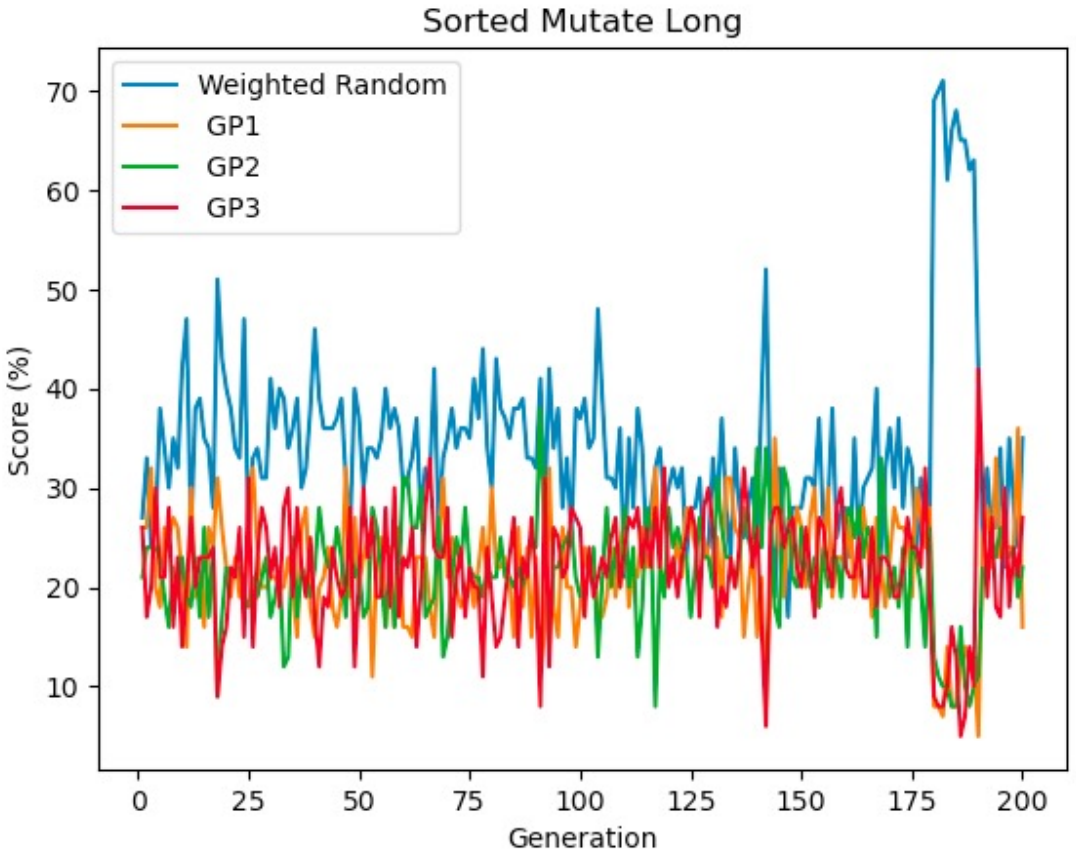

1 tournament
* 5 tournaments
000
GP_A
GP_B
GP_C
WR
] 1 Game * 100
2
3
4
1
mutate
next generation
GP_A'
GP_B'
GP_C'
WR

Approach #1 c

```
# mutate the best player
temp = c.pop(Color.RED, None)
temp = c.pop(None, None)
st = sorted(c.items(), key=lambda x: x[1], reverse=True)

try:
    players[1].setData(players[playerDict[st[0][0]]].copyData())
    players[playerDict[st[0][0]]].mutate()
    players[2].setData(players[playerDict[st[0][0]]].copyData())
    players[playerDict[st[1][0]]].mutate()
    players[3].setData(players[playerDict[st[1][0]]].copyData())
except:
    print('An exception occurred')
    logging.warning('An exception occurred')
```

Approach #1 c


Sorted Mutate
Weighted Random
GP1
GP2
GP3
Score (%)
Generation
10
15
20
25
30
35
40
45
0
10
20
30
40
50


Sorted Mutate Long
Weighted Random
GP1
GP2
GP3
Score (%)
Generation
10
20
30
40
50
60
70
0
25
50
75
100
125
150
175
200

Limitations and Problems

- Player Size Limitation 4 bots
- Search space is too large
- No Cross Overs

Approach #2

- Create Generations of 100 Bots
- Future Generations have the weights of "good" bots in previous Generation / are "offspring" of good bots

```
def create_new_player(weights1,weights2):
    player = GP(color=Color.RED)
    weights = []
    for i in range(12):
        if random.randint(0,1) == 0:
            weights.append(weights1[i])
        else:
            weights.append(weights2[i])
    player.set_weights(weights)

    return player
```

Creating First Generation

```
class GP(Player):

    def __init__(self,color):
        super().__init__(color)
        self.INITIAL_WEIGHTS_BY_ACTION_TYPE = {
            ActionType.BUILD_CITY: random.randint(100,10000),
            ActionType.BUILD_SETTLEMENT: random.randint(100,10000),
            ActionType.BUY_DEVELOPMENT_CARD: random.randint(100,10000),
            ActionType.BUILD_ROAD: random.randint(100,10000)
        }

        self.MID_WEIGHTS_BY_ACTION_TYPE = {
            ActionType.BUILD_CITY: random.randint(100,10000),
            ActionType.BUILD_SETTLEMENT: random.randint(100,10000),
            ActionType.BUY_DEVELOPMENT_CARD: random.randint(100,10000),
            ActionType.BUILD_ROAD: random.randint(100,10000)
        }


        self.LATE_WEIGHTS_BY_ACTION_TYPE = {
            ActionType.BUILD_CITY: random.randint(100,10000),
            ActionType.BUILD_SETTLEMENT: random.randint(100,10000),
            ActionType.BUY_DEVELOPMENT_CARD: random.randint(100,10000),
            ActionType.BUILD_ROAD: random.randint(100,10000)
        }
```

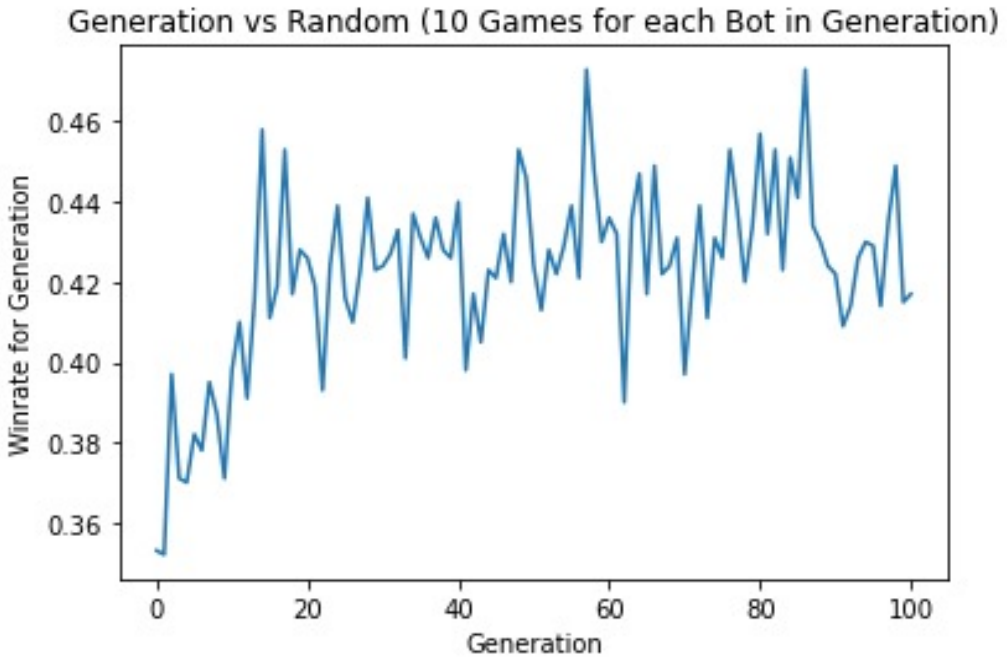
Creating The Next Generation

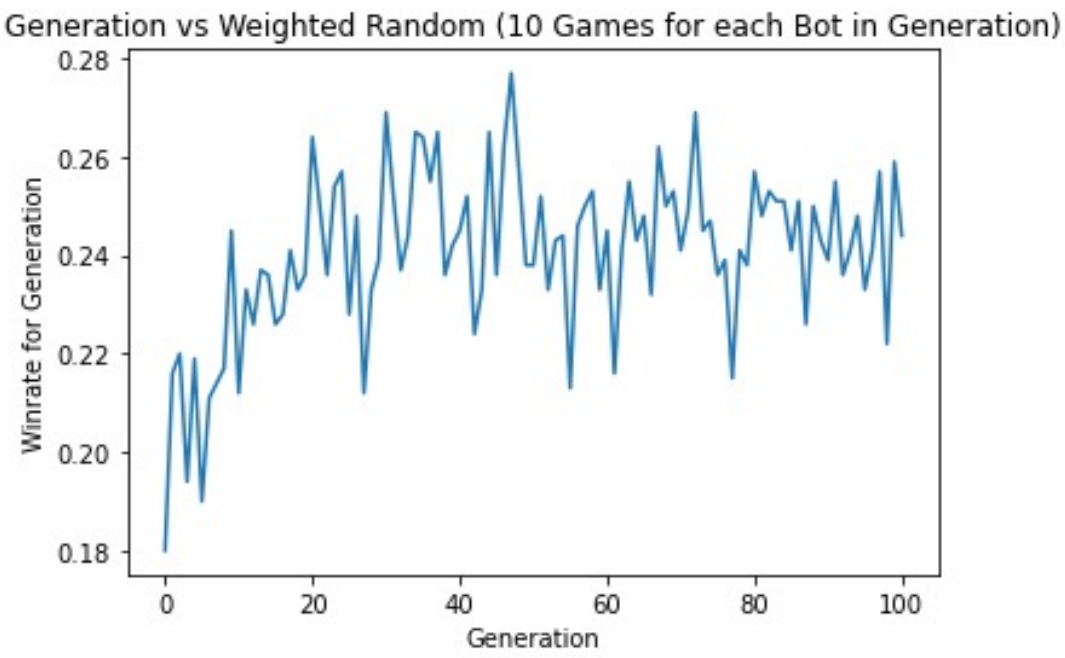
```
for x in range(100):
    randomWeights1 = winning_weights[random.randint(0,len(winning_weights)-1)]
    randomWeights2 = winning_weights[random.randint(0,len(winning_weights)-1)]
    gen2["{0}".format(x)] = create_new_player(winning_weights[randomWeights1],winning_weights[randomWeights2])
```

```
def create_new_player(weights1,weights2):
    player = GP(color=Color.RED)
    weights = []
    for i in range(12):
        if random.randint(0,1) == 0:
            weights.append(weights1[i])
        else:
            weights.append(weights2[i])
    player.set_weights(weights)

    return player
```

Results of 100 Generations:

- Weights Converged to Single Weight
- [6217, 9865, 2767, 111, 7019, 8607, 1596, 393, 9135, 3414, 7137, 219]





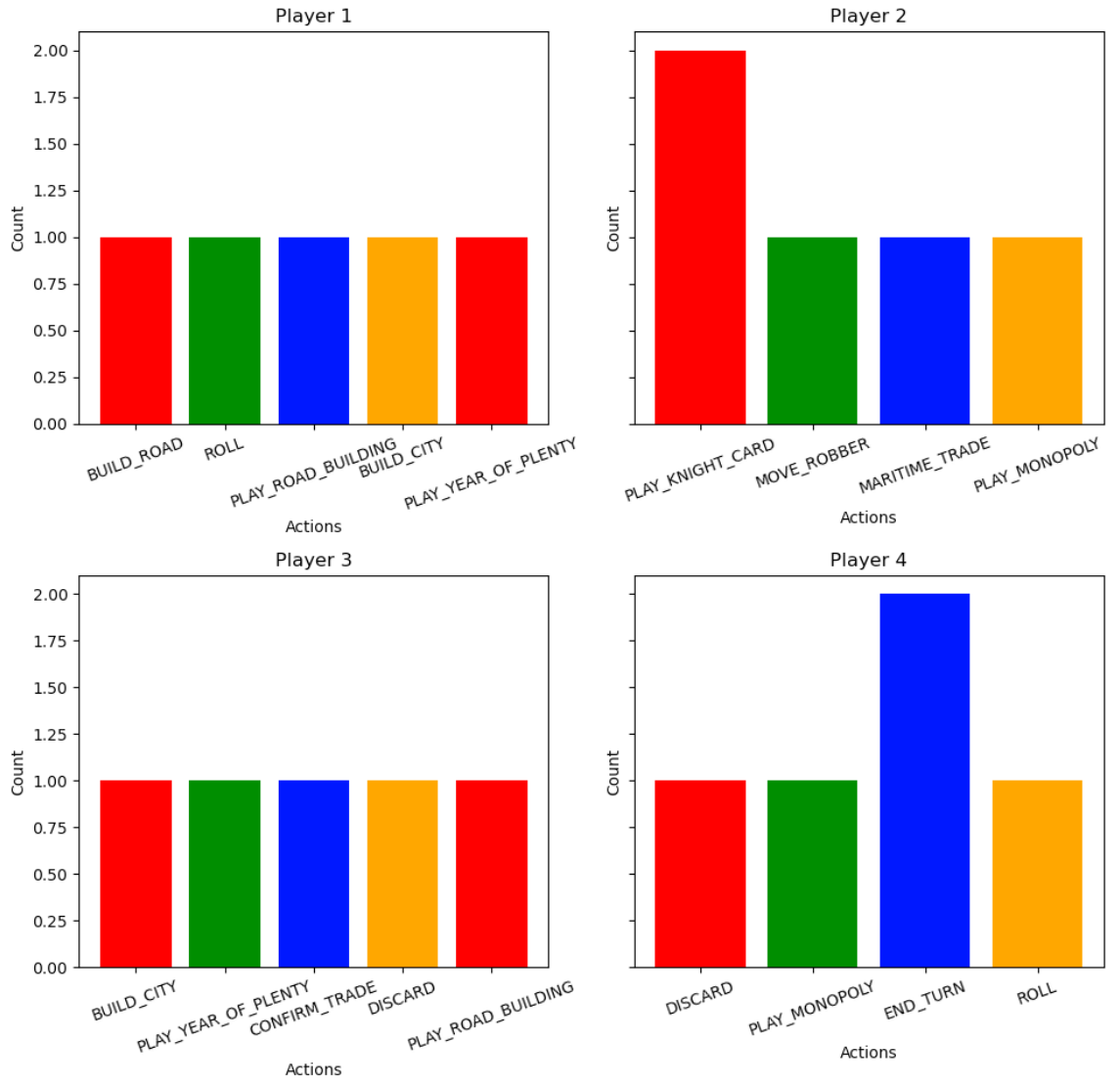
Analysis

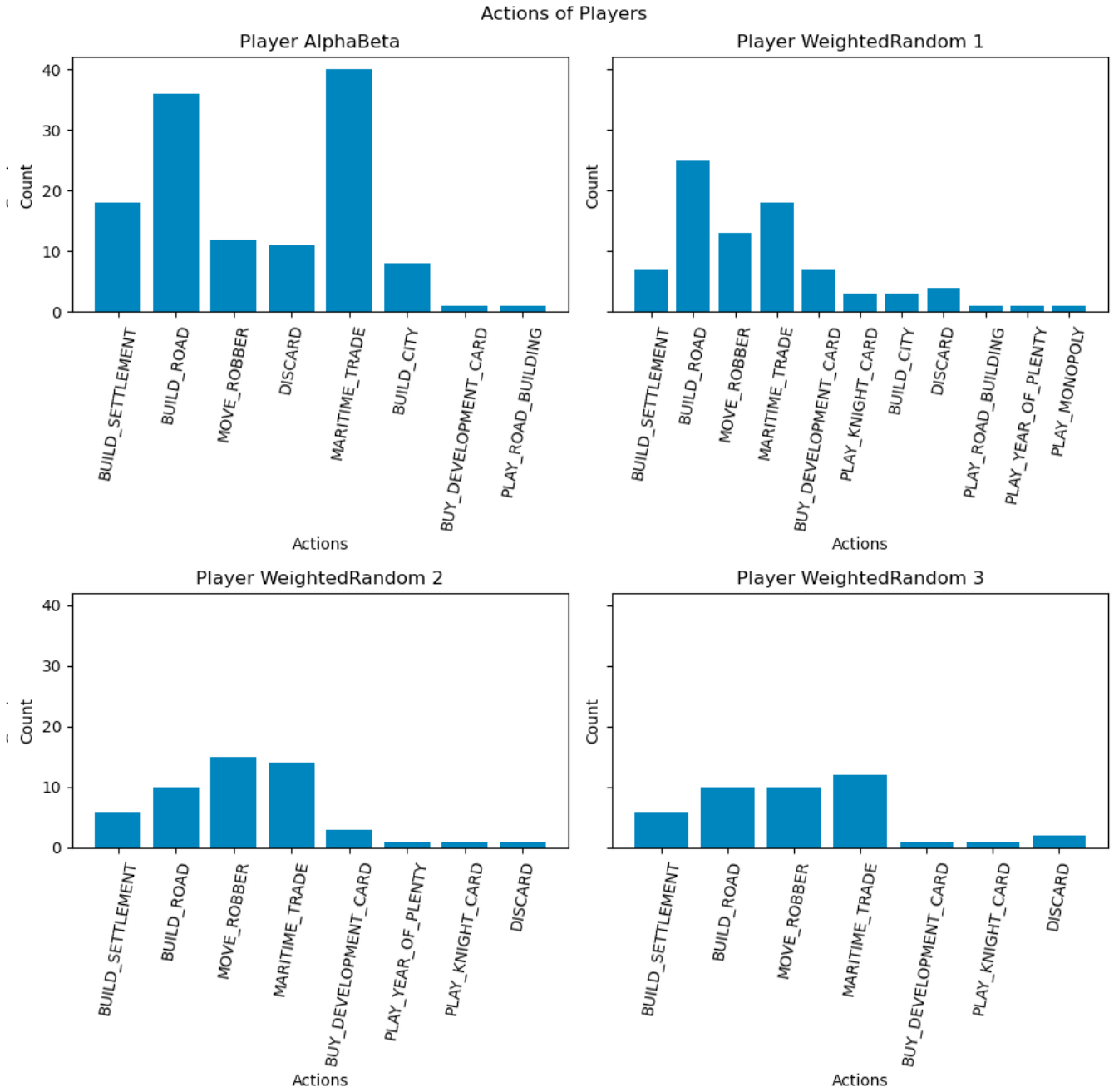
- Used Concepts from class
  - Matplotlib
  - Classes
- We created an object class that would run through data and give us visualizations of our players and their actions.

```
]:
import time
import matplotlib.pyplot as plt
import numpy as np


class Analysis:
    def __init__(self, action_map):
        self.game = None
        self.actions = []
        self.action_map = action_map

    def set_game(self, game_number):
        self.game = game_number

    def record_action(self, action_name, id):
        if id not in range(1, 5):
            print('Error: ID must be between 1 and 4.')
            return

        if action_name not in self.action_map:
            print(f'Error: {action_name} is not a valid action.')
            return

        timestamp = time.time()
        action_id = self.action_map[action_name]
        self.actions.append((action_id, id, timestamp))

    def generate_plot(self):
        actions_count = {}
        for action_id, id, timestamp in self.actions:
            if id in actions_count:
                actions_count[id] += 1
            else:
                actions_count[id] = 1

        x = list(actions_count.keys())
        y = [actions_count[i] for i in x]


        fig, axs = plt.subplots(2, 2, figsize=(10, 10))
        colors = plt.cm.cool(np.linspace(0, 1, len(x)))
        axs[0, 0].set_title('Player 1')
        axs[0, 0].bar(x,y)
        axs[0, 1].set_title('Player 2')
        axs[0, 1].bar(x,y)
        axs[1, 1].bar(x, y)
        axs[1, 0].bar(x, y)
        axs[1, 0].set_title('Player 3')
        axs[1, 1].set_title('Player 4')

        fig.text(0.5, 0.04, 'Action Type', ha='center')
        fig.text(0.04, 0.5, 'Action Count', va='center', rotation='vertical')

        plt.show()
```

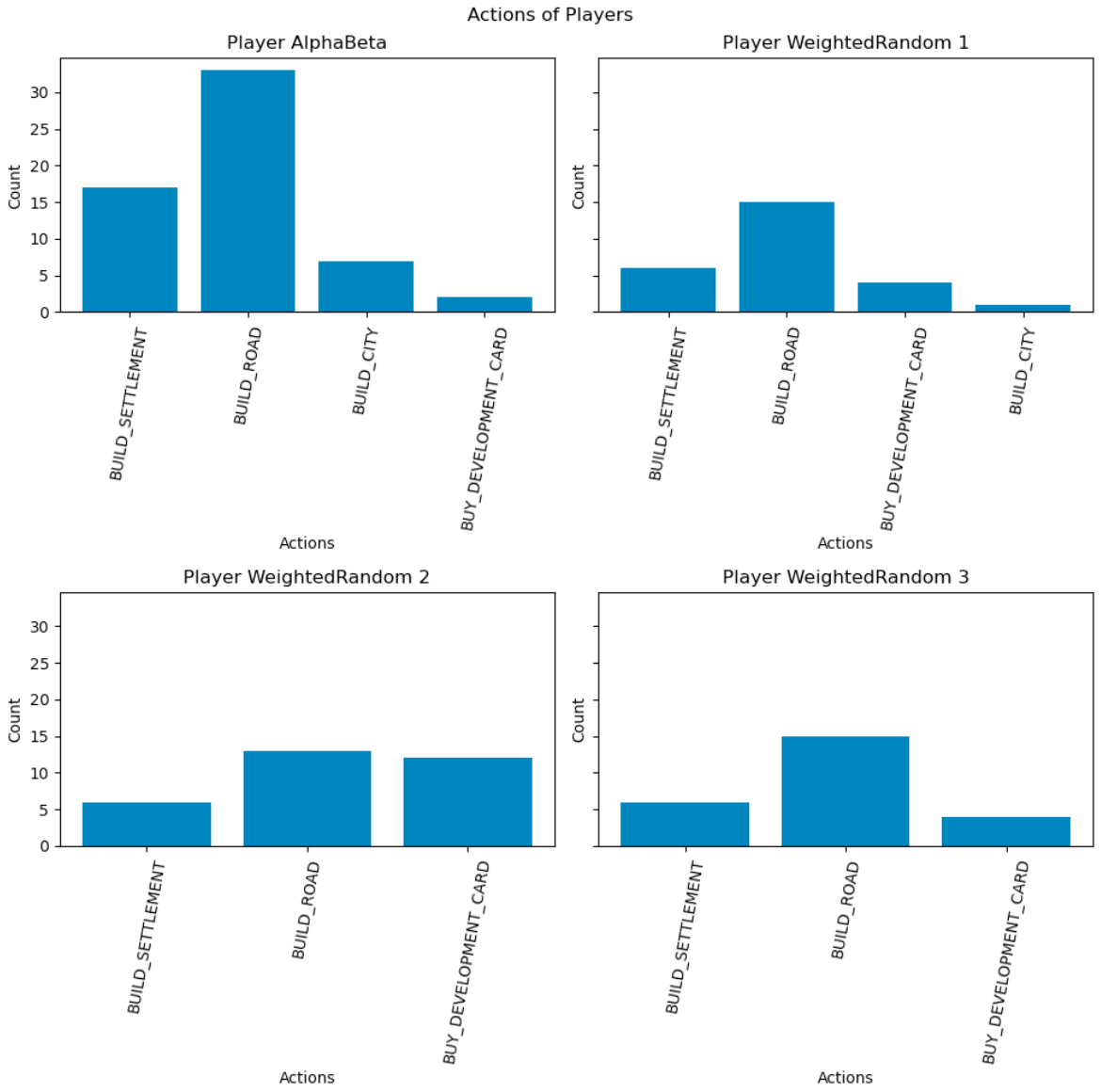
Analysis Testing

- Before we used the class, we wanted to make sure it ran properly.
- This is a hand created practice to prep for real data.



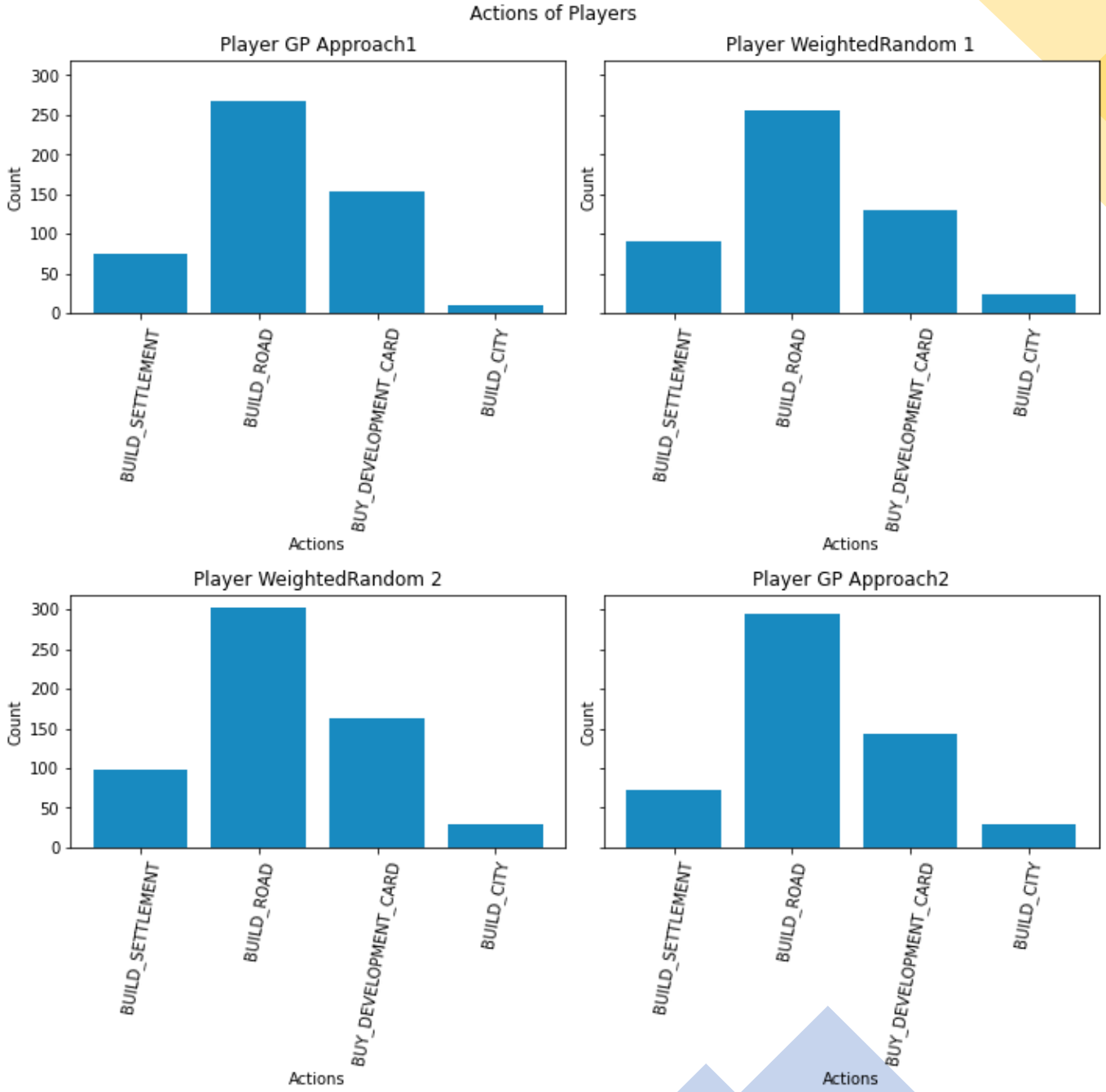

Actions of Players
Player AlphaBeta
Player WeightedRandom 1
Player WeightedRandom 2
Player WeightedRandom 3
Count
Actions
BUILD_SETTLEMENT
BUILD_ROAD
MOVE_ROBBER
DISCARD
MARITIME_TRADE
BUILD_CITY
BUY_DEVELOPMENT_CARD
PLAY_ROAD_BUILDING
PLAY_KNIGHT_CARD
PLAY_YEAR_OF_PLENTY
PLAY_MONOPOLY

AlphaBeta: 100



Winrate

WeightedRandom 2 : 30%
GP Approach1 : 16%
GP Approach2 : 23%
WeightedRandom 1 : 30%



How do our bots compare?

- WeightedRandom was better than Bot #1
- WeightedRandom was equal to Bot #2
- Bot #2 was better than Bot #1

```
WeightedRandom 1 : 57
GP Approach1 : 40
```

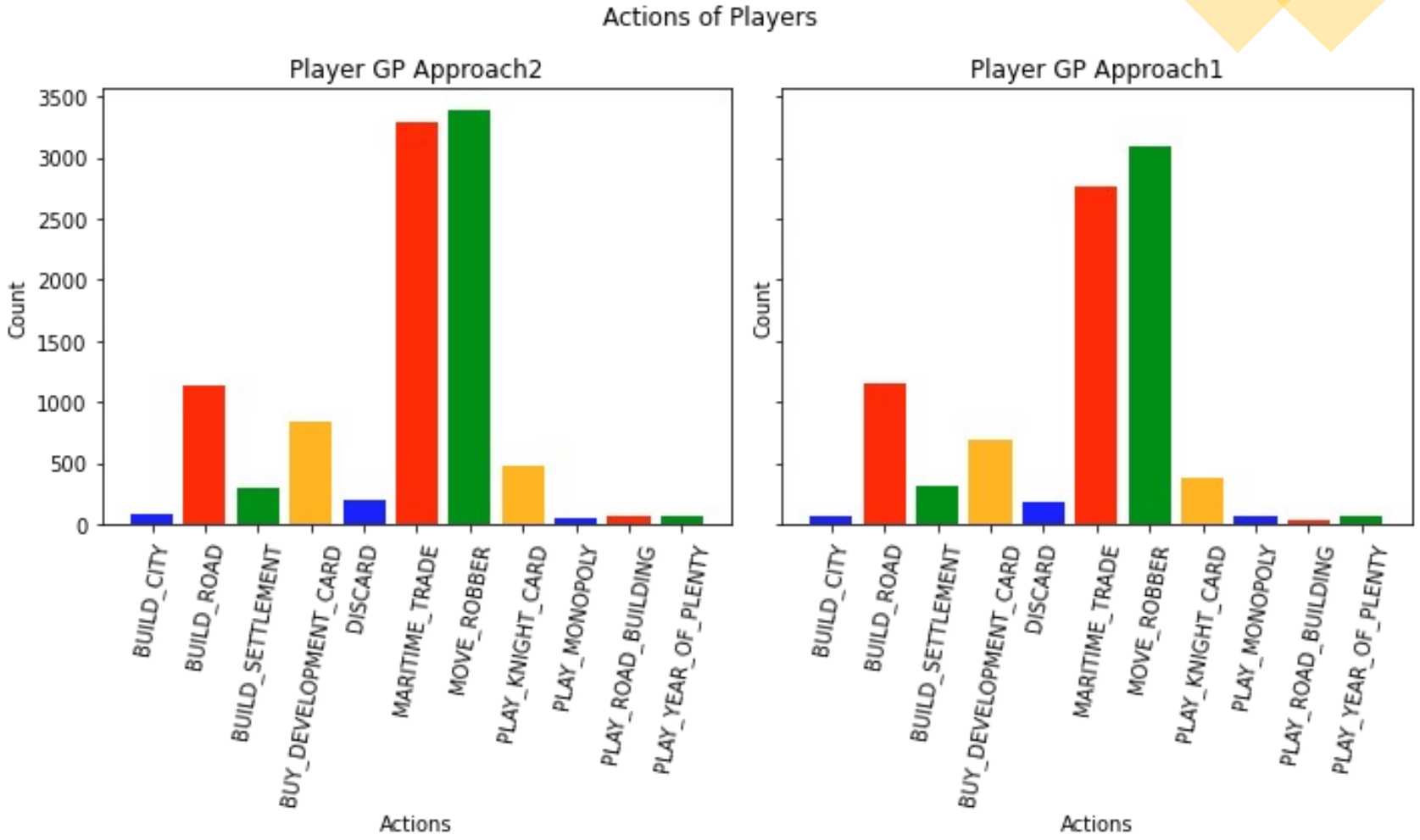
```
Weighted Random 1 : 48
GP Approach2 : 52
```

```
GP Approach1 : 41
GP Approach2 : 58
```

Over 200 Games

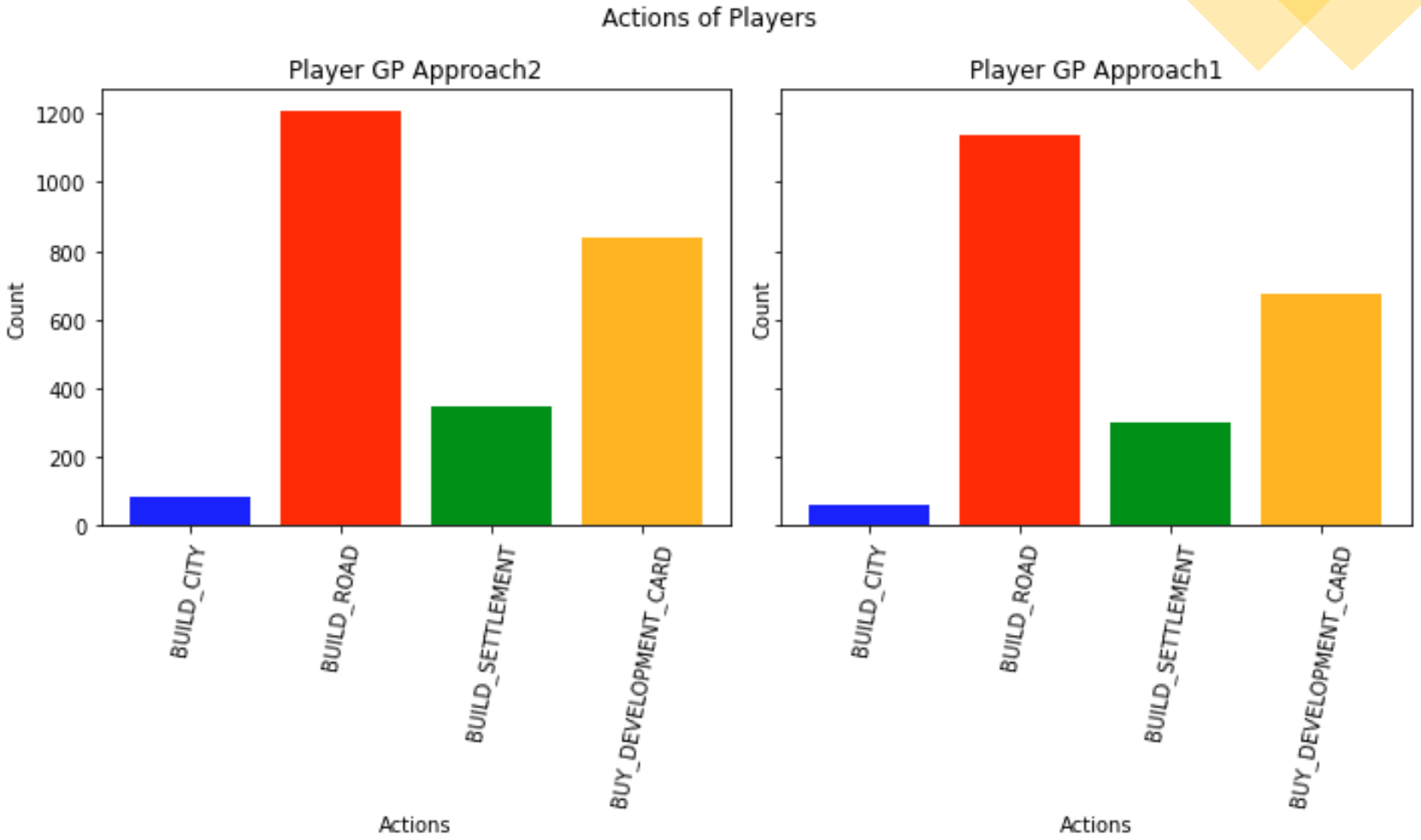
Winrate

GP Approach1 : 40%
GP Approach2 : 60%



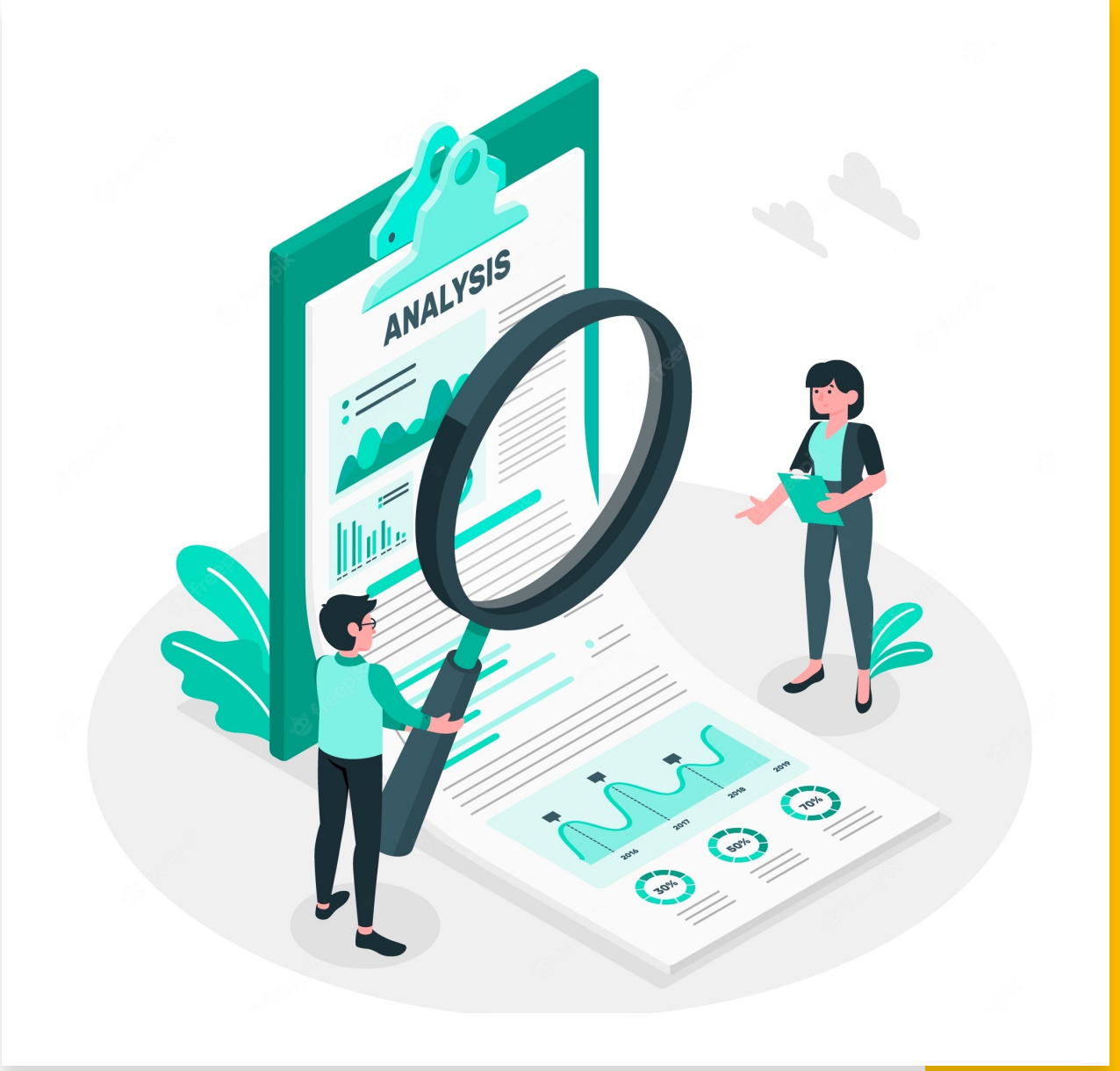
Winrate

GP Approach1 : 40%
GP Approach2 : 60%



Analysis of actions

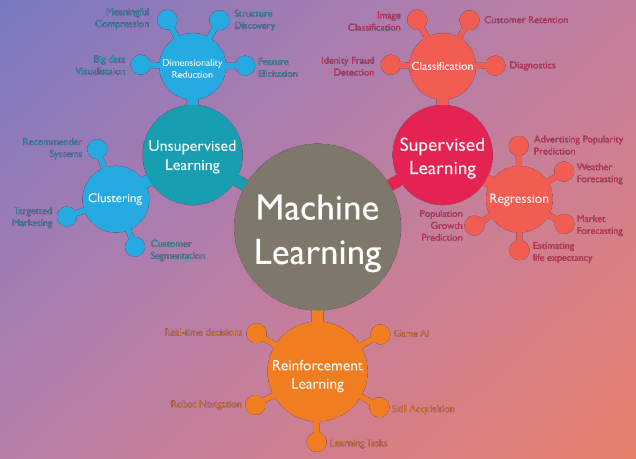
- Analysis of our actions allows us to determine the differences in our created methods against the weighted randoms.
- This also allows for us to visualize our findings for an easier understanding.



Src: Powerpoint Designer

Relation to 202 Content

- Documentation
- Classes
  - Inheritance
- Classification
  - Machine Learning
- Project Management
  - GitHub
  - Pair Programing
  - Testing



Src: Powerpoint Designer

How to continue our work?

IMPROVEMENTS ON SPEED

FACTOR IN OTHER ASPECTS

Src: Powerpoint Designer

Src: Powerpoint Designer

Conclusion

- Adding early, middle, late game values didn't improve the bot
- Our weights were completely different
- There are many ways to correctly play the game well
- Too much random for a more meaningful impact in the game

Acknowledgments

- Dr. Tom Finzell
- Sandeep Vemulapalli
- Heidi Olsen
- Nathan Haut