AI Investigation: Using Tianji Horse Racing Game to test strategy for simple deterministic games

AI Investigation Part 1

Goal: test basic strategy to determine dominant option (win data only, win data + loss data)

Also preliminary exploration for multiple-dominant-strategy games

Creator: `Lang (Ron) Chen 2022`

In [1]:
```
import random
import math as m
```

Define Game

A game where all three of our horses are of lower power (definitely will lose) than the matching ranked enemy horse. However we can win by doing (3, 1, 2).

An alternative game is set up where there are more than one way to win (3, 2, 1) (3, 1, 2). This is used as a preliminary exploration for games with multiple dominant strategies.

In [2]:
```
# Game data for standard Tianji game
NPC_ability = (3.5, 2.5, 1.5)
our_ability = (3, 2, 1)
```

In [3]:
```
# # Game data for nonstandard Tianji game - multiple ways to win
# NPC_ability = (3.5, 2, 1.5)
# our_ability = (3, 2.5, 1)
```

In [4]:
```
# Our Options
choice = (1, 2, 3)
```

In [5]:
```
def winloss(seq):
    wins = 0
    for i in range(len(seq)):
        if our_ability[seq[i]-1] > NPC_ability[i]:
            wins += 1
    if wins >= 2:
        return 1
    return 0
```

In [6]:
```
def validation_TianJi(final_choice, choice):
    if len(set(final_choice)) != len(choice):
        return False
    for i in range(len(final_choice)):
        if final_choice[i] not in choice:
            return False
    return True
```

Simulation

In [7]:

```
RUNS = 100000
sample = list()
victory = list()
for i in range(RUNS):
    obs = random.sample(choice, 3)
    sample.append(obs)
    victory.append(winloss(obs))
```

In [8]:

```
print(f'Wins: {sum(victory)}')
print(f'Win rate: {sum(victory)/RUNS}')
```

```
Wins: 16397
Win rate: 0.16397
```

Manipulation (Win data only) *i.e. proportions*

In [9]:

```
# Preprocessing the data
winindex_algo1 = list()
for i in range(10000):
    if victory[i]:
        winindex_algo1.append(i)

winsamples_algo1 = list()
for i in range(len(winindex_algo1)):
    winsamples_algo1.append(sample[winindex_algo1[i]])
```

Algorithm 1 (Win data only) *i.e. proportions*

Algorithm explanation: counting up the number of appearences of each of 0, 1, 2 for each position of the winning games. And then picking the max to fill that position

In [10]:

```
# Core Algorithm

final_choice_algo1 = [-1, -1, -1]

viewing_details_tally_algo1 = list()
viewing_details_tmp_algo1 = list()

for i in range(len(choice)):
    tally = {1:0, 2:0, 3:0}
    for j in range(len(winsamples_algo1)):
        tally[winsamples_algo1[j][i]] += 1
    tmp = list(tally.items())
    tmp.sort(key = lambda x:x[1], reverse = True)
    final_choice_algo1[i] = tmp[0][0]

    viewing_details_tmp_algo1.append(tmp)
    viewing_details_tally_algo1.append(tally)

final_choice_algo1
```

Out[10]:

```
[3, 1, 2]
```

The algorithm successfully returned the only solution: [3. 1, 2]

Validation

In [11]:
```
validation_TianJi(final_choice_algo1, choice)
```
Out[11]:
```
True
```
Emperical Testing

In [12]:
```
victory_algo1 = list()
for i in range(RUNS):
    victory_algo1.append(winloss(final_choice_algo1))
```
In [13]:
```
print(f'wins: {sum(victory_algo1)}')
print(f'win rate: {sum(victory_algo1)/RUNS}')
```
```
wins: 100000
win rate: 1.0
```
Viewing Details

In [14]:
```
viewing_details_tally_algo1
```
Out[14]:
```
[{1: 0, 2: 0, 3: 1606}, {1: 1606, 2: 0, 3: 0}, {1: 0, 2: 1606, 3: 0}]
```
In [15]:
```
viewing_details_tmp_algo1
```
Out[15]:
```
[[(3, 1606), (1, 0), (2, 0)],
 [(1, 1606), (2, 0), (3, 0)],
 [(2, 1606), (1, 0), (3, 0)]]
```

Manipulation (Use both win and lose data) *i.e. mean*

In [16]:
```
# Preprocessing the data
winsamples_algo2 = list()
losesamples_algo2 = list()
for i in range(len(sample)):
    if victory[i]:
        winsamples_algo2.append(sample[i])
    else:
        losesamples_algo2.append(sample[i])
```
Algorithm 2 (Use both win and lose data) *i.e. mean*

In [17]:
```
# Core Algorithm
final_choice_algo2 = [-1, -1, -1]

viewing_details_tally_algo2 = list()
viewing_details_tmp_algo2 = list()

for i in range(len(choice)):
    tally = {1:0, 2:0, 3:0}
    for j in range(len(winsamples_algo2)):
        tally[winsamples_algo2[j][i]] += 1
    for j in range(len(losesamples_algo2)):
        tally[losesamples_algo2[j][i]] -= 1
    tmp = list(tally.items())
    tmp.sort(key = lambda x:x[1], reverse = True)
    final_choice_algo2[i] = tmp[0][0]
```

```
    viewing_details_tmp_algo2.append(tmp)
    viewing_details_tally_algo2.append(tally)

final_choice_algo2
```

Out[17]:
```
[3, 1, 2]
```

The algorithm successfully returned the only solution: [3, 1, 2]

Validation

In [18]:
```
validation_TianJi(final_choice_algo1, choice)
```

Out[18]:
```
True
```

Emperical Testing

In [19]:
```
victory_algo2 = list()
for i in range(RUNS):
    victory_algo2.append(winloss(final_choice_algo2))
```

In [20]:
```
print(f'wins: {sum(victory_algo1)}')
print(f'win rate: {sum(victory_algo1)/RUNS}')
```

```
wins: 100000
win rate: 1.0
```

Viewing Details

In [21]:
```
viewing_details_tally_algo2
```

Out[21]:
```
[{1: -33370, 2: -33683, 3: -153},
 {1: -379, 2: -33061, 3: -33766},
 {1: -33457, 2: -462, 3: -33287}]
```

In [22]:
```
viewing_details_tmp_algo2
```

Out[22]:
```
[[(3, -153), (1, -33370), (2, -33683)],
 [(1, -379), (2, -33061), (3, -33766)],
 [(2, -462), (3, -33287), (1, -33457)]]
```

Problem with code:

These two algorithms did not use mean/proportions to summarise the statistics, rather only the raw count. Whilst it did not affect the results of this game, it is unfair for other games because the number of times we simulate a certain choice for a certain spot is random (and almost certainly not equal), meaning raw counts are like 'unscaled data'.

From algorithm 2 onwards this error has been amended.