Blotto with 300 Jane Streeters with Monte Carlo

Ziyang Ding - Duke University

April 2, 2019

Solution: (0,0,12,17,22,2,3,42,2,0)

Abstract

After Blotto with myself many times, it is easy to recognize that Blotto is a game that **No Strategy always win**. Better strategies can always derived to beat the previous one. Therefore, this is a game without Nash equilibria. However, while I cannot derive a strategy that win every game, but I can propose **good strategy that has high probability of wining**. A basic procedure is as follows:

1. Propose a set of possible "good" strategies of Jane Streeters Ω_S. A strategy is accepted as "good" strategy if it meets the required winning threshold (80%) against random strategy.
2. Label the good strategies we've proposed as $S_1, S_2 \dots S_n$, where $S \in \Omega_S$ is a random variable with a rationally designed distribution.
3. Apply Monte Carlo sampling and simulated annealing, an overall likely-winning strategy S' can be proposed.
4. Conduct psychological qualitative analysis and slightly modify to S' for strategy reinforcement.
5. The modified strategy S is as desired.

1 Introduction

1.1 General procedure explanation

With the procedure been already posted above in abstract, here I make clearer explanation of each procedure and what is the reason of doing so.

1. **Step 1**: The game is to compete with your opponents. And from K-level thinking, **I need to assume that my opponents, Jane Streeters, are intelligent opponents who propose good strategies**. Therefore, any strategy that I need to propose should be K+1 level. To reach that K+1 level strategy, we need to derive the strategy pool Ω_S of 300 Jane Streeters. Therefore, a set of **good strategies**, each of which represents strategies proposed by some Jane Streeters, is needed.

Def: A **solution** is a 10-D vector $\mathbf{x} = (x_1, x_2, \ldots, x_{10})$, with x_i representing the number of soldiers assigned to castle i.

Def: A **strategy** S is a motive (idea) that can generate a set of solutions. **e.g.** I plan to win the game by winning castle 8,9,10. This is a strategy. The solution generated by this strategy can be $(0, 0, 0, 0, 0, 0, 33, 33, 34)$ or others that focuses on winning 8,9,10.

As we aim to derive "good strategies" to mimic the 300 Jane Streeters strategy pool, what exactly are those "good strategies". As the game doesn't have Nash equilibrium, we aim to propose good strategies that has high probability of winning. Therefore, I define "good strategy" as follows:

Def: A **good strategy** is a strategy that its generated (related) solutions can win above 80% of the Random Solutions.

Def: Random Solutions S_0 are randomly solutions drawn uniformly from all the possible solutions.

2. **Step 2**: after we've collected several good strategies $S_1, S_2, \ldots, S_n$, we may assume that our opponent will take random $S \in \Omega_S = \{S_1, S_2, \ldots, S_n\}$ each time with some probability mass on $\mathbb{P}(S = S_i) = P_i, i \in \{1, 2, \ldots, n\}$. So we need to define such discreet probability mass $\mathbb{P}$ function on the discreet event space.

3. **Step 3**: Now we have a distribution on S, notice that $\forall i$, $S_i \in \Omega_S$ **is a strategy with a probability mass function to generate some solutions**. So the Monte Carlo process is to sample $s : s_1, s_2, \ldots, s_k, \ldots, s_n$ from $\mathbb{P}(S)$, and then sample solution $\mathbf{x_k}$ from each s_k we've just sampled, we get a pool of solutions by Jane Streeters: $\mathbf{x_1}, \mathbf{x_2}, \ldots \mathbf{x_n}$. In short, s is **random variable of random variable**, $\mathbb{P}(s = S_i) = \mathbb{P}(S_i)$.

   With the sampled pool of solutions of Jane Streeters, use simulated annealing to find the best solution. Randomly initialize several solutions, create neighbourhood, and iterate to find the global solution S'.

4. **Step 4**: Sometimes we might get solution such as $\mathbf{x} = (5, 10, 15, 20, 25, 25, 0, 0, 0, 0)$. We know that people are sensitive of assigning 5 divisible numbers, so we tactically assign some 1 some 6 for example in our solution, like $\mathbf{x} = (1, 10, 15, 20, 25, 25, 1, 1, 1, 1)$. This might even be better strategy as I bet on people bet nothing on the last 4, rather than directly give up. If I bet correct, then I get lot more points, even if I lost castle 1, which is not too big a deal. So, qualitative adjustment is needed.

5. **Step 5: Tadaa!** ☺

1.2 Preparation

As we've mentioned above: in Step 1, we need to sample Random Solutions S_0 to filter good solutions. But how to sample Random Solutions?

My way is to sample from uniform Dirichlet distribution of 10 dimension:

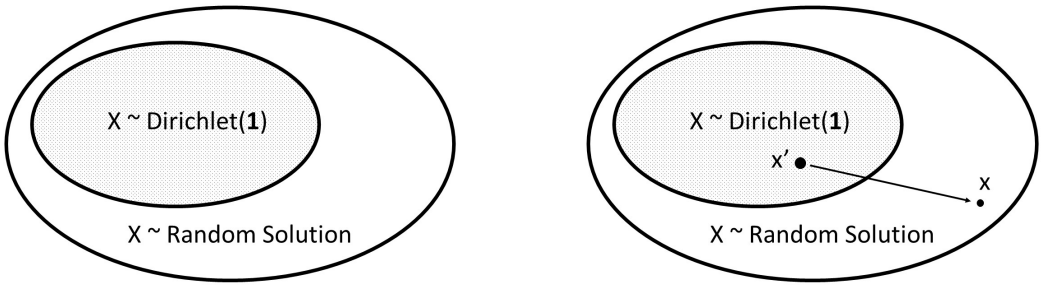
$$\mathbf{x} = \text{round}(100\mathbf{x}')$$
$$\mathbf{x}' \sim \text{Dir}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$$

Notice that sum of all dimension of x might be a little different from 100, we just minus that difference randomly at some dimension.

The code I use for generating Random Solutions is as follows:

```
def makeStrat(alphas,n):
    """
    Generate n solutions using Dirichlet Distribution.
    Random Solution uses alphas = (1,1,1,1,1,1,1,1,1,1)
    """
    strats = np.random.dirichlet(alphas,n)
    strats *= 100
    strats = np.rint(strats)
    adjust = 100 - np.sum(strats,axis=1)
    for i in range(n):
        strats[i,0] += adjust[i]
        np.random.shuffle(strats[i])
    return strats
```

However, the Dirichlet support is $x > 0$, therefore, it eliminates the possibility of having 0 entries of the solution.



(a) Discrepancy (b) exists Better Solution

Figure 1: Discrepancy between Random Solution space and Dirichlet distribution sample space

However, as we've shown above that **the game has no best strategy**, giving 100 sample points, any strategy that lies in Random Solution Space Ω_{S_0} can be beat by **many** solutions in Dirichlet sample space. Therefore, $\forall \mathbf{x} \in \Omega_S, \exists \mathbf{x}' \in \Omega_{\text{Dir}}$, that **x'** is better than **x**. Therefore, it is representative to sample solutions from Ω_{Dir}, the Dirichlet distribution. And it is easy to sample solutions from such distribution.

Now, as we're all prepared, we can move on to derive Jane Streeters' good solutions.

2 Deriving "Good" Strategies

2.1 Strategy 1: Choker
Choke the opponent

One possible good strategy that easily come to mind is to win castle 5,7,8 or 6,7,8. The reason of doing so is the fact 1+2+3+4+6 or 1+2+3+4+5 are both smaller than 20. In short, **even I choose to lose all the 5 castles**, I still have the chance of winning.

However, there are 2 obvious rules of the game:

1. Because the game proceeds in order, it is always important to win the castles at the beginning, rather than at the end, which are weighted more heavily, as it is very probable that the game ends before it reaches the big numbered castles: you lose.

2. Because the game has limited resources, the less castle to assign soldiers, the more probable of winning the game at these castles, as we can assign more soldiers, whereby increase winning probability.

There are many other ways of choking your opponents. But the best strategies are definitely weighing 5,7,8 or 6,7,8. It requires winning minimal numbers of castles, and is still relatively close to the beginning.

To get samples that concentrates at 5,7,8 or 6,7,8, I still choose to sample from 10-D Dirichlet distribution. The code I use is as attached. Slightly different from makeStrat as it shouldn't shuffle.

```
def SampleIntDir(alphas,n):
    """
    Generate n particular dirichlet with specified alpha
    Parameter: n strategies
    Return 2D np array
    """
    strats = np.random.dirichlet(alphas,n)
    strats *= 100
    strats = np.rint(strats)
    adjust = 100 - np.sum(strats,axis=1)
    for i in range(n):
        dim = np.random.randint(0,10)
        strats[i,dim] += adjust[i]
    return strats
```

This will return the desired solutions. To know if the 10-D α parameters are well designed, the code I use to test the 80% threshold is attached below

After optimizing the Integrated Risk, I determine to the following distribution, where each distribution will reach expected score 0.91 and 0.94.

$$S_1 = \begin{cases} \boldsymbol{\alpha}_{\mathbf{1_{578}}} = (1,1,1,1,100,1,100,100,1,1) & (5,7,8) \text{ Strategy} \\ \boldsymbol{\alpha}_{\mathbf{1_{678}}} = (1,1,1,1,1,100,100,100,1,1) & (6,7,8) \text{ Strategy} \end{cases}$$

Both meets the "good strategy" threshold 80%. Therefore, we'll have both of them. Notice that as they both are chocker strategies, therefore, $S_1 = \boldsymbol{\alpha_{578}}$ or $\boldsymbol{\alpha_{678}}$, and we need to assign some probability for $\mathbb{P}(S_1 = \boldsymbol{\alpha_{578}}), \mathbb{P}(S_1 = \boldsymbol{\alpha_{678}})$. It is reasonable to see that castle happens earlier that castle 6, and because it is a smaller castle, it requires less soldiers to win, as people put more weight on bigger castles. So I will assign:

$$\mathbb{P}(S_1 = \boldsymbol{\alpha_{1_{578}}}) = 0.7$$
$$\mathbb{P}(S_1 = \boldsymbol{\alpha_{1_{678}}}) = 0.3$$

```
def getMultiMultiScore(myStrats, yourStrats):
    """
    Calculate average win
    Parameter: MyStrat ndarrays, testing ndarrays
    Return winning rate
    """
    sumsum = 0
    for j in range(myStrats.shape[0]):
        sum = 0
        for i in range(yourStrats.shape[0]):
            sum += getScore(myStrats[j], yourStrats[i])
    sumsum += sum/yourStrats.shape[0]
    return sumsum

def main():
    test =SampleIntDir([1,1,1,1,100,1,100,100,1,1],10000) ## change para here

    print("The score we of this sampling dis is",
          getMultiMultiScore(test, makeStrat([1,1,1,1,1,1,1,1,1,1],10000)))


if __name__== "__main__":
    main()
```

2.2 Strategy 2: Beater
Light beginning, heavy middle, beat the Choker

As strategy 1 can easily come to mind, and considering it's effectiveness, most Jane Streeters do strategy 1, leaving 1,2,3,4 few emphasis. This gives some other Jane Streeters a chance to come up with **Play the beginning, beat the Choker** strategy.

These people will arm very heavy on castle 5 and 6, give a little amount to 1,2,3,4. As there are only 2 castles needs heavy arm, less than that 3 castles in the choker strategy, so there is a positive chance of beating chokers simultaneously on 5 and 6. As for 1,2,3,4, as Chockers don't care 1,2,3,4, few soldiers are needed to win 1,2,3,4, leaving them easy to win.

Now, the mission is to pick the best strategy. As the strategy's motive is to beat the chockers,

it is unfair to test this strategy only on random strategies solution space. The sample space must also include information from S_1 samples. Therefore, the testing solutions I used is a mixture of such proportion, because the Choker Strategy is just so easy to think about and it perform very well, so most people will apply it rather than Random Solutions:

75% S_1: Strategy 1 Chocker solutions
25% Strategy 0 Random Solutions

After optimizing, here I choose the following parameter for beater strategy:

$$\boldsymbol{\alpha_2} = (5, 5, 10, 15, 100, 100, 1, 10, 10, 1)$$
$$S_2 \sim \text{Dirichlet}(\boldsymbol{\alpha_2})$$

This distribution can achieve a winning rate of 92.4% over 70% strategy 1 and 30% random strategy solution space. Therefore, it is definitely good strategy.

2.3 Strategy 3: Innovator Iconoclastic, discrete

While it is easy to come up with strategies that stresses winning consecutive castles, innovators will tend to outscore the previous by stressing nonconsecutive (discrete) castles.

Notice that from what we've discussed above: we wish to have strategy that aim to win the minimum number of castles whose numbers are the smallest possible. The innovator strategy (Strategy 3) can find 2+4+6+8 and 2+4+7+8 really appealing. Comparing to other discrete emphasis, these strateges aims to win only 4 castles, and 2 of them is in the early half. However, notice that castle 6 is very hard to win as the existence of both **chocker** and **beater** strategy that emphasis A LOT on arming heavily on 6. Therefore, 2+4+7+8 might be a better choice.

This time, the good strategy testing solution space incorporates Random Solutions S_0, strategy 1 (Chocker) S_1, and strategy 2 (beater) solutions S_2. Because **if any one of these strategy is not anticipated, the innovator won't do the discrete emphasis, as consecutive emphasis is already enough and way easier to implement**. Also, I give S_2 only 20% the weight because it took me **very long time** to find the optimal S_2 beater strategy that can beat S_1 choker in the last step. So I believe few people can come up with it.

20% S_2: Strategy 2 beater solutions
60% S_1: Strategy 1 Chocker solutions
20% : Stragey 0 Random Solutions

After optimization, I find the following distribution:

$$\boldsymbol{\alpha_3} = (1, 10, 1, 15, 1, 1, 100, 50, 10, 1)$$
$$S_3 \sim \text{Dir}(\boldsymbol{\alpha_3})$$

Note: I didn't put any constraint on castle 6 during optimization. But the result shows that we can give up castle 6 this time. The hypothesis of winning 7 instead of 6 is verified.

The strategy achieves 0.889% of winning rate. Therefore, it's a good strategy.

2.4 Strategy 4: Idealist Overly counting on proportionality and expectation

Note: This is not a good strategy, but it is necessary to include it.

It is obvious that Jane Streeters are good at math. But no matter how smart people are, they make mistakes sometimes. Idealist believe that **as there are too many good strategies**, it is good to be neutral, therefore winning majority of the game. For this game it is not true. But to make the strategy pool more inclusive, I'm earnest to include the strategy:

$$\boldsymbol{\alpha_4} = (2, 4, 7, 9, 11, 13, 15, 18, 21, 1)$$
$$S_2 \sim \text{Dirichlet}(\boldsymbol{\alpha_4})$$

As we can see that it arms each castle proportional to its score weight. **And these idealist are also clever, as they find that** $20 + 20 < 55 = 1 + 2 + \ldots 10$**, and a tie at a castle has very very low probability of happening given we have so much choices, so they cannot count on that! So the probability of the blotto proceed to 10 is almost 0.** Therefore, the idealist will assign absolutely 0 to castle 10, which is why $\alpha_{10} = 1$.

The winning rate against **only random strategy space** (100% Random Solutions S_0) is 0.93! This is also how idealists make mistakes because they didn't anticipate their opponents' bias towards other good strategies. So, if I compete this strategy against the solution space below (same as the one I used in determining S_3), the winning rate suddenly becomes 0.36, **even worse than baseline 0.5!**

20%	S_2: Strategy 2 beater solutions
60%	S_1: Strategy 1 Chocker solutions
20%	: Strategy 0 Random Solutions

2.5 Strategy 5: Computationist harness the brutal force of computer

Recall that a strategy is defined as a motive (idea) than can generate a set of solutions, solve the problem solely by harnessing the brutal force of computer should be counted as another strategy. These computationist (I have to call them as "computationist" or otherwise I can only call them as "computer" which doesn't mean people) is able to find a crude loss function and easily conduct optimization, by which they can find a set of best solutions in the entire solution space.

I'm not a computer science major, but I believe that I can design such algorithm to solve the problem numerically. Algorithm 1 proposed as below. It is very similar as **Simulated Annealing** and **Metropolis Hasting**, but less complicated. It is already enough for this optimization.

To calculate and sample from neighbourhood, I use the following code:

```
def getNeighbourhood(x,n):
    xs = []
    for i in range(n):
        noise = np.random.randint(-2,2,size=9)
        y = x + np.append(noise,-np.sum(noise))
        if np.all(np.greater_equal(y,np.zeros(10))) and
           np.all(np.less_equal(y,np.ones(10)*100)):
            xs.append(y)
    return np.array(xs)
```

Algorithm 1: Finding the best solutions

Result: Given a set random initialized solutions, the algorithm can iterate and update each solutions and at last converge to a set of best solutions R

Initialize $n = 1000$ Random Solutions $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$ in the entire solution space;

for $i = 1, 2, 3 \ldots n$ **do**

  while $\boldsymbol{x_i}$ *not converged* **do**

    Generate $q = 500$ solutions $\{\boldsymbol{a_1}, \boldsymbol{a_2}, \ldots, \boldsymbol{a_q}\} = A$ from Dirichlet$(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$;

    Sample $m = 500$ points uniformly from $\boldsymbol{x}_i$'s neighbourhood $\{\boldsymbol{x_i}\}_m$;

    Find the $\boldsymbol{x'} \in \{\boldsymbol{x_i}\}_m$ that has highest winning rate with respect to A;

    Update $\boldsymbol{x}_i = \boldsymbol{x'}$;

  end

  if $\boldsymbol{x_i}$ *has winning rate* $\geq$ *0.8* **then**

    keep $\boldsymbol{x_i}$ in result set R;

  else

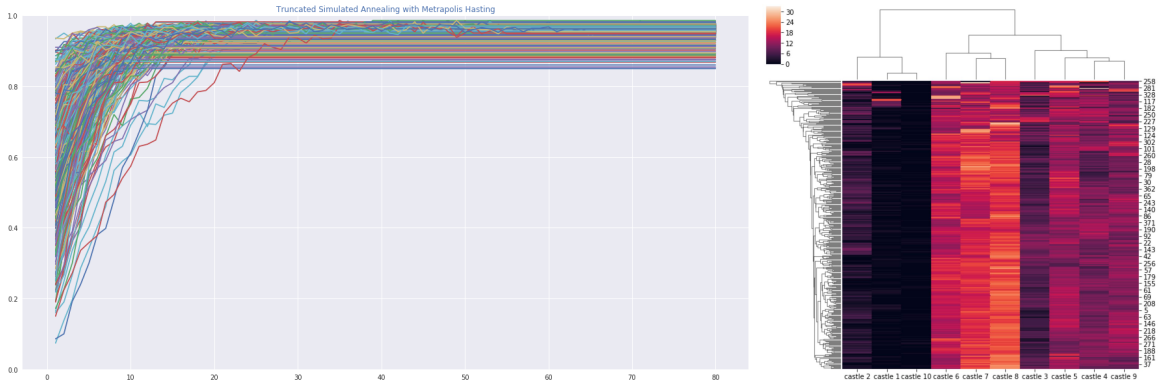
    delete $\boldsymbol{x_i}$;

  end

end

Return R

The algorithm gives back 385 solutions, with average winning rate at 9.37, mainly 6,7,8 focused. But from hierarchical clustering, there are 3 clusters. I'll incorporate all clusters solutions in Ω_{S_5}.



(a) Convergence (b) Solution Cluster

Figure 2: Computational strategy generated solutions

3 Create Strategy Random Variable and Probability

Now all strategies are as follows:

$$\begin{aligned}
\textbf{Random} \quad & S_0 \sim \text{Dirichlet}(1,1,1,1,1,1,1,1,1,1) \\
\textbf{Choker} \quad & S_1 \sim \begin{cases} \text{Dirichlet}(1,1,1,1,100,1,100,100,1,1) & 70\% \\ \text{Dirichlet}(1,1,1,1,1,100,100,100,1,1) & 30\% \end{cases} \\
\textbf{Beater} \quad & S_2 \sim \text{Dirichlet}(5,5,10,15,100,100,1,10,10,1) \\
\textbf{Innovator} \quad & S_3 \sim \text{Dirichlet}(1,10,1,15,1,1,100,50,10,1) \\
\textbf{Idealist} \quad & S_4 \sim \text{Dirichlet}(2,4,7,9,11,13,15,18,21,1) \\
\textbf{Computationist} \quad & S_5 \sim 385 \text{ data Solution set } A
\end{aligned}$$

Define random variable as $S \in \Omega_S = \{S_0, S_1, S_2, S_3, S_4, S_5\}$, assign probability as

$$\begin{aligned}
\textbf{Random} \quad & \mathbb{P}(S = S_0) = 0.2 \\
\textbf{Choker} \quad & \mathbb{P}(S = S_1) = 0.48 \\
\textbf{Beater} \quad & \mathbb{P}(S = S_2) = 0.16 \\
\textbf{Innovator} \quad & \mathbb{P}(S = S_3) = 0.16 \\
\textbf{Idealist} \quad & \mathbb{P}(S = S_4) = 0.3 \\
\textbf{Computationist} \quad & \mathbb{P}(S = S_5) = 0.15
\end{aligned}$$

Reasons are as follows:

1. S_0 will present, but only few bewildered will do it.
2. S_1 is easy to come up with and it is very efficient. Most people will use it.
3. S_2 is little harder to come up with but is very efficient in beating S_1.
4. S_3 is innovative and is therefore harder to come up with.
5. S_4 will be avoided by most people. But still will few use it.
6. S_5 is very favored by Computer Scientist, therefore being embraced.

As the sampling distribution of $S' \in \Omega_S$ is specified, we can start Monte Carlo Sampling and Optimization:

4 Monte Carlo Sampling with Optimization

This part comes straight forward. I'll just write out the algorithm that I used to find the set of best solutions. The algorithm is very similar as Algorithm 1 as the computationist optimization method. It only differ in what each solution is comparing to, rather than Random Solutions sampled from Dirichlet$(1,1,1,1,1,1,1,1,1,1)$, it is sampled from Ω_S. Also, I raised the passing threshold to 90%, as I don't need too many solutions. Using this method, we can find best solutions.

Algorithm 2: Finding the best final solutions

Result: Given a set random initialized solutions, the algorithm can iterate and update each solutions and at last converge to a set of best solutions R

Initialize $n = 50$ Random Solutions $\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}$ in the entire solution space;

for $i = 1, 2, 3 \ldots n$ **do**

  while $\boldsymbol{x_i}$ *not converged* **do**

    Generate $q = 10000$ solutions $\{\boldsymbol{a_1}, \boldsymbol{a_2}, \ldots, \boldsymbol{a_q}\} = A$ from Ω_S;

    Sample $m = 5000$ points uniformly from $\boldsymbol{x}_i$'s neighbourhood $\{\boldsymbol{x_i}\}_m$;

    Find the $\boldsymbol{x'} \in \{\boldsymbol{x_i}\}_m$ that has highest winning rate with respect to A;

    Update $\boldsymbol{x}_i = \boldsymbol{x'}$;

  end

  if $\boldsymbol{x_i}$ *has winning rate* $\geq$ *0.9* **then**

    keep $\boldsymbol{x_i}$ in result set S';

  else

    delete $\boldsymbol{x_i}$;

  end

end

Return S'

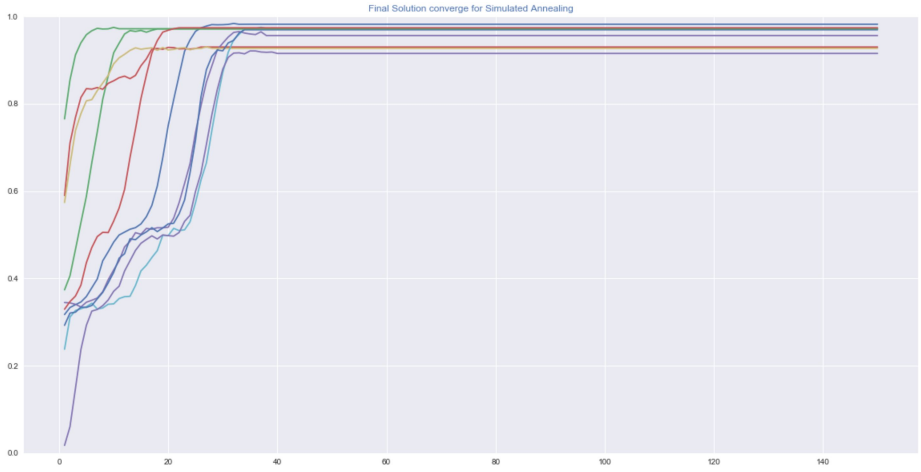


Figure 3: Convergence of 10 best solutions

We've rejected many solutions that converged to low winning rate local minima. But We've also got 10 following very good solutions. Select one. To **avoid that someone uses similar method as my solution**, I need to find the one below that can beat the other 9 directly:

$$S'_1 = (0, 0, 13, 17, 22, 2, 3, 41, 2, 0) \qquad S'_2 = (0, 8, 6, 12, 0, 0, 21, 39, 13, 1)$$
$$S'_3 = (3, 10, 3, 14, 3, 20, 39, 7, 1, 0) \qquad S'_4 = (1, 0, 0, 12, 15, 17, 2, 38, 15, 0)$$
$$S'_5 = (5, 8, 12, 14, 2, 2, 39, 0, 18, 0) \qquad S'_6 = (0, 6, 6, 14, 0, 19, 0, 39, 16, 0)$$
$$S'_7 = (0, 0, 2, 8, 18, 2, 23, 39, 8, 0) \qquad S'_8 = (0, 4, 9, 16, 0, 20, 0, 39, 11, 1)$$
$$S'_9 = (3, 5, 1, 0, 19, 21, 2, 40, 8, 1) \qquad S'_{10} = (3, 9, 3, 14, 2, 20, 38, 6, 5, 0)$$

And the winner is $S' = S'_1 = (0, 0, 13, 17, 22, 2, 3, 41, 2, 0)$, it wins all other 9.

5 Qualitative Modification and Reinforcement

It easy to see that the solution aims to win 3,4,5,8. And it has to win all of them! Notice it assigns 41 to castle 8. And usually people will always put 1 more on castles that they want to win in order to surpass the opponent by a tiny margin, so the probability of having a tie at castle 8 is pretty high. So I decide to change the solution to $(0, 0, 12, 17, 22, 2, 3, 42, 2, 0)$

Final Solution $S_{\textbf{final}} = (0, 0, 12, 17, 22, 2, 3, 42, 2, 0)$

Appendix-code

5.1 Packages

```
import numpy as np
import random
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from sklearn.cluster import DBSCAN
from sklearn.cluster import AgglomerativeClustering
import seaborn as sns
from random import choices
from numba import jit
```

5.2 Functions

```
## Calculate win or not win
## Parameter: MyStract ndarray, yourStrat ndarray
## Return win 1 or lose 0
def getScore(myStrat, yourStrat):
    myScore = 0
    yourScore = 0
    i = 0
    while myScore < 20 and yourScore < 20 and i<10:
        if (myStrat[i] > yourStrat[i]):
            myScore += (i+1);
        if myStrat[i] < yourStrat[i]:
            yourScore += (i+1);
        i+=1
    if myScore >= 20:
        return 1
    if yourScore >= 20:
        return 0
    else:
        return 0


## Generate n random dirichlet
## Parameter: n strategies
## Return 2D np array
def makeStrat(alphas,n):
    strats = np.random.dirichlet(alphas,n)
    strats *= 100
    strats = np.rint(strats)
    adjust = 100 - np.sum(strats,axis=1)
    for i in range(n):
```

```
        strats[i,0] += adjust[i]
        np.random.shuffle(strats[i])
    return strats

## Generate n particular dirichlet with specified alpha
## Parameter: n strategies
## Return 2D np array
def SampleIntDir(alphas,n):
    strats = np.random.dirichlet(alphas,n)
    strats *= 100
    strats = np.rint(strats)
    adjust = 100 - np.sum(strats,axis=1)
    for i in range(n):
        dim = np.random.randint(0,10)
        strats[i,dim] += adjust[i]
    return strats

## Calculate average win
## Parameter: MyStrat ndarray, testing ndarrays
## Return winning rate
def getMultiScore(myStrat, yourStrats):
    sum = 0
    for i in range(yourStrats.shape[0]):
        sum +=getScore(myStrat, yourStrats[i])
    return sum/yourStrats.shape[0]

## Calculate average win
## Parameter: MyStrat ndarrays, testing ndarrays
## Return winning rate
def getMultiMultiScore(myStrats, yourStrats):
    sumsum = 0
    for j in range(myStrats.shape[0]):
        sum = 0
        for i in range(yourStrats.shape[0]):
            sum +=getScore(myStrats[j], yourStrats[i])
    sumsum += sum/yourStrats.shape[0]
    return sumsum

## Get a neighbourhood
def getNeighbourhood(x,n):
    xs = []
    for i in range(n):
        noise = np.random.randint(-2,2,size=9)
        y = x + np.append(noise,-np.sum(noise))
        if np.all(np.greater_equal(y,np.zeros(10))) and
           np.all(np.less_equal(y,np.ones(10)*100)):
            xs.append(y)
    return np.array(xs)
```

```
## Update random solutions in optimization
def update(x, ref):
    max = 0
    win = 0
    for i in range(x.shape[0]):
        if getMultiScore(x[i],ref) > win:
            max = x[i]
            win = getMultiScore(x[i],ref)
    return (max,win)

# sample testing solutions
def getSingleRef(strat,dic,S5data):
    if strat != 'S5':
        return SampleIntDir(dic[strat],1)[0]
    else:
        i = np.random.randint(S5data.shape[0])
        return S5data[i]
```

5.3 Miscellaneous Main Functions

```
"""
Pick the unique final solution
"""
def main():
    bests = pd.read_csv('FinalSolutionsBig.csv', sep=',',header=None).values

    for i in range(bests.shape[0]):
        print(bests[i],": winning rate :", getMultiScore(bests[i],bests))

if __name__== "__main__":
    main()

#################### Separating Line ######################

"""
Testing the solution
"""
def main():

    bests = pd.read_csv('FinalSolutionsBig.csv', sep=',',header=None).values


    S5=pd.read_csv('computationalApproachCentroids.csv', sep=',',header=None).values
    StratsInfo = {'S0': [1,1,1,1,1,1,1,1,1,1],
              'S11': [1,1,1,1,100,1,100,100,1,1],
              'S12': [1,1,1,1,1,100,100,100,1,1],
              'S2': [5,5,10,15,100,100,1,10,10,1],
```

```
                'S3': [1,10,1,15,1,1,100,50,10,1],
                'S4': [2,4,7,9,11,13,15,18,21,1],
                'S5': S5}
    Strats = ["S0","S11","S12","S2","S3","S4","S5"]
    #weight = [0.02,0.336,0.144,0.16,0.16,0.03,0.15]
    weight = [0.1,0.328,0.15,0.11,0.2,0.04,0.1]
    refSize = 10000

    ##This part get ref
    ref = []
    for i in range(refSize):
        ref.append(getSingleRef(choices(Strats, weight)[0],StratsInfo,S5))
    ref =np.array(ref)

    for j in range(bests.shape[0]):
        print(getMultiScore(bests[j],ref))


if __name__== "__main__":
    main()

################### Separating Line ######################

"""
This is for the last Monte Carlo find best solution
"""
def main():

    S5=pd.read_csv('computationalApproachCentroids.csv', sep=',',header=None).values
    StratsInfo = {'S0': [1,1,1,1,1,1,1,1,1,1],
                'S11': [1,1,1,1,100,1,100,100,1,1],
                'S12': [1,1,1,1,1,100,100,100,1,1],
                'S2': [5,5,10,15,100,100,1,10,10,1],
                'S3': [1,10,1,15,1,1,100,50,10,1],
                'S4': [2,4,7,9,11,13,15,18,21,1],
                'S5': S5}
    Strats = ["S0","S11","S12","S2","S3","S4","S5"]
    weight = [0.02,0.336,0.144,0.16,0.16,0.03,0.15]

    maxIter = 150
    numInitial = 50
    refSize = 10000
    stopThre = 4
    neighbourSize = 5000
    mpl.style.use("seaborn")
    fig, ax = plt.subplots(figsize=(20, 10))
    ax.set_ylim([0, 1])
    ax.set_title('Final Solution converge for Simulated Annealing'.format("seaborn"), color='C0')
```

```
x = makeStrat([1,1,1,1,1,1,1,1,1,1],numInitial)
strats = []
wins = []
for j in range(numInitial):
    converge = False
    good = False
    print("Processing",j+1,"initial point:")
    centroid = x[j]
    win = []
    for i in range(maxIter):
        if i% 50 == 0:
            print("   iteration:",i)
        neigh = getNeighbourhood(centroid, neighbourSize)

        ##This part get ref
        ref = []
        for i in range(refSize):
            ref.append(getSingleRef(choices(Strats, weight)[0],StratsInfo,S5))
        ref =np.array(ref)

        inter = update(neigh,ref)

        if np.sum(np.abs(centroid- inter[0])) < stopThre:
            print("   Converged!!")
            converge = True
            break

        centroid = inter[0]
        win.append(inter[1])
        print(win[-1])

    ## Reject or Accept sample solution
    if win[-1] >= 0.85:
        good = True

    if good and converge:
    ## Reject or Accept sample Solution
        strats.append(centroid)
        pd.DataFrame(np.array(strats)).to_csv("FinalSolutions.csv",
                                        header=None, index=None)

    ## Make or not make solution plot
        win = np.concatenate((np.array(win),np.ones(maxIter-len(win))*win[-1]), axis = 0)
        wins.append(win)
        ax.plot([i+1 for i in range(len(win))],  win)

ax.plot([i+1 for i in range(len(win))],  [0.85 for i in range(len(win))],
```

```
                                linestyle = '--', color = "yellow")

    plt.show()
    fig.savefig('FinalSolutionConvergence.jpg')

if __name__== "__main__":
    main()

################### Separating Line ######################
"""
Interim Code I use to test strategy motives
"""

def main():

    #for i in [14,15,16,17]:

    test = SampleIntDir([10,10,10,10,10,10,10,10,10,10],1)
    centroid = test[0]
    for i in range(1000):
        lol1 =SampleIntDir([1,1,1,1,100,1,100,100,1,1],700)
        lol2 =SampleIntDir([1,1,1,1,1,100,100,100,1,1],300)
        lol = np.concatenate((lol1,lol2),axis=0)
        random = makeStrat([1,1,1,1,1,1,1,1,1,1],300)
        ref1 = np.concatenate((lol,random),axis=0)
        beater = SampleIntDir([5,5,10,15,100,100,1,10,10,1],300)
        ref2 = np.concatenate((beater,ref1),axis=0)
        innovator = SampleIntDir([1,10,1,15, 1, 1, 100, 50, 10,1],400)
        ref  = np.concatenate((beater,ref2),axis=0)

        neigh = getNeighbourhood(centroid, 500)
        inter = update(neigh,ref)
        centroid = inter[0]

        if i&5 ==0:
            print("processing", i+1,"point")
            print("win rate=",inter[1])
            print(centroid)

    print("The score of this sampling dist is",
                 getMultiMultiScore(test, ref))

if __name__== "__main__":
    main()


################### Separating Line ######################
```

```
"""
This is for Computationist approach clustering and analyzing the solutions
"""

def main():
    strats=pd.read_csv('computationalApproachCentroids.csv', sep=',',header=None)

    kmeans = AgglomerativeClustering(3).fit(strats)
    colors = ["black","blue","red","yellow","green"]
    col = [colors[kmeans.labels_[i]] for i in range(len(strats))]
    X_embedded = PCA(n_components=2).fit_transform(strats)
    plt.scatter(X_embedded[:,0], X_embedded[:,1], color = col)

    strats=pd.DataFrame(data=strats.values, columns = ["castle 1","castle 2","castle 3",
                            "castle 4","castle 5","castle 6",
                            "castle 7","castle 8","castle 9","castle 10"])
    g = sns.clustermap(strats,metric="cityblock")


if __name__== "__main__":
    main()

################### Separating Line ######################
"""
This is for Computationist approach find the best solutions
"""

def main():
    myStrat = np.array([10,10,10,20,20,27,1,1,1,1])
    centroid = myStrat

    maxIter = 80
    numInitial = 1000
    refSize = 500
    stopThre = 5
    neighbourSize = 500
    mpl.style.use("seaborn")
    fig, ax = plt.subplots(figsize=(20, 10))
    ax.set_ylim([0, 1])
    ax.set_title('Truncated Simulated Annealing
            with Metrapolis Hasting'.format("seaborn"), color='C0')

    x = makeStrat([1,1,1,1,1,1,1,1,1,1],numInitial)
    strats = []
    wins = []
    for j in range(numInitial):
        converge = False
        good = False
```

```
        print("Processing",j+1,"initial point:")
        centroid = x[j]
        win = []
        for i in range(maxIter):
            if i% 50 == 0:
                print("   iteration:",i)
            neigh = getNeighbourhood(centroid, neighbourSize)
            ref = makeStrat([1,1,1,1,1,1,1,1,1,1],refSize)
            inter = update(neigh,ref)

            ## Convergence check
            if np.sum(np.abs(centroid- inter[0])) < stopThre:
                print("   Converged!!")
                converge = True
                break

            centroid = inter[0]
            win.append(inter[1])

        ## Reject or Accept sample solution
        if win[-1] >= 0.85:
            good = True

        if good and converge:
        ## Reject or Accept sample Solution
            strats.append(centroid)
            pd.DataFrame(np.array(strats)).to_csv("computationalApproachCentroids.csv"
                                                  ,header=None, index=None)

        ## Make or not make solution plot
            win = np.concatenate((np.array(win),
                        np.ones(maxIter-len(win))*win[-1]), axis = 0)
            wins.append(win)
            ax.plot([i+1 for i in range(len(win))],  win)

    ## Threshold line
    ax.plot([i+1 for i in range(len(win))],  [0.85 for i in range(len(win))],
            linestyle = '--', color = "yellow")

    ## Show the plot, save the plot
    plt.show()
    fig.savefig('computationalApproach.jpg')

if __name__== "__main__":
    main()
```