Jane Street 2021 Blotto Tournament Submission

Steven DiSilvio

May 2021

1 Solution

$$[\mathbf{1, 1, 42, 27, 11, 5, 6, 3, 4, 0}]$$

2 Rationale

When considering the problem at hand, my first thought was to imagine as many possible strategies as I could that would be utilized by my opponents given the two ways of winning points: either directly or by forfeiture of the remaining castles via 3 consecutive victories. For each strategy I also thought of specific strategies to counter them, and in such a way was able to build up a comprehensive set of strategies that my solution should ideally guard against as effectively as possible. To better explicate this concept, here are two examples of strategies and corresponding solutions which I considered:

- Clustering: Consolidate most soldiers into 3 consecutive castles, relatively evenly distributed, in order to maximize the chances of winning 3 consecutively. Examples include $[0, 0, 34, 31, 31, 1, 1, 1, 1, 2]$ as well as $[0, 0, 34, 29, 28, 1, 1, 1, 1, 2]$.
- Cluster Buster: Divert soldiers to dominate every third castle after either the first, second or third in order to minimize the chance of an opponent winning 3 consecutively, and hence directly opposing Clustering. One example is $[31, 1, 1, 31, 1, 1, 31, 1, 1, 1]$.

In an effort to limit the length of this paper not every strategy considered is discussed. However, once these strategies were laid out, I used a multinomial distribution to create solutions implementing each strategy, which were incorporated into a matrix containing the entire solution set (SolutionSet in the code below). Within this matrix, I represented what I felt was the likelihood that each strategy would be utilized by opponents by including certain classes of solutions more or less than others.

From these solutions, I then wrote code to perform stochastic hill climbing on the data, starting with various different solutions I would change between runs

(set to a uniform distribution in the code below as an example) and allowing the optimization process to be carried out. However, important to note is that rather than simply using the effectiveness against solutions in the solution matrix as the objective function, I also factored in previous solutions discovered during the optimization process, giving each equal weights. I figured that most people applying won't be utilizing strategies that can easily be thought of, and so as my algorithm discovered optimal solutions it added them to a separate matrix of solutions to compare against. Below are the main functions utilized:

```
### Functions of Importance ###

def win_rate(solution, allSolutions):
  wins = 0
  for i in range(allSolutions.shape[0]):
    wins = wins+1 if solution_wins(solution, allSolutions[i]) else
    wins
  return wins/(allSolutions.shape[0])

def get_neighbor(solution):
  neighbor = solution.copy()
  nonzero_indices = []
  for i in range(10):
    if solution[i] >0:
      nonzero_indices.append(i)
  index_to_remove_from = random.choice(nonzero_indices)
  index_to_add_to = random.randint(0,9)
  neighbor[index_to_remove_from] = neighbor[index_to_remove_from] -
    1
  neighbor[index_to_add_to] = neighbor[index_to_add_to] + 1
  return neighbor

def simulated_anneal(currentSolution, originalSolutions,
    improvedSolutions, overallBestRate, overallBestSolution ):
  modifiedSolution = get_neighbor(currentSolution)
  modifiedRate = (win_rate(modifiedSolution, originalSolutions) +
    win_rate(modifiedSolution, improvedSolutions))/(2)
  currentRate = (win_rate(currentSolution, originalSolutions) +
    win_rate(currentSolution, improvedSolutions))/(2)
  if modifiedRate > overallBestRate:
    overallBestRate = modifiedRate
    overallBestSolution = modifiedSolution
  if random.random() < .05 or modifiedRate > currentRate:
    currentSolution = modifiedSolution
    np.vstack([improvedSolutions , modifiedSolution])

  return currentSolution, originalSolutions, improvedSolutions,
    overallBestRate, overallBestSolution
```

Using these functions, the actual implementation to find the solution was quite simple, shown below:

```
### Main Method ###

current = np.array([10,10,10,10 ,10, 10, 10, 10, 10, 10])
originalSolutionSet = SolutionSet
improvedSolutionSet = SolutionSet.copy()
bestWinRate = 0
winningSolution = np.array([10,10,10,10 ,10, 10, 10, 10, 10, 10])

def printit():
  threading.Timer(10.0, printit).start()
  print(bestWinRate, winningSolution)

printit()

while(True):
  current, originalSolutionSet, improvedSolutionSet, bestWinRate,
    winningSolution = simulated_anneal(current, originalSolutionSet
    , improvedSolutionSet, bestWinRate, winningSolution)
```

Once I saw the algorithm converge on a solution and no longer change, I had my answer. I continued to tweak the parameters as I saw fit, particularly with respect to the proportions for strategies that other opponents would use. By comparing the various solutions I received I noticed a similar pattern amongst the way they distributed their soldiers, and chose the particular solution I did as through numerous trials I found it had a slight advantage.

Using my algorithm and the various assumptions I made regarding the distribution of opponent solutions, my solution was estimated to be roughly 89% effective.