PROJECT REPORT

BITS Pilani K K Birla Goa Campus

Optimal Coalition Structure problem with Voting indices computation using influence matrix

In partial fulfillment of the course MATH F366 - LABORATORY PROJECT
Instructor: Dr Himadri Mukherjee

2015B4A30823G Prahalad Atreyaa A

November 2018

ACKNOWLEDGEMENTS

We would like to thank Dr. Himadri Mukherjee for his patient guidance as well as the encouragement throughout the course of this project.

ABSTRACT

The optimal coalition structure problem has been solved using the subset generator function. We have included the influence matrix to factor in the relationship between various agents. We have solved the optimal coalition structure problem and applied it to the problem of passing a bill in the parliament. We also explore evolutionary algorithms approach to solve the problem. Specifically, we use genetic algorithms to solve the problem.

Contents

1 Introduction

1.1 Problem Statement

Let there be n players, namely,$P = \{ p_1, p_2, ..., p_n \}$, with each p_i having weights $w_1, w_2, ..., w_n$, where $w_i \in R$. Additionally, let A = $(a_{ij}) \subset M_n(R)$ be the association matrix (i.e. influence matrix) and $| a_{ij} | \leq 1$. 'A' describes the relation that the i^{th} player has with the j^{th} player. We have made an implicit assumption here that the matrix will be symmetric as both player i and player j perceive each other by the same amount. Let q be the quota where $q \in R$ and $q > 0$.

Definition 1.1.1: Any non-empty, and non-singleton subset of P is called a Coalition C, i.e., any $C \subset P$ is called a Coalition.

Definition 1.1.2: $W = \{C \quad | \quad \sum_{P_i \in C} w_i > q\}$ is called the set of winning coalitions. It contains all those coalitions which when formed can clear the quota q, owing to their weigths.

Definition 1.1.3: $P_j = P \in C$ is called critical if the following conditions are met. They are as follows:
i) $C \in W$.
ii) $\sum_{P_j \in C} w_i - \sum_{i=1}^{n} a_{ij} w_i < q$.

Definition 1.1.4: The voting index is defined in equation 1 as follows:

$$\beta_i = \frac{\{C \quad | \quad P_i \quad is \quad critical\}}{2^n - 1} \tag{1}$$

1.2 Solving the optimal structure problem using influence matrix

Any non-empty, and non-singleton subset of P is called a coalition C. The coalition structure is the manner in which players are distributed in coalitions of various sizes. For instance, the coalition structure, when there are three players, are namely: $\{1,2\},\{2,3\},\{3,1\}$, and $\{1,2,3\}$. $\{1,2,3\}$ is a special case, and is called the grand coalition as it contains all the players. The number of coalition structures for each n is given by the sum of Stirling Numbers of the Second Kind $SN(n,k)-1$.

$$\begin{aligned} CS(n) &= \sum_{k=1}^{n} SN(n,k) - 1 \\ &= \frac{1}{k!}\sum_{i=0}^{k}(-1)^k \binom{k}{i}(k-i)^n - 1 \end{aligned} \tag{2}$$

The code for solving the problem in the specific case of computing voting indices for passing a bill in the parliament are as follows:

```
# coding: utf-8

# In[36]:

import pandas as pd
import numpy as np
from itertools import chain, combinations

weights = map(float, pd.read_excel("weight.xlsx", sheet_name=0).weight)
def main(q):
    df = pd.read_excel("assoc.xlsx", sheet_name=0)
    a_matrix = df.values.tolist()
    winners = winning_coaltions(weights, q)
    critical_winners = critical_winning_coalitions(weights, q, winners,a_matrix)
    ww = list(enumerate(weights))
    arr = []
    for coalition in ww:
        count = 0
        for winner in critical_winners:
            if coalition in winner:
                count += 1
        arr.append(count*1.0/len(powerset(weights)))
    B_total = len(critical_winners)*1.0/len(powerset(weights))
#     print "B_total", B_total
    ww = [ww[i]+(arr[i]/B_total,) for i in range(len(weights))]
#     print "\nB_i"
#     printing(ww)

    return ww, B_total
#     printing(critical_winners)


#to be compared with quota
def critical_coalition_value(coalition, w, A):
    v = []
    for i in xrange(len(A)):
```

```
        v.append(np.dot(w, np.array(A[i])))
    sum_A = reduce(lambda x,y: x+y, v)
    return sum(map(lambda x: x[1], coalition)) - sum_A


#To find all the subsets
def powerset(iterable):
    '''
    eg: powerset of the list [1,2,3]
    powerset([1,2,3]) --> (1) (2) (3) (1,2) (1,3) (2,3) (1,2,3)
    '''
    s = list(iterable)
    return list(chain.from_iterable(combinations(s, r) for r in range(1, len(s)+1)))


def winning_coaltions(agent_list, quota):
    winning_coaltions = []
    agent_list_enumerated = enumerate(agent_list)
    for coalition in powerset(agent_list_enumerated):
        if sum(map(lambda x: x[1], coalition)) >= quota:
            winning_coaltions.append(list(coalition))
    return winning_coaltions


def critical_winning_coalitions(agent_list, quota, winning_coaltions, influence_matrix)
    :
    winning_critical_coaltions = []
    for coalition in winning_coaltions:
        if critical_coalition_value(coalition, agent_list, influence_matrix) < quota:
            winning_critical_coaltions.append(coalition)
    return winning_critical_coaltions

def printing(arr):
    for x in arr:
        print x
import matplotlib.pyplot as plt
import numpy as np

def data(start_q, end_q, step_size):
    arr = []
    for q in range(start_q, end_q, step_size):
        ww, B_total = main(q)
        arr.append(ww+[B_total,q])
    return arr
super_arr = data(100, 544, 1)
print super_arr
super_dict = {el:[] for el in weights}
for i in super_arr:
    for j in i:
        if type(j) == tuple:
            super_dict[j[1]].append(j[2])
super_dict
x = range(100, 544, 1)
#To plot graphs of voting indices vs quota for each player- 7 graphs
for key, val in super_dict.iteritems():
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_title(key)
    ax.plot(x, val, 'b.')
    plt.ylabel("Bi over quotas for "+str(key))
    plt.xlabel("Quotas")
    fig.savefig("Bi_over_quotas_for_"+str(key)+'.png')
plt.show()
#To plot graphs of total voting indices vs quota

arr = []
```

```
for i in super_arr:
    arr.append(i[-2])

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title("total_bi with diff quotas")
ax.plot(x, arr, 'b.')
fig.savefig('total_bi_with_diff_quotas.png')
plt.show()

#To output the association matrix after reading the data from the corresponding #excel
    sheet
df = pd.read_excel("assoc.xlsx", sheet_name=0)
df
#To output the weight row matrix after reading the data from the corresponding #excel
    sheet
weights = map(float, pd.read_excel("weight.xlsx", sheet_name=0).weight)
weights
```

The graphs of β_i vs quota q for all the players gives us an idea of how they will behave and how their coalition formation pattern is affected, when quota is varied. The graphs for all the players are given below:

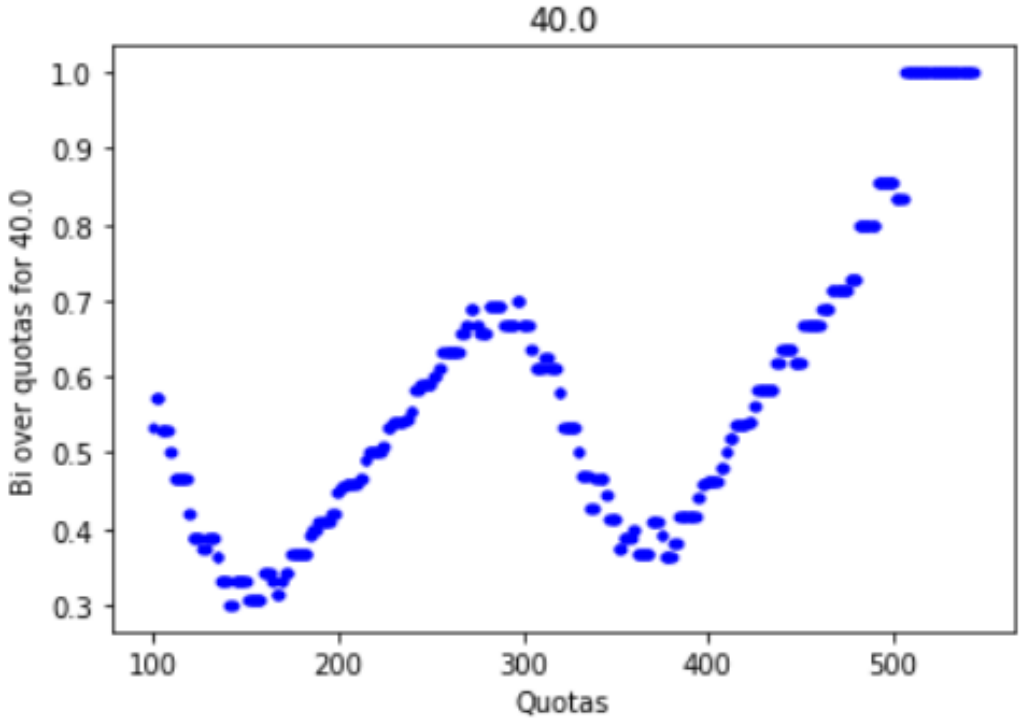

Figure 1: Graph of player 1's voting index β_1 vs q.

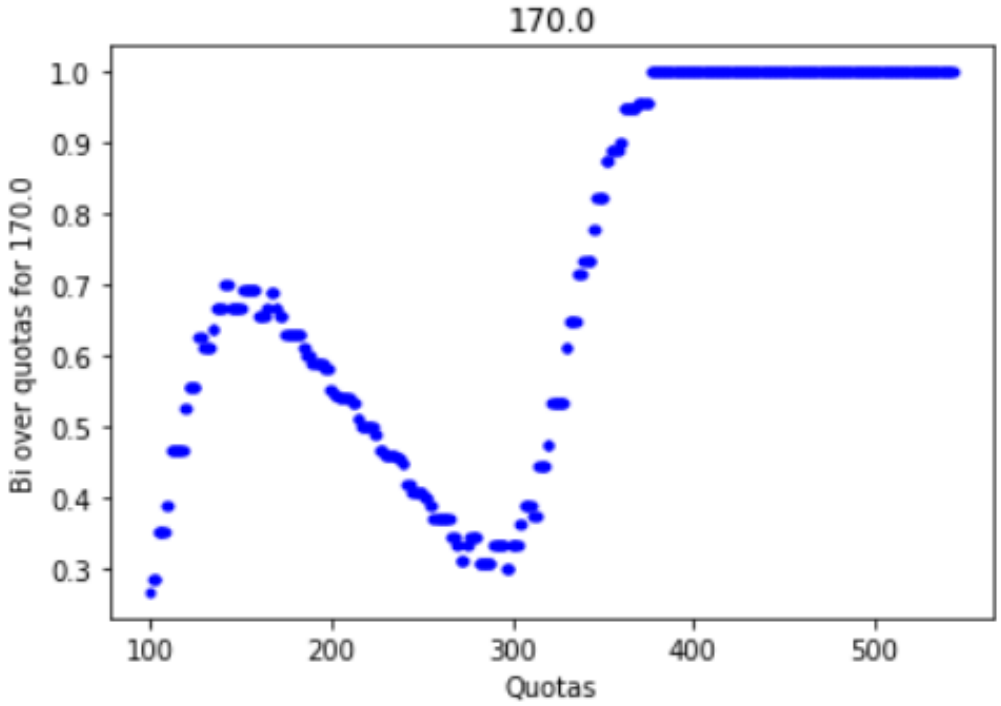

Figure 2: Graph of player 2's voting index β_2 vs q.

Figure 8 shows the total voting index (β_{total}) behaviour with respect to quota q. Figure 9 shows the association matrix that is used. It is computed and normalized from past ten years data. Also, Figure 10 shows the weight matrix.

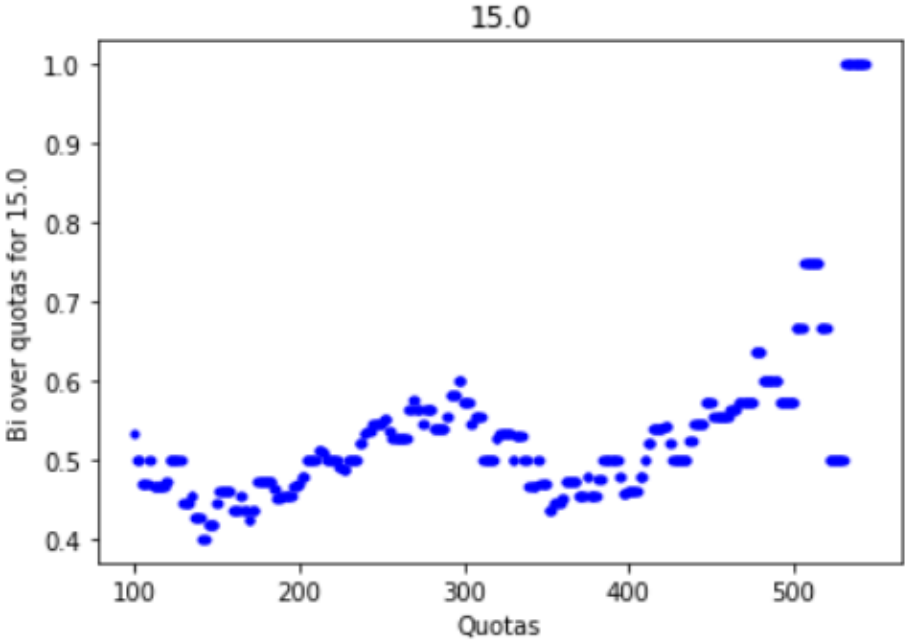

Figure 3: Graph of player 3's voting index β_3 vs q.

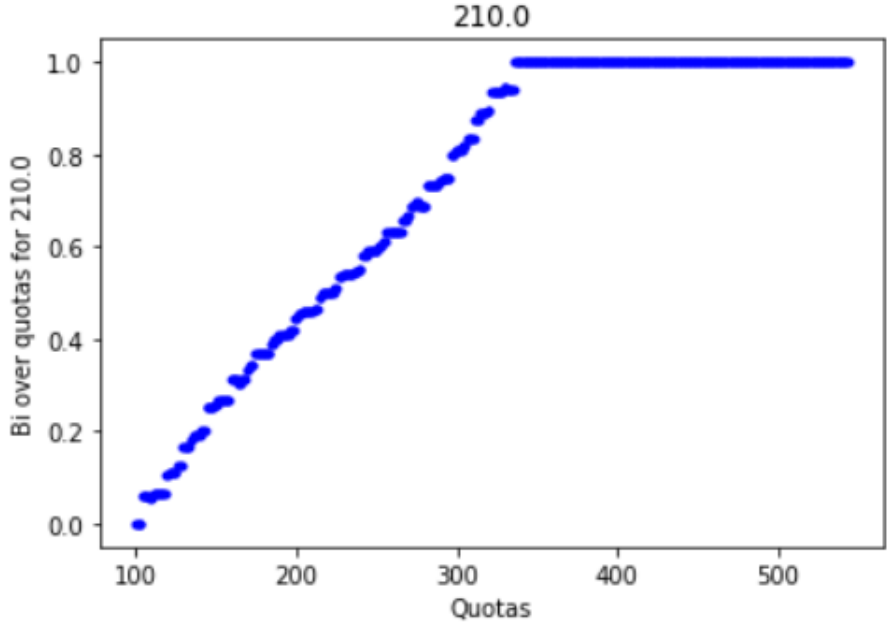

Figure 4: Graph of player 4's voting index β_4 vs q.

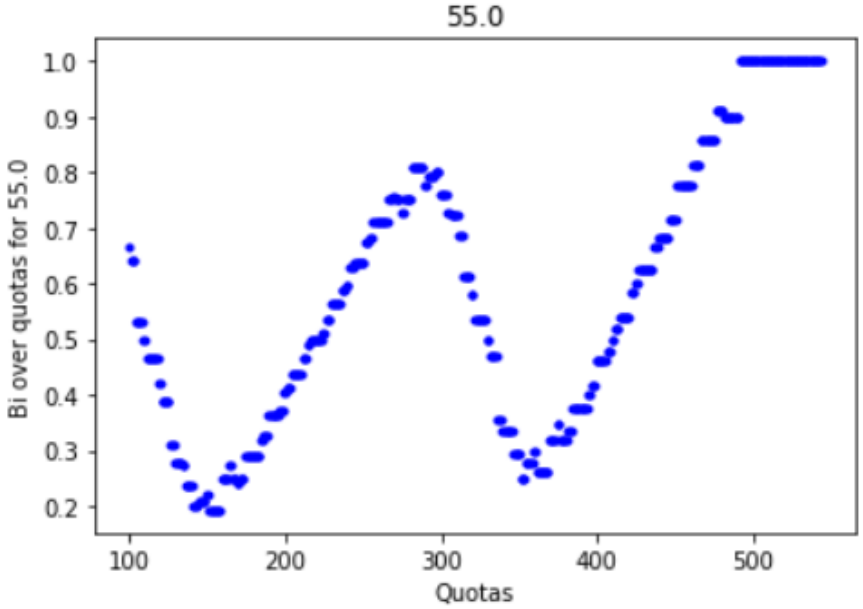

Figure 5: Graph of player 5's voting index β_5 vs q.

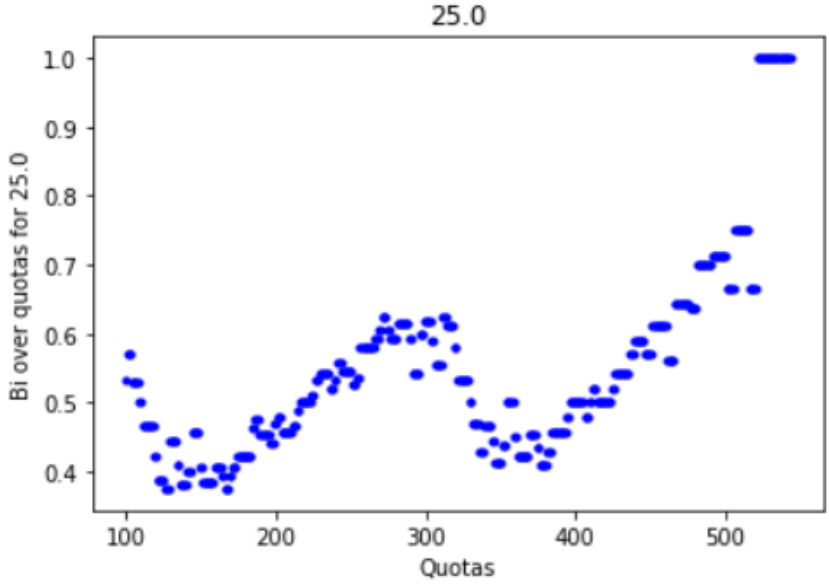

Figure 6: Graph of player 6's voting index β_6 vs q.

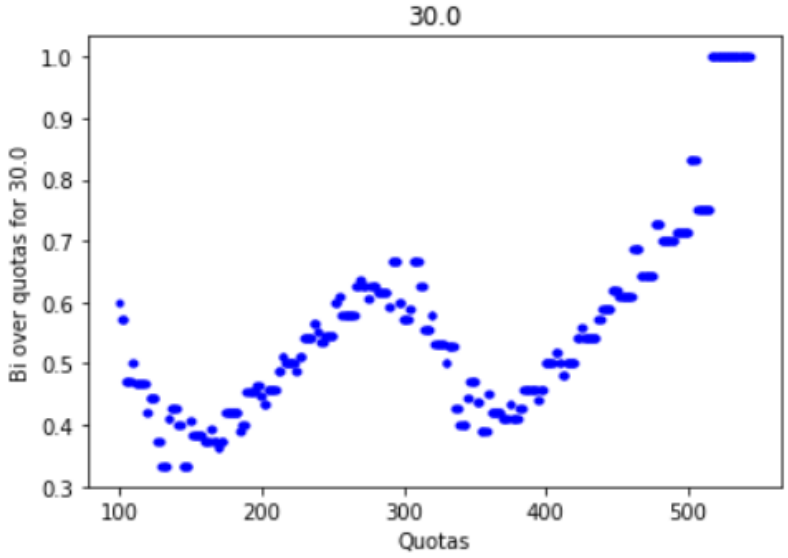

Figure 7: Graph of player 7's voting index β_7 vs q.

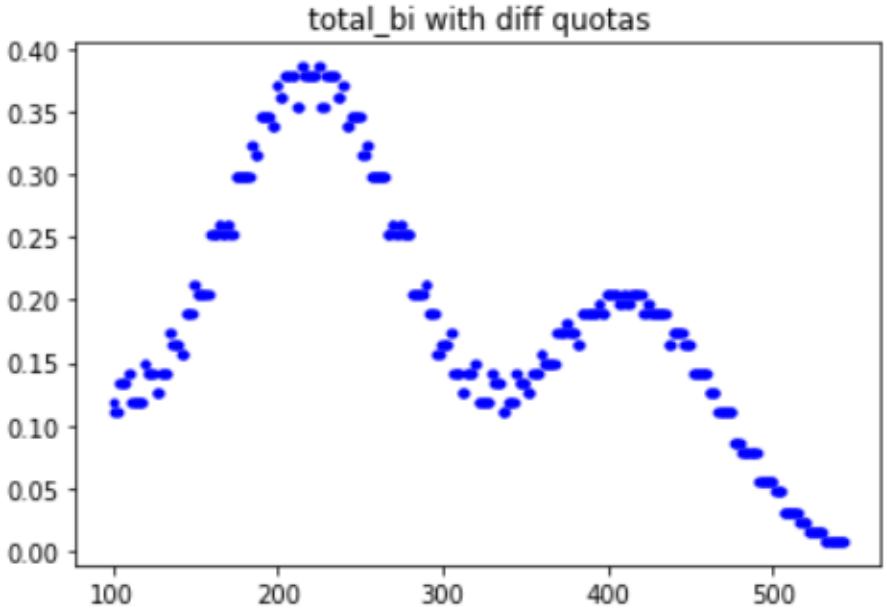

Figure 8: Graph of total voting index β_{total} vs q.

Out[15]:

	A	B	C	D	E	F	G
A	1.0	-1.0	-0.6	0.6	0.3	-0.7	-0.4
B	-1.0	1.0	0.7	0.1	0.1	0.5	-0.4
C	-0.6	0.7	1.0	-0.1	0.1	0.2	-0.4
D	0.6	0.1	-0.1	1.0	0.1	-0.5	-0.4
E	0.3	0.1	0.1	0.1	1.0	0.1	-0.4
F	-0.7	0.5	0.2	-0.5	0.1	1.0	-0.4
G	-0.4	-0.4	-0.4	-0.4	-0.4	-0.4	1.0

Figure 9: Association matrix.

```
Out[16]: [210.0, 170.0, 55.0, 40.0, 30.0, 25.0, 15.0]
```

Figure 10: Weight-Row matrix.

2 Evolutionary algorithms approach

We can use genetic algorithms to solve the same problem. The first step to use genetic algorithms is to model it correctly. In the model, we first make the initial population by picking up n random winning coalitions. Hence, our initial population size is n, and contains a refined pool of genes as we choose from the winning coalitions set instead of the set of all coalitions. Then, we model the DNA. Additionally, we also build the fitness function from the Definition (1.1.3). The agent's fitness in the population is higher if the Left hand side of the above equation comes out to be more negative. If it turns out to be positive, then we take fitness as 0.
The DNA crossover function selects two parents at random from the winning coalitions pool that is generated in the initial population phase and based on the midpoint, we select the members of coalition from the union of the coalition members of both the parents.
We still need to figure out a mutation rate that makes the solution converge within a specific error range. The code for the genetic algorithm model is as follows:

```
import random
import pandas as pd
#Initial population pool of size n is generated which satisfy the winning #coalition
    condition
def ga_winning_coaltions(agent_list, quota, n):
        winning_coaltions = []
        agent_list_enumerated = enumerate(agent_list)
        count = 0
        while count < n:
            num = random.randint(1,len(agent_list))
            coalition = random.sample(agent_list, num)
            if sum(coalition) >= quota:
                winning_coaltions.append(list(coalition))
                count += 1
        return winning_coaltions

#to be compared with quota
def ga_critical_coalition_value(coalition, w, A):
        v = []
        for i in xrange(len(A)):
            v.append(np.dot(w, np.array(A[i])))
        sum_A = sum(v)
        return sum(coalition) - sum_A
#DNA class
class DNA():
    def __init__(self, n):
        weights = map(float, pd.read_excel("weight.xlsx", sheet_name=0).weight)
        self.genes = ga_winning_coaltions(weights, 300, n)

    def fitness(self, agent_list, quota, influence_matrix):
        genes = self.genes
        winning_critical_coaltions = []
        for coalition in genes:
```

```
            fitness_score = ga_critical_coalition_value(coalition, agent_list,
                influence_matrix) - quota
            if fitness_score > 0:
                fitness_score = 0
            winning_critical_coaltions.append(tuple(coalition, fitness_score))
        #self.winning_critical_coaltions = winning_critical_coaltions
        return winning_critical_coaltions


# Crossover function
def DNA_crossover(partner1, partner2):
    weights = len(map(float, pd.read_excel("weight.xlsx", sheet_name=0).weight))
    lesser_partner = min(len(partner1), len(partner2))
    bigger_partner = max(len(partner1), len(partner2))
    midpoint = random.randint(lesser_partner/2, (weights+ bigger_partner)/2)
    # Half from one, half from the other
    final_list = list(set(partner1) | set(partner2))
    child = random.sample(final_list, midpoint)
    return child
    #To print the output: parents and children of the parents
    q = 300
    a = DNA(8)
    # a.fitness(weights, q, list(df))
    starter_gene = a.genes
    parents = random.sample(starter_gene, 2)
    parents
    child = DNA_crossover(parents[0], parents[1])
    child
    child1 = DNA_crossover(parents[0], parents[1])
    child1
```

The output is shown in Figure 11 and Figure 12.

```
Out[449]: [[170.0, 30.0, 25.0, 15.0, 210.0], [170.0, 210.0, 30.0, 25.0, 15.0, 55.0]]
```

Figure 11: Parents.

```
Out[551]: [30.0, 15.0, 210.0, 55.0, 170.0, 25.0]
```

Figure 12: Child.

3 Conclusion

The voting index of all the players tells us about the coalition structure that might arise at a particular quota. This can be seen from the graphs. The evolutionary approach will yield faster and more accurate results once the mortality rate, and mutation rate is decided for larger values of n. Deciding the mutation rate presents the biggest challenge for solving the optimal coalition structure problem. But once that's decided we can ensure that the evolutionary algorithm converges to solution witihn a tolerable error limit.

4 Refrences

[1] Election Commission of India-Last year election data, https://eci.gov.in/.
[2] Nature-inspired programming recepies, http://www.cleveralgorithms.com/nature-inspired/physical/memeticalgorithm.html.
[3] De Jong, K.A, 2006. A evolutionary computation: a unified approach. MIT press.
[4] Howe 1998. The traveling salesrep problem, edge assembly crossover, and 2-opt. Parallel problem solving from Nature V. Springer.
[5] Genetic Algorithm for Coalition Formation, Alan MehlenBacher, April, 2010.