**Discrete spatial competition**

**Abstract**

This paper presents a discrete spatial competition game and proposes a program, written in pseudo-code, to solve the game.

**Problem**

Let represent the dimensionality of spatial competition. Furthermore, let represent the number of producers and represent the number of strategies. Each strategy is a coordinate in . Each producer chooses a strategy . The outcomes of the game are -tuples of the strategies. For example, in a 2-producer 2-strategy game, the outcome where producer plays strategy and producer plays strategy is denoted by .

strategies = Set of strategies e.g. [[0,0], [0,1], [1,0], [1,1]]

producers = Set of producers e.g. [0,1]

outcomes = product(strategies, repeat=producers)

There are individual outcomes to consider. For each outcome there is a corresponding payoff vector which assigns payoffs for each producer. The payoffs depend on the decisions of consumers. Let represent the number of consumers. Each consumer decides the quantity they purchase from each producer based on the utility they derive. The quantity demanded then determines profits which depend on demand, price and cost. Several simplifications to the model are made in this paper. First, utility only depends on distance. Second, each consumer buys an equal quantity. Finally, prices are set to one and costs are set to zero. There are also limitations of the program itself. The program works in a way that assigns equal payoffs to locations that are near and far from consumers. To reduce the demand in locations that are far away from consumers, the quantity demanded is multiplied by a factor representing how far away a location is to the average consumer.

consumers = Set of consumers e.g. [0,1,…,99]

demand = [][]

payoff = [][]

for each tuple i:

distances = [][]

for each consumer j:

utilities = []

maxutility = -

for each producer k:

distance = norm(producers[k], consumers[j])

distances[j][k] = distance

utility = - distance

utilities[k] = utility

if (utility > maxutility):

maxutility = utility

for each producer k:

if (utilities[k] == maxutility):

demand[k] += 1

aggregatedemand = sum(demand)

for each producer j:

avgdistance = mean(distances)

maxdistance = max(distances)

if (maxdistance > 0):

demand[i][j] =

(demand[i][j]/aggregatedemand) \*

(1 – avgdistance / maxdistance)

else:

demand[i][j] = demand[i][j]/aggregatedemand

payoff[i][j] = demand[i][j] \* (1 - 0)

**Solution**

The problem consists of finding the best strategy or strategies for each producer. The solution concepts come from game theory. The game is solved by enumerating all pure strategy Nash equilibria and/or finding the mixed strategy Nash equilibrium.

A pure strategy Nash equilibrium is an outcome which no producer has an incentive to change unilaterally. The following program computes the pure strategy Nash equilibria of the game. For each producer, it computes the set of best response outcomes. To do this, it iterates all outcomes and uses them as pivots to find the best responses to a given set of strategies of the other players. The same outcome can be a best response multiple times so the program eliminates duplicate best response outcomes. Finally, the program decides that an outcome is a Nash equilibrium if it is simultaneously a best response for all producers.

score = []

for each producer i:

bestresponseoutcomes = []

bestresponsepayoffs = []

for each tuple j:

responseoutcome = []

responsepayoff = []

for each tuple k:

validmove = true

for each producer l:

if (i != l):

if (tuples[j][l] != tuples[k][l])

validmove = false

break

if (validmove):

responseoutcome.append(tuples[k])

responsepayoff.append(payoffs[k])

bestresponsepayoff = -

bestresponseindices = []

for each responseoutcome k:

if (responsepayoff[k][i] == bestresponsepayoff):

bestresponseindices.append(k)

elif (responsepayoff[k][i] > bestresponsepayoff):

bestresponseindices = []

bestresponseindices.append(k)

bestresponsepayoff = responsepayoff[k][i]

for each bestresponseindex k:

bestresponseoutcomes.append(

responseoutcome[bestresponseindex[k]]

)

bestresponsepayoffs.append(

responsepayoff[bestresponseindices[k]]

)

uniquebestresponseoutcomes = []

uniquebestresponseoutcomes = []

for each bestresponseoutcome j:

unique = true

for each uniquebestresponseoutcome k:

if (

bestresponseoutcomes[j] == uniquebestresponseoutcomes[k]

):

unique = false

break

if (unique):

uniquebestresponseoutcomes.append(

bestresponseoutcomes[j]

)

for each tuple k:

if (bestresponseoutcomes[j] == tuples[k]):

score[k] = score[k] + 1

break

for each tuple i:

if(score[i] == producers):

print(tuples[i])

A mixed strategy Nash equilibrium is a set of weightings of strategies which no player has an incentive to change unilaterally. The following program computes the mixed-strategy Nash equilibrium of the game. For each producer, it initialises weights as a uniform distribution over the strategies. Then, it repeats the following steps until the program converges. For each producer, it uses a finite difference method to find the gradient of the expected payoff function with respect to the weights of each strategy and adjusts the weights in the direction of the gradient. The result of the optimization is the mixed strategy Nash equilibrium of the game.

weights = [][]

for each producer i:

for each strategy j:

weights[j][i] = 1 / length(strategies)

learningrate = 0.0001

threshold = 1e-8

converged = False

while not converged:

newweights = weights.copy()

for each producer i:

gradient = estimategradient(weights,i)

newweights[:][i] += learningrate\*gradient

converged = True

for each strategy i:

for each producer j:

if (abs(weights[i][j]-newweights[i][j]) > threshold):

converged = False

break

if (converged = False):

break

print(weights)

def mixedpayoff(weights):

tupleweights = []

for each producer i:

for each tuple j:

for each strategy k:

if (strategies[k] == tuples[j][i]):

tupleweights[j] =

weights[k][i] \* tupleweights[j]

mixedpayoffs = [][]

for each producer i:

for each tuple j:

for each strategy k:

if (strategies[k] == tuples[j][i]):

mixedpayoffs[j][i] =

tupleweights[j]\*payoffs[k][i]

return mixedpayoffs

def estimategradient(weights,producer):

gradient = []

step = 0.0001

for each strategy i:

plus = weights.copy()

plus[i][producer] = plus[i][producer]+step

minus = weights.copy()

minus[i][producer] = minus[i][producer]-step

gradient[i] = (mixedpayoff(plus) – mixedpayoff(minus)) / (2\*step)

return gradient