**2η εργασία Οικονομικής Θεωρίας και Αλγορίθμων**

**Ονοματεπώνυμο:** Μαρία-Βασιλική Πετροπούλου **ΑΜ:** 1072540

**Ονοματεπώνυμο:** Ίων-Απόστολος Μπουρνάκας-Δρακόπουλος **ΑΜ:** 1075475

**Θεωρητικό Μέρος**

**ΘΕΩΡΗΤΙΚΟ ΕΡΩΤΗΜΑ ΘΕ1 (Γεννήτρια Παιχνιδιών)**

**ΘΕ1.1:** Ωφέλεια 0 για την παίκτρια γραμμής από το συγκεκριμένο προφίλ σημαίνει ότι στην γραμμή 1 την οποία αυτή παίζει, στις πρώτες k στήλες (δηλαδή στις στήλες μεταξύ των οποίων “μιξάρει” ο παίκτης στήλης) δεν υπάρχει κανένα κελί (1,0), άρα η παίκτρια γραμμής δεν έχει κανένα κέρδος από καμία στήλη. Γνωρίζουμε όμως με βάση τους κανόνες που τηρεί η γεννήτρια κατά την παραγωγή παιχνιδιών ότι σε κάθε στήλη υπάρχει τουλάχιστον ένα (1,0)-κελί. Συνεπώς η παίκτρια γραμμής μπορεί να τα πάει καλύτερα παίζοντας γραμμή/γραμμές όπου υπάρχουν κελιά (1,0) στις πρώτες k στήλες. Άρα υπάρχουν γραμμή/γραμμές που δίνουν θετικό όφελος απέναντι στο και άρα η παίκτρια δεν είναι σε ισορροπία.

**ΘΕ1.2:** Αν η παίκτρια γραμμής έχει ωφέλεια > 0, αυτό σημαίνει ότι τουλάχιστον σε μία από τις πρώτες k στήλες (τις στήλες μεταξύ των οποίων μιξάρει ο παίκτης στήλης) της πρώτης γραμμής υπάρχει κελί (1,0). Γνωρίζουμε όμως με βάση τους κανόνες που τηρεί η γεννήτρια κατά την παραγωγή παιχνιδιών ότι κάθε γραμμή έχει τουλάχιστον ένα κελί (0,1). Συνεπώς, δεδομένου ότι η παίκτρια γραμμής επιλέγει αμιγώς την γραμμή 1, ο παίκτης στήλης μπορεί να αυξήσει το κέρδος του παίζοντας μόνο την στήλη/τις στήλες της πρώτης γραμμής που έχουν κελί/κελιά (0,1). Άρα μπορεί να βρει καλύτερη στρατηγική απέναντι στην στρατηγική = e1 της αντιπάλου, και άρα δεν βρίσκεται σε ισορροπία.

**ΘΕΩΡΗΤΙΚΟ ΕΡΩΤΗΜΑ ΘΕ2 (Εντοπισμός Αυστηρά Κυριαρχούμενων Δράσεων)**

(Αποδεικνύεται μόνο το ΘΕ2.1, το ΘΕ2.2 είναι πλήρως συμμετρικό)

**ΘΕ2.1:** η αμιγής στρατηγική ei για την οποία ισχύει ότι το κέρδος της παίκτριας γραμμής είναι 0 δεν κυριαρχείται αυστηρά από καμία αμιγή στρατηγική, καθώς για να γίνει αυτό θα έπρεπε να υπάρχει γραμμή στο μητρώο R ωφέλειας της παίκτριας όπου κάθε κελί έχει τιμή 1 (αντίστοιχα γραμμή μόνο με κελιά (1,0) στο διμητρώο)). Γνωρίζουμε όμως με βάση τους κανόνες που τηρεί η γεννήτρια κατά την παραγωγή παιχνιδιών πως αποκλείεται να προκύψει τέτοιο διμητρώο, αφού πρέπει κάθε γραμμή να έχει τουλάχιστον ένα (0,1)-κελί (αντίστοιχα το μητρώο R πρέπει σε κάθε γραμμή του να έχει τουλάχιστον ένα μηδενικό στοιχείο).

Θα υπάρχει ωστόσο μεικτή στρατηγική από την οποία κυριαρχείται αυστηρά η ei. Συγκεκριμένα, η ei κυριαρχείται από οποιαδήποτε μεικτή στρατηγική x πληροί και τις 2 παρακάτω προϋποθέσεις:

1. Το support(x) (δηλαδή οι γραμμές μεταξύ των οποίων μιξάρει η παίκτρια γραμμής) περιλαμβάνει μόνο μη μηδενικές γραμμές του R, και
2. Το support(x) περιλαμβάνει μόνο γραμμές του R των οποίων τα μη μηδενικά στοιχεία (δηλαδή τα αντίστοιχα στοιχεία (1,0) στο διμητρώο) βρίσκονται σε μη μηδενικές στήλες του C (μητρώο ωφέλειας του παίκτη στήλης).

Η 2) διασφαλίζει ότι ο παίκτης στήλης θα δώσει θετική μάζα πιθανότητας στην στήλη/στήλες με το μη μηδενικό/ μη μηδενικά στοιχεία της παίκτριας γραμμής, έτσι ώστε η αναμενόμενη ωφέλεια της παίκτριας για αυτήν την γραμμή να είναι πάντα θετική και άρα να κυριαρχεί αυστηρά επί της ei. Για τον ίδιο λόγο απαιτείται και η 1), καθώς η αναμενόμενη ωφέλεια της παίκτριας γραμμής για μηδενική γραμμή θα είναι πάντα 0 ανεξαρτήτως της στρατηγικής του παίκτη στήλης.

**Προγραμματιστικό Μέρος**

Σε αυτό το σημείο θα αρχίσει η παρουσίαση του κώδικα, με την σειρά που αυτός εκτελείται όταν τρέχουμε το πρόγραμμα. Στην αρχή του προγράμματος, αφού εκτυπωθούν στην οθόνη οι κανόνες και συνοπτικά η λειτουργία του, το πρόγραμμά μας τρέχει τις παρακάτω γραμμές κώδικα που είναι υπεύθυνες για τα input του χρήστη και την διαχείρισή τους.

numOfRandomGamesToSolve = 0

while True:

    print("Would you like to load the R and C matrices from a file?")

    choice = input("Enter 'y' for yes or 'n' for no: ")

    if choice == "y":

*# detect if there are already R and C files in the directory*

        if os.path.isfile("R.out") and os.path.isfile("C.out"):

            print("R.out and C.out files detected in path. Would you like to load them?")

            choice = input("Enter 'y' for yes or 'n' for no: ")

            if choice == "y":

                R = np.genfromtxt("R.out", *delimiter*=",", *skip\_header*=0)

                C = np.genfromtxt("C.out", *delimiter*=",", *skip\_header*=0)

                m = R.shape[0]

                n = R.shape[1]

                break

            else:

                print("No or incorrect input detected. So negative answer assumed.")

*# ask the user what is the file name for the R matrix*

        while True:

            Rfilename = input(

                "Enter the name of the file containing the R matrix: ")

*# ask the user what is the file name for the C matrix*

            Cfilename = input(

                "Enter the name of the file containing the C matrix: ")

*# add the .out extension to the file names if not already present*

            if not Rfilename.endswith('.out'):

                Rfilename += '.out'

            if not Cfilename.endswith('.out'):

                Cfilename += '.out'

*# check if the files exist*

            if os.path.isfile(Rfilename) and os.path.isfile(Cfilename):

                break

            else:

                print("The files you provided do not exist. Try again...")

*# load the R and C matrices from the files*

        R = np.genfromtxt(Rfilename, *delimiter*=",", *skip\_header*=0)

        C = np.genfromtxt(Cfilename, *delimiter*=",", *skip\_header*=0)

        m = R.shape[0]

        n = R.shape[1]

        break

    elif choice == "n":

        print("Generating random R and C matrices based on your specifications:")

        maxNumOfRandomGamesToSolve = 10000

        maxNumberOfActions = 20

        m, n = determineGameDimensions()

        G10, G01 = determineNumGoodCellsForPlayers(m, n)

        numOfRandomGamesToSolve = determineNumRandomGamesToSolve()

        earliestColFor01 = 0

        earliestRowFor10 = 0

        break

    else:

        print("Invalid input. Please enter 'y' or 'n'.")

Ο κώδικας αρχικοποιεί την μεταβλητή numOfRandomGamesToSolve με 0 (είναι η μεταβλητή που αποθηκεύει πόσα παιχνίδια επέλεξε ο χρήστης ότι θέλει να λυθούν).

Ο κώδικας εισέρχεται σε έναν άπειρο βρόχο χρησιμοποιώντας μια while True και ρωτάει τον χρήστη εάν θέλει να φορτώσει τους πίνακες R και C από ένα αρχείο (στο path του χρήστη). Ο χρήστης μπορεί να εισάγει «y» για ναι ή «n» για όχι.

Εάν ο χρήστης επιλέξει να φορτώσει τους πίνακες από ένα αρχείο (επιλογή == "y"), ο κώδικας ελέγχει εάν τα αρχεία R.out και C.out υπάρχουν στον τρέχοντα κατάλογο. Εάν υπάρχουν και τα δύο αρχεία, ρωτά τον χρήστη εάν θέλει να τα φορτώσει (επιλογή == "y"). Εάν ο χρήστης επιλέξει να φορτώσει τα αρχεία, ο κώδικας διαβάζει τους πίνακες από τα αρχεία χρησιμοποιώντας την np.genfromtxt() και τους εκχωρεί στις μεταβλητές R και C. Ύστερα ελέγχονται οι διαστάσεις των R και C. Αν βρεθεί ότι είναι ίδιες ο βρόχος τερματίζει με χρήση break και το πρόγραμμα συνεχίζει, αλλιώς ο βρόχος επαναλαμβάνεται. Οι διαστάσεις των πινάκων (m και n) καθορίζονται με βάση το σχήμα των 2 πινάκων.

Εάν ο χρήστης εισαγάγει «n» ή οποιαδήποτε άλλη μη έγκυρη είσοδο, υποδεικνύοντας ότι δεν θέλει να φορτώσει τα υπάρχοντα αρχεία (R.out, C.out), ο κώδικας θα θεωρήσει αρνητική απάντηση και θα συνεχίσει στο επόμενο βήμα.

Στο επόμενο βήμα ζητείται από τον χρήστη να εισαγάγει τα ονόματα αρχείων για τους πίνακες R και C. Ο κώδικας θα συνεχίσει να ζητά τα ονόματα αρχείων μέχρι να δοθούν έγκυρα ονόματα αρχείων και να υπάρχουν και τα δύο αρχεία στον κατάλογο. Το πρόγραμμα δέχεται τα αρχεία είτε με το full name τους (δηλαδή με την .out κατάληξη) είτε χωρίς. Σε περίπτωση που λείπει η κατάληξη το πρόγραμμα την προσθέτει από μόνο του.

Μόλις εισαχθούν έγκυρα ονόματα αρχείων και εφόσον υπάρχουν και τα δύο αρχεία, ο κώδικας θα διαβάσει τους πίνακες από τα αρχεία χρησιμοποιώντας την np.genfromtxt() και θα τους αντιστοιχίσει στις μεταβλητές R και C. Ύστερα ελέγχονται οι διαστάσεις των R και C. Αν βρεθεί ότι είναι ίδιες ο βρόχος τερματίζει με χρήση break και το πρόγραμμα συνεχίζει, αλλιώς ο βρόχος επαναλαμβάνεται. Οι διαστάσεις των πινάκων (m και n) καθορίζονται με βάση το σχήμα των 2 πινάκων. Στη συνέχεια, ο κώδικας εξέρχεται από τον βρόχο με τη χρήση break και το πρόγραμμα προχωρά με τους φορτωμένους πίνακες για περαιτέρω επεξεργασία.

Εάν ο χρήστης επιλέξει να μην φορτώσει τους πίνακες από ένα αρχείο (επιλογή == "n"), ο κώδικας προχωρά στη δημιουργία τυχαίων πινάκων R και C με βάση τις προδιαγραφές του χρήστη. Προτρέπει τον χρήστη να εισαγάγει τις διαστάσεις του παιχνιδιού (m και n) και τον αριθμό των καλών κελιών για τους παίκτες (G10 και G01). Καθορίζει επίσης τον αριθμό των τυχαίων παιχνιδιών προς επίλυση (numOfRandomGamesToSolve) με βάση τα δεδομένα του χρήστη. Στη συνέχεια, ο βρόχος παύει με χρήση διακοπής.

Εάν ο χρήστης εισαγάγει μια μη έγκυρη είσοδο, δηλαδή ούτε «y» ούτε «n», εμφανίζεται ένα μήνυμα σφάλματος και ο βρόχος συνεχίζεται μέχρι να δοθεί έγκυρη είσοδος.

Συνολικά, αυτός ο κώδικας χειρίζεται την είσοδο του χρήστη για να καθορίσει εάν θα φορτωθούν πίνακες από αρχεία ή αν θα δημιουργηθούν τυχαίοι πίνακες. Πραγματοποιεί επίσης την ανάγνωση πινάκων από αρχεία (εφόσον χρειαστεί) και τις εκχωρεί σε μεταβλητές για περαιτέρω επεξεργασία. Σημειώνεται πως στην περίπτωση φόρτωσης πινάκων δεν πραγματοποιείται έλεγχος και για τα περιεχόμενά τους, καθώς γίνεται παραδοχή πως αυτά θα έχουν πάντα μόνο επιτρεπτές τιμές (0 και 1), και άρα είναι ευθύνη του χρήστη να διασφαλίσει ότι φορτώνει σωστά αρχεία.

games = 0

flag = 0

flag1 = 0

choice1 = 0

if (numOfRandomGamesToSolve > 1):

    print("Mass experiment detected. Skipping save to path question and limiting outputs for easier understanding"

          " of results.")

    registerE\_DMP= {

        "[0,0.1)": 0,

        "[0.1,0.2)": 0,

        "[0.2,0.3)": 0,

        "[0.3,0.4)": 0,

        "[0.4,0.5)": 0,

        "[0.5,0.6)": 0,

        "[0.6,0.7)": 0,

        "[0.7,0.8)": 0,

        "[0.8,0.9)": 0,

        "[0.9,1.0]": 0

    }

    registerW\_DMP = {

        "[0,0.1)": 0,

        "[0.1,0.2)": 0,

        "[0.2,0.3)": 0,

        "[0.3,0.4)": 0,

        "[0.4,0.5)": 0,

        "[0.5,0.6)": 0,

        "[0.6,0.7)": 0,

        "[0.7,0.8)": 0,

        "[0.8,0.9)": 0,

        "[0.9,1.0]": 0

    }

    registerE\_DEL = {

        "[0,0.1)": 0,

        "[0.1,0.2)": 0,

        "[0.2,0.3)": 0,

        "[0.3,0.4)": 0,

        "[0.4,0.5)": 0,

        "[0.5,0.6)": 0,

        "[0.6,0.7)": 0,

        "[0.7,0.8)": 0,

        "[0.8,0.9)": 0,

        "[0.9,1.0]": 0

    }

    registerW\_DEL = {

        "[0,0.1)": 0,

        "[0.1,0.2)": 0,

        "[0.2,0.3)": 0,

        "[0.3,0.4)": 0,

        "[0.4,0.5)": 0,

        "[0.5,0.6)": 0,

        "[0.6,0.7)": 0,

        "[0.7,0.8)": 0,

        "[0.8,0.9)": 0,

        "[0.9,1.0]": 0

    }

    registerE\_FPPBR = {

        "[0,0.1)": 0,

        "[0.1,0.2)": 0,

        "[0.2,0.3)": 0,

        "[0.3,0.4)": 0,

        "[0.4,0.5)": 0,

        "[0.5,0.6)": 0,

        "[0.6,0.7)": 0,

        "[0.7,0.8)": 0,

        "[0.8,0.9)": 0,

        "[0.9,1.0]": 0

    }

    registerW\_FPPBR = {

        "[0,0.1)": 0,

        "[0.1,0.2)": 0,

        "[0.2,0.3)": 0,

        "[0.3,0.4)": 0,

        "[0.4,0.5)": 0,

        "[0.5,0.6)": 0,

        "[0.6,0.7)": 0,

        "[0.7,0.8)": 0,

        "[0.8,0.9)": 0,

        "[0.9,1.0]": 0

    }

    registerE\_FPUN = {

        "[0,0.1)": 0,

        "[0.1,0.2)": 0,

        "[0.2,0.3)": 0,

        "[0.3,0.4)": 0,

        "[0.4,0.5)": 0,

        "[0.5,0.6)": 0,

        "[0.6,0.7)": 0,

        "[0.7,0.8)": 0,

        "[0.8,0.9)": 0,

        "[0.9,1.0]": 0

    }

    registerW\_FPUN = {

        "[0,0.1)": 0,

        "[0.1,0.2)": 0,

        "[0.2,0.3)": 0,

        "[0.3,0.4)": 0,

        "[0.4,0.5)": 0,

        "[0.5,0.6)": 0,

        "[0.6,0.7)": 0,

        "[0.7,0.8)": 0,

        "[0.8,0.9)": 0,

        "[0.9,1.0]": 0

    }

    DMPepsAPPROX\_list = []

    DELepsAPPROX\_list = []

    FPPBRepsAPPROX\_list = []

    FPUNepsAPPROX\_list = []

    R\_list = []

    C\_list = []

    flag = 1

if (numOfRandomGamesToSolve == 0):

    flag1 = 1

    numOfRandomGamesToSolve = 1

Αυτός ο κώδικας δημιουργεί μεταβλητές και δομές δεδομένων ανάλογα με την τιμή του numOfRandomGamesToSolve.

1. Αρχικοποιεί τις μεταβλητές games, flag, flag1 και choice1 με 0 .
2. Εάν το numOfRandomGamesToSolve είναι μεγαλύτερο από 1, υποδεικνύοντας μαζικό πείραμα, ο κώδικας συνεχίζει με τα ακόλουθα βήματα:

* Εκτυπώνει ένα μήνυμα που υποδεικνύει την ανίχνευση μαζικού πειράματος και εξηγεί ότι θα παραλείψει την ερώτηση σχετικά με αποθήκευση σε διαδρομή και θα περιορίσει τις εξόδους για ευκολότερη κατανόηση των αποτελεσμάτων.
* Αρχικοποιεί πολλά λεξικά (registerE\_DMP, registerW\_DMP, registerE\_DEL, registerW\_DEL, registerE\_FPPBR, registerW\_FPPBR, registerE\_FPUN, registerW\_FPUN) με κλειδιά που αντιπροσωπεύουν διαφορετικά εύρη τιμών ([0,0.1), [0.1,0.2), [0.2,0.3),… [0.8,0.9), [0.9,1.0]) και οι αρχικές τιμές του κάθε εύρους ορίζονται 0. Αυτά τα λεξικά χρησιμοποιούνται για την αποθήκευση της συχνότητας εμφάνισης των ε σε κάθε εύρος τιμών κατά τη διάρκεια του πειράματος.
* Αρχικοποιεί πολλές λίστες (DMPepsAPPROX\_list, DELepsAPPROX\_list, FPPBRepsAPPROX\_list, FPUNepsAPPROX\_list, R\_list, C\_list) ως κενές λίστες. Αυτές χρησιμοποιούνται για την αποθήκευση των αντίστοιχων τιμών κατά τη διάρκεια του πειράματος (για τα epsApprox του κάθε αλγορίθμου έτσι ώστε να μπορούμε να βρούμε τα δυο χειροτέρα παιχνίδια και τα αντίστοιχα R και C των παιχνιδιών στους αντίστοιχους πίνακες).
* Ορίζει τη σημαία flag σε 1 για να υποδείξει ότι διεξάγεται ένα μαζικό πείραμα.

1. Εάν το numOfRandomGamesToSolve είναι 0, υποδεικνύοντας μεμονωμένο παιχνίδι, ο κώδικας ορίζει το flag1 σε 1 και ενημερώνει το numOfRandomGamesToSolve σε 1.

Ο σκοπός αυτών των βημάτων είναι να ρυθμιστούν οι απαραίτητες μεταβλητές, δομές δεδομένων και σημαίες με βάση την τιμή του numOfRandomGamesToSolve προκειμένου να χειριστούν είτε μαζικά πειράματα είτε μεμονωμένα παιχνίδια στα επόμενα τμήματα του προγράμματος.

Στο επόμενο μέρος του κώδικα έχουμε μια μεγάλη while η οποία είναι υπεύθυνη τόσο για την εκτέλεση των πειραμάτων και την παραγωγή των απαραίτητων αποτελεσμάτων όσο και για τα μεμονωμένα παιχνίδια. Το αρχικό μέρος αυτής της while είναι υπεύθυνο για την επιλογή του χρήστη σχετικά με ποιον αλγόριθμο θα τρέξει, καθώς και την αποθήκευση των R,C μητρώων, η οποία γίνεται υπό συνθήκη. Αυτή η συνθήκη είναι αρχικά ο χρήστης να μην έχει κάνει φορτώσει ήδη από τον φάκελο τα μητρώα R, C, γιατί τότε προφανώς η αποθήκευση των ήδη υπαρχόντων αρχείων είναι αχρείαστη. Δεύτερη συνθήκη για την αποθήκευση των μητρώων είναι η μη διεξαγωγή πειράματος, αφού διαφορετικά αν θέλαμε να διεξάγουμε ένα πείραμα μεγάλου μεγέθους θα έπρεπε σε κάθε δημιουργία μητρώων R,C να διακόπτεται το πρόγραμμα και να περιμένουμε την επιλογή του χρήστη για αποθήκευση η όχι.

if (flag1 == 0):

        EXITCODE = -5

        numOfAttempts = 0

*# TRY GETTING A NEW RANDOM GAME*

*# REPEAT UNTIL EXITCODE = 0, ie, a valid game was constructed.*

*# NOTE: EXITCODE in {-1,-2,-3} indicates invalid parameters and exits the program)*

        while EXITCODE < 0:

*# EXIT CODE = -4 ==> No problem with parameters, only BAD LUCK, TOO MANY 01-elements within 10-eligible area*

*# EXIT CODE = -5 ==> No problem with parameters, only BAD LUCK, ALL-01 column exists within 10-eligible area*

            numOfAttempts += 1

            print("Attempt #" + str(numOfAttempts) +

                  " to construct a random game...")

            EXITCODE, R, C = generate\_winlose\_game\_without\_pne(

                m, n, G01, G10, earliestColFor01, earliestRowFor10)

            if EXITCODE in [-1, -2, -3]:

                print(bcolors.ERROR + "ERROR MESSAGE MAIN 1: Invalid parameters were provided for the construction of the random game." + bcolors.ENDC)

                exit()

    if (flag == 0 and flag1 == 0):

        print("Would you like to save the R and C matrices to a file?")

        choice = input("Enter 'y' for yes or 'n' for no: ")

        if choice == "y":

            while True:

*# ask the user what is the file name for the R matrix*

                Rfilename = input(

                    "Enter the name of the file to save the R matrix: ")

*# ask the user what is the file name for the C matrix*

                Cfilename = input(

                    "Enter the name of the file to save the C matrix: ")

*# check if the user put valid file names*

                if Rfilename != "" and Cfilename != "" and Rfilename != Cfilename:

                    break

                else:

                    print("Invalid file names. Try again...")

*# add the .out extension to the file names if not already present*

            if not Rfilename.endswith('.out'):

                Rfilename += '.out'

            if not Cfilename.endswith('.out'):

                Cfilename += '.out'

*# save the R and C matrices to the files*

            np.savetxt(Rfilename, R, *delimiter*=',', *fmt*='%1.4e')

            np.savetxt(Cfilename, C, *delimiter*=',', *fmt*='%1.4e')

        else:

            print("No or incorrect input detected. So negative answer assumed.")

        drawBimatrix(m, n, R, C)

    if (flag1 == 1):

        drawBimatrix(m, n, R, C)

*# SEEKING FOR PNE IN THE GAME (R,C)...*

    (i, j) = checkForPNE(m, n, R, C)

    if (i, j) != (0, 0):

        print(bcolors.MSG + "A pure NE (", i, ",", j,

              ") was discovered for (R,C)." + bcolors.ENDC)

        exit()

    else:

        print(bcolors.MSG + "No pure NE exists for (R,C). Looking for an approximate NE point..." + bcolors.ENDC)

    reduced\_m, reduced\_n, reduced\_R, reduced\_C, dominated\_rows, dominated\_cols = removeStrictlyDominatedStrategies(

        m, n, R, C)

    if (flag == 0 or flag1 == 1):

        print(bcolors.MSG +

              "Reduced bimatrix, after removal of strictly dominated actions:")

        drawBimatrix(reduced\_m, reduced\_n, reduced\_R, reduced\_C)

    if (choice1 == 0):

        while True:

*# Prompt the user to choose which algorithm to run*

            print("Which algorithm would you like to run?")

            print("1. DMP Algorithm")

            print("2. 𝐷𝐸𝐿 Algorithm")

            print("3. 𝐹𝑃𝑃𝐵𝑅 Algorithm")

            print("4. 𝐹𝑃𝑈𝑁𝐼𝐹𝑂𝑅𝑀 Algorithm")

            print("5. All of the above")

            choice1 = input("Enter your choice (1/2/3/4/5): ")

            if choice1 in ["1", "2", "3", "4", "5"]:

                break

            else:

                print("Invalid input. Try again...")

Αυτός ο κώδικας αποτελείται από πολλά μπλοκ υπό όρους και βρόχους. Ας αναλύσουμε κάθε μέρος:

1. Εάν το flag1 είναι 0 (δηλαδή ο χρήστης δεν φόρτωσε τα μητρώα από αρχείο):

* Το EXITCODE αρχικοποιείται σε -5.
* Το numOfAttempts έχει οριστεί σε 0.
* Ένας βρόχος while ξεκινά με συνθήκη EXITCODE < 0 για να προσπαθήσει να δημιουργηθεί ένα τυχαίο παιχνίδι μέχρι να κατασκευαστεί ένα έγκυρο παιχνίδι.
* Μέσα στον βρόχο, ο αριθμός των προσπαθειών αυξάνεται και εκτυπώνεται ένα μήνυμα που υποδεικνύει τον αριθμό προσπάθειας.
* Η συνάρτηση generate\_winlose\_game\_without\_pne καλείται με τις παρεχόμενες παραμέτρους (m, n, G01, G10, earliestColFor01, earliestRowFor10) για τη δημιουργία του παιχνιδιού.
* Εάν το EXITCODE είναι -1, -2 ή -3, υποδεικνύοντας μη έγκυρες παραμέτρους, εκτυπώνεται ένα μήνυμα σφάλματος και το πρόγραμμα τερματίζεται.

1. Εάν η flag είναι 0 και η flag1 είναι 0 (στην περίπτωση που έχουμε δημιουργήσει με το πρόγραμμα τους R,C και δεν διεξάγεται πείραμα):

* Εκτυπώνεται ένα μήνυμα που ρωτά τον χρήστη εάν θέλει να αποθηκεύσει τους πίνακες R και C σε ένα αρχείο.
* Λαμβάνεται η επιλογή του χρήστη ("y" για ναι ή "n" για όχι).
* Εάν η επιλογή είναι "y", ξεκινά ένας βρόχος while για να ζητήσει από τον χρήστη έγκυρα ονόματα αρχείων για τους πίνακες R και C.
* Οι πίνακες R και C αποθηκεύονται στα καθορισμένα αρχεία χρησιμοποιώντας τη συνάρτηση np.savetxt.

1. Εάν το flag1 είναι 1 (στην περίπτωση που ο χρηστης φόρτωσε τα μητρώα από αρχείο):

* Η συνάρτηση drawBimatrix καλείται να εμφανίσει το διμητρώο χρησιμοποιώντας τους πίνακες R και C.

1. Η συνάρτηση checkForPNE καλείται με όρισμα τους πίνακες R και C για έλεγχο ύπαρξης αμιγούς ισορροπίας Nash (PNE). Εάν βρεθεί PNE, εκτυπώνεται μήνυμα με αυτήν και το πρόγραμμα τερματίζει.
2. Εάν δεν βρεθεί PNE:

* Η συνάρτηση removeStrictlyDominatedStrategies καλείται με ορίσματα τους πίνακες R και C για να παραχθεί μειωμένο διμητρώο, αφαιρώντας τις αυστηρά κυριαρχούμενες δράσεις και από τους 2 πίνακες, εάν αυτές υπάρχουν. Εάν δεν υπάρχουν, τότε το αρχικό διμητρώο μένει ως έχει.
* Εάν η flag είναι 0 ή η flag1 είναι 1, το διμητρώο εμφανίζεται χρησιμοποιώντας τη συνάρτηση drawBimatrix.

1. Εάν η choice1 είναι 0 (μεταβλητή που έχουμε ορίσει αρχικά 0 και είναι υπεύθυνη να αποθηκεύει την επιλογή του χρήστη για τον ποιο αλγόριθμο θα τρέξει):

* Ένας βρόχος while ξεκινά για να ζητήσει από τον χρήστη να επιλέξει ποιον αλγόριθμο θα εκτελεστεί.
* Εμφανίζονται οι διαθέσιμες επιλογές και λαμβάνεται η επιλογή του χρήστη.

Συνολικά αυτός ο κώδικας περιλαμβάνει την δημιουργία τυχαίων παιχνιδιών, την αποθήκευση πινάκων σε αρχεία, τον έλεγχο για αμιγείς ισορροπίες Nash, την κατάργηση στρατηγικών που κυριαρχούνται και την δυνατότητα ο χρήστης να επιλέξει ποιον αλγόριθμο θα εκτελέσει.

Τώρα ας εξηγήσουμε την removeStrictlyDominatedStrategies που κατασκευάστηκε.

def removeStrictlyDominatedStrategies(*m*, *n*, *R*, *C*):

    print(bcolors.IMPLEMENTED + '''

    ROUTINE: removeStrictlyDominatedStrategies ''' + bcolors.ENDC)

*# PRE:    A win-lose bimatrix game, described by the two payoff matrices, with payoff values in {0,1}.*

*# POST:   The subgame constructed by having all strictly dominated actions removed.*

*#          ''' + bcolors.ENDC)*

*# Remove strictly dominated rows*

    check = 0

    dominated\_rows = set()

    dominated\_cols = set()

    while check == 0:

        num\_removals = 0

        for i in range(*m*):

            if i not in dominated\_rows and all(*R*[i][k] <= *R*[j][k] for k in range(*n*) for j in range(*m*) if j != i):

                if not all(*R*[i][k] == *R*[j][k] for k in range(*n*) for j in range(*m*) if j != i):

*# if the two rows aren't exactly the same (to avoid erasing weakly dominated rows)*

                    dominated\_rows.add(i)

                    num\_removals += 1

                    for z in range(*n*):

*R*[i][z] = 0

*C*[i][z] = 0

*# Remove strictly dominated columns*

        for j in range(*n*):

            if j not in dominated\_cols and all(*C*[i][j] <= *C*[i][k] for k in range(*n*) for i in range(*m*)):

                if not all(*C*[i][j] == *C*[i][k] for k in range(*n*) for i in range(*m*)):

*# if the two columns aren't exactly the same (to avoid erasing weakly dominated columns)*

                    dominated\_cols.add(j)

                    num\_removals += 1

                    for z in range(*m*):

*R*[z][j] = 0

*C*[z][j] = 0

*# Check if any row or column was deleted in this iteration, if not then reduction is complete*

        if num\_removals == 0:

            check = 1

*# delete strictly dominated rows*

    reduced\_R = np.delete(*R*, list(dominated\_rows), *axis*=0)

    reduced\_C = np.delete(*C*, list(dominated\_rows), *axis*=0)

    reduced\_m = reduced\_R.shape[0]

*# delete strictly dominated columns*

    reduced\_C = np.delete(reduced\_C, list(dominated\_cols), *axis*=1)

    reduced\_R = np.delete(reduced\_R, list(dominated\_cols), *axis*=1)

    reduced\_n = reduced\_C.shape[1]

    return reduced\_m, reduced\_n, reduced\_R, reduced\_C, dominated\_rows, dominated\_cols

Αυτός ο κώδικας ορίζει μια συνάρτηση που ονομάζεται removeStrictlyDominatedStrategies, η οποία λαμβάνει ως είσοδο τις διαστάσεις ενός παιχνιδιού δισδιάστατου πίνακα (m και n) και τους πίνακες ωφέλειας (R και C). Η συνάρτηση αφαιρεί από το παιχνίδι στρατηγικές που κυριαρχούνται αυστηρά, και επιστρέφει το μειωμένο παιχνίδι (τα μειωμένα μητρώα και τις νέες διαστάσεις τους) μαζί με τους δείκτες των γραμμών και στηλών που κυριαρχούνται.

Ακολουθεί ανάλυση του κώδικα:

1. Δύο σύνολα, dominated\_rows και dominated\_cols, αρχικοποιούνται για να αποθηκεύουν τους δείκτες των κυριαρχούμενων γραμμών και στηλών αντίστοιχα.
2. Ένας βρόχος while χρησιμοποιείται για επανάληψη του κώδικα μέχρι να μην μπορεί να γίνει άλλη απαλοιφή γραμμής ή στήλης. Στην αρχή κάθε επανάληψης του βρόχου, γίνεται αρχικοποίηση της μεταβλητής num\_removals με τιμή 0.
3. Μέσα στον βρόχο, ο αλγόριθμος ελέγχει για αυστηρά κυριαρχούμενες σειρές:

* Για κάθε γραμμή i στο εύρος m (πλήθος γραμμών) και κάθε k στο εύρος n (πλήθος στηλών), ελέγχει εάν το i δεν βρίσκεται ήδη στην dominated\_rows και εάν όλες οι τιμές στην γραμμή i είναι μικρότερες ή ίσες από τις αντίστοιχες τιμές σε όλες τις άλλες γραμμές (j != i) στην ίδια στήλη.
* Εάν η συνθήκη ικανοποιείται και οι γραμμές που συγκρίνονται δεν είναι ακριβώς οι ίδιες, η γραμμή i θεωρείται κυριαρχούμενη.
* Η κυριαρχούμενη γραμμή i προστίθεται στην dominated\_rows και τα κελιά της παίρνουν τιμή 0 στους πίνακες R και C (ώστε να μην επηρεάζουν τους ελέγχους κυριαρχίας σε τυχόν επόμενες επαναλήψεις του βρόχου). Επιπλέον η μεταβλητή num\_removals αυξάνεται κατά ένα.

1. Αφού ελέγξει για κυριαρχούμενες γραμμές, ο αλγόριθμος προχωρά στον έλεγχο για κυριαρχούμενες στήλες:

* Για κάθε στήλη j στο εύρος n και κάθε γραμμή i στο εύρος m, ελέγχει εάν το j δεν είναι ήδη στην dominated\_cols και εάν όλες οι τιμές στην στήλη j είναι μικρότερες ή ίσες από τις αντίστοιχες τιμές σε όλες τις άλλες στήλες στην ίδια γραμμή.
* Εάν η συνθήκη ικανοποιείται και οι στήλες δεν είναι ακριβώς οι ίδιες, η στήλη j θεωρείται κυριαρχούμενη.
* Η κυριαρχούμενη στήλη j προστίθεται στην dominated\_cols και τα κελιά της παίρνουν τιμή 0 στους πίνακες R και C (ώστε να μην επηρεάζουν τους ελέγχους κυριαρχίας σε τυχόν επόμενες επαναλήψεις του βρόχου). Επιπλέον η μεταβλητή num\_removals αυξάνεται κατά ένα.

1. Μετά την ολοκλήρωση των επαναλήψεων, ο αλγόριθμος ελέγχει εάν κάποια γραμμή ή στήλη διαγράφηκε στην τρέχουσα επανάληψη με την βοήθεια της μεταβλητής num\_removals. Εάν δεν έγιναν διαγραφές, δηλαδή αν η num\_removals έχει τιμή 0, η διαδικασία μείωσης ολοκληρώνεται και ο βρόχος τερματίζει, καθώς η μεταβλητή check γίνεται 1 και η συνθήκη παύει να πληρείται.
2. Οι κυριαρχούμενες γραμμές διαγράφονται από τους πίνακες ωφέλειας R και C χρησιμοποιώντας τη συνάρτηση np.delete και οι μειωμένοι πίνακες αποθηκεύονται στους reduced\_R και reduced\_C, αντίστοιχα.
3. Ομοίως, οι στήλες που κυριαρχούνται διαγράφονται από τους μειωμένους πίνακες, με αποτέλεσμα το reduced\_R και reduced\_C να ενημερώνονται περαιτέρω. Ενημερώνονται επίσης οι μεταβλητές reduced\_m και reduced\_n.
4. Οι διαστάσεις του μειωμένου παιχνιδιού (reduced\_m και reduced\_n), οι reduced\_R και reduced\_C και οι δείκτες των κυριαρχούμενων γραμμών και στηλών επιστρέφονται ως έξοδος της συνάρτησης.

Συνολικά, αυτή η συνάρτηση εφαρμόζει έναν αλγόριθμο για την αφαίρεση στρατηγικών που κυριαρχούνται αυστηρά από ένα διμητρικό παιχνίδι. Προσδιορίζει και εξαλείφει επαναληπτικά τις γραμμές και στήλες που κυριαρχούνται, μέχρι να μην είναι δυνατή περαιτέρω μείωση.

Τώρα ας εξηγήσουμε την checkForPNE που κατασκευάστηκε.

def checkForPNE(*m*, *n*, *R*, *C*):

    print(bcolors.IMPLEMENTED + '''

    ROUTINE: checkForPNE '''

          + bcolors.ENDC)

*# PRE:    Two mxn payoff matrices R,C, with real values (not necessarily in [0,1])*

*# METHOD: Checking for the existence of a pure Nash equilibrium (PNE).*

*# POST:   (0,0) if no pure NE exists for (R, C), or else a pair of actions (i, j) that constitute a pure NE.*

*# ''' + bcolors.ENDC)*

    poss\_row = []

    poss\_col = []

    max\_of\_each\_row = np.max(*R*, *axis*=0)  *# finds the max value of each  row*

    max\_of\_each\_col = np.max(*C*, *axis*=1)  *# finds the max value of each  column*

*# finds the indexes of all cells with the max row value*

    for i in range(*m*):

        for j in range(*n*):

            if *R*[i][j] == max\_of\_each\_row[i]:

                poss\_row.append((i, j))

*# finds the indexes of all cells with the max column value*

    for j in range(*n*):

        for i in range(*m*):

            if *C*[i][j] == max\_of\_each\_col[i]:

                poss\_col.append((i, j))

*# checks if any cells have max value for both row and column, if yes it returns the first it finds as a PNE*

    for i in range(0, len(poss\_row)):

        for j in range(0, len(poss\_col)):

            if poss\_row[i] == poss\_col[j]:

                return (i, j)

    return (0, 0)

Αυτός ο κώδικας ορίζει μια συνάρτηση που ονομάζεται checkForPNE η οποία λαμβάνει ως είσοδο τις διαστάσεις (m και n) και τους πίνακες ωφέλειας (R και C) ενός δισδιάστατου παιχνιδιού. Η συνάρτηση ελέγχει την ύπαρξη αμιγούς ισορροπίας Nash (PNE) στο συγκεκριμένο παιχνίδι.

Ακολουθεί ανάλυση του κώδικα:

1. Μια κενή λίστα poss\_row ορίζεται για την αποθήκευση των κελιών του μητρώου R που έχουν τιμή ίση με την μέγιστη τιμή της γραμμής στην οποία βρίσκονται, και αντίστοιχα μια κενή λίστα poss\_col ορίζεται για την αποθήκευση των κελιών του μητρώου C που έχουν τιμή ίση με την μέγιστη τιμή της στήλης στην οποία βρίσκονται.
2. Η μέγιστη τιμή κάθε γραμμής στον πίνακα R υπολογίζεται χρησιμοποιώντας την συνάρτηση np.max με axis=0, η οποία επιστρέφει έναν πίνακα που περιέχει την μέγιστη τιμή κάθε γραμμής του R.
3. Η μέγιστη τιμή κάθε στήλης στον πίνακα C υπολογίζεται χρησιμοποιώντας την συνάρτηση np.max με axis=1, η οποία επιστρέφει έναν πίνακα που περιέχει την μέγιστη τιμή κάθε στήλης του C.
4. Δύο βρόχοι χρησιμοποιούνται για να ελεγχθούν τα στοιχεία του πίνακα R και να βρεθούν τα κελιά που έχουν τιμή ίση με την μέγιστη τιμή της γραμμής στην οποία βρίσκονται. Οι δείκτες αυτών των κελιών προστίθενται στην λίστα poss\_row.
5. Δύο βρόχοι χρησιμοποιούνται για να ελεγχθούν τα στοιχεία του πίνακα C και να βρεθούν τα κελιά που έχουν τιμή ίση με την μέγιστη τιμή της στήλης στην οποία βρίσκονται. Οι δείκτες αυτών των κελιών προστίθενται στην λίστα poss\_col.
6. Στη συνέχεια, η συνάρτηση ελέγχει εάν υπάρχουν αντίστοιχα κελιά (κελιά με ίδιους δείκτες) που έχουν τη μέγιστη τιμή της γραμμής τους στο R και την μέγιστη τιμή της στήλης τους στο C. Εάν υπάρχουν τέτοια κελιά, η συνάρτηση επιστρέφει τους δείκτες του πρώτου τέτοιου κελιού που βρέθηκε ως αμιγή ισορροπία Nash (PNE).
7. Εάν δεν βρεθεί PNE, η συνάρτηση επιστρέφει (0, 0) για να υποδείξει την απουσία PNE στο δεδομένο παιχνίδι.

Συνολικά, αυτή η συνάρτηση ελέγχει την ύπαρξη Pure Nash Equilibrium, βρίσκοντας κελιά με τις μέγιστες τιμές τόσο στη γραμμή όσο και στη στήλη. Επιστρέφει τους δείκτες του πρώτου κελιού που βρέθηκε ως PNE ή (0, 0) εάν δεν βρεθεί PNE.

Τώρα ας εξηγήσουμε την interpretReducedStrategiesForOriginalGame που κατασκευάστηκε.

def interpretReducedStrategiesForOriginalGame(*reduced\_x*, *reduced\_y*, *R*, *C*, *reduced\_R*, *reduced\_C*, *dominated\_rows*, *dominated\_cols*):

*# print(bcolors.IMPLEMENTED + '''*

*# ROUTINE:    interpretReducedStrategiesForOriginalGame ''' + bcolors.ENDC)*

*# PRE:        A profile of strategies (reduced\_x,reduced\_y) for the reduced*

*#             game (reduced\_R,reduced\_C), without (0,\*)-rows or (\*,0)-columns.*

*# POST:       The corresponding profile for the original game (R,C).*

*# ''' + bcolors.ENDC)*

    x = np.zeros(*R*.shape[0])

    y = np.zeros(*C*.shape[1])

    i = 0

    j = 0

    check = 0

    if (*reduced\_R*.shape[0] == *R*.shape[0] and *reduced\_R*.shape[1] == *R*.shape[1] and *reduced\_C*.shape[0] == *C*.shape[0] and *reduced\_C*.shape[1] == *C*.shape[1]):

        x = *reduced\_x*

        y = *reduced\_y*

    else:

        if (len(*reduced\_x*) == *R*.shape[0]):

            x = *reduced\_x*

        else:

            for i in range(len(x)):

                if i not in *dominated\_rows*:

                    if check != 0:

                        x[i] = *reduced\_x*[i - check]

                    else:

                        x[i] = *reduced\_x*[i]

                else:

                    x[i] = 0

                    check += 1

        if (len(*reduced\_y*) == *C*.shape[1]):

            y = *reduced\_y*

        else:

            check = 0

            for j in range(len(y)):

                if j not in *dominated\_cols*:

                    if check != 0:

                        y[j] = *reduced\_y*[j - check]

                    else:

                        y[j] = *reduced\_y*[j]

                else:

                    y[j] = 0

                    check += 1

    return x, y

Αυτός ο κώδικας ορίζει συνάρτηση ονόματι interpretReducedStrategiesForOriginalGame, η οποία λαμβάνει διάφορες παραμέτρους εισόδου: reduced\_x και reduced\_y (προφίλ στρατηγικών για το μειωμένο παιχνίδι), R και C (αρχικοί πίνακες ωφέλειας), reduced\_R και reduced\_C (πίνακες ωφέλειας για το μειωμένο παιχνίδι) και dominated\_rows και dominated\_cols (οι δείκτες των κυριαρχούμενων γραμμών και στηλών αντίστοιχα).

Ακολουθεί εξήγηση του κώδικα:

1. Η συνάρτηση αρχικοποιεί δύο πίνακες, x και y, με μηδενικά. Αυτοί οι πίνακες θα αποθηκεύουν τα προφίλ των στρατηγικών για το αρχικό παιχνίδι.
2. Οι μεταβλητές i, j και check αρχικοποιούνται με 0.
3. Η συνάρτηση ελέγχει εάν οι διαστάσεις των μειωμένων\_R, μειωμένων\_C, R και C ταιριάζουν. Εάν το κάνουν, σημαίνει ότι το μειωμένο παιχνίδι είναι ίδιο με το αρχικό παιχνίδι (δεν πραγματοποιήθηκε μείωση) και η συνάρτηση ορίζει τα x και y ίσα με τα μειωμένα\_x και μειωμένα\_y, αντίστοιχα.
4. Εάν δεν ταιριάζει έστω μία από τις 2 διαστάσεις, η συνάρτηση προχωρά στην ανακατασκευή του προφίλ των στρατηγικών για το αρχικό παιχνίδι.
5. Η συνάρτηση ελέγχει αν το μήκος του μειωμένου\_x είναι ίσο με τον αριθμό των γραμμών στο R. Εάν είναι, ορίζει το x ίσο με το μειωμένο\_x.
6. Εάν το μήκος του μειωμένου\_x δεν είναι ίσο με τον αριθμό των γραμμών στο R, η συνάρτηση επαναλαμβάνεται πάνω από τα στοιχεία του x. Εάν ο δείκτης i δεν βρίσκεται στο σύνολο των κυριαρχούμενων σειρών, εκχωρεί την αντίστοιχη τιμή από το μειωμένο\_x στο x[i-check]. Ο έλεγχος μεταβλητής καταγράφει τον αριθμό των κυριαρχούμενων γραμμών που έχουν συναντηθεί μέχρι στιγμής. Εάν το i ανήκει στο σύνολο των κυριαρχούμενων γραμμών, το x[i] γίνεται 0 και ο έλεγχος αυξάνεται.
7. Η συνάρτηση εκτελεί μια παρόμοια διαδικασία για την ανασύνθεση του προφίλ των στρατηγικών για τον παίκτη στήλης (y), λαμβάνοντας υπόψη το σύνολο των στηλών που κυριαρχούνται.
8. Τέλος, η συνάρτηση επιστρέφει τα ανακατασκευασμένα προφίλ στρατηγικών, x και y, για το αρχικό παιχνίδι.

Συνολικά, αυτή η συνάρτηση ερμηνεύει τα προφίλ των στρατηγικών για το μειωμένο παιχνίδι και αναδομεί τα αντίστοιχα προφίλ για το αρχικό παιχνίδι, λαμβάνοντας υπόψη την (πιθανή) παρουσία κυριαρχούμενων γραμμών και στηλών.

Τώρα μπορούμε να δούμε τα υπόλοιπα περιεχόμενα της while loop που περιγράψαμε πιο πριν. Όπως αναφέραμε, αναλόγως την τιμή του choice1 που εισάγει ο χρήστης θα εκτελεστεί και το αντίστοιχο κομμάτι κώδικα που αφορά τον επιλεγμένο αλγόριθμο. Ο χρήστης μπορεί να επιλέξει να τρέξει όλους τους αλγορίθμους, οπότε και θα εκτελεστούν μαζί όλα όσα θα αναφερθούν παρακάτω. Αρχικά ας υποθέσουμε ότι ο χρήστης επιλέγει choice1=1 (DMP αλγόριθμο). Τότε ο κώδικας που τρέχει είναι ο παρακάτω:

if choice1 == "1":

*# EXECUTING DMP ALGORITHM...*

        start\_time = time.time()

        x, y, DMPepsAPPROX, DMPepsWSNE = approxNEConstructionDMP(

            reduced\_m, reduced\_n, reduced\_R, reduced\_C)

        DMPx, DMPy = interpretReducedStrategiesForOriginalGame(

            x, y, R, C, reduced\_R, reduced\_C, dominated\_rows, dominated\_cols)

        end\_time = time.time()

        elapsed\_time = end\_time - start\_time

        DMPtotalTime += elapsed\_time

*# if (flag == 0 or flag1 == 1):*

        print(bcolors.MSG + PLUSLINE)

        print("\tConstructed solution for DMP:")

        print(MINUSLINE)

        print("\tDMPx =", DMPx, "\n\tDMPy =", DMPy)

        print("\tDMPepsAPPROX =", DMPepsAPPROX,

            ".\tDMPepsWSNE =", DMPepsWSNE,

            ".\tRuntime =", elapsed\_time, "seconds", "." + bcolors.ENDC)

        print(PLUSLINE + bcolors.ENDC)

        if(flag==1):

            DMPepsAPPROX\_list.append(DMPepsAPPROX)

            R\_list.append(R)

            C\_list.append(C)

            if 0 <= DMPepsAPPROX < 0.1:

                interval = "[0,0.1)"

            elif 0.1 <= DMPepsAPPROX < 0.2:

                interval = "[0.1,0.2)"

            elif 0.2 <= DMPepsAPPROX < 0.3:

                interval = "[0.2,0.3)"

            elif 0.3 <= DMPepsAPPROX < 0.4:

                interval = "[0.3,0.4)"

            elif 0.4 <= DMPepsAPPROX < 0.5:

                interval = "[0.4,0.5)"

            elif 0.5 <= DMPepsAPPROX < 0.6:

                interval = "[0.5,0.6)"

            elif 0.6 <= DMPepsAPPROX < 0.7:

                interval = "[0.6,0.7)"

            elif 0.7 <= DMPepsAPPROX < 0.8:

                interval = "[0.7,0.8)"

            elif 0.8 <= DMPepsAPPROX < 0.9:

                interval = "[0.8,0.9)"

            elif 0.9 <= DMPepsAPPROX <= 1:

                interval = "[0.9,1.0]"

            registerE\_DMP[interval] += 1

            if 0 <= DMPepsWSNE < 0.1:

                interval1 = "[0,0.1)"

            elif 0.1 <= DMPepsWSNE < 0.2:

                interval1 = "[0.1,0.2)"

            elif 0.2 <= DMPepsWSNE < 0.3:

                interval1 = "[0.2,0.3)"

            elif 0.3 <= DMPepsWSNE < 0.4:

                interval1 = "[0.3,0.4)"

            elif 0.4 <= DMPepsWSNE < 0.5:

                interval1 = "[0.4,0.5)"

            elif 0.5 <= DMPepsWSNE < 0.6:

                interval1 = "[0.5,0.6)"

            elif 0.6 <= DMPepsWSNE < 0.7:

                interval1 = "[0.6,0.7)"

            elif 0.7 <= DMPepsWSNE < 0.8:

                interval1 = "[0.7,0.8)"

            elif 0.8 <= DMPepsWSNE < 0.9:

                interval1 = "[0.8,0.9)"

            elif 0.9 <= DMPepsWSNE <= 1:

                interval1 = "[0.9,1.0]"

            registerW\_DMP[interval1] += 1

            if games == numOfRandomGamesToSolve:

                print("\tTotal Time for DMP =", DMPtotalTime, "seconds",

                    ".\tAvg Time for DMP =", DMPtotalTime/numOfRandomGamesToSolve, "seconds", "." + bcolors.ENDC)

                with open("PkDMPApproxNEHistogram.out", "w") as f:

                    for key in registerE\_DMP.keys():

                        f.write("%s,%s\n" % (key, registerE\_DMP[key]))

                with open("PkDMPWSNEHistogram.out", "w") as f:

                    for key in registerW\_DMP.keys():

                        f.write("%s,%s\n" % (key, registerW\_DMP[key]))

*#find the 2 largest epsAPPROX in the list and their corresponding R and C*

                max1 = max(DMPepsAPPROX\_list)

                max1\_index = DMPepsAPPROX\_list.index(max1)

                R1 = R\_list[max1\_index]

                C1 = C\_list[max1\_index]

                DMPepsAPPROX\_list.remove(max1)

                max2 = max(DMPepsAPPROX\_list)

                max2\_index = DMPepsAPPROX\_list.index(max2)

                R2 = R\_list[max2\_index]

                C2 = C\_list[max2\_index]

*#save the R1 and C1 in a .out file with a name PkALGApproxNEWorstGame1ROW and PkALGApproxNEWorstGame1COL*

                np.savetxt("PkDMPApproxNEWorstGame1ROW.out", R1, *delimiter*=',', *fmt*='%1.4e')

                np.savetxt("PkDMPApproxNEWorstGame1COL.out", C1, *delimiter*=',', *fmt*='%1.4e')

*#save the R2 and C2 in a .out file with a name PkALGApproxNEWorstGame2ROW and PkALGApproxNEWorstGame2COL*

                np.savetxt("PkDMPApproxNEWorstGame2ROW.out", R2, *delimiter*=',', *fmt*='%1.4e')

                np.savetxt("PkDMPApproxNEWorstGame2COL.out", C2, *delimiter*=',', *fmt*='%1.4e')

*#plot the histogram of epsAPPROX*

                plt.bar(registerE\_DMP.keys(), registerE\_DMP.values(), *color*='g')

                plt.xlabel('Epsilon')

                plt.ylabel('Frequency')

                plt.title('Histogram of Epsilon Approximate Nash Equilibrium')

                plt.xticks(*rotation*=45, *ha*='right', *fontsize*=8)

                plt.subplots\_adjust(*bottom*=0.2)  *# Adjust the bottom margin*

                plt.savefig('PkDMPApproxNEHistogram.jpg')

                plt.clf()

*#plot the histogram of epsWSNE*

                plt.bar(registerW\_DMP.keys(), registerW\_DMP.values(), *color*='g')

                plt.xlabel('Epsilon')

                plt.ylabel('Frequency')

                plt.title('Histogram of Epsilon Weakly Stable Nash Equilibrium')

                plt.xticks(*rotation*=45, *ha*='right', *fontsize*=8)

                plt.subplots\_adjust(*bottom*=0.2)  *# Adjust the bottom margin*

                plt.savefig('PkDMPWSNEHistogram.jpg')

                plt.clf()

1. Εάν η choice1 είναι ίση με "1", εκτελεί τον αλγόριθμο DMP.
2. Ξεκινά ένα χρονόμετρο για τη μέτρηση του χρόνου εκτέλεσης.
3. Καλεί τη συνάρτηση approxNEConstructionDMP με τις παραμέτρους reduced\_m, reduced\_n, reduced\_R και reduced\_C, και εκχωρεί τις επιστρεφόμενες τιμές στα x, y, DMPepsAPPROX και DMPepsWSNE.
4. Καλεί τη συνάρτηση interpretReducedStrategiesForOriginalGame με παραμέτρους x, y, R, C, reduced\_R, reduced\_C, dominated\_rows και dominated\_cols και εκχωρεί τις επιστρεφόμενες τιμές στα DMPx και DMPy.
5. Υπολογίζει τον χρόνο που έχει παρέλθει αφαιρώντας τον χρόνο έναρξης από την τρέχουσα ώρα.
6. Προσθέτει τον χρόνο που έχει παρέλθει στη μεταβλητή DMPtotalTime.
7. Εκτυπώνει πληροφορίες σχετικά με την κατασκευασμένη λύση για DMP, συμπεριλαμβανομένων των DMPx, DMPy, DMPepsAPPROX και DMPepsWSNE.
8. Εάν flag = 1 (άρα εκτελούμε πείραμα), προσθέτει το DMPepsAPPROX στη λίστα DMPepsAPPROX\_list, προσθέτει το R στη λίστα R\_list και προσθέτει το C στη λίστα C\_list (που όπως είπαμε χρησιμοποιούνται για να βρεθούν τα δυο χειροτέρα παιχνίδια και τα αντίστοιχά τους R και C).
9. Καθορίζει το διάστημα στο οποίο ανήκουν τα DMPepsAPPROX και DMPepsWSNE και αυξάνει την αντίστοιχη μέτρηση στα λεξικά registerE\_DMP και registerW\_DMP.
10. Εάν τα παιχνίδια είναι ίσα με το numOfRandomGamesToSolve, εκτυπώνει τον συνολικό χρόνο και τον μέσο χρόνο για το DMP και, στην συνέχεια, εκτελεί τις ακόλουθες ενέργειες:

* Γράφει τις τιμές από το registerE\_DMP σε ένα αρχείο με το όνομα "PkDMPApproxNEHistogram.out".
* Γράφει τις τιμές από το registerW\_DMP σε ένα αρχείο με το όνομα "PkDMPWSNEHistogram.out".
* Βρίσκει τις δύο μεγαλύτερες τιμές στη λίστα DMPepsAPPROX\_list και τους αντίστοιχους δείκτες τους.
* Ανακτά τους αντίστοιχους πίνακες R και C από τις R\_list και C\_list με βάση τα ευρετήρια.
* Αποθηκεύει τους πίνακες R στα "PkDMPApproxNEWorstGame1ROW.out" και "PkDMPApproxNEWorstGame2ROW.out".
* Αποθηκεύει τους πίνακες C στα "PkDMPApproxNEWorstGame1COL.out" και "PkDMPApproxNEWorstGame2COL.out".
* Σχεδιάζει ένα ιστόγραμμα του DMPepsAPPROX και το αποθηκεύει στο "PkDMPApproxNEHistogram.jpg".
* Σχεδιάζει ένα ιστόγραμμα του DMPepsWSNE και το αποθηκεύει στο "PkDMPWSNEHistogram.jpg".

Συνοπτικά, αυτό το απόσπασμα κώδικα εκτελεί τον αλγόριθμο DMP, καταγράφει τα αποτελέσματα και εκτελεί διάφορους υπολογισμούς, εργασίες εκτύπωσης και διαχείρισης αρχείων με βάση το περιβάλλον και τις συνθήκες εκτέλεσης.

Ο παραπάνω κώδικας, με αλλαγές σε ορισμένα ονόματα μεταβλητών, είναι ακριβώς ίδιος και για τους άλλους αλγορίθμους καθώς και για την εκτέλεση όλων των αλγορίθμων μαζί. Οπότε αρκεί τώρα να εξηγήσουμε πώς προέκυψαν τα αποτελέσματα. Για αυτό πρέπει να εξηγήσουμε τις συναρτήσεις για τους αλγορίθμους DMP, DEL, FPPBR, FPUN.

Ας αρχίσουμε εξηγώντας τον κώδικα για τον DMP αλγόριθμο.

def approxNEConstructionDMP(*m*, *n*, *R*, *C*):

*# print(bcolors.IMPLEMENTED + '''*

*# ROUTINE: approxNEConstructionDMP ''' + bcolors.ENDC)*

*# PRE:    A bimatrix game, described by the two payoff matrices, with payoff values in [0,1].*

*# POST:   A profile of strategies (x,y) produced by the DMP algorithm.''' + bcolors.ENDC)*

*# Initialize strategies x and y*

    x = np.full(*R*.shape[0], 0.)

    y = np.full(*C*.shape[1], 0.)

    initial\_move = np.random.randint(*m*)

    x[initial\_move] = 1.

*# Best response for column player*

    ymax = np.argmax(np.dot(*C*.T, x))

    y[ymax] = 1.

*# Best response for row player*

    xmax = np.argmax(np.dot(*R*, y))

    xnew = np.full(*R*.shape[0], 0.)

    xnew[xmax] = 1.

    x = (x + xnew)/2

*# Check if the game is zero-sum*

    count = 0

    for i in range(*m*):

        for j in range(*n*):

            if *R*[i][j] == - *C*[i][j]:

                count = count+1

    if count == *m*\**n*:  *# if the above condition is true for every cell*

        x, y = solveZeroSumGame(*m*, *n*, *R*)

*# Compute the approximation guarantees*

    epsAPPROX, epsWSNE = computeApproximationGuarantees(*m*, *n*, *R*, *C*, x, y)

*# Return the profile of strategies and the approximation guarantees*

    return x, y, epsAPPROX, epsWSNE

Αυτό το απόσπασμα κώδικα ορίζει τη συνάρτηση approxNEConstructionDMP, η οποία λαμβάνει τέσσερις παραμέτρους: m, n, R και C (αυτά τα οποία προέκυψαν από την μείωση του αρχικού διμητρώου). Λειτουργεί ως εξής:

1. Αρχικοποιεί τις στρατηγικές x και y ως πίνακες μηδενικών με μήκη ίσα με τον αριθμό των γραμμών στο R και τον αριθμό των στηλών στο C, αντίστοιχα.
2. Επιλέγεται τυχαία μία ακέραια τιμή initial\_move μεταξύ 0 και m-1, η οποία θα χρησιμοποιηθεί στην συνέχεια προκειμένου να προσομοιωθεί η αυθαίρετη αρχική δράση του παίχτη γραμμής.
3. Δίνει στο στοιχείο του x με δείκτη ίσο με initial\_move τιμή 1, διαμορφώνοντας έτσι την επιλεγμένη στρατηγική για τον παίχτη γραμμής.
4. Υπολογίζει την βέλτιστη απόκριση για τον παίχτη στήλης βρίσκοντας τον δείκτη ymax που μεγιστοποιεί το dot γινόμενο των CT και x. Το στοιχείο ymax του y ορίζεται σε 1, υποδεικνύοντας την επιλεγμένη στρατηγική για τον παίχτη στήλης.
5. Υπολογίζει την βέλτιστη απόκριση για τον παίχτη γραμμής βρίσκοντας τον δείκτη xmax που μεγιστοποιεί το dot γινόμενο των R και y. Δημιουργεί έναν νέο πίνακα xnew με όλα τα στοιχεία του μηδενικά εκτός από το στοιχείο με δείκτη xmax, το οποίο έχει οριστεί σε 1.
6. Ενημερώνει το x ως μέσο όρο του x και του xnew, συνδυάζοντας τις δύο στρατηγικές του παίχτη γραμμής.
7. Ελέγχει αν το παιχνίδι είναι μηδενικού αθροίσματος ελέγχοντας το άθροισμα κάθε ζεύγους αντίστοιχων κελιών των R και C.
8. Εάν το άθροισμα όλων των κελιών (m\*n στο πλήθος) ισούται με 0, τότε πράγματι πρόκειται για παιχνίδι μηδενικού αθροίσματος, και συνεπώς καλείται η συνάρτηση solveZeroSumGame με ορίσματα m, n και R για να ληφθούν οι βέλτιστες στρατηγικές x και y για το παιχνίδι.
9. Καλεί τη συνάρτηση computeApproximationGuarantees με παραμέτρους m, n, R, C, x και y, για να υπολογίσει τις εγγυήσεις προσέγγισης epsAPPROX και epsWSNE.
10. Επιστρέφει τις στρατηγικές x και y, μαζί με τις εγγυήσεις προσέγγισης epsAPPROX και epsWSNE.

Συνοπτικά, αυτή η συνάρτηση υλοποιεί την εύρεση μιας κατά προσέγγιση ισορροπίας Nash μέσω του αλγορίθμου DMP. Αρχικοποιεί κατάλληλα τις στρατηγικές, εκτελεί τους υπολογισμούς βέλτιστης απόκρισης, χειρίζεται την περίπτωση παιχνιδιού μηδενικού αθροίσματος και υπολογίζει τις εγγυήσεις προσέγγισης για τις στρατηγικές.

Στην συνέχεια αναλύεται η computeApproximationGuarantees που αναφέρθηκε προηγουμένως.

def computeApproximationGuarantees(*m*, *n*, *R*, *C*, *x*, *y*):

*# print(bcolors.IMPLEMENTED + '''*

*# ROUTINE: computeApproximationGuarantees''' + bcolors.ENDC)*

*# PRE:    A bimatrix game, described by the two payoff matrices, with payoff values in [0,1].*

*#         A profile (x,y) of strategies for the two players.*

*# POST:   The two NASH approximation guarantees, epsAPPROX and eps in [0,1].''' + bcolors.ENDC)*

*# Compute the payoffs for player 1 and player 2*

    payoff\_1 = np.dot(*x*, np.dot(*R*, *y*))

    payoff\_2 = np.dot(*x*, np.dot(*C*, *y*))

*# Compute the approximate Nash equilibrium (APPROX-NE) payoffs for player 1 and player 2*

    approx\_ne\_payoff\_1 = np.max(np.dot(*R*, *y*))

    approx\_ne\_payoff\_2 = np.max(np.dot(*C*.T, *x*))

    epsAPPROXrow = approx\_ne\_payoff\_1-payoff\_1

    epsAPPROXcol = approx\_ne\_payoff\_2-payoff\_2

*# Compute the well-supported Nash equilibrium (WSNE) payoffs for player 1 and player 2*

    wsne\_payoff\_1 = np.min(np.dot(*R*, *y*))

    wsne\_payoff\_2 = np.min(np.dot(*C*.T, *x*))

    approx\_wsne\_payoff1 = np.max(np.dot(*R*, *y*))

    approx\_wsne\_payoff2 = np.max(np.dot(*C*.T, *x*))

    wsepsAPPROXrow = approx\_wsne\_payoff1-wsne\_payoff\_1

    wsepsAPPROXcol = approx\_wsne\_payoff2-wsne\_payoff\_2

*# Compute the approximation guarantees*

    epsAPPROX = max(epsAPPROXrow, epsAPPROXcol)

    epsWSNE = max(wsepsAPPROXrow, wsepsAPPROXcol)

    return epsAPPROX, epsWSNE

Το απόσπασμα κώδικα ορίζει τη συνάρτηση computeApproximationGuarantees, η οποία λαμβάνει έξι παραμέτρους: m, n, R, C, x και y. Ακολουθεί μια επισκόπηση της λειτουργία της:

1. Υπολογίζει τις ωφέλειες για τον παίκτη 1 και τον παίκτη 2, παίρνοντας το dot γινόμενο του x με το dot γινόμενο των R και y και με το dot γινόμενο των C και y αντίστοιχα.
2. Υπολογίζει τις ωφέλειες της κατά προσέγγιση ισορροπίας Nash (APPROX-NE) για τον παίκτη 1 και τον παίκτη 2, λαμβάνοντας το μέγιστο του dot γινόμενου των R και y και του dot γινομένου των CΤ και x, αντίστοιχα.
3. Υπολογίζει τις διαφορές μεταξύ των κατά ωφελειών της κατά προσέγγιση ισορροπίας Nash και των αρχικών payoff για τον παίκτη 1 και τον παίκτη 2, με αποτέλεσμα το epsAPPROXrow και το epsAPPROXcol.
4. Υπολογίζει τις ωφέλειες της καλά στηριγμένης ισορροπίας Nash (WSNE) για τον παίκτη 1 και τον παίκτη 2, παίρνοντας το ελάχιστο του dot γινομένου των R και y και του dot γινομένου του CΤ και x, αντίστοιχα.
5. Υπολογίζει τις διαφορές μεταξύ των κατά προσέγγιση ωφελειών WSNE και των αρχικών WSNE payoff για τον παίκτη 1 και τον παίκτη 2, με αποτέλεσμα τα wsepsAPPROXrow και wsepsAPPROXcol.
6. Καθορίζει το μέγιστο μεταξύ epsAPPROXrow και epsAPPROXcol ως εγγύηση προσέγγισης epsAPPROX.
7. Καθορίζει το μέγιστο μεταξύ wsepsAPPROXrow και wsepsAPPROXcol ως εγγύηση προσέγγισης epsWSNE.
8. Επιστρέφει τις εγγυήσεις προσέγγισης epsAPPROX και epsWSNE.

Συνοπτικά, αυτή η συνάρτηση υπολογίζει τις εγγυήσεις προσέγγισης για ένα δεδομένο προφίλ στρατηγικών (x, y) σε ένα διμητρικό παιχνίδι με βάση τους αντίστοιχους τύπους.

Αφού αναλύθηκε το πώς παράγονται τα epsApprox και epsWSNE, μπορούμε να δούμε και τους υπόλοιπους αλγορίθμους.

Ας δούμε τον κώδικα για τον DEL αλγόριθμο :

def approxNEConstructionDEL(*m*, *n*, *R*, *C*):

*# print(bcolors.IMPLEMENTED + '''*

*# ROUTINE: approxNEConstructionDEL '''+bcolors.ENDC)*

*# PRE:    A bimatrix game, described by the two payoff matrices, with payoff values in [0,1].*

*# POST:   A profile of strategies (x,y) produced by the DEL algorithm.''' + bcolors.ENDC)*

*# Check if the game is zero-sum*

    count = 0

    for i in range(*m*):

        for j in range(*n*):

            if *R*[i][j] == - *C*[i][j]:

                count = count+1

    if count == *m*\**n*:  *# if the above condition is true for every cell*

        x, y = solveZeroSumGame(*m*, *n*, *R*)

        epsAPPROX, epsWSNE = computeApproximationGuarantees(*m*, *n*, *R*, *C*, x, y)

        return (x, y, epsAPPROX, epsWSNE)

*# if not zero sum then the algorithm starts*

*# Calculation of MAXMIN strategies and guaranteed payoffs for both players*

    xrow, yrow = solveZeroSumGame(*m*, *n*, *R*)  *# MAXMIN strategy for row player*

*# Guaranteed payoff for row player*

    Vrow = np.dot(np.transpose(xrow), np.dot(*R*, yrow))

*# MAXMIN strategy for column player*

    xcol, ycol = solveZeroSumGame(*m*, *n*, -*C*)

*# Guaranteed payoff for column player*

    Vcol = np.dot(np.transpose(xcol), np.dot(*C*, ycol))

*# Compare the guaranteed payoffs and transpose matrices if necessary*

    if Vrow < Vcol:

        temp\_xrow, temp\_yrow, temp\_Vrow = xrow, yrow, Vrow

        xrow, yrow, Vrow = xcol, ycol, Vcol

        xcol, ycol, Vcol = temp\_xrow, temp\_yrow,  temp\_Vrow

*# SCENARIO 1: No player has a guaranteed payoff greater than 2/3*

    if Vrow <= 2/3:

        x = xcol

        y = yrow

*# SCENARIO 2: Guaranteed utility > 2/3 for row player, but column player finds no utility > 2/3 against xrow*

    elif np.max(np.dot(xrow.T, *C*)) <= 2/3:

        x = xrow

        y = yrow

    else:

*# SCENARIO 3: Row player is guaranteed utility > 2/3 and column player finds utility > 2/3 against xrow*

        j = np.argmax(np.dot(xrow.T, *C*))

        i = np.argmax(np.logical\_and(*R*[:, j] > 1/3, *C*[:, j] > 1/3))

        x = np.zeros(*m*)

        x[i] = 1

        y = np.zeros(*n*)

        y[j] = 1

    epsAPPROX, epsWSNE = computeApproximationGuarantees(*m*, *n*, *R*, *C*, x, y)

    return (x, y, epsAPPROX, epsWSNE)

Το απόσπασμα κώδικα ορίζει την συνάρτηση approxNEConstructionDEL, η οποία λαμβάνει τέσσερις παραμέτρους: m, n, R και C. Ακολουθεί μια επισκόπηση της λειτουργίας της:

1. Ελέγχει εάν το παιχνίδι είναι μηδενικού αθροίσματος ελέγχοντας κάθε κελί των payoff πινάκων R και C. Εάν η συνθήκη R[i][j] == -C[i][j] ισχύει για όλα τα αντίστοιχα κελιά, καλεί τη συνάρτηση solveZeroSumGame για να λάβει τις στρατηγικές (x, y) για το παιχνίδι μηδενικού αθροίσματος. Στη συνέχεια, υπολογίζει τις εγγυήσεις προσέγγισης epsAPPROX και epsWSNE χρησιμοποιώντας τη συνάρτηση computeApproximationGuarantees και επιστρέφει (x, y, epsAPPROX, epsWSNE).
2. Εάν το παιχνίδι δεν είναι μηδενικού αθροίσματος, ο αλγόριθμος συνεχίζει. Υπολογίζει τις στρατηγικές MAXMIN και για τους δύο παίκτες καλώντας τη συνάρτηση solveZeroSumGame ξεχωριστά για R και -C. Υπολογίζει επίσης τις εγγυημένες ωφέλειες και για τους δύο παίκτες: Vrow για τον παίκτη γραμμής και Vcol για τον παίκτη στήλης.
3. Εάν το Vrow είναι μικρότερο ή ίσο με 2/3, ορίζει το x ως την στρατηγική MAXMIN του παίκτη στήλης (xcol) και το y ως την στρατηγική MAXMIN του παίκτη γραμμής (yrow).
4. Εάν η μέγιστη ωφέλεια του παίκτη στήλης έναντι του xrow είναι μικρότερη ή ίση με 2/3, ορίζει το x και το y ως τις MAXMIN στρατηγικές του παίκτη γραμμής (xrow και yrow).
5. Διαφορετικά, εισέρχεται στο ΣΕΝΑΡΙΟ 3, όπου ο παίκτης γραμμής έχει εγγυημένη ωφέλεια μεγαλύτερη από 2/3 και ο παίκτης στήλης βρίσκει ωφέλεια μεγαλύτερη από 2/3 έναντι του xrow. Βρίσκει τη στήλη j που μεγιστοποιεί την ωφέλεια του παίκτη στήλης έναντι του xrow και, στην συνέχεια, βρίσκει την γραμμή i που ικανοποιεί τις συνθήκες R[:, j] > 1/3 και C[:, j] > 1/3. Ορίζει το x με κωδικοποίηση one-hot για την σειρά i και y με κωδικοποίηση one-hot για την στήλη j.
6. Τέλος, υπολογίζει τις εγγυήσεις προσέγγισης epsAPPROX και epsWSNE χρησιμοποιώντας την συνάρτηση computeApproximationGuarantees με ορίσματα τις λαμβανόμενες στρατηγικές (x, y), και επιστρέφει (x, y, epsAPPROX, epsWSNE).

Συνοπτικά, αυτή η συνάρτηση βρίσκει ένα προφίλ στρατηγικών (x, y) εφαρμόζοντας τον DEL αλγόριθμο σε ένα διμητρικό παιχνίδι. Εξετάζει διαφορετικά σενάρια με βάση τις εγγυημένες αποδόσεις και επιλέγει τις στρατηγικές ανάλογα.

Ας δούμε τον κώδικα για τον FPPBR (Fictitious Play Pure Best Response) αλγόριθμο :

def approxNEConstructionFPPBR(*m*, *n*, *R*, *C*):

*# print(bcolors.IMPLEMENTED + '''*

*# ROUTINE: approxNEConstructionFP ''' + bcolors.ENDC)*

*# PRE:    A bimatrix game, described by the two payoff matrices, with payoff values in [0,1].*

*# POST:   A profile of strategies (x,y) produced by the FICTITIOUS PLAY algorithm (Pure Best Response version).''' + bcolors.ENDC)*

*# checking if the game is zero-sum*

    count = 0

    for i in range(*m*):

        for j in range(*n*):

            if *R*[i][j] == - *C*[i][j]:

                count = count+1

    if count == *m*\**n*:  *# if the above condition is true for every cell*

        χ, ψ = solveZeroSumGame(*m*, *n*, *R*)

        epsAPPROX, epsWSNE = computeApproximationGuarantees(*m*, *n*, *R*, *C*, χ, ψ)

        return (χ, ψ, epsAPPROX, epsWSNE)

*# if not zero sum then the algorithm starts*

*# steps 1,2 , initializing the variables*

    T = 10

    x = np.full(*R*.shape[0], 0.)

    x[0] = 1.

    y = np.full(*C*.shape[1], 0.)

    y[0] = 1.

    χ, ψ = x, y

*# step 3, calculating pure best responses*

    for t in range(1, T-1, 1):

*# row player*

        xmax = np.argmax(np.dot(*R*, ψ))

        x = np.full(*R*.shape[0], 0.)

        x[xmax] = 1.

        χ = 1/(t+1) \* (t\*χ + x)

*# column player*

        ymax = np.argmax(np.dot(*C*.T, χ))

        y = np.full(*C*.shape[1], 0.)

        y[ymax] = 1.

        ψ = 1/(t+1) \* (t\*ψ + y)

    epsAPPROX, epsWSNE = computeApproximationGuarantees(*m*, *n*, *R*, *C*, χ, ψ)

    return (χ, ψ, epsAPPROX, epsWSNE)

Το απόσπασμα κώδικα ορίζει τη συνάρτηση approxNEConstructionFPPBR, η οποία λαμβάνει τέσσερις παραμέτρους: m, n, R και C. Ακολουθεί μια επισκόπηση της λειτουργίας της:

1. Ελέγχει εάν το παιχνίδι είναι μηδενικού αθροίσματος επαναλαμβάνοντας σε κάθε κελί των πινάκων πληρωμής R και C. Εάν η συνθήκη R[i][j] == -C[i][j] ισχύει για κάθε κελί, καλεί τη συνάρτηση solveZeroSumGame για τη λήψη των στρατηγικών (χ, ψ) για το παιχνίδι μηδενικού αθροίσματος. Στη συνέχεια, υπολογίζει τις εγγυήσεις προσέγγισης epsAPPROX και epsWSNE χρησιμοποιώντας τη συνάρτηση computeApproximationGuarantees με τις στρατηγικές (χ, ψ) και επιστρέφει (χ, ψ, epsAPPROX, epsWSNE).
2. Εάν το παιχνίδι δεν είναι μηδενικού αθροίσματος, ο αλγόριθμος προχωρά με τον αλγόριθμο Fictitious Play (παραλλαγή Pure Best Response). Αρχικοποιούνται οι μεταβλητές T, x, y, χ και ψ. Το T αντιπροσωπεύει τον αριθμό των επαναλήψεων, και εφόσον δεν του δίνεται συγκεκριμένη τιμή στην εκφώνηση έχει επιλεγεί αυθαίρετα 10. Προκειμένου να προσομοιωθεί η αυθαίρετη αρχική αμιγής δράση των παικτών κατά την εκκίνηση του παιχνιδιού, τα διανύσματα x,y (μήκους ίσου με τις γραμμές και τις στήλες του διμητρώου αντίστοιχα) αρχικοποιούνται με 0, και επιλέγεται τυχαία τα στοιχεία x[0], y[0] τους να λάβουν τιμή 1. Αρχικά τα χ, ψ ισούνται με τα x,y αντίστοιχα.
3. Εκτελεί τα κύρια βήματα του αλγόριθμου Fictitious Play για T-1 επαναλήψεις:

* Υπολογίζει την αμιγή βέλτιστη απόκριση για τον παίκτη γραμμής βρίσκοντας τον δείκτη της στρατηγικής που μεγιστοποιεί το όφελος απέναντι στην στρατηγική ψ του αντιπάλου. Ενημερώνει τα x και χ ανάλογα.
* Υπολογίζει την αμιγή βέλτιστη απόκριση για τον παίκτη στήλης βρίσκοντας τον δείκτη της στρατηγικής που μεγιστοποιεί το όφελος απέναντι στην στρατηγική χ του αντιπάλου. Ενημερώνει το y και το ψ ανάλογα.
* Η ενημέρωση των χ, ψ γίνεται χρησιμοποιώντας σταθμισμένους μέσους όρους της τρέχουσας στρατηγικής και των προηγούμενων στρατηγικών που έχουν παιχτεί ήδη.

1. Τέλος, αφού ολοκληρωθούν οι επαναλήψεις, υπολογίζει τις εγγυήσεις προσέγγισης epsAPPROX και epsWSNE με την συνάρτηση computeApproximationGuarantees και όρισμα τις στρατηγικές (χ, ψ) και επιστρέφει (χ, ψ, epsAPPROX, epsWSNE).

Συνοπτικά, αυτή η συνάρτηση κατασκευάζει ένα προφίλ στρατηγικών (χ, ψ) χρησιμοποιώντας τον αλγόριθμο Fictitious Play (παραλλαγή Pure Best Response) σε ένα διμητρικό παιχνίδι. Ξεκινά από αυθαίρετες αμιγείς δράσεις των 2 παιχτών και έπειτα συνεχίζει ενημερώνοντας επαναληπτικά τις στρατηγικές τους με βάση αμιγείς βέλτιστες αποκρίσεις. Στο τέλος υπολογίζονται οι εγγυήσεις προσέγγισης.

Tέλος ας δούμε τον κώδικα για τον FPUNI (Fictitious Play Uniform) αλγόριθμο :

def approxNEConstructionFPUNI(*m*, *n*, *R*, *C*):

*# print(bcolors.IMPLEMENTED + '''*

*# ROUTINE: approxNEConstructionFP ''' + bcolors.ENDC)*

*# PRE:    A bimatrix game, described by the two payoff matrices, with payoff values in [0,1].*

*# POST:   A profile of strategies (x,y) produced by the FICTITIOUS PLAY algorithm (Uniform version).''' + bcolors.ENDC)*

    count = 0

    for i in range(*m*):

        for j in range(*n*):

            if *R*[i][j] == - *C*[i][j]:

                count = count+1

    if count == *m*\**n*:  *# if the above condition is true for every cell*

        x, y = solveZeroSumGame(*m*, *n*, *R*)

        epsAPPROX, epsWSNE = computeApproximationGuarantees(*m*, *n*, *R*, *C*, x, y)

        return (x, y, epsAPPROX, epsWSNE)

*# steps 1,2 , initializing the variables*

    T = 10

    x = np.full(*R*.shape[0], 1/*m*)

    y = np.full(*C*.shape[1], 1/*n*)

    χ, ψ = x, y

*# step 3, calculating pure best responses*

    for t in range(1, T-1, 1):

        pbr\_list = set()

*# row player*

        xmax = np.max(np.dot(*R*, ψ))

        for i in range(*m*):

            if (np.dot(*R*, ψ))[i] == xmax:

                pbr\_list.add(i)

        random\_move = random.choice(list(pbr\_list))

        x = np.full(*R*.shape[0], 0)

        x[random\_move] = 1

        χ = 1/(t+1) \* (t\*χ + x)

        pbr\_list = set()

*# column player*

        ymax = np.max(np.dot(*C*.T, χ))

        for i in range(*n*):

            if (np.dot(*C*.T, χ))[i] == ymax:

                pbr\_list.add(i)

        random\_move = random.choice(list(pbr\_list))

        y = np.full(*C*.shape[1], 0)

        y[random\_move] = 1

        ψ = 1/(t+1) \* (t\*ψ + y)

*# Check if the game is zero-sum*

    count = 0

    for i in range(*m*):

        for j in range(*n*):

            if *R*[i][j] == - *C*[i][j]:

                count = count+1

    if count == *m*\**n*:  *# if the above condition is true for every cell*

        x, y = solveZeroSumGame(*m*, *n*, *R*)

    epsAPPROX, epsWSNE = computeApproximationGuarantees(*m*, *n*, *R*, *C*, χ, ψ)

    return (χ, ψ, epsAPPROX, epsWSNE)

Το απόσπασμα κώδικα ορίζει τη συνάρτηση approxNEConstructionFPUNI, η οποία λαμβάνει τέσσερις παραμέτρους: m, n, R και C. Ακολουθεί μια επισκόπηση της λειτουργίας της.

1. Ελέγχει εάν το παιχνίδι είναι μηδενικού αθροίσματος ελέγχοντας κάθε κελί των πινάκων ωφελειών R και C. Εάν η συνθήκη R[i][j] == -C[i][j] ισχύει για όλα τα αντίστοιχα κελιά, καλεί τη συνάρτηση solveZeroSumGame για να λάβει τις στρατηγικές (x, y) για το παιχνίδι μηδενικού αθροίσματος. Στη συνέχεια, υπολογίζει τις εγγυήσεις προσέγγισης epsAPPROX και epsWSNE χρησιμοποιώντας τη συνάρτηση computeApproximationGuarantees με τις αρχικές στρατηγικές (x, y) και επιστρέφει (x, y, epsAPPROX, epsWSNE).
2. Εάν το παιχνίδι δεν είναι μηδενικού αθροίσματος, ο αλγόριθμος προχωρά με τον αλγόριθμο Fictitious Play (παραλλαγή Uniform). Αρχικοποιούνται οι μεταβλητές T, x, y, χ και ψ. Το T αντιπροσωπεύει τον αριθμό των επαναλήψεων, ο οποίος έχει επιλεγεί αυθαίρετα να είναι 10. Προκειμένου να δοθεί ίδια μάζα πιθανότητας σε όλες τις δράσεις κατά την εκκίνηση του παιχνιδιού, τα διανύσματα x,y (μήκους ίσου με τις γραμμές και τις στήλες του διμητρώου αντίστοιχα) αρχικοποιούνται με 1/m και 1/n, αντίστοιχα. Αρχικά τα χ, ψ ισούνται με τα x,y αντίστοιχα.
3. Εκτελεί τα κύρια βήματα του αλγόριθμου Fictitious Play για T-1 επαναλήψεις:

* Υπολογίζει τις αμιγείς βέλτιστες αποκρίσεις για τον παίκτη γραμμής, βρίσκοντας όλες τις δράσεις (σε περίπτωση που είναι περισσότερες από μία) που του εξασφαλίζουν μέγιστο κέρδος απέναντι στην στρατηγική ψ του αντιπάλου και προσθέτοντάς τες στην κενή λίστα pbr\_list.
* Επιλέγει τυχαία μια στρατηγική από τη λίστα pbr\_list και ενημερώνει τα x και χ ανάλογα. Στην συνέχεια η λίστα pbr\_list καθαρίζεται.
* Εκτελεί την ίδια διαδικασία για να βρει τις βέλτιστες αμιγείς αποκρίσεις για τον παίκτη στήλης: βρίσκει όλες τις δράσεις (σε περίπτωση που είναι περισσότερες από μία) που του εξασφαλίζουν μέγιστο κέρδος απέναντι στην στρατηγική χ του αντιπάλου και τις προσθέτει στην λίστα pbr\_list.
* Επιλέγει τυχαία μια στρατηγική από το pbr\_list και ενημερώνει τα y και ψ ανάλογα.
* Η ενημέρωση των χ, ψ γίνεται χρησιμοποιώντας σταθμισμένους μέσους όρους της τρέχουσας στρατηγικής και των προηγούμενων στρατηγικών που έχουν παιχτεί ήδη.

1. Τέλος, αφού ολοκληρωθούν οι επαναλήψεις, υπολογίζει τις εγγυήσεις προσέγγισης epsAPPROX και epsWSNE με την συνάρτηση computeApproximationGuarantees και ορίσματα τις στρατηγικές (χ, ψ) και επιστρέφει (χ, ψ, epsAPPROX, epsWSNE).

Συνοπτικά, αυτή η συνάρτηση κατασκευάζει ένα προφίλ στρατηγικών (χ, ψ) χρησιμοποιώντας τον αλγόριθμο Fictitious Play (παραλλαγή Uniform) σε ένα διμητρικό παιχνίδι. Ξεκινά δίνοντας ίδια μάζα πιθανότητας σε όλες τις δράσεις και για τους 2 παίχτες, και έπειτα συνεχίζει ενημερώνοντας επαναληπτικά τις στρατηγικές τους με βάση αμιγείς βέλτιστες αποκρίσεις. Στο τέλος υπολογίζονται οι εγγυήσεις προσέγγισης.

Αφού λοιπόν είδαμε και αναλύσαμε τον κώδικα θα δείξουμε πώς δουλεύει. Όταν ξεκινά το πρόγραμμα ο χρήστης βλέπει την παρακάτω οθόνη, η οποία του δίνει την επιλογή να φορτώσει τους πίνακες R, C από κάποιο αρχείο.

A blue rectangle with white text

Description automatically generated with low confidence

Στην περίπτωση που ο χρήστης πατήσει y :

* Αν υπάρχουν ήδη αρχεία στο path με όνομα R.out και C.out τότε το πρόγραμμα τον ρωτάει αν θέλει να φορτώσει αυτά τα αρχεία ή κάποια άλλα.



* Αν ο χρήστης πατήσει ναι (y) και εδώ τότε το πρόγραμμα θα φορτώσει τα αρχεία και θα φτιάξει τους πίνακες R, C με βάση τα περιεχόμενά τους. Μετά θα γίνει αφαίρεση όλων των strictly dominated γραμμών/στηλών (εάν υπάρχουν) και θα εμφανιστεί το μενού που σχετίζεται με την επιλογή του αλγορίθμου που θα εκτελεστεί. Αυτά φαίνονται στην παρακάτω εικόνα:

A screenshot of a computer program

Description automatically generated with medium confidence

* Στην περίπτωση που ο χρήστης πατήσει όχι (n) στην ερώτηση για την φόρτωση των αρχείων με όνομα R.out και C.out που υπάρχουν στο path (ενώ έχει πατήσει ναι στην φόρτωση αρχείων), το πρόγραμμα του ζητάει να πληκτρολογήσει τα ονόματα των αρχείων για τους πίνακες R και C που θέλει να φορτώσει. Σημειώνεται πως το πρόγραμμα δεν θα δεχτεί ως input ονόματα αρχείων που δεν υπάρχουν στο path. Αν λοιπόν ο χρήστης επιχειρήσει να φορτώσει αρχείο που δεν εντοπίζεται στο path το πρόγραμμα απλά θα του ζητά να πληκτρολογήσει το όνομα του αρχείου ξανά μέχρι να δοθεί κάποιο έγκυρο όνομα.



* Αφού ο χρήστης φορτώσει τα αρχεία που επιθυμεί το πρόγραμμα θα εκτελέσει τον ίδιο κώδικα με προηγουμένως. Δηλαδή θα κάνει μείωση των πινάκων (αν γίνεται) και ύστερα θα του δώσει πάλι την επιλογή να τρέξει όποιον από τους αλγορίθμους θέλει.

1. Για την επιλογή του DMP αλγορίθμου:

A screenshot of a computer

Description automatically generated with medium confidence

Όπως φαίνεται το πρόγραμμα εκτελεί τον αλγόριθμο και στο τέλος του τυπώνει τα x και y τα οποία προέκυψαν, καθώς και τα ζητούμενα epsApprox, epsWSNE, και το Runtime.

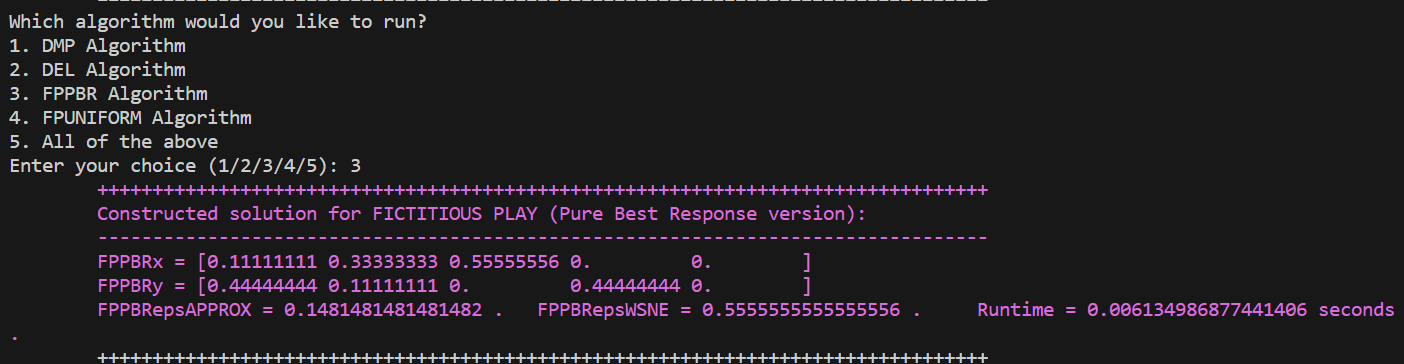
1. Για την επιλογή του DEL αλγορίθμου:

A screen shot of a computer

Description automatically generated with low confidence

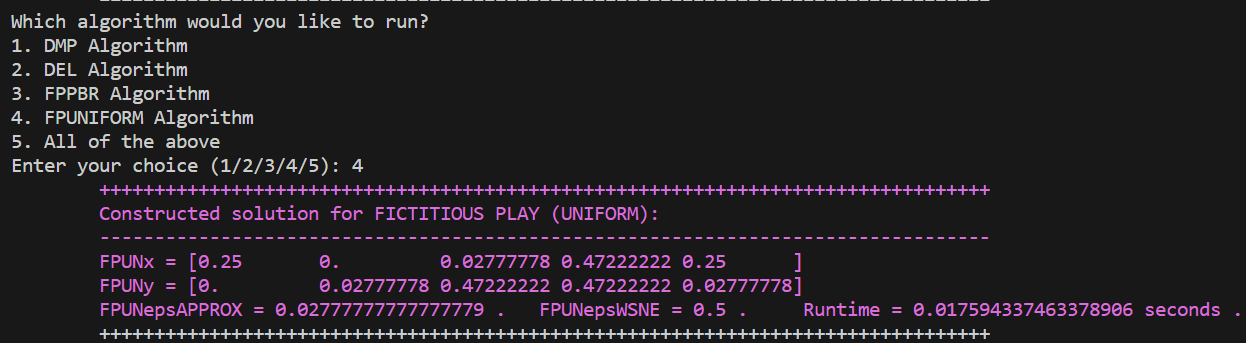
Όπως φαίνεται το πρόγραμμα εκτελεί τον αλγόριθμο και στο τέλος του τυπώνει τα x και y τα οποία προέκυψαν, καθώς και τα ζητούμενα epsApprox, epsWSNE, και το Runtime.

1. Για την επιλογή του FP αλγορίθμου (Pure Best Response):



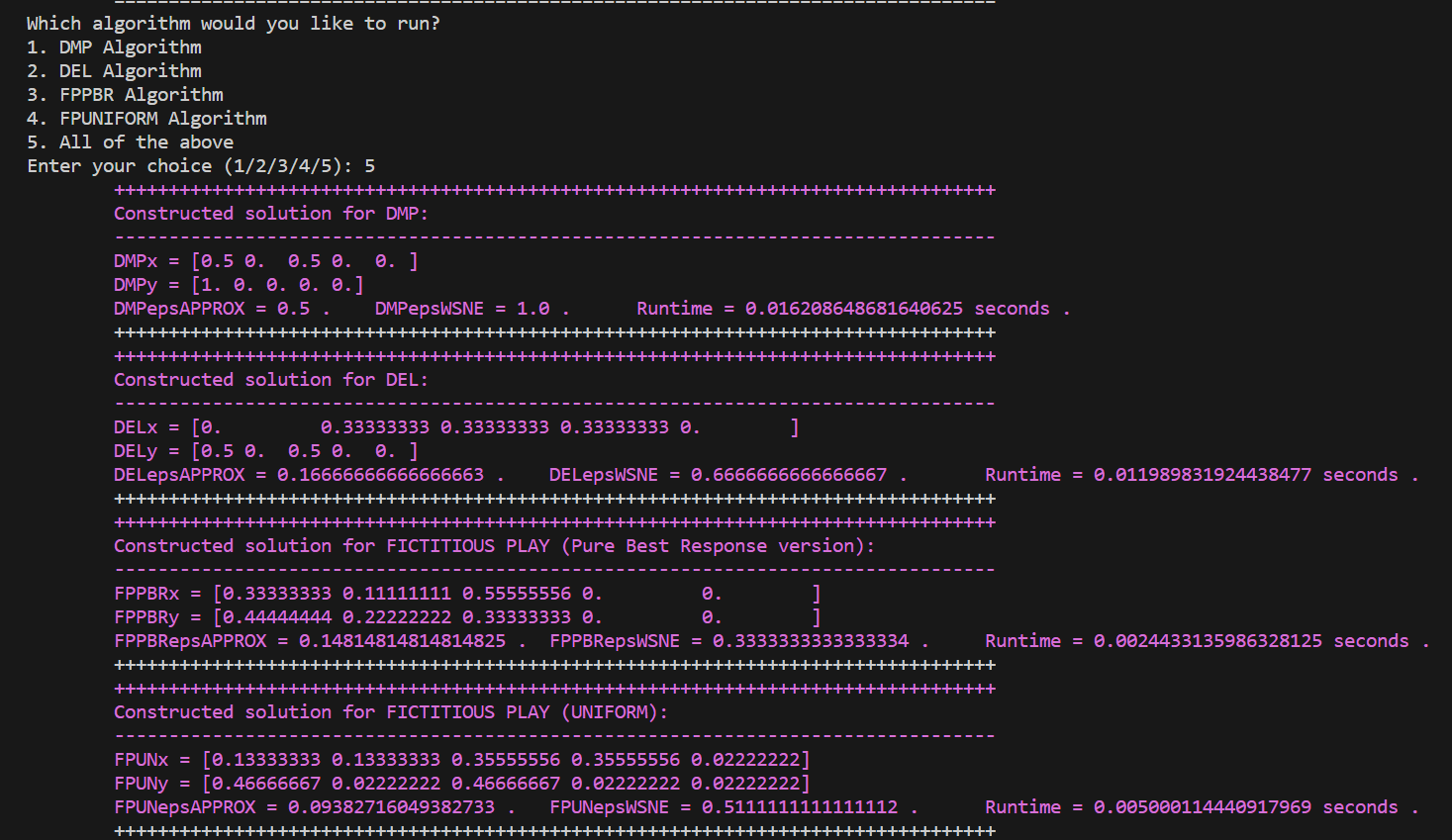
Όπως φαίνεται το πρόγραμμα εκτελεί τον αλγόριθμο και στο τέλος του τυπώνει τα x και y τα οποία προέκυψαν, καθώς και τα ζητούμενα epsApprox, epsWSNE, και το Runtime.

1. Για την επιλογή του FP αλγορίθμου (UNIFORM):



Όπως φαίνεται το πρόγραμμα εκτελεί τον αλγόριθμο και στο τέλος του τυπώνει τα x και y τα οποία προέκυψαν, καθώς και τα ζητούμενα epsApprox, epsWSNE, και το Runtime.

1. Για την επιλογή της εκτέλεσης όλων των αλγορίθμων:



Όπως φαίνεται το πρόγραμμα εκτελεί επιτυχώς όλους τους αλγορίθμους. Στο τέλος του τυπώνει τα x και y τα οποία προέκυψαν, καθώς και τα ζητούμενα epsApprox, epsWSNE και το Runtime για κάθε αλγόριθμο με την σειρά. (Να σημειωθεί ότι οι χρόνοι που προκύπτουν είναι τόσο μικροί λόγω του μεγέθους των αρχικών πινάκων).

Σε περίπτωση που δοθεί άκυρη είσοδος (όχι y ή n) στην ερώτηση σχετικά με την φόρτωση των αρχείων, η ερώτηση επαναλαμβάνεται έως ότου δοθεί κάποια έγκυρη είσοδος. Στην περίπτωση που ο χρήστης πατήσει n (δεν θέλει να φορτώσει αρχείο από το path) :

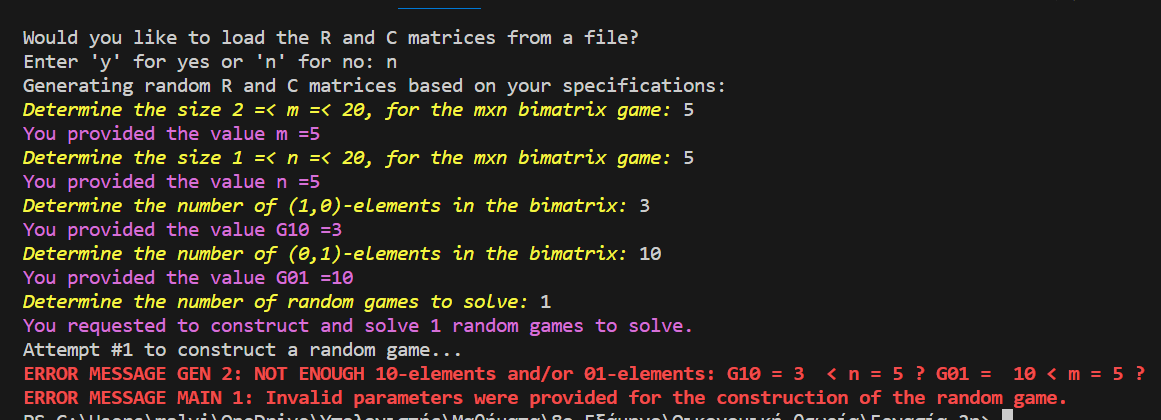
* Ζητείται από τον χρήστη να εισάγει τις επιθυμητές προδιαγραφές του παιχνιδιού (διαστάσεις m, n του παιχνιδιού, πλήθος (1,0) – και (0,1) – στοιχείων και πόσα παιχνίδια θα εκτελεστούν). Αν οι τιμές που δοθούν είναι έγκυρες, τότε θα παραχθούν επιτυχώς όσα παιχνίδια επέλεξε ο χρήστης με αυτές τις προδιαγραφές, αλλιώς θα τυπωθεί κατάλληλο μήνυμα σφάλματος.

Επίσης, αν έχει επιλεγεί εκτέλεση μεμονωμένου παιχνιδιού (πλήθος 1), το πρόγραμμα αφού δημιουργήσει επιτυχώς το παιχνίδι ρωτά τον χρήστη αν επιθυμεί να αποθηκεύσει τα μητρώα R και C που προέκυψαν. Αν επιλέξει όχι (n) ή δώσει μη έγκυρη είσοδο (ερμηνεύεται ως όχι), τότε θα τυπωθούν το αρχικό και το μειωμένο παιχνίδι και στην συνέχεια ο χρήστης θα επιλέξει ποιον αλγόριθμο θέλει να εκτελέσει ακριβώς όπως και προηγουμένως. Αν επιλέξει ναι (y), τότε στην συνέχεια θα του ζητηθεί να ονομάσει τα 2 αρχεία που θα κατασκευαστούν (ένα για κάθε μητρώο), και ύστερα αυτά τα αρχεία θα αποθηκευτούν στο path. Σημειώνεται πως σε περίπτωση που ο χρήστης δεν εισάγει την ορθή κατάληξη “.out” στο όνομα των αρχείων αυτό θα συμπληρωθεί αυτόματα από τον κώδικα. Ύστερα εμφανίζονται το αρχικό και το μειωμένο παιχνίδι και το πρόγραμμα συνεχίζει όπως και στις άλλες περιπτώσεις.

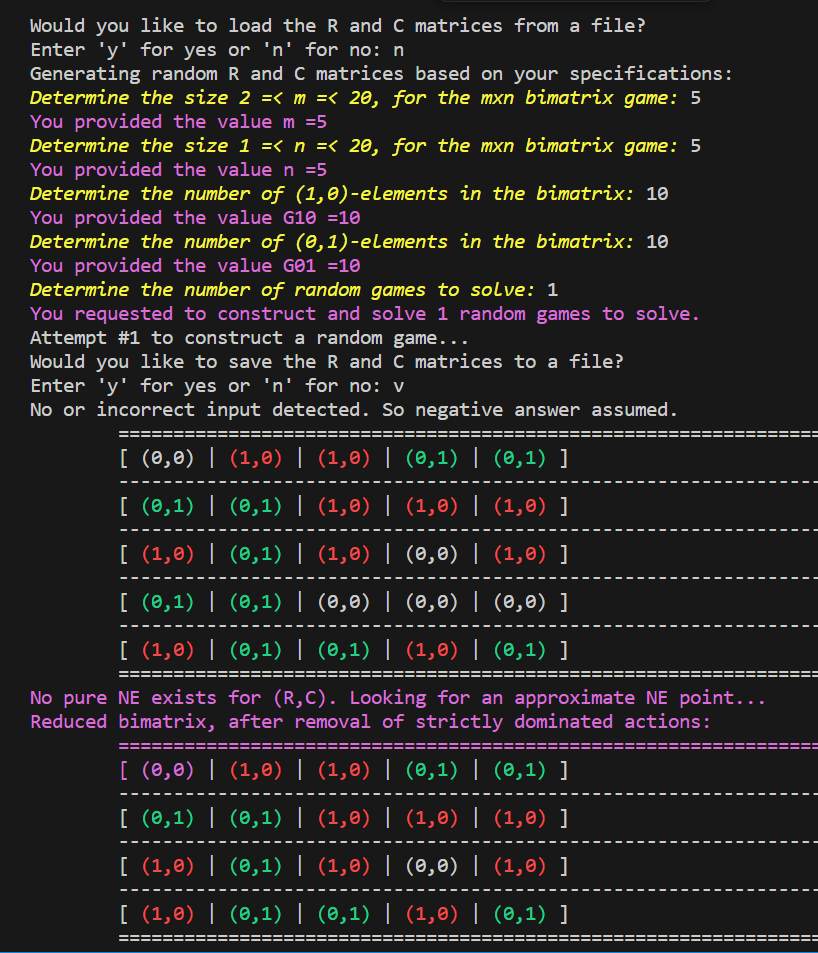
Τέλος, αν επιλεγεί εκτέλεση πειράματος (δηλαδή αν επιλεγεί δημιουργία 2 ή περισσότερων παιχνιδιών), τότε παρακάμπτεται αυτόματα η ερώτηση σχετικά με την αποθήκευση και τυπώνεται σχετικό μήνυμα. Δηλαδή στην συγκεκριμένη περίπτωση δεν υπάρχει δυνατότητα αποθήκευσης των μητρώων. Αυτό γίνεται για λόγους ροής του προγράμματος και αναγνωσιμότητας, καθώς διαφορετικά θα μπορούσε να δημιουργηθεί πρόβλημα κατά την εκτέλεση μεγάλων πειραμάτων. Αντιθέτως, σε αυτήν την περίπτωση δημιουργούνται τα ζητούμενα αρχεία σχετικά με την εκτέλεση πειραμάτων, δηλαδή εξάγονται ιστογράμματα για τα ε και αποθηκεύονται τα αρχεία από το οποία προέκυψαν, ενώ αποθηκεύονται επίσης τα 2 χειρότερα παιχνίδια. Κατά τα άλλα το πρόγραμμα λειτουργεί όπως και στις άλλες περιπτώσεις.

Παρακάτω δίνονται screenshots με παραδείγματα για κάθε μία από αυτές τις περιπτώσεις:

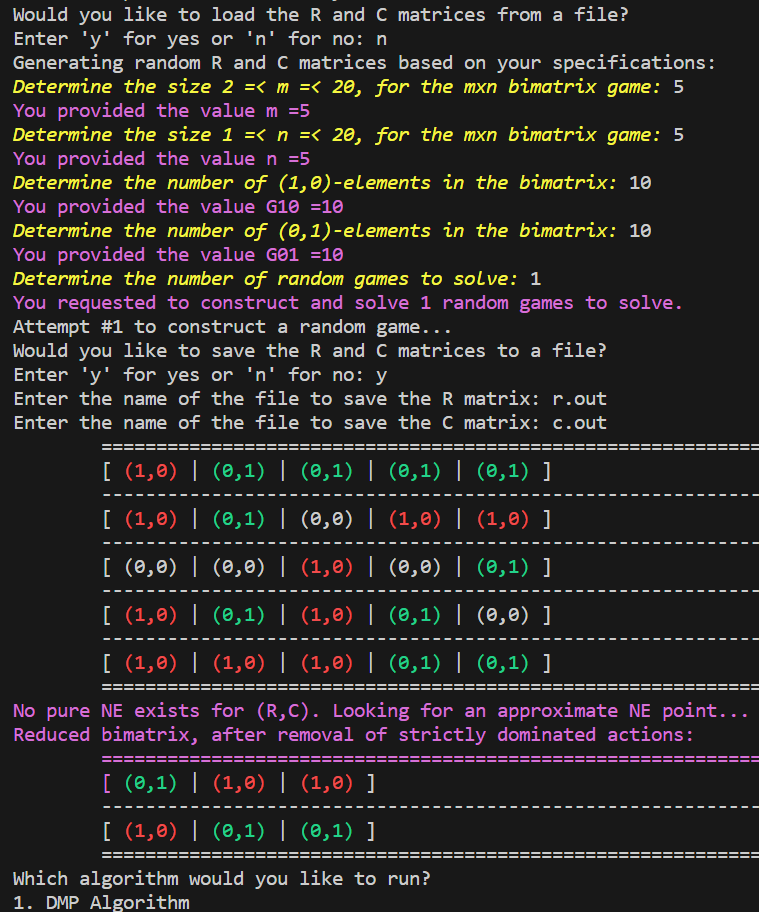
* Περίπτωση αποτυχημένης προσπάθειας δημιουργίας παιχνιδιού:



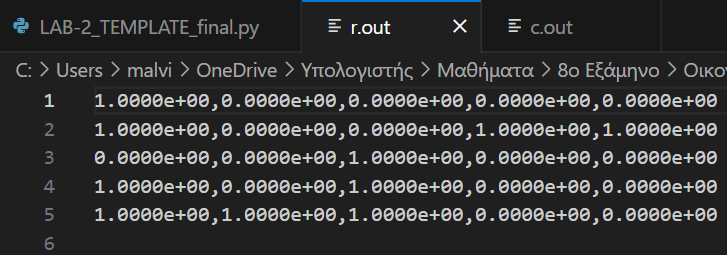
* Περίπτωση επιτυχημένης δημιουργίας μεμονωμένου παιχνιδιού χωρίς αποθήκευση (για λόγους συντομίας δεν παρουσιάζεται η επιλογή και εκτέλεση αλγορίθμου):

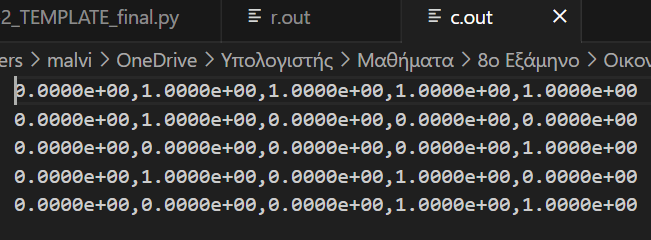


* Περίπτωση επιτυχημένης δημιουργίας μεμονωμένου παιχνιδιού με αποθήκευση (για λόγους συντομίας δεν παρουσιάζεται η επιλογή και εκτέλεση αλγορίθμου):

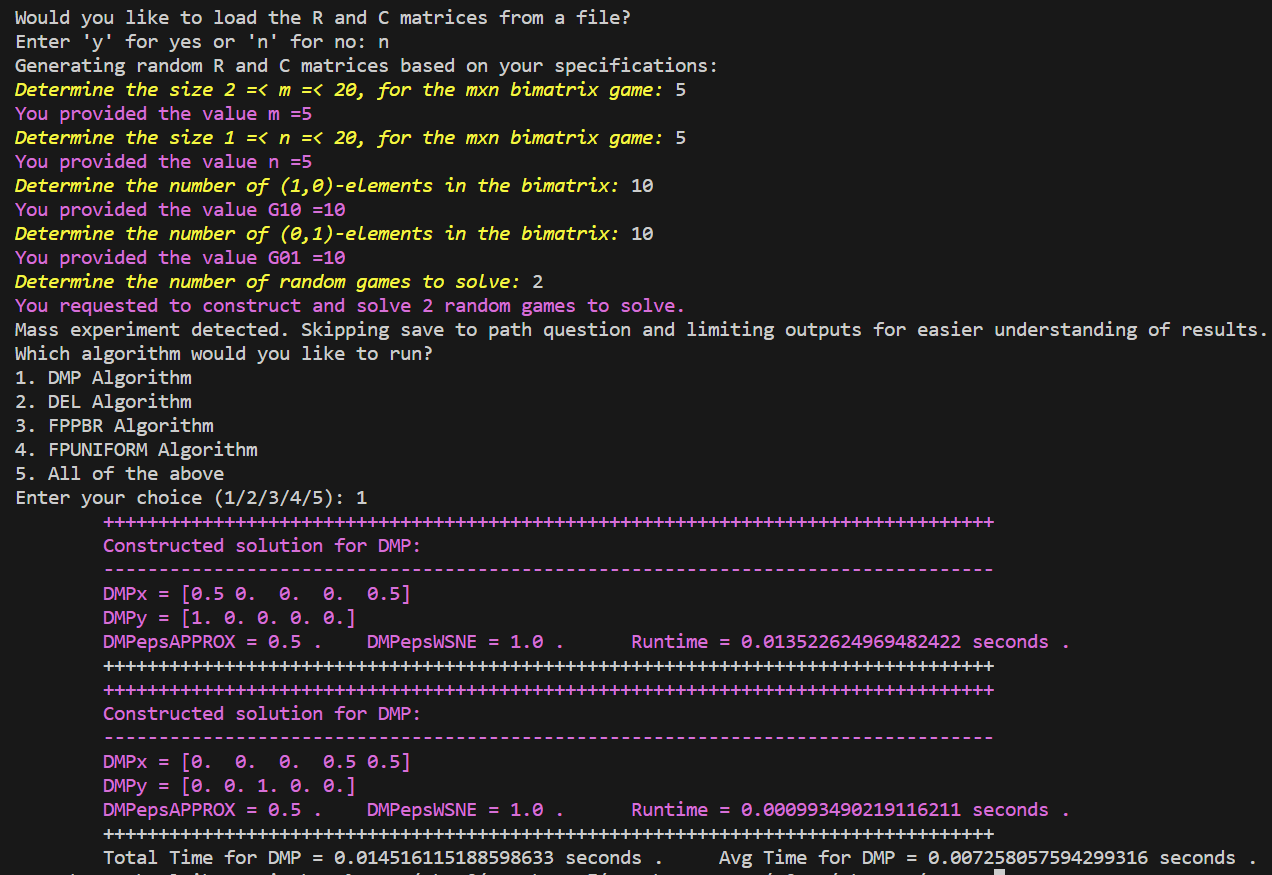


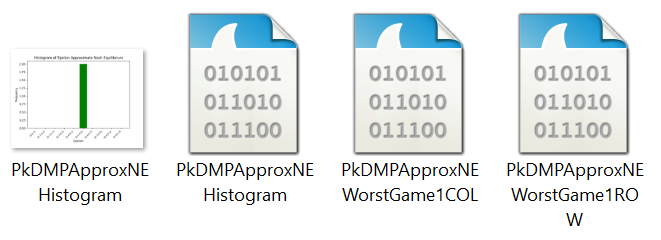
Από αυτήν την εκτέλεση παράγονται στο path του κώδικα τα αρχεία r.out και c.out, τα οποία απεικονίζουν τα μητρώα R και C που αντιστοιχούν στο παιχνίδι που κατασκευάστηκε (το αρχικό πριν την μείωση γραμμών/στηλών):

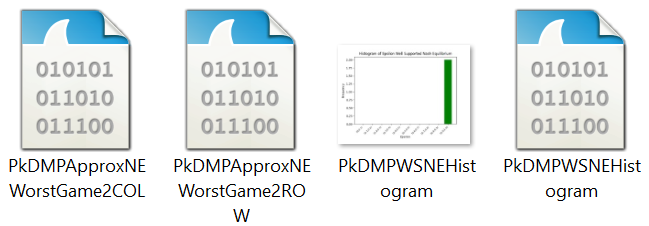




* Περίπτωση επιτυχημένης εκτέλεσης πειράματος. Στην συγκεκριμένη περίπτωση πλήθος παιχνιδιών = 2. Για λόγους συντομίας και επειδή θα αναλυθούν στην συνέχεια τα αποτελέσματα των πειραμάτων, παρουσιάζονται απλώς τα αρχεία που δημιουργούνται από αυτήν την εκτέλεση χωρίς να επεξηγούνται περαιτέρω. Επιλέχθηκε τυχαία εκτέλεση του DMP αλγορίθμου:





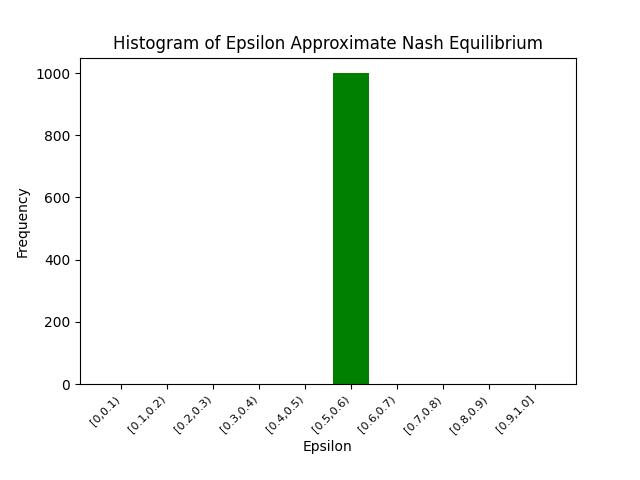


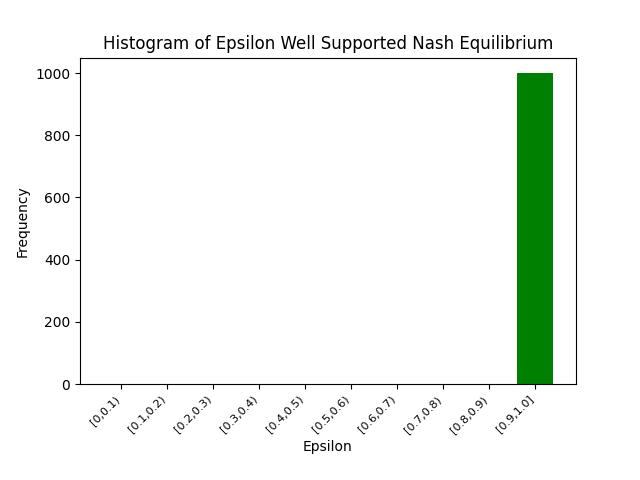
**Πειραματικό Μέρος**

Παρακάτω δίνονται τα ιστογράμματα για κάθε πείραμα ξεχωριστά. Για κάθε αλγόριθμο που παρατίθεται, το πάνω ιστόγραμμα θα αντιστοιχεί στο εapprox και το κάτω στο εwsne:

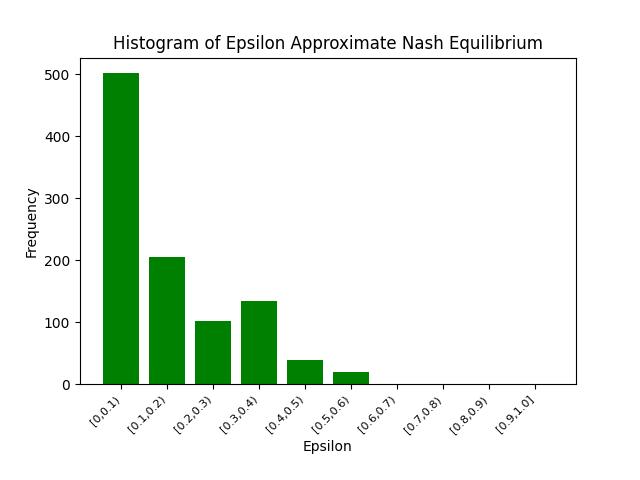
**ΠΕΙΡΑΜΑ 1**

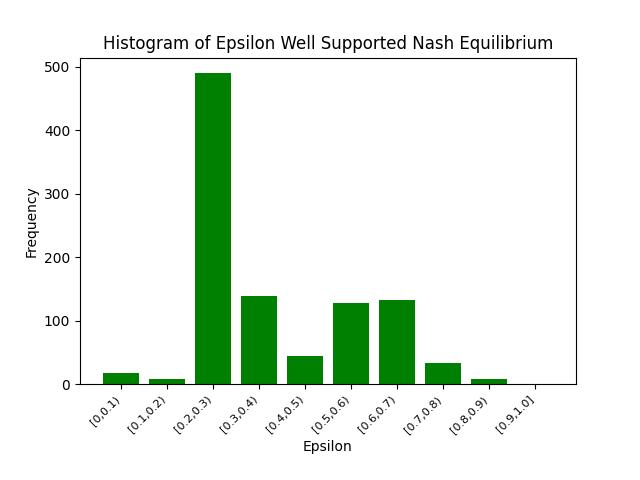
* **DMP:**



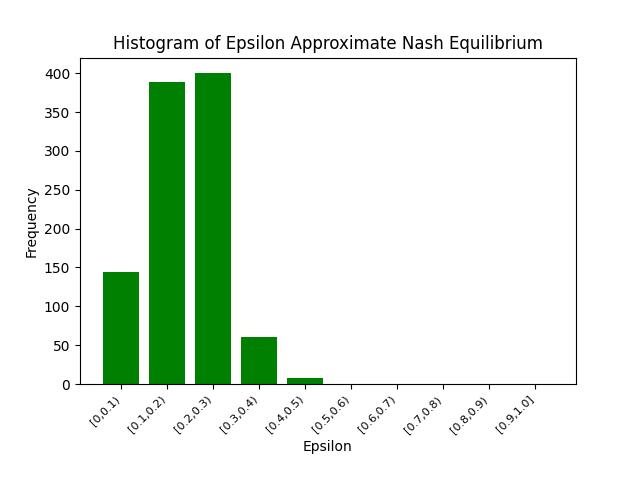


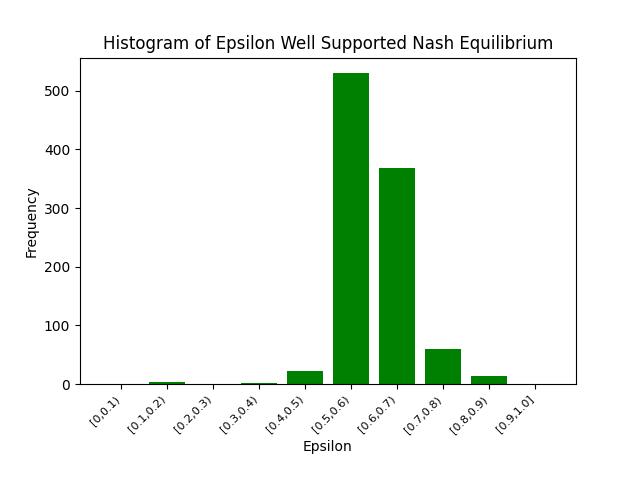
* **DEL:**



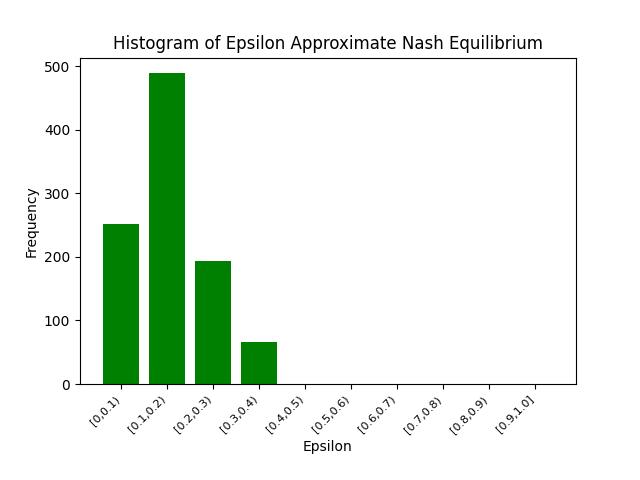


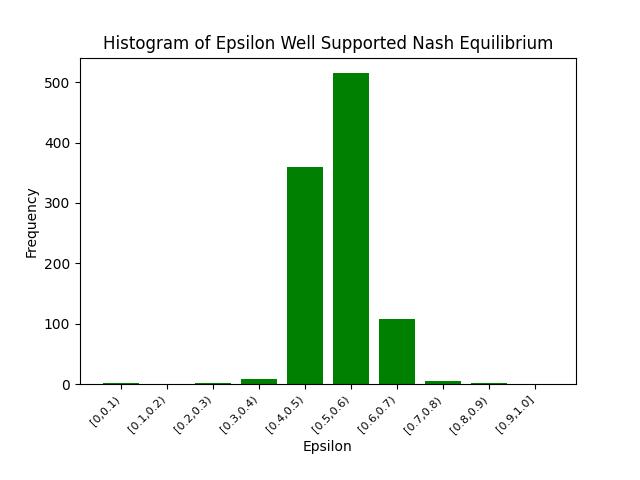
* **FPPBR:**





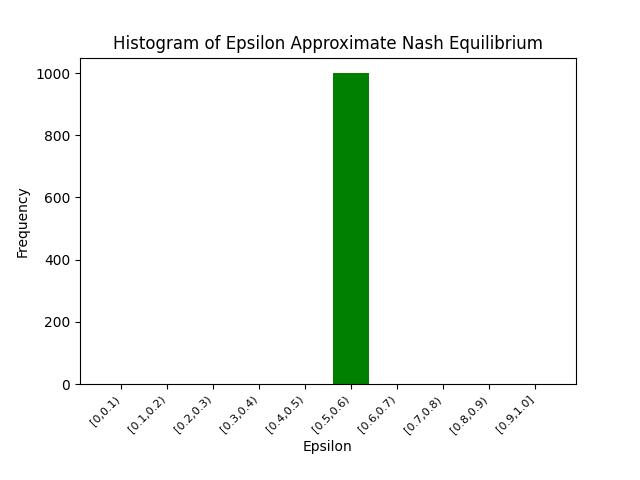
* **FPUNI:**

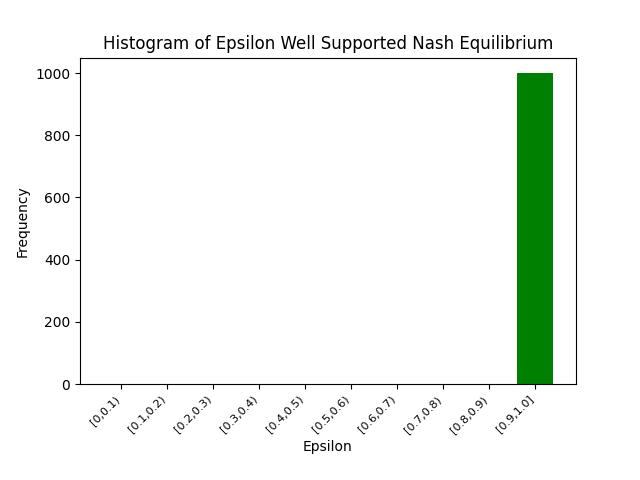




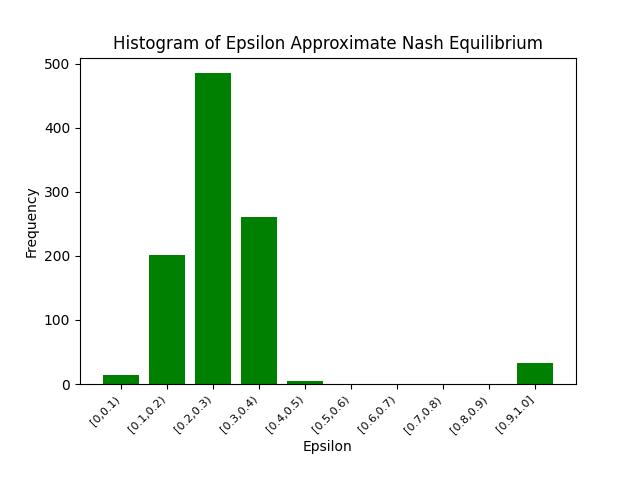
**ΠΕΙΡΑΜΑ 2**

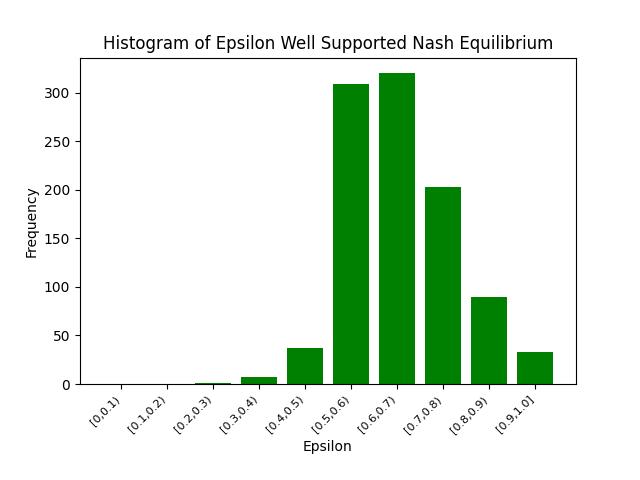
* **DMP:**



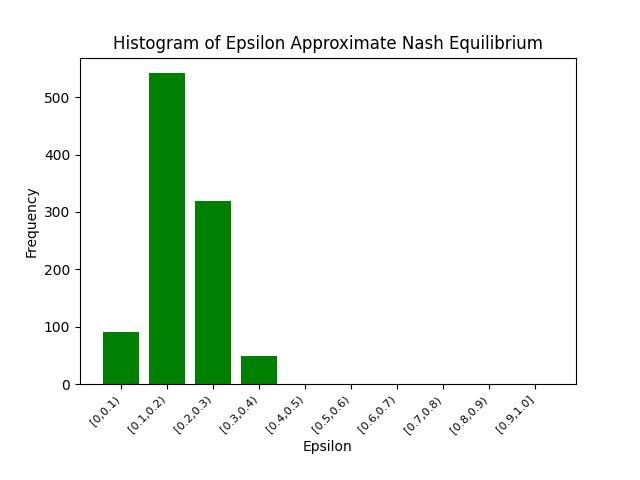


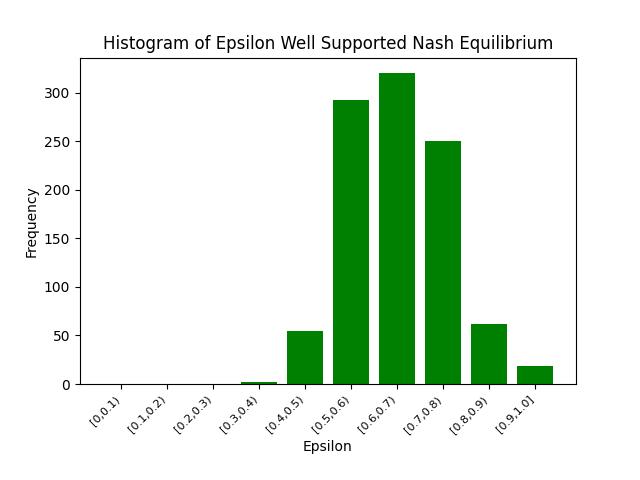
* **DEL:**



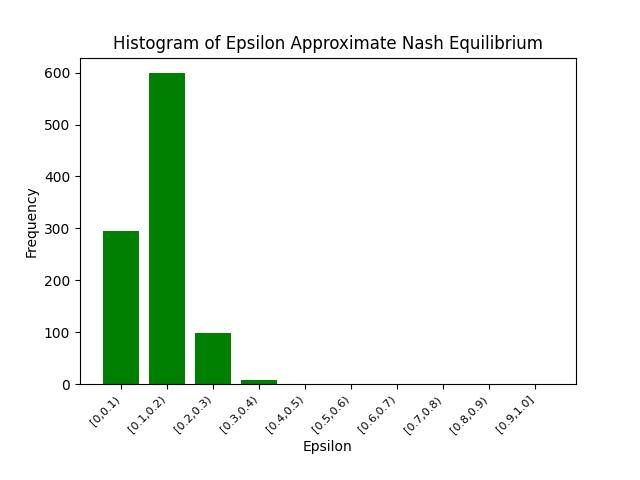


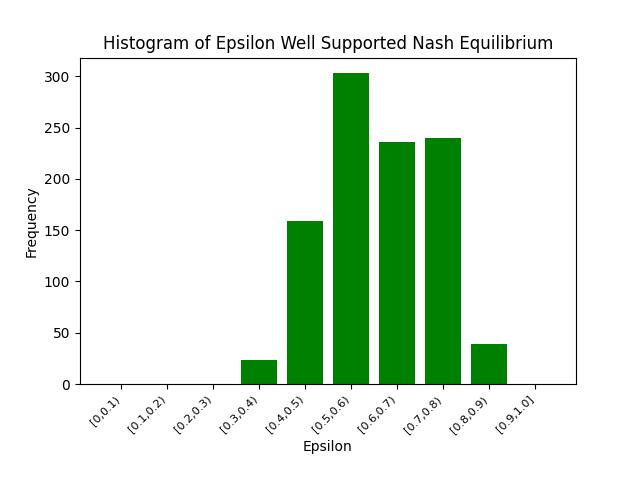
* **FPPBR:**





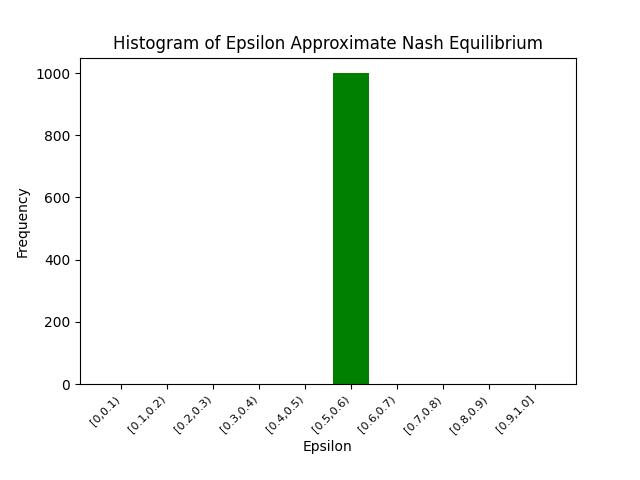
* **FPUNI:**

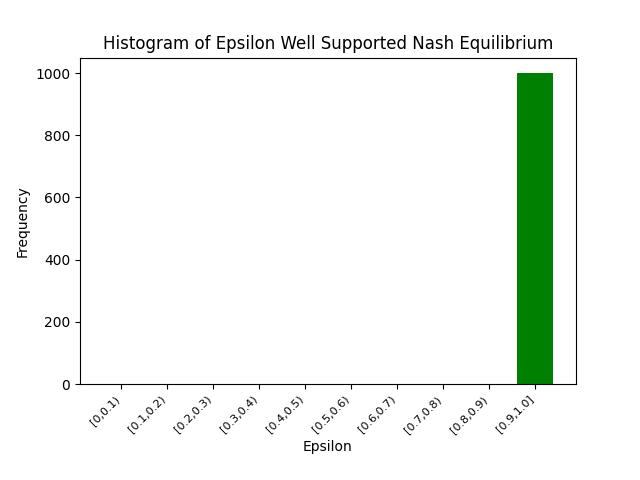




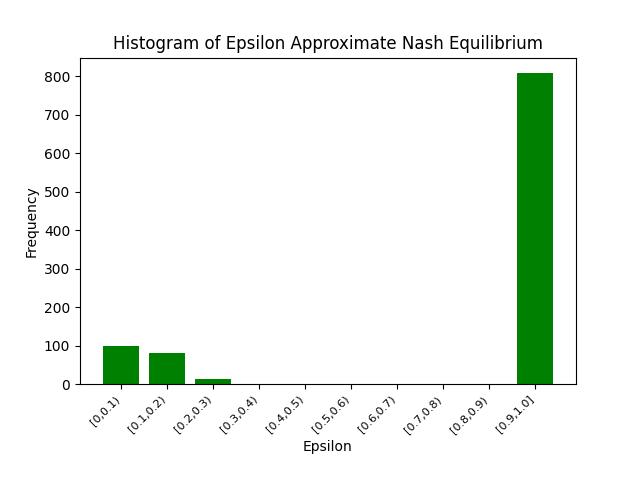
**ΠΕΙΡΑΜΑ 3**

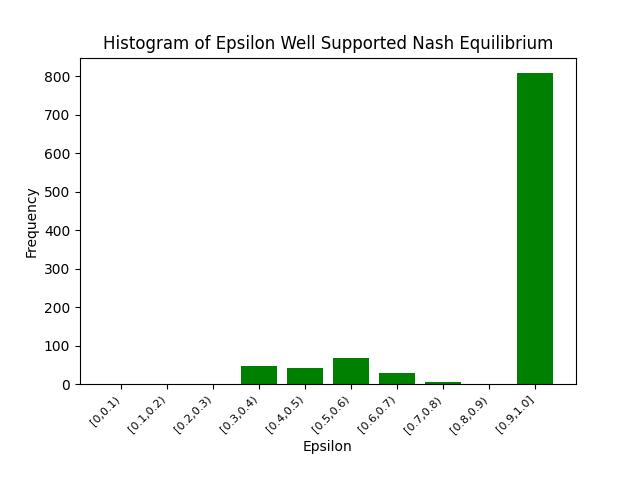
* **DMP:**



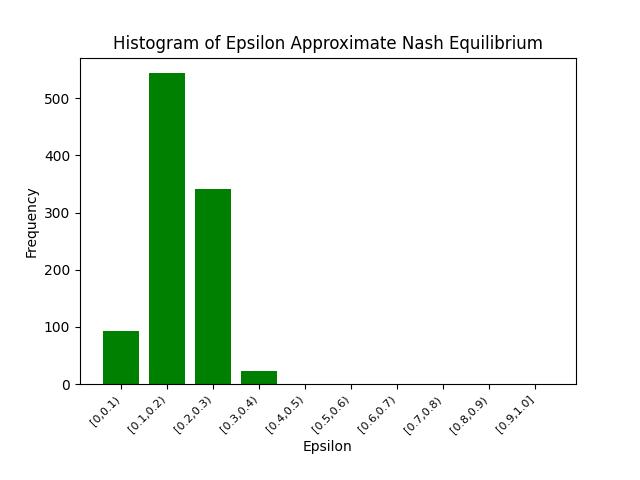


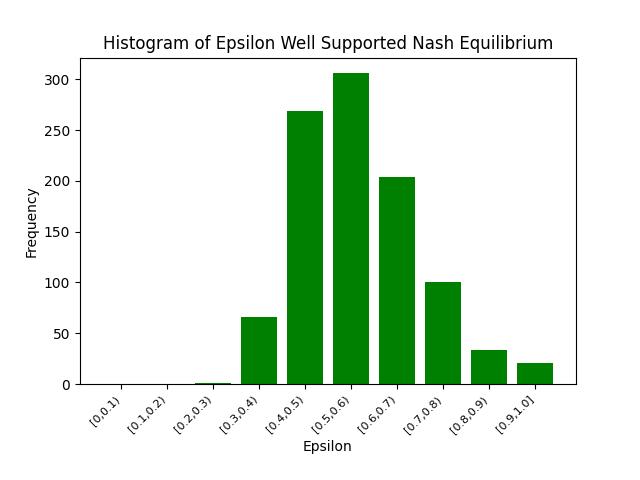
* **DEL:**



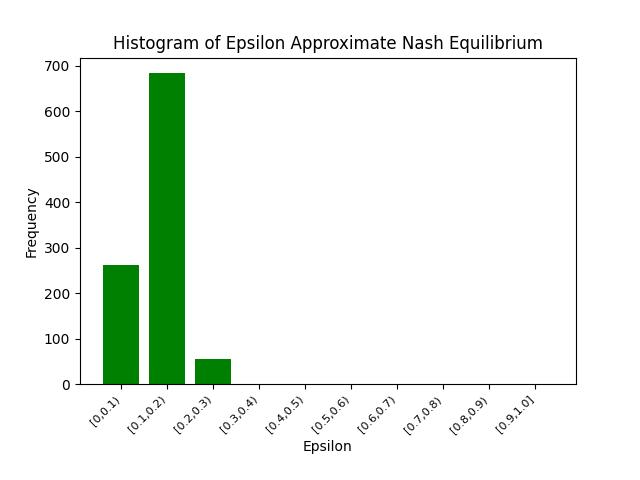


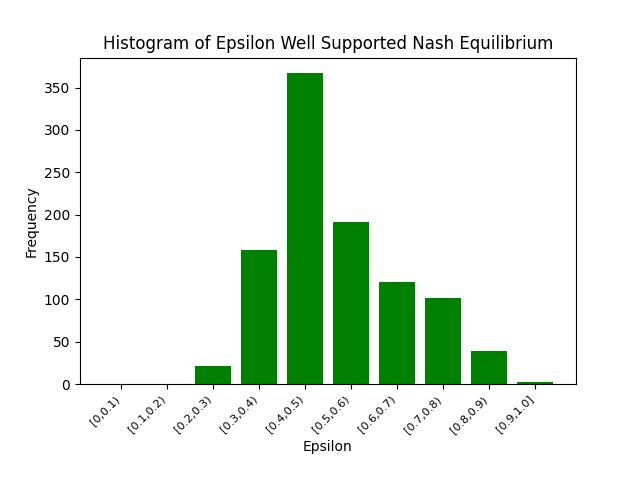
* **FPPBR:**





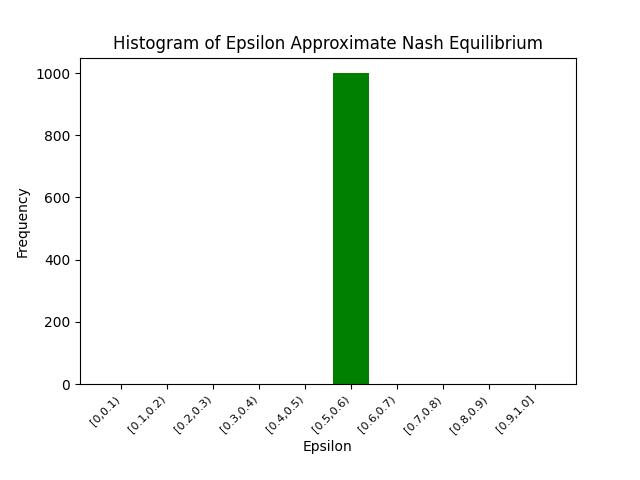
* **FPUNI:**

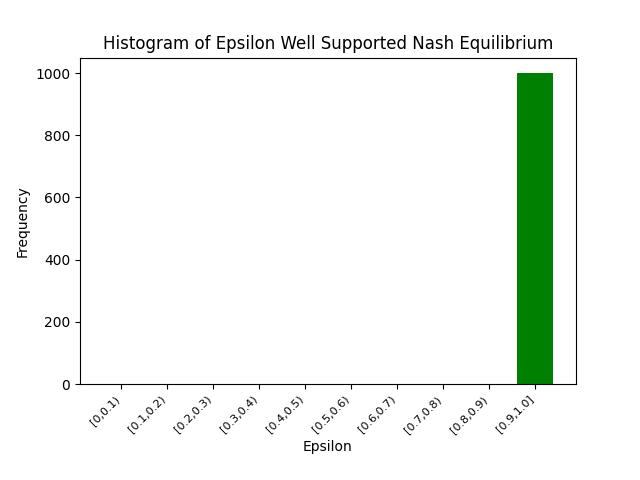




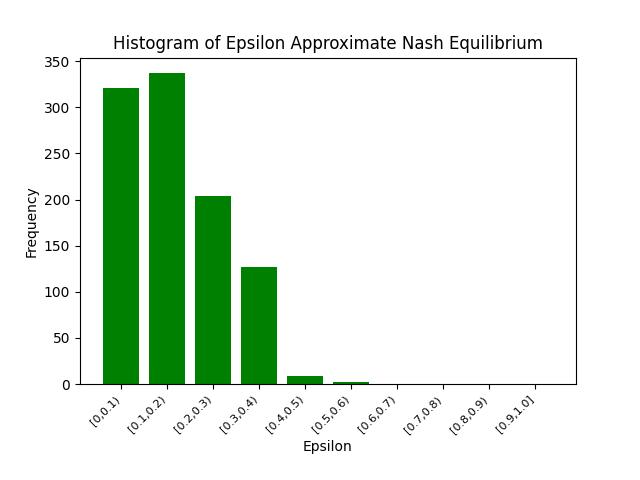
**ΠΕΙΡΑΜΑ 4**

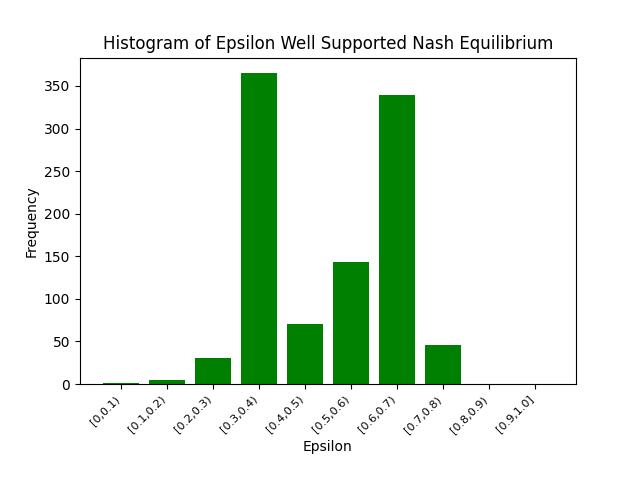
* **DMP:**



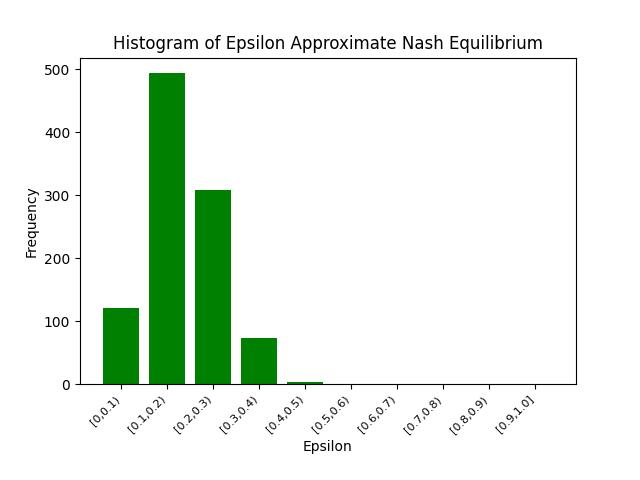


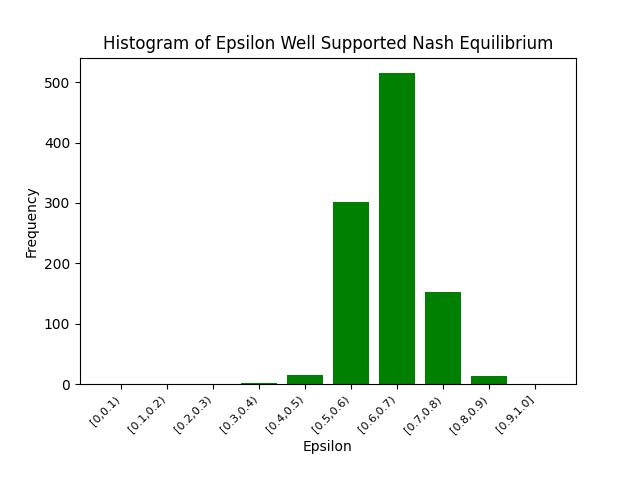
* **DEL:**



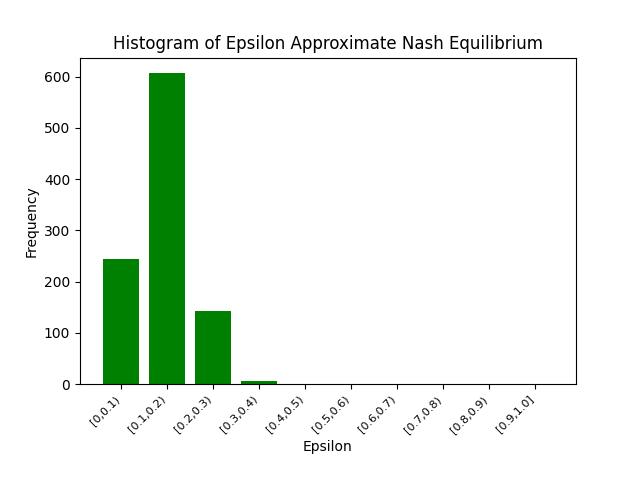


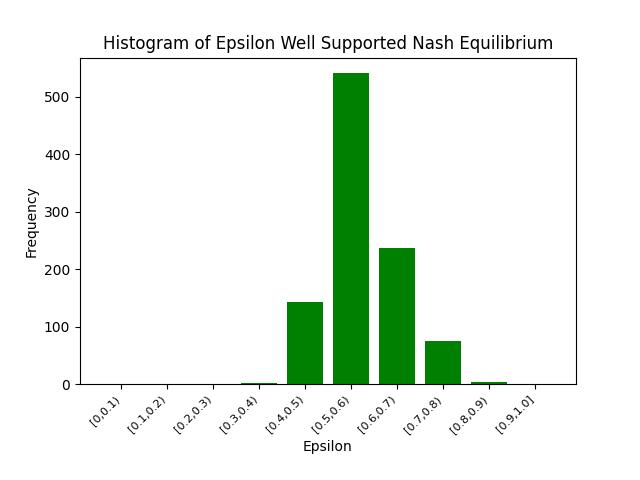
* **FPPBR:**





* **FPUNI:**

****



Όπως φαίνεται από τα παραπάνω ιστογράμματα, η επίδοση του DMP είναι σταθερή και ίδια σε κάθε πείραμα, με τιμές 0.5 και 1 για το εapprox και το εwsne αντίστοιχα. Παρατηρούμε επίσης πως οι δύο παραλλαγές του Fictitious Play δίνουν σε όλες τις περιπτώσεις τα βέλτιστα αποτελέσματα, το οποίο είναι μάλλον αναμενόμενο καθώς αυτοί οι αλγόριθμοι λειτουργούν “μαθαίνοντας” και προσαρμόζοντας κατάλληλα τις στρατηγικές των παικτών. Αντιθέτως, ο DMP σχεδόν πάντα δίνει τα χειρότερα αποτελέσματα στο μεγαλύτερο μέρος των μετρήσεων (αν όχι σε όλο), πιθανώς εξαιτίας της απλότητάς του σε σχέση με τους υπόλοιπους αλγορίθμους. Μοναδική εξαίρεση σε αυτό αποτελεί το Πείραμα 3, όπου η απόδοση του DEL είναι η χειρότερη.

Η απόκλιση αυτή μπορεί να οφείλεται στην μεγάλη διαφορά του πλήθους στοιχείων G10 και G01 στο πείραμα 3, μιας και ίσως να ευνοείται κάποιο συγκεκριμένο σενάριο του DEL που δεν παράγει ιδιαίτερα καλά αποτελέσματα. Πράγματι, συγκρίνοντας τα πειράματα μεταξύ τους παρατηρούμε ότι όσο η διαφορά στο πλήθος των στοιχείων αυξάνεται τόσο χειρότερη γίνεται η απόδοσή του. Οι υπόλοιποι αλγόριθμοι δεν φαίνεται να επηρεάζονται σημαντικά από αυτό.

Επίσης παρατηρούμε γενικά πως όσο το συνολικό πλήθος των στοιχείων G10 και G01 μειώνεται, τόσο τα ε τείνουν να μεγαλώνουν. Μια πιθανή εξήγηση για αυτό είναι ότι όσο “αραιώνουν” τα στοιχεία με κέρδος (για οποιονδήποτε από τους 2 παίκτες) στο διμητρώο τόσο πιο δύσκολο είναι να βρεθεί προσεγγιστική ισορροπία που δεν απαιτεί μεγάλο συμβιβασμό, δηλαδή δεν προσφέρει αρκετά μειωμένο κέρδος στους παίκτες σε σχέση με το θεωρητικό μέγιστο.

Τέλος, όπως φαίνεται τα αποτελέσματα του DEL δεν είναι τα αναμενόμενα, δηλαδή το εwsne πολλές φορές ξεπερνά το αναμενόμενο άνω όριο του 0.667. Αυτό οφείλεται σε λάθος μας είτε στην υλοποίηση του DEL, είτε πιο πιθανώς στην υλοποίηση του υπολογισμού του εwsne εντός της computeApproximationGuarantees.