**1η εργασία Οικονομικής Θεωρίας και Αλγορίθμων**

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

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

**Θεωρητικό Μέρος**

Παρακάτω εξετάζεται ξεχωριστά κάθε περίπτωση Ν κενών κελιών, όπου Ν ≤ 5. Χρησιμοποιείται αναπαράσταση με στοίβες, τόσο της αρχικής παρτίδας όσο και των παρτίδων που ακολουθούν και κληροδοτούνται διαδοχικά από παίκτη σε παίκτη. Το πράσινο βέλος υποδεικνύει κίνηση που γίνεται από τον πράσινο, ενώ το κόκκινο βέλος κίνηση που γίνεται από τον κόκκινο παίκτη. Σε κάθε περίπτωση θεωρείται ότι ξεκινά πρώτος ο πράσινος παίκτης.

* **Περίπτωση 0 κελιών:**

Πρόκειται για την παρτίδα (0), η οποία αποτελεί τετριμμένη **παρτίδα ήττας**, καθώς ο πράσινος δεν μπορεί να κάνει καμία κίνηση και άρα το παιχνίδι τελειώνει με νίκη του κόκκινου.

* **Περίπτωση 1 κελιού:**

Πρόκειται για την παρτίδα (1), η οποία αποτελεί τετριμμένη **παρτίδα νίκης**, καθώς ο πράσινος έχει μόνο μία πιθανή κίνηση (να γεμίσει το μοναδικό κενό κελί που έχει απομείνει) και η κίνηση αυτή οδηγεί σε νίκη του, αφού κληροδοτεί στον κόκκινο την τετριμμένη παρτίδα ήττας της περίπτωσης 0:

1. (0) , ΠΝ

* **Περίπτωση 2 κελιών:**

Από αυτό το σημείο και ύστερα εξετάζονται πλέον και διαφορετικές περιπτώσεις αρχικών στοιβών, καθώς υπάρχουν περισσότερες από μία πιθανές διατάξεις των κενών κελιών στο ταμπλό. Οι δύο μη-συμμετρικές διατάξεις για την περίπτωση 2 κενών κελιών είναι οι εξής:

* Μη διαδοχικά κελιά: είναι **παρτίδα ήττας**, καθώς ο πράσινος έχει δυνατότητα να γεμίσει μόνο ένα από τα 2 κενά κελιά, και άρα αναγκαστικά κληροδοτεί στον κόκκινο την τετριμμένη παρτίδα νίκης της περίπτωσης 1. Η αντίστοιχη αναπαράσταση με στοίβες δίνεται παρακάτω.

(1,1) (1) (0) , ΠΗ

* Διαδοχικά κελιά (όπου κανένα κελί δεν ανήκει στην κύρια διαγώνιο): είναι **παρτίδα νίκης**, καθώς ο πράσινος μπορεί με την κίνησή του να γεμίσει και τα 2 κενά κελιά, και άρα να κληροδοτήσει στον κόκκινο την τετριμμένη παρτίδα νίκης της περίπτωσης 0.

1. (0) , ΠΝ

Σημείωση: για την συγκεκριμένη περίπτωση διαφαίνεται μία ακόμα υποπερίπτωση, αυτή όπου τα 2 κελιά είναι διαδοχικά αλλά κάποιο ανήκει στην κύρια διαγώνιο. Αυτή η περίπτωση αποτελεί και πάλι παρτίδα ήττας για τον πράσινο, καθώς τα κελιά της διαγωνίου δεν μπορούν να παιχτούν διαδοχικά με άλλα κελιά και άρα θα πρέπει αναγκαστικά να γεμίσει μόνο ένα κελί. Η αναπαράστασή της μέσω στοιβών μπορεί να γίνει ως εξής:

(1,1) (1) (0) .

Παρατηρούμε δηλαδή πως, αν και εκ πρώτης όψεως μοιάζει με ξεχωριστή περίπτωση, ουσιαστικά μπορεί να αναχθεί στην πρώτη από τις 2 “βασικές” υποπεριπτώσεις και άρα δεν αποτελεί καινούργια διακριτή υποπερίπτωση.

Επιπλέον, στις περιπτώσεις που ακολουθούν οι υποπεριπτώσεις παρουσιάζονται κυρίως με βάση τις αρχικές στοίβες και όχι την ακριβή διάταξη των κελιών στον χώρο. Αυτό αφενός γιατί είναι αρκετά δύσκολο να εξεταστούν όλες οι πιθανές διατάξεις των κελιών (ειδικά για την περίπτωση με τα 5 κενά κελιά), και αφετέρου επειδή κάθε περίπτωση διάταξης μπορεί τελικά να αναχθεί σε κάποια από τις “βασικές” υποπεριπτώσεις (όπως έγινε και με την προηγούμενη). Εξαιτίας αυτού δεν θα ξαναγίνει επίσης ξεχωριστή μελέτη για την κύρια διαγώνιο, και όταν αναφέρονται διαδοχικά κελιά θα θεωρείται πως κανένα από αυτά δεν βρίσκεται πάνω σε αυτήν.

* **Περίπτωση 3 κελιών:**

Παρακάτω δίνονται οι 3 μη-συμμετρικές υποπεριπτώσεις αρχικών στοιβών, καθώς και μία πιθανή διάταξη των κελιών για την καθεμία:

* (3): πρόκειται για μία μοναδική στοίβα με 3 κελιά, και η μοναδική διάταξη που είναι δυνατόν να αντιστοιχεί σε αυτήν είναι τα 3 διαδοχικά κελιά. Είναι **παρτίδα νίκης** καθώς ο πράσινος μπορεί με την κίνησή του να γεμίσει και τα 3 κελιά, κληροδοτώντας στον κόκκινο την τετριμμένη παρτίδα ήττας (0):

1. (0) , ΠΝ

* (2,1): πρόκειται για 2 στοίβες, μία με ένα και μία με 2 κενά κελιά. Μια πιθανή διάταξη για αυτή την αναπαράσταση είναι η περίπτωση 2 διαδοχικών κελιών και ενός “σπαστού” κελιού (δηλαδή ενός κελιού που δεν είναι διαδοχικό με κανένα άλλο). Είναι **παρτίδα νίκης**, καθώς ο πράσινος μπορεί, γεμίζοντας το κατάλληλο κελί, να κληροδοτήσει στον κόκκινο την παρτίδα (1,1). Η παρτίδα (1,1) αποτελείται από 2 στοίβες ίδιου μεγέθους, και άρα όπως γνωρίζουμε από την θεωρία (και από την αντίστοιχη περίπτωση για 2 κενά κελιά) αποτελεί παρτίδα ήττας για τον παίκτη που θα παίξει σε αυτήν.

(2,1) (1,1) (1) (0) , ΠΝ

* (1,1,1): πρόκειται για 3 στοίβες που αποτελούνται η καθεμία από ένα κελί, και μία πιθανή διάταξη για αυτήν είναι η περίπτωση όπου υπάρχουν 3 σπαστά κελιά. Είναι **παρτίδα νίκης**, καθώς όμοια με πριν ο πράσινος μπορεί να κληροδοτήσει στον κόκκινο την παρτίδα ήττας (1,1).

(1,1,1) (1,1) (1) (0) , ΠΝ

Όπως διαπιστώνουμε από την παραπάνω ανάλυση, η περίπτωση τριών κενών κελιών αποτελεί πάντα παρτίδα νίκης για τον παίκτη που παίζει σε αυτήν, εφόσον αυτός παίζει βέλτιστα.

* **Περίπτωση 4 κελιών:**

Παρακάτω δίνονται οι 5 μη-συμμετρικές υποπεριπτώσεις αρχικών στοιβών, καθώς και μία πιθανή διάταξη των κελιών για την καθεμία:

* (4): πρόκειται για μία μοναδική στοίβα με 4 κελιά, και μπορεί να αντιστοιχεί μόνο στην περίπτωση τεσσάρων διαδοχικών κελιών. Είναι **παρτίδα νίκης**, καθώς ο πράσινος μπορεί να γεμίσει τα 2 διαδοχικά μη-ακριανά κελιά, κληροδοτώντας στον κόκκινο δύο σπαστά κελιά, δηλαδή την παρτίδα ήττας (1,1).

(4) (1,1) (1) (0) , ΠΝ

* (3,1): πρόκειται για 2 στοίβες, μία με ένα και μία με 3 κελιά. Μια πιθανή διάταξη για αυτήν είναι 3 διαδοχικά κελιά και ένα σπαστό. Είναι **παρτίδα νίκης**, καθώς ο πράσινος μπορεί να γεμίσει 2 από τα 3 διαδοχικά κελιά, και όμοια με πριν να κληροδοτήσει την παρτίδα ήττας (1,1) στον αντίπαλο.

(3,1) (1,1) (1) (0) , ΠΝ

* (2,2): πρόκειται για 2 στοίβες των 2 κελιών η καθεμία. Μία πιθανή διάταξη για την συγκεκριμένη περίπτωση είναι δύο ζεύγη διαδοχικών κελιών, όπου τα δύο ζεύγη δεν τέμνονται μεταξύ τους με κανέναν τρόπο. Γνωρίζουμε από την θεωρία πως πρόκειται για **παρτίδα ήττας**, αφού αποτελείται από 2 στοίβες ίδιου μεγέθους. Πράγματι, ο κόκκινος μπορεί να κερδίσει το παιχνίδι παίζοντας ως “καθρέφτης”, αντιγράφοντας δηλαδή την προηγούμενη κίνηση του πράσινου. Παρακάτω δίνονται οι δύο πιθανές εξελίξεις της παρτίδας, οι οποίες όπως είναι αναμενόμενο καταλήγουν και οι δύο σε ήττα για τον πράσινο:

(2,2) (2,1) (1,1) (1) (0) , ΠΗ

(2,2) (2) (0) , ΠΗ

* (2,1,1): μία στοίβα 2 κελιών και 2 στοίβες με ένα κελί η καθεμία. Μπορεί να αντιστοιχεί σε διάταξη 2 συνεχόμενων και 2 σπαστών κελιών, και αποτελεί **παρτίδα νίκης**, αφού ο πράσινος γεμίζοντας τα δύο διαδοχικά κελιά μπορεί να κληροδοτήσει στον κόκκινο την παρτίδα ήττας (1,1).

(2,1,1) (1,1) (1) (0) , ΠΝ

* (1,1,1,1): 4 στοίβες των τεσσάρων κελιών η καθεμία, το οποίο μπορεί να αντιστοιχεί σε διάταξη τεσσάρων σπαστών κελιών. Είναι **παρτίδα ήττας**, καθώς οι παίκτες αναγκαστικά θα παίζουν από ένα κελί την φορά και το αρχικό πλήθος των κελιών είναι άρτιος αριθμός.

(1,1,1,1) (1,1,1) (1,1) (1) (0) , ΠΗ

* **Περίπτωση 5 κελιών:**

Παρακάτω δίνονται οι 7 μη-συμμετρικές υποπεριπτώσεις αρχικών στοιβών, καθώς και μία πιθανή διάταξη των κελιών για την καθεμία:

* (5): πρόκειται για μία στοίβα με 5 κελιά, και η μοναδική διάταξη που μπορεί να της αντιστοιχεί είναι να υπάρχουν 5 διαδοχικά κενά κελιά. Πρόκειται για **παρτίδα νίκης**, καθώς ο πράσινος μπορεί να γεμίσει τα 3 μη-ακριανά κελιά, και άρα να κληροδοτήσει στον κόκκινο την παρτίδα ήττας (1,1).

(5) (1,1) (1) (0) , ΠΝ

* (4,1): μία στοίβα τεσσάρων κελιών και μία στοίβα ενός κελιού, το οποίο μπορεί να πρόκειται για διάταξη τεσσάρων διαδοχικών κελιών και ενός σπαστού. Πρόκειται για **παρτίδα νίκης**, αφού και πάλι ο πράσινος μπορεί να γεμίσει τα κατάλληλα 3 κελιά από τα 4 διαδοχικά, αφήνοντας στον κόκκινο την παρτίδα ήττας (1,1).

(4,1) (1,1) (1) (0) , ΠΝ

* (3,2): πρόκειται για 2 στοίβες, μία των δύο και μία των τριών κελιών. Μπορεί να αντιστοιχεί σε διάταξη 2 διαδοχικών μεταξύ τους κελιών και τριών διαδοχικών κελιών, όπου τα δύο και τρία διαδοχικά κελιά δεν τέμνονται μεταξύ τους με κανέναν τρόπο. Είναι **παρτίδα νίκης**, καθώς ο πράσινος γεμίζοντας ένα ακριανό κελί από τα 3 διαδοχικά κληροδοτεί στον κόκκινο την παρτίδα (2,2), η οποία όπως έχει αναλυθεί ήδη αποτελεί παρτίδα ήττας. Παρακάτω δίνονται οι δύο πιθανές εξελίξεις της παρτίδας, οι οποίες όπως αναμένεται καταλήγουν και οι δύο σε νίκη του πράσινου:

(3,2) (2,2) (2,1) (1,1) (1) (0) , ΠΝ

(3,2) (2,2) (2) (0) , ΠΝ

* (3,1,1): μία στοίβα τριών κελιών και δύο στοίβες του ενός κελιού η καθεμία, το οποίο πιθανώς αντιστοιχεί σε διάταξη τριών διαδοχικών κελιών και δύο σπαστών. Είναι **παρτίδα νίκης**, αφού ο πράσινος γεμίζοντας τα 3 διαδοχικά κελιά κληροδοτεί στον κόκκινο την παρτίδα ήττας (1,1).

(3,1,1) (1,1) (1) (0) , ΠΝ

* (2,2,1): πρόκειται για 2 στοίβες των δύο κελιών και μία στοίβα του ενός. Μία πιθανή διάταξη που της αντιστοιχεί είναι 2 ζεύγη διαδοχικών κελιών τα οποία δεν τέμνονται μεταξύ τους με κανέναν τρόπο, και επιπλέον ένα σπαστό κελί. Πρόκειται για **παρτίδα νίκης**, αφού γεμίζοντας το σπαστό κελί ο πράσινος αφήνει στον κόκκινο την γνωστή παρτίδα ήττας (2,2). Παρακάτω δίνονται οι δύο πιθανές εξελίξεις της παρτίδας, οι οποίες όπως αναμένεται καταλήγουν και οι δύο σε νίκη του πράσινου:

(2,2,1) (2,2) (2,1) (1,1) (1) (0) , ΠΝ

(2,2,1) (2,2) (2) (0) , ΠΝ

* (2,1,1,1): μία στοίβα 2 κελιών και 3 στοίβες του ενός κελιού η καθεμία. Μπορεί να πρόκειται για διάταξη με 2 διαδοχικά κελιά και 3 σπαστά και αποτελεί **παρτίδα νίκης**, καθώς γεμίζοντας ένα από τα δύο διαδοχικά κελιά ο πράσινος κληροδοτεί στον κόκκινο την παρτίδα (1,1,1,1), η οποία όπως έχει αναλυθεί ήδη στην περίπτωση 4 είναι παρτίδα ήττας.

(2,1,1,1) (1,1,1,1) (1,1,1) (1,1) (1) (0) , ΠΝ

* (1,1,1,1,1): πέντε στοίβες του ενός κελιού η καθεμία, το οποίο μπορεί να αντιστοιχεί σε διάταξη με πέντε σπαστά κελιά. Εφόσον οι παίκτες θα παίζουν αναγκαστικά ένα κελί την φορά και ο αρχικός αριθμός των κελιών είναι περιττός, πρόκειται για **παρτίδα νίκης** του πράσινου.

(1,1,1,1,1) (1,1,1,1) (1,1,1) (1,1) (1) (0) , ΠΝ

Όπως διαπιστώνουμε από την παραπάνω ανάλυση, η περίπτωση πέντε κενών κελιών αποτελεί πάντα παρτίδα νίκης για τον παίκτη που παίζει σε αυτήν, εφόσον αυτός παίζει βέλτιστα.

**Προγραμματιστικό Μέρος**

Παραδοχές που αφορούν συναρτήσεις που δίνονταν έτοιμες στο template:

* + Αλλάξαμε την isBoardFull και ο νέος της κώδικας είναι όπως φαίνεται παρακάτω.

def isBoardFull(*board*, *N*):

*# Function for checking if the board is full*

    for i in range(1, *N*\**N* + 1):

*# if the cell does not contain G or R then it is empty*

        if *board*[i] != 'G' and *board*[i] != 'R':

            return False  *# return False if there is an empty cell*

    return True

Επιστρέφει δηλαδή True αν όλα τα tiles του πίνακα έχουν γραφτεί από πάνω με το γράμμα G ή R. Αυτό δείχνει ότι κάποιος από τους δυο παίκτες το έχει χρησιμοποιήσει σε κίνησή του, συνεπώς αν όλα τα κελιά είναι “γραμμένα” τότε το ταμπλό είναι γεμάτο.

* + Δεν χρησιμοποιήσαμε την δοσμένη συνάρτηση getRowAndColumn, μιας και όταν δόθηκε είχαμε φτάσει σε σημείο στο πρόγραμμά μας όπου είχαμε υλοποιήσει ήδη την λειτουργία της με άλλες συναρτήσεις.

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 ο τρόπος που παίζεται το παιχνίδι, το πρόγραμμά μας τρέχει τις παρακάτω γραμμές κώδικα πριν αρχίσουν τα input του χρήστη.

maxNumMoves = 3

playNewGameFlag = True

while playNewGameFlag:

if not startNewGame():

break

N = getBoardSize()

nimBoard = initializeBoard(N)

initialNimBoard = nimBoard.copy()

playerLetter, computerLetter = inputPlayerLetter()

turn = whoGoesFirst()

diagonalCells = getDiagonalCells(nimBoard)

computerStrategy = howComputerPlays()

Αρχικά δηλώνουμε τον μέγιστο αριθμό κινήσεων (πόσα κελιά μπορεί να παίξει ο υπολογιστής ή ο παίκτης σε μία κίνηση) και ένα flag που χρησιμοποιείται για να ξέρουμε ποτέ ο παίκτης επιλέγει να αρχίσει το παιχνίδι από την αρχή. Στη συνέχεια το Ν (το μέγεθος του board) αλλά και το ίδιο το board oπως θα το τυπώνουμε αρχικοποιούνται. Όπως φαίνεται έχουμε ορίσει ένα ακόμα board ονόματι initialNimBoard, το οποίο θα έχει την μορφή του αρχικού board καθ’ όλη την διάρκεια του παιχνιδιού (μιας και ο πίνακας nimBoard αλλάζει κατά την εκτέλεση του προγράμματος). Ύστερα δηλώνονται οι μεταβλητές που με βάση τις εισαγωγές του χρήστη στα αρχικά μηνύματα αποθηκεύουν ποιο γράμμα θα έχει ο παίκτης και ποιο ο υπολογιστής, καθώς και ποιος θα παίξει πρώτος, το οποίο αποφασίζεται από την whoGoesFirst (με βάση αυτήν η πιθανότητα του καθενός να παίξει πρώτος είναι 50%). Ο πίνακας diagonalCells είναι ένας πίνακας που με την βοήθεια της συνάρτησης getDiagonalCells βρίσκει και αποθηκεύει τα κελιά που είναι στην κύρια διαγώνιο του board, όπως φαίνεται παρακάτω και στην getDiagonalCells. Τέλος η μεταβλητή computerStrategy επιστρέφει το όνομα της στρατηγικής που θα χρησιμοποιήσει ο υπολογιστής, το οποίο δίνεται από τον χρήστη στην αρχή του προγράμματος.

*# define a function that returns the name of the cells in the diagonal of the nimboard*

def getDiagonalCells(*nimBoard*):

*# get the length of the nimboard*

length = len(*nimBoard*)

*# get the number of rows and columns*

rows = int(math.sqrt(length))

*# create a list of the diagonal cells*

diagonalCells = []

*# create a list of the diagonal cells*

for i in range(0, rows):

cell\_name = *nimBoard*[i\*rows + i + 1]

number = re.findall(r'\d+', cell\_name)[0]

diagonalCells.append(int(number))

return diagonalCells

Συνεχίζουμε τώρα αφού ο χρήστης έχει υποβάλει τις προτιμήσεις του (μέγεθος του board, στρατηγική του υπολογιστή και χρώμα που προτιμά να έχει).

playermoves = []

*# provided this code make the computer play with the random strategy*

    while not isBoardFull(nimBoard, N):

        if turn == 'player':

            playermoves = []

*# Player's turn.*

            move = getPlayerMove(nimBoard, diagonalCells, initialNimBoard)

            while (move == False):

                move = getPlayerMove(

                    nimBoard, diagonalCells, initialNimBoard)

            playermoves.extend(move)

            makeMove(nimBoard, playerLetter, move)

            drawNimPalette(nimBoard, N)

            if (isBoardFull(nimBoard, N) is True):

                drawNimPalette(nimBoard, N)

                print(

                    bcolors.HEADER + 'Congrats you have beaten the computer!' + bcolors.ENDC)

                break

            else:

                turn = 'computer'

        else:

*# Computer's turn.*

            move = getComputerMove(

                nimBoard, computerStrategy, initialNimBoard, playermoves)

            print(move)

            makeMove(nimBoard, computerLetter, move)

            drawNimPalette(nimBoard, N)

            if (isBoardFull(nimBoard, N) is True):

                drawNimPalette(nimBoard, N)

                print(

                    bcolors.HEADER + 'The computer has beaten you! You lose.' + bcolors.ENDC)

                break

            else:

                turn = 'player'

Αρχικά έχουμε μια λίστα που αρχικά είναι κενή (playermoves). Αυτή η λίστα θα χρησιμοποιηθεί για την στρατηγική copycat, αποθηκεύοντας την κίνηση που έκανε ο παίκτης στην αμέσως προηγούμενη σειρά του.

Έστω ότι είναι η σειρά του παίκτη να παίξει. Τότε καλείται αρχικά η συνάρτηση getPlayerMove, ο κώδικας της οποίας δίνεται παρακάτω.

def getPlayerMove(*nimBoard*, *diagonalCells*, *initialNimBoard*):

*# This function returns the player's move.*

*# the player will choose a tile that is empty and will fill it with his label. Then if he chooses to continue he will choose another tile and so on until he has picked 3 consecutive tiles in the same row or column*

possibleMoves = getAvailableCells(*nimBoard*)

choice = []

while (1):

print(bcolors.QUESTION +

'[Q4] Which is the first tile you choose:' + bcolors.ENDC)

num = input()

if num.isdigit():

num = int(num)

if num in possibleMoves: *# if the selected tile is available*

choice.append(num)

if num in *diagonalCells*: *# if a main diagonal cell is chosen as the first move, the player can choose no more tiles*

return choice

while (1):

print(bcolors.QUESTION +

'[Q5] Which is the second tile you choose:' + bcolors.ENDC)

num1 = input()

if num1.isdigit():

num1 = int(num1)

if num1 in possibleMoves and checkValidMove(*initialNimBoard*, choice + [num1]):

choice.append(num1)

while (1):

print(bcolors.QUESTION +

'[Q6] Which is the third tile you choose:' + bcolors.ENDC)

num2 = input()

if num2.isdigit():

num2 = int(num2)

if num2 in possibleMoves and checkValidMove(*initialNimBoard*, choice + [num2]):

choice.append(num2)

if checkValidMove2(choice):

return choice

else:

print(bcolors.ERROR +

'ERROR 8: The tiles you chose were not consecutive. Try again from the beginning...' + bcolors.ENDC)

return False

else:

print(bcolors.ERROR +

'ERROR 6: The tile you chose is not available or invalid. Try again...' + bcolors.ENDC)

else:

print(bcolors.MSG +

'Your input was not an integer so your turn was terminated.' + bcolors.ENDC)

if checkValidMove2(choice):

print(bcolors.GREEN +

'Move executed succesfully!' + bcolors.ENDC)

return choice

else:

print(bcolors.ERROR +

'ERROR 8: The tiles you chose were not consecutive. Try again from the beginning...' + bcolors.ENDC)

return False

else:

print(bcolors.ERROR +

'ERROR 5: The tile you chose is not available or invalid. Try again...' + bcolors.ENDC)

else:

print(bcolors.MSG +

'Your input was not an integer so your turn was terminated.' + bcolors.ENDC)

print(bcolors.GREEN +

'Move executed succesfully!' + bcolors.ENDC)

return choice

else:

print(bcolors.ERROR +

'ERROR 6: The tile you chose is not available. Try again...' + bcolors.ENDC)

else:

print(bcolors.ERROR +

'ERROR 7: Your input was not an integer. Please enter a valid choice!' + bcolors.ENDC)

Η συνάρτηση αυτή, μαζί με κάποιες άλλες που καλούνται μέσα από αυτή, είναι υπεύθυνη να παίρνει τις κινήσεις που εισάγει ο παίκτης και να ελέγχει αν είναι νόμιμες. Αν δεν είναι, τότε τυπώνονται τα αντίστοιχα μηνύματα λάθους και είτε του δίνεται η ευκαιρία να προσπαθήσει ξανά είτε ολοκληρώνεται η σειρά του και κρατιούνται οι επιλογές που έχει κάνει μέχρι τότε (ανάλογα το λάθος).

Ας αρχίσουμε την ανάλυση αυτής της συνάρτησης. Αρχικά καλούμε την συνάρτηση getAvailableCells που παίρνει ως όρισμα το εκάστοτε τρέχον board. Αυτή η συνάρτηση επιστρέφει κάθε φορά που καλείται μια λίστα με όλα τα κελιά του board που είναι ελεύθερα και μπορεί να παίξει είτε ο παίκτης είτε ο υπολογιστής. Στην συνέχεια δημιουργείται μια λίστα ονόματι choice, που θα είναι η λίστα που επιστρέφει η συνάρτηση getPlayerMove και θα περιέχει τα κελιά της κίνησης του παίκτη. Όπως φαίνεται όλη η υπόλοιπη συνάρτηση είναι μέσα σε μια while, μέσα στην οποία υπάρχουν άλλες παρόμοιες while. Αυτό γίνεται γιατί το πρόγραμμα έχει φτιαχτεί έτσι ώστε, αναλόγως το θέμα που παρουσιάζεται με τις επιλογές του χρήστη, να του δίνεται σε κάποιες περιπτώσεις η επιλογή να επαναλάβει όλη την κίνηση ή κάποια μέρη της, συνεπώς πρέπει να είναι μέσα σε μια while loop.

Έτσι μέσα στην πρώτη while βρίσκεται η επιλογή του πρώτου κελιού της κίνησης του παίκτη. Εφόσον ο παίκτης πρέπει να δώσει τουλάχιστον ένα κελί για την κίνηση του, αυτό το κομμάτι είναι απαραίτητο. Αν ο χρήστης εισάγει κάποια μη αποδεκτή είσοδο (κάτι εκτός από ακέραιο που αντιπροσωπεύει αριθμό κελιού στο board ή κελί που δεν μπορεί να επιλεγεί επειδή είναι κατειλημμένο, δηλαδή δεν είναι στην λίστα availableMoves), τότε ξανατυπώνεται μήνυμα να εισάγει το πρώτο κελί μέχρι να δώσει έγκυρη είσοδο. Επίσης αν η επιλογή του είναι κάποιο κελί της διαγωνίου, τότε η σειρά του ολοκληρώνεται αυτόματα κρατώντας μόνο αυτό το κελί ως επιλογή.

Για το δεύτερο κελί, υπάρχουν και άλλες προϋποθέσεις που θα πρέπει να πληροί η αντίστοιχη είσοδος. Αρχικά πρέπει πάλι να είναι κάποιος ακέραιος από την λίστα availableMoves, ενώ πρέπει επίσης να είναι νόμιμη η κίνηση που αποτελείται από τον συνδυασμό του πρώτου και του δεύτερου κελιού. Αυτό ο έλεγχος πραγματοποιείται από την συνάρτηση checkValidMove που φαίνεται παρακάτω.

def checkValidMove(*nimBoard*, *move*):

*# get the length of the move*

*# print(move)*

length = len(*move*)

*# check if the move is valid*

if length == 3:

if *move*[length-1] != *move*[length-2]:

*# check if the move is in the same row*

if *move*[length-1] in getRowCells(*nimBoard*, *move*[length-2]):

if *move*[length-1] in getRowCells(*nimBoard*, *move*[0]):

return True

*# check if the move is in the same column*

elif *move*[length-1] in getColumnCells(*nimBoard*, *move*[length-2]):

if *move*[length-1] in getColumnCells(*nimBoard*, *move*[0]):

return True

else:

return False

else:

if *move*[length-1] in getRowCells(*nimBoard*, *move*[0]):

return True

*# check if the move is in the same column*

if *move*[length-1] in getColumnCells(*nimBoard*, *move*[0]):

return True

else:

return False

elif length == 2:

if *move*[length-1] not in getDiagonalCells(*nimBoard*):

if *move*[length-1] in getRowCells(*nimBoard*, *move*[0]):

return True

*# check if the move is in the same column*

elif *move*[length-1] in getColumnCells(*nimBoard*, *move*[0]):

return True

else:

return False

Όπως είδαμε προηγουμένως η συνάρτηση καλείται με όρισμα το αρχικό board, μιας και θέλουμε να δούμε αν η κίνηση είναι νόμιμη σε ό,τι αφορά την θέση των κελιών (έχουμε ήδη ελέγξει αν το κελί είναι κατειλημμένο). Το δεύτερο όρισμα με το οποίο καλείται είναι η λίστα choice, η οποία περιέχει τα κελιά που έχει εισάγει ήδη ο παίκτης μαζί με το επόμενο κελί που θέλει να παίξει. Μέσα στην checkValidMove έχουμε δυο περιπτώσεις ανάλογα με το αν ελέγχεται το δεύτερο ή τρίτο κελί της κίνησης του παίκτη.

Αν πρόκειται για το δεύτερο κελί, τότε τα πράγματα είναι απλά: κοιτάμε αν το προηγούμενο κελί (στην περίπτωση αυτή το πρώτο) είναι στην ίδια γραμμή ή στην ιδιά στήλη με το δεύτερο. Για τον έλεγχο αυτό χρησιμοποιούνται οι συναρτήσεις getRowCells, getColumnCells, οι οποίες επιστρέφουν λίστα με τα κελιά που βρίσκονται στην ίδια γραμμή ή στήλη με ένα συγκεκριμένο κελί αντίστοιχα. Αν τα 2 κελιά βρίσκονται όντως στην ίδια γραμμή ή στήλη τότε η συνάρτηση επιστρέφει true και το κελί προστίθεται στην λίστα με τα κελιά της κίνησης του παίκτη, ενώ αν όχι τυπώνεται μήνυμα που λέει ότι αυτό το κελί είναι λάθος και πρέπει να προσπαθήσει ξανά.

Προτού δούμε πώς ελέγχεται η είσοδος του τρίτου κελιού, σημειώνεται πως το πρόγραμμα έχει φτιαχτεί έτσι ώστε ο παίκτης να μπορεί να τερματίζει την σειρά του με παραπάνω από έναν τρόπους, διατηρώντας τις επιλογές κελιών που έχει κάνει ως εκείνο το σημείο. Ο ένας τρόπος είναι να δώσει ως είσοδο κάτι εκτός από ακέραιο αριθμό (πχ γράμμα, πραγματικό αριθμό κοκ), και ο άλλος είναι να δώσει δύο ή τρία ίδια κελιά. Πχ με την εισαγωγή 2,3,3 ο παίκτης τερματίζει την σειρά του έχοντας ως κίνηση τα κελιά 2 και 3. Το ίδιο θα συμβεί και αν δώσει ως είσοδο 2,2,3 , ή 2,3, z ή 3,2, z. Όπως φαίνεται τα κελιά δεν χρειάζεται να δοθούν με την σειρά που εμφανίζονται στο board, αρκεί τελικά να βρίσκονται στην ίδια γραμμή ή στήλη, ενώ η επιλογή του z ως μη έγκυρου χαρακτήρα για τις τελευταίες δύο περιπτώσεις ήταν τυχαία.

Για τον έλεγχο του τρίτου κελιού, αρχικά ελέγχεται αν αυτό είναι ίδιο με το δεύτερο (αν είναι τότε ο έλεγχος είναι ίδιας λογικής με τον έλεγχο που περιγράφηκε προηγουμένως). Αν δεν είναι ίδια, τότε πρέπει να διαπιστωθεί αν το τρίτο κελί βρίσκεται είτε στην ίδια γραμμή είτε στην ίδια στήλη και με τα δύο προηγούμενα κελιά. Ο έλεγχος γραμμής πραγματοποιείται με την getRowCells, ενώ ο έλεγχος στήλης με την getColumnCells. Η checkValidMove ελέγχει επίσης αν όλα τα κελιά της κίνησης είναι μη διαγώνια, και επιστρέφει true αν ναι ή false αν όχι αντίστοιχα.

Τώρα ας επιστρέψουμε στην getPlayerMove. Αν το δεύτερο κελί περάσει τον έλεγχο της checkValidMove, τότε εισάγεται στην λίστα με τα κελιά της κίνησης του παίκτη. Ακολουθεί το τρίτο κελί. Γίνονται πάλι οι ίδιοι έλεγχοι (αν είναι έγκυρος ακέραιος, αν το κελί είναι διαθέσιμο και αν επιστρέφει true η checkValidMove), και αν πραγματοποιηθούν επιτυχώς μπαίνει και το τρίτο κελί στην λίστα. Ωστόσο δεν έχει ελεγχθεί ακόμα αν τα δοσμένα κελιά είναι και διαδοχικά μεταξύ τους. Αυτό γίνεται με την checkValidMove2 που δίνεται παρακάτω.

def checkValidMove2(*move*):

length = len(*move*)

*move*.sort(*reverse*=False) *# arranges the move list into ascending order*

*# check if the move is valid*

*# check if the move is consecutive*

if length == 3:

*# if some of the tiles have been chosen more than once:*

if *move*[0] == *move*[1] == *move*[2]:

return True

elif (*move*[0] == *move*[1]) and (*move*[0] == *move*[2] - 1 or *move*[0] == *move*[2] - N):

return True

elif (*move*[1] == *move*[2]) and (*move*[0] == *move*[1] - 1 or *move*[0] == *move*[1] - N):

return True

*# if all the tiles have been chosen exactly once:*

*# check if the tiles are consecutive in the same row*

elif (*move*[0] == *move*[1] - 1) and (*move*[0] == *move*[2] - 2):

return True

*# check if the tiles are consecutive in the same column*

elif (*move*[0] == *move*[1] - N) and (*move*[0] == *move*[2] - 2\*N):

return True

else:

return False

elif length == 2:

*# if the player chose the same tile twice*

if *move*[0] == *move*[1]:

return True

*# if the two chosen tiles are different*

else:

*# check if the tiles are consecutive in the same row*

if *move*[0] == *move*[1] - 1:

return True

*# check if the tiles are consecutive in the same column*

elif *move*[0] == *move*[1] - N:

return True

else:

return False

Όπως προαναφέρθηκε, η checkValidMove2 δρα συμπληρωματικά με την checkValidMove για τον έλεγχο της ορθότητας των κινήσεων που πραγματοποιούνται από τον παίκτη. Δέχεται ως όρισμα μια λίστα κινήσεων μήκους 2 ή 3 και επιστρέφει είτε true είτε false, ανάλογα με το αν η συνθήκη που ελέγχει πληρείται ή όχι. Συγκεκριμένα, αφού η checkValidMove ελέγξει πως τα κελιά που επιλέχθηκαν βρίσκονται όλα είτε στην ίδια γραμμή είτε στην ίδια στήλη και όχι πάνω στην διαγώνιο, η checkValidMove2 ελέγχει αν τα κελιά αυτά είναι και διαδοχικά, εξετάζοντας τις θέσεις τους στο ταμπλό. Επειδή τα κελιά δεν είναι ανάγκη να δοθούν από τον παίκτη με την σειρά που εμφανίζονται στο ταμπλό, αρκεί τελικά να είναι διαδοχικά, εκτελείται αφού ο παίκτης ολοκληρώσει την κίνησή του, σε αντίθεση με την checkValidMove που εκτελείται μετά από κάθε επιλογή κελιού. Σε περίπτωση που η checkValidMove2 επιστρέψει false, αν δηλαδή δοθεί έστω και ένα μη διαδοχικό κελί, τότε η getPlayerMove εκτελείται από την αρχή και ο παίκτης καλείται να επιλέξει εκ νέου ποια κελιά θέλει να παίξει, χωρίς να αποθηκεύεται καμία από τις προηγούμενες επιλογές του. Ο κώδικας που αφορά την επανεκτέλεση της getPlayerMove βρίσκεται στην main, εντός των τμημάτων κώδικα που σχετίζονται με την κίνηση του παίκτη, και φαίνεται και στο τμήμα κώδικα που ακολουθεί (while (move == False): … ).

Αφού τελειώσαμε με την getPlayerMove (δηλαδή όλους τους ελέγχους για την εγκυρότητα των κελιών που επιλέγει ο παίκτης για την κίνηση του), προχωράμε παρακάτω στο main πρόγραμμά μας.

*# Player's turn.*

            move = getPlayerMove(nimBoard, diagonalCells, initialNimBoard)

            while (move == False):

                move = getPlayerMove(

                    nimBoard, diagonalCells, initialNimBoard)

            playermoves.extend(move)

            makeMove(nimBoard, playerLetter, move)

            drawNimPalette(nimBoard, N)

            if (isBoardFull(nimBoard, N) is True):

                drawNimPalette(nimBoard, N)

                print(

                    bcolors.HEADER + 'Congrats you have beaten the computer!' + bcolors.ENDC)

                break

            else:

                turn = 'computer'

        else:

*# Computer's turn.*

            move = getComputerMove(

                nimBoard, computerStrategy, initialNimBoard, playermoves)

            print(move)

            makeMove(nimBoard, computerLetter, move)

            drawNimPalette(nimBoard, N)

            if (isBoardFull(nimBoard, N) is True):

                drawNimPalette(nimBoard, N)

                print(

                    bcolors.HEADER + 'The computer has beaten you! You lose.' + bcolors.ENDC)

                break

            else:

                turn = 'player'

Όταν ο παίκτης τελειώσει με την επιλογή της κίνησής του και εφόσον αυτή είναι νόμιμη, η κίνηση εισάγεται στον πίνακα playermoves που περιγράφηκε στην αρχή, ο οποίος αποθηκεύει την τελευταία κίνηση του παίκτη (για την περίπτωση στρατηγικής copycat). Στην συνέχεια καλείται η συνάρτηση makeMove, η οποία είναι υπεύθυνη για την εισαγωγή της κίνησης στο ταμπλό, όπως φαίνεται στον κώδικά της παρακάτω.

def makeMove(*nimBoard*, *letter*, *move*):

*# This function places the letter on the board at the move*

*# (i.e. it changes the board).*

    if isinstance(*move*, list):

        for i in range(len(*move*)):

*nimBoard*[*move*[i]] = *letter*

*nimBoard*[0] = *nimBoard*[0] + 1

Σημειώνεται ότι εντός αυτής της συνάρτησης γίνεται και η αύξηση της μεταβλητής counter, η οποία δείχνει πόσα tiles του ταμπλό είναι κατειλημμένα σε κάθε γύρο.

Αφού λοιπόν εισάγουμε την κίνηση στο ταμπλό συνεχίζουμε με την εκτύπωση του ταμπλό στην οθόνη με την drawNimPalette. Τέλος, υπάρχει συνθήκη η οποία αν διαπιστώσει ότι μετα την ολοκλήρωση της κίνησης το ταμπλό είναι γεμάτο, τυπώνει μήνυμα νίκης για τον παίκτη.

Στην περίπτωση που είναι η σειρά του υπολογιστή να παίξει, εκτελείται το τμήμα κώδικα που φαίνεται παρακάτω.

        else:

*# Computer's turn.*

            move = getComputerMove(

                nimBoard, computerStrategy, initialNimBoard, playermoves)

            print(move)

            makeMove(nimBoard, computerLetter, move)

            drawNimPalette(nimBoard, N)

            if (isBoardFull(nimBoard, N) is True):

                drawNimPalette(nimBoard, N)

                print(

                    bcolors.HEADER + 'The computer has beaten you! You lose.' + bcolors.ENDC)

                break

            else:

                turn = 'player'

Όπως φαίνεται το κομμάτι κώδικα για τον υπολογιστή είναι παρόμοιο με το αντίστοιχο του παίκτη. Οι μόνες διαφορές είναι το μήνυμα νίκης που τώρα είναι μήνυμα ήττας για τον παίκτη αν το ταμπλό είναι γεμάτο, η else στο τέλος που χρησιμοποιείται για την αναδρομική εκτέλεση του κώδικα και την συνεχή εναλλαγή σειράς, και κυρίως η κλήση της getComputerMove. Αυτή είναι και η συνάρτηση που θα αναλυθεί στην συνέχεια.

def getComputerMove(*nimBoard*, *computerStrategy*, *initialNimBoard*, *playermoves*):

*# This function returns the computer's move.*

*# The computer will choose a random move from the list of*

*# available moves.*

    if *computerStrategy* == 'random':

        return getComputerMove\_random(*nimBoard*, *initialNimBoard*)

    else:

        if *computerStrategy* == 'first free':

            return getComputerMove\_firstfit(*nimBoard*, *initialNimBoard*)

        else:

            if *computerStrategy* == 'copycat':

                return getComputerMove\_copycat(*nimBoard*, *initialNimBoard*, *playermoves*)

            else:

                print(

                    bcolors.ERROR + 'ERROR 2: Incomprehensible strategy was provided. Try again...' + bcolors.ENDC)

Στον παραπάνω κώδικα της getComputerMove υπάρχουν 3 υποπεριπτώσεις, μία για κάθε πιθανή στρατηγική του υπολογιστή. Ανάλογα ποια στρατηγική επιλέχθηκε στην αρχή του παιχνιδιού, γίνεται κλήση της αντίστοιχης συνάρτησης.

Με την κλήση της getComputerMove εκτελείται παρόμοιος κώδικας για όλες τις στρατηγικές:

def getComputerMove\_random(*nimBoard*, *initialNimBoard*):

*# This function returns a random move from the list of*

*# available moves.*

*# return a random available tile to start the move*

    possibleMoves = getAvailableCells(*nimBoard*)

    choice = []

*# if there's more than five available cells left (case of 0 or 1 cell left are considered trivial and also included)*

    if len(possibleMoves) > 5 or len(possibleMoves) < 2:

        choice = randomMoreThanFiveLeft(*nimBoard*, *initialNimBoard*)

*# if there's five or less available cells left (not including 0 or 1 cell left)*

    elif len(possibleMoves) == 2:

        choice = twoCellsLeft(

            'random', *nimBoard*, *initialNimBoard*, possibleMoves, [])

    elif len(possibleMoves) == 3:

        choice = threeCellsLeft(

            'random', *nimBoard*, *initialNimBoard*, possibleMoves, [])

    elif len(possibleMoves) == 4:

        choice = fourCellsLeft(

            'random', *nimBoard*, *initialNimBoard*, possibleMoves, [])

    elif len(possibleMoves) == 5:

        choice = fiveCellsLeft(

            'random', *nimBoard*, *initialNimBoard*, possibleMoves, [])

    return choice

def getComputerMove\_firstfit(*nimBoard*, *initialNimBoard*):

*# This function returns the first free move from the list of*

*# available moves.*

*# return the first available tile to start the move*

    possibleMoves = getAvailableCells(*nimBoard*)

    choice = []

*# if there's more than five available cells left (case of 0 or 1 cell left are considered trivial and also included)*

    if len(possibleMoves) > 5 or len(possibleMoves) < 2:

        choice = firstfreeMoreThanFiveLeft(*nimBoard*, *initialNimBoard*)

*# if there's five or less available cells left (not including 0 or 1 cell left)*

    elif len(possibleMoves) == 2:

        choice = twoCellsLeft('first free', *nimBoard*,

*initialNimBoard*, possibleMoves, [])

    elif len(possibleMoves) == 3:

        choice = threeCellsLeft(

            'first free', *nimBoard*, *initialNimBoard*, possibleMoves, [])

    elif len(possibleMoves) == 4:

        choice = fourCellsLeft(

            'first free', *nimBoard*, *initialNimBoard*, possibleMoves, [])

    elif len(possibleMoves) == 5:

        choice = fiveCellsLeft(

            'first free', *nimBoard*, *initialNimBoard*, possibleMoves, [])

    return choice

def getComputerMove\_copycat(*nimBoard*, *initialNimBoard*, *playermoves*):

*# This function returns the copycat move from the list of*

*# available moves.*

*# return the first available tile to start the move*

    possibleMoves = getAvailableCells(*nimBoard*)

    choice = []

*# if there's more than five available cells left (case of 0 or 1 cell left are considered trivial and also included)*

    if len(possibleMoves) > 5 or len(possibleMoves) < 2:

        choice = copycatMoreThanFiveLeft(

*nimBoard*, *initialNimBoard*, *playermoves*)

*# if there's five or less available cells left (not including 0 or 1 cell left)*

    elif len(possibleMoves) == 2:

        choice = twoCellsLeft(

            'copycat', *nimBoard*, *initialNimBoard*, possibleMoves, *playermoves*)

    elif len(possibleMoves) == 3:

        choice = threeCellsLeft(

            'copycat', *nimBoard*, *initialNimBoard*, possibleMoves, *playermoves*)

    elif len(possibleMoves) == 4:

        choice = fourCellsLeft(

            'copycat', *nimBoard*, *initialNimBoard*, possibleMoves, *playermoves*)

    elif len(possibleMoves) == 5:

        choice = fiveCellsLeft(

            'copycat', *nimBoard*, *initialNimBoard*, possibleMoves, *playermoves*)

    return choice

Η μόνη διαφορά μεταξύ των συναρτήσεων αυτών είναι ότι ανάλογα την περίπτωση καλούν μια άλλη στρατηγική, και ότι στην copycat υπάρχει και η λίστα με την προηγουμένη κίνηση του παίκτη.

Στην αρχή δηλώνεται η λίστα choice, η οποία λαμβάνει και στην συνέχεια επιστρέφει την κίνηση του υπολογιστή. Το κυριότερο κομμάτι όμως είναι οι έλεγχοι που πραγματοποιούνται για το πόσα κενά κελιά απομένουν στο ταμπλό. Αν αυτά τα κελιά είναι περισσότερα από 5, ο υπολογιστής παίζει κανονικά την αρχική επιλεγμένη στρατηγική του καλώντας την αντίστοιχη συνάρτηση. Αντιθέτως αν τα ελεύθερα κελιά είναι λιγότερα από 5, τότε ο υπολογιστής ελέγχει αν μπορεί να κάνει κάποια κίνηση που θα του εξασφαλίσει νίκη και αν ναι κάνει αυτή την κίνηση, αλλιώς συνεχίζει να παίζει με την αρχική του στρατηγική. Αυτό γίνεται με κλήση των κατάλληλων συναρτήσεων twoCellsLeft, threeCellsLeft, fourCellsLeft και fiveCellsLeft (για 2, 3, 4 και 5 εναπομείναντα κενά κελιά αντίστοιχα), οι οποίες και αναλύονται στην συνέχεια.

Σημειώνεται πως έχουμε επιλέξει στην περίπτωση που μένει μόνο 1 κενό κελί ο υπολογιστής να συνεχίζει να παίζει με την αρχική του στρατηγική, καθώς πρόκειται για την τετριμμένη παρτίδα νίκης και άρα όπως και να παίξει είναι σίγουρο ότι θα κερδίσει.

def twoCellsLeft(*mode*, *nimBoard*, *initialNimBoard*, *possibleMoves*, *playermoves*):

    global result\_type

    result\_type = 'undefined'

    choice2 = []

*# if the two cells are consecutive and none of them is on the diagonal then the computer chooses these cells and wins*

    if *possibleMoves*[1] in getCellNeighbours(*possibleMoves*[0], *initialNimBoard*) and *possibleMoves*[0] not in getDiagonalCells(*initialNimBoard*) and *possibleMoves*[1] not in getDiagonalCells(*initialNimBoard*):

        choice2.extend([*possibleMoves*[0], *possibleMoves*[1]])

        result\_type = 'specific'

*# else if the cells can't both be chosen (because one of them is on the diagonal or they are not consecutive)*

*# then the computer continues playing according to its previous strategy*

    else:

        if *mode* == 'random':

            t\_choice = randomMoreThanFiveLeft(*nimBoard*, *initialNimBoard*)

            result\_type = 'random'

        elif *mode* == 'first free':

            t\_choice = firstfreeMoreThanFiveLeft(*nimBoard*, *initialNimBoard*)

            result\_type = 'first free'

        elif *mode* == 'copycat':

            t\_choice = copycatMoreThanFiveLeft(

*nimBoard*, *initialNimBoard*, *playermoves*)

            result\_type = 'copycat'

        choice2.extend(t\_choice)

    return choice2

Η twoCellsLeft δέχεται ως ορίσματα την επιλεγμένη στρατηγική για τον υπολογιστή (random, first fit ή copycat), την τρέχουσα και την αρχική μορφή του ταμπλό, μία λίστα πιθανών κινήσεων μήκους 2 και μία λίστα με την τελευταία κίνηση του παίκτη (αφορά μόνο την περίπτωση όπου η επιλεγμένη στρατηγική είναι η copycat, για τις υπόλοιπες περιπτώσεις δίνεται ως όρισμα απλά μία κενή λίστα). Η συνάρτηση αυτή καλείται όταν έχουν απομείνει ακριβώς 2 κενά κελιά στο ταμπλό και είναι η σειρά του υπολογιστή να παίξει. Επιστρέφει την κίνηση που θα πρέπει να κάνει ο υπολογιστής προκειμένου να κερδίσει, εάν αυτό είναι δυνατόν, και αν όχι τότε καλεί την αντίστοιχη συνάρτηση έτσι ώστε ο υπολογιστής να συνεχίσει να παίζει σύμφωνα με την αρχική του στρατηγική.

Γνωρίζουμε από την θεωρητική ανάλυση πως για να κερδίσει παρτίδα με δύο κενά κελιά αυτός του οποίου είναι η σειρά να παίξει, θα πρέπει να μπορεί να γεμίσει και τα δύο κελιά, δηλαδή πρέπει τα δύο κενά κελιά να είναι διαδοχικά και μη διαγώνια. Η συνάρτηση ελέγχει αν ισχύουν οι δύο προϋποθέσεις και αν ναι τότε επιστρέφει ως κίνηση τα 2 κελιά, αλλιώς ο υπολογιστής συνεχίζει να παίζει με βάση την αρχική του στρατηγική. Επιπλέον, ανάλογα πώς θα παίξει τελικά ο υπολογιστής, η τιμή της global μεταβλητής result\_type που αρχικά είναι undefined ενημερώνεται κατάλληλα. Η τιμή αυτή θα χρησιμοποιηθεί στην συνέχεια στις αντίστοιχες συναρτήσεις για 4 και 5 κενά κελιά, ώστε να μπορέσει να πραγματοποιηθεί αναδρομικά ο έλεγχος για το αν η δοσμένη παρτίδα είναι παρτίδα νίκης ή όχι.

def threeCellsLeft(*mode*, *nimBoard*, *initialNimBoard*, *possibleMoves*, *playermoves*):

    global result\_type

    result\_type = 'undefined'

    choice3 = []

*# if the three cells are not consecutive and one of them has the other 2 as adjacent (with none them on the diagonal) then the computer picks the one in the middle*

    if getCellwithNumNeighbours(2, *initialNimBoard*, *possibleMoves*) == False:

*# if that’s not the case the computer plays the selected strategy*

        if getCellwithNumNeighbours(1, *initialNimBoard*, *possibleMoves*) == False:

            if *mode* == 'random':

                t\_choice = randomMoreThanFiveLeft(*nimBoard*, *initialNimBoard*)

            elif *mode* == 'first free':

                t\_choice = firstfreeMoreThanFiveLeft(*nimBoard*, *initialNimBoard*)

            elif *mode* == 'copycat':

                t\_choice = copycatMoreThanFiveLeft(

*nimBoard*, *initialNimBoard*, *playermoves*)

            choice3.extend(t\_choice)

*# if there is a cell with 1 neighbour (neither of them in the diagonal), then that's the computer's next move*

        else:

            t\_choice = getCellwithNumNeighbours(

                1, *initialNimBoard*, *possibleMoves*)

            choice3.append(t\_choice)

*# if there is a cell with 2 neighbours*

    else:

*# if the three cells are consecutive and none of them is on the diagonal then the computer chooses these cells and wins*

        t\_choice = getCellwithNumNeighbours(2, *initialNimBoard*, *possibleMoves*)

*# check if the three cells are consecutive and the move is legal and not on the diagonal if yes then the computer chooses the three cells and wins*

        temp = []

        temp.extend(*possibleMoves*)

        if (abs(*possibleMoves*[0] - *possibleMoves*[1]) == 1 or abs(*possibleMoves*[0] - *possibleMoves*[1]) == N) and (abs(*possibleMoves*[1] - *possibleMoves*[2]) == 1 or abs(*possibleMoves*[1] - *possibleMoves*[2]) == N) and *possibleMoves*[0] not in getDiagonalCells(*initialNimBoard*) and *possibleMoves*[1] not in getDiagonalCells(*initialNimBoard*) and *possibleMoves*[2] not in getDiagonalCells(*initialNimBoard*) and checkValidMove(*initialNimBoard*, temp):

            choice3.extend(temp)

            result\_type = 'consecutive'

        else:

            choice3.append(t\_choice)

    return choice3

Η threeCellsLeft δέχεται τα ίδια ορίσματα με την twoCellsLeft (με την διαφορά ότι η λίστα πιθανών κινήσεων έχει μήκος 3) και καλείται στην περίπτωση που έχουν απομείνει ακριβώς 3 κενά κελιά στο ταμπλό και είναι η σειρά του υπολογιστή να παίξει. Επιστρέφει κίνηση που θα διασφαλίσει την νίκη του υπολογιστή αν αυτό είναι απαραίτητο, ενώ αν δεν είναι, δηλαδή αν έχουμε ουσιαστικά την στοίβα (1,1,1) όπου είναι σίγουρο πως όπως και να παίξει θα κερδίσει, συνεχίζει να παίζει με την στρατηγική που είχε εξαρχής. Υπενθυμίζεται πως από την θεωρητική ανάλυση η περίπτωση τριών κενών κελιών είναι σίγουρα παρτίδα νίκης για αυτόν του οποίου είναι η σειρά να παίξει (εννοείται αν παίζει βέλτιστα), συνεπώς ο υπολογιστής πάντα θα κερδίζει αν βρεθεί σε αυτήν την κατάσταση.

Αρχικά γίνεται έλεγχος αν υπάρχει κενό κελί με 2 γειτονικά κελιά τα οποία είναι επίσης κενά, χρησιμοποιώντας την συνάρτηση getCellwithNumNeighbours με κατάλληλα ορίσματα (num = 2). Αν επιστραφεί κελί, τότε σημαίνει πως τα 3 κελιά είτε είναι διαδοχικά είτε σχηματίζουν γωνία μεταξύ τους. Ελέγχεται τι από τα 2 ισχύει και ο υπολογιστής παίζει ανάλογα: αν τα κελιά είναι διαδοχικά και δεν βρίσκονται στην κύρια διαγώνιο τότε γεμίζει και τα 3 κελιά, ενώ αν σχηματίζεται γωνία παίζει το κελί που επιστρέφει η συνάρτηση getCellwithNumNeighbours (το γωνιακό), αφήνοντας στον παίκτη 2 κενά και μη διαδοχικά κελιά, δηλαδή ουσιαστικά την παρτίδα ήττας (1,1).

Αν η getCellwithNumNeighbours επιστρέψει false για num = 2, τότε τρέχει και πάλι αλλά αυτή την φορά με num = 1, ώστε να διαπιστωθεί αν υπάρχουν δύο διαδοχικά κενά κελιά. Αν ναι, τότε ο υπολογιστής παίζει το κελί που επιστρέφεται από την συνάρτηση, το οποίο θα είναι ένα από τα δύο διαδοχικά. Αν η συνάρτηση επιστρέψει πάλι false τότε σημαίνει πως δεν υπάρχουν καθόλου διαδοχικά κελιά (περίπτωση (1,1,1)) και άρα όπως και να παίξει ο υπολογιστής θα κερδίσει. Ειδικά για αυτήν την περίπτωση έχουμε επιλέξει ο υπολογιστής να συνεχίζει να παίζει με την αρχική του στρατηγική, καλώντας την αντίστοιχη συνάρτηση. Σε κάθε περίπτωση θα κληροδοτηθεί στον παίκτη η παρτίδα ήττας (1,1). Τέλος, η τιμή της global μεταβλητής result\_type που αρχικά είναι undefined ενημερώνεται κατάλληλα αν ο υπολογιστής γεμίσει 3 κελιά. Η τιμή αυτή θα χρησιμοποιηθεί στην συνέχεια στην αντίστοιχη συνάρτηση για 5 κενά κελιά, ώστε να μπορέσει να πραγματοποιηθεί αναδρομικά ο έλεγχος για κίνηση που εξασφαλίζει νίκη στον υπολογιστή.

def fourCellsLeft(*mode*, *nimBoard*, *initialNimBoard*, *possibleMoves*, *playermoves*):

    choice4 = []

    global result\_type4

    result\_type4 = 'undefined'

*# picks every combination of 2 available cells and, if they are consecutive and non-diagonal,*

*# checks if they give a winning move*

    for i in range(0, 4):

        t\_posmoves = []

        t\_posmoves.append(*possibleMoves*[i])

        for j in range(0, 4):

            checklist = []

            if i != j:

                t\_posmoves.append(*possibleMoves*[j])

                twoCellsLeft(*mode*, *nimBoard*, *initialNimBoard*,

                             t\_posmoves, *playermoves*)

*# checks if the 2 cells are consecutive and not on the diagonal*

                if result\_type == 'specific':

                    for k in range(0, 4):

                        if *possibleMoves*[k] not in t\_posmoves:

                            checklist.append(*possibleMoves*[k])

                    twoCellsLeft(*mode*, *nimBoard*,

*initialNimBoard*, checklist, *playermoves*)

*# checks if the 2 cells that were previously chosen give a winning move*

                    if result\_type == 'random' or result\_type == 'first free' or result\_type == 'copycat':

                        choice4.extend(t\_posmoves)

                        return choice4

                del t\_posmoves[1]

*# if no winning move found, then computer plays according to its previous strategy*

    if *mode* == 'random':

        t\_choice = randomMoreThanFiveLeft(*nimBoard*, *initialNimBoard*)

        result\_type4 = 'random'

    elif *mode* == 'first free':

        t\_choice = firstfreeMoreThanFiveLeft(*nimBoard*, *initialNimBoard*)

        result\_type4 = 'first free'

    elif *mode* == 'copycat':

        t\_choice = copycatMoreThanFiveLeft(

*nimBoard*, *initialNimBoard*, *playermoves*)

        result\_type4 = 'copycat'

    choice4.extend(t\_choice)

    return choice4

Η fourCellsLeft δέχεται τα ίδια ορίσματα με την threeCellsLeft (με την διαφορά ότι η λίστα πιθανών κινήσεων έχει μήκος 4) και καλείται στην περίπτωση που έχουν απομείνει ακριβώς 4 κενά κελιά στο ταμπλό και είναι η σειρά του υπολογιστή να παίξει. Επιστρέφει κίνηση που θα διασφαλίσει την νίκη του υπολογιστή, εάν υπάρχει, αλλιώς επιστρέφει κίνηση που κάνει ο υπολογιστής σύμφωνα με την αρχική του στρατηγική. Για να αποφασίσουμε τι έλεγχοι χρειάζεται να γίνουν ώστε να διαπιστωθεί αν πρόκειται για παρτίδα νίκης, αξιοποιούμε την θεωρητική ανάλυση. Αν ο υπολογιστής με την κίνησή του γεμίσει ένα κελί, τότε στον παίκτη κληροδοτείται παρτίδα με 3 κενά κελιά, η οποία όπως αναλύσαμε στο θεωρητικό μέρος είναι σίγουρη παρτίδα νίκης για αυτόν που παίζει (θεωρώντας πάντα ότι ο παίκτης έχει στόχο να κερδίσει και άρα παίζει βέλτιστα). Επίσης, αν ο υπολογιστής παίξει γεμίζοντας 3 κελιά τότε θα μείνει ένα κενό κελί, συνεπώς ο παίκτης θα κληθεί να παίξει στην τετριμμένη παρτίδα νίκης. Άρα λοιπόν δεν έχει νόημα να ελεγχθούν περιπτώσεις όπου ο υπολογιστής θα γεμίσει ένα ή τρία κελιά, αφού κάτι τέτοιο είναι σίγουρο πως θα οδηγήσει σε ήττα του. Για αυτόν τον λόγο στον κώδικα ελέγχεται μόνο αν ο υπολογιστής μπορεί να εξασφαλίσει την νίκη του γεμίζοντας 2 κελιά με την κίνησή του.

Ο κώδικας λειτουργεί ως εξής, με χρήση διαδοχικών βρόχων for: αρχικά επιλέγεται ένα δυνατό ζεύγος διακριτών μεταξύ τους κελιών (για αυτό είναι απαραίτητη η συνθήκη i != j, ώστε να μην επιλεχθεί κάποιο κελί με τον εαυτό του) από την λίστα πιθανών κινήσεων και αποθηκεύεται στην βοηθητική λίστα t\_posmoves. Στην συνέχεια ελέγχεται αν τα συγκεκριμένα δύο κελιά είναι διαδοχικά και μη διαγώνια, καλώντας την συνάρτηση twoCellsLeft με όρισμα τα δύο κελιά. Αν η twoCellsLeft επιστρέψει τιμή της μεταβλητής result\_type ίση με specific, τότε σημαίνει πως πράγματι τα κελιά είναι διαδοχικά και μη διαγώνια (το specific προκύπτει επειδή αν είχαμε μόνο αυτά τα 2 κελιά, δηλαδή αν έτρεχε κατευθείαν η twoCellsLeft με όρισμα αυτά τα 2 κελιά χωρίς να κληθεί αναδρομικά από άλλη συνάρτηση, τότε ο υπολογιστής θα επέλεγε να τα γεμίσει προκειμένου να κερδίσει, κάνοντας έτσι μία “συγκεκριμένη” κίνηση).

Αν επιστραφεί result\_type διάφορο του specific, τότε το ζεύγος δεν είναι διαδοχικό ή/και μη διαγώνιο, και το πρόγραμμα συνεχίζει επιλέγοντας νέο πιθανό ζεύγος κελιών. Αντίθετα, αν το result\_type είναι specific τότε τα 2 κελιά παραμένουν υποψήφια για την κίνηση του υπολογιστή, και ο έλεγχός τους συνεχίζεται. Μέσω ενός βρόχου for, βρίσκονται τα 2 κελιά που βρίσκονται στις πιθανές κινήσεις αλλά δεν έχουν εισαχθεί στην λίστα t\_posmoves, και εισάγονται σε μία μέχρι τότε κενή βοηθητική λίστα με όνομα checklist. Για να είναι τα κελιά της t\_posmoves κίνηση νίκης για τον υπολογιστή αν τα γεμίσει, θα πρέπει τα 2 κελιά που θα απομείνουν (δηλαδή τα κελιά της checklist) να είναι παρτίδα ήττας για τον παίκτη. Θα πρέπει δηλαδή τα κελιά της checklist να είναι είτε μη διαδοχικά είτε κάποιο από τα δύο να είναι διαγώνιο.

Για αυτόν τον έλεγχο χρησιμοποιείται και πάλι η twoCellsLeft, αυτήν την φορά με όρισμα την checklist. Αν επιστραφεί τιμή της result\_type ίση με random, first free ή copycat, τότε σημαίνει πως πράγματι τα 2 κελιά της checklist αποτελούν παρτίδα ήττας (επιστροφή αυτών των τιμών για την result\_type από την twoCellsLeft σημαίνει πως δεν βρέθηκε κίνησης νίκης για αυτά τα δύο κελιά, και άρα αν η twoCellsLeft είχε τρέξει κατευθείαν με όρισμα τα 2 κελιά χωρίς να κληθεί αναδρομικά από άλλη συνάρτηση τότε ο υπολογιστής θα συνέχιζε να παίζει με την αρχική του στρατηγική). Αν λοιπόν βρεθεί πως τα κελιά της checklist αποτελούν παρτίδα ήττας, τότε σημαίνει πως τα κελιά της t\_posmoves είναι κίνηση νίκης για τον υπολογιστή, και αυτή είναι η κίνησή του. Αντιθέτως αν τα κελιά της checklist δεν αποτελούν παρτίδα ήττας, δηλαδή αν η twoCellsLeft επιστρέψει result\_type ίσο με specific, τότε το πρόγραμμα συνεχίζει επιλέγοντας νέο πιθανό ζεύγος κελιών για την t\_posmoves. Ο έλεγχος επαναλαμβάνεται μέχρι είτε να βρεθεί και να επιλεχθεί κίνηση νίκης (αν υπάρχει), είτε να εξεταστούν όλα τα πιθανά ζευγάρια κελιών της λίστας πιθανών κινήσεων, οπότε αφού δεν έχει βρεθεί κίνηση νίκης το πρόγραμμα συνεχίζει αφήνοντας τον υπολογιστή να παίξει με την αρχική του στρατηγική.

Τέλος, ανάλογα την κίνηση που θα κάνει τελικά ο υπολογιστής, η τιμή της global μεταβλητής result\_type4 η οποία ως τώρα ήταν undefined ενημερώνεται κατάλληλα. Αυτή η μεταβλητή θα χρησιμοποιηθεί στην συνέχεια από την συνάρτηση fiveCellsLeft, προκειμένου να ελεγχθεί αναδρομικά αν η εξεταζόμενη κίνηση είναι κίνηση νίκης ή όχι.

def fiveCellsLeft(*mode*, *nimBoard*, *initialNimBoard*, *possibleMoves*, *playermoves*):

    choice5 = []

*# picks every combination of 3 available cells and, if they are consecutive and non-diagonal,*

*# checks if they give a winning move*

    for i in range(0, 5):

        t\_posmoves = []

        t\_posmoves.append(*possibleMoves*[i])

        for j in range(0, 5):

            if i != j:

                t\_posmoves.append(*possibleMoves*[j])

                for m in range(0, 5):

                    checklist = []

                    if m != j and m != i:

                        t\_posmoves.append(*possibleMoves*[m])

                        threeCellsLeft(

*mode*, *nimBoard*, *initialNimBoard*, t\_posmoves, *playermoves*)

*# checks if the 3 cells are consecutive and not on the diagonal*

                        if result\_type == 'consecutive':

                            for k in range(0, 5):

                                if *possibleMoves*[k] not in t\_posmoves:

                                    checklist.append(*possibleMoves*[k])

                            twoCellsLeft(*mode*, *nimBoard*,

*initialNimBoard*, checklist, *playermoves*)

*# checks if the 3 cells that were previously chosen give a winning move*

                            if result\_type == 'random' or result\_type == 'first free' or result\_type == 'copycat':

                                choice5.extend(t\_posmoves)

                                return choice5

                        del t\_posmoves[2]

                del t\_posmoves[1]

*# picks every cell individually, and examines if playing it will give a winning move. If yes, the selected cell is played*

    for i in range(0, 5):

        t\_posmoves = []

        t\_posmoves.append(*possibleMoves*[i])

        checklist = []

        for j in range(0, 5):

            if *possibleMoves*[j] not in t\_posmoves:

                checklist.append(*possibleMoves*[j])

        fourCellsLeft(*mode*, *nimBoard*, *initialNimBoard*, checklist, *playermoves*)

        if result\_type4 == 'random' or result\_type4 == 'first free' or result\_type4 == 'copycat':

            choice5.append(t\_posmoves[0])

            return choice5

Η fiveCellsLeft δέχεται τα ίδια ορίσματα με την threeCellsLeft (με την διαφορά ότι η λίστα πιθανών κινήσεων έχει μήκος 5) και καλείται στην περίπτωση που έχουν απομείνει ακριβώς 5 κενά κελιά στο ταμπλό και είναι η σειρά του υπολογιστή να παίξει. Γνωρίζουμε από την θεωρία πως περίπτωση 5 κενών κελιών είναι παρτίδα νίκης για αυτόν που παίζει σε αυτήν (εννοείται βέλτιστα), συνεπώς η συγκεκριμένη συνάρτηση επιστρέφει πάντα κίνηση νίκης και για αυτό δεν χρειάζεται να κληθεί κάποια από τις συναρτήσεις στρατηγικής του υπολογιστή (random/firstfree/copycatMoreThanFiveLeft). Προκειμένου να αποφασίσουμε τι έλεγχοι χρειάζεται να γίνουν ώστε να διαπιστωθεί αν πρόκειται για παρτίδα νίκης, ανατρέχουμε και πάλι στην θεωρητική ανάλυση. Αν ο υπολογιστής με την κίνησή του γεμίσει δύο κελιά, τότε αφήνει στον παίκτη παρτίδα με 3 κενά κελιά, η οποία όπως αναφέραμε και προηγουμένως αποτελεί παρτίδα νίκης. Άρα ο κώδικας θα πρέπει να ελέγχει αν ο υπολογιστής μπορεί να κάνει κίνηση νίκης γεμίζοντας είτε τρία κελιά είτε ένα.

Η λογική που ακολουθείται για τον έλεγχο των τριών και του ενός κελιού είναι ίδια με την λογική που ακολουθήθηκε στην fourCellsLeft για τον έλεγχο των δύο κελιών. Στο πρώτο μέρος της συνάρτησης, εξετάζεται αν ο υπολογιστής μπορεί να γεμίσει 3 κελιά και αν αυτό θα του δώσει νίκη έναντι του παίκτη. Μέσω διαδοχικών βρόχων for, επιλέγεται μία πιθανή 3άδα διακριτών μεταξύ τους κελιών από την λίστα πιθανών κινήσεων, και αποθηκεύεται στην προσωρινή βοηθητική λίστα t\_posmoves. Στην συνέχεια θα πρέπει να ελεγχθεί αν τα 3 επιλεγμένα κελιά είναι διαδοχικά και μη διαγώνια. Για την πραγματοποίηση αυτού του ελέγχου καλείται η threeCellsLeft με όρισμα την t\_posmoves, και αν η τιμή της μεταβλητής result\_type πάρει τιμή consecutive τότε τα κελιά είναι πράγματι διαδοχικά και μη διαγώνια.

Αν η result\_type δεν πάρει τιμή consecutive, τότε η 3άδα δεν είναι διαδοχική ή/και μη διαγώνια, και το πρόγραμμα συνεχίζει επιλέγοντας νέα πιθανή τριάδα. Αντιθέτως αν τα 3 κελιά είναι όντως διαδοχικά και μη διαγώνια, τότε παραμένουν υποψήφια κίνηση του υπολογιστή και θα πρέπει να ελεγχθούν τα 2 εναπομείναντα κελιά της λίστας πιθανών κινήσεων. Με χρήση βρόχου for, τα 2 αυτά κελιά εντοπίζονται και εισάγονται σε μία προσωρινή βοηθητική λίστα ονόματι checklist η οποία ήταν προηγουμένως κενή. Στην συνέχεια, ακριβώς όπως στην περίπτωση της fourCellsLeft, ελέγχεται αν αυτά τα 2 κελιά αποτελούν παρτίδα ήττας καλώντας την twoCellsLeft με όρισμα την checklist. Αν η μεταβλητή result\_type πάρει τιμή random, first free ή copycat τότε τα 2 κελιά αποτελούν πράγματι παρτίδα ήττας, και ο υπολογιστής επιλέγει ως κίνησή του την τρέχουσα τιμή της t\_posmoves. Αν η μεταβλητή result\_type δεν λάβει κάποια από τις προηγούμενες τιμές, τότε τα 2 κελιά αποτελούν παρτίδα νίκης και το πρόγραμμα συνεχίζει εξετάζοντας νέα πιθανή τριάδα. Το συγκεκριμένο τμήμα κώδικα επαναλαμβάνεται μέχρι είτε να βρεθεί και να παιχτεί νικητήρια τριάδα (αν υπάρχει), είτε να εξαντληθούν όλες οι δυνατές τριάδες της λίστας πιθανών κινήσεων.

Αν δεν βρεθεί νικητήρια τριάδα, τότε το πρόγραμμα προχωρά στο δεύτερο μέρος της συνάρτησης, όπου εντοπίζεται το κελί που πρέπει να παίξει ο υπολογιστής προκειμένου να κερδίσει (υπενθυμίζεται πως είναι σίγουρο ότι υπάρχει τουλάχιστον ένα τέτοιο κελί, αφού παρτίδα με 5 κενά κελιά είναι σίγουρη παρτίδα νίκης για αυτόν που παίζει σε αυτήν). Όμοια με πριν, επιλέγεται ένα κελί από την λίστα πιθανών κινήσεων, εισάγεται στην t\_posmoves, και τα 4 εναπομείναντα κελιά εντοπίζονται και εισάγονται στην checklist. Ύστερα ελέγχεται αν τα κελιά της checklist αποτελούν παρτίδα ήττας, εκτελώντας την fourCellsLeft με όρισμα την checklist. Αν η result\_type4 πάρει τιμή random, first free ή copycat, δηλαδή αν η επιλεγμένη τετράδα είναι πράγματι παρτίδα ήττας, τότε η τιμή της t\_posmoves επιλέγεται ως κίνηση του υπολογιστή. Αντίθετα αν η result\_type4 δεν πάρει κάποια από τις προηγούμενες τιμές, τότε η τετράδα δεν αποτελεί παρτίδα ήττας και το πρόγραμμα συνεχίζει επιλέγοντας νέο κελί από την λίστα πιθανών κινήσεων. Το τμήμα κώδικα τερματίζει μόλις βρεθεί η πρώτη νικητήρια κίνηση, την οποία και επιστρέφει ως κίνηση του υπολογιστή.

Παρακάτω αναλύονται μερικές ακόμα βοηθητικές συναρτήσεις που χρησιμοποιήθηκαν για την επίτευξη της λειτουργίας των προηγούμενων.

def getCellwithNumNeighbours(*num*, *initialNimBoard*, *possibleMoves*):

    for i in range(len(*possibleMoves*)):

        count = getNumberofNeighbours(

*possibleMoves*[i], *initialNimBoard*, *possibleMoves*)

        if count == *num*:

            return *possibleMoves*[i]

    return False

Η συνάρτηση getCellwithNumNeighbours δέχεται ως είσοδο έναν αριθμό num, την αρχική μορφή του ταμπλό και μία λίστα πιθανών κινήσεων με αυθαίρετο μέγεθος. Επιστρέφει το πρώτο κελί της λίστας πιθανών κινήσεων που βρίσκει το οποίο έχει ακριβώς num στο πλήθος γειτονικά κελιά εντός της λίστας πιθανών κινήσεων. Αν δεν υπάρχει τέτοιο κελί, τότε επιστρέφει false. Για την επίτευξη αυτής της λειτουργίας χρησιμοποιείται η συνάρτηση getNumberofNeighbours, η οποία θα αναλυθεί στην συνέχεια, εντός ενός βρόχου for. Μέσω του βρόχου δίνονται ως κελιά-ορίσματα στην getNumberofNeighbours τα κελιά της λίστας κινήσεων το ένα μετά το άλλο, και αυτή ελέγχει και επιστρέφει το πλήθος γειτόνων που έχει το εκάστοτε κελί εντός της λίστας πιθανών κινήσεων.

def getNumberofNeighbours(*cell*, *initialNimBoard*, *possibleMoves*):

    count = 0

    neighbours = getCellNeighbours(*cell*, *initialNimBoard*)

    for i in range(0, len(neighbours)):

        if neighbours[i] in *possibleMoves*:

            count = count + 1

    return count

Η getNumberofNeighbours δέχεται ως ορίσματα ένα κελί, την αρχική μορφή του ταμπλό και μία λίστα πιθανών κινήσεων η οποία έχει αυθαίρετο μήκος, και δίνει ως έξοδο έναν αριθμό που δείχνει πόσα από τα γειτονικά κελιά του δοσμένου κελιού βρίσκονται στην λίστα των πιθανών κινήσεων. Για την εύρεση όλων των γειτόνων χρησιμοποιείται η getCellNeighbours, η οποία θα αναλυθεί στην συνέχεια, ενώ η εύρεση του αριθμού γειτόνων εντός της λίστας possibleMoves πραγματοποιείται με έναν απλό βρόχο for.

def getCellNeighbours(*cell*, *initialNimBoard*):

    neighbours = []

    row1 = getRowCells(*initialNimBoard*, 1)

    col1 = getColumnCells(*initialNimBoard*, 1)

    rowN = getRowCells(*initialNimBoard*, N\*N - N + 1)

    colN = getColumnCells(*initialNimBoard*, N\*N)

*# cells of first row*

    if (*cell* in row1) and *cell* != N:

        neighbours.append(*cell*+1)

        neighbours.append(*cell*+N)

        if *cell* != 1:

            neighbours.append(*cell*-1)

*# cells of first column*

    elif *cell* in col1:

        neighbours.append(*cell*+1)

        neighbours.append(*cell*-N)

        if *cell* != (N\*N - N + 1):

            neighbours.append(*cell*+N)

*# cells of last row*

    elif *cell* in rowN:

        neighbours.append(*cell*-1)

        neighbours.append(*cell*-N)

        if *cell* != N\*N:

            neighbours.append(*cell*+1)

*# cells of last column*

    elif *cell* in colN:

        neighbours.append(*cell*-1)

        neighbours.append(*cell*+N)

        if *cell* != N:

            neighbours.append(*cell*-N)

*# other cells*

    else:

        neighbours.append(*cell*-1)

        neighbours.append(*cell*+1)

        neighbours.append(*cell*-N)

        neighbours.append(*cell*+N)

    return neighbours

Τέλος, η getCellNeighbours δέχεται ως είσοδο ένα κελί και την αρχική μορφή του ταμπλό, και επιστρέφει όλα τα γειτονικά κελιά του κελιού που δόθηκε ως όρισμα. Ανάλογα με το είδος κελιού που δόθηκε (γωνιακό, ακριανό ή ενδιάμεσο), τα γειτονικά κελιά υπολογίζονται με βάση την σχετική τους θέση ως προς το δοσμένο κελί. Για την εύρεση των ακριανών κελιών (δηλαδή των κελιών των πρώτων και τελευταίων γραμμών και στηλών που δεν είναι γωνιακά) χρησιμοποιούνται οι συναρτήσεις getColumnCells και getRowCells με κατάλληλα ορίσματα.

Αφού λοιπόν αναλύσαμε τι γίνεται στην περίπτωση που ο υπολογιστής διαπιστώσει ότι υπάρχουν 5 ή λιγότερα κενά κελιά στο ταμπλό, μπορούμε να αναλύσουμε τώρα πώς λειτουργεί κάθε στρατηγική του υπολογιστή.

* + Για την Random στρατηγική:

Καλείται η randomMoreThanFiveLeft :

def randomMoreThanFiveLeft(*nimBoard*, *initialNimBoard*):

    possibleMoves = getAvailableCells(*nimBoard*)

    choice = []

    tile = random.choice(possibleMoves)

    choice.append(tile)

*# randomly select a number between 1 and 3*

    num = random.randint(1, 3)

*#  select if the next tile will be in the same row or column*

    if random.randint(0, 1) == 0:

        where = 'row'

    else:

        where = 'column'

    if random.randint(0, 1) == 0:

        to\_where = 'up'

    else:

        to\_where = 'down'

    i = 1

    if (num > 1 and tile not in getDiagonalCells(*initialNimBoard*)):

        while i < num:

            flag = 0

*# check the where and to\_where and then check the board if the next tile move is valid.*

*# If it is valid then append it to the choice list if not then break*

            if where == 'row' and to\_where == 'up':

                next\_tile = tile - 1

                if next\_tile in possibleMoves and checkValidMove(*initialNimBoard*, choice + [next\_tile]):

                    tile = next\_tile

                    choice.append(tile)

                else:

*# if the move is not valid, change the to\_where variable and try again*

                    to\_where = 'down'

                    if num == 3 and i == 2:

                        tile = choice[0]

                    flag = 1

            if where == 'row' and to\_where == 'down':

                next\_tile = tile + 1

                if next\_tile in possibleMoves and checkValidMove(*initialNimBoard*, choice + [next\_tile]):

                    tile = next\_tile

                    choice.append(tile)

                else:

*# if the move is not valid, change the to\_where variable and try again*

                    to\_where = 'up'

                    i = i-1

                    if num == 3 and i == 2:

                        tile = choice[0]

*# if the value of flag is 1 then break the loop (to\_where is already changed)*

                    if flag == 1:

                        break

            if where == 'column' and to\_where == 'up':

                next\_tile = tile - int(math.sqrt(len(*nimBoard*)))

                if next\_tile in possibleMoves and checkValidMove(*initialNimBoard*, choice + [next\_tile]):

                    tile = next\_tile

                    choice.append(tile)

                else:

*# if the move is not valid, change the to\_where variable and try again*

                    to\_where = 'down'

                    if num == 3 and i == 2:

                        tile = choice[0]

                    flag = 1

            if where == 'column' and to\_where == 'down':

                next\_tile = tile + int(math.sqrt(len(*nimBoard*)))

                if next\_tile in possibleMoves and checkValidMove(*initialNimBoard*, choice + [next\_tile]):

                    tile = next\_tile

                    choice.append(tile)

                else:

*# if the move is not valid, change the to\_where variable and try again*

                    to\_where = 'up'

                    i = i-1

                    if num == 3 and i == 2:

                        tile = choice[0]

*# if the value of flag is 1 then break the loop (to\_where is already changed)*

                    if flag == 1:

                        break

            i = i+1

    return choice

Αρχικά δημιουργείται μια λίστα choice που στο τέλος θα περιέχει τα κελιά που επέλεξε ο υπολογιστής να παίξει. Στην συνέχεια επιλέγεται τυχαία και ισοπίθανα πόσα κελιά θα γράψει ο υπολογιστής, αν θα είναι στην ίδια στήλη ή γραμμή και η κατεύθυνσή τους (από δεξιά προς αριστερά ή το αντίθετο για γραμμή και από πάνω προς κάτω ή το αντίθετο για στήλη). Αφού γίνουν και αποθηκευτούν όλες αυτές οι τυχαίες επιλογές προχωράμε στην κύρια λειτουργία της συνάρτησης.

Αν το νούμερο των κελιών που επιλέχθηκε είναι μεγαλύτερο από 1 (αν είναι 1 απλά το γράφει, και στην λίστα βρίσκεται μόνο αυτό το κελί) και το πρώτο κελί δεν ανήκει στην κύρια διαγώνιο (από την εκφώνηση αν κάποιος επιλέξει πρώτο κελί στην διαγώνιο δεν επιτρέπεται να επιλέξει άλλο κελί στην συγκεκριμένη σειρά), τότε εκτελείται η if. Εντός της υπάρχουν 4 περιπτώσεις όπου πραγματοποιούνται σχεδόν ίδιες λειτουργίες. Η είσοδος σε κάθε περίπτωση εξαρτάται από τις τυχαίες τιμές που πήραν οι μεταβλητές στην αρχή της συνάρτησης σχετικά με την επιλογή κατεύθυνσης και στήλης ή γραμμής.

Η συνάρτηση έχει φτιαχτεί έτσι ώστε να προσπαθεί να κάνει κίνηση με μέγεθος όσο πιο κοντά γίνεται στην τυχαία τιμή της μεταβλητής num. Για παράδειγμα αν έχει επιλέξει να παίξει κατά γραμμή, να γράψει 3 κελιά και να έχει κατεύθυνση προς τα αριστερά, το πρόγραμμά μας έχει την δυνατότητα αν συναντήσει εμπόδιο οποιουδήποτε είδους να αλλάξει την κατεύθυνση (να την κάνει προς τα δεξιά), για να ελέγξει αν μπορεί να γράψει προς τα εκεί τα κελιά που υπολείπονται. Παρακάτω δίνεται ένα παράδειγμα λειτουργίας, όπου ο υπολογιστής έχει το πράσινο χρώμα και τώρα είναι η σειρά του.

Έστω ότι το τυχαίο κελί που επιλέχθηκε αρχικά είναι το 4, θέλει να παίξει κατά γραμμή προς τα αριστερά και να γράψει 3 κελιά. Δεν θα τα καταφέρει, καθώς αφού γράψει στο 4 και στο 3 όπως φαίνεται το 2 δεν είναι διαθέσιμο. Άρα κανονικά θα τελείωνε εκεί, παίζοντας αναγκαστικά μόνο τα κελιά 4 και 3. Όμως με την αναδρομή που υπάρχει στον κώδικα (και θα εξηγηθεί στην συνέχεια) ο υπολογιστής αφού βρει εμπόδιο στο κελί 2 θα εξετάσει και την άλλη κατεύθυνση (δεξιά εδώ) του αρχικού κελιού (4), προσπαθώντας έτσι να γράψει πράγματι 3 κελιά. Συνεπώς αφού το κελί 5 είναι διαθέσιμο θα γράψει τελικά 4,3,5, και θα ολοκληρώσει την σειρά του έχοντας μια νόμιμη κίνηση μεγέθους num που παίχτηκε κατά γραμμή και περιέχει το αρχικό κελί.

Αυτή η δυνατότητα έχει προστεθεί έτσι ώστε ο υπολογιστής να μπορεί να κάνει κινήσεις μήκους 2 και 3 με φυσιολογική συχνότητα, καθώς αυτές πρακτικά θα γίνονταν αρκετά πιο σπάνια αν ο υπολογιστής ήταν αναγκασμένος να παίζει τηρώντας αυστηρά όλες τις αρχικές του επιλογές, λόγω των διαφόρων εμποδίων που ενδέχεται να συναντήσει κατά την κίνησή του.

A picture containing text, screenshot, font, number

Description automatically generated

Επιστρέφοντας στον κώδικα, βλέπουμε ότι υπάρχει μια μορφή αναδρομικότητας. Αν παραδείγματος χάριν παίζει κατά γραμμή, τότε στον κώδικα θα ελέγχεται η κατεύθυνση που επιλέχθηκε αρχικά, και αν απαγορεύεται να παίξει το επόμενο κελί προς αυτήν την κατεύθυνση τότε υπάρχουν δυο σενάρια. Αν το κελί που προσπάθησε να παίξει είναι το δεύτερο της κίνησής του, τότε απλά εξετάζει την αντίθετη κατεύθυνση (αφού πρώτα αυξήσει ένα flag ώστε να ξέρει ότι έγινε ήδη μια αλλαγή κατεύθυνσης και να μην παγιδευτεί σε loop). Αν το κελί είναι το τρίτο της κίνησής του, τότε εκτός από την κατεύθυνση πρέπει να αλλάξει και το κελί δίπλα από το οποίο θα ψάξει.

Πχ στο προηγούμενο παράδειγμα, αφού γραφτεί το κελί 3 και διαπιστωθεί ότι δεν μπορεί να γραφτεί το 3ο κελί επειδή το κελί 2 δεν είναι διαθέσιμο, δεν αρκεί να αλλάξει μόνο η κατεύθυνση. Θα πρέπει να αλλάξει και το κελί από το οποίο θα αρχίσει η νέα αναζήτηση, ώστε η νέα αναζήτηση να αρχίσει από το αρχικό κελί (το 4) και όχι από το 3.

Για αυτό και υπάρχει υπό συνθήκη η εντολή tile = choice[0], η οποία αλλάζει το κελί από όπου ξεκινά την αναζήτηση ο υπολογιστής. Αν όμως είμαστε στην περίπτωση όπου η to\_where = down τότε δεν μπορούμε απλά να αλλάξουμε την μεταβλητή, αφού ο κώδικας για την περίπτωση to\_where = up έχει προσπεραστεί ήδη. Για αυτόν τον λόγο μειώνεται το i, που είναι ο iterator της while, ώστε να ξανατρέξει η while με ίδιο iterator (έχοντας και εδώ την μεταβλητή flag και αλλάζοντας την κατάλληλα με την αλλαγή του iterator). Έτσι τώρα έχει ολοκληρωθεί η συνάρτηση για την επιλογή κίνησης από τον υπολογιστή με την στρατηγική random.

* Για την First Fit:

Καλείται η firstfreeMoreThanFiveLeft:

def firstfreeMoreThanFiveLeft(*nimBoard*, *initialNimBoard*):

*# This function returns the first free move from the list of*

*# available moves.*

*# return the first available tile to start the move*

    possibleMoves = getAvailableCells(*nimBoard*)

    choice = []

    tile = possibleMoves[0]

    choice.append(tile)

*# randomly select a number between 1 and 3*

    num = random.randint(1, 3)

*#  select if the next tile will be in the same row or column*

    if random.randint(0, 1) == 0:

        where = 'row'

    else:

        where = 'column'

    to\_where = 'down'

    if (num > 1 and tile not in getDiagonalCells(*initialNimBoard*)):

        for i in range(1, num):

*# check if the next move is valid*

            if where == 'row' and to\_where == 'down':

                next\_tile = tile + 1

                if next\_tile in possibleMoves and checkValidMove(*initialNimBoard*, choice + [next\_tile]):

                    tile = next\_tile

                    choice.append(tile)

                else:

                    break

            elif where == 'column' and to\_where == 'down':

                next\_tile = tile + int(math.sqrt(len(*nimBoard*)))

                if next\_tile in possibleMoves and checkValidMove(*initialNimBoard*, choice + [next\_tile]):

                    tile = next\_tile

                    choice.append(tile)

                else:

                    break

    return choice

Όπως φαίνεται, πάλι γίνεται τυχαία επιλογή για τον αριθμό των κελιών της κίνησης, καθώς και για το αν η κίνηση θα γίνει κατά στήλη ή κατά γραμμή. Αυτό που αλλάζει σε σχέση με πριν είναι ότι αυτή την φορά το πρώτο κελί δεν είναι τυχαίο αλλά το πρώτο ελεύθερο, και η κατεύθυνση είναι πάντα προς τα δεξιά κατά γραμμή και προς τα κάτω κατά στήλη (αλλιώς η κίνηση θα έπεφτε πάνω σε κατειλημμένα κελιά). Ο κώδικάς της είναι πιο απλός από της randomMoreThanFiveLeft, μιας και το to\_where είναι σταθερό και άρα διακρίνονται 2 περιπτώσεις. Αν παίζει κατά γραμμή τότε ελέγχει αν το διπλανό κελί είναι ελεύθερο και αποτελεί νόμιμη κίνηση, και αν ναι τότε το βάζει στην λίστα με τα κελιά, ενώ αν όχι τότε ολοκληρώνει την κίνησή του έχοντας επιλέξει μόνο τα πρώτα κενά κελιά που βρήκε στο ταμπλό. Το ίδιο συμβαίνει και αν παίζει κατά στήλη.

* Για την Copycat:

Καλείται η copycatMoreThanFiveLeft:

def copycatMoreThanFiveLeft(*nimBoard*, *initialNimBoard*, *playermoves*):

*# This function returns the copycat move from what the player played. If the computer goes first or the player move can be copied then the computer chooses either a random move with getComputerMove\_random or a first free move with getComputerMove\_firstfit.*

*# If the players move cant be copied then the move randomly chosen must contain as many tiles as the player move or less, but not more.*

*# return the first available tile to start the move*

    possibleMoves = getAvailableCells(*nimBoard*)

    init\_possibleMoves = getAvailableCells(*initialNimBoard*)

    choice = []

*# if the computer goes first*

    if len(possibleMoves) == len(init\_possibleMoves):

        if random.randint(0, 1) == 0:

            choice = getComputerMove\_random(*nimBoard*, *initialNimBoard*)

        else:

            choice = getComputerMove\_firstfit(*nimBoard*, *initialNimBoard*)

*# if the player goes first*

    else:

*# find the move the player performed last and copy it by doing the same move symmetrically by the diagonal, The players last move is in the playermoves list*

*# find the tile that is symmetric to the player move*

*# if the players move is on the diagonal then the computer chooses the tile in the diagonal that is symmetric in the center of the diagonal.*

*# If the cell is not available then the computer chooses a random cell in the diagonal.*

        if *playermoves*[0] in getDiagonalCells(*initialNimBoard*):

            diagon = getDiagonalCells(*initialNimBoard*)

            for i in range(len(diagon)):

                if *playermoves*[0] == diagon[i]:

                    if diagon[len(diagon)-1-i] in possibleMoves:

choice.append(diagon[len(diagon)-1-i])

                    else:

*# check first if any of the elements of the diagonal are possible moves*

*# if not then choose a random move*

                        if not any(x in possibleMoves for x in diagon):

                            choice.append(possibleMoves[0])

                        else:

*# choice is equal to a random empty cell in the diagonal*

                            while True:

                                choice = []

                                choice.append(random.choice(diagon))

                                if choice[0] in possibleMoves:

                                    break

*# if choice is not empty then break*

                if choice:

                    break

*# if the players move is not on the diagonal then the computer chooses the tile that is symmetric to the player move,the symmetry is determined by the diagonal*

        else:

            flag = 0

            for i in range(len(*playermoves*)):

                row = (*playermoves*[i] - 1) // N  *# 0-indexed row*

                col = (*playermoves*[i] - 1) % N   *# 0-indexed column*

*# find the tile that is symmetric to the player move*

                symmetric = col \* N + row + 1

                choice.append(symmetric)

                if choice[i] not in possibleMoves:

                    flag = 1

            if flag == 1:

*# even if one tile of move is not valid, choose a random move*

                if random.randint(0, 1) == 0:

                    choice = getComputerMove\_random(*nimBoard*, *initialNimBoard*)

                    while len(choice) > len(*playermoves*):

                        choice = getComputerMove\_random(

*nimBoard*, *initialNimBoard*)

                else:

                    choice = getComputerMove\_firstfit(

*nimBoard*, *initialNimBoard*)

                    while len(choice) > len(*playermoves*):

                        choice = getComputerMove\_firstfit(

*nimBoard*, *initialNimBoard*)

    return choice

Αρχικά όπως και στις προηγούμενες συναρτήσεις ορίζουμε στην αρχή την λίστα που θα έχει τα κελιά της κίνησης, και έπειτα βρίσκουμε πόσα ελεύθερα κελιά υπάρχουν στο ταμπλό στον τρέχοντα γύρο. Αν τα τρέχοντα ελεύθερα κελιά ισούνται με τα κελιά του αρχικού ταμπλό (δηλαδή αν είναι η πρώτη κίνηση του παιχνιδιού), τότε ο υπολογιστής επιλέγει τυχαία μία από τις προηγούμενες δύο στρατηγικές και εκτελεί την πρώτη κίνησή του με βάση αυτήν. Αυτό γίνεται επειδή σε αυτήν την περίπτωση δεν υπάρχει κίνηση να αντιγράψει, αφού ο παίκτης δεν έχει παίξει ακόμα. Αν ο παίκτης έχει παίξει έστω και μια φορά, τότε πάλι διακρίνονται υποπεριπτώσεις.

Αν ο παίκτης έχει παίξει κελί της διαγωνίου, τότε ο υπολογιστής προσπαθεί να παίξει το συμμετρικό κελί της διαγωνίου ως προς το κέντρο του ταμπλό. Αυτό γίνεται με την choice.append(diagon[len(diagon)-1-i]),

όπου diagon λίστα με όλα τα κελιά της διαγωνίου. Αν όμως το συγκεκριμένο κελί της διαγωνίου δεν είναι διαθέσιμο, τότε γεμίζει ένα άλλο τυχαίο κελί της διαγωνίου. Αν κανένα κελί της διαγωνίου δεν είναι ελεύθερο, τότε επιλέγεται το πρώτο μη-κενό κελί του ταμπλό. Ο σχετικός έλεγχος πραγματοποιείται με την if not any(x in possibleMoves for x in diagon).

Για το κύριο κομμάτι της συνάρτησης, αν ο παίκτης κάνει μία κίνηση ο υπολογιστής θα πρέπει να την αντιγράψει και να παίξει την συμμετρική αυτής ως προς την διαγώνιο. Αφού έχουμε την playermoves (την κίνηση του παίκτη που θέλουμε να αντιγράψει ο υπολογιστής), θα πρέπει όλα τα συμμετρικά κελιά των κελιών της playermoves να είναι ελεύθερα. Αν είναι, τότε απλά προστίθενται στην λίστα choice και επιλέγονται. Αν έστω και ένα δεν είναι, τότε η μεταβλητή flag γίνεται 1 και ο υπολογιστής επιλέγει τυχαία και ισοπίθανα αν θα εκτελέσει την κίνησή του με random ή first fit στρατηγική. Και στις δυο περιπτώσεις θα πρέπει το μήκος της κίνησης να είναι το πολύ ίσο με το μήκος της κίνησης του παίκτη που προσπαθούμε να αντιγράψουμε. Συνεπώς όποια στρατηγική και να επιλεχθεί, ελέγχεται αν το αποτέλεσμά της είναι μικρότερο ή ίσο από το αποτέλεσμα της κίνησης του παίκτη. Μόλις βρεθεί μία τέτοια κίνηση, αποθηκεύεται στην choice και εκτελείται.

Αφού αναλύσαμε τις συναρτήσεις και το πρόγραμμα, ας δούμε πως λειτουργεί στην πράξη με μερικά ενδεικτικά παραδείγματα. Σημειώνεται ότι κάθε κίνηση του υπολογιστή πριν τυπωθεί στο ταμπλό τυπώνεται και ως λίστα, έτσι ώστε να είναι ξεκάθαρο ποια κελιά έπαιξε ο υπολογιστής στην σειρά του. Στα παρακάτω στιγμιότυπα ο παίκτης παίζει με κόκκινο και ο υπολογιστής με πράσινο χρώμα.

* Παράδειγμα Random στρατηγικής:

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

Παρακάτω όπως φαίνεται προσπαθήσαμε να δώσουμε κελί της διαγωνίου ως 2ο κελί της κίνησης, και το πρόγραμμα ορθά μάς εμπόδισε. Φαίνεται επίσης πως γράφοντας το αρχικό κελί δύο φορές και μετά το διπλανό του τερματίσαμε την σειρά μας έχοντας επιλέξει αυτά τα δυο κελιά.

A screenshot of a computer program

Description automatically generated with medium confidence

Εδώ βλέπουμε πως δεν έχει σημασία με ποια σειρά δίνονται τα κελιά, αρκεί τελικά να είναι διαδοχικά:

A screenshot of a computer program

Description automatically generated with medium confidence

Όπως φαίνεται παρακάτω, αφού γεμίσουμε το κελί 7 ο υπολογιστής καλείται να παίξει σε παρτίδα όπου υπάρχουν 5 κενά κελιά, συνεπώς θα ψάξει να βρει νικητήρια κίνηση. Στην συγκεκριμένη παρτίδα δεν μπορεί να παίξει 3 συνεχόμενα κελιά, ενώ όποιο κελί και να παίξει του εξασφαλίζει νίκη, άρα όπως είναι αναμενόμενο επιλέγει ως κίνηση το πρώτο ελεύθερο κελί που συναντά, δηλαδή το 6:

A screenshot of a computer

Description automatically generated with low confidence

Εδώ βλέπουμε ότι αν δώσουμε ως είσοδο κάτι εκτός από ακέραιο, τότε κρατάει τα κελιά που έχουμε δώσει ως τώρα και λήγει την σειρά μας (αντί για 2ο κελί έχει δοθεί ως είσοδος το κενό, πατώντας enter). Επιπλέον, όπως φαίνεται ο υπολογιστής στην σειρά του βλέπει 3 κενά κελιά, 2 διαδοχικά και ένα σπαστό, άρα όπως είναι αναμενόμενο με βάση τον τρόπο που λειτουργεί η threeCellsLeft επιλέγει να γεμίσει κάποιο από τα δύο διαδοχικά:

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with low confidence

* Παράδειγμα First Fit στρατηγικής:

A screenshot of a computer game

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with low confidence

Όπως φαίνεται παρακάτω, αφού γεμίσουμε το κελί 16 ο υπολογιστής καλείται να παίξει σε παρτίδα όπου υπάρχουν 4 κενά κελιά, άρα θα ψάξει να βρει νικητήρια κίνηση εάν υπάρχει. Η παρτίδα στην οποία παίζει είναι η (2,2), η οποία ως γνωστόν αποτελεί παρτίδα ήττας, και άρα όπως είναι αναμενόμενο ο υπολογιστής δεν βρίσκει νικητήρια κίνηση και συνεχίζει να παίζει με στρατηγική first fit, για αυτό και βλέπουμε ότι η κίνησή του ξεκινά από το πρώτο ελεύθερο κελί του ταμπλό (το 8).

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

* Παράδειγμα Copycat στρατηγικής:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Αν ο παίκτης τοποθετήσει κελί στην διαγώνιο, ο υπολογιστής απαντά παίζοντας το συμμετρικό ως προς το κέντρο κελί της διαγωνίου (εφόσον αυτό είναι δυνατόν):

A screenshot of a computer

Description automatically generated with low confidence

A screenshot of a computer

Description automatically generated with low confidence

A screenshot of a computer program

Description automatically generated with medium confidence

Όπως βλέπουμε στο παρακάτω στιγμιότυπο, αν έστω και ένα από τα συμμετρικά κελιά της κίνησης του παίκτη είναι πιασμένα, τότε ο υπολογιστής κάνει μια random ή first fit κίνηση (στην συγκεκριμένη περίπτωση random).

A screenshot of a computer program

Description automatically generated with low confidence

Και παρακάτω φαίνεται πως αν δεν μπορεί να βρει το συμμετρικό της διαγωνίου και δεν υπάρχουν άλλα κενά κελιά στην διαγώνιο τότε ο υπολογιστής επιλέγει το πρώτο κενό κελί στο ταμπλό ως κίνησή του.

A screenshot of a computer

Description automatically generated with low confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

Και εδώ όπως είναι αναμενόμενο ο υπολογιστής αναγνωρίζει πως έχει παρτίδα νίκης, και συνεπώς παύει να παίζει με την αρχική στρατηγική του και επιλέγει την πρώτη νικητήρια κίνηση που βρίσκει, δηλαδή το κελί 6:

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer game

Description automatically generated with medium confidence

Σημειώνεται ότι για την έναρξη του παιχνιδιού (ή για έναρξη νέου παιχνιδιού μετά την ολοκλήρωση του προηγούμενου) οποιαδήποτε είσοδος αρχίζει από y ή Y εκλαμβάνεται ως “yes”, ενώ οποιαδήποτε άλλη είσοδος τερματίζει το πρόγραμμα.