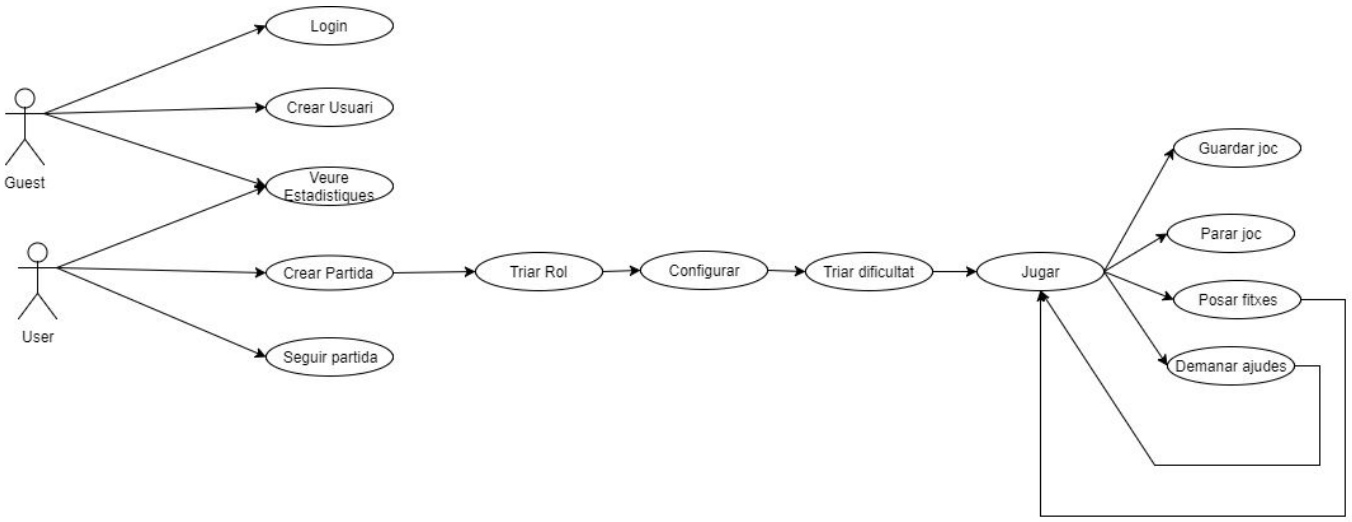
MASTERMIND

Versió 2.0

Elena Alonso González <elena.alonso.gonzalez@est.fib.upc.edu>
Oriol Borrell Roig <oriol.borrell.roig@est.fib.upc.edu>
Alexis Rico Carreto <alexis.rico@est.fib.upc.edu>
08/01/2018

Índex

1. Casos d'ús



- Guest

Un guest (convidat) al executar el programa, pot triar entre identificar-se (en cas que ja tingui un compte creat), crear un nou usuari, o veure els millors resultats de tots els usuaris existents. Un cop es registrat o identificat aquest guess passa a ser User.

- User

Un user pot veure els seus records o el rànquing de tots els usuaris existents, Crear una nova partida, o jugar en una partida creada anteriorment. En cas de crear una Partida, haurà de seguir els passos següents:

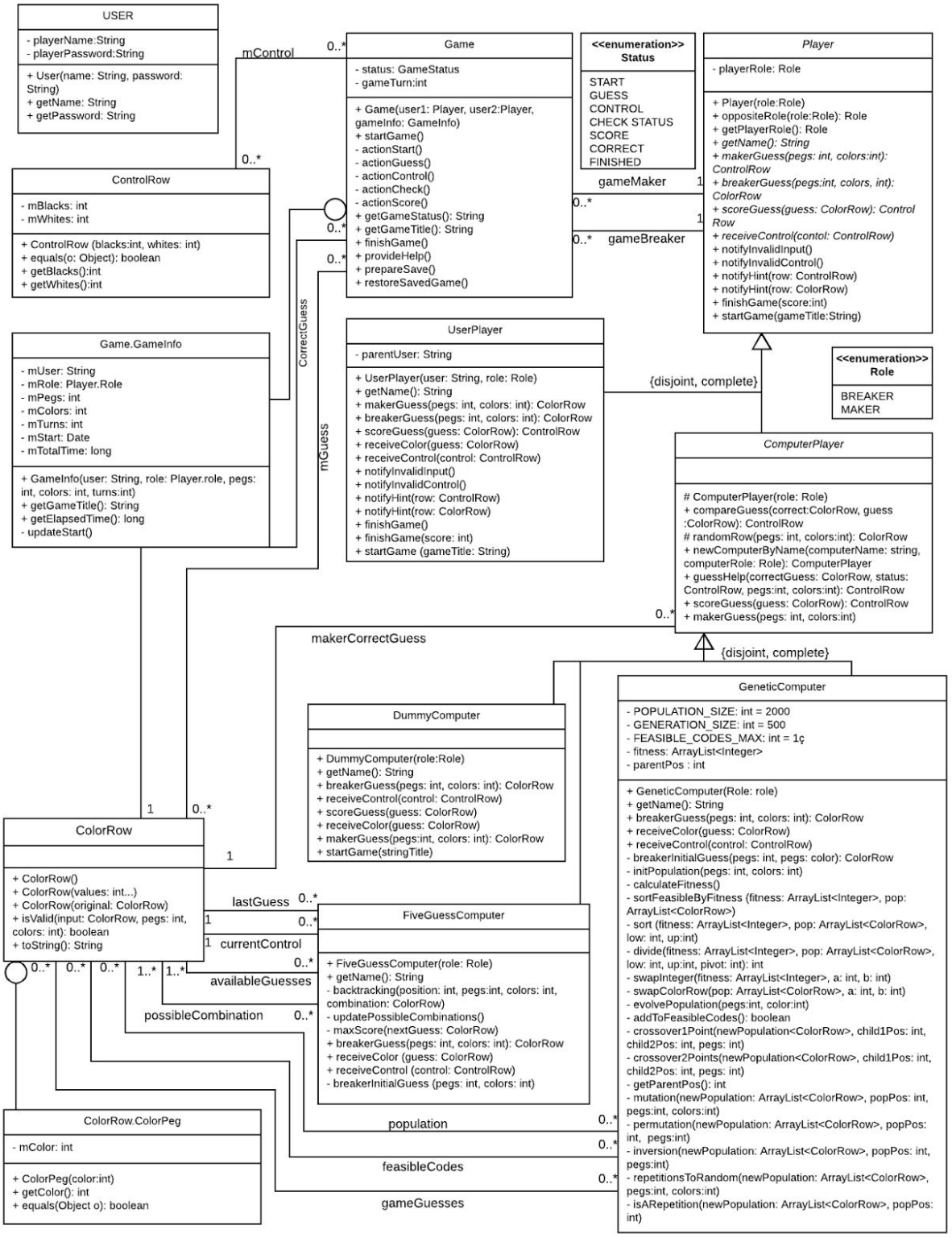
- Escollir entre rol de Maker o Breaker. Si el rol triat és Breaker, haurà de triar també l'algorisme amb que vol que l'ordinador resolgui la combinació triada.
- Triar una dificultat o personalitzar-se una partida. Si tria la segona opció haurà de triar el nombre de fitxes, el nombre de colors possibles en una combinació.

Un cop escollits els paràmetres anteriors, es crearà la partida, i començaran les rondes. En cada ronda, el jugador tindrà la possibilitat de:

- Guardar i sortir del joc, per poder seguir jugant en un altre moment.
- Parar el joc, i sortir sense guardar (en cas de no haver guardat abans).
- Introduir un nou intent (en cas de que el rol sigui Breaker) o control (en cas de que rol sigui Maker).
- Demanar ajudes.

En el cas de parar el joc, l'usuari tornarà al principi de tot (dintre de les possibilitats d' usuari registrat).

2. Diagrama estàtic de domini de l'esquema conceptual de les dades (disseny)



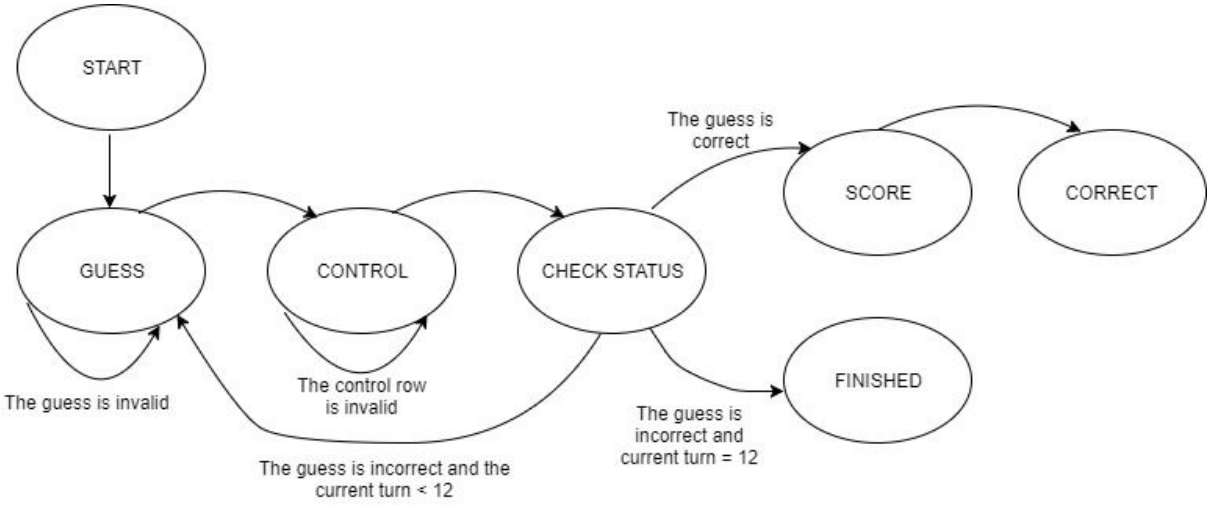
3. Funcionalitats Principals

3.1. Interacció amb l'usuari

La classe que s'encarrega de gestionar els torns per tal que la interacció amb l'usuari funcioni correctament, és la classe Game. En aquesta classe, podem trobar els següents atributs:
gameMaker → és el jugador (usuari o ordinador) que crea la combinació a endevinar.
gameBreaker → és el jugador (usuari o ordinador) que intenta endevinar la combinació.
gameInfo → és una classe GameInfo que conté els atributs usuari, rol, fitxes, colors i torns de la partida.
mGuess → és un arrayList que conté tots els intents que ha fet el Breaker durant la partida.
mControl→ és un arrayList que conté totes les combinacions de control que ha tornat el Maker.
gameTurn → és el nombre de torns que porta jugats el Breaker.
gameStatus → correspon a un Status que és una enumeració que indica en què estat es troba la partida. Els estats poden ser:

- START → el maker decideix la combinació que haurà d'endevinar el breaker
- GUESS → el breaker dona una combinació per tal d'endevinar la combinació feta pel maker
- CONTROL → el maker retorna una combinació de control respecte a la última combinació feta pel breaker
- STATUS CHECK → es comprova quin serà el següent estat: GUESS, SCORE o FINISHED
- SCORE → es calcula una puntuació pel breaker, en cas de ser guanyador i usuari
- CORRECT → la partida ha finalitzat havent endevinat la combinació
- FINISHED → la partida ha finalitzat perquè s'han acabat els torns

És aquest gameStatus el que defineix l'estat de la partida i, depenent d'aquest és decideix el comportament que s'ha de dur a terme. El canvi d'estats funcionaria així:



Depenent de l'estat en que ens trobem es crida a una funció o a una altra. En el cas de START, GUESS i CONTROL, la funció a la que instancien, crida a una altra funció de la classe abstracta Player, i depenent de si es tracta d'un jugador usuari, o un jugador ordinador, el comportament serà diferent.

Entre aquests estats, destaquen dos funcions, **breakerGuess()** i **compareGuess()**.

BreakerGuess en cas de tractar-se de un jugador ordinador, és el que ha de executar un dels algorismes, el triat per l'usuari, per tal d'acabar encertant la combinació. Aquests algorismes poden ser: el Five Guess, el Genetic o el Dummy.
El Dummy és un algorisme que tria les combinacions a provar de forma aleatòria sense tenir en compte les combinacions de fitxes de control retornades pel Maker.
Els altres dos algorismes, degut a la seva complexitat, els explicarem més endevant.

L'altra funció esmentada, **compareGuess()**, donades dos combinacions de fitxes de colors, calcula la combinació de fitxes de control que hi ha entre elles dues. És una funció que s'utilitza tant per respondre a una combinació feta pel Breaker en cas que l'ordinador sigui Maker, com per comprovar la correctesa de la seqüència de control retornada per l'usuari Maker o donar-li una ajuda aquest indicant la combinació correcta a posar. Aquesta funció, com veurem més endevant, és utilitzada també al FiveGuess.

El primer que fa la funció és modificar el nombre de fitxes blanques amb el valor 0, i fer una copia de les combinacions passades per paràmetre perquè aquestes no es vegin modificades.

A continuació executa un bucle per recòrrer totes les posicions que té una combinació. Es compara la mateixa posició de la combinació correcta i la combinació introduida. Si coincideixen s'incrementa el nombre de negres i modifiquem el valor per aquesta posició en les còpies per nul, perquè després el programa no les pugui tornar a contar com a blanques.

Per calcular el nombre de blanques, es fa un bucle igual per recòrrer la còpia de la combinació correcta, però dintre d'aquest es necessari fer-ne un altre que recorri la còpia de la combinació introduïda ja que ara les fitxes que siguin iguals no es trobaran a la mateixa posició. Si una fitxa esta en totes dues combinacions s'incrementarà el nombre de blanques en 1 i la posició que ha coincidit passarà a ser nul en la còpia de la combinació introduida. En el cas de la còpia de la combinació correcta no cal fer-ho perquè no tornarem a recòrrer aquella posició.

Per últim creem una nova fila de control amb els valors obtinguts i la retornem.

Una altra funció interessant que podem trobar és l'**actionscore()**, que s'executa en cas que l'Status sigui SCORE i per tant, la partida hagi finalitzat amb èxit. Només s'obtindrà puntuació en cas de ser Breaker l'usuari. La fòrmual que utilitzem és:
ombre de colors ^ nombre de pegs / gameTurn

3.2. Five Guess Algorithm

L'algorisme Five Guess, com bé indica el seu nom es tracta d'un algorisme que, mitjançant la tècnica de minimax, endevina la combinació proposada pel Code Breaker en una mitjana de 4.340 torns pel cas habitual de 4 posicions i 6 colors. Aquest algorisme va ser demostrat per Donald Knuth en 1977.

Per procedir a la programació de l'algorisme, vam crear una classe anomenada FiveGuessComputer, que és extensió de la classe ComputerPlayer i que té els següents atributs:

totalCombinations → és un int que indica el nombre de possibles combinacions totals amb les pegs i els colors donats. En el joc original sempre és 1296, ja que tenim 4 fitxes i 6 colors (4^6), però nosaltres permetem jugar amb el nombre de fitxes i colors que l'usuari desitgi i per tant l'hem de calcular per cada nova partida.

lastGuess → és un objecte ColorRow que és un arraylist de Pegs i ens indica la última combinació que hem provat per tal de endevinar la combinació feta pel Code Maker.

currentControl → és un objecte ControlRow, format pel nombre de fitxes blanques i negres que el Code Maker ens ha tornat en la última jugada.

possibleCombinations → és una arrayList de ColorRow, on estan totes les combinacions que encara no han sigut descartades, és a dir, que podrien ser la combinació guanyadora.

availableGuesses → és una arrayList de ColorRow on estan totes les combinacions que encara no s'han provat.

FUNCIONS PRINCIPALS

L'algorisme està composat, en el nostres cas, per cinc funcions diferents **breakerGuess(), backtracking(), breakerInitialGuess(), updatePossibleCombinations() i scoreGuess()**.

L'algorisme comença amb la crida a la funció **breakerGuess()**. El primer que ha de fer la funció quan és invocada per primera vegada en un nou joc és omplir dues llistes amb totes les combinacions possibles tenint en compte el nombre de fitxes i colors permesos. Nosaltres hem fet dos ArrayList<ColorRow> anomenats possibleCombinations i availableGuesses. Per obtenir totes les operacions, vam utilitzar l'algorisme **backtracking().**

Per últim, s'ha d'assignar la que serà la primera combinació que provarem, que en el nostre cas l'anomenarem last guess, per poder tindre-la en compte el proper cop que executem la funció. Per fer-ho instanciarem la funció **breakerInitialGuess()**.

En el cas de l'algorisme original per 4 pegs i 6 colors la primera combinació a provar sempre és 1 1 2 2, per a 5 pegs hem comprovat que serà 1 1 2 2 3 i per la resta de nombres

calculem una combinació random. Una altra opció hagués sigut utilitzar també el minimax per obtenir la combinació que més combinacions eliminaria, però l'execució seria més costosa en temps.

En cas de no ser el primer cop que es crida a la funció en una mateixa partida, el primer que s'ha de fer és eliminar totes les combinacions que ja no són possibles. Per fer-ho, hem utilitzat la funció **updatePossibleCombinations()**. El que fa és comparar cadascuna de les combinacions que encara poden ser guanyadores (possibleCombinations) amb la última combinació provada. Per a aquesta tasca utilitzem l'algoritme utilitzat abans **compareGuess()**. A continuació comparem les fitxes de control donades per l'algorisme amb el control obtingut del Maker. En cas de no ser iguals, és a dir, el nombre de blanques i de negres no coincideixen en ambdues combinacions, la combinació que està sent comparada s'eliminarà de la llista que estàvem examinant.

A continuació, s'avalua quina combinació serà la que més possibles combinacions eliminarà en la següent iteració.

L'algorisme consisteix primerament en buscar per cada combinació el mínim de combinacions que s'elimininarien per cada combinació de fitxes de control que podem obtenir, i després obtenir el màxim d'entre aquests mínims.

Per tant el primer que s'ha de fer és calcular per cada combinació el mínim de combinacions que s'eliminarien per cada control. El mínim d'eliminades el calcularíem de la següent manera:

mínim d'eliminades = nombre de combinacions no provades - màxim d'encerts

Per no repetir l'operació contínuament, es busca el màxim d'encerts i un cop el tenim es procedeix a fer la resta.

Amb el fi d'aconseguir aquest màxim, nosaltres hem utilitzat la funció maxscore() per cadascuna de les combinacions que continuen estant a la llista de combinacions no provades (availableGuesses).

L'algoritme original fa una cerca exhaustiva i recorre per cada combinació, per cada possible combinació de fitxes de control, cadascuna de les combinacions. Per tant el cost seria de:

*combinacions no provades (availableGuesses) * controls * combinacions possibles(possibleCombinations)*

En el nostre cas hem optimitzar l'algorisme per a que, tot i que s'incrementi el cost en espai, disminueixi el cost en temps d'execució i només hagi de recòrrer:

*combinacions no provades * combinacions possibles*

Degut a que per dos combinacions només pot haver una única combinació de fitxes de control, per tal de no haver de recòrrer tots els controls hem creat un HashMap que té com a clau les fitxes de control i pren com a valor el nombre d'encerts per una combinació amb cadascuna de les combinacions. Cada cop que per dues combinacions es dóna un tipus de control, s'incrementa en 1, el valor per aquell control. Finalment recorrem tot el HashMap per trobar el màxim, i aquest serà el valor que buscàvem.

Un cop obtingut el nombre màxim d'encerts d'una combinació, ja es pot trobar quin serà el mínim nombre de combinacions eliminades fent la resta esmentada. Quan tenim el resultat, l'hem de comparar amb el mínim de combinacions eliminades més gran que teníem anteriorment; si el que hi havia era més petit, s'ha de substituir pel nou. En cas que siguin iguals, si l'anterior ja pertenyia a una combinació que també formava part de les possibles combinacions que encara poden ser guanyadores, no s'ha de fer res, ja que Knuth segueix la convenció de, en cas d'empat, agafar la combinació amb un valor numèric més petit.

Si en cas contrari, la combinació no es troba entre les possibles combinacions guanyadores i la nova si, es canviarà; sinó, seguint la convenció, es deixarà la que ja teníem.

Un cop comprovat el màxim d'encerts per cada combinació establirem com a pròxim intent, aquella combinació que hem obtingut amb màxima puntuació, la eliminarem de combinacions disponibles i l'anomenarem lastGuess, per a què al següent torn quan s'invoqui de nou la funció, ja tingui l'anterior intent.

Al pròxim intent de resoldre la combinació haurem de fer el mateix, començant per actualitzar la llista de possibles combinacions. Seguirem el mateix procés durant 4 torns i, al cinqué ja haurà resolt la combinació triada pel Maker.

3.3. Genetic Algorithm

L'algoritme genètic es basa en generar un conjunt de possibles resultats aleatoriament, i aquests, enfrentar-los respecte els altres intents que hem fet anteriorment, per obtenir una classificació (fitness). A continnuació, l'algoritme evoluciona la població mitjançant crossovers, permutacions, mutacions i inversions amb l'objectiu de millorar els resultats, i obtenir un codi factible.

Per procedir a la programació de l'algorisme, vam crear una classe anomenada FiveGuessComputer, que és extensió de la classe ComputerPlayer i que té els següents atributs:

POPULATION_SIZE → Constant que diu el tamany de la població que calcularem.

GENERATION_SIZE → Constant per fixar el numero de generacions.

FEASIBLE_CODES_MAX→Constant per determinar els feasable codes que crearem

ArrayList<ColorRow> gameGuesses → Per guardar els anteriors intents enviats a corretgir.
ArrayList<Integer> turnBlacks → Per guardar les anteriors correccions (negres).
ArrayList<Integer> turnWhites → Per guardar les anteriors correccions (blanques)
ArrayList<Integer> fitnes → per cada població[i], fitness[i] conte el fitness calculat de la població
ArrayList<ColorRow> feasibleCode → En aquesta varambe anirem afegint els feasible codes. Com que tenim per constant que el numero de feasibles sigui 1, nomes guardarà una COlorRow.
ArrayList<ColorRow> population → Per guardar la poblacio que anem generant.
Integer parentPos → Integer que ens diu una posicio de la poblacio per generar el crossover .

FUNCIONS PRINCIPALS

L'algoritme esta composat per la funció breakerguess() que es la que inicaialitza i executa totes les altres.

breakerGuess()

Calcular el seguent guess. Per fer-ho, s'inicialitza una població aleatòria, i es calcula l'aptitud per a aquesta població. Després d'això, genera una nova població utilitzant

crossover, mutació, inversió i permutació, fins que n'hi hagi prou codis factibles, i torna un d'aquests a l'atzar.

Veiem-ho amb més detall:

Primer de tot hem de comprovar que no sigui el primer intent, perqué en cas de que ho sigui no tindrem dades per calcular. En cas de que sigui el primer torn, generem un dels torns que tenim guardats amb breakerInitialGuess(pegs, colors).

En cas de que no sigui el primer torn, primer inicialitzem una població. Cada element de la població es generat de manera random. Posteriorment calculem el fitness per cada element de la població, i apliquem un quick sort per ordenar la població per fitness.

Un cop ja tenim la població inicialitzada ja podem començar a evolucionar-la. Sempre i quant no haguem trobat cap codi factible i no haguem superat el numero de cops que volem evolucionar la població, evolucionem la població mitjançant els mètodes que expliquem mes endavant i tornem a calcular el fitness de la nova població. en cas de que hi hagi algun codi factible haurem acabat. En cas contrari, tornarem a iterar. Per últim ens guardem l'intent a la variable gameGuess, i retornem l'intent.

calculateFitness()

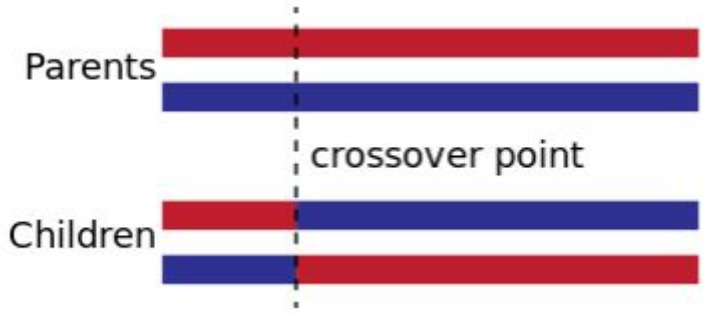
Per calcular el fitness d'una ColorRow el que fem és corregir-la com si els anteriors guess fossin la solució. Restem al resultat de nombre de negres obtingut el resultat de negres que realment va obtenir la combinació. Fem el mateix per les blanques, i al final ho sumem tot.

evolvePopulation(int pegs, int colors)

Evolucionem la població mitjançant crossover (de 1 i 2 punts), mutacions, inversions, i permutacions. Per generar la nova població, un de cada dos esta generat amb un crossover (amb un 50% de ser de 1 punt, i un 50% de ser de 2 punts).

Independentment per cada valor de la nova població, fem una mutació amb una probabilitat de 0'03, una permutació amb probabilitat de 0'03 i una inversió amb probabilitat de 0'02.
En cas de haver generat un codi duplicat, aquest el canviem per un codi random, i actualitzem la població.

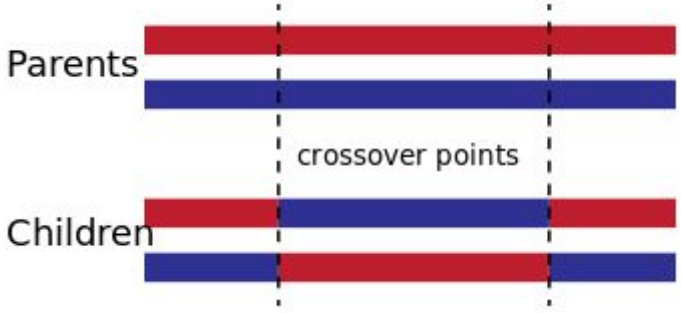
crossover1Point(ArrayList<ColorRow> newPopulation, int child1Pos, int child2Pos, int pegs)
L'objectiu es conseguir el següent resultat:



Per fer-ho, escollim la posició del separador de manera random, i apliquem els canvis de ColorRows.

crossover2Points(ArrayList<ColorRow> newPopulation, int child1Pos, int child2Pos, int pegs)

L'objectiu es conseguir el següent resultat:



Per fer-ho, escollim la posició dels separadors de manera random, i apliquem els canvis de ColorRows. En cas necessari invertim els separadors.

addToFeasibleCodes()

Un codi c és feasible si es donen els mateixos valors per a Xk i Yk per a tots els guess k que s'han jugat fins a aquell moment, si c fos el codi secret. Per tant, hem de comparar cada element de la població per tots els intents de guess generats anteriorment i mirar que compleixi la condició. En cas de que la compleixi, de que feasibleCodes no estigui ple, i de que no estigui ja afegit, afegim l'element de la població a feasibleCodes .

4. Criteris de la nostra aplicació

- Considerem que les fitxes negres són aquelles que indiquen que en la nostra combinació i la correcta, coincideix un color (un nombre en el nostre cas) en la mateixa posició.
- Per tant, les fitxes blanques indiquen que hi ha un color que es repeteix en ambdues combinacions però no està en la posició correcta.
- Només introduïm en el rànquing els usuaris que han guanyat algun cop jugant com a breaker.