Ubermind

Grup 43.2

Albert Bausili Fernández (albert.bausili)

Sergio Delgado Ampudia (sergio.delgado.ampudia)

David Molina Mesa (david.molina.m)

Noa Yu Ventura Vila (noa.yu.ventura)

1. Capa de Dades

Com normalment el cost de carregar i escriure informació de persistència depèn molt del sistema operatiu i/o del hardware del que disposa l'ordinador, assmirem que totes les funcions tenen cost constant.

1.1 Persistència Rànquings

Hem fet aquesta implementació per disminuir l'acoblament entre capes. PersistenciaRanquings conté les funcions per a guardar i recuperar les dades dels ranquings d'usuaris i de partides. Té 4 mètodes:

- **public static void guardarRanquingUsuaris(LinkedList<Pair<Float, Integer>> ranq, Dificultat dif):** En aquest mètode, entra com a paràmetre un objecte rànquing d'usuaris que volem guardar i la dificultat del rànquing. La funció guarda en un fitxer txt de la forma Usuaris_"dif".txt el rànquing d'usuaris, i si ja existeix el sobreescriu.
    - Cost: O(1) en tots els casos.
- **public static void guardarRanquingPartides(LinkedList<Pair<Integer, Integer>> ranq, Dificultat dif):** Aquest mètode guarda en un fitxer txt de la forma Partides_"dif".txt el rànquing de partides per la dificultat dif, i si ja existeix el sobreescriu.
    - Cost: O(1) en tots els casos.
- **public static RanquingUsuaris tornarRanquingUsuaris(Dificultat dif):** Podem utilitzar aquest mètode per recuperar del disc el rànquing d'usuaris de la dificultat dif**.**
    - Cost: O(1) en tots els casos.
- **public static RanquingPartides tornarRanquingPartides(Dificultat dif):** Podem utilitzar aquest mètode per recuperar del disc el rànquing de partides de la dificultat dif**.**
    - Cost: O(1) en tots els casos.
- **boolean comprovarSiRanquing (Boolean usuaris, Dificultat dif):** Funcio per a comprovar si existeix en disc algun dels ranquings en concret.
    - Cost: O(1) en tots els casos.
    - Explicació: Necessitem aquesta funció per evitar fer lectures a fitxers que no existeixin si encara no han sigut creats mai.

1.2 Persistència Usuaris

La classe PersistenciaUsuaris conté les funcions per a guardar i recuperar les dades dels usuaris, així com la llista d'aquests. Té 4 mètodes:

- **guardarUsuari(Pair<Integer, Quartet<String[], Integer, Boolean, Idioma>> usr, Pair<Integer[], Date> est, int[] rec, ArrayList<Integer> hist):** Guarda en un fitxer txt de la forma "uid".txt l'usuari, les estadístiques, els rècords i el seu historial, i si ja existeix el sobreescriu.
    - Cost: O(1) en tots els casos.
    - Explicació: Hem decidit guardar-ho tot junt, ja que, sempre que es ve a rescatar sempre es rescaten els quatre objectes alhora i així minimitzar les obertures.
- **public static void guardarNomUid(HashMap<String, Integer> nomUid):** Guarda en un fitxer amb el nom nomUid.txt la llista de noms relacionats amb els seus respectius uids, i si ja existeix el sobrescriu.
    - Cost: O(1) en tots els casos.
    - Explicació: Els guardem i llegim tots de memoria junts, ja que els necessitem tenir guardats en memòria per optimitzar les cerques.
- **public static Quartet<Usuari, Estadistiques, Records, Historial> tornarUsuari(int uid):** Llegeix del fitxer de l'usuari el seu objecte, el de les seves estadístiques, els seus records i el seu historial i ho torna en un tipus generic. Retorna un tipus generic del tipus Quartet que conté els objectes Usuari, Estadistiques, Records i Historial de l'usuari demanat.
    - Cost: O(1) en tots els casos.
- **public static HashMap<String, Integer> tornarNomUid():** Llegeix del fitxer de nomUid la llista de noms amb els seus respectius identificadors. Retorna un Hashmap amb els continguts de noms com a key i els seus identificadors com a value
    - Cost: O(1) en tots els casos.
- **boolean comprovarSiNomUid():** Funció per a comprovar si existeix en el disc l'arxiu nomUid o si encara no s'ha guardat mai.
    - Cost: O(1) en tots els casos.
    - Explicació: Necessitem aquesta funció per evitar fer lectures a fitxers que no existeixin si encara no han sigut creats mai.

1.3 Persistència Partida

Conté les funcions per a guardar i recuperar les dades de les partides i resultats del usuaris

- **void guardarPartida(Quartet<Integer, Dificultat, TipusPartida, Boolean> part, int[] tulr):** Guarda en un fitxer txt de la forma "uid".txt la partida, si ja existeix el sobreescriu
- **Pair<Partida, Tauler> tornarPartida(int pid):** Llegeix del fitxer de la partida l'objecte partida i tauler i els torna.
    - Cost: O(1) en tots els casos.
- **void guardarMaxPartides(int pid):** Guarda en un fitxer amb el nom maxpartides.txt l'identificador més gran fins el moment.
    - Cost: O(1) en tots els casos.

- **int tornarMaxPartides():** Retorna el identificador mes gran fins el moment.
    - Cost: O(1) en tots els casos.
- **boolean existeixMaxPartides():** Comprova si existeix el fitxer maxpartides.txt.
    - Cost: O(1) en tots els casos.
    - Explicació: Necessitem aquesta funció per evitar fer lectures a fitxers que no existeixin si encara no han sigut creats mai.
- **void guardarResultat(int rid, Quartet<TipusPartida, int[], LocalDateTime, Duration> res):** Guarda un nou resultat a la llista de resultats de persistencia.
    - Cost: O(1) en tots els casos.
    - Explicació: Hem decidit guardar tot en un sol fitxer indexat segons el identificador, ja que, quan es venen a recuperar sempre es recupera un set no buit de tamany no definit i normalment és més de 1 i per tant és més eficient per minimitzar el nombre de obertures/tancaments de fitxers.
- **ArrayList<Resultat> tornarResultats(Set<Integer> ids):** Torna una llista d'objectes resultat que corresponen a les ids introduïdes.
    - Cost: O(n) on n és el nombre d'ids.

2. Capa Domini

2.1 Enums

- **Dificultat:**
  - Enum que representa les possibles dificultats de les partides.
  - Pot prendre els valors: fàcil, intermig, dificil.
- **Idioma:**
  - Enum que representa els possibles idiomes en que es pot jugar.
  - Pot prendre els valors: castella, catala, angles, frances, alemany.
- **TipusPartida**
  - Enum que representa el tipus de partida que es pot jugar.
  - Pot prendre valors: ranked, entrenament.
- **Colors**
  - Enum que representa els possibles colors de les fitxes que hi ha a les partides.
  - Pot prendre els valors: vermell, blau, verd, groc, taronja, rosa, lila, blanc, negre (a més de buit, el qual fem servir per representar que en aquella posició del tauler encara no hi hem ficat cap fitxa).

2.2 Controlador Domini

Classe que controla totes les funcionalitats del sistema.

2.2.1 Funcions relacionades amb partida:

- **public Pair<Colors,Integer> demanarAjuda(int uid):** Funció que et dóna l'ajuda per a l'usuari. Et dóna la posició correcte d'una de les fitxes del codi del codemaker.
  - Cost: O(n); on n és el nombre de columnes del tauler.
- **public Colors[][] getTauler(int id):** Funció que retorna tot el tauler de la partida que té activa l'usuari amb ID igual a la que ens passen per paràmetre.
  - Cost: O(n*m); on n és el nombre de files, m és el nombre de columnes del tauler.

2.2.2 Funcions relacionades amb rànquing:

- **public ArrayList<LinkedList<Pair<Float, Integer>>> ranquingUsuaris(Dificultat dif)**: Obté els rànquings d'usuaris.
    - Cost: Cost: O(1) en tots els casos.
- **public ArrayList<LinkedList<Pair<Integer, Integer>>> ranquingPartides(Dificultat dif)**: Obté els rànquings de partides .
    - Cost: O(1) en tots els casos.
- **public void actualitzaRanquing(Dificultat dif, int uid, int puntuacio):** Quan un usuari acaba una partida s'han d'actualitzar els rànquings de partides i usuaris
    - Cost: O(1) en tots els casos.

2.2.3 Funcions relacionades amb usuari:

- **public int altaUsuari(String nomU, String contrasenya)**: Dona d'alta un nou usuari al sistema amb atributs nom i contrasenya.
    - Cost: O(1) amortitzat.
    - Excepcions: IncompatibleClassChangeError No s'han pogut modificar les llistes de la class.
- **public void baixaUsuari(int uid)**: Dona de baixa un usuari del sistema
    - Cost: O(1) en tots els casos.
    - Excepcions:Si l'usuari no existeix, es llença IllegalArgumentException
- **public int loginUsuari(String nom, String contrasenya)**: Intenta fer login amb les credencials introduïdes.
    - Cost: O(1) en tots els casos.
- **public void canviaUsuari(byte mode, int uid, String[] nou)**: Funció que canvia el nom de l'usuari, la contrasenya o els dos.
    - Cost: O(1) en tots els casos.
    - Excepcions: IllegalArgumentException Mode conté un valor no contemplat.
- **public boolean comprovarNom(String nom)**: Comprova si existeix un usuari amb el nom introduit.
    - Cost: O(1) en tots els casos.

- **public String[] dadesUsuari(int id) { return cu.getUsuari(id);** Funció per obtenir totes les dades d'un usuari. Retorna un array de quatre posicions amb els paràmetres en l'ordre uid, nom, contrassenya, tePartidaPausada
    - Cost: O(1) en tots els casos.
- **public Date dataCreacioCompteUsuari(int id):** Funció amb la que s'obté la data de creació del compte de l'usuari
    - Cost: O(1) en tots els casos.
- **public Resultat[] historialUsuari(int id):** Obté l'historial de l'usuari en forma d'array de resultats de les partides que ha jugat l'usuari. Retorna un array de resultats
    - Cost: O(1).
- **public int[] estadistiquesUsuari(int id)**: Funció que obté totes les estadístiques del usuari.Retorna un array
    - Cost: O(1).
- **public void actualitzaEstadistiques(int uid, int[] resultat)**: Quan un usuari acaba una partida, s'han d'actualitzar les seves estadístiques
    - Cost: O(1).
- **public void actualitzaHistorial(int uid, int rid)**: Quan un usuari termina una partida, s'ha d'actualitzar el seu historial
    - Cost: O(1).

2.2.4 Funcions per la gestió de partides:

- **public int reiniciarPartida(int uid)**: L'usuari inicia una nova partida. Aparellem l'usuari amb la partida que acaba de començar.
    - Cost: O(1).
- **public void pausarPartida(int id):** Funció per pausar una partida activa i guardar la informació corresponent.
    - Cost: O(1).
- **public void reprendrePartida(int id):** Funció per reanudar una partida pausada amb l'estat en que es trobava abans.
    - Cost: O(1).

- **public void iniciarPartida(int uid, Dificultat dif, TipusPartida tipusP, LocalDateTime tempsI, int nFiles, int nColumns, int nColors)**: Inicia una nova partida amb tots els atributs necessaris i afegeix una posició a la llista de partides actives
    - Cost: O(1).
- **public void abandonarPartida(int uid):** L'usuari pot abandonar la partida y aquesta no compta per el rànquing ni estadístiques
    - Cost: O(1).
- **public void acabarPartida(int id):** Acabem la partida fent: 1. Afegint el resultat al conjunt de resultats 2. Actualitzant el rànquing 3. Actualitzant l'historial 4. Actualitzant les estadístiques de l'usuari que ha acabat la partida Retorna l'ID de la partida acabada
    - Cost: O(1).

2.2.5 Funcions diverses

- **public Pair<Boolean,Colors[]> comprovarCodi(int id, Colors[] entered_code):** Comprovem si el codi del codebreaker que s'ha entrat és el correcte i informem si queden més files per seguir jugant una altra ronda. Retornem una parella que conté: 1. un boolean que indica si hi ha més files per jugar una altra ronda 2. un array de tantes posicions com columnes tingui el paràmetre entrat indicant el nombre de colors que s'han encertat.
    - Cost O(1).
- **public ArrayList<Colors[]> iniciaJugadaIA(int pid, int nColors, Colors[] colorsSeleccionats)**: Comprovem si el codi del codebreaker que s'ha entrat és el correcte i informem si queden més files per seguir jugant una altra ronda
    - Cost: O(n^2) cas pitjor i mig, O(n) cas millor; on n és el nombre de columnes del tauler.
- **public void comprovarRecords(int uid, boolean guanyat, int rondesUsuari, TipusPartida tipusP, Dificultat dif):** Comprovem quins records s'han completat en aquesta partida.
    - Cost: O(1).
- **public ArrayList<Pair<String,Boolean>> consultarRecords(int uid)**: Retorna tota la informació sobre els records de l'usuari
    - Cost: O(1).

2.3 Controlador Rànquing

Classe que controla la funcionalitat dels dos tipus de rànquings, de partides i usuaris, que s'utilitzen constantment cada cop que qualsevol usuari juga una partida, a més, cada usuari pot accedir a qualsevol dels rànquings dividits per dificultat.

2.3.1 Mètodes

- **public RanquingPartides[] consultarRanquingPartides():** Retornen els conjunts de rànquings de partides
    - Cost O(1).
- **public RanquingUsuaris[] consultarRanquingUsuaris():** Retornen els conjunts d'usuaris
    - Cost O(1).
- **public void afegirRanquingPartida(Dificultat dif, int punt, int uid):** Afegeix una nova posicio al rànquing de partides amb els dos paràmetres de entrada
    - Cost: O(1) en cas millor, O(log(n)) en cas pitjor; on n és el nombre d'usuaris que hi ha al rànquing.
- **public void afegirRanquingUsuari(Dificultat dif, float winrate, int uid):** Afegeix una nova posicio al rànquing d' usuaris amb els dos paràmetres d'entrada.
    - Cost: O(1) en cas millor, O(log(n)) en cas pitjor; on n és el nombre d'usuaris que hi ha al rànquing.
- **public void modificarRanquingUsuari(Dificultat dif, float winrate, int uid)**: Quan un usuari obté un nou winrate, s'elimina del rànquing d'usuaris i s'afegeix de nou amb el seu nou winrate
    - Cost: O(1) en cas millor, O(log(n)) en cas pitjor; on n és el nombre d'usuaris que hi ha al rànquing.

2.4 Classe Rànquing i les seves subclasses

Aquesta és una classe abstracta que fa referència als rànquings del sistema, hi ha dos possibles rànquings, de persones amb el seu winrate i identificador d'usuari, i rànquings de partides, amb les millors puntuacions i identificador dels usuaris que l'han fet. Cada rànquing es divideix en 3 diferents segons la dificultat de les partides jugades (fàcil, intermig i difícil). Rànquing te atribut dificultat i un getter.

2.4.1 Atributs

- **dificultat:** atribut de tipus Dificultat utilitzat per distingir la dificultat d'un rànquing.
- **Rpartides de la subclasse RanquingPartides:** atribut de tipus LinkedList<Pair<Integer, Integer>> on hi guardem parelles de puntuacio amb la ID de l'usuari que l'ha fet.
- **Rpartides de la subclasse RanquingUsuaris:** atributs de tipus inkedList<Pair<Float, Integer>> on hi guardem parelles de winrate amb la ID de l'usuari a qui correspon el winrate.

Les dues subclasses (Rànquing Partides i Rànquing Usuaris) contenen linkedlist de pairs per emmagatzemar els dos atributs que apareixen al rànquing. També tenen un mètode per poder afegir una nova posició al rànquing cada cop que es juga una partida. Utilitzem una cerca binària per fer-ho de forma eficient.

2.4.2 Mètodes

- **RanquingPartides(Dificultat dif)**: Crea un nou rànquing de partides amb la dificultat escollida
    - Cost: O(1) en tots els casos.
- **RanquingPartides(LinkedList<Pair<Integer, Integer>> rp, Dificultat dif)**: Creadora utilitzada en la persistencia.
    - Cost: O(1) en tots els casos.
- **RanquingUsuaris(Dificultat dif)**: Crea un nou rànquing d'usuaris amb la dificultat escollida
    - Cost: O(1) en tots els casos.
- **RanquingUsuaris(LinkedList<Pair<Float, Integer>> ru, Dificultat dif):** Creadora utilitzada en la persistencia
    - Cost: O(1) en tots els casos.
- **public void afegirRanquingPartides(int punt, int uid)**
- **public void afegirRanquingUsuaris(Float winrate, int uid)**
    - Cost: O(1) millor cas i O(log(n)) pitjor on n és el nombre de partides que hi ha al rànquing.

- **public void eliminarRanquingUsuaris(int uid):** Mètode per eliminar una posició del rànquing que elimina la posició d'un jugador al rànquing.
    - Cost: O(1) millor cas i O(n) pitjor on n és el nombre de partides que hi ha al rànquing.

2.5 Controlador Usuaris

Classe que controla les funcionalitats de les classes Usuaris i Estadistiques. Conté 3 estructures d'arrays per emmagatzemar tots els usuaris, historials i estadístiques.

2.5.1 Mètodes

- **public int afegirUsuari(String nomU, String contrassenyaU):** Crea un nou usuari amb el següent id i s'afegeix a la llista de paràmetres i fa lo propi amb les estadístiques d'aquest usuari.
    - Cost: O(1) cas normal, O(n) pitjor cas.
    - Excepcions: IncompatibleClassChangeError No s'han pogut modificar les llistes de la classe.
- **public void esborrarUsuari(int uid):** Marca a l'usuari amb identificador uid com esborrat del sistema.
    - Cost: O(1).
    - Excepcions: L'usuari amb id=uid no existeix.
- **private boolean existeixUsuari(int uid):** Comproba si un usuari ha estat esborrat. retorna un boolean per saber-ho.
    - Cost: O(1).
- **public boolean existeixNom(String nom)**: Comproba si existeix un usuari amb el nom introduit
    - Cost: O(1).
- **public int comprovarCredencials(String nom, String contrasenya)**: Comprova els credencials de l'usuari
    - Cost: O(1).
- **public String[] getUsuari (int uid**) :Introdueix les dades de l'usuari en un Array de strings i ho retorna.
    - Cost: O(1) en tots els casos.

- Excepcions: IllegalArgumentException si l'usuari no existeix i IndexOutOfBoundsException si l'índex es troba en una posició no contemplada.

- **public void canviaNom (int uid, String nouNom)**: Funció per canviar el nom d'usuari a un usuari.
  - Cost: O(1).
  - Excepcions: L'usuari amb id=uid no existeix.

- **public void canviaContrassenya (int uid, String novaContrasenya):** Funció per canviar la contrasenya d'un usuari.
  - Cost: O(1).
  - Excepcions: L'usuari amb id=uid no existeix.

- **public void afegirResultatHistorial (int uid, int rid):** Afegeix l'identificador del resultat que ha realitzat l'usuari uid en el seu historial.
  - Cost: O(1) en tots els casos.
  - Excepcions: L'usuari amb id=uid no existeix.

- **public void afegirEstadistiques(int id, int[] resultat):** Afegeix les estadístiques de resultat a les estadístiques de l'usuari identificat per id
  - Cost: O(1).
  - Excepcions: L'usuari amb id=uid no existeix.

- **public ArrayList<Pair<String, Boolean>> consultarRecords(int id):** Reenvia els rècords que l'usuari ha completat i no ha completat juntament amb les descripcions d'aquests, retorna una arrayList que conté els rècords amb el text i un boleà indicant si l'usuari els ha assolit o no.
  - Cost: O(1) en tots els casos.

- **public void comprovarRecords(int[] res):** Comprova si l'usuari ha realitzat algún record en la seva última partida.
  - Cost: O(1) en tots els casos.
- **void canviarIdiomaUsuari (int uid, Idioma nouIdioma)**: Canviem el nou idioma preferit de l'usuari per l'introduït.
  - Cost: O(1) en tots els casos.

- **void persistenciaUsuari ():** Funció auxiliar que envia a persistencia l'usuari, estadístiques, records i historial actuals.

2.6 Classe Usuari

Classe que fa referència als usuaris del sistema que utilitzaran el joc. Els usuaris tenen un nom d'usuari per poder identificar-los i amb el que es veuen reflectits als rànquings, un nom i una contrasenya amb els que s'han registrat. També tenen un int per saber si tenen una partida activa i un boolean per saber si ha estat donat de baixa.

2.6.1 Atributs

- **uid:** Identificador de l'usuari dins del joc, l'utilitzem per distinguir un usuari d'altres i es el nom que es mostrarà als rànquings.
- **nom:** Nom amb el que es registra un usuari al joc i amb el que fa login.
- **contrasenya:** Contrasenya amb la que l'usuari es registra al joc i fa login.
- **pidPartidaActiva:** int per saber si l'usuari te una partida activa (1) o no (0).
- **esborrat:** Boleà per saber si un usuari s'ha donat de baixa **.**
- **idiomaPreferit:** Atribut per saber quin dels 5 idiomas posibles ha escollit l'usuari peel joc**.**

2.6.2 Mètodes

- **Usuari(int id, String nomU, String contrassenyaU):** Mètode per crear un nou usuari amb les paràmetres demanats
    - Cost: O(1) en tots els casos.

- **Usuari(int id, String nomU, String contrasenyaU, int pid, boolean existeix, Idioma idioma):** Constructor per a la persistencia, es defineixen tots els paràmetres
    - Cost: O(1) en tots els casos.

2.7 Classe Estadístiques

Classe que conté les funcions i les estructures de dades per a mantenir i actualitzar les estadístiques de l'usuari. Com atributs, conté un identificador, i dos ints per cada dificultad (fàcil, intermèdia, difícil) per saber el nombre de partides guanyades i perdudes. També té dos ints per cada dificultat per saber el temps total jugat de cada usuari i la duració mitjana per partida, millor puntuació de cada usuari i puntuació mitjana per partida. Per últim, conté la data de creació del compte d'un usuari.

2.7.1 Atributs

- **eid:** Identificador de les estadístiques que es igual al de l'usuari amb aquelles estadístiques.
- **partidesGuanyadesFacil:** Atribut per saber el número de partides que ha guanyat un usuari en partides amb dificultat fàcil.
- **partidesGuanyadesIntermig:** Atribut per saber el número de partides que ha guanyat un usuari en partides amb dificultat intermitja.
- **partidesGuanyadesIDificil:** Atribut per saber el número de partides que ha guanyat un usuari en partides amb dificultat difícil.
- **partidesPerdudesFacil:** Atribut per saber el número de partides que ha perdut un usuari en partides amb dificultat fàcil.
- **partidesPerdudesIntermig:** Atribut per saber el número de partides que ha perdut un usuari en partides amb dificultat intermitja.
- **partidesPerdudesDificil:** Atribut per saber el número de partides que ha perdut un usuari en partides amb dificultat difícil.
- **tempsTotalJugatFacil:** Atribut per saber el número de temps en segons que un usuari ha jugat en partides amb dificultat fàcil.
- **tempsTotalJugatIbtermig:** Atribut per saber el número de temps en segons que un usuari ha jugat en partides amb dificultat intermitja.
- **tempsTotalJugatDificil:** Atribut per saber el número de temps en segons que un usuari ha jugat en partides amb dificultat difícil.
- **duracioPromigFacil:** Atribut per saber el temps promig en segons que un usuari ha passat en partides amb dificultat fàcil.
- **duracioPromigIntermig:** Atribut per saber el temps promig en segons que un usuari ha passat en partides amb dificultat intermitja.
- **duracioPromigDificil:** Atribut per saber el temps promig en segons que un usuari ha passat en partides amb dificultat difícil.
- **millorPuntuacioFacil:** Atribut per saber la millor puntuació d'un usuari en partides de dificultat fàcil, es va actualitzant segons l'usuari aconsegueix millor puntuació en partides.
- **millorPuntuacioIntermig:** Atribut per saber la millor puntuació d'un usuari en partides de dificultat intermitja, es va actualitzant segons l'usuari aconsegueix millor puntuació en partides.

- **millorPuntuacioDificil:** Atribut per saber la millor puntuació d'un usuari en partides de dificultat difícil, es va actualitzant segons l'usuari aconsegueix millor puntuació en partides.
- **puntuacioPromigFacil:** Atribut per saber la puntuació promig d'un usuari en partides de dificultat fàcil, es va actualitzant segons l'usuari juga partides.
- **puntuacioPromigIntermig:** Atribut per saber la puntuació promig d'un usuari en partides de dificultat intermitja, es va actualitzant segons l'usuari juga partides
- **puntuacioPromigDifícil:** Atribut per saber la puntuació promig d'un usuari en partides de dificultat difícil, es va actualitzant segons l'usuari juga partides.
- **creacioCompte:** Atribut per saber la data en la que un usuari es va donar d'alta al joc.

2.7.2 Mètodes

- **Estadistiques(int id):** Crea una nou set d'estadístiques per l'usuari amb identificador igual a id i ho inicialitza totes les variables.
    - Cost: O(1) en tots els casos.
- **Estadistiques(int id, Integer[] estats, Date data):** Creadora utilitzada en la persistencia, es defineixen tots els paràmetres..
    - Cost: O(1) en tots els casos.
- **void actualitzarEstadistiques (int[] resultat):** Actualitza les estadístiques del usuari afegint els seus últims resultats.
    - Cost: O(1) en tots els casos.
    - Excepcions: IncompatibleClassChangeError No s'han pogut canviar les variables de la classe i IllegalArgumentException Argument resultat[0] conté un valor invàlid.
- **float getRatiVictories (Dificultat diff):** Funció per calcular el winrate d'un usuari (partides guanyades/partides perdudes). Retorna aquest valor.
    - Cost: O(1) en tots els casos.
    - Excepcions: IllegalArgumentException L'argument diff conté un valor erroni.
- **boolean getPrimeraPartida (Dificultat diff)**: Comprova si la partida que acaba de jugar es la primera en aquella dificultat i retorna un boolean amb el resultat.
    - Cost: O(1) en tots els casos.

2.8 Classe Rècords

Classe que conté les funcions i les estructures de dades per a mantenir i actualitzar les els rècords de l'usuari. Com atributs, conté tots els rècords pero pode saber la quantitat de vegades que l'ha aconseguit. Els records són jugar els tipus de partides, guanyar els diferents tipus de partides, jugar 5 partides de cada tipus (1 rècord per tipus), guanyar 5 partides en cada tipus i guanyar cada tipus de partides en 5 rondes.

2.8.1 Atributs

- **recid:** Identificador dels récords de cada usuari.
- **records[]:** Vector on es guarden tots els records de cada usuari. Hi ha 32 posicions, cadascuna consta d'un int que es refereix a un record número de vegades que un usuari ha aconseguit completar un récord.

2.8.2 Mètodes

- **Records(int id):** Crea un nou set de records per l'usuari amb identificador igual a id i ho inicialitza totes les variables.
    - Cost: O(1) en tots els casos.
- **Records(int id, int[] rec)**: Crea un objecte record amb tot definit, s'utilitza a paersistencia.
    - Cost: O(1) en tots els casos.
- **void actualitzarRecords (int[] resultat):** Funció que actualitza els rècords de l'usuari comprovant si ha realitzat algún a la seva última partida.
    - Cost: O(1) en tots els casos.
- **ArrayList<Pair<String, Boolean>> retornaRecords ()**: Reenvia els records que l'usuari ha completat i no ha completat amb la seva descripció.
    - Cost: O(1) en tots els casos.

2.9 Classe Historial

Classe que conté tots els atributs i mètodes per gestionar els historials dels usuaris. Són una llista de partides amb resultats que ha jugat un jugador i es guarden perquè l'usuari pugui veure-ho.

2.9.1 Atributs

private final int hid: Identificador de l'historial.

2.9.2 Mètodes

- **Historial(int id):** Creadora de l'objecte historial.
    - Cost: O(1) en tots els casos.

- **Historial(int id, ArrayList<Integer> hist):** Crea un nou historial amb partides ja afegides (o no), s'utilitza a persistencia.
    - Cost: O(1) en tots els casos.
- **ArrayList<Integer> getPartides:** S'obté el llistat de partides que ha jugat un usuari.
    - Cost: O(1) en tots els casos.
- **void afegirResultat (int rid):** Afegim una partida amb identificador rid a l'historial.
    - Afegim una partida amb identificador rid a l'historial.

2.10 Controlador Partida

Classe que controla les funcionalitats de les classes partida, resultat i tauler. Conté 3 estructures de hashmaps per emmagatzemar partides, resultats i taulers. A més. conté una variable per guardar quina és la ID amb valor màxim que hi ha fins ara.

2.10.1 Atributs

- **Partida partidaActiva:** partida que está activa i li correspon l'ID que té guardat l'usuari.
- **max_id_partides:** Variable per guardar quina és la ID amb valor màxim que hi ha fins ara en el conjunt de partides.
- **Tauler taulerActiu:** tauler que está actiu i li correspon l'ID que té guardat l'usuari.

2.10.2 Mètodes

- **public int afegirPartida(Dificultat dif, TipusPartida tipusP, LocalDateTime tempsI, int nFiles, int nColumns, int nColors)**: Funció per afegir una nova partida a la llista de partides actives i crear el tauler. Retorna el ID de la partida.
    - Cost: O(n); on n és el nombre de columnes que té el tauler de la partida que volem afegir.
    - Excepcions: IncompatibleClassChangeError No s'han pogut modificar les llistes de la classe.
- **public void crearCodiIA(int id, int nColumns, int nColors)**: Funció per crear el codi de la IA quan fa de codemaker.
    - Cost: O(n) en el pitjor cas; on n és el nombre de columnes.
- **public void esborrarPartida(int id):** Funció per borra una partida i el seu tauler.
    - Cost: O(1) en tots els casos.

- Excepcions: IncompatibleClassChangeError No s'han pogut modificar les llistes de la classe.

- **public int afegirResultat(TipusPartida tipusP, byte rondesUsuari, byte rondesIA, LocalDateTime tempsI, Duration tempsP, int puntuacio):** Funció per afegir el resultat d'una partida acabada a la llista de resultats. Retorna l'ID del resultat creat.
  - Cost: O(1) en tots els casos.
  - Excepcions: IncompatibleClassChangeError No s'han pogut modificar les llistes de la classe.

- **public ArrayList<Pair<int[],LocalDateTime>> getResultatMultiple(Integer[] ids)**: Retorna un array de resultats amb els identificadors que entren pel parametre ids
  - Cost: O(1) en el millor cas, O(n) en el pitjor cas; on n és el nombre de resultats que volem obtenir (la mida del array).
- **public Pair<Boolean,Colors[]> comprovarCodi(int id, Colors[] entered_code)**: Funció amb la que es comprova si el codi del codebreaker que s'ha entrat és el correcte i informem si queden més files per seguir jugant una altra ronda. Retorna una parella que conté: 1. un boolean que indica si hi ha més files per jugar una altra ronda 2. un array de tantes posicions com columnes tingui el paràmetre entrat indicant quants colors s'han encertat.
  - Cost: O(n^2) cas pitjor i mig, O(n) cas millor; on n és el nombre de columnes del tauler.

- **public Pair<Colors,Integer> demanarAjuda(int id): public Pair<Colors,Integer> demanarAjuda(int id**): L'usuari demana ajuda en la partida. La funció fa: 1. Si l'usuari ha encertat la fitxa del codi de l'esquerra del tot, llavors li confirmem que l'ha encertat 2. Si no l'ha encertat, li indiquem de quin color és la fitxa de l'esquerra del tot
  - Cost: Cost: O(n); on n és el nombre de columnes del tauler.

- **public void incTempsPartida(int id):** Incrementem el temps de partida, la partida ja té els valors necessaris per calcular en quant s'ha d'incrementar.

- **public int calcularPuntuacio(int id):** Funció per calcular la puntuació total d'un usuari al acabar una partida.
  - Cost: O(1).

- **public ArrayList<Colors[]> obteJugadesIA(int id, int nColors, Colors[] colorsSeleccionatsUsuari)**: Recolecta tots els intents que fa la IA en la resolució del codi

- Cost:O(maxSteps * (colors^n + n^3)); on n és la longitud de la solució i colors és el nombre de colors que es poden utilitzar.

2.11 Classe Partida

Classe que fa referència a totes les partides del joc. Conté els següents atributs: Un int per identificar una partida, en enum per saber la dificultat i altre per saber el tipus de partida, un boolean per saber si s'ha utilitzat l'ajuda i altre per saber si es el torn de la IA o de l'usuari. Un byte per saber el nombre de rondes de la IA i altre per les de l'usuari. Finalment té la duració de la partida i el temps d'inici.

2.11.1 Atributs

- **pid:** Identificador d'una partida activa (que està en estat de pausa o està sent jugada).
- **dificultat:** variable per distingir la dificultat d'una partida (fàcil, intermitja o difícil).
- **tipuspartida:** Variable per distingir el tipus de partida (ranked o de prova).
- **usatAjuda:** Variable per saber si un usuari ha utilitzat l'ajuda durant la partida.
- rondesIA: Variable que comptabilitza les rondes que ha jugat la IA en una partida com a codebreaker.
- rondesUsuari: Variable que comptabilitza les rondes que ha jugat un jugador en una partida com a codebreaker.
- **tempsPartida:** Variable que conté el temps en segons que ha durat una partida al finalitzar.
- **iniciPartida:** Temps exacte en el qual s'ha iniciat o reprès una partida.
- **tornIA:** Indica si es el torn de la IA o de l'usuari.
- **tempsInici:** Conté en quin moment es va crear la partida.
- **puntuació:** Conté la puntuació final d'un usuari a una partida.

2.11.2 Mètodes

- **Partida(int id, Dificultat dif, TipusPartida tipusP, LocalDateTime tempsI)**: Crea una nova partida amb els paràmetres demanats, i els que no ens donen tenen el valor per defecte. Retorna la partida creada.
    - Cost: O(1).
- **void incRondesIA()**: Posem l'atribut rondesIA de la partida al valor que ens passen per paràmetre.

- Cost: O(1) en tots els casos.
- Excepcions: IncompatibleClassChangeError si no s'han pogut canviar les variables de la classe.

- **void incRondesUsuari()**: Posem l'atribut rondesUsuari de la partida al valor que ens passen per paràmetre.
  - Cost: O(1) en tots els casos.
  - Excepcions: IncompatibleClassChangeError si no s'han pogut canviar les variables de la classe.

- **void incTempsPartida(Duration sumTemps)**: Funció incrementa l'atribut tempsPartida de la partida en el valor que passen per paràmetre.
  - Cost: O(1).
  - Excepcions: IncompatibleClassChangeError No s'han pogut canviar les variables de la classe.

- **int calcularPuntuacio(byte nColumns, byte nColors):** Funció per calcular la puntuació final de la partida. Es retorna aquesta puntuació.
  - Cost: O(1).
  - Excepcions: IncompatibleClassChangeError No s'han pogut canviar les variables de la classe.

- **ArrayList<Colors[]> intentsMaquina(int nColors, Colors[] solucio):** Recol·lecta tots els intents que fa la IA en la resolució del codi.
  - Cost: O(maxSteps * (colors^n + n^3)); on n és la longitud de la solució i colors és el nombre de colors que és.

2.12 Classe Tauler

Classe que fa referència al tauler de la partida del joc. Conté com atributs: Un int per identificar-lo, un int per nombre de files, altre per nombre de columnes i altre per número de colors.

2.12.1 Atributs

- **tid:** identificador del tauler.
- **nfiles:** Número de files que conté el tauler, varia segons la dificultat de la partida.
- **ncolumnes:** Número de columnes que conté el tauler, varia segons la dificultat de la partida.

- **ncolors:** Número de colors posibles de les fitxes que conté el tauler, varia segons la dificultat de la partida.
- **filaactual:** Fila que se està adivinant del tauler en un moment determinat d'una partida
- **pistes:** Posibles pistes que es poden donar durant la partida**.**

2.12.2 Mètodes

- **Tauler(int _tid, int _nfiles, int _ncolumnes, int _ncolors):** Crea un nou tauler amb els paràmetres demanats.
    - Cost: O(1).

- **public Pair<Boolean,Colors[]> comprovarCodi(Colors[] codiProposat):** Funció que comprova el codi que proposa el Codebreaker amb el que proposa el Codemaster retornant si s'han arribat a l'última fila del tauler i la resposta del Codemaster al codi proposat. Retorna un pair amb un boolean que indica si s'ha arribat al màxim de files i la resposta de colors * (negre, blanc i buit) que donaria el Codemaster en format Array.
    - Cost: O(n^2) cas pitjor i mig, O(n) cas millor.

- **void setInitialCode(Colors[] codiSetejar):** Col·loca el codi a endevinar del Codemaker a la fila 0 del tauler (cada color a una cel·la).
    - Cost: O(n); on n és la longitud del codi.

- **public Pair<Colors, Integer> pistaFitxa()**: Obté una pista aleatoria que no s'hagi donat anteriorment.
    - Cost: O(n); on n és la longitud la solución.

2.13 Classe Resultat

Classe que fa referència als resultats de totes les partides acabades del joc. Conté com atributs: Un int per identificar cada resultat. Un enum de tipuspartida per saber el tipus, un int per saber el nombre de rondes de l'usuari i altre per saber las de la IA. El temps d'inici, la duració i la puntuació final.

2.13.1 Atributs

- **rid:** Identificador d'un resultat d'una partida acabada.
- **tipusPartida:** Tipus de partida que ha finalitzat (ranked o entrenament).

- **rondesUsuari:** Número de rondes que ha jugat l'usuari com a codebreaker a una partida finalitzada.
- **rondesIA:** Número de rondes que ha jugat la IA com a codebreaker a una partida finalitzada.
- **tempsInici:** Moment exacte en el que ha començat la partida.
- **temps:** Duració total de la partida en segons.
- **puntuació:** Puntuació final de l'usuari a la partida.

2.13.2 Mètodes

- **Resultat(int _rid, TipusPartida _tipusPartida, int _rondesUsuari, int _rondesIA, LocalDateTime _tempsInici, Duration _temps, int _puntuacio):** Crea un nou resultat amb els paràmetres demanats.
    - Cost: O(1).

2.14 Interfície Màquina

Interfície que contindrà les classes que implementen implementacions dels algorismes Genetic i Five Guess.

2.15 Classe MaquinaFiveGuess

Classe que implementa la Interfície Maquina amb l'algorisme Five Guess. Conte com atributs: Un int per identificar el nombre de colors són que presents a la solució i un altre int per limitar la quantitat de generacions de possibles solucions.

2.15.1 Atributs

- **colors:** nombre de colors que es poden fer servir en el codi solució.
- **maxSteps:** nombre màxim d'intents que té la IA per encertar.

2.15.2 Mètodes

- **public List<List<Integer>> solve(List<Integer> solution) throws Exception:** Crea una llista de combinacions proposades com solucions. Aquestes són les jugades que fa la maquina. La longitud d'aquesta depèn de quantes passes ha necessitat l'algorisme per trobar la solució, poden ser maxSteps passes si no la troba.
    - Cost: O(maxSteps * (colors^n + n^3)); on n es la longitud de la solució.

- **public List<List<Integer>>generarCandidatsInicials(int posicions, int colors):** Crea un llista de combinacions candidates inicials a partir del nombre de posicions que hi ha al tauler (les columnes) i el nombre de colors amb el que es juga.
    - Cost: O(colors^posicions), atès que es tracta a una combinatòria d'aquests.

- **public Map<String, Set<List<Integer>>> avaluarPossibles(List<List<Integer>> possibles, List<Integer> solucio):** Crea un diccionari d'avaluacions - conjunt de combinacions a partir d'una llista de combinacions possibles i la solució a la qual es vol arribar. Aquesta funció el que fa és agrupar totes les combinacions que tinguin mateixa avaluació.
    - Cost: O(n^m) essent n la longitud de la llista i m la longitud de les subllistes.

- **private String generaClauAvaluacio(List<Integer> solucio, List<Integer> possible):** Genera una clau d'avaluació donada una combinació possible comparant-se amb la solució a la qual es vol arribar. Aquesta clau ve a ser una etiqueta que evalua que propera és aquesta possible solució a la solució real.
    - Cost: O(n^2) cas pitjor i mig, O(n) cas millor (essent n la longitud de les llistes solució i possible).

- **public List<List<Integer>> ordenarPerDiferencia(List<List<Integer>> possibles, List<Integer> solucio):** Ordena una llista de combinacions que li passem per paràmetre per grau de diferència amb la combinació de la solució que també li passem per paràmetre. Aquest grau de diferència representa quant difereix un codi d'un altre.
    - Cost: O(n*k) cas pitjor, mig i millor (tenint en compte que calcularDiferencia() té cost O(n) essent n la mida de solució i k la quantitat de combinacions possibles).

- **private int calculaDiferencia(List<Integer> codi1, List<Integer> codi2):** Calcula el grau de diferencia entre dos codis donats.
    - Cost: O(n) cas pitjor i mig, O(1) cas millor (essent n la mida de les llistes codi1 i codi2).

- **public List<Integer> generarJugada(List<List<Integer>> possibles):** Genera una combinació la qual representarà una de les jugades de la màquina.
    - Cost: O(n) cas pitjor i mig, O(1) cas millor.

- **public List<List<Integer>> obtenirSubconjunt(List<List<Integer>> possibles):** Obté un subconjunt de la llista de combinacions possibles (el primer 20% d'aquesta)

- Cost: O(n) cas pitjor i mig, O(1) cas millor (essent n la longitud de la llista de possibles).

- **private List<List<Integer>> generarPossibles(List<List<Integer>> possibles, List<Integer> jugada, Map<String, Set<List<Integer>>> avaluacions):** Filtra la llista de possibles combinacions de tal manera que només retorna aquelles que tenen la mateixa clau d'avaluació que la jugada que li passem per paràmetre.
    - Cost: O(n^3) cas pitjor i mig, O(1) cas millor (essent n la longitud de la llista de possibles).

2.16 Classe MaquinaGenetic

Classe que implementa la Interfície Maquina amb l'algorisme genètic.

2.16.1 Atributs

- **maxGeneracions:** Quantes vegades entra al bucle y es igual al número de files del tauler.
- **nIndividusPoblacio:** El nombre d'individus que hi haurà en la població escollida.
- **nColors:** El nombre de colors que hi han en una partida i s'utilitzen.

2.16.2 Mètodes

- **MaquinaGenetic(int n_colors, int n_maxSteps):** Creadora de la implementació de MaquinaGenetic.
    - Cost: O(1) en tots els casos**.**

- **List<List<Integer>> solve(List<Integer> solution):** Creadora de l'implementació de la màquina genétic.
    - Cost: O(n) en el millor cas, O(maxGeneracions * m * n^3) en el pitjor dels casos; on n és el nombre d'individus de la població, maxGeneracions és els intents.

- **double maxFitness(Individu solucio):** Entra com a paràmetre la solució individu solució que l'algoritme ha de trobar y retorna el màxim fitness que es pot aconseguir (és igual al fitness de la solució, que és igual al nombre de gens que té l'individu solució)
    - Cost: O(1) en tots els casos.

2.17 Classe Població

Classe que representa un conjunt d'individus agafats aleatòriament per agafar el millor y fer funcionar l'algoritme genétic. El millor s'obté comparant quants gens de l'individu són iguals a l'individu solució. Si no encertem llavors hem de seguir proposant més individus potencialment bons. Per fer-ho fem evolucionar la població, i fa servir dues maneres: recombinació i mutació. Després, seguim generant diferents individus potencialment bons fins que encertem (ho sabem si els seu fitness és igual al nombre de gens de la solució) o fins que se'ns acabin els intents.

2.17.1 Mètodes

- **Poblacio(int tamany, boolean inicialitza, int numGens, int nColors):** Creadora de la població
    - Cost: O(n); on n és igual al tamany.
- **int numIndividus():** retorna el nombre d'individus de la població
    - Cost: O(1).
- **boolean existeixIndividu(Individu ind, int limit)**: Indica si ja existeix l'individu o no
    - Cost: O(n*m); on n és el nombre d'individus que hi ha a la població, i m és el nombre de gens que tenen aquests.
- **Individu individuMesApte(Individu solucio):** Inicialitza l'individu demanat a partir de la seva posició a l'individu que ens passen per paràmetre.
    - Cost: O(n) en tots els casos; on n és el nombre d'individus de la població.
- **valid(Individu solucio, Individu lastGen, Individu ind)**: Indica si l'individu és vàlid per a afegir a la població.
    - Cost: O(n); on n és el nombre de gens que té un individu.
- **Individu seleccionaIndividu(Individu solucio, int numTornejos, int numGens, int nColors):** Seleccionem individus aleatòriament per fer-los competir en el torneig i agafar-ne el millor.
    - Cost: O(n*m) en tots els casos; on n és el nombre d'individus que tindrà el torneig i m és el nombre d'individus que té la població actual.
- **Poblacio evolucionaPoblacio(Individu solucio, int nColors, Individu lastGen):** Fem evolucionar la població que tenim en l'actual generació per fer-ne una de millor a la següent generació.
    - Cost: O(m*n^2); on n és el nombre d'individus que té la població i m és el nombre de gens que tenen aquests.

2.18 Classe Individu

Classe que serveix per utilitzar l'algorisme genetic, agafa un individu i s'intenta agafar una solució comparant quants gens de l'individu són iguals a l'individu solució.

2.18.1 Atributs

- **numGensPerDefecte:** Nombre de gens que te un individu, es igual al nombre de colors de la solució
- **byte[]: gens:** Posició dels seus gens, que fan referencia a la posició de colors
- **fitness:** Nombre que mesura la semblança amb la solució, es calcula veient si la posició dels gens és igual als colors

2.18.2 Mètodes

- **Individu(int numGens):** Constructora d'individu.
    - Cost: O(1).
- **Individu(List<Integer> solucio):** Constructora exclusivament de l'individu solució.
    - Cost: O(n) en tots els casos; on n és el nombre de colors que tingui la solució.
- **numGens():** Obtenim el nombre de gens que té l'individu.
    - Cost: O(1) en tots els casos.
- **void inicialitzaIndividu(int n_colors):** Assignem valors random als gens de l'individu.
    - Cost: O(1) en tots els casos.
- **double fitnessIndividu(Individu solucio):** Calculem el fitness de l'individu a partir de la solució, la qual té fitness màxim.
    - Cost: O(n) en tots els casos; on n és el nombre de gens que té l'individu.

3. Capa presentació

3.1 Controlador Presentació

S'encarrega de la comunicació entre les diferents vistes. També s'encarrega de presentar a les vistes informació de les capes inferiors.

3.2 Vista Benvinguda

Es la primera vista que veu un usuari quan carrega el joc on se li dona la benvinguda y seguidament passa a la següent vista, la principal.

3.3 Vista Principal

Es la primera vista que veu l'usuari al entrar al joc, aquí és on podrà pasar a la vista de fer login si l'usuari ja té un compte creat o a la vista de registrarse si es un nou usuari.

3.4 Vista MenuPrincipal

Quan l'usuari ha fet login i és dins del joc aquesta és la vista que veu, el menú principal. Des d'aquí, l'usuari pot iniciar una partida o accedir al seu perfil per veure estadístiques, historial, rànquing… i les seves respectives vistes.

3.5 Vista ConfiguracioPartida

Quan l'usuari vol iniciar una partida li apareix aquesta vista on ha de decidir tant el tipus de partida (prova o ranked) i la dificultat d'aquesta (fàcil, intermèdia o difícil), una vegada escollit això ja passa a jugar la partida i la seva respectiva vista.

3.6 Vista CodeMaker

En aquesta vista l'usuari farà el rol de Code Maker seleccionant la combinacio de colors.

3.6 Vista CodeBreaker

En aquesta vista l'usuari farà el rol de Code Breaker. Podrà seleccionar les fitxes corresponents als colors per

3.7 Vista Ajuda

Aquesta vista permet a l'usuari veure unes indicacions generals de la mecànica de joc de Ubermind.

3.8 Vista Perfil i Subvistes

Aquesta vista es mostra quan l'usuari accedeix a través del menú principal, aquí, l'usuari podrà escollir múltiples opcions com veure les seves estadístiques amb la seva vista corresponent, el seu historial de partides on veurà per ordre les partides jugades i el seu resultat, i els records que l'usuari ha aconseguit jugant partides, quantes vegades, i els que li falten per aconseguir.

A més, desde aquesta vista de perfil podrà accedir a altres per fer funcions com canviar contrasenya, on l'usuari haurà d'escriure dos cops la seva nova contrasenya, canviar idioma, on l'usuari haurà d'escollir entre 1 dels 5 idiomes disponibles o canviar nom, on l'usuari escriurà dos cops el seu nou nom.

Com a última funció del perfil i nova subvista, l'usuari podrà esborrar el seu compte, li apareixerà un missatge preguntat si està segur i haurà de fer clic en un botó per confirmar aquesta acció.

3.9 VistaSeleccioRanquing i Ranquings

Des del menú principal, l'usuari podrà accedir a la vista de selecció dels rànquings, on l'usuari decidirà entre dues opcions, mostrar el rànquing d'usuaris o de partides fent clic en un dels 2 botons. Li apareixerà una nova vista, la de Rànquing usuaris en la que haurà de seleccionar una dificultat del rànquing entre les tres possibles (fàcil, intermèdia o difícil) i una vegada escollit amb un botó, veurà una llista ordenada amb noms d'usuaris i el seu winrate de major a menor, o rànquing partides, també aquí escollirà la dificultat i veurà amb una llista de noms d'usuaris i puntuacions de partides ordenades de major a menor.

4. Estructures de dades i algorismes

4.1 Algorisme FiveGuess

Aquest algorisme implementa una estratègia en el qual es generen totes les combinacions possibles i se les avalua per tal d'així saber quines d'elles estan més properes a la solució que volem arribar.

A més d'això se les agrupa segons la "nota" amb la qual han estat avaluades per així iterar sobre les millors combinacions possibles. Aquest algorisme utilitza estructures de dades com List (per representar els codis) , HashSets (per agrupar-los en conjunts) i HashMap (per agrupar aquests per "nota" d'avaluació). Quant a la complexitat de l'algorisme trobem que donat a les cerques e insercions que es realitzen el seu cost és de O(maxSteps * (colors^n + n^3)) on n és la longitud de la solució, colors és nombre de colors amb el qual es juga aquella partida i maxSteps el nombre màxim de jugades que fa la màquina.

4.2 Algorisme Genètic

Aquest algorisme implementa una estratègia en la qual s'intenta encertar el codi solució basant-nos en quant de bo és el nostre individu, és a dir, determinant el "fitness" de l'individu que proposem.

Primer es crea una població (conjunt d'individus) aleatòriament i s'agafa el millor, el qual s'obté comparant quants gens de l'individu són iguals a l'individu solució. Si no encertem llavors hem de continuar suggerint més individus potencialment bons. Per fer-ho fem evolucionar la població, i ho fem recombinant els individus de la població. Després, encara generem diferents individus potencialment bons fins que encertem (ho sabem si els seu fitness és igual al nombre de gens de la solució) o fins que se'ns acabin els intents.

L'algoritme fa servir un array de bytes per als gens dels individus i un d'individus per al conjunt d'individus d'una població, ja que sabem la seva mida des del primer moment i, per tant, no és necessari guardar-ho en una estructura dinàmica. El cost és O(n) en el millor cas, O(maxGeneracions * m * n^3) en el pitjor dels casos; on n és el nombre d'individus de la població, maxGeneracions és els intents màxims que té l'algoritme per endevinar el codi solució, i m és el nombre de columnes que té la solució.

4.3 Estructures de dades utilitzades

4.3.1 Controlador Rànquing

- **LinkedList<Pair<Float, Integer>> Ranquing Partides Facil/Intermig/Difícil**
    - Llistes utilitzades per controlar els 3 tipus diferents de rànquings d'usuaris. És una llista amb 3 posicions, ja que hi ha 3 rànquings, un per cada dificultat (fàcil, intermèdia o difícil). Hem considerat aquesta estructura, perquè considerem que una llista és el més bo per controlar els 3 tipus de rànquings, que són les linkedlist de pairs, perquè hem d'estar constantment buscant elements, afegint i eliminant.
    - **Alternatives:** Hashmap també ens seria útil, però no hem escollit aquesta opció per les operacions d'afegir i eliminar tan constants.
- **LinkedList<Pair<Integer, Integer>> Ranquings Partides Fácil/Intermig/Difícil**
    - Llistes utilitzades per controlar els 3 tipus diferents de rànquings de partides. És una llista amb 3 posicions, ja que hi ha 3 rànquings, un per cada dificultat (fàcil, intermèdia o difícil). Hem considerat aquesta estructura, perquè considerem que una llista és el més bo per controlar els 3 tipus de rànquings, que són les linkedlist de pairs, perquè hem d'estar constantment buscant elements de la llista, afegint i eliminant.
    - **Alternatives:** Hashmap també ens seria útil, però no hem escollit aquesta opció per les operacions d'afegir i eliminar tan constants.

4.3.2 Rànquing Partides

- **LinkedList<Pair<Integer, Integer>> Rpartides**
    - Llista utilitzada per representar un rànquing de partides que emmagatzema un pair amb la puntuació d'un usuari a una partida i l'identificador d'aquell usuari per, aquesta llista sempre estarà ordenada, ja que es controla l'ordre en afegir noves posiciones. Hem utilitzat aquesta estructura, perquè és útil per tenir una llista de pairs que s'ha d'ordenar.
    - Alternatives: Hashmap també ens sería útil, però no hem escollit aquesta opció per les operacions d'afegir i eliminar tan constants.

4.3.3 Rànquing Usuaris

- **LinkedList<Pair<Float, Integer>> Rusuaris**
    - Llista utilitzada per representar un rànquing d'usuaris que emmagatzema un pair amb el winrate (partidesguanyades/perdudes) d'un usuari i l'identificador d'aquell usuari per, aquesta llista sempre estarà ordenada, ja que es controla l'ordre a l'afegir noves posiciones. Hem utilitzat aquesta estructura, perquè és útil per tenir una llista de pairs que s'ha d'ordenar.
    - **Alternatives**: Hashmap també ens seria útil, però no hem escollit aquesta opció per les operacions d'afegir i eliminar tan constants.

4.3.4 Controlador usuaris

- **ArrayList<Usuari> cjt_usuaris**
    - Estructura de dades que emmagatzema la llista d'usuaris, els identificadors de l'ArrayList són els mateixos uid dels usuaris, quan es crea un usuari aquest s'afegeix al final de la llista, hem triat una llista, ja que és una manera eficient de tenir emmagatzemats i anar afegint tots els usuaris.

- **HashMap<String, Integer> cjt_nomUid**
    - Estructura de dades que relaciona els noms dels usuaris amb la seva uid. Hem triat un map ja permet emmagatzemar dos variables i que ens dona una manera eficient de buscar els strings en una llista llarga i no ordenada.

- **ArrayList<ArrayList<Integer>> cjt_historialUsuaris**
    - Estructura de dades que emmagatzema els identificadors dels resultats de les partides que han realitzat els usuaris, quan es crea un usuari aquest s'afegeix al final de la llista, hem triat una llista, ja que es pot afegir una posició de manera eficient.

- **ArrayList<Estadistiques> cjt_estadistiques**
    - Estructura de dades que emmagatzema la llista d'estadístiques, els identificadors de la ArrayList són els mateixos eid de les estadístiques, quan es crea una estadística aquesta s'afegeix al final de la llista. L'implementació de la llista d'una manera eficient de fer això.

- **ArrayList<Records> cjt_records**
    - Estrucutra de dades que enmagatzema la llista de records dels diferents usuaris, els identificadors de la ArrayList són els mateixos Recid dels records, els nous records s'afegeixen al final de la llista, hem triat una llista, ja que es pot afegir una posició de manera eficient.

4.3.5 Records

- **int[] records:**
    - Vector de ints amb 32 posicions que emmagatzema tots els records possibles per cada usuari. Cada posició representa un record i s'utilitza un int per portar un registre del número de vegades que un usuari ha completat aquest record. Per exemple la posició 13 del vector és guanyar una partida en dificultat difícil i en mode ranked. Si un usuari no ha aconseguit això, en la posició 13 del vector hi haurà un 0, si l'ha aconseguit 4 vegades hi haurà un 4. Considerem que un vector és el més adequat per aquesta utilització, ja que només hem d'incrementar i consultar.

    - **Alternativa**: List, hem pensat també en aquesta opció, perquè podria usar-se també sense cap problema, però un vector és millor, perquè sabem el nombre de posicions que hi haurà i els accessos són eficients.

4.3.6 Historial:

- **ArrayList<Integer> partides;**
    - Llista amb els identificadors de les partides que ha jugat un usuari, representa un historial de partides, i cada usuari té el seu. Considerem que un vector és el

més adequat per aquesta utilització, ja que només hem de fer operacions d'afegir i consultar.

4.3.7 Controlador Partida (CP)

En controlador partida tenim un hashmap:

- **HashMap<Integer,Resultat> cjt_resultats**

Cadascun guarda una clau (que és un Integer) i un objecte (que és una classe). La clau és l'identificador de l'objecte que estem guardant, i el segon paràmetre és l'objecte en sí, en aquest cas són els objectes Partida, Resultat i Tauler. Hem escollit HashMaps perquè és el mètode més eficient per a fer les operacions que es requereixen en la gestió de partides. El HashMap ens permet accedir i esborrar elements en cost O(1), i afegir-ne un en cost O(1) amortitzat, fent que l'aplicació sigui més eficient que no pas altres estructures de dades que discutirem a continuació.

4.3.7.1 Alternatives (CP)

- Un LinkedHashMap es podria haver fet servir, però en les operacions bàsiques és un pèl més lent que HashMap (que és la seva superclasse).

- Els Dictionary també són igual d'útils que els HashMaps, tot i que són considerats obsolets per java, i fer servir llibreries obsoletes és una gran bretxa de seguretat.

- HashTable és també una bona opció que podríem haver fet servir, ja que gairebé no hi ha diferències respecte al HashMap. Aquesta, a diferència del HashMap, permet sincronització, és a dir, permet multi-threading. Com en aquest projecte no ens demanen ni especifiquen que l'aplicació sigui multi-threaded, assumirem que només s'executa en un sol thread d'un sol core, de manera que l'execució és seqüencial i no hauria de sorgir cap error a causa d'això.

4.3.7.2 Conclusió (CP)

- Per al cjt_resultats, sabem que un resultat mai s'esborrarà, i que els elements es poden anar afegint de manera no ordenada. Això vol dir que estem en el mateix cas d'abans, no podem accedir als elements per la posició on es troben, sinó que necessitem tenir una clau amb l'identificador de l'objecte corresponent, per tant, per aquest cas podem aplicar la lògica següent: necessitem una estructura de dades que ens permeti accedir amb l'identificador de l'objecte, no per la posició on està guardada, que és el que ho diferencia de moltes altres estructures de dades com l'Array, ArrayList, HashSet, etc.

4.3.8 Població

private Individu[] individus: Estructura de dades que emmagatzema elements que són Individus, és una mostra aleatòries de colors que serveixen per resoldre una combinació mitjançant l'algorisme genètic. Considerem que un vector és el més adequat per aquesta utilització, ja que només hem de consultar.