Descripción de estructuras de datos y algoritmos

Tabla de contenidos

Five-guess algorithm 1
Algoritmo de obtención del feedback de un código 3
Cálculo del multiplicador de puntuación según la dificultad (custom) 4
Cálculo de puntuación CodeBreaker 5
Creación del código de CodeMaker 6
Genetic algorithm **7**
ParserPresentation **9**
ParserPersistencia **10**
Persistencia **10**
ViewBoard **11**

Five-guess algorithm

Cabecera: public List<List<Integer>> solve(List<Integer> solution)

El algoritmo Five Guess nos permite resolver las partidas de Mastermind que la máquina juega de code breaker. Este algoritmo consiste en crear un set con todas las posibles soluciones de la partida dependiendo de la longitud del código, si se permiten colores repetidos, dejar espacios vacíos o el número de colores, e ir reduciéndolo a medida que se obtienen los feedbacks.

Primeramente se obtiene un código que servirá como primer intento el cual se hace sin información y por lo tanto puede ser random, sin embargo existen combinaciones específicas que dan mejores resultados según la longitud del código y si se pueden repetir colores. Se obtiene un set de todas las posibilidades (el cuál al principio es bastante grande) y el feedback de primera guess. Una vez tenemos esa información habrá que ir eliminando los códigos del set y generando nuevas guesses.

Eliminar códigos del set consiste en comparar cada elemento del set con la guess actual, obtener el feedback (es decir la comparación de los dos códigos), y en caso de que ese feedback sea diferente del feedback de la guess actual, eliminarlo, de esta forma eliminamos todos los códigos imposibles. Una vez tenemos el set con las posibles soluciones, hay que encontrar una guess de entre todas las combinaciones, a poder ser de las que sean posibles (ya que de esta forma es posible ganar en el próximo turno), que en el peor de los casos, es decir en el caso en que el feedback recibido elimine menos códigos, sea la que más códigos elimine (la mejor opción en el peor escenario), por lo tanto contaremos el número de códigos (de los códigos posibles) que salva cada combinación para cada feedback y guardaremos los máximos (los que más salvan, es decir el peor escenario), y de entre todos los máximos, escogeremos las combinaciones que salven al mínimo número de códigos. Dentro de este grupo de códigos intentaremos escoger uno que sea uno de los códigos posibles si puede ser.

Método solve principal:

- List<Integer> guess: Representa al código de nuestra guess actual.
- List<List<Integer>> turns: Representa cada turno de la partida con cada una de las guesses.
- Pair<Integer,Integer> feedback: Representa el feedback obtenido por la guess actual, el primer número son los colores correctos en posiciones correctas y el segundo son los colores correctos en posiciones incorrectas.
- List<List<Integer>> combinations: Representa a todas las combinaciones de códigos compatibles con las opciones de la partida.
- List<List<Integer>> possibleCodes: Representa las combinaciones de códigos que todavía pueden ser la solución.

Otros métodos auxiliares:

- Map<Pair<Integer,Integer>, Integer> count: Representa por cada feedback possible, el número de códigos posibles que salvaría una combinación.

- Map<List<Integer>,Integer> scores: Representa a cada una de las combinaciones y el número máximo de códigos posibles que salva cada una.
- List<List<Integer>> possibleNextGuesses: Representa a la lista de códigos que en el peor caso eliminan a más códigos.

- Stack<List<Integer>> stack: Utilizada para generar todas las permutaciones posibles.

Algoritmo de obtención del feedback de un código

Cabecera: public Pair<Integer,Integer> compareSecretCode(Ball_color[] code)

Este método es utilizado para comparar un código enviado con el código secreto guardado en la clase Feedback para ver cuántos elementos están en la posición correcta y cuántos colores correctos están en la posición incorrecta. El método toma un array de objetos de tipo "Ball_color" como parámetro y devuelve un par de enteros que representan la cantidad de elementos que están en la posición correcta y la cantidad de colores en la posición incorrecta.

Para llevar a cabo esta tarea, se utilizan las siguientes estructuras de datos y algoritmos:

El método utiliza un array booleano para llevar un registro de qué elementos ya han sido revisados. En este caso, se utilizan dos arrays booleanos: uno para el código enviado y otro para el código secreto.

Se utiliza un for para comparar cada elemento del código enviado con el código secreto. Si un elemento coincide en posición y valor, se incrementa un contador y se marca en los arrays booleanos correspondientes que ese elemento ya ha sido revisado.

Si aún no todos los elementos han sido encontrados en la posición correcta, se utiliza otro for para buscar elementos que estén en la posición incorrecta. Para hacer esto, se recorre el array enviado y se busca en el array del código secreto cualquier elemento que no haya sido revisado y que coincida en valor. Si se encuentra un elemento que coincide, se incrementa otro contador y se marca en los array booleanos correspondientes que esos elementos ya han sido revisados.

Gracias a este método, podemos darle feedback al jugador sobre el código propuesto y así poder continuar con el flow de la partida.

Cálculo del multiplicador de puntuación según la dificultad (custom)

Cabecera: setMulCus()

Este método pertenece a la clase DifficultyCustom extendida de la clase Difficulty y se utiliza para calcular el "multiplier" de la dificultad personalizada en función de los diferentes parámetros establecidos para el juego.

El multiplicador de la puntuación del jugador se utiliza para valorar más una partida compleja que una con una dificultad menor, de esta forma, podemos aumentar la puntuación del jugador y así verse reflejado en los rankings finales.

La fórmula utilizada para el cálculo del multiplicador se basa en los siguientes parámetros:

repeated_colors: indica si se permiten colores repetidos en el código secreto.
number_positions: indica la cantidad de posiciones en el código secreto.
empty_colors: indica si se permiten colores vacíos en el código secreto.
max_turns: indica la cantidad máxima de turnos permitidos para adivinar el código secreto.

Antes de calcular el multiplicador, se ha tenido que seleccionar la dificultad custom con los parámetros documentados establecidos. A continuación, explicamos a qué factores les damos más importancia proporcionando valores más altos:

Si se permiten colores repetidos en el código secreto, se suma 4.0 al multiplicador.
Si la cantidad de posiciones en el código secreto es mayor o igual a 4, se suma 2^(number_positions-4) - 1 al multiplicador; en caso contrario, se suma 2^(number_positions-4).
Si se permiten colores vacíos en el código secreto, se suma 2.0 al multiplicador.
Finalmente, si la cantidad máxima de turnos es mayor o igual a 8, se suma 2^((max_turns - 8) * -0.55) al multiplicador; en caso contrario, se suma 2^((max_turns - 8) * -0.55) - 1 al multiplicador.

Cuando GameStats vaya a calcular la puntuación final el multiplicador jugará un papel importante ya que multiplicará los parámetros más relativamente importantes y así crear un mayor impacto. Este cálculo está documentado en la siguiente página.

Cálculo de puntuación CodeBreaker

Cabecera: calculateScoreCB(int maxTurns, double multiplier): int

Este método se encuentra en la clase GameStats y se utiliza para calcular la puntuación final de un juego en el que el jugador tiene el rol de "codebreaker". La puntuación final se basa en la cantidad de turnos, tiempo transcurrido y pistas utilizadas durante el juego.

La fórmula utilizada para calcular la puntuación es la siguiente:

score = (int) (((1000*__maxTurns__ - 500*__this.getNumTurn()__ + 1) * __multiplier__) / (__getElapsedTime()__ + 1 + __getNumHintsUsed()__))

Esta fórmula tiene en cuenta tres factores importantes: el número de turnos, el tiempo transcurrido y las pistas utilizadas. Como la intención del codebreaker es adivinar el código secreto en el menor número de turnos posible, y con la menor cantidad de pistas y tiempo transcurrido se le premiará al jugador cuanto menos tiempo haya pasado, menos pistas se hayan utilizado y menos turnos se hayan necesitado para adivinar el código. Así pues, teniendo esto en cuenta, mayor será la puntuación final del jugador.

La fórmula utiliza una combinación de estos factores, con un peso mayor en los turnos y el tiempo transcurrido, y un peso menor, pero significativo, en las pistas utilizadas. Esto se refleja en el numerador de la fórmula, donde se multiplican los turnos restantes por 1000 y se le suma 1 (para evitar divisiones por cero), y luego se multiplica por el multiplicador de la dificultad. Este multiplicador lo recibe de la clase Dificultad, el cual se obtiene a partir de las opciones de la partida. Cuanta más dificultad, más se le premia al jugador. En el denominador de la fórmula incluimos el tiempo transcurrido y las pistas utilizadas ya que queremos que acaben reduciendo la puntuación a medida que transcurre la partida.

Cabe destacar que es una fórmula creada por nosotros pero se basa en, básicamente, premiar el buen rendimiento del jugador.

Es importante destacar que esta fórmula se utiliza solo para calcular la puntuación final de los jugadores que tienen el rol de "codebreaker". Para los jugadores con el rol de "codemaker", se utiliza una fórmula diferente (solamente el número de turnos que ha tardado la máquina en resolver el código secreto), ya que la forma de jugar es diferente y no sería justo comparar a ambos jugadores con la misma fórmula.

Creación del código de CodeMaker

Cabecera: solve (List<Integer> solution): List<<List<Interger>>

Este algoritmo implementado con la interface Computer, consiste en crear un código para la partida cuando la máquina juega de CodeMaker. El algoritmo consiste en crear un código con 8 números si se permiten espacios vacíos, o 7 si no se permiten. La creación del código es random, sin embargo se comprueba que no se contengan colores repetidos en el caso de que no se permita repetir colores.

- List<Integer> randomCode: Representa el código random que se crea.
- List<List<Integer>> returnValue: Representa a la lista de un elemento (el código creado) que se retorna debido a la implementación de Computer.

Genetic algorithm

En esta clase está implementado el algoritmo genético que se encargará de solucionar el código secreto propuesto por el jugador codemaker. Este algoritmo de computer solo se llamará cuando el el jugador real escoge ser codemaker y escoge la dificultad custom seleccionando el algoritmo genético.

Su implementación consiste en un bucle donde intentará resolver el código turno por turno, lo cual acabará devolviendo todo el tablero final hasta llegar a la solución deseada o los turnos máximos propuestos.

El primer guess está hardcodeado dependiendo de la longitud del código, y a partir de ese punto, empezará a generar una población de posibles soluciones.

Todo el workflow del algoritmo está en el método solve(). Con el primer guess dado, comprobamos que no sea el código y secreto, y si no es el caso seguimos. A partir de ahí, cogemos esa población, la ordenamos según su fitness (aptitud) y la evolucionamos. De esta forma conseguimos la siguiente población con las siguientes posibles soluciones mejoradas. Reordenamos la población según su fitness, y la mejor de todas será el próximo guess (y si ya ha sido usado, se mira siempre el siguiente de la lista). Y así hasta llegar al final de turnos máximos o a la solución deseada.

Para calcular el fitness, lo que hace el algoritmo es conseguir el feedback del código que se desea calcular respecto a todos los feedback guesses del historial del juego. Así pues, se va iterando por todos los turnos y se le da un valor de aptitud según los pines que vaya obteniendo en la comparación de códigos. A este valor, lo que hacemos es darle un valor número a cada pin rojo o blanco, el cual el rojo tiene más peso (el doble) ya que nos interesa más.

Y para evolucionar la población, primero guardamos los dos mejores guesses de la población anterior para que sean los padres. Así se guarda la “élite” por poblaciones. Y a partir de ellas, les hacemos un crossover y generamos algunas posibles soluciones que puedan salir con las opciones dadas de dificultad. A partir de ese punto, empezamos con las mutaciones.La mutación consiste en cambiar random uno de sus colores o cambiar colores de dos posiciones del código.

De esta forma, calculando el fitness respecto a poblaciones anteriores y evolucionando las mejores opciones, el algoritmo acaba aprendiendo y llegando a qué código es la solución.

El problema que encontramos en este algoritmo es que, en nuestro juego, dejamos mucha personalización, y pues podemos hacer partidas de códigos de hasta 7 colores (8 más vació), dejamos colores repetidos, y pues hay tantas posibilidades que con un máximo de 10 turnos que dejamos, lo más probable es que no llegue ya que necesitaría más turnos.

- Métodos principales:
    - evolvePopulation(List<List<Integer>> newPopulation): conjunto de métodos ara evolucionar la población pasada por parámetro
    - List<Integer> permutation_crossover(List<Integer> code1, List<Integer>

code2): crea un código a paritr del crossover de dos códigos. Si consideramos que el tamaño promedio de las listas de entrada es n y lengthCode es una constante, entonces el tiempo de complejidad de este método sería aproximadamente O(n).

- inversion(List<List<Integer>> newPopulation,List<Integer> code1): método de mutación donde cambiamos la posición de dos colores del código pasado. El tiempo de complejidad de este método es constante, es decir, O(1). El motivo es que el número de operaciones realizadas no depende del tamaño de las listas o de ninguna otra variable, excepto lengthCode, que se asume como una constante.
- mutate(List<List<Integer>> newPopulation,List<Integer> code1): cambiamos un color de manera random del código pasado. Aquí su complejidad depende de la longitud del código, si asumimos n, O(n).
- double calculateSingleFitness(List<Integer> code): método para calcular el fitness de un código donde comparamos con el historial de partida. El tiempo de complejidad de este método es O(n), donde n es la longitud de prev_feedback.
- List<List<Integer>> sortPopulationByFitness(List<List<Integer>> codes): ordenamos la poblacion pasado según los fitness calculados anteriormente mediante bubble sort por comodidad y por su complejidad buena. Podríamos haber usado otros algoritmos de sorting, pero este tiene buenos resultado en complejidad. El bubble sort es $O(n^2)$ en el peor de los casos debido al doble for. Aunque O(n) en un avg case.

ParserPresentation

Esta clase existe para poder enviarle estructuras de datos que la capa de presentación pueda entender.
Básicamente, como en nuestra capa de dominio usamos la clase Pair implementada por nosotros, este parser se dedica a pasarlo todo a map.entry ya que sería su clase alternativa y ya dada por java de forma default. Además de pasar cualquier dato Ball_color a un string donde la presentación pueda entenderlo y tratarlo.

- Métodos principales:
    - List<Map.Entry<String, Integer>> convertToEntryList(ArrayList<Pair<String, Integer>> pairList)
    - Map<String, Map.Entry<String, Integer>> convertToEntryMap(Map<String, Pair<String, Integer>> pairMap)
    - String[][] convertBoardToString(Ball_color[][] board)
    - List<Map.Entry<Integer, Integer>> convertToEntryList(Pair<Integer, Integer>[] pairArray)

Todos estos métodos dependen del tamaño de la estructura pasado por parámetro, lo que si asumimos com n, tienen una complejidad de O(n).

ParserPersistencia

En esta clase, como ya hemos documentado, nos sirve para cambiar las estructuras de datos de diferentes elementos para que la capa de persistencia pueda entenderlos y tratarlos. Como el uso de Ball_color o Pair, que son clases propias, se les da una alternativa que pueda entender. De un lado a otro.

- Métodos principales:
    - List<String> convertMapToList(Map<String, Pair<String, Integer>> map): convierte el mapa de pair a lista de strings para presentacion.
    - Map<String, Pair<String, Integer>> convertListToMap(List<String> list): convierte la lista de strings en el mismo mapa que se usa en dominio.
    - List<String> combineRankings(ArrayList<Pair<String, Integer>> rankingCB, ArrayList<Pair<String, Integer>> rankingCM): coge los dos rankings, los junta y los pasa a una lista de string para que persistencia lo entienda.
    - Pair<ArrayList<Pair<String, Integer>>, ArrayList<Pair<String, Integer>>> splitRankings(List<String> combinedList): transforma esa lista de strings en los dos rankings con las estructuras que dominio usa.

Todos los métodos dependen del tamaño que tiene la estructura pasada por parámetro, si asumimos un tamaño de n, la complejidad es de O(n).

Para poder separar bien los dos rankings, hemos metido palabras clave en la lista de strings para saber cada elemento a donde pertenece.

Persistencia

La capa de persistencia, tal y como describe su nombre tiene la función de persistir los datos de la aplicación. En nuestro caso existen tres tipos de datos que hemos separado en diferentes archivos: los datos relacionados con los ránkings del juego, los que están relacionados con los récords, y finalmente la partida. Para ello se utilizan tres clases que manejan cada uno de los archivos.

Las tres clases tienen en común las funciones de load (cargar) y save (guardar):

- Métodos principales:
    - List<String> loadX(): Deserializa los datos del archivo con el que se trabaja y los retorna en formato de List<String>, donde cada elemento está serializado está separado por comas.
    - save(X): Serializa los datos pasados en la List<String> X de forma que cada elemento de la lista se escribe en el archivo separado por comas.

*X es el tipo de datos

Hemos decidido tratar los datos que recibe la persistencia como List<String> con tal de poder desacoplarlos de la capa de dominio y de esta forma que la persistencia adquiera cierta independencia a los cambios del dominio, o a la forma de serializar los datos.

En cuanto a la estructura de datos escogida, la List<String> tiene los contras de no ser tan eficiente como un Map, pero con tal de poder controlar los elementos pasados, y saber siempre el orden de estos, hemos decidido utilizar una List<String>.

Finalmente en la forma de serializar, hemos decidido utilizar archivos csv, ya que hemos creído que la capa de persistencia de la aplicación no requiere mucha complejidad, y como además no requerimos serializar clases enteras no creemos que sea necesario el uso de json u otros.

ViewBoard

En esta clase está implementado el algoritmo para la generación del tablero modularmente. Este algoritmo de generación sólo se llamará cuando se conozcan los parámetros con los que se va a jugar en la partida, es decir, cuando se sepa el rol que va a tener el jugador, la longitud del código a resolver y la cantidad de turnos a jugar.

Su implementación consiste en dos bucles anidados que recorren una matriz previamente declarada de tamaño maxTurns+1 * maxBalls, en la que, dependiendo de la posición donde se está iterando (y en ciertos casos dependiendo también del rol del jugador), se le asignará una imagen u otra. El algoritmo funciona de la siguiente manera:

- Si estamos en la primera esquina de la matriz (posición 0,0), se asigna la imagen de la esquina superior izquierda (0,0.png).
- Para cada posición de la matriz en la primera fila que no sea el inicio o el final, se asigna la imagen superior central (0,X.png).
- Para la última posición de la primera fila, se asigna la imagen de la esquina superior derecha (0,end.png).
- Si estamos en la tercera esquina de la matriz (posición end,0), se asigna la imagen de la esquina inferior izquierda (end,0.png).
- Para cada posición de la matriz en la última fila que no sea el inicio o el final, se asigna la imagen inferior central (end,X.png).
- Para la última posición de la última fila, se asigna la imagen de la esquina inferior derecha (0,end.png).
- Si estamos en la primera posición de una fila no inicial ni final de la matriz (posición 1…end-1,0), se asigna la imagen central izquierda (X,0.png).
- Si estamos en la última posición de una fila no inicial ni final de la matriz (posición 1…end-1,end), se asigna la imagen central derecha (X,end.png).
- Para la cualquier posición que no se corresponda con ninguna de las anteriores, se asigna la imagen central (X,X.png).

Esta resolución genera un problema, como cada imagen tiene unas dimensiones diferentes, las imágenes no se pueden ubicar correctamente de una forma sencilla.

Esto se resuelve accediendo a los atributos de la posición previamente gestionada. Es decir, para la primera posición de la matriz, sabiendo la longitud del código, la cantidad de turnos y las dimensiones de las imágenes que se van a utilizar, se calcula el punto origen donde tiene que colocarse la primera imagen. Una vez esta se ha colocado, el resto de imágenes solamente tienen que acceder a una casilla previa contigua por filas o columnas, es decir, para una matriz(i,j) cualquier casilla con j > 0, deberá acceder a la casilla j-1, conseguir su posición origen, sumarle el tamaño en horizontal de la imagen en j-1 y utilizar este resultado como posición origen para su casilla actual; para cualquier casilla con i > 0 y j = 0, deberá acceder a la casilla i-1 y j=0, conseguir su posición origen, sumarle el tamaño en vertical de la imagen en i-1 y utilizar este resultado como posición origen para su casilla actual.
De esta manera se puede generar cualquier tablero, ya que la primera posición se calcula automáticamente dependiendo de las dimensiones totales que vaya a ocupar el tablero y el resto se calculan en función de la anterior.

Adicionalmente, al mismo tiempo que se genera el tablero, se genera una matriz de coordenadas que se utilizará más adelante para colocar individualmente cada código en la matriz. Estas coordenadas se generan de forma similar al tablero. Como la primera posición siempre irá en la misma coordenada respecto a la coordenada origen de la matriz de del tablero anterior, solamente será necesario acceder a la matriz del tablero, conseguir la coordenada inicial de la posición 0,0 y sumarle un offset concreto en horizontal y en vertical. Una vez que tenemos esta coordenada, al tener el resto de posiciones de los códigos a la misma distancia, simplemente hay que sumarle un offset en horizontal para cualquier posición de la matriz que aumente en j y sumarle un offset en vertical para cualquier posición de la matriz que aumente en i.

A parte del código que el jugador vaya construyendo, también será necesario colocar el feedback en el tablero. Para esto se utilizan unos “tiles” que cambian dependiendo de la longitud del código. Para colocar estos se hace de forma similar al resto de cálculos explicados previamente, se calcula la posición del primero en referencia a la primera posición de la matriz del tablero si el jugador tiene el rol de CodeBreaker, o en referencia a la última posición de la matriz del tablero si el jugador tiene el rol de CodeMaker; una vez calculado esto, para el resto de posiciones, que irá en función de la cantidad de turnos, se le suma un offset en vertical ya que están a la misma distancia.

Una vez estos “tiles” del feedback están colocados correctamente, se calcula y se almacena la posición donde irá cada bola del feedback. Para que esto pueda ser modular y ajustable a la longitud del código, se calcula respecto a cada “tile” del feedback.

- Métodos principales:
    - init_board(): Es la función principal de la clase, se encarga de almacenar las imagenes necesarias en variables locales, calcular el tamaño que tendrá el tablero y ejecutar el algoritmo para la generación del tablero y obtención de coordenadas para los posteriores códigos.
    - calculate_feedback_coords(): Calcula y coloca los “tiles” del feedback en la posición del tablero.
    - calculate_feedback_code_coords(): Calcula las posiciones para colocar el feedback.

En esta clase la función con un coste más elevado es init_board(), ya que depende de la cantidad de colores y de la cantidad de turnos, por lo que siendo maxTurns = n y maxBalls = m, tiene un coste de Θ((n+1)*m), el resto de funciones tienen un coste lineal respecto a maxTurns excepto calculate_number_coords() que tiene un coste lineal respecto maxBalls.

Estos algoritmos utilizan estructuras de datos en las que se puede realizar un acceso aleatorio y se pueden recorrer secuencialmente, es decir, matrices y arrays, ya que al permitir un acceso secuencial facilitan la generación del tablero y del feedback y al permitir un acceso aleatorio facilita, por ejemplo, la colocación de colores en otras partes del programa que lo necesiten.