MASTERMIND

Breve descripción de las estructuras de datos y algoritmos

Projectes de Programació: Quadrimestre Primavera 2022-23

Grup 42.5:
Elsa Boix Solina
Joel Macías Rojas
Ricard Medina Amado
Marcel Sánchez Roca

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Breve descripción de las estructuras de datos y algoritmos

FIVE GUESS ALGORITHM

Explicación de las estructuras de datos utilizadas

1. **possibleCodes**: Es una lista de combinaciones representadas por enteros que representa las combinaciones que aún no han sido descartadas.
2. **totalcombinacionesPosibles**: Es una lista de todas las posibles combinaciones que pueden existir en la partida. Es generada a partir del nivel de dificultad del juego, representado por la clase NivelDificultad, que determina el número de columnas y repeticiones permitidas en la combinación.
3. **solucionesEnviadas**: Es una lista de listas de enteros que representa las combinaciones de colores que han sido enviadas por el algoritmo en turnos anteriores del juego. Se utiliza para llevar un registro de las combinaciones enviadas y devolverlas al finalizar el turno.
4. **enviosCandidatos**: Es una lista de combinaciones que representa las posibles combinaciones de colores que el algoritmo considera enviar en el próximo turno del juego. Se genera a partir de las combinaciones de colores en **totalcombinacionesPosibles** que tienen la puntuación más baja basándonos en las respuestas recibidas.
5. **envioActual**: Es una lista de enteros que representa la combinación de colores que el algoritmo enviará en el turno actual del juego.
6. **solucion**: Es una lista de enteros que representa la combinación de colores secreta que el algoritmo trata de adivinar.

Explicación del algoritmo

El algoritmo utiliza las estructuras de datos mencionadas para generar nuevas combinaciones a enviar, evaluar las posibles soluciones basándonos en las

respuestas recibidas y decidir qué combinación enviar en cada turno, con el objetivo de adivinar la combinación secreta en el menor número de intentos posible. A continuación se describe su funcionamiento paso por paso:

Se inicializa el número de ronda (ronda) en 1 y se guarda el nivel de dificultad del juego (representado por el objeto NivelDificultad) en la variable nivel.

Se implementan tres métodos privados: getMinScore, getMaxScore y generaNuevoEnvio, que son utilizados para calcular la puntuación mínima, máxima y generar la nueva combinación de código a ser enviada en el siguiente turno, respectivamente. Además se inicializa **totalcombinacionesPosibles** mediante backtracking obteniendo así todas las combinaciones posibles. **PossibleCodes** en este momento tiene el mismo contenido.

En primer lugar, se realiza un envío predeterminado para obtener el primer *feedback,* comparando el ***envioActual*** con la *solución*. Si este *feedback,* nos indica que ya hemos encontrado la solución se devuelve ***solucionesEnviadas*** conteniendo únicamente este envío preestablecido.

En el caso que no coincida con la solución, el algoritmo generará una nueva. El método ***generaNuevoEnvio()*** es el corazón del algoritmo.

Primero, se crea un mapa llamado **contadorPuntuaciones** para contar cuántas veces aparece cada resultado de fichas posibles (representado por un String). Para cada combinación de **totalcombinacionesPosibles** se comparará con todas las de **possibleCodes** y se mantendrá un recuento de los *feedbacks*.

Una vez iterado por todo **possibleCodes**, se obtiene la combinación con más apariciones, se inserta en **puntuaciones** y se vacía el mapa **contadorPuntuaciones** para dejarlo listo para la siguiente iteración. Se repite para todo **totalcombinacionesPosibles.**

Una vez obtenidas todas combinaciones con el número máximo de apariciones, nos quedaremos con el valor más pequeño y nos quedaremos con todas combinaciones con ese número mínimo de apariciones que insertamos en **enviosCandidatos**.

Únicamente queda elegir qué combinación enviar. Para ello se llama a **obtenSiguienteEnvio(),** que mirará primero si alguno de los **enviosCandidatos** aparecer en los **possiblesCodes**, si es así envía la primera combinación que aparezca, sinó, la primera combinación que aparezca en **totalcombinacionesPosibles.**

En resumen, el algoritmo FiveGuess utiliza una estrategia de búsqueda exhaustiva para generar combinaciones candidatas a ser enviadas en cada turno, basándose en la puntuación obtenida por cada combinación en intentos anteriores, con el objetivo de encontrar la solución correcta con la menor cantidad de intentos posibles.

GENETIC ALGORITHM

Explicación de las estructuras de datos utilizadas

1. **private final int POPULATION_SIZE:** Variable de entorno para delimitar el tamaño de la población sobre la que queremos trabajar.
   Ventaja: Podemos cambiar de manera sencilla y en qualquier momento el tamaño de la población.
2. **private final int GENERATION_SIZE:** Variable de entorno para delimitar el número de generaciones con las que queremos trabajar.
   Ventaja: Podemos cambiar de manera sencilla y en qualquier momento el número de veces que “evolucionaremos” la población.
3. **private final int NMAX_CANDIDATOS:** Variable de entorno para delimitar el número de combinaciones candidatas a enviar.
   Ventaja: Podemos cambiar de manera sencilla y en qualquier momento el número de envios candidatos que generaremos.
4. **List< List<Integer>> solucionesEnviadas:** Lista con las combinaciones enviadas por algoritmo.Ventaja: Sabemos los envios que ya hemos probado para no repetirlos y además luego podemos imprimirlos en el tablero en la capa de presentación.
5. **List< Pair<List<Integer>, Integer >> población:** Lista de tamaño **POPULATION_SIZE** que contiene las combinaciones junto con su *fitness*, este valor es determinante a la hora de utilizar la combinacion para generar nuevas en las siguientes generaciones. Estas combinaciones se irán modificando durante **GENERATION_SIZE** veces mediante *mutaciones, intercambios, inversiones y cruces* para poder otener los nuevos envios y ordenadas segun su *fitness.*
   Ventaja: Al utilizar el pair no necesitamos otra lista adicional donde se guarde el fitness de cada combinacion en el mismo índice, y por lo tanto solo es ncesario ordenar una lista.

6. **List< Pair<List<Integer>, Integer >> enviosCandidatos:** Lista que contiene los envios candidatos obtenidos en esa ronda por el algoritmo junto con su *fitness.* Consideramos que un envio es candidato si da como resultado los mismos valores de blancas y negras para todas las combinaciones que han sido jugadas hasta esa etapa.
7. **List<Pair<Integer, Integer> > negrasBlancasEnvios:** Lista que contiene un pair con el numero de negras y blancas obtenidas después de hacer cada envio, utilizado para la función de *fitness* de cada combinación.
   Ventaja: Accedemos de manera sencilla a las fichas negras o blancas obtenidas accediendo a la posición “numero del envio” que se quiere acceder.
8. **NivelDificultad nivel:** Ventaja: Si en un futuro cambiamos el número de casillas o colores no hará falta hacer cambios al código del algoritmo.
9. **Integer parentPos:** Variable para obtener las posiciones de las combinaciones que utilizaremos para modificar otras al evolucionar la población.

Explicación algoritmo y métodos

Se entiende un envío como la comprobación de la solución proporcionada por el usuario con una combinación generadad por el algoritmo.

El algoritmo se pone en marcha al llamar al método resolve. El algoritmo se sustenta sobre cuatro métodos principales que son **evolucionarPoblacion, calcularAptitud, ordenarPorAptitud y añadirCandidatos.**

En primer lugar, se hace un primer envío de una solución predeterminada. A continuación, se genera una población, y para cada combinación de esta se le asigna un número que será el indicador de lo buena que es con **calcularAptitud()** , cuanto más bajo mejor.

Para calcular la aptitud de cada caombinación, cada solucion enviada, se compara con la combinación. Se utilizan las dos listas que contienen el numero de fichas negras y blancas obtenidas en cada envío para determinar qué coincidencias hay. Luego, se hace la comparacion del envío con la combinacion en cuestion y se sacan las fichas negras y blancas obtenidas.
Finalmente se calcula la diferencia entre la cantiad de fichas negras obtenidas con el envio realizado y con la comparación con la combinación, y lo mismo para las blancas. Este valor se suma y se añade a su puntuación. Este proceso se repite para cada envío realizado.
A continuación se orndena la lista a partir de el valor obtenido anteriormente, a menor puntuación mejor es el codigo.

Una vez ya tenemos inicializado todo entramos en el bucle hasta encontrar la solución o, en el peor de los casos, llegar a los 10 envíos. Para cada ronda se llamaran a los métodos **evolucionarPoblacion, calcularAptitud, ordenarPorAptitud y añadirCandidatos.**

Para **evolucionar** nuestra población, tenemos 4 métodos que se llamarán de manera aleatoria. El principal es **cruce1pos(),** este método obtendrá el valor en la posicion *j* de una combinación padre y se lo insertará en la misma posición a la combinación hija. Las combinaciones padres se accederá de manera aleatoria, pero siempre intentado acceder a las primera combinaciones de la población ya que son las que más se aceracan a la solución. Este método se llamará con una probabilidad de 0.5 .

Para aportar un poco más de aleatoriedad, tenemos los metodos **mutar, intercambiar e invertir. Mutar** se encarga de asignar un color aleatorio a en una posicioón aleatoria. **Intercambiar** cambia dos colores de posición, el uno por el otro. E **invertir**, selecciona aleatoriamente una porción de una combinación en una lista de soluciones candidatas y la invierte, intercambiando los valores dentro de la porción seleccionada. Estos valores se realizan con una probabilidad de 0.03, 0.03 y 0.02 respectivamente.

Tras evolucionar la población, calcular la nueva aptitud de las combinaciones y ordenarlas, se llama a la funcion **añadirCandidatos** para ver si alguno de las combinaciones puede ser candidato a ser enviada. En primer lugar, recorre cada combinacion de la población. Luego, itera sobre las listas **negrasSol** y **blancasSol**, que almacenan las fichas negras y blancas esperadas para cada combinación enviada. Dentro de este bucle, se verifica si el feedback obtenido al comparar la solución candidata con cada combinación coincide con las fichas negras y blancas esperadas. Si no hay coincidencia, se pasa a la siguiente combinación en la población.

Si todas las combinaciones enviadas coinciden con las fichas esperadas para una solución candidata, se procede a verificar si la lista **enviosCandidatos** ha alcanzado su límite máximo de capacidad **NMAX_CANDIDATOS**. Si aún hay espacio disponible, se verifica si la solución candidata ya está presente en la lista. Si no está presente y tampoco se encuentra en la lista **solucionesEnviadas**, se añade a **enviosCandidatos**.

ranking TreeMap<String, Integer>

Esta estructura es un TreeMap que tiene como clave un String y valor un Integer. Esto lo usamos para implementar toda la lógica de los rankings, donde el String será el nombre del Jugador y el Integer la puntuación.

En nuestra implementación se usa de la siguiente manera:

1. Cada vez que se acaba una partida se llama a la función setRanking(), donde se inserta el usuario con la puntuación obtenida en el ranking.
2. Se añade una nueva entrada en el TreeMap<>, si el usuario ya estaba presente en el TreeMap se renueva la entrada si la puntuación es mayor que la ya presente. Si el usuario no estaba presente se añade una nueva entrada y se ordena según la puntuación. De mayor a menor.

Ventajas de usar TreeMap:

- Nos permite tener una ordenación automática y evitar usuarios repetidos.
- Nos permite asociar de manera muy cómoda el ID de un usuario con su puntuación, así que a la hora de imprimir el ranking por pantalla se hace de forma sencilla.

Desventajas de usar TreeMap:

- Nos limita un poco la información que podemos guardar del ranking, ya que por ejemplo si se quisiera que además de la puntuación se guardará el número de rondas que se ha necesitado no podríamos ya que no acepta tener tres valores.

partidas arrayList<String>

Esta estructura es un arrayList que tiene como clave un String. Esto lo usamos para manejar el guardado de partidas, identificándose con un String.

En nuestra implementación se usa de la siguiente manera:

- Cada vez que se guarda una partida se llama a la función guardarPartida(), donde se inserta el String que identifica a la partida y se puede utilizar para cargarla en un futuro y seguir jugandola. Este arrayList es único para cada usuario.

Ventajas de usar arrayList:

- Nos permite guardar Strings de forma sencilla y poder recorrer la lista para mostrarselos al usuario fàcilmente.
- Nos permite hacer gets de los elementos de una forma sencilla.
- Es una estructura de datos que puede ir creciendo a medida que crezca la cantidad de partidas guardadas, por lo que adapta el tamaño a las necesidades del usuario.

Desventajas de usar arrayList:

- El coste de insertar y eliminar puede ser grande, pero no consideramos que sea importante en nuestro proyecto.

records int[]

Esta estructura es un array de enteros usado para representar las 5 mejores puntuaciones del usuario.

En nuestra implementación se usa de la siguiente manera:

- Cada vez que se acaba una partida se mira si la puntuación obtenida es mayor que alguna de las 5 que hay actualmente para ese usuario y si es así la cambia y actualiza el resto de posiciones.

Ventajas de usar arrayList:

- Nos permite guardar de forma sencilla las mejores puntuaciones.

Desventajas de usar arrayList:

- Al hacer una inserción hay que hacer una reordenación del resto de puntuaciones. No hemos considerado el uso de otra estructura ya que el tamaño de esta siempre será 5 y el coste computacional de la reordenación no será excesivo.

tablero Color[]

Esta estructura es un array de Colors utilizado en la presentación para almacenar la combinación de colores creada por el codeMaker.

En nuestra implementación se usa de la siguiente manera:

- Cuando el usuario se encuentra en el turno de codeMaker debe insertar una combinación de colores que será la solución que la máquina deberá encontrar. Para guardar esta combinación usamos lo más sencillo que se nos ha ocurrido que es un array de Colors, cuyo tamaño será de 4 o 5 dependiendo del nivel de dificultad.

Ventajas de usar arrayList:

- Es una forma muy sencilla de guardar la combinación de colores.
- Es fácil referirse a cada posición para su manipulación posterior.

tablero Color[][]

Esta estructura es una matriz de Colors usada en la presentación para representar al conjunto de combinaciones creadas por el usuario.

En nuestra implementación se usa de la siguiente manera:

- Cuando se accede al turno de codeMaker o la máquina genera sus soluciones en la simulación se insertan las diferentes combinaciones usadas para llegar a la solución. El máximo número de filas será 10 y columnas será 4 o 5 dependiendo del nivel de la partida.

Ventajas de usar una matriz:

- Es una forma sencilla de guardar las combinaciones usadas, que permite identificar cada ronda del turno de codeMaker con una fila distinta, correspondiendo cada fila a una combinación.
- Las combinaciones se van insertando en filas consecutivas lo que permite guardar el progreso que se está realizando para llegar a la solución final.

solucion Bola[][]

Esta estructura es una matriz de Colors usada en la presentación para representar al feedback de las combinaciones creadas por el usuario, es decir los indicativos de si después de introducir una bola la solución es correcta, parcialmente correcta o incorrecta.

En nuestra implementación se usa de la siguiente manera:

- Con esta matriz podemos guardar el feedback de todas las combinaciones introducidas por el usuario, por lo que cuando el usuario introduce una combinación se le devuelve un feedback respecto a esta y se almacena en esta matriz.

Ventajas de usar una matriz:

- Es una forma sencilla de guardar el feedback para cada combinación, que permite identificar cada fila con el feedback recibido para cada inserción. De esta forma el usuario puede observar combinaciones ya introducidas y ayudarse de su feedback para poder hacer una nueva combinación.