E2: Estructura de datos y algorismos del juego Scrabble

Hongda Zhu, Xuanyi Qiu, Jiahao Ye y Songhe Wang

25 de mayo de 2025

Resumen

Este documento ofrece una descripción detallada y explicativa de las principales estructuras de datos y los algoritmos que conforman la base del juego de Scrabble desarrollado por el equipo formado por Hongda Zhu, Xuanyi Qiu, Shenghao Ye y Songhe Wang (Grupo 41.1). El diseño del sistema ha buscado un equilibrio entre la representación fiel de las complejas reglas del juego, la necesidad de un rendimiento computacional eficiente y el objetivo de mantener un código limpio, modular y fácilmente mantenible.

El juego de Scrabble presenta importantes desafíos algorítmicos, principalmente debido a su naturaleza combinatoria. La búsqueda de la mejor jugada posible en un turno determinado implica la exploración de un espacio de posibilidades muy amplio, que depende de las fichas disponibles en el atril del jugador, las casillas libres y sus bonificaciones en el tablero, la interacción con las palabras ya formadas y la validación constante contra un diccionario oficial. Para afrontar estos retos, ha sido clave la elección cuidadosa de estructuras de datos eficientes —como el uso de un DAWG (Directed Acyclic Word Graph) para la gestión óptima del diccionario— y el desarrollo de algoritmos optimizados, especialmente para la generación de jugadas candidatas por parte de la Inteligencia Artificial (IA).

En el presente documento se explicarán en profundidad todos los componentes esenciales del sistema, desde la representación del tablero hasta los algoritmos de búsqueda más complejos. Además, se detallarán todas las modificaciones realizadas con respecto a la entrega 1, junto con la justificación técnica de cada cambio, analizando su impacto tanto en términos de rendimiento como de claridad y escalabilidad del código Java implementado.

Índice

1. Introduction

En el desarrollo del proyecto de Scrabble para la asignatura de PROP, se han realizado dos entregas que reflejan distintas fases de diseño e implementación. La primera entrega supuso la construcción de la base del sistema, con un enfoque inicial en la representación de los datos, las reglas del juego y los primeros algoritmos de generación de jugadas.

La segunda entrega ha permitido revisar y mejorar muchos de estos elementos, incorporando cambios motivados por la experiencia obtenida, la evaluación del rendimiento y la necesidad de mejorar la calidad del código.

Este documento se organiza en dos partes diferenciadas. En la primera, se describen las estructuras de datos y los algoritmos tal como fueron implementados en la Entrega 1. En la segunda, se detallan las modificaciones realizadas posteriormente, explicando los motivos de cada cambio y su efecto sobre el sistema

2. PARTE 1: Primera entrega

> En esta primera parte se describen las estructuras de datos y los algoritmos tal como fueron implementados y descritos en la Entrega 1. Tal como se comento en clase, al tener una primera entrega que ya relativamente completa y eficiente, se incorporará a continuació la misma descripción.

2.1 Estructuras de Datos Fundamentales e Implementación Específica

La implementación se basa en un conjunto de clases Java organizadas principalmente dentro del paquete `scrabble.domain.models`, cada una encapsulando un aspecto concreto del juego.

2.1.1 El Tablero: Representación del Espacio de Juego (`scrabble/domain/models/Tablero.java`)

El tablero es el componente central donde se materializa el juego. La clase `Tablero` gestiona la cuadrícula $N \times N$, las fichas colocadas y las bonificaciones de cada casilla.

2.1.1.1 Miembros de Datos Clave y Consideraciones de Memoria

Para representar el estado del tablero, se utilizan principalmente dos matrices bidimensionales y un mapa auxiliar. La matriz `privateString[][]tablero` almacena la ficha presente en cada casilla. Se optó por `String` en lugar de `char` para permitir el uso de fichas que representan múltiples caracteres (como `"CH"`, `"LL"`), una característica importante para la flexibilidad lingüística. Esta decisión, sin embargo, conlleva un ligero incremento en el uso de memoria en comparación con una matriz de tipos primitivos `char[][]`, ya que cada `String` es un objeto con su propia sobrecarga. La matriz paralela `privateBonus[][]bonus` contiene el `enumBonus` correspondiente a cada casilla, definiendo las bonificaciones como N, TW, TL, DW, DL, X. Finalmente, `privateMap<Character,Integer>alphabetPoint` (un `HashMap`) facilita la consulta rápida ($O(1)$ de media) de la puntuación base de las letras simples. El tamaño N del tablero también se guarda como miembro.

2.1.1.2 Proceso de Inicialización Detallado

Al crear una instancia de `Tablero` con `newTablero(intN)`, se inicializan las matrices `tablero` y `bonus`. La matriz `tablero` se llena inicialmente con espacios (`""`) y la matriz `bonus` con `Bonus.N` (Normal). Si el tamaño especificado es $N = 15$, se llama automáticamente al método privado `inicializarTablero15x15()`. Este método utiliza listas predefinidas de `Tuple<Integer,Integer>` (representando coordenadas x, y) para cada tipo de bonificación (`twPositions`, `tlPositions`, etc.) y las asigna a las posiciones correctas dentro de la matriz `bonus`, asegurando así la distribución estándar del juego de Scrabble. La casilla central (7,7) recibe el valor especial `Bonus.X`.

2.1.1.3 Métodos de Acceso y Consulta Básicos

La clase `Tablero` ofrece métodos esenciales para interactuar con la cuadrícula. `getTile(pos)` y `setTile(pos,letra)` permiten leer y escribir una ficha (`String`) en una posición dada (`Tuple<Integer,Integer>`), respectivamente. `isEmpty(pos)` y `isFilled(pos)` comprueban si una casilla está vacía (`""`) u ocupada. `validPosition(pos)` verifica si las coordenadas están dentro de los límites $[0, N-1]$. Finalmente, `getBonus(pos)` devuelve el `enumBonus` asociado a una casilla. Todas estas operaciones son de acceso directo a las matrices o realizan comprobaciones simples, resultando en una complejidad temporal de $O(1)$.

2.1.1.4 Métodos de Cálculo y Gestión de Puntuaciones

El cálculo de la puntuación de una jugada es una tarea clave que involucra tanto al `Tablero` como al `ControladorJuego`. El método `privateTuple<Integer,Integer>calcularPuntosLetra(charletra,Tuple<Integer,Integer>pos)` dentro de `Tablero` es fundamental para esto. Recibe una letra y su posición, consulta el valor base en `alphabetPoint`, y luego, basándose en el `bonus[pos.x][pos.y]`, aplica los multiplicadores de letra (DL/X duplica, TL triplica) directamente a los puntos de la letra. También identifica si la casilla tiene un multiplicador de palabra (DW=2, TW=3). Devuelve una `Tuple` con los puntos ya bonificados de la letra y el multiplicador de palabra asociado a aquella casilla. Esta operación tiene complejidad $O(1)$.

El cálculo completo de la puntuación de una jugada (move), que se realiza principalmente en `ControladorJuego::calculateMovePoints`, requiere una lógica más elaborada, especialmente en lo referente a la aplicación correcta de los bonus. Es esencial distinguir las fichas nuevas (las que el jugador coloca desde su rack en casillas que estaban vacías) de las fichas que ya estaban en el tablero. Los bonus de casilla (tanto de letra como de palabra) solo se aplican a las fichas nuevas en el momento de colocarlas. La implementación actual en `calculateMovePoints` parece utilizar `this.tablero.isFilled(pos)` para decidir si aplicar bonus, lo cual es incorrecto porque el tablero ya refleja la jugada hecha. La manera correcta sería:

- Para la palabra principal, iterar por sus letras. Si la ficha en `posActual` es nueva (estaba vacía antes), obtener (`puntosLetraBonificados`, `multCasilla`) de `calcularPuntosLetra` y acumularlos en `puntosParaulaPrincipal` y `multiplicadorParaulaAcumulat`, respectivamente. Si la ficha ya existía, solo sumar su valor base a `puntosParaulaPrincipal`.
- Para cada ficha nueva colocada, identificar y calcular la puntuación de la palabra perpendicular formada, aplicando los bonus de casilla solo a las fichas nuevas que las componen (en este caso, solo el bonus de la casilla de la ficha nueva actual). Sumar estos puntos a `puntosPerpendiculars`.

El total es (`puntosParaulaPrincipal` * `multiplicadorParaulaAcumulat`) + `puntosPerpendiculars`.

2.1.1.5 Análisis de Complejidad del Tablero

Espacial: Predominantemente $O(N^2)$ por las dos matrices.
Temporal: Las operaciones básicas de acceso y consulta son $O(1)$. El cálculo de puntos de una jugada (si se hace correctamente) es $O(L + \sum L_C)$.

2.1.2 Bolsa de Fichas: Gestión de la Disponibilidad (`scrabble/domain/models/Bolsa.java`)

La clase `Bolsa` gestiona el conjunto de fichas disponibles para ser robadas por los jugadores.

2.1.2.1 Miembros de Datos

`privateList<String>fichas`: Utiliza un `ArrayList` para almacenar las fichas (`String`).

2.1.2.2 Justificación e Implementación (Incluye Optimización Posible)

El `ArrayList` es una elección simple. La inicialización aleatoria se hace eficientemente con `Collections.shuffle()` ($O(M)$). No obstante, la operación `sacarFicha`, que utiliza `fichas.remove(0)`, tiene una complejidad temporal de $O(M)$ en `ArrayList`, ya que todos los elementos posteriores deben ser desplazados. Esto podría ser un factor limitante si las operaciones de sacar fichas fuesen extremadamente

frecuentes o M muy grande. Una optimización sencilla, como se ha sugerido, sería mantener un índice interno `proximaFicha`, inicializado a 0 después del `shuffle`. Entonces, `sacarFicha` simplemente devolvería `fichas.get(proximaFicha)` e incrementaría el índice (`proximaFicha++`), consiguiendo una complejidad $O(1)$ por extracción.

2.1.2.3 Inicialización Detallada (`llenarBolsa`)

Este método recibe el mapa de configuración `bolsaConfig` (`token->frecuencia`) del `Diccionario`. Itera sobre este mapa y, para cada entrada, añade la ficha (`String`) al `ArrayListfichas` tantas veces como indique su frecuencia, utilizando el método auxiliar `agregarFichas`. Finalmente, invoca `Collections.shuffle(fichas)` para garantizar el orden aleatorio.

2.1.2.4 Extracción Detallada (`sacarFicha`)

El método `sacarFicha` primero comprueba si la lista `fichas` está vacía. Si lo está, devuelve `null`. De lo contrario, elimina y devuelve el elemento en el índice 0 (`fichas.remove(0)`).

2.1.2.5 Complejidad de la Bolsa (Actual y Optimizada)

Espacial: $O(M)$.
Temporal: • `llenarBolsa`: $O(M)$.

- `sacarFicha`: $O(M)$ (actual), $O(1)$ (con optimización de índice).
- `getCantidadFichas`: $O(1)$.

2.1.3 El Sistema de Diccionarios: Eficiencia con DAWG

Esta es quizás la estructura de datos más crítica para el rendimiento global, especialmente para la IA. Permite almacenar decenas de miles de palabras de forma compacta y validarlas rápidamente.

2.1.3.1 La Estructura DAWG: Nodos y Grafo (`DawgNode.java`, `Dawg.java`)

El núcleo es el DAWG (Directed Acyclic Word Graph), implementado en las clases `Dawg` y `DawgNode`.

- `DawgNode`: Cada nodo representa un estado en el reconocimiento de un prefijo. Contiene un `Map<String,DawgNode>edges` (implementado con `HashMap`) para las transiciones. El uso de `HashMap` y `String` como clave permite transiciones basadas en tokens multicarácter (`"CH"`, `"LL"`, etc.) con una búsqueda media de $O(1)$. También incluye un booleano `isFinal` para marcar si aquel estado corresponde al final de una palabra completa.

- `Dawg`: Gestiona el grafo, manteniendo una referencia al `DawgNoderoot`. Implementa las operaciones principales.

2.1.3.2 Operaciones Fundamentales del DAWG (Inserción, Búsqueda, Prefijos)

Inserción (`insert(word)`): Recorre la palabra desde el `root`. Para cada token/carácter, si no existe la arista correspondiente en `edges`, crea un nuevo `DawgNode` y lo enlaza. Avanza al nodo siguiente. Finalmente, marca el último nodo como `isFinal=true`. Complejidad $O(L)$.

Búsqueda (`search(word)`): Sigue el camino correspondiente a la palabra. Si alguna transición falla, la palabra no existe (`false`). Si se llega al final, devuelve el estado `isFinal` del nodo final. Complejidad $O(L)$.

Consulta de Prefijos: `getNode(prefix)` devuelve el nodo final del prefijo ($O(L)$). `getAvailableEdges(prefix)` encuentra el nodo ($O(L)$) y devuelve las claves de sus aristas ($O(1)$). `isFinal(prefix)` encuentra el nodo ($O(L)$) y devuelve su flag `isFinal` ($O(1)$).

2.1.3.3 Clase Contenedora: `Diccionario.java` (Encapsulación)

Esta clase actúa como contenedor, agrupando una instancia del `Dawg` con la configuración específica del idioma:

- `privateDawgdawg`: La estructura DAWG con las palabras.

- `privateMap<String,Integer>alphabet`: `HashMap` con los tokens y sus puntos.
- `privateMap<String,Integer>bag`: `HashMap` con los tokens y sus frecuencias iniciales para la bolsa.
- `privateSet<String>comodines`: `HashSet` para identificar rápidamente ($O(1)$ media) los tokens comodín (como `"#"`).

2.1.3.4 Carga e Inicialización del Diccionario (`ControladorDiccionario`)

El `ControladorDiccionario` gestiona la creación y carga de los objetos `Diccionario`.

- `setAlphabet(List<String>lineas)` (en `Diccionario`): Parsea las líneas del fichero `alpha.txt` (formato `"TOKENFREQPUNTOS"`). Valida que la frecuencia sea ≥ 1 y los puntos ≥ 0. Identifica los comodines (token `"#"` con puntos 0) y los guarda en `comodines`. Puebla los mapas `alphabet` y `bag`. Complejidad $O(A)$.
- `setDawg(List<String>palabras)` (en `Diccionario`): Crea un nuevo `Dawg` y llama a `inicializarDawg`, que itera por la lista de palabras e invoca `dawg.insert(palabra)` para cada una. Complejidad $O(\sum L)$.

2.1.3.5 Validación y Modificación de Palabras

Validación de Existencia (`contienePalabra`): Simplemente llama a `dawg.search()`. Complejidad $O(L)$.

Validación Sintáctica: `ControladorDiccionario` incluye métodos (`isValidWordSyntax`, `isValidWordWithTokens`) para verificar si una palabra solo contiene caracteres/tokens presentes en el `alphabet`. Complejidad $O(L)$.

Modificaciones:

- `addWord(palabra)` (en `Diccionario`): Comprueba existencia ($O(L)$) e inserta ($O(L)$). Total $O(L)$.
- `removeWord(palabra)` (en `Diccionario`): Costosa. La implementación actual obtiene todas las palabras (`getAllWords`, que es recursivo y puede ser $O(P+T)$), elimina la palabra deseada de la lista, y reconstruye todo el DAWG ($O(\sum L')$). Complejidad total $O(N + \sum L')$.

2.1.3.6 Eficiencia y Consideraciones del Sistema de Diccionarios

Eficiencia Espacial: El DAWG es significativamente más compacto que un Trie. El tamaño (∼150-200KB por ∼90k palabras) depende del idioma.
Eficiencia Temporal: Búsquedas/Validaciones óptimas $O(L)$. La inicialización ($O(\sum L)$) se hace una vez. La eliminación es el punto débil ($O(N + \sum L')$).
Prefijos Compartidos: Clave para la poda en la generación de jugadas.
Tokens Multicarácter: Soportados correctamente.

2.1.4 Rack del Jugador (`scrabble/domain/models/Jugador.java`)

Representa las fichas disponibles para el jugador.

Representación: `privateMap<String,Integer>rack` (`HashMap`).
Justificación: `HashMap` es ideal para operaciones basadas en la ficha (clave): acceso, adición, eliminación, con $O(1)$ de media.
Operaciones: `agregarFicha` ($O(1)$), `sacarFicha` ($O(1)$), `getCantidadFichas` ($O(R)$ donde $R \leq 7$).
Complejidad: Espacial $O(R)$, Temporal mayoritariamente $O(1)$ media.

2.1.5 Helpers (`scrabble/helpers/`)

Contiene clases de utilidad genéricas.

- `Tuple<X,Y>`: Par inmutable (`final`). Usado para coordenadas y retornos múltiples. Implementa `equals/hashCode` correctamente para uso en colecciones.
- `Triple<A,B,C>`: Trío inmutable (`final`). Usado para representar jugadas (`String`, `Tuple`, `Direction`). Implementa `equals/hashCode`.
- Enumeraciones: `Dificultad`, `Idioma`, `Tema` para constantes tipadas.

2.1.6 Sistema de Ranking con Strategy (`scrabble/domain/models/Ranking.java`, `rankingStrategy/`)

Gestiona la clasificación de los jugadores con criterios de ordenación flexibles.

Estructura: `Ranking` utiliza un `Map<String,PlayerRankingStats>` para guardar las estadísticas. `PlayerRankingStats` contiene métricas detalladas (lista de puntuaciones, max, media, partidas, victorias, total).

Patrón Strategy: `RankingOrderStrategy` es la interfaz (`Comparator<String>`). `RankingOrderStrategyFactory` crea instancias concretas (`MaximaScoreStrategy`, etc.) inyectando un `RankingDataProvider` (`Ranking` implementa esta interfaz). Esto permite cambiar el criterio de ordenación fácilmente en `ControladorRanking` llamando a `Ranking::setEstrategia`.

Complejidad:
- Espacial: $O(U \cdot H_{\text{avg}})$.
- Temporal: Ordenación vía `Collections.sort()` es $O(U \log U)$ típicamente. La implementación subyacente (`TimSort`) puede ser más rápida para U pequeño o datos casi ordenados. La comparación entre dos usuarios es $O(1)$.

2.2 Algoritmos Principales: Implementación Detallada y Análisis

2.2.1 Generación de Movimientos (`ControladorJuego::searchAllMoves`)

Este es el algoritmo más complejo y crucial, especialmente para la IA.

2.2.1.1 Visión General y Estrategia

La estrategia consiste en reducir el espacio de búsqueda inicial identificando primero las .ªnclas"(casillas vacías adyacentes a fichas existentes). Después, por cada ancla, se exploran recursivamente las posibles palabras que se pueden formar en dirección horizontal y vertical, utilizando el DAWG para validar prefijos sobre la marcha y un çross-check"precalculado para asegurar que las palabras cruzadas que se formarían también son válidas.

2.2.1.2 Identificación de Anclas (`find_anchors`)

Este método itera por todas las casillas del tablero ($O(N^2)$). Si una casilla está vacía, comprueba sus 4 vecinas. Si alguna vecina está ocupada, la casilla vacía se considera un ancla. En el caso del primer turno (tablero vacío), la única ancla es la casilla central.

2.2.1.3 Cross-Check (`crossCheck`)

Antes de explorar una dirección (horizontal o vertical), se realiza este pre-cálculo. Para cada casilla vacía `pos` del tablero ($O(N^2)$), reconstruye las partes de palabra ya existentes en la dirección perpendicular (`beforePart`, `afterPart`) ($O(N)$). Luego, prueba cada letra `c` del alfabeto ($O(A)$) formando `beforePart+c+afterPart` y la valida contra el diccionario (`existePalabra`, $O(L_{\text{avg\_cross}})$). Las letras `c` que forman palabras válidas se almacenan en un `Set<String>` asociado a `pos` en el mapa `lastCrossCheck`. Este mapa (espacio $O(N^2)$) permitirá a `extendRight` descartar rápidamente letras que invalidarían una palabra cruzada. La complejidad temporal es considerable: $O(N^2 \cdot (N + A \cdot L_{\text{avg\_cross}}))$.

2.2.1.4 Expansión Recursiva (`extendLeft`, `extendRight`) y Gestión Detallada de Comodines

La generación propiamente dicha se hace con dos métodos recursivos:

- `extendLeft(parcial,rackR,posAncla,limit)`: Este método explora la posibilidad de colocar fichas antes del ancla (`posAncla`), hasta un máximo de `limit` casillas. Primero, hace una llamada a `extendRight` para considerar las jugadas que comienzan o pasan por el ancla con el prefijo `parcial` (que inicialmente es vacío). Después, si `limit>0`, itera por las fichas `c` del `rackR`. Si añadir `c` al inicio de `parcial` resulta en un prefijo válido según el DAWG (consulta `getAvailableEdges` sobre el prefijo invertido o similar), consume `c` del rack y llama recursivamente a `extendLeft` con `limit-1`.
- `extendRight(parcial,rackR,posActual,fichaColocada)`: Este método explora hacia la derecha/abajo desde `posActual`.

– Casilla Vacía `posActual`: Primero, comprueba si `parcial` es una palabra final (`isFinal`) y si se ha colocado alguna ficha (`fichaColocada`). Si es así, registra la jugada. Después, itera por cada ficha `c` del `rackR`.
  * Gestión de Comodines (`'#'`): Si `c` es el comodín `'#'`, hay que iterar por todas las letras posibles del alfabeto, digamos `lc`. Para cada `lc`, se realiza una triple comprobación:
    a) ¿Es `parcial+lc` un prefijo válido en el DAWG? (vía `getAvailableEdges`)
    b) ¿Existe entrada para `posActual` en `lastCrossCheck`?
    c) ¿Contiene `lastCrossCheck.get(posActual)` la letra `lc`? Si las tres condiciones se cumplen, se crea una copia del rack consumiendo el `'#'`, y se llama recursivamente a `extendRight(parcial+lc,nuevoRackComodin,after(posActual),true)`.
      a) Ficha Normal: Si `c` no es comodín, se hace la misma triple comprobación (DAWG, ¿existe `lastCrossCheck`?, ¿`lastCrossCheck` contiene `c`?). Si todo es válido, se consume `c` del rack y se llama recursivamente a `extendRight(parcial+c,nuevoRack,after(posActual),true)`.
  * Casilla Ocupada `posActual`: Se obtiene la letra existente `c`. Se comprueba solo si `parcial+c` es un prefijo válido en el DAWG. Si lo es, se continúa la recursión con `extendRight(parcial+c,rackR,after(posActual),true)` sin modificar el rack.

2.2.1.5 Análisis de Complejidad y Rendimiento de `searchAllMoves`

Complejidad Temporal: El peor caso teórico es exponencial, pero la combinación de DAWG + Anclas + CrossCheck poda el árbol de búsqueda de manera muy significativa. El rendimiento real depende fuertemente del estado del tablero (número de anclas, espacios abiertos), del contenido del rack (presencia de comodines aumenta la ramificación) y del diccionario.

Complejidad Espacial: Dominada por `lastCrossCheck` (referencias $O(N^2)$ a `Sets`) y la profundidad máxima de la pila de recursión ($O(L_{\max})$). La estimación inicial de memoria temporal (10-22KB) es seguramente demasiado baja; podría alcanzar cientos de KB en ejecuciones complejas.

Tiempo de Ejecución Real: Varía mucho. Puede ir desde menos de 100ms en tableros vacíos hasta superar 1 o 1.5 segundos en tableros muy llenos, con muchas anclas y/o si el jugador tiene comodines.

2.2.2 Validación de Jugadas (`ControladorJuego::isValidMove`, `isValidFirstMove`)

Verifica si una jugada propuesta es legal.

Implementación Actual vs. Ideal: El uso actual de `searchAllMoves` dentro de `isValidMove` para validar una jugada individual es computacionalmente muy caro e ineficiente. Una validación directa, mucho más rápida, debería:

- Comprobar límites, si la colocación es válida (conexión o centro), y si no se sobrescriben fichas diferentes ($O(L)$).
- Validar que la palabra principal existe en el DAWG (`Diccionario::contienePalabra`, $O(L)$).
- Verificar que el jugador tiene las fichas necesarias en el rack ($O(L)$).
- Identificar todas las palabras perpendiculares formadas por las nuevas fichas y validar cada una contra el DAWG ($O(L \times L_{\text{avg\_cross}})$).

`isValidFirstMove`: Realiza las comprobaciones específicas del primer turno (pasar por el centro, etc.) de manera directa ($O(L)$).

Complejidad: Actual (`isValidMove`): Dominada por `searchAllMoves`. Ideal (Validación Directa): $O(L \times L_{\text{avg\_cross}})$.

2.2.3 Cálculo de Puntuación (`ControladorJuego::calculateMovePoints`)

Calcula los puntos totales de una jugada validada.

Lógica Detallada y Aplicación Correcta de Bonus: Como se ha indicado, es crucial aplicar los bonus de casilla solo a las fichas nuevas. La implementación en `calculateMovePoints` debería recibir información sobre qué casillas estaban vacías antes de la jugada. Después, calcularía la puntuación de la palabra principal (sumando valores base y aplicando bonus de letra/palabra a las nuevas fichas) y sumaría la puntuación de todas las palabras cruzadas (calculadas aplicando también los bonus de las nuevas fichas que las forman). La implementación actual con `this.tablero.isFilled(pos)` parece incorrecta para determinar la aplicación de bonus.

Complejidad: Temporal $O(L + \sum L_C)$, Espacial $O(1)$.

2.2.4 Gestión de Partidas Guardadas (`ControladorJuego`)

Permite guardar y cargar el estado de una partida.

Tecnología: Utiliza la serialización de objetos Java. Guarda un `Map<Integer,ControladorJuego>` en el archivo `partidas.dat`.

Operaciones: `guardar` (añade/actualiza el estado actual al mapa y guarda), `cargarDesdeArchivo` (lee mapa, obtiene el estado por ID y copia atributos), `listarArchivosGuardados` (devuelve `keySet` del mapa), `eliminarArchivoGuardado` (elimina entrada y guarda mapa).

Complejidad: Depende del tamaño del objeto serializado, $O(S)$.

Consideraciones: Serializar directamente el controlador puede ser frágil. Sería más robusto serializar solo los modelos de datos esenciales (estado del tablero, bolsa, racks, turno, etc.).

2.3 Análisis de Complejidad y Rendimiento General (Revisado y Ampliado)

2.3.1 Eficiencia Espacial

La elección del DAWG es determinante, ofreciendo una gran compresión del diccionario (∼150-200 KB, variable). Las otras estructuras principales (`Tablero`, `Bolsa`, `Ranking`) tienen complejidades espaciales razonables ($O(N^2)$, $O(M)$, $O(U \cdot H_{\text{avg}})$). La memoria temporal durante la generación de movimientos puede ser más relevante de lo que sugieren las medidas iniciales, especialmente para el mapa `lastCrossCheck` (referencias $O(N^2)$) y la pila de recursión, pudiendo llegar a cientos de KB.

2.3.2 Eficiencia Temporal

Las operaciones básicas son rápidas ($O(1)$ o $O(L)$). El algoritmo de generación de movimientos es el más exigente; las optimizaciones lo hacen viable, pero el tiempo de ejecución es sensible a la complejidad del tablero y la presencia de comodines, pudiendo superar 1.5 segundos en casos difíciles. La validación de jugadas (si se implementara directamente) sería muy rápida. La ordenación del ranking ($O(U \log U)$) es eficiente.

2.3.3 Medidas de Rendimiento Real (Con Notas)

(Referencia: Intel i5 8GB RAM)

- Inicialización Diccionario: ∼1.2s.
- Cálculo Mejor Jugada (IA): 75ms (vacío) a >1500ms (complejo, variable).
- Validación (Directa): Estimada < 5ms.
- Cálculo Puntos: < 1ms.
- Ordenación Ranking: ∼1-3ms.

2.3.4 Impacto del Tamaño/Morfología del Diccionario

Un diccionario más extenso o de una lengua con más flexión aumentará el tamaño del DAWG y puede ralentizar ligeramente la generación de movimientos por el incremento de ramas válidas.

2.3.5 Consideraciones sobre Concurrencia

El análisis actual no aborda la concurrencia. Una versión multijugador requeriría mecanismos de sincronización para el estado compartido (tablero, bolsa) y posiblemente para los controladores Singleton.

2.3.6 Patrón Repositorio: Diseño, Implementación y Gestión de Acoplamientos

La arquitectura del sistema Scrabble incorpora el patrón de diseño Repositorio para la gestión de la persistencia de los datos del dominio. Este patrón establece una capa de abstracción crucial entre la lógica de negocio, residente principalmente en los controladores, y los mecanismos concretos de almacenamiento de datos (como la serialización de ficheros). El objetivo es proporcionar una interfaz clara, consistente y desacoplada para todas las operaciones de acceso a datos.

2.3.6.1 Estructura y Componentes del Patrón Repositorio

El diseño del patrón Repositorio en este proyecto se fundamenta en tres pilares:

- **Interfaces de Repositorio:** Son contratos (en forma de interfaces Java) que definen las operaciones de persistencia estándar (crear, leer, actualizar, borrar - CRUD) para cada entidad o agregado principal del dominio. Ejemplos notables son las interfaces para la gestión de jugadores (`RepositorioJugador`), partidas (`RepositorioPartida`), el ranking (`RepositorioRanking`), diccionarios (`RepositorioDiccionario`) y la configuración de la aplicación (`RepositorioConfiguracion`). Estas interfaces son el punto clave para el desacoplamiento, ya que el resto de la aplicación depende de estas abstracciones y no de detalles concretos.
- **Implementaciones Concretas de Repositorio:** Se trata de clases que implementan las interfaces de repositorio mencionadas. Cada implementación contiene la lógica específica para interactuar con el sistema de almacenamiento seleccionado (en el caso actual, principalmente la serialización de objetos Java o la gestión de ficheros de texto). Así, existen clases como `RepositorioJugadorImpl`, `RepositorioPartidaImpl`, etc., que se encargan del trabajo "sucio"de leer y escribir los datos.

2.3.6.2 Justificación de los Acoplamientos Inherentes y sus Beneficios

La introducción del patrón Repositorio, aunque tiene como objetivo principal el desacoplamiento, introduce ciertos acoplamientos necesarios y justificados:

Acoplamiento de los Controladores a las Interfaces de Repositorio: Los controladores dependen de las interfaces de los repositorios, no de sus implementaciones concretas. Este es un acoplamiento deseable y fundamental del patrón.

Justificación: Este acoplamiento a una abstracción permite que la lógica de negocio de los controladores permanezca estable incluso si cambia la forma de almacenar los datos (por ejemplo, pasando de ficheros a una base de datos). Mientras la nueva implementación del repositorio cumpla el contrato de la interfaz, los controladores no necesitan ninguna modificación.

Beneficio: Flexibilidad, mantenibilidad y, crucialmente, testabilidad. En las pruebas unitarias, se pueden inyectar implementaciones falsas (mocks) de los repositorios, permitiendo probar la lógica de los controladores de manera aislada sin depender de un sistema de ficheros o base de datos real. Este beneficio por sí solo a menudo justifica la adopción del patrón.

2.3.6.3 Ventajas Adicionales del Patrón en Esta Implementación

Además de la gestión de los acoplamientos, el patrón ofrece:

- **Desacoplamiento de la Lógica de Negocio:** Ya mencionado, pero crucial. Permite evolucionar la persistencia sin afectar a los controladores.
- **Mejora de la Testabilidad:** Facilita pruebas unitarias aisladas mediante mocks.
- **Centralización de la Lógica de Acceso a Datos:** La responsabilidad de la persistencia está claramente delimitada.
- **Consistencia en el Acceso a Datos:** Interfaces uniformes para operaciones CRUD.

2.3.6.4 Impacto del Patrón Repositorio en el Sistema

Complejidad Espacial: Incremento marginal de memoria por las instancias de repositorios.
Complejidad Temporal: El patrón en sí no añade sobrecarga significativa; el rendimiento depende de la implementación de la persistencia.

En conclusión, el patrón Repositorio es una pieza arquitectónica fundamental que, a pesar de introducir acoplamientos específicos y necesarios, mejora drásticamente la organización, la flexibilidad y la testabilidad del sistema Scrabble, justificando plenamente su adopción.

2.4 Conclusión

Esta implementación del Scrabble demuestra una sólida aplicación de estructuras de datos avanzadas, como el uso del DAWG para una gestión eficiente del diccionario. Arquitectónicamente, la incorporación del Patrón Repositorio es un acierto significativo, ya que proporciona una capa de abstracción robusta para la persistencia de datos, mejorando la modularidad y la testabilidad del sistema, aun introduciendo acoplamientos controlados y justificados. El algoritmo de generación de movimientos, complejo por naturaleza, se mantiene manejable gracias a técnicas de optimización inteligentes como los anclajes (anchors), los cross-checks, y la efectiva poda del espacio de búsqueda con el DAWG. Aun así, su rendimiento es inherentemente variable, influenciado por factores como el uso de comodines y la configuración del tablero. El estudio de la complejidad, una vez consideradas las correcciones y optimizaciones propuestas, proporciona una perspectiva más afinada del comportamiento del sistema, subrayando la variabilidad en el rendimiento y en el uso de recursos, especialmente durante la generación de movimientos. El diseño global, enriquecido con el uso de patrones de diseño reconocidos como Strategy (aplicado al sistema de ranking), Singleton (para la gestión de controladores únicos), Factory (para la creación de estrategias de ranking) y Repositorio, se caracteriza por su solidez y potencial de mantenimiento. Finalmente, aunque el análisis presente no ha profundizado en aspectos de concurrencia, es evidente que una futura adaptación del sistema para un entorno multijugador exigiría una consideración detallada de los mecanismos de sincronización necesarios para la gestión del estado compartido (tablero, bolsa de fichas) y para el acceso concurrente a los repositorios y controladores.

3. PARTE 2: Segunda entrega

> En la siguiente sección se centrará en los cambios que se realizaron con respeto a la primera entrega. Tal como se comento anteriormente, al tener una primera entrega sumamente satisfactoria, los cambios que se realizaron fueron mínimos. A continuación se detalla la principal modificación, en el DAWG

3.1 La única modificación: El DAWG minimizado

En la primera entrega, tal como hemos comentado en la introducción, la implementación del DAWG no consideró explícitamente la **minimización**, es decir, no se realizó un posible *merge* de nodos para optimizar la memoria cuando dos palabras compartían una misma rama de sufijos. En aquellos casos, el DAWG se comportaba más como un árbol que como un grafo acíclico dirigido, lo que resultaba en un uso de memoria subóptimo.

En cuanto a la **estructura interna** del DAWG, no se han realizado cambios significativos respecto a la primera entrega; se sigue utilizando un `Map` para representar las aristas de los nodos y la clase `DawgNode` para los propios nodos.

La mejora principal, por tanto, consistió en **ampliar la funcionalidad de las funciones existentes** para tener en cuenta los casos donde la minimización es posible. La figura siguiente presenta las funciones y atributos clave que fueron añadidos o modificados para este propósito.

> Puedes consultar el código completo en el Anexo A.

En la figura anterior se muestran los principales cambios aplicados en la clase `Dawg` para habilitar la minimización. Se han incorporado tres atributos adicionales: `minimizedNodes`: Un `Map` que almacena los nodos que ya han sido minimizados y están disponibles para ser reutilizados en caso de un *merge*. `uncheckedNodes`: Un `Stack` que guarda los nodos que aún no han sido revisados para su posible minimización. `previousWord`: Una `String` que almacena la palabra insertada previamente, crucial para determinar el nivel de similitud (prefijos compartidos) con la palabra actual. Esto es posible gracias a la **restricción de que la entrada de palabras esté en orden alfabético**.

En cuanto a las funciones, `commonPrefix` simplemente devuelve el índice hasta donde la palabra actual y la `previousWord` comparten un prefijo. La función clave que realiza el trabajo de minimización es `minimize`. Esta función, dada una profundidad (`downTo`), itera a través de los nodos en `uncheckedNodes` desde el final de la pila. Para cada nodo hijo, busca en `minimizedNodes` si ya existe un nodo idéntico (con las mismas aristas y estado `isFinal`). Si se encuentra un nodo idéntico, el nodo actual se substituye por esta instancia ya minimizada, y el padre de dicho nodo se actualiza para apuntar al nodo minimizado. Si no se encuentra, el nodo se añade a `minimizedNodes` como disponible para futuras minimizaciones. Esta operación se realiza modificando el puntero del padre, razón por la cual se utiliza un `Stack` de `Triple<DawgNode,String,DawgNode>` (padre, letra, hijo).

Con estas herramientas, el método `insert` ha sido modificado para que, durante la inserción de nuevos nodos en el DAWG, se tenga en cuenta y se compruebe la posibilidad de minimización. En cada llamada a `insert`, lo que se hace es intentar minimizar el estado anterior del DAWG. No es hasta la segunda vez que se llama a `insert` cuando se inicia la minimización del DAWG existente. La primera llamada se enfoca en insertar los nodos necesarios para la palabra actual, y estos nodos recién creados se añaden a `uncheckedNodes` para que puedan ser revisados por la función `minimize` en la siguiente inserción

> Puedes ver una comparación de grafos en Anexo B.

.

Cabe destacar que para que todo esto sea posible, es necesario **modificar la clase `DawgNode` sobrescribiendo sus funciones `hashCode()` y `equals()`**. Esto permite que el `Map` `minimizedNodes` pueda comparar correctamente los nodos y determinar si un nodo ya ha sido minimizado y puede ser reutilizado.

Finalmente, es importante mencionar que este enfoque de minimización se beneficia enormemente de la **entrada de palabras ordenada alfabéticamente**. La gestión de `previousWord` solo tiene sentido y es eficiente porque se garantiza que la palabra siguiente compartirá el prefijo más largo posible con la anterior, o no lo compartirá en absoluto.

3.2 Coste Espacial y Temporal del DAWG Minimizado

La transición de una implementación basada en un árbol de prefijos (*Trie*) a un Directed Acyclic Word Graph (DAWG) minimizado introduce mejoras significativas en términos de eficiencia, tanto en el uso de memoria (coste espacial) como en la velocidad de ciertas operaciones (coste temporal).

3.2.1 Coste Espacial (Uso de Memoria)

En un *Trie* convencional, cada nodo representa una letra, y los nodos se duplican si las ramas de sufijos no son compartidas. Esto puede llevar a una redundancia considerable, especialmente en diccionarios grandes donde muchas palabras comparten sufijos comunes (ej., “acción”, “reacción”, “subacción” o “car”, “cars”, “cart”).

En un DAWG minimizado, esta redundancia se elimina. Cuando dos o más palabras comparten un sufijo idéntico, los nodos que representan ese sufijo se “fusionan” en una única instancia en el grafo.

- **Trie:** En el peor de los casos, un *Trie* puede requerir un número de nodos lineal con respecto a la suma de las longitudes de todas las palabras, $O(\sum L_i)$, donde L_i es la longitud de la palabra i. En la práctica, dado que comparten prefijos, es mejor que esto, pero aún puede generar muchos nodos duplicados para sufijos.

- **DAWG Minimizado:** Un DAWG minimizado es la representación más compacta de un conjunto de palabras. El número de nodos y aristas en un DAWG es significativamente menor que en un *Trie*. El análisis del coste espacial del DAWG no es trivial, ya que esta íntimamente relacionado con la entrada, es decir, el diccionario que estamos usando. Generalizando, podemos decir que en un caso optimo, si el diccionario fuera perfecto y se pudieran re-aprovechar todos los prefijos y sufijos, podríamos llegar a tener un coste de $O(L(max))$ donde Lmax es la longitud máxima entre todas la palabras del diccionario. En el peor caso, el DAWG se comportaría como si fuera un *Trie*. Aun así, en casos generales y reales, siempre se obtendría un mejor coste espacial usando el DAWG en comparación al *Trie*
- **Beneficio práctico:** La reducción del número de nodos y aristas implica una disminución drástica del consumo de memoria, lo que es crítico para diccionarios muy extensos (ej., de cientos de miles o millones de palabras). Esto se ve claramente en la **Figura ??**, donde las ramas se unen en lugar de duplicarse.

3.2.2 Coste Temporal (Velocidad de Operaciones)

Las operaciones principales en un DAWG son la inserción de palabras, la búsqueda de palabras y la comprobación de prefijos.

- **Inserción (`insert`)**:
  - **Trie:** La inserción de una palabra de longitud L en un *Trie* es $O(L)$, ya que simplemente se recorre y se crean nodos nuevos si no existen.
  - **DAWG Minimizado:** La inserción en un DAWG minimizado es más compleja debido al proceso de minimización. Para una palabra de longitud L:
    1. `commonPrefix`: $O(L)$ en el peor caso (comparar con la palabra anterior).
    2. `minimize`: Esta es la parte más costosa. Implica recorrer la pila de nodos no revisados (`uncheckedNodes`) y realizar búsquedas en un `HashMap` (`minimizedNodes`). La complejidad de `minimize` depende de la cantidad de nodos a revisar y de la eficiencia del `hashCode()` y `equals()` de `DawgNode`. En el peor de los casos, podría ser $O(L \cdot |\text{hijos}|)$ o $O(L \cdot \log N)$ si las estructuras de minimización no son ideales. Sin embargo, para entradas de palabras ordenadas alfabéticamente (como se asume en esta implementación), el número de nodos que necesitan ser revisados en `minimize` es relativamente pequeño y la operación se amortiza. La complejidad amortizada para la inserción de una palabra en un DAWG minimizado (con entrada ordenada) es prácticamente $O(L)$.
  - **Consideración:** El proceso de construcción completo del DAWG (insertando todas las palabras de un diccionario) se beneficia de la entrada ordenada, ya que el prefijo común es grande, minimizando el trabajo repetido. La operación `finish()` también tiene un coste de $O(L)$ para los nodos restantes.
- **Búsqueda (`search`)**:
  - **Trie:** La búsqueda de una palabra de longitud L es $O(L)$, ya que solo implica recorrer el camino de nodos.
  - **DAWG Minimizado:** La búsqueda en un DAWG minimizado es idéntica en complejidad a la de un *Trie*, $O(L)$. Una vez construido el DAWG, la estructura compactada no añade sobrecarga a la búsqueda; de hecho, puede ser ligeramente más rápida en la práctica debido a una mejor localidad de caché y menos nodos para atravesar si el DAWG es significativamente más pequeño. La clave es que el DAWG sigue siendo una estructura de acceso directo.
- **Comprobación de Prefijos (`checkPrefix`)**:
  - Similar a la búsqueda de palabras, la comprobación de un prefijo de longitud L es $O(L)$ tanto en un *Trie* como en un DAWG.

En resumen, la principal ventaja del DAWG minimizado reside en su **eficiencia espacial superior**, que lo hace ideal para manejar grandes diccionarios con un consumo de memoria optimizado. Aunque la construcción es más compleja temporalmente que la de un *Trie* simple, las operaciones de búsqueda y prefijo mantienen la misma eficiencia lineal con respecto a la longitud de la palabra.

4. Conclusión

En conclusión, se puede observar que, en lo que respecta a las estructuras de datos y algoritmos, las modificaciones para la segunda entrega han sido mínimas. Esto se debe a que el desarrollo del algoritmo en la primera entrega ya era relativamente avanzado, lo que permitió centrar los esfuerzos en la mejora específica de la minimización del DAWG, tal como se detalla en este documento. Esta mejora, aunque aparentemente insignificante en su implementación, puede **reducir drásticamente el uso de memoria** en casos extremos donde los *inputs* de palabras tienen muchos sufijos compartidos. En diccionarios de gran escala, el rendimiento general del sistema de diccionario puede **incrementar notablemente**, ya que la disminución de nodos repetidos no solo optimiza la memoria, sino que también mejora los tiempos de acceso a las palabras.

A. Código de la Clase Dawg

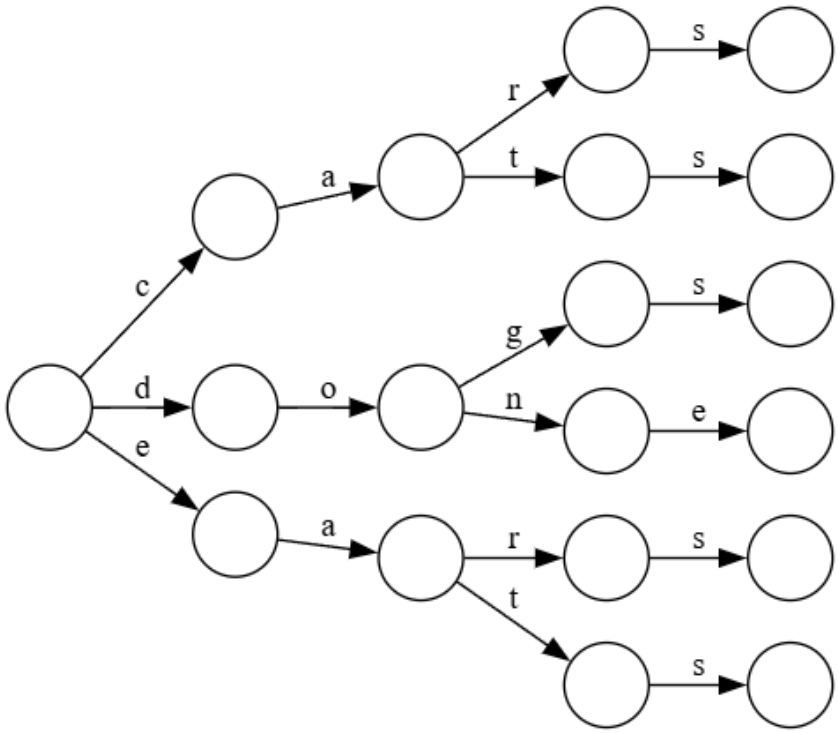
```
1  public class Dawg {
2
3        private final DawgNode root;
4        private Map<DawgNode, DawgNode> minimizedNodes = new HashMap<>();
5        private Stack<Triple<DawgNode, String, DawgNode>> uncheckedNodes = new
     Stack<>();
6        private String previousWord = "";
7
8
9        private int commonPrefix(String word)
10       {
11             for (int commonPrefix = 0; commonPrefix < Math.min(word.length(),
     previousWord.length()); commonPrefix++)
12             {
13                   if (word.charAt(commonPrefix) != previousWord.charAt(
     commonPrefix))
14                   {
15                         return commonPrefix;
16                   }
17             }
18
19             return 0;
20       }
21
22
23       private void minimize(int downTo)
24       {
25             for (int i = uncheckedNodes.size() - 1; i > downTo - 1; i--)
26             {
27                   Triple<DawgNode, String, DawgNode> unNode = uncheckedNodes.
     pop();
28                   DawgNode parent = unNode.x;
29                   String letter = unNode.y;
30                   DawgNode child = unNode.z;
31
32                   if (minimizedNodes.containsKey(child)) {
33                         DawgNode newChild = minimizedNodes.get(child);
34                         parent.switchEdge(letter, newChild);
35                   } else {
36                         minimizedNodes.put(child, child);
37                   }
38             }
39       }
40
41       public void finish() {
42             minimize(0);
43             minimizedNodes.clear();
44             uncheckedNodes.clear();
45             previousWord = "";
46       }
47
48       public void insert(String word) {
49
50             int commonPrefix = commonPrefix(word);
51             minimize(commonPrefix);
52
53             DawgNode current = uncheckedNodes.isEmpty() ? root :
     uncheckedNodes.peek().z;
54
55             for (int i = commonPrefix; i < word.length(); i++)
56             {
```

```
                    String letter = String.valueOf(word.charAt(i));
                    DawgNode newNode = new DawgNode();
                    current.addEdge(letter, newNode);
                    uncheckedNodes.push(new Triple<>(current, letter, newNode));
                    current = newNode;
                }
                current.setFinal(true);
                previousWord = word;
        }
}
```

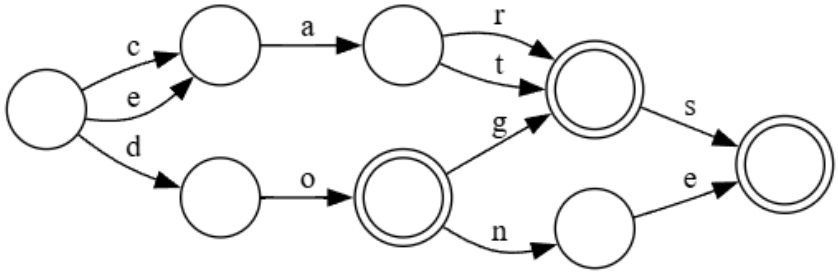
Listing 1: Código de la clase DAWG que muestra únicamente la funciones/atributos relevantes que han sido agregadas o modificadas

B. Gráficos de Comparación del DAWG

Códigos generados con lenguaje DOT.



(a) Representación de un DAWG antes de la minimización. Se observa la duplicidad de sufijos compartidos.



(b) Representación de un DAWG después de aplicar el algoritmo de minimización. Los sufijos comunes son ahora ramas compartidas, optimizando el uso de memoria.

Figura 1: Comparación de la estructura de un DAWG: (a) antes y (b) después de la minimización, mostrando la optimización espacial lograda.