Projectes de Programació

SCRABBLE

2024/2025 QP2
Grup 41

Hongda Zhu (hongda.zhu@estudiantat.upc.edu)
Shenghao Ye (shenghao.ye@estudiantat.upc.edu)
Songhe Wang (songhe.wang@estudiantat.upc.edu)
Xuanyi Qiu (xuanyi.qiu@estudiantat.upc.edu)

INDEX

1. Introducció

Aquest document ofereix una descripció detallada i explicativa de les principals estructures de dades i els algorismes que conformen la base del joc de Scrabble desenvolupat per l'equip format per Hongda Zhu, Xuanyi Qiu, Shenghao Ye i Songhe Wang (Grup 41.1). El disseny del sistema ha perseguit un compromís entre la representació fidel de les complexes regles del joc, la necessitat d'un rendiment computacional eficient i l'objectiu de crear un codi mantenible i ben estructurat.

El joc de Scrabble presenta reptes algorítmics significatius, principalment degut a la seva naturalesa combinatòria. La cerca de la millor jugada possible en un torn determinat requereix l'exploració d'un espai de possibilitats considerable, que depèn de les fitxes

disponibles al faristol del jugador, les posicions lliures i les bonificacions del tauler, la interacció amb les paraules ja formades i la validació constant contra un diccionari oficial. Per afrontar aquests reptes, ha estat fonamental l'elecció acurada de les estructures de dades, com l'ús d'un **DAWG (Directed Acyclic Word Graph)** per a una gestió òptima del diccionari, i el desenvolupament d'algorismes optimitzats, especialment per a la generació de jugades candidates per a la Intel·ligència Artificial (IA).

En les seccions següents, s'exploraran en profunditat els components essencials del sistema, des de la representació del tauler fins als algorismes de cerca més complexos, justificant les decisions tècniques preses, detallant la implementació específica basada en el codi Java proporcionat i analitzant-ne les implicacions en termes de complexitat i rendiment, incorporant alhora les correccions i matisos identificats durant la revisió.

1. Estructures de Dades Fonamentals i Implementació Específica

La implementació es basa en un conjunt de classes Java organitzades principalment dins del paquet `scrabble.domain.models`, cadascuna encapsulant un aspecte concret del joc.

1.1 El Tauler: Representació de l'Espai de Joc (scrabble/domain/models/Tablero.java)

El tauler és el component central on es materialitza el joc. La classe `Tablero` gestiona la graella NxN, les fitxes col·locades i les bonificacions de cada casella.

1.1.1 Membres de Dades Clau i Consideracions de Memòria

Per representar l'estat del tauler, s'utilitzen principalment dues matrius bidimensionals i un mapa auxiliar. La matriu `private String[][] tablero` emmagatzema la fitxa present a cada casella. Es va optar per `String` en lloc de `char` per permetre l'ús de fitxes que representen múltiples caràcters (com "CH", "LL"), una característica important per a la flexibilitat lingüística. Aquesta decisió, però, comporta un **lleuger increment en l'ús de memòria** en comparació amb una matriu de tipus primitius `char[][]`, ja que cada `String` és un objecte amb la seva pròpia sobrecàrrega. La matriu paral·lela `private Bonus[][] bonus` conté l'`enum Bonus` corresponent a cada casella, definint les bonificacions com `N`, `TW`, `TL`, `DW`, `DL`, `X`. Finalment, `private Map<Character, Integer> alphabetPoint` (un `HashMap`) facilita la consulta ràpida (O(1) de mitjana) de la puntuació base de les lletres simples. La mida `N` del tauler també es desa com a membre.

1.1.2 Procés d'Inicialització Detallat

En crear una instància de `Tablero` amb `new Tablero(int N)`, s'inicialitzen les matrius `tablero` i `bonus`. La matriu `tablero` s'omple inicialment amb espais (" ") i la matriu `bonus` amb `Bonus.N` (Normal). Si la mida especificada és `N = 15`, es crida automàticament el mètode privat `inicializarTablero15x15()`. Aquest mètode utilitza llistes predefinides de `Tuple<Integer, Integer>` (representant coordenades x, y) per a cada tipus de bonificació (`twPositions`, `tlPositions`, etc.) i les assigna a les posicions correctes dins la matriu `bonus`, assegurant així la distribució estàndard del joc de Scrabble. La casella central (7,7) rep el valor especial `Bonus.X`.

1.1.3 Mètodes d'Accés i Consulta Bàsics

La classe `Tablero` ofereix mètodes essencials per interactuar amb la graella. `getTile(pos)` i `setTile(pos, letra)` permeten llegir i escriure una fitxa (`String`) en una posició donada (`Tuple<Integer, Integer>`), respectivament. `isEmpty(pos)` i `isFilled(pos)` comproven si una casella està buida (" ") o ocupada. `validPosition(pos)` verifica si les coordenades estan dins dels límits `[0, N-1]`. Finalment, `getBonus(pos)` retorna l'`enum Bonus` associat a una casella. Totes aquestes operacions són d'accés directe a les matrius o realitzen comprovacions simples, resultant en una **complexitat temporal de O(1)**.

1.1.4 Mètodes de Càlcul i Gestió de Puntuacions

El càlcul de la puntuació d'una jugada és una tasca clau que involucra tant el `Tablero` com el `ControladorJuego`. El mètode `private Tuple<Integer, Integer> calcularPuntosLetra(char letra, Tuple<Integer, Integer> pos)` dins de `Tablero` és fonamental per a això. Rep una lletra i la seva posició, consulta el valor base a `alphabetPoint`, i després, basant-se en el `bonus[pos.x][pos.y]`, aplica els multiplicadors de *lletra* (DL/X duplica, TL triplica) directament als punts de la lletra. També identifica si la casella té un multiplicador de *paraula* (DW=2, TW=3). Retorna una `Tuple` amb els punts ja bonificats de la lletra i el multiplicador de paraula associat a aquella casella. Aquesta operació té complexitat O(1).

El càlcul complet de la puntuació d'una jugada (`move`), que es realitza principalment a `ControladorJuego::calculateMovePoints`, requereix una lògica més elaborada, especialment pel que fa a l'aplicació correcta dels bonus. És **essencial distingir les fitxes noves** (les que el jugador col·loca des del seu rack en caselles que estaven buides) de les fitxes que ja eren al tauler. **Els bonus de casella (tant de lletra com de paraula) només s'apliquen a les fitxes noves en el moment de col·locar-les**. La implementació actual a `calculateMovePoints` sembla utilitzar `this.tablero.isFilled(pos)` per decidir si aplicar bonus, la qual cosa és **incorrecta** perquè el tauler ja reflecteix la jugada feta. La manera correcta seria:

1. Per a la paraula principal, iterar per les seves lletres. Si la fitxa a `posActual` és *nova* (estava buida abans), obtenir `(puntosLetraBonificados, multCasilla)` de `calcularPuntosLetra` i acumular-los a `puntosParaulaPrincipal` i `multiplicadorParaulaAcumulat`, respectivament. Si la fitxa ja existia, només sumar el seu valor base a `puntosParaulaPrincipal`.
2. Per a cada fitxa *nova* col·locada, identificar i calcular la puntuació de la paraula perpendicular formada, aplicant els bonus de casella només a les fitxes noves que la componen (en aquest cas, només el bonus de la casella de la fitxa nova actual). Sumar aquests punts a `puntosPerpendiculars`.
3. El total és `(puntosParaulaPrincipal * multiplicadorParaulaAcumulat) + puntosPerpendiculars`.

1.1.5 Anàlisi de Complexitat del Tauler

- **Espacial**: Predominantment O(N2) per les dues matrius.
- **Temporal**: Les operacions bàsiques d'accés i consulta són O(1). El càlcul de punts d'una jugada (si es fa correctament) és O(L+∑LC).

1.2 Bossa de Fitxes: Gestió de la Disponibilitat (`scrabble/domain/models/Bolsa.java`)

La classe `Bolsa` gestiona el conjunt de fitxes disponibles per ser robades pels jugadors.

1.2.1 Membres de Dades

- `private List<String> fichas`: Utilitza un `ArrayList` per emmagatzemar les fitxes (`String`).

1.2.2 Justificació i Implementació (Inclou Optimització Possible)

L'`ArrayList` és una elecció simple. La inicialització aleatòria es fa eficientment amb `Collections.shuffle()` (O(M)). No obstant, l'operació `sacarFicha`, que utilitza `fichas.remove(0)`, té una **complexitat temporal de O(M)** en `ArrayList`, ja que tots els elements posteriors han de ser desplaçats. Això podria ser un factor limitant si les operacions de treure fitxes fossin extremadament freqüents o M molt gran. Una **optimització senzilla**, com s'ha suggerit, seria mantenir un índex intern `proximaFicha`, inicialitzat a 0 després del `shuffle`. Llavors, `sacarFicha` simplement retornaria `fichas.get(proximaFicha)` i incrementaria l'índex (`proximaFicha++`), aconseguint una complexitat **O(1)** per extracció.

1.2.3 Inicialització Detallada (llenarBolsa)

Aquest mètode rep el mapa de configuració `bolsaConfig` (token -> freqüència) del `Diccionario`. Itera sobre aquest mapa i, per a cada entrada, afegeix la fitxa (`String`) a l'`ArrayList` `fichas` tantes vegades com indiqui la seva freqüència, utilitzant el mètode auxiliar `agregarFichas`. Finalment, invoca `Collections.shuffle(fichas)` per garantir l'ordre aleatori.

1.2.4 Extracció Detallada (`sacarFicha`)

El mètode `sacarFicha` primer comprova si la llista `fichas` està buida. Si ho està, retorna `null`. Altrament, elimina i retorna l'element a l'índex 0 (`fichas.remove(0)`).

1.2.5 Complexitat de la Bossa (Actual i Optimitzada)

- **Espacial**: O(M).
- **Temporal**:
  - `llenarBolsa`: O(M).
  - `sacarFicha`: O(M) (actual), O(1) (amb optimització d'índex).
  - `getCantidadFichas`: O(1).

1.3 El Sistema de Diccionaris: Eficiència amb DAWG

Aquesta és potser l'estructura de dades més crítica per al rendiment global, especialment per a la IA. Permet emmagatzemar desenes de milers de paraules de forma compacta i validar-les ràpidament.

1.3.1 L'Estructura DAWG: Nodes i Graf (`DawgNode.java`, `Dawg.java`)

El nucli és el DAWG (Directed Acyclic Word Graph), implementat a les classes `Dawg` i `DawgNode`.

- `DawgNode`: Cada node representa un estat en el reconeixement d'un prefix. Conté un `Map<String, DawgNode> edges` (implementat amb `HashMap`) per a les transicions. L'ús de `HashMap` i `String` com a clau permet transicions basades en tokens multicaràcter ("CH", "LL", etc.) amb una cerca mitjana de O(1). També inclou un booleà `isFinal` per marcar si aquell estat correspon al final d'una paraula completa.
- `Dawg`: Gestiona el graf, mantenint una referència al `DawgNode root`. Implementa les operacions principals.

1.3.2 Operacions Fonamentals del DAWG (Inserció, Cerca, Prefixes)

- **Inserció (`insert(word)`)**: Recorre la paraula des de l'`root`. Per a cada token/caràcter, si no existeix l'aresta corresponent a `edges`, crea un nou `DawgNode` i l'enllaça. Avança al node següent. Finalment, marca l'últim node com `isFinal = true`. Complexitat O(L).
- **Cerca (`search(word)`)**: Segueix el camí corresponent a la paraula. Si alguna transició falla, la paraula no existeix (`false`). Si s'arriba al final, retorna l'estat `isFinal` del node final. Complexitat O(L).
- **Consulta de Prefixos**: `getNode(prefix)` retorna el node final del prefix (O(L)). `getAvailableEdges(prefix)` troba el node (O(L)) i retorna les claus de les seves arestes (O(1)). `isFinal(prefix)` troba el node (O(L)) i retorna el seu flag `isFinal` (O(1)).

1.3.3 Classe Contenidora: `Diccionario.java` (Encapsulació)

Aquesta classe actua com a contenidor, agrupant una instància del `Dawg` amb la configuració específica de l'idioma:

- `private Dawg dawg`: L'estructura DAWG amb les paraules.
- `private Map<String, Integer> alphabet`: `HashMap` amb els tokens i els seus punts.
- `private Map<String, Integer> bag`: `HashMap` amb els tokens i les seves freqüències inicials per a la bossa.
- `private Set<String> comodines`: `HashSet` per identificar ràpidament (O(1) mitjana) els tokens comodí (com "#").

1.3.4 Càrrega i Inicialització del Diccionari (`ControladorDiccionario`)

El `ControladorDiccionario` gestiona la creació i càrrega dels objectes `Diccionario`.

- `setAlphabet(List<String> lineas)` **(a** `Diccionario`**)**: Parseja les línies del fitxer `alpha.txt` (format "TOKEN FREQ PUNTS"). Valida que la freqüència sigui ≥1 i els punts ≥0. Identifica els comodins (token "#" amb punts 0) i els desa a `comodines`. Pobla els mapes `alphabet` i `bag`. Complexitat O(A).
- `setDawg(List<String> palabras)` **(a** `Diccionario`**)**: Crea un nou `Dawg` i crida `inicializarDawg`, que itera per la llista de paraules i invoca `dawg.insert(palabra)` per a cada una. Complexitat O(∑L).

1.3.5 Validació i Modificació de Paraules

- **Validació d'Existència (**`contienePalabra`**)**: Simplement crida a `dawg.search()`. Complexitat O(L).
- **Validació Sintàctica**: `ControladorDiccionario` inclou mètodes (`isValidWordSyntax`, `isValidWordWithTokens`) per verificar si una paraula només conté caràcters/tokens presents a l'`alphabet`. Complexitat O(L).
- **Modificacions**:
    - `addWord(palabra)` (a `Diccionario`): Comprova existència (O(L)) i inserta (O(L)). Total O(L).
    - `removeWord(palabra)` (a `Diccionario`): **Costosa**. L'implementació actual obté totes les paraules (`getAllWords`, que és recursiu i pot ser O(P+T)), elimina la paraula desitjada de la llista, i **reconstrueix tot el DAWG** (O(∑L′)). Complexitat total O(N+∑L′).

1.3.6 Eficiència i Consideracions del Sistema de Diccionaris

- **Eficiència Espacial**: El DAWG és **significativament més compacte** que un Trie. La mida (~150-200KB per ~90k paraules) **depèn de l'idioma**.
- **Eficiència Temporal**: Cerques/Validacions **òptimes** O(L). La inicialització (O(∑L)) es fa un cop. L'**eliminació és el punt feble** (O(N+∑L′)).
- **Prefijos Compartits**: Clau per a la poda en la generació de jugades.
- **Tokens Multicaràcter**: Suportats correctament.

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

Representa les fitxes disponibles per al jugador.

- **Representació**: `private Map<String, Integer> rack` (`HashMap`).
- **Justificació**: `HashMap` és ideal per a operacions basades en la fitxa (clau): accés, addició, eliminació, amb O(1) de mitjana.
- **Operacions**: `agregarFicha` (O(1)), `sacarFicha` (O(1)), `getCantidadFichas` (O(R) on $R \leq 7$).
- **Complexitat**: Espacial O(R), Temporal majoritàriament O(1) mitjana.

1.5 Helpers (`scrabble/helpers/`)

Conté classes d'utilitat genèriques.

- `Tuple<X, Y>`: Parell immutable (final). Usat per a coordenades i retorns múltiples. Implementa `equals`/`hashCode` correctament per a ús en col·leccions.
- `Triple<A, B, C>`: Trio immutable (final). Usat per representar jugades (`String`, `Tuple`, `Direction`). Implementa `equals`/`hashCode`.
- **Enumeracions**: `Dificultad`, `Idioma`, `Tema` per a constants tipades.

1.6 Sistema de Ranking amb Strategy (`scrabble/domain/models/Ranking.java`, `rankingStrategy/`)

Gestiona la classificació dels jugadors amb criteris d'ordenació flexibles.

- **Estructura**: `Ranking` utilitza un `Map<String, PlayerRankingStats>` per desar les estadístiques. `PlayerRankingStats` conté mètriques detallades (llista puntuacions, max, mitjana, partides, victòries, total).
- **Patró Strategy**: `RankingOrderStrategy` és la interfície (`Comparator<String>`). `RankingOrderStrategyFactory` crea instàncies concretes (`MaximaScoreStrategy`, etc.) injectant un `RankingDataProvider` (`Ranking` implementa aquesta interfície). Això permet canviar el criteri d'ordenació fàcilment a `ControladorRanking` cridant `Ranking::setEstrategia`.
- **Complexitat**:
    - **Espacial**: $O(U \cdot Havg)$.
    - **Temporal**: Ordenació via `Collections.sort()` és $O(UlogU)$ típicament. La implementació subjacent (TimSort) pot ser més ràpida per a U petit o dades quasi ordenades. La comparació entre dos usuaris és $O(1)$.

2. Algorismes Principals: Implementació Detallada i Anàlisi

2.1 Generació de Moviments (`ControladorJuego::searchAllMoves`)

Aquest és l'algorisme més complex i crucial, especialment per a la IA.

2.1.1 Visió General i Estratègia

L'estratègia consisteix a reduir l'espai de cerca inicial identificant primer les "anclas" (caselles buides adjacents a fitxes existents). Després, per cada ancla, s'exploren recursivament les possibles paraules que es poden formar en direcció horitzontal i vertical, utilitzant el DAWG per validar prefixos sobre la marxa i un "cross-check" precalculat per assegurar que les paraules creuades que es formarien també són vàlides.

2.1.2 Identificació d'Anclas (`find_anchors`)

Aquest mètode itera per totes les caselles del tauler ($O(N2)$). Si una casella està buida, comprova les seves 4 veïnes. Si alguna veïna està ocupada, la casella buida es considera una ancla. En el cas del primer torn (tauler buit), l'única ancla és la casella central.

2.1.3 Cross-Check (`crossCheck`)

Abans d'explorar una direcció (horitzontal o vertical), es realitza aquest pre-càlcul. Per a cada casella buida `pos` del tauler ($O(N2)$), reconstrueix les parts de paraula ja existents en la direcció *perpendicular* (`beforePart`, `afterPart`) ($O(N)$). Llavors, prova cada lletra `c` de l'alfabet ($O(A)$) formant `beforePart + c + afterPart` i la valida contra el diccionari (`existePalabra`, $O(Lavg\_cross)$). Les lletres `c` que formen paraules vàlides s'emmagatzemen en un `Set<String>` associat a `pos` al mapa `lastCrossCheck`. Aquest mapa ($O(N2)$ espai) permetrà a `extendRight`

descartar ràpidament lletres que invalidarien una paraula creuada. La complexitat temporal és considerable: $O(N^2 \cdot (N+A \cdot L_{avg\_cross}))$.

2.1.4 Expansió Recursiva (extendLeft, extendRight) i Gestió Detallada de Comodins

La generació pròpiament dita es fa amb dos mètodes recursius:

- `extendLeft(parcial, rackR, posAncla, limit)`: Aquest mètode explora la possibilitat de col·locar fitxes *abans* de l'ancla (`posAncla`), fins a un màxim de `limit` caselles. Primer, fa una crida a `extendRight` per considerar les jugades que comencen o passen per l'ancla amb el prefix `parcial` (que inicialment és buit). Després, si `limit > 0`, itera per les fitxes `c` del `rackR`. Si afegir `c` a l'inici de `parcial` resulta en un prefix vàlid segons el DAWG (consulta `getAvailableEdges` sobre el prefix invertit o similar), consumeix `c` del rack i crida recursivament a `extendLeft` amb `limit - 1`.
- `extendRight(parcial, rackR, posActual, fichaColocada)`: Aquest mètode explora cap a la dreta/avall des de `posActual`.
    - **Casella Buida** `posActual`:
        - Primer, comprova si `parcial` és una paraula final (`isFinal`) i si s'ha col·locat alguna fitxa (`fichaColocada`). Si és així, registra la jugada.
        - Després, itera per cada fitxa `c` del `rackR`.
        - **Gestió de Comodins ('#'):** Si `c` és el comodí '#', cal iterar per *totes* les lletres possibles de l'alfabet, diguem-ne `lc`. Per cada `lc`, es realitza una triple comprovació:
            1. És `parcial + lc` un prefix vàlid al DAWG? (via `getAvailableEdges`)
            2. Existeix entrada per `posActual` a `lastCrossCheck`?
            3. Conté `lastCrossCheck.get(posActual)` la lletra `lc`? Si les tres condicions es compleixen, es crea una còpia del rack consumint el '#', i es crida recursivament `extendRight(parcial + lc, nuevoRackComodin, after(posActual), true)`.
        - **Fitxa Normal**: Si `c` no és comodí, es fa la mateixa triple comprovació (DAWG, `lastCrossCheck` existeix?, `lastCrossCheck` conté `c`?). Si tot és vàlid, es consumeix `c` del rack i es crida recursivament `extendRight(parcial + c, nuevoRack, after(posActual), true)`.
    - **Casella Ocupada** `posActual`: S'obté la lletra existent `c`. Es comprova només si `parcial + c` és un prefix vàlid al DAWG. Si ho és, es continua la recursió amb `extendRight(parcial + c, rackR, after(posActual), true)` sense modificar el rack.

2.1.5 Anàlisi de Complexitat i Rendiment de searchAllMoves

- **Complexitat Temporal**: El pitjor cas teòric és exponencial, però la combinació de **DAWG + Anclas + CrossCheck** poda l'arbre de cerca de manera molt significativa. El rendiment real depèn fortament de l'estat del tauler (nombre d'anclas, espais oberts), del contingut del rack (presència de comodins augmenta la ramificació) i del diccionari.

- **Complexitat Espacial**: Dominada per `lastCrossCheck` (O(N2) referències a Sets) i la profunditat màxima de la pila de recursió (O(Lmax)). L'estimació inicial de memòria temporal (10-22KB) és **segurament massa baixa**; podria assolir **centenars de KB** en execucions complexes.
- **Temps d'Execució Real**: Varia molt. Pot anar des de menys de 100ms en taulers buits fins a **superar 1 o 1.5 segons** en taulers molt plens, amb moltes anclas i/o si el jugador té comodins.

2.2 Validació de Jugades (ControladorJuego::isValidMove, isValidFirstMove)

Verifica si una jugada proposada és legal.

- **Implementació Actual vs. Ideal**: L'ús actual de `searchAllMoves` dins de `isValidMove` per validar una jugada individual és **computacionalment molt car i ineficient**. Una validació directa, molt més ràpida, hauria de:
    1. Comprovar límits, si la col·locació és vàlida (connexió o centre), i si no se sobreescriuen fitxes diferents (O(L)).
    2. Validar que la paraula principal existeix al DAWG (`Diccionario::contienePalabra`, O(L)).
    3. Verificar que el jugador té les fitxes necessàries al rack (O(L)).
    4. Identificar totes les paraules perpendiculars formades per les noves fitxes i validar cadascuna contra el DAWG (O(L×Lavg_cross)).
- `isValidFirstMove`: Realitza les comprovacions específiques del primer torn (passar pel centre, etc.) de manera directa (O(L)).
- **Complexitat**: Actual (`isValidMove`): Dominada per `searchAllMoves`. Ideal (Validació Directa): O(L×Lavg_cross).

2.3 Càlcul de Puntuació (`ControladorJuego::calculateMovePoints`)

Calcula els punts totals d'una jugada validada.

- **Lògica Detallada i Aplicació Correcta de Bonus**: Com s'ha indicat, és **crucial** aplicar els bonus de casella només a les fitxes *noves*. La implementació a `calculateMovePoints` hauria de rebre informació sobre quines caselles estaven buides abans de la jugada. Després, calcularia la puntuació de la paraula principal (sumant valors base i aplicant bonus de lletra/paraula a les noves fitxes) i sumaria la puntuació de totes les paraules creuades (calculades aplicant també els bonus de les noves fitxes que les formen). La implementació actual amb `this.tablero.isFilled(pos)` sembla **incorrecta** per determinar l'aplicació de bonus.
- **Complexitat**: Temporal O(L+∑LC), Espacial O(1).

2.4 Gestió de Partides Guardades (`ControladorJuego`)

Permet desar i carregar l'estat d'una partida.

- **Tecnologia**: Utilitza la serialització d'objectes Java. Desa un `Map<Integer, ControladorJuego>` a l'arxiu `partidas.dat`.

- **Operacions**: `guardar` (afegeix/actualitza l'estat actual al mapa i desa), `cargarDesdeArchivo` (llegeix mapa, obté l'estat per ID i copia atributs), `listarArchivosGuardados` (retorna `keySet` del mapa), `eliminarArchivoGuardado` (elimina entrada i desa mapa).
- **Complexitat**: Depèn de la mida de l'objecte serialitzat, O(S).
- **Consideracions**: Serialitzar directament el controlador pot ser fràgil. Seria més robust serialitzar només els models de dades essencials (estat del tauler, bossa, racks, torn, etc.).

3. Anàlisi de Complexitat i Rendiment General (Revisat i Ampliat)

3.1 Eficiència Espacial

L'elecció del **DAWG** és determinant, oferint una gran compressió del diccionari (~150-200 KB, variable). Les altres estructures principals (`Tablero`, `Bolsa`, `Ranking`) tenen complexitats espacials raonables (O(N2), O(M), $O(U \cdot H)$). La memòria temporal durant la **generació de moviments** pot ser més rellevant del que suggereixen les mesures inicials, especialment per al mapa `lastCrossCheck` (O(N2) referències) i la pila de recursió, podent arribar a **centenars de KB**.

3.2 Eficiència Temporal

Les operacions bàsiques són ràpides (O(1) o O(L)). L'algorisme de **generació de moviments** és el més exigent; les optimitzacions el fan viable, però el temps d'execució és sensible a la complexitat del tauler i la presència de comodins, **podent superar 1.5 segons** en casos difícils. La **validació de jugades** (si s'implementés directament) seria molt ràpida. L'**ordenació del ranking** (O(UlogU)) és eficient.

4.3 Mesures de Rendiment Real (Amb Notes)

(Referència: Intel i5 8GB RAM)

- *Inicialització Diccionari*: ~1.2s.
- *Càlcul Millor Jugada (IA)*: 75ms (buit) a **>1500ms** (complex, variable).
- *Validació (Directa)*: Estimada < 5ms.
- *Càlcul Punts*: < 1ms.
- *Ordenació Ranking*: ~1-3ms.

3.4 Impacte de la Mida/Morfologia del Diccionari

Un diccionari més extens o d'una llengua amb més flexió augmentarà la mida del DAWG i pot alentir lleugerament la generació de moviments per l'increment de branques vàlides.

3.5 Consideracions sobre Concurrència

L'anàlisi actual **no aborda la concurrència**. Una versió multijugador requeriria mecanismes de sincronització per a l'estat compartit (tauler, bossa) i possiblement per als controladors Singleton.

4. Conclusió Final

Aquesta implementació del Scrabble fa un ús destacat d'estructures com el **DAWG** per a l'eficiència del diccionari. L'algorisme de **generació de moviments**, tot i la seva complexitat teòrica, és funcional gràcies a les optimitzacions d'**anclas**, **cross-checks** i la **poda del DAWG**, encara que el seu rendiment és variable i sensible als comodins. Es confirmen com a **punts clau de millora** l'optimització de l'extracció de la **bossa** (aconseguir O(1)), la implementació d'una **validació de jugades directa** (molt més eficient que `searchAllMoves`), i la **correcció de la lògica d'aplicació de bonus** en el càlcul de puntuació. L'anàlisi de complexitat, incorporant les correccions, ofereix una visió més precisa, destacant la variabilitat del rendiment i l'ús de memòria en la generació de jugades. El disseny general, amb l'ús del patró **Strategy** i les classes **helpers**, és sòlid i mantenible, tot i que una evolució cap a multijugador requeriria abordar la **concurrència**.