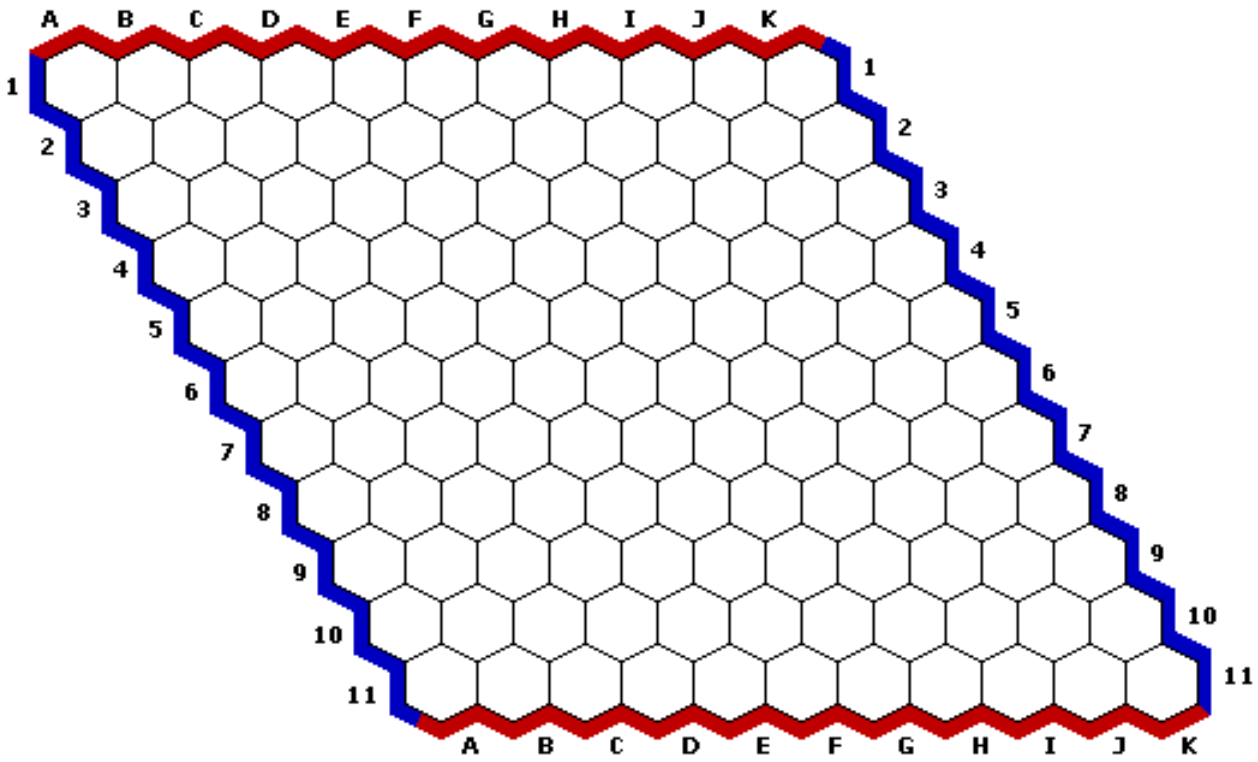
ACTIVITAT 3 - HEX



Alex Matilla Santos
Pau Ortiz Borrás

Alex Matilla Santos Pau Ortíz Borrás

Índex

Alex Matilla Santos Pau Ortíz Borrás

1. Introducció

A l'hora de dur a terme aquesta activitat basada en el joc de taula HEX i en l'algoritme de Dijkstra amb minimax adaptat a IDS, l'entorn de desenvolupament utilitzat ha estat ***Apache NetBeans***, una eina IDE robusta per desenvolupar i depurar el codi dels jugadors i algoritmess, i ***Git/GitHub*** per gestionar el control de versions i la col·laboració en equip. A mès a mès, en aquest projecte també hem implementat l'us de google Drive com a eina extra de Backup, guardant una còpia per dia de treball. La integració entre aquestes eines ha facilitat l'organització, la sincronització de codi i la revisió de versions anteriors, permetent un flux de treball eficient.

Durant el desenvolupament del projecte, hem adoptat una metodologia de treball col·laborativa centrada en la comunicació constant i l'organització. Hem implementat "***Daily Meetings***" per revisar el progrés diari i assegurar que tots els membres estiguessin alineats amb les tasques. A més, hem utilitzat videotrucades per ***Discord*** per a la resolució de problemes i discussions tècniques més detallades. La plataforma ***Git/GitHub*** ha estat clau per a la gestió del codi, permetent-nos fusionar les nostres aportacions de forma eficient, compartir avenços i solucionar conflictes de codi de manera ràpida. Aquesta metodologia ha afavorit un desenvolupament fluït i coordinat.

2. Plantejament i organització del codi

El projecte implementa dos jugadors automàtics per al joc HEX, basats tots dos en l'algoritme d'exploració conegut com algoritme de **Dijkstra**, amb una adaptació personalitzada de l'algoritme **Minimax** utilitzant la tècnica de **poda Alfa-Beta**, el qual pot funcionar amb una profunditat fixa (Ai_Dijkstra), o amb la tècnica de profunditat variable coneguda com a **IDS** (Ai_Dijkstra_IDS). L'organització del codi (en especial, la separació de l'algoritme de Dijkstra en la seva pròpia classe) està dissenyada per maximitzar la modularitat, la llegibilitat i la flexibilitat, facilitant així l'ajust de les diferents estratègies segons les necessitats de cada partida.

2.1. Estructura general i funcions principals

El codi es troba encapsulat dins de les dues classes que representen els diferents jugadors (**Ai_Dijkstra i Ai_Dijkstra_IDS**), que implementa les interfícies **IPlayer** i **IAuto**. Aquestes classes contenen les següents funcionalitats clau:

- **Configuració inicial**: Permet definir la profunditat màxima de cerca de l'algorisme MiniMax mitjançant el constructor (en cas d'utilitzar el jugador amb profunditat màxima).
- **Execució de moviments**: La funció **`move(HexGameStatus s)`** decideix la millor casella per al moviment actual, triant entre les dues implementacions de l'algoritme Minimax i Dijkstra (amb profunditat màxima o amb IDS i TimeOut).
- **Heurística**: Utilitza l'algoritme de Dijkstra per determinar la mínima distància fins a l'altre costat del tauler, per tal de determinar la millor posició on col·locar la fitxa.

3. Heurística i Implementació

3.1. Disseny de l'heurística

L'heurística utilitzada per avaluar els estats del joc es basa en la distància mínima entre les peces d'un jugador i el costat oposat del tauler, calculada mitjançant l'algoritme de Dijkstra. L'objectiu és determinar quin jugador té una posició més avantatjosa a mesura que el joc es desenvolupa. La distància es calcula per a cada jugador, i el valor d'un estat es determina comparant les distàncies dels jugadors.

- **Càlcul de la distància utilitzant Dijkstra**: L'algoritme de Dijkstra es fa servir per calcular la distància mínima des del costat inicial fins al costat oposat del tauler per a cada jugador. El cost de les caselles es defineix com a:

```java
// Aplicació de l'algorisme de Dijkstra.
        while (!cua.isEmpty()) {
            Point actual = cua.poll();
            if (visited[actual.x][actual.y]) continue;
            visited[actual.x][actual.y] = true;

            for (Point veí : estat.getNeigh(actual)) {
                if (visited[veí.x][veí.y]) continue;
                int costMoviment;
                if (estat.getPos(veí.x, veí.y) == 0) {
                    costMoviment = 1;       // Casella buida
                } else if ((jugador == PlayerType.PLAYER1 && estat.getPos(veí.x, veí.y) == 1)
                  ||     (jugador == PlayerType.PLAYER2 && estat.getPos(veí.x, veí.y) == -1)) {
                    costMoviment = 0;       // Casella ocupada pel jugador
                } else {
                    costMoviment = 1000;    // Casella ocupada per l'oponent
                }

                int nouCost = costos[actual.x][actual.y] + costMoviment;

                if (nouCost < costos[veí.x][veí.y]) {
                    costos[veí.x][veí.y] = nouCost;
                    cua.add(veí);
                }
            }
        }
```

  - **0** per a les caselles ocupades pel jugador (màxim avantatge per al jugador).
  - **1** per a les caselles buides (es poden ocupar en el següent moviment).
  - **Infinit** (**1000**) per a les caselles ocupades pel contrari (no accessibles pel jugador).

  D'aquesta manera, l'algorisme de Dijkstra s'utilitza per trobar el camí més curt que permeti al jugador avançar cap al costat oposat del tauler, tenint en compte les seves pròpies peces i les peces del contrari. Cap destacar que som conscients de la efectivitat de la tècnica del sistema de "ponts" i adjacències. Però degut a complicacions a l'hora d'entendre la seva aplicació al codi, hem decidit deixar-lo fora, utilitzant en substitució un sistema de camins amb adjacències directes.

- **Avaluació de l'estat del joc:** La funció avaluar compara les distàncies mínimes dels dos jugadors (el jugador actual i el contrari). Aquest valor es fa servir per assignar un valor numèric a l'estat del joc.

```java
private int avaluar(HexGameStatus estat, int profunditat) {
...
...
    int distJugador = Dijkstra.calcularDistancia(estat, _el_meu_player);
    int distOponent = Dijkstra.calcularDistancia(estat,
                                                 PlayerType.opposite(_el_meu_player));
    return distOponent - distJugador;
}
```

    - Quan el joc no ha acabat, el valor heurístic es calcula com la diferència entre la distància mínima del contrari (distància més curta per a l'oponent) i la distància mínima del jugador actual (distància més curta per al jugador). Així, la heurística intenta maximitzar la distància del contrari i minimitzar la pròpia distància, indicant que un valor negatiu és favorable al jugador.

- **Impacte de la profunditat:** La profunditat restant del Minimax té un impacte directe sobre l'heurística. A mesura que la profunditat de cerca augmenta, es penalitza més les victòries ràpides i es premien les victòries obtingudes en més moviments. Això ajuda a evitar decisions que condueixin a guanyar de manera immediata però que no siguin estratègicament sòlides:

```java
private int avaluar(HexGameStatus estat, int profunditat) {
    if (estat.isGameOver()) {
        if (estat.GetWinner() == _el_meu_player) {
            return 1000 + profunditat;
            // Afegeix la profunditat per premiar victòries més ràpides
        } else {
            return -1000 - profunditat;
            // Penalitza derrotes més ràpides
        }
    }
        ...
        ...
}
```

    - Quan l'estat és una situació de victòria, es retorna un valor elevat (positiu per al jugador guanyador i negatiu per al perdedor), penalitzant els estats amb victòries ràpides i premiant aquells amb victòries més lentes. Això permet que l'algorisme tingui una millor comprensió de l'evolució a llarg termini del joc.

3.2. Implementació de l'algorisme

La implementació del programa contempla dues variants del jugador amb l'algorisme Minimax amb poda alfa-beta i Dijkstra: Una amb **profunditat fixa** i una funcionant mitjançant **IDS** (Iterative Deepening Search) amb sistema de timeout. A continuació, comparem les dues estratègies en termes de funcionament i eficàcia computacional:

- **Implementació amb profunditat fixa**
  - L'algorisme de Minimax amb profunditat fixa calcula les millors jugades a partir d'un nombre predeterminat de moviments, sense tenir en compte la durada de l'execució. L'objectiu és explorar fins a una profunditat màxima fixada i retornar la millor jugada trobada en aquell nivell.
  - La profunditat màxima es passa com a paràmetre al constructor del jugador, i el mètode move fa servir aquesta profunditat per generar el millor moviment possible:

```java
@Override
public PlayerMove move(HexGameStatus s) {
    _el_meu_player        = s.getCurrentPlayer();
    Point millorMoviment  = null;
    int millorValor       = -1000;
    int nodesExplorats    = 0;
    List<Point> moviments = obtenirMoviments(s);

    if (moviments.isEmpty()) {
        // Si no hi ha moviments possibles, retornar un moviment nul
        return new PlayerMove(null, 0L, profunditatMaxima, SearchType.MINIMAX);
    }

    for (Point moviment : moviments) {
        HexGameStatus nouEstat = new HexGameStatus(s);
        nouEstat.placeStone(moviment);

        int[] resultat = minimax(nouEstat, profunditatMaxima - 1,-1000,1000,false, 0);
        int valor       = resultat[0];
        nodesExplorats += resultat[1];

        if (valor > millorValor) {
            millorValor    = valor;
            millorMoviment = moviment;
        }
    }

    if (millorMoviment == null) {
        // Escollir el primer moviment vàlid com a alternativa
        millorMoviment = moviments.get(0);
    }

      return new PlayerMove(millorMoviment, (long) nodesExplorats, profunditatMaxima,
SearchType.MINIMAX);
}
```

En el nostre cas, el Minimax explora tots els possibles moviments fins a una profunditat fixa (**profunditatMaxima**) i després triem el millor moviment basant-se en la valoració de cada estat del joc a aquella profunditat.

- **Implementació amb Iterative Deepening Search (IDS) i Timeout**
  - L'algorisme **IDS**, combinat amb un límit de temps (**timeout**), és una tècnica que busca augmentar progressivament la profunditat de cerca fins que s'esgoti el temps disponible. L'algorisme executa múltiples iteracions de cerca Minimax amb profunditats creixents, i cada iteració es fa amb un límit de temps que permet ajustar dinàmicament la profunditat en funció del temps restant.
  - En aquest cas, el mètode ***move(HexGameStatus s)*** utilitza un bucle per augmentar la profunditat fins que es compleixi el timeout. En cada iteració, es realitza una cerca Minimax amb la profunditat corresponent i es calcula el millor moviment fins que el temps s'esgoti.

```java
@Override
    public PlayerMove move(HexGameStatus s) {
        _el_meu_player        = s.getCurrentPlayer();
        Point millorMoviment  = null;
        int millorValor       = -1000;
        int nodesExplorats    = 0;
        List<Point> moviments = obtenirMoviments(s);

        if (moviments.isEmpty()) {
            // Si no hi ha moviments possibles, retornar un moviment nul
            return new PlayerMove(null, 0L, 1, SearchType.MINIMAX);
        }

        // IDS: Comencem amb profunditat 1 i augmentem en cada iteració.
        for (int profunditatActual = 1; !timeout; profunditatActual++) {
            Point millorMovimentActual = null;
            int millorValorActual      = -1000;

            for (Point moviment : moviments) {
                if (timeout) break;

                HexGameStatus nouEstat = new HexGameStatus(s);
                nouEstat.placeStone(moviment);
                int[] resultat = minimax(nouEstat, profunditatActual - 1, -1000, 1000,
        false, 0);
                int valor       = resultat[0];
                nodesExplorats += resultat[1];

                if (valor > millorValorActual) {
                    millorValorActual = valor;
                    millorMovimentActual = moviment;
                }
            }
            if (!timeout) {
                millorValor = millorValorActual;
                millorMoviment = millorMovimentActual;
            }
        }
        if (millorMoviment == null) {
            millorMoviment = moviments.get(0);
        }
        return    new    PlayerMove(millorMoviment,    (long)    nodesExplorats,    1,
SearchType.MINIMAX);
}
```

Finalment, adjuntem una taula comparativa on recopilem les propietats, avantatges i desavantatges entre les dues implementacions:

Característiques	Profunditat Fixa	IDS amb Timeout
Simplicitat d'implementació	Més senzill, ja que no cal gestionar el temps ni la profunditat variable.	Més complexa, ja que cal gestionar el temps i augmentar dinàmicament la profunditat.
Control del temps d'execució	No garanteix que l'algorisme compleixi els límits de temps, ja que pot explorar profunditats massa grans.	Millor control sobre el temps d'execució, ajustant la profunditat a les restriccions temporals.
Eficiència en l'ús del temps	Pot no ser eficient en limitacions de temps, ja que es fixa la profunditat i pot excedir-se.	Més eficient, ja que ajusta la profunditat per adaptar-se al temps disponible.
Qualitat de la jugada final	Pot produir moviments subòptims si la profunditat de cerca és massa curta.	Millora la qualitat dels moviments a mesura que incrementa la profunditat.
Rendiment en entorns de joc dinàmics	Pot ser menys eficient en jocs amb profunditat d'anàlisi alta, ja que no es pot adaptar al temps real disponible.	Adaptatiu i flexible, ideal en entorns amb límits de temps o condicions variables.
Ús de recursos	Pot consumir més recursos computacionals en profunditats més altes, amb un creixement exponencial dels nodes explorats.	Més eficient en el consum de recursos, ja que no explora més del necessari per adaptar-se al temps disponible.
Gestió de l'estratègia de cerca	Estratègia fixa que no s'ajusta a la dinàmica de la partida, limitant la profunditat d'anàlisi.	Estratègia més dinàmica que ajusta la profunditat en funció del temps disponible i el context del joc.
Flexibilitat	Menys flexible, ja que no pot ajustar-se als canvis en el temps o a les condicions del joc.	Més flexible, ja que permet una adaptació segons les condicions en temps real.

3.3. Importància de l'ordre d'exploració dels fills

Després de diverses execucions, recollim en una taula comparativa les estadístiques mitjanes d'execució de cada implementació:

Resultats	Profunditat fixa	IDS amb Timeout
Nivells baixats	3	1
Nodes explorats	130475	478877
Temps d'execució (s)	4	10

Nivells baixats: la implementació de IDS amb Timeout només ha arribat al nivell 1, mostrant que el mecanisme de timeout de la nostra implementació no està gestionat adequadament. Idealment, IDS hauria de començar amb nivells baixos (1) i avançar gradualment fins al màxim nivell possible dins del temps límit (en el cas de les proves, 10 segons), el qual no és el cas. En canvi, la profunditat Fixa ha arribat fins al nivell 3 (màxima profunditat de les proves), optimitzant el càlcul sense la restricció de control temporal.

Nodes explorats: IDS amb Timeout ha explorat significativament més nodes (478,877) comparat amb Profunditat Fixa (130,475). Això pot ser indicatiu que IDS està malgastant temps en recalcular iteracions anteriors sense avançar.

Temps d'execució: El IDS amb Timeout utilitza tot el temps disponible (10 segons), però el fet que només baixi un nivell mostra una implementació poc òptima del control del temps. En comparació, Profunditat Fixa utilitza 4 segons per arribar al nivell 3, demostrant un ús més eficient del temps dins de les seves limitacions.

Una vegada comparats els resultats, podem veure que la nostra implementació de la metodologia IDS no ha estat la millor, ja que no és capaç d'explorar més d'un nivell en el temps màxim especificat, a sobre de no triar el moviment òptim. Per altra banda, la implementació amb Profunditat fixa mostra resultats molt més positius i adequats, dintre de les seves limitacions.

4. Conclusions

Una vegada finalitzat aquest projecte, tots dos integrants del grup pensem que aquesta activitat, malgrat no haver estat un èxit, ha sigut un viatge intens i interessant. Sense lloc a dubte, ha sigut l'entregable que més posa en pràctica i més relaciona els continguts i metodologies de l'assignatura.

Com ja es va donar a l'anterior activitat, la toma de contacte amb l'entorn de desenvolupament i la fluïdesa del treball no han donat cap mena de problema, i tampoc hem trobat complicacions a l'hora d'implementar i configurar l'entorn ni les eines de treball.

Ara, comparat a l'anterior activitat basada en el connecta4, aquest projecte ja dona per suposat el coneixement i l'agilitat a l'hora de crear una implementació bàsica de l'algoritme Mini-Max, sent una part secundària en el còmput total de l'aplicació. L'estrella del projecte ha sigut, sense cap classe de dubte, la implementació pràctica de l'algoritme de Dijkstra. Aquest algoritme d'exploració ha sigut el nucli del projecte.

A classe, aquest s'explica de manera teòrica amb escasses referències a la programació. Això causa que la seva implementació en codi recaigui completament en mans de l'estudiant, sense cap mena d'indicació de per on començar. Malgrat ser una decisió que fomenta la creativitat de l'estudiant, donant lloc a moltes implementacions diferents (graf d'adjacències, taules de valor heurístic, taules hash, cues de prioritat, etc.), pensem que això dona lloc a la coneguda "analysis paralysis", sent que la gran quantitat d'opcions i la poca guia sobre aquestes ens ha fet sentir aclaparats.

Així i tot, amb MOLTA ajuda del professorat (Bernat), hem aconseguit una versió funcional del Dijkstra. Fent èmfasi en la paraula funcional, i no en l'adjectiu "òptima".

I és que l'algoritme de Dijkstra s'ha menjat la major part del nostre temps de desenvolupament, fent que, tot i l'ampliació de termini de l'entrega, no hàgim pogut dedicar el temps requerit per fer les optimitzacions necessàries al codi, el qual s'ha manifestat en forma d'absència de la tècnica de ponts i adjacències, i la no funcionalitat del jugador basat en IDS.

Seguidament, Referint-nos al projecte en si, no estem satisfets amb el resultat, ja que, com hem dit abans, la manca de temps i la dedicació al Dijkstra ens ha perjudicat greument, fent que no aconseguim tots els objectius pensats per l'entrega. Igualment, valorem l'experiència adquirida, pensem que aquest pla de treball ens ajudarà de cara a quan hagim de desenvolupar algun altre producte sense pautes en el futur.

En conclusió, considerem que aquest projecte ha estat una altra experiència enriquidora per nosaltres com a programadors, i volem sobretot donar les gràcies a Bernat, ja que ha estat responent correus nostres (gairebé diàriament) resolent tots els dubtes plantejats d'una manera excel·lent.

Esperem no veure'ns a la recuperació!!!!