**UNIVERSIDAD DE EXTREMADURA**

Escuela Politécnica

*Grado en Ingeniería Informática en Ingeniería del Software*

**DOCUMENTACIÓN DE PROYECTO: PASO AL REY**

**ENTREGA: 11 DE ABRIL DE 2025**


**Miguel Castelló Sosa**

**mcastelly@alumnos.unex.es**


Asignatura: **Inteligencia Artificial y Sistemas Inteligentes**

Área de conocimiento: **Lenguajes y Sistemas Informáticos**

Departamento: **Ingeniería de Sistemas Informáticos y Telemáticos**

**2025**

# Tabla de contenido

# 1.Descripción de la instalación

## 1.1 Requisitos previos

Para empezar el desarrollo del nuestro código, tuvimos que empezar con la instalación del IntelliJ IDEA Community Edition, un entorno de desarrollo integrado (IDE) gratuito y de código abierto creado por JetBrains, diseñado especialmente para el desarrollo en el lenguaje de programación Java, aunque también soporta otros lenguajes como Kotlin, Groovy y Scala. En este proyecto trabajaremos con el lenguaje Java por lo cual al finalizar la instalación fue necesario instalar el paquete específico para este lenguaje **JDK Java Development Kit.**

## 1.2. Descargar e instalar IntelliJ IDEA Community Edition

Luego de definir el entorno con el que trabajaríamos, tuvimos que seguir unos pasos específicos para la instalación de mismo, estos se enumeran a continuación:

a. Abrir el navegador y entrar en la página oficial de JetBrains: https://www.jetbrains.com/idea/
b. Hacer clic en el botón **"Download"**.
c. Elegir la versión **Community Edition**, que es gratuita y suficiente para la mayoría de los proyectos Java Figura 1.

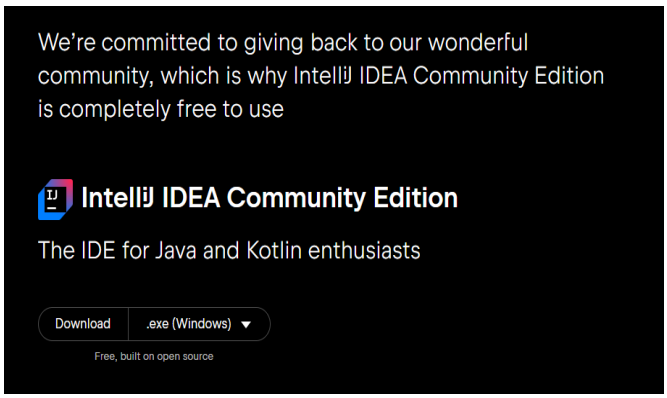


*Figura 1. Versión IntelliJ IDEA Community Edition.*

d. Descarga el instalador correspondiente a nuestro sistema operativo: En nuestro caso Windows y cuyo ejecutable es: **ideaIC-2024.3.5.exe.**
e. Ejecutar el archivo descargado e iniciar la instalación:

- Aceptar los términos de licencia.
- Elegir la carpeta de instalación.
- Marcar las opciones deseadas, en nuestro caso:
- Crear accesos directos.

f. Hacer clic en "Install" y esperar a que finalizara.

g. Una vez instalado, abrimos el IntelliJ IDEA.

## 1.3. Instalación del JDK

Durante la configuración inicial del entorno de desarrollo, se optó por instalar el JDK directamente desde IntelliJ IDEA, aprovechando su asistente integrado.

El procedimiento fue el siguiente:

a. Al abrir IntelliJ IDEA por primera vez, se eligió la opción de crear un nuevo proyecto.
b. En la ventana de configuración del proyecto, el sistema detectó que no había un JDK instalado y ofreció la posibilidad de descargar uno automáticamente.
c. Se seleccionó la versión recomendada (JDK 24), adecuada para el desarrollo del proyecto, y se inició la descarga directamente desde la interfaz del IDE como se muestra en la figura 2.
d. IntelliJ IDEA gestionó automáticamente la descarga e instalación del JDK, sin necesidad de salir del entorno ni configurar rutas manualmente.
e. Una vez finalizado el proceso, el JDK quedó configurado como el SDK principal del proyecto, lo que permitió compilar y ejecutar el código sin inconvenientes.

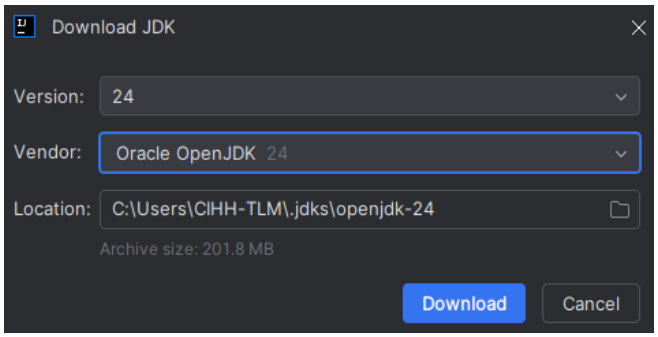


*Figura 2. Descarga de versión JDK24.*

# 2. Métodos de desarrollo del software

Durante el desarrollo del proyecto, se siguió un enfoque incremental, comenzando por la estructura base del sistema y avanzando hacia la implementación de funciones más complejas.

## 2.1 Metodología

El proceso o metodología se llevó a cabo en el siguiente orden:

1. **Definición de clases principales**
   Se inició el desarrollo creando las clases fundamentales del sistema, estableciendo una arquitectura clara y orientada a objetos. Entre las primeras clases desarrolladas se incluyeron aquellas relacionadas con el tablero de juego, las piezas, y la lógica de movimientos. Esta base permitió organizar el resto del código de forma coherente.

2. **Declaración de variables y estructuras iniciales**
   Se definieron las variables necesarias para representar el estado del juego, como la matriz del tablero, la posición de las piezas y los posibles movimientos. También se establecieron estructuras auxiliares como listas o arrays.

3. **Implementación del tablero**
   A continuación, se codificó la representación lógica del tablero, utilizando una matriz bidimensional para reflejar el espacio de juego. Esta implementación incluía tanto la inicialización del tablero como los métodos para visualizarlo y actualizar su estado.

4. **Creación de las piezas y sus características**
   Posteriormente, se implementaron las clases que representaban las piezas del juego, incluyendo sus atributos (posición, tipo) y su comportamiento. Se diseñaron métodos que permitieran verificar movimientos válidos.

5. **Desarrollo de la lógica de movimientos**

   Una vez que las piezas y el tablero estuvieron definidos, se desarrollaron los algoritmos encargados de calcular los movimientos posibles desde una posición dada, evaluando condiciones como obstáculos, límites del tablero, y objetivos del programa.

6. **Integración de la lógica de resolución**

   Finalmente, se integraron los algoritmos de resolución (por ejemplo, escalada simple y A*), los cuales se encargaban de encontrar una secuencia de movimientos óptimos para alcanzar la meta propuesta. Esta parte del desarrollo incluyó también la recopilación de métricas como tiempo de ejecución y número de nodos generados.

## 2.2 Sesiones de trabajo

De manera general las sesiones de trabajo la dividimos en 7 sesiones de trabajo, como se muestra en la figura 3.

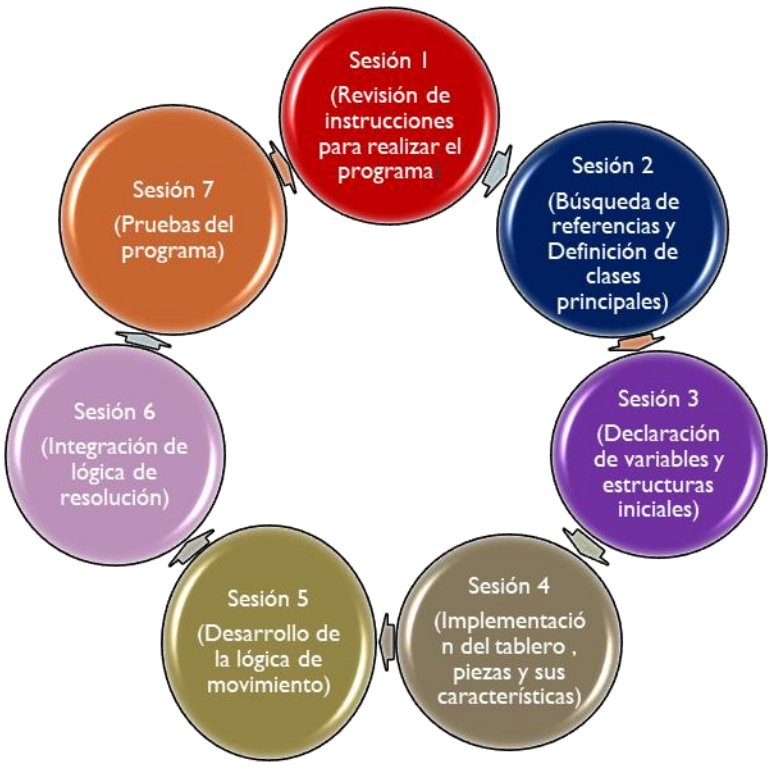


*Figura 3. Sesiones de trabajo para la realización del proyecto.*

**Sesión 1: Revisión de instrucciones para realizar el programa**

Durante esta primera sesión, se configuró el entorno de desarrollo en IntelliJ IDEA y se verificó el funcionamiento del JDK. Se creó la estructura base del proyecto, organizando los paquetes y archivos necesarios.

**Sesión 2: Búsquedas de referencia y definición de clases principales**

Se realizó una búsqueda de referencias que influyeran algunos códigos relacionados a nuestro proyecto. Se definieron las clases principales que formarían la arquitectura del sistema, como el Tablero, las Piezas y la clase controladora del juego.

**Sesión 3: Declaración de variables y estructuras de datos**

En esta etapa se declararon las variables necesarias para representar el estado del tablero, las piezas y los movimientos. Se utilizaron estructuras como matrices para modelar el tablero y listas para almacenar rutas y nodos. Esta sesión fue clave para sentar las bases de la lógica del juego.

**Sesión 4: Implementación del tablero**

Se desarrolló la clase encargada de representar el tablero de juego, utilizando una matriz bidimensional. Se implementaron métodos para inicializar, actualizar y mostrar el estado del tablero. Esta representación fue esencial para permitir la visualización y evolución del juego durante la ejecución.

**Sesión 5: Desarrollo de las piezas y sus movimientos**

Se implementaron las clases correspondientes a las piezas del juego, definiendo sus atributos y comportamientos. Se desarrollaron métodos que permitieran identificar movimientos válidos, tomar decisiones sobre el desplazamiento, y evaluar posibles restricciones según la posición.

**Sesión 6: Lógica de movimientos y validaciones**

Durante esta sesión se trabajó en la lógica de movimientos del juego. Se desarrollaron los algoritmos que evaluaban si una pieza podía moverse a cierta posición, y en qué condiciones. También se implementaron funciones para detectar obstáculos y verificar condiciones de finalización.

**Sesión 7: Integración de algoritmos de resolución**

En la última etapa, se implementaron los algoritmos de resolución utilizados en la búsqueda de soluciones: escalada simple y *A estrella (A)**. Se integraron con el sistema para que trabajaran sobre el tablero ya definido. También se incluyó el registro de datos como tiempo de ejecución y cantidad de nodos generados, con el fin de analizarlos posteriormente.

**Sesión 8: Pruebas del programa**

Se realiza la compilación del programa, corrigiendo los errores iniciales ( por cada sesión), en este punto, tuvo que realizar la corrección, por ejemplo, del espacio entre la coma y la ficha, en el fichero que nos entregaron, todo esto para poder correr el programa.

# 3. Descripción de los algoritmos de resolución empleados en la solución

## 3.1 Técnicas de resolución empleadas

Para resolver el problema propuesto en el proyecto, se emplearon dos técnicas de resolución basadas en algoritmos de búsqueda:

- **Escalada Simple:**

Esta técnica se basa en seleccionar, en cada paso, el estado sucesor que más se aproxima al objetivo, según una función heurística. No mantiene un historial de estados anteriores, lo que puede llevar al algoritmo a quedar atrapado en óptimos locales o mesetas sin solución.

- **A Estrella (A*):**

Es un algoritmo de búsqueda heurística informada que combina el coste acumulado desde el inicio (g(n)) con una estimación del coste restante hasta el objetivo (h(n)). Esta técnica permite encontrar rutas óptimas, siempre que la función heurística sea admisible y consistente. A diferencia de la escalada, A* explora múltiples caminos y mantiene una cola de prioridad para determinar el orden de exploración.

## 3.2 Pseudocódigo de los algoritmos

**Escalada Simple:**

```
scss
CopiarEditar
función EscaladaSimple(nodoInicial):
    nodoActual ← nodoInicial

    mientras nodoActual no sea objetivo:
        sucesores ← generarSucesores(nodoActual)
        mejorSucesor ← seleccionarConMejorHeurística(sucesores)

        si mejorSucesor es peor o igual que nodoActual:
            retornar "Solución no encontrada"

        nodoActual ← mejorSucesor

    retornar nodoActual
```

**A Estrella (A*):**

```
markdown
CopiarEditar
función AEstrella(nodoInicial):
    abiertos ← cola de prioridad con nodoInicial
    cerrados ← conjunto vacío

    mientras abiertos no esté vacío:
        nodoActual ← extraer con menor (g + h) de abiertos

        si nodoActual es objetivo:
            retornar reconstruirRuta(nodoActual)

        mover nodoActual de abiertos a cerrados

        para cada sucesor en generarSucesores(nodoActual):
            si sucesor en cerrados:
                continuar

            calcular g(sucesor) y h(sucesor)

            si sucesor no está en abiertos o tiene mejor coste:
                establecer padre de sucesor ← nodoActual
                añadir o actualizar sucesor en abiertos

    retornar "Sin solución"
```

## 3.3 Estructuras de datos utilizadas

- **Matriz bidimensional (`tablero[][]`):**

  Se utilizó para representar el estado del juego, almacenando la posición de cada pieza en un grid de `n x m`.

- **Listas o arrays dinámicos (`ArrayList`, `List`):**

  Se usaron para almacenar los sucesores generados, las rutas propuestas y el camino final hacia la solución.

- **Cola de prioridad (en A*):**

  Implementada mediante una estructura que ordena los nodos según su valor de `f(n) = g(n) + h(n)`. Esta estructura fue clave para el buen funcionamiento del algoritmo A estrella.

- **Conjunto de nodos cerrados (`Set`):**
  Empleado para evitar la reexploración de estados ya visitados, optimizando la búsqueda.

- **Referencia a nodos padres:**
  Cada nodo guarda una referencia a su nodo padre, lo que permite reconstruir el camino una vez se alcanza el estado objetivo.

# 4. Resultados obtenidos en las baterías de pruebas para cada una de las soluciones

## 4.1. Pruebas con técnica de Escalada Simple y A*

Se resumen en la Tabla 1.

***Tabla 1: Estadísticas de tiempo de ejecución y nodos generados***

| TABLERO | TECNICA DE RESOLUCION | SOLUCIÓN | TIEMPO DE EJECUCIÓN | NODOS GENERADOS |
|---|---|---|---|---|
| PASOALREY1 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **6 ms** | **1** |
| **PASOALREY1** | A* | b1, a2, b2, c2, b1, c1, d2, c2, d2 | 34 ms | 153 |
| PASOALREY2 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **0 ms** | **1** |
| **PASOALREY2** | A* | b4, c4, b4 | 2 ms | 5 |
| PASOALREY3 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **1 ms** | **1** |
| **PASOALREY3** | A* | No se encontró solución con A*. | 10 ms | 198 |
| PASOALREY4 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **1 ms** | **1** |
| **PASOALREY4** | A* | a4, b3, b2, c1, d1, e1, e2, d2, e3, e2, e3 | 4 ms | 107 |
| PASOALREY5 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **0 ms** | **1** |
| **PASOALREY5** | A* | No se encontró solución con A*. | 1 ms | 35 |
| PASOALREY6 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **0 ms** | **1** |
| **PASOALREY6** | A* | d5, d4, c4, b4, c5, b4 | 3 ms | 57 |
| PASOALREY7 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **0 ms** | **1** |
| **PASOALREY7** | A* | a5, b5, c5, c4, b4, c3, d3, c4, c3, d2, d3, d2 | 11ms | 491 |
| PASOALREY8 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **0ms** | **1** |
| **PASOALREY8** | A* | b4, c4, d4, e5, d5, d4, c4, d5, d4, c3, c4, b3, b2, c3, b2 | 4ms | 336 |
| PASOALREY9 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **0 ms** | **1** |
| **PASOALREY9** | A* | e3, d2, c1, b1, b2, a1, b1, c1, b2, a2, a3, b4, c5, d4, e3, d2, c1, d1, e1, e2, d2, e3, e2, e1, d1, c1, b1, b2, a3, b4, c5, d4, e3, d4 | 5ms | 466 |
| PASOALREY10 | **Escalada Simple** | **Escalada Simple no encontró solución.** | **0 ms** | **1** |
| **PASOALREY10** | A* | b1, c2, c1, d2, c2, d2 | 1 ms | 12 |

# 5. Conclusiones sobre los resultados para cada una de las técnicas de resolución empleadas.

## 5.1. Estadísticas de acierto y tiempo

Se puede observar que estadísticamente la técnica de Escalada simple no tiene aciertos, específicamente en este tipo de problemas. Lo contrario ocurre con la técnica o algoritmo estrella.

## 5.2. Ventajas y desventajas de cada técnica

El proyecto ha demostrado cómo los algoritmos de búsqueda informada como **A*** pueden resolver problemas espaciales con restricciones y heurísticas definidas. Por otro lado, la diferencia en rendimiento y capacidad de resolución entre **Escalada simple** y **A*** es evidente, sobre todo cuando el espacio de estados es limitado, pero contiene obstáculos. Por otro lado, hemos incluido una tabla comparativa de las ventajas y desventajas utilizando ambas técnicas (Figura 4).

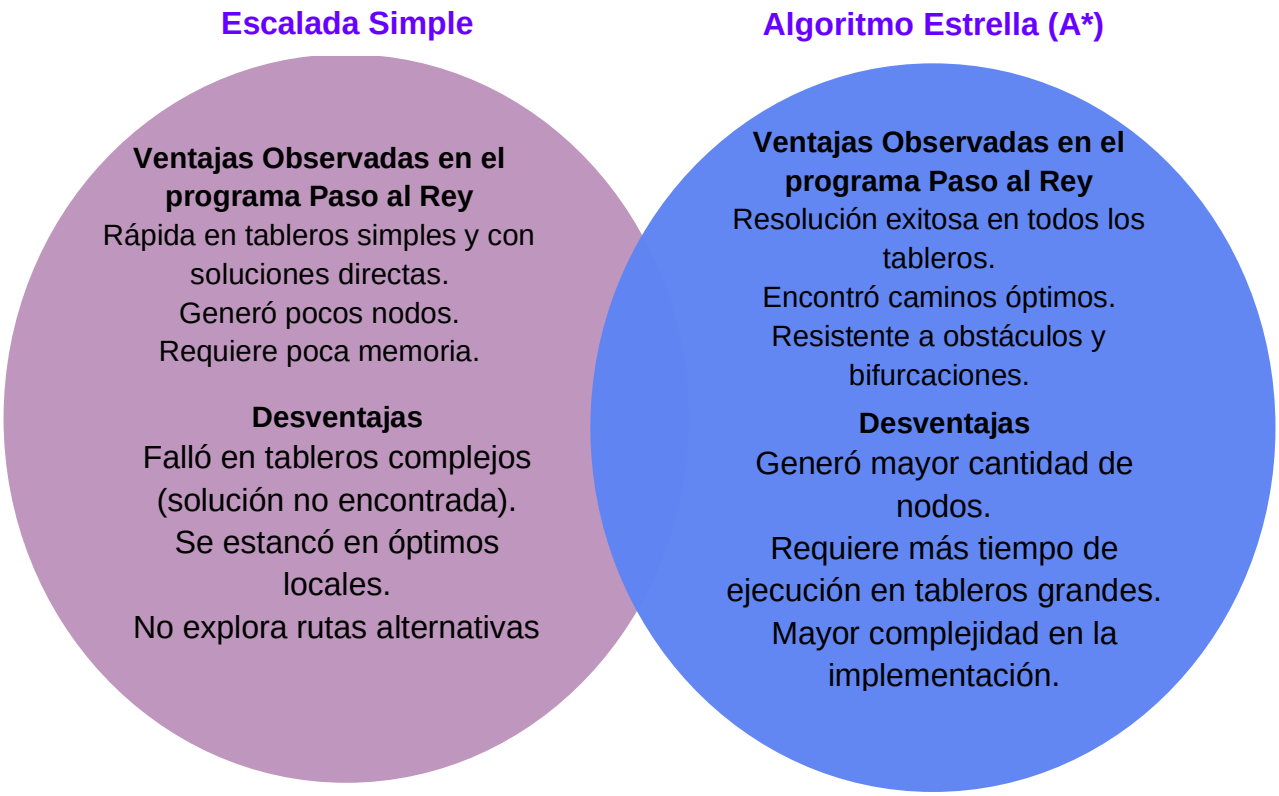


Figura 4. Ventaja y desventaja de cada una de las técnicas utilizadas.

Pensamos que una mejora orientada a usuario sería representar visualmente el tablero y los movimientos paso a paso usando una librería de interfaz gráfica como JavaFX o incluso exportar las soluciones como animaciones en consola, sería una implementación interesante aplicada al juego.

# 6. Referencias


1. Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
   Se utilizó para aclarar conceptos sobre los algoritmos de búsqueda como A* y técnicas de resolución.
2. Oracle. (s.f.). *The Java™ Tutorials*. Recuperado el 10 de abril de 2025, de https://docs.oracle.com/javase/tutorial/
   Documentación oficial para el desarrollo en lenguaje Java.
3. JetBrains. (s.f.). *IntelliJ IDEA Documentation*. Recuperado el 10 de abril de 2025, de https://www.jetbrains.com/idea/documentation/
   Documentación oficial de IntelliJ IDEA, usada para instalar, configurar y ejecutar el proyecto.
4. GeeksforGeeks. (s.f.). *A* Search Algorithm*. Recuperado el 10 de abril de 2025, de https://www.geeksforgeeks.org/a-search-algorithm/
   Explicación clara del algoritmo A* con ejemplos y código.
5. GeeksforGeeks. (s.f.). *Hill Climbing Algorithm in Artificial Intelligence*. Recuperado el 10 de abril de 2025, de https://www.geeksforgeeks.org/hill-climbing-algorithm-in-artificial-intelligence/
   Fuente utilizada para entender y aplicar la técnica de Escalada Simple.
6. Java2Blog. (s.f.). *How to implement a chess board in Java*. Recuperado el 10 de abril de 2025, de https://java2blog.com/java-chess-board/
   Útil como referencia para representar el tablero y las piezas en el entorno Java.
7. Stack Overflow. (s.f.). *Various discussions and solutions for search algorithms in Java*. Recuperado el 10 de abril de 2025, de https://stackoverflow.com/
   Comunidad utilizada para resolver errores comunes y dudas durante el desarrollo.

# Anexo 1

(Programa implementado y explicación de rutina)

**Figura**

```
package model;

enum TipoFigura {
   REY, TORRE, ALFIL, MURO, HUECO, OBJETIVO, VACIO
}

public class Figura {
   private TipoFigura tipo;
   private Posicion pos;

   public Figura(TipoFigura tipo, int x, int y) {
      this.tipo = tipo;
      this.pos = new Posicion(x, y);
   }

   public TipoFigura getTipo() {
      return tipo;
   }

   public Posicion getPos() {
      return pos;
   }

   public boolean puedeMoverA(Posicion destino, Figura[][] tablero) {
      if (tablero[destino.x][destino.y].tipo != TipoFigura.HUECO)
         return false;

      int dx = destino.x - pos.x;
      int dy = destino.y - pos.y;

      return switch (tipo) {
         case REY -> Math.abs(dx) <= 1 && Math.abs(dy) <= 1;
         case TORRE -> (dx == 0 || dy == 0);
         case ALFIL -> Math.abs(dx) == Math.abs(dy);
         default -> false;
      };
   }


}
```

**Posición**

```
package model;

import java.util.Objects;

public class Posicion {
    int x, y;

    public Posicion(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Posicion p)
            return this.x == p.x && this.y == p.y;
        return false;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

**Tablero**

```
package model;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class Tablero {
    private static final int N = 5;
    private Figura[][] matriz = new Figura[N][N];
    private Posicion posRey, posHueco, posObjetivo;
    private double heuristica;
    private int costo;
    private List<String> movimientos = new ArrayList<>();

    public void cargarDesdeArchivo(String ruta) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(ruta));
        for (int i = 0; i < N; i++) {
            String[] linea = br.readLine().split(", ");
            for (int j = 0; j < N; j++) {
                char c = linea[j].charAt(0);
                TipoFigura tipo;

                switch (c) {
                    case 'M' -> tipo = TipoFigura.MURO;
                    case 'H' -> tipo = TipoFigura.HUECO;
                    case 'R' -> {
                        tipo = TipoFigura.REY;
                        posRey = new Posicion(i, j);
                    }
                    case 'O' -> {
                        tipo = TipoFigura.OBJETIVO;
                        posObjetivo = new Posicion(i, j);
                    }
                    case 'A' -> tipo = TipoFigura.ALFIL;
                    case 'T' -> tipo = TipoFigura.TORRE;
                    default -> tipo = TipoFigura.VACIO;
                }

                if (tipo == TipoFigura.HUECO)
                    posHueco = new Posicion(i, j);

                matriz[i][j] = new Figura(tipo, i, j);
            }
        }
        br.close();
        calcularHeuristica();
    }

    public void calcularHeuristica() {
        int dx = posObjetivo.x - posRey.x;
        int dy = posObjetivo.y - posRey.y;
        heuristica = Math.sqrt(dx * dx + dy * dy);
```

```
}

//Si el rey esta en el objetivo
public boolean esObjetivo() {
    return posRey.equals(posObjetivo);
}

public void mostrar() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            System.out.print(simb(matriz[i][j].getTipo()) + " ");
        }
        System.out.println();
    }
}

//Transforma el Tipo de Figura al símbolo en carácter
private String simb(TipoFigura tipo) {
    return switch (tipo) {
        case MURO -> "M";
        case HUECO -> "H";
        case REY -> "R";
        case TORRE -> "T";
        case ALFIL -> "A";
        case OBJETIVO -> "O";
        default -> ".";
    };
}

public List<Tablero> generarSucesores() {
    List<Tablero> sucesores = new ArrayList<>();
    int[][] direcciones = {{-1,0},{1,0},{0,-1},{0,1},{-1,-1},{-1,1},{1,-1},{1,1}};

    // Movimiento normal hacia el hueco
    for (int[] d : direcciones) {
        int i = posHueco.x + d[0];
        int j = posHueco.y + d[1];

        if (i >= 0 && i < N && j >= 0 && j < N) {
            Figura figura = matriz[i][j];
            if (figura.getTipo() == TipoFigura.REY ||
                    figura.getTipo() == TipoFigura.TORRE ||
                    figura.getTipo() == TipoFigura.ALFIL) {

                if (figura.puedeMoverA(posHueco, matriz)) {
                    Tablero copia = this.clonar();
                    copia.intercambiar(figura.getPos(), posHueco);
                    copia.movimientos.addAll(this.movimientos);
                    copia.movimientos.add(posToCoord(figura.getPos()));
                    copia.costo = this.costo + 1;
                    copia.calcularHeuristica();
                    sucesores.add(copia);
                }
            }
        }
```

```
        }

        // Movimiento especial: Rey al objetivo si están adyacentes y el hueco también está
junto al Rey
        for (int[] d : direcciones) {
            int rx = posRey.x + d[0];
            int ry = posRey.y + d[1];

            if (rx == posObjetivo.x && ry == posObjetivo.y) {
                for (int[] dh : direcciones) {
                    int hx = posRey.x + dh[0];
                    int hy = posRey.y + dh[1];

                    if (hx == posHueco.x && hy == posHueco.y) {
                        Tablero copia = this.clonar();
                        copia.matriz[posRey.x][posRey.y] = new Figura(TipoFigura.HUECO,
posRey.x, posRey.y);
                        copia.matriz[posObjetivo.x][posObjetivo.y] = new Figura(TipoFigura.REY,
posObjetivo.x, posObjetivo.y);
                        copia.posRey = new Posicion(posObjetivo.x, posObjetivo.y);
                        copia.posHueco = new Posicion(posRey.x, posRey.y);
                        copia.movimientos.addAll(this.movimientos);
                        copia.movimientos.add(posToCoord(posRey));
                        copia.costo = this.costo + 1;
                        copia.calcularHeuristica();
                        sucesores.add(copia);
                    }
                }
            }
        }

        return sucesores;
    }

    public void intercambiar(Posicion p1, Posicion p2) {
        Figura temp = matriz[p1.x][p1.y];
        matriz[p1.x][p1.y] = matriz[p2.x][p2.y];
        matriz[p2.x][p2.y] = temp;

        TipoFigura tipo1 = matriz[p1.x][p1.y].getTipo();
        TipoFigura tipo2 = matriz[p2.x][p2.y].getTipo();

        if (tipo1 == TipoFigura.REY) posRey = new Posicion(p1.x, p1.y);
        if (tipo2 == TipoFigura.REY) posRey = new Posicion(p2.x, p2.y);
        if (tipo1 == TipoFigura.HUECO) posHueco = new Posicion(p1.x, p1.y);
        if (tipo2 == TipoFigura.HUECO) posHueco = new Posicion(p2.x, p2.y);
    }

    public Tablero clonar() {
        Tablero copia = new Tablero();
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                Figura f = matriz[i][j];
                copia.matriz[i][j] = new Figura(f.getTipo(), i, j);
            }
```

```
    }

    copia.posRey = new Posicion(posRey.x, posRey.y);
    copia.posHueco = new Posicion(posHueco.x, posHueco.y);
    copia.posObjetivo = new Posicion(posObjetivo.x, posObjetivo.y);
    copia.costo = this.costo;
    return copia;
}

public static String posToCoord(Posicion pos) {
    return "" + (char)('a' + pos.y) + (pos.x + 1);
}

public double getHeuristica() {
    return heuristica;
}

public int getCosto() {
    return costo;
}

public List<String> getMovimientos() {
    return movimientos;
}

public Figura[][] getMatriz() {
    return matriz;
}

public void setMatriz(Figura[][] matriz) {
    this.matriz = matriz;
}

public Posicion getPosRey() {
    return posRey;
}

public void setPosRey(Posicion posRey) {
    this.posRey = posRey;
}

public Posicion getPosHueco() {
    return posHueco;
}

public void setPosHueco(Posicion posHueco) {
    this.posHueco = posHueco;
}

public Posicion getPosObjetivo() {
    return posObjetivo;
}

public void setPosObjetivo(Posicion posObjetivo) {
    this.posObjetivo = posObjetivo;
```

```
    }

    public void setHeuristica(double heuristica) {
        this.heuristica = heuristica;
    }

    public void setCosto(int costo) {
        this.costo = costo;
    }

    public void setMovimientos(List<String> movimientos) {
        this.movimientos = movimientos;
    }
}
```

**Solucionador**

```
package control;

import model.Tablero;

import java.util.*;

public class Solucionador {

    static void escaladaSimple(Tablero inicial) {
        Tablero actual = inicial;
        boolean mejora = true;
        int nodosExplorados = 0; // Contador de nodos explorados
        int iteraciones = 0;

        System.out.println("\nIniciando búsqueda por Escalada Simple...");

        while (mejora && !actual.esObjetivo() && iteraciones < 1000) {
            mejora = false;
            iteraciones++;
            nodosExplorados++; // Incrementar contador


            List<Tablero> sucesores = actual.generarSucesores();
            Tablero mejor = null;
            double mejorHeuristica = Double.MAX_VALUE;

            for (Tablero sucesor : sucesores) {
                if (sucesor.getHeuristica() < mejorHeuristica) {
                    mejorHeuristica = sucesor.getHeuristica();
                    mejor = sucesor;
                }
            }

            if (mejor != null && mejor.getHeuristica() < actual.getHeuristica()) {
                actual = mejor;
                mejora = true;
                System.out.println("Iteración " + iteraciones + ": h = " + actual.getHeuristica());
            }
        }

        if (actual.esObjetivo()) {
            System.out.println("\n¡Objetivo alcanzado con Escalada Simple!");
            System.out.println("Pasos: " + actual.getCosto());
            System.out.println("Nodos explorados: " + nodosExplorados);
            System.out.println("Movimientos: " + String.join(", ", actual.getMovimientos()));
            actual.mostrar();
        } else {
            System.out.println("\nEscalada Simple no encontró solución.");
            System.out.println("Nodos explorados: " + nodosExplorados);
            System.out.println("Mejor solución parcial encontrada (h = " + actual.getHeuristica()
+ "):");
            actual.mostrar();
        }
```

```
    }

    public static void aEstrella(Tablero inicial) {
        PriorityQueue<Tablero> cola = new PriorityQueue<>(
                Comparator.comparingDouble(t -> t.getHeuristica() + t.getCosto())
        );
        Set<String> visitados = new HashSet<>();
        cola.add(inicial);
        int nodosExplorados = 0; // Contador de nodos explorados


        while (!cola.isEmpty()) {
            Tablero actual = cola.poll();
            nodosExplorados++; // Incrementar contador

            if (actual.esObjetivo()) {
                System.out.println("\n¡Objetivo alcanzado con A*!\nPasos: " + actual.getCosto());
                System.out.println("Nodos explorados: " + nodosExplorados);
                System.out.println("Movimientos: " + String.join(", ", actual.getMovimientos()));
                actual.mostrar();
                return;
            }

            String hash = actual.getPosRey().getX() + "," + actual.getPosRey().getY() + "," +
                    actual.getPosHueco().getX() + "," + actual.getPosHueco().getY();

            if (visitados.contains(hash)) continue;
            visitados.add(hash);

            for (Tablero sucesor : actual.generarSucesores()) {
                cola.add(sucesor);
            }
        }

        System.out.println("\nNo se encontró solución con A*.");
    }
}
```

**Main**

```
package control;

import model.Tablero;

import java.io.IOException;

public class Main {

    public static void escaladaSimple(Tablero t){

        // Ejecutar algoritmo de Escalada Simple
        System.out.println("\n--- Algoritmo de Escalada Simple ---");
        Tablero tablero2 = t.clonar(); // Crear una copia para el segundo algoritmo

        long tiempoInicio = System.currentTimeMillis(); //Comienza el contador
        Solucionador.escaladaSimple(tablero2);
        long tiempoFin = System.currentTimeMillis();

        System.out.println("\nTiempo empleado (Escalada Simple): " + (tiempoFin -
tiempoInicio) + " ms");//Anoto el tiempo
    }


    public static void aEstrella(Tablero t){
        // Ejecutar algoritmo A*
        System.out.println("\n--- Algoritmo A* ---");
        Tablero tablero2 = t.clonar(); // Crear una copia para el segundo algoritmo

        long tiempoInicio = System.currentTimeMillis(); //Comienza el contador
        Solucionador.aEstrella(tablero2);
        long tiempoFin = System.currentTimeMillis();

        System.out.println("\nTiempo empleado (A*): " + (tiempoFin - tiempoInicio) + "
ms");//Anoto el tiempo
    }



    public static void main(String[] args) throws IOException {

        //VERSION SENCILLA PARA PRUEBAS


        for(int i=1;i<11;i++){
            Tablero tablero = new Tablero();
            tablero.cargarDesdeArchivo("./files/PASOALREY"+i+".TXT");
            System.out.println("-----------------------------------------------------");
            System.out.println("Tablero PASOALREY"+i);
            escaladaSimple(tablero);
            aEstrella(tablero);
            //tablero.mostrar();
        }
```

```
        //Cargo el archivo y lo muestro
//        tablero.cargarDesdeArchivo("./files/PASOALREY3.TXT");
//        System.out.println("Tablero inicial:");
//        tablero.mostrar();




    }
}
```

# Anexo 2
# Explicación del código en Java.

**1. enum TipoFigura**

Define los diferentes tipos de elementos que pueden aparecer en el tablero:

java
CopiarEditar
REY, TORRE, ALFIL, MURO, HUECO, OBJETIVO, VACIO

Esto permite clasificar cada celda del tablero y aplicar reglas específicas según el tipo.

**2. Clase Posicion**

Representa las coordenadas (x, y) de cualquier elemento del tablero.

- Se sobrescriben los métodos equals() y hashCode() para poder comparar posiciones y usarlas en estructuras como Set.

**3. Clase Figura**

Asocia un TipoFigura con una Posicion concreta. Cada pieza del tablero (Rey, Torre, etc.) es una instancia de esta clase.

**4. Clase Movimiento**

Representa un movimiento individual: qué figura se mueve y a qué posición (donde estaba el hueco).

**5. Clase Tablero**

Es la estructura central que representa el estado del tablero en un momento dado.

**Atributos:**

- matriz: matriz de 5x5 con todas las figuras.
- hueco, rey, objetivo: posiciones especiales.
- historial: lista de los movimientos realizados hasta el momento.
- heuristica: distancia del Rey al objetivo (usada por los algoritmos).
- coste: número de pasos/movimientos realizados.

**Métodos:**

**clonar ()**

Crea una copia profunda del tablero actual. Se utiliza para generar los estados sucesores sin modificar el original.

**cargarDesdeFichero(String ruta)**

Lee el fichero de entrada y construye el tablero con base en los caracteres M, H, R, O, A, T.

**calcularHeuristica()**

Calcula la distancia euclídea entre el Rey y el Objetivo. Es la función heurística h(n) usada por A* y Escalada.

**esObjetivo()**

Retorna true si el Rey está en la misma posición que el Objetivo.

**generarSucesores()**

Genera los posibles movimientos válidos desde el estado actual:

- Busca las figuras adyacentes al hueco.
- Verifica si pueden moverse según las reglas del ajedrez (Rey: 1 casilla; Torre: línea recta; Alfil: diagonal).
- Genera nuevos tableros con los movimientos válidos aplicados.

**movimientoValido(Figura f, Posicion destino)**

Define las reglas específicas de movimiento para cada tipo de figura.

**6. Algoritmo de búsqueda: escalada (Tablero inicial)**

Implementa **escalada simple**:

- Genera todos los sucesores.
- Ordena por la mejor heurística.
- Si encuentra una mejora (h(n) más baja), avanza a ese estado.
- Se detiene si no mejora o alcanza el objetivo.

Registra el número de nodos generados.

**7. Algoritmo de búsqueda: aEstrella(Tablero inicial)**

Implementa el algoritmo **A***:

- Usa una cola con prioridad (PriorityQueue) ordenada por f(n) = g(n) + h(n).
- Evita ciclos mediante una tabla de posiciones ya visitadas (Set cerrados).
- Al encontrar el objetivo, retorna el tablero final con el historial de movimientos.

También muestra el número total de nodos generados.

**8. Método main()**

Coordina todo el proceso:

1. Carga el tablero desde PASOALREY.TXT.
2. Ejecuta el algoritmo de **escalada simple**, imprime:
    - Tiempo
    - Técnica

- Movimientos realizados

3. Ejecuta el algoritmo **A***, imprime lo mismo.

Los movimientos se expresan como coordenadas estilo ajedrez: (b,3).