UNIVERSIDAD
TECNOLÓGICA NACIONAL

INTELIGENCIA
ARTIFICIAL

TP I : Búsqueda

IZAGUIRRE, Ezequiel
ezequielmizaguirre@gmail.com

MOLINA VAN LEEUWEN, Jeremías
jere.movale@gmail.com

2024 – Grupo **3**

Regional Santa Fe

RESUMEN

Este informe presenta la conceptualización de un agente inteligente basado en objetivos para una variante del juego **Among Us**, en el cual se asume el rol de *impostor*. Su objetivo es sabotear las operaciones de una nave espacial y asesinar (eliminar) a los tripulantes.

En la **primer sección** se describe la definición conceptual del problema y del agente: su relación con el ambiente de juego, sus percepciones y acciones correspondientes, así como la estructura de datos necesaria para representar los estados. Se establecen los valores iniciales y finales, la prueba de meta y los operadores para resolver el problema utilizando algoritmos de búsqueda.

En la **segunda sección** se muestra la implementación del agente en el lenguaje de programación Java 11 usando la librería FAIA. Se describen las clases usadas, sus relaciones y las distintas estrategias de búsqueda utilizadas para resolver el problema y generar la secuencia de acciones.

En la **tercer sección** se presentan los resultados de las corridas del programa descripto en la sección anterior con cada estrategia, algunas conclusiones y para acabar un resumen. Veremos que la capacidad de resolución del agente depende enormemente de la estrategia usada.

ÍNDICE

1 DEFINICIÓN CONCEPTUAL

1.1 • Introducción

El objetivo del trabajo práctico es implementar un agente *inteligente* (reactivo y proactivo) capaz de jugar y ganar una versión simplificada del videojuego **Among Us**, en el rol de *impostor*, en base a las reglas descriptas en el enunciado. El juego se considera resuelto cuando no quedan jugadores con el rol *tripulante* por asesinar y todos los sabotajes han sido realizados.

En la siguiente sección se describe las características del ambiente y el agente, los estados de interés y el problema a resolver.

1.2 • Características del ambiente

El **mapa** de juego, donde nuestro agente y los tripulantes deben desenvolverse, está compuesto de una serie de secciones conectadas entre sí a través de pasillos, puertas y tuberías, como se muestra en la siguiente imagen.

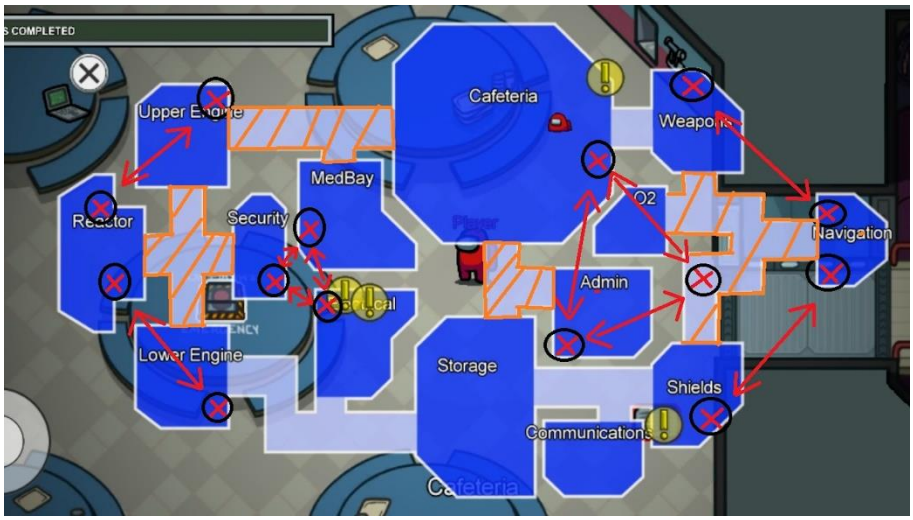


Ilustración 1: Mapa de Among Us donde las 'x' y sus flechas representan las conexiones a través de tuberías.

Nuestro agente, que definiremos como un **agente basado en objetivos**, representará este mapa usando como estructura un **grafo no dirigido**. Esto es apropiado porque facilita su lectura y además permite aplicar algoritmos de toma decisiones de navegación, como buscar el camino más corto entre 2 secciones, por ejemplo.

Usando grafos, el mapa queda representado como sigue.

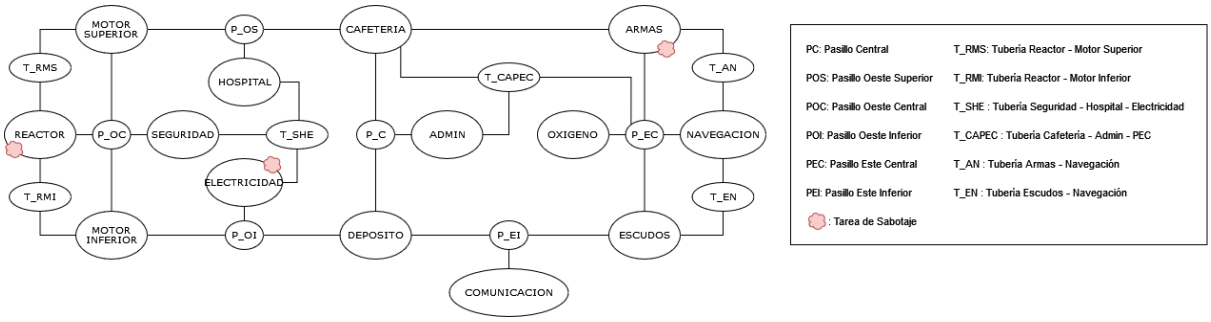


Ilustración 2: Mapa representado como grafo, con las leyendas a la derecha.

El ambiente puede **clasificarse** de la siguiente forma.

- **Parcialmente observable**

Nuestro agente solo es capaz de percibir el estado del ambiente en el cual se encuentra. Aunque posee capacidades extrasensoriales, estas solo pueden ser activadas esporádicamente.

- **Determinísta**

Se puede trazar una correspondencia directa entre las acciones del agente y sus consecuencias en el entorno. Por ejemplo: si asesina a un tripulante, habrá menos tripulantes vivos; si se desplaza, cambiará de ambiente.

- **Secuencial**

Las acciones pasadas del agente tienen consecuencias en sus acciones futuras.

- **Estático**

Es estático porque el ambiente no evoluciona en el tiempo entre que el agente toma una decisión y ejerce una acción.

- **Discreto**

Es discreto porque el agente posee un conjunto discreto de acciones y percepciones que puede obtener del ambiente, ya que el sistema evoluciona en cada salto o ciclo de percepción y no de forma continua.

- **Agente individual**

Podría considerarse un ambiente multiagente si los tripulantes tuviesen algún grado de autonomía o reactividad. Sin embargo, en el trabajo práctico se propone que sólo el impostor sea conciente de su entorno y el movimiento de los tripulantes sea aleatorio. Por tanto, existe un único agente y un único objetivo.

1.3 • Percepciones

Para poder resolver el problema, es necesario obtener la siguiente información de nuestro entorno.

- Ambiente actual.
- Destinos alcanzables desde el ambiente actual.
- Cantidad de tripulantes totales.
- Tripulantes en el ambiente actual.
- Posibilidad de sabotaje (si es o no un ambiente saboteable).
- Ubicación de todos los tripulantes (si posee sensor extrasensorial).[1]

La información restante (como energía actual, rutas del mapa, etc) es almacenada y gestionadas internamente por el agente.

1.4 • Estados

A continuación, se definen las estructuras (**clases**) de las entidades de interés.

Estas entidades son implementadas como instancias de sus clases correspondientes. El agente obtiene información de sus estados cada vez que actualiza un ciclo de percepción.

Class Impostor	
Atributo	Tipo de dato
Ambiente_actual	**Ambiente**
Ambientes_adyacentes	Lista de **Ambiente**
Ambientes_conocidos	Lista de pares
Tripulantes_adyacentes	Lista de **Tripulante**
Tripulantes_vivos	Lista de pares
Tripulantes_muertos	Lista de pares
Tripulantes_totales	Entero constante
Sabotajes_restantes	Lista de **Sabotajes**
Energía_actual	Entero
Sensor_disponible	Booleano

Tabla 1: Clase del impostor

Notas

- Ambientes_conocidos posee una lista donde cada elemento es un par entre un **Ambiente** y otro **Ambiente** al que habría que desplazarse si se deseara llegar al primero, desde la ubicación actual, con la menor cantidad de pasos. Esto puede resultar útil para seleccionar la ruta con menor gasto de energía

[1] Le permite al agente conocer el nombre de la sección donde cada uno se encuentra, pero no provee información sobre la ruta a seguir.

Tripulantes_vivos y Tripulantes_muertos poseen ambos una lista donde cada elemento es un par entre un **Tripulante** y el **Ambiente** en el que se encontraba cuando se lo vió por última vez.

Class Ambiente	
Atributo	Tipo de dato
Identificador	String o Entero
Sabotajes_disponibles	Lista de **Sabotajes**
Tipo[2]	Enumerable
Adyacentes	Lista de **Ambiente**
Tripulantes_actuales	Lista de **Tripulantes**

Tabla 2: Clase del ambiente

Notas

- Tipo acepta los valores: *sección*, *pasillo* o *tubería*.

Class Sabotaje	
Atributo	Tipo de dato
Identificador	String o Entero
Completado	Booleano

Tabla 3: Clase de sabotaje

Desde el punto de vista del agente, no hay información de interés que se pueda obtener de un tripulante, pero agregamos la clase por motivos semánticos.

Class Tripulante	
Atributo	Tipo de dato
Identificador	String o Entero

Tabla 4: Clase de tripulante

1.4.1 • Estados iniciales

Al comienzo del juego, se establecen los siguientes valores para cada entidad. Serán *aleatorios* aquellos cuyo valor queda resuelto al iniciar el juego después de aplicar una función aleatoria.

Class Impostor	
Atributo	Valor
Ambiente_actual	*aleatorio*
Ambientes_adyacentes	***Ambiente_actual**.adyacentes*
Ambientes_conocidos	***Ambientes_adyacentes***
Tripulantes_adyacentes	***Ambiente_actual**.tripulantes_actuales*
Tripulantes_vivos	***Tripulantes_adyacentes ; Ambiente_actual***
Tripulantes_muertos	*Vacio*

[2] Puede resultar interesante diferenciar los ambientes para luego realizar análisis.

Tripulantes_totales	*Valor inicial*
Sabotajes_restantes	*Valor inicial*
Energía_actual	*aleatorio [30-150]*
Sensor_disponible	*aleatorio [3-5]*

Class Ambiente	
Atributo	Valor
Identificador	*Valor inicial*
Sabotajes_disponibles	*Valor inicial*
Tipo	*Valor inicial*
Adyacentes	*Valor inicial*
Tripulantes_actuales	*aleatorio*

Class Sabotaje	
Atributo	Valor
Identificador	*Valor inicial*
Completada	*false*

1.4.2 • Estados finales

Definimos que el final del juego es el conjunto de estados donde impostor **gana**. En estos cada entidad posee los siguientes valores.

Estados ganadores

Será cualquier estado donde se cumpla simultaneamente que:

- La cantidad de tripulantes que faltan por asesinar es cero.
- La cantidad de sabotajes restantes es cero.

Los demás atributos pueden tomar cualquier valor (*indefinido*), siempre dentro de la propia lógica del juego. Por ejemplo, debido a las precondiciones de los operadores, esta claro que un agente no puede ganar con un nivel de energía negativo.

Class Impostor	
Atributo	Valor
Ambiente_actual	*indefinido*
Ambientes_adyacentes	***Ambiente_actual**.adyacentes*
Ambientes_conocidos	*indefinido*
Tripulantes_adyacentes	*Vacio*
Tripulantes_vivos	*Vacio*

Tripulantes_muertos	*Size valor inicial de* ***Tripulantes_totales***
Tripulantes_totales	*Valor inicial*
Sabotajes_restantes	*Vacio*
Energía_actual	*indefinido [>= 0]*
Sensor_disponible	*indefinido*

Class Ambiente	
Atributo	Valor
Identificador	*Valor inicial*
Sabotajes_disponibles	*indefinido*
Tipo	*Valor inicial*
Adyacentes	*Valor inicial*
Tripulantes_actuales	*Vacio*

Class Sabotaje	
Atributo	Valor
Identificador	*Valor inicial*
Completada	*true*

1.5 • Operadores

Para poder resolver el problema, es necesario que nuestro agente interactue con su entorno. Le proponemos contar con los siguientes operadores o acciones.

Quedarse() | Uso de energía: **1** *Mantiene la posición actual un ciclo*

Desplazarse(ambiente**)** | Uso de energía: **1** *Se desplaza a otra sección*

```
If ambiente_actual.adyacentes.includes(ambiente) then:
    Ambiente_actual = ambiente;
    Ambientes_adyacentes = ambiente.adyacentes;
    End;
```

Asesinar(tripulante**)** | Uso de energía: **3** *Elimina a un tripulante*

```
If tripulantes_adyacentes.includes(tripulante)
AND ambiente_actual.tipo equals (seccion OR pasillo) then:
    Tripulantes_muertos.add(tripulante);
    End;
```

Activar_sensor() | Uso de energía: **1** *Activa capacidad extrasensorial*

If sensor_disponible then:

Tripulantes_vivos.add(*all,ambientes*);

Sensor_disponible = false;

End;

Sabotear(sabotaje**)** | Uso de energía: **2** *Ejecuta un sabotaje*

If sabotajes_restantes.includes(sabotaje)

AND ambiente_actual.sabotajes_disponibles.includes(sabotaje) then:

Sabotajes_restantes.delete(sabotaje);

End;

Notas

- Como precondición, cada operador puede ser ejecutado si y solo sí el agente posee una energía restante igual o mayor a la requerida por dicha acción.

1.6 • Prueba de meta

Llegar al estado ganador involucra cumplir 3 condiciones: *meta de asesinatos, meta de sabotajes* y *meta de energía***.**

- **Meta de asesinatos**

If tripulantes_muertos.length == tripulantes_totales then: **ÉXITO**

- **Meta de sabotaje**

IF sabotajes_restantes.length == 0 then: **ÉXITO**

- **Meta de energía**

If energía_restante >= 0 then: **ÉXITO**

Luego, la **prueba de meta** devuelve un resultado exitoso cuando estan condiciones se cumplen simultáneamente.

Prueba_de_meta(estado**)**

if **meta_de_asesinatos** AND **meta_de_sabotaje** AND **meta_de_energía**

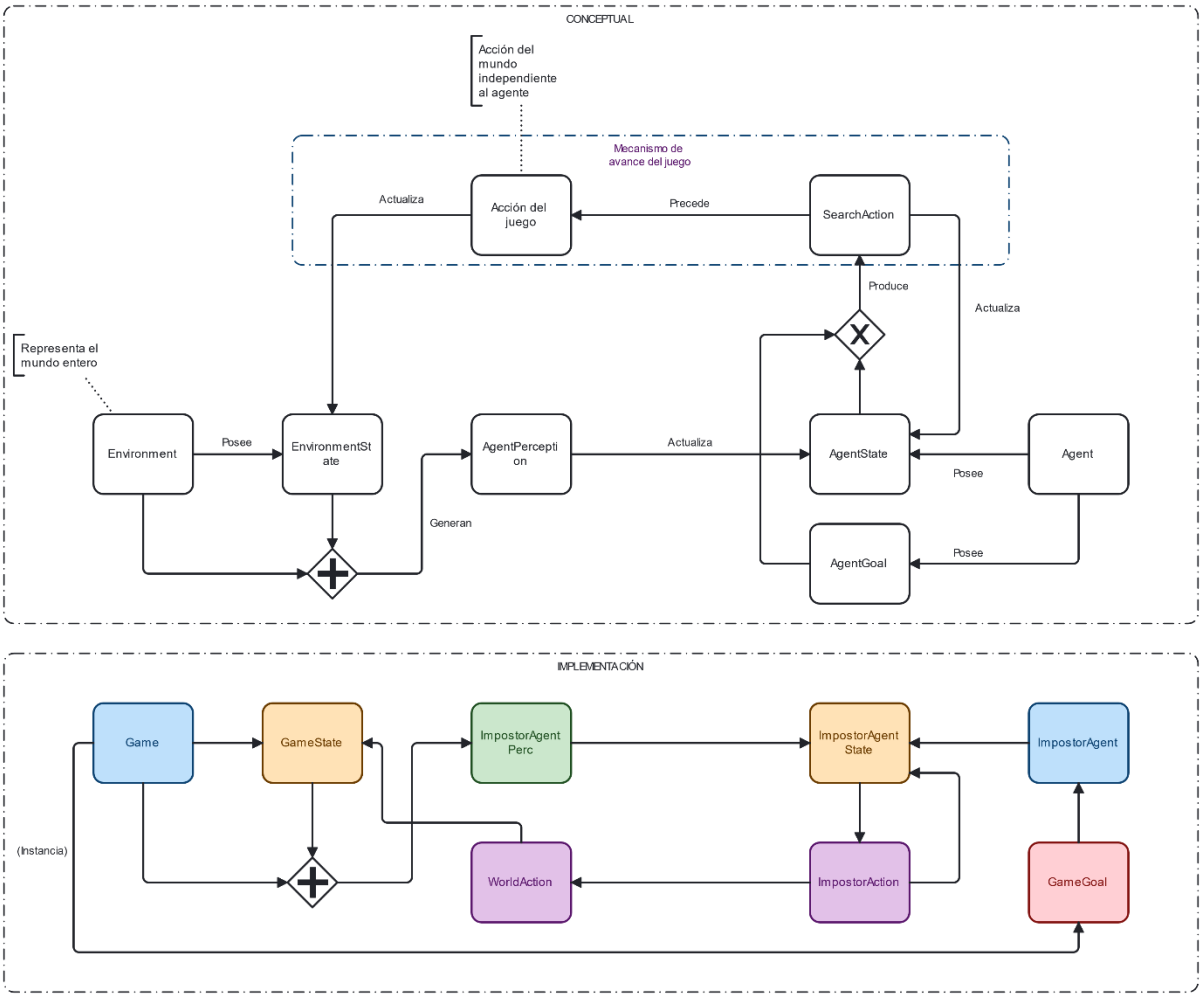
then: **ÉXITO**

2 IMPLEMENTACIÓN

2.1 • Introducción

Para implementar el agente se utilizó el lenguaje Java en su versión 11 junto con la librería FAIA desarrollada por la cátedra.

La estructura de clases y sus relaciones se muestra en la figura de abajo. El grupo encerrado en líneas punteadas en la parte de arriba agrupa las clases abstractas que sugiere la librería, mientras que abajo se encuentran nuestras implementaciones. Este conjunto forma todas las entidades del juego.



2.2 • Entidades y funciones

A continuación se detallan las entidades mostradas anteriormente.

Entidades de FAIA	Implementado por
▪ Environment	▪ Game

Game es la clase principal que representa al mundo y juego en su conjunto. Mantiene atributos estáticos y dinámicos del juego, los últimos siendo modificados por su GameState a lo largo de la simulación.

▪ EnvironmentState	▪ GameState

GameState representa el estado del mundo en un momento determinado. En su instancia se encuentran todos los atributos que varían a lo largo del juego.

▪ Agent	▪ ImpostorAgent

La clase ImpostorAgent, al igual que Game, representa al agente como entidad del juego y posee tanto atributos estáticos como dinámicos, los últimos son manejados por su estado.

▪ AgentState	▪ ImpostorAgentState

El impostorAgentState contiene toda la información que el agente va recibiendo del juego. Se entiende como una base de datos que es actualizada en cada ciclo de percepción con los valores del ImpostorAgentPerc. Se utiliza para decidir la siguiente acción a realizar.

▪ AgentPerception	▪ ImpostorAgentPerc

Las instancias de ImpostorAgentPerc son creadas por el Game y rellenadas con información del GameState en cada ciclo de percepción.

Como la capacidad sensorial del agente es limitada, solo un subgrupo de todos los atributos del mundo es entregado. En general, este es el estado de la habitación actual y la ubicación de todos los tripulantes cuando se tiene capacidad extrasensorial.

Un aspecto importante es que el agente no puede recibir esta información "así como es" desde el GameState, sino que debe ser transformada a un tipo válido en base a lo que pueden observar sus sensores. Por ejemplo, si bien cada tripulante se representa en el GameState como una instancia de una clase (**CrewMember**) junto con sus atributos, el agente no observa la "clase", sino que su percepción le entrega un valor textual indicando el nombre del tripulante y su ubicación, también textual. Toda esta lógica es manejada por el Game en su método correspondiente.

- AgentGoal
- GameGoal

GameGoal contiene la lógica de decisión acerca del objetivo del agente y de cuándo este se alcanza o no. Notar que el juego puede estar en 3 estados:

- En curso
- Ganado
- Perdido

Mientras se esté en curso, el agente seguirá buscando acciones que realizar. El estado *perdido* se alcanza si ya no queda energía disponible y los objetivos, o prueba de meta, no han sido cumplidos.

En nuestra implementación, esta clase es instanciada por Game.

- SearchAction
- ImpostorAction

ImpostorAction no es en sí una clase sino un conjunto de ellas (*Kill, Sabotage, Move)*, todas descendientes de SearchAction y que forman el abanico de acciones del agente. Se decidió eliminar las acciones *Quedarse* y *Activar Sensor* durante la implementación por motivos de diseño.

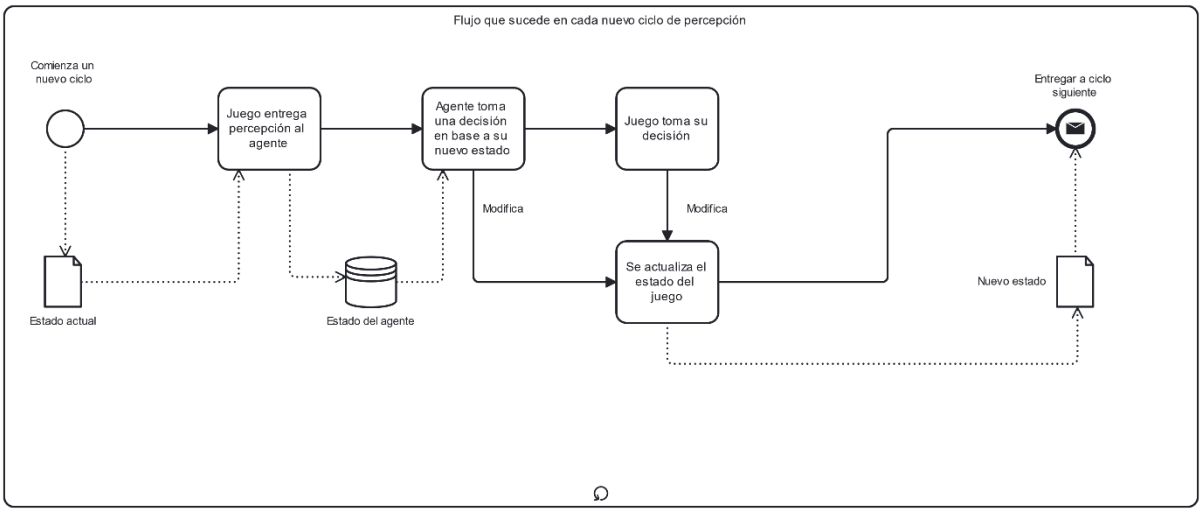
Como se explica a continuación, entregarle información extrasensorial no es una acción del agente sino algo que maneja el juego automáticamente.

- Acción del juego
- WorldAction

La acción del juego no es una clase base de FAIA, sino una que se agregó a efectos de enfatizar que el avance de la simulación no ocurre solo por efectos del agente sino también por otras acciones ajenas a él. En cada ciclo, se debe incrementar el tiempo, reubicar tripulantes, activar sensores, entre otros. Estas son modificaciones al mundo propias del juego y controladas por WorldAction.

El método de WorldAction se ejecuta luego de que el agente haya tomada su decisión y es el mecanismo de avance hacia la siguiente percepción.

Para expandir en WorldAction y entender el avance del juego, se muestra a continuación un diagrama de flujo que muestra qué ocurre en cada ciclo.



Vemos que primero se entrega la percepción del ciclo actual al agente. Éste actualiza su estado y ejecuta una acción. Luego, se ejecuta la WorldAction. Ambas acciones modifican el estado del mundo, cuyo resultado será un nuevo estado de juego para el siguiente ciclo.

- Amongus
- Main

Finalmente tenemos la clase Amongus que contiene el método *Main()* para inicializar el juego con los parámetros definidos en Game y la estrategia en ImpostorAgent.

2.2.1 • Atributos

A continuación se listarán los atributos que posee cada clase y se detallarán los necesarios. Durante la implementación se tuvieron que hacer modificaciones a la lista definida en el modelo conceptual, ya que se fueron encontrando problemas y/o situaciones donde era necesario repensar cómo funciona el mundo.

- **Game**

```java
//Representa el juego en su conjunto
public class Game extends Environment map = null
{
    public final HashMap<String,Room> map = new HashMap<>();
    public final HashMap<String, Sabotage> sabotages = new HashMap<>();

    private final ImpostorAgent agent;
    private final GameState state;
    private final GameGoal goal;
    private final List<GameState> gameHistory = new ArrayList<>();

    // -- Parámetros de juego
    public static int MAX_ENERGY = 150;
    public static int MIN_ENERGY = 30;
    public static int MAX_CREW = 8;        //Con Costo uniforme: Hasta 6 | Con profundidad: 0..* | Con Ava
    public static int MIN_CREW = 8;
    public static int MAX_CREW_STEP_TIME = 3;           //Tiempo máximo para moverse
    public static int MIN_CREW_STEP_TIME = 1;           //Tiempo mínimo para moverse
    public static int MAX_AGENT_SENSOR_STEP_TIME = 5;   //Tiempo máximo para tener el sensor
    public static int MIN_AGENT_SENSOR_STEP_TIME = 3;   //Tiempo mínimo para tener el sensor
```

Se observa que se encuentran todos los aspectos que conciernen al juego.

- Map posee todas las habitaciones, formadas por una clase Room.
- gameHistory es la lista de estados que se atraviesan a lo largo de una simulación (se utiliza en el parte gráfica).
- Tenemos parámetros para definir características del juego, como rango de energía del agente, cantidad de tripulantes, etc.

- **GameState**

```java
//El GameState contiene el estado completo del juego
public class GameState extends EnvironmentState
{
    private static Game environment;

    //Atributos del mundo
    private static HashMap<String,Room> map;
    private static HashMap<String,Sabotage> sabotages;
    private static HashMap<String, CrewMember> crews = new HashMap<>();
    private final List<RoomState> roomStates = new ArrayList<>();
    private final List<CrewMemberState> crewStates = new ArrayList<>();
    private Long gameTime;

    //Atributos del agente
    private Room agentRoom;
    private Long agentEnergy;
    private Long agentSensorLastTime; //Cuando estuvo activo por última vez

    //Se activa cuando el juego debe darle información extrasensorial al agente (en la sgte percepción)
    private boolean omniscientAgent;
```

Acá se encuentran todas las clases que guardan los estados de todas las entidades. Observar que tenemos tanto un Room y un RoomState, como también un CrewMember y CrewMemberState.

- **ImpostorAgent**

```
public class ImpostorAgent extends SearchBasedAgent
{
    private final ImpostorAgentState myState;
    private final GameGoal myGoal;
    private final Vector<SearchAction> myActions = new Vector<>();
```

- **ImpostorAgentPerc**

```
//Como ve el mundo nuestro agente
public class ImpostorAgentPerc extends Perception
{
    private final String currentRoomSensor;
    private final List<String> cardinalSensor = new ArrayList<>(5);    //[O,N,E,S]
    private final List<String> crewPresentSensor = new ArrayList<>();
    private final Long energySensor;
    private final String sabotageSensor;
    private final Long gameTimeSensor;

    //Hay información extrasensorial?
    private boolean extraInfoAvail;

    //Lista de pares entre un tripulante y su ubicación (extrasensorial).
    private final List<Pair<String,String>> extraSensor = new ArrayList<>();
```

Como se explicó antes, esta clase guarda los atributos que percibe el agente en un formato adecuado.

Los distintos sensores pueden:

- Descubrir la habitación actual.
- Descubrir habitaciones vecinas.
- Descubrir lugares saboteables.
- Descubrir tripulantes en la habitación.
- Medir la energía actual.
- Recibir información de la ubicación de todos los tripulantes.

- **ImpostorAgentState**

```java
//Subconjunto de datos del mapa que el agente conoce y guarda
public class ImpostorAgentState extends SearchBasedAgentState
{
    private AgentRoomState currentRoom;

    //Habitación de la que vengo
    private AgentRoomState previousRoom;

    //Habitaciones conocidas / Submapa del agente → Nota: Actualmente se asume que
    private final HashMap<String,AgentRoomState> knownRooms = new HashMap<>();

    private final ArrayList<String> crewKilled = new ArrayList<>();

    //Nombre del tripulante vivo / última ubicación conocida - tiempo de avistamient
    private final HashMap<String, Pair<String,Long>> aliveCrew = new HashMap<>();

    private final static ArrayList<String> requiredSabotages = new ArrayList<>();
    private final ArrayList<String> doneSabotages = new ArrayList<>();

    private Long gameTime;
    private Long energy;

    /*
        DecisionCost → Marca el 'costo' de todas las decisiones que se han tomado.
        NO es igual al costo de energía. Esto se usa en costo uniforme para parametr
        Es cambiado por las acciones del agente.
    */
    private Long decisionCost;
    private boolean sensorAvailable;
```

- **GameGoal**

```java
public class GameGoal extends GoalTest
{
    public GameState gameState;

    public GameGoal(GameState gameState) {
        this.gameState = gameState;
    }

    @Override
    public boolean isGoalState(AgentState agentState)
    {
        var agent = (ImpostorAgentState) agentState;

        //Prueba de meta final
        boolean crewKilledTest = agent.getCrewKilled().size() == gameState.getCrews().size();
        boolean doneSabotageTest = agent.getDoneSabotages().size() == gameState.getSabotages().size();

        return crewKilledTest && doneSabotageTest;
    }

    //Si no es estado final y se quedó sin energía, perdió
    public boolean agentFailed(AgentState agentState)
    {
        return !this.isGoalState(agentState) && gameState.getAgentEnergy() == 0;
    }


}
```

2.3 • Árbol de búsqueda

Dado que el operador de meta implica asesinar a todos los tripulantes y sabotear las habitaciones, es necesario que el agente conozca la ubicación de cada sabotaje y tripulante para que pueda crear el árbol y alcanzar el estado final. Si no se tuviese esta información, nunca se podrían completar todas las tareas (porque el árbol no sabe dónde buscarlas).

El agente conoce desde el principio cuáles lugares son saboteables, pero **no** sabe la habitación actual de los tripulantes en todo momento. En nuestra implementación se optó por distribuir las ubicaciones desconocidas de manera aleatoria. Así, el árbol puede encontrar una solución y decidir la siguiente acción. Cuando el agente se encuentre con una habitación saboteable o un tripulante, la percepción le entregará esta información y podrá realizar la acción correspondiente. Es decir, si bien le estamos llenando el mapa de ubicaciones falsas, toda primera acción que se tome se hará en base a la información real.

Por otro lado, a la hora de armar el árbol de decisión, no se debe considerar que la energía decrementa con cada nodo (solo en los de primer nivel). Esto tiene 2 justificaciones:

- Si se consumiese energía se podría llegar a una situación donde el agente cree que perdió pero en realidad puede ganar porque el arbol de búsqueda no puede contemplar el movimiento aleatorio de los tripulantes en cada ciclo.
- No tiene sentido que en una ejecución "imaginaria" de las acciones se consume energía.
- Se considera solo en el primer nivel porque ahí se encuentra la siguiente acción que se tomaría. Si se detecta que no se posee la energía suficiente, la simulación termina y el agente pierde.

2.4 • Costos, acciones y heurísticas

Para la ejecución del algoritmo se propusieron 5 estrategías de búsqueda que se dividen en: **no informadas** e **informadas**.

Estrategias no informadas

Se implementó una estrategía **en amplitud** y otra **en profundidad**. Ninguna de las 2 utiliza información del dominio para buscar la siguiente acción, pero aún así difieren enormemente en su viabilidad. Como se mostrará en la tercera sección, cuando las acciones se ordenan de forma que *asesinar* aparezca como primer opción (seguido de *sabotear* y *desplazarse)*, es posible encontrar una siguiente acción con la estrategia en amplitud con un bajo costo computacional y de memoria porque todas las ramas de decisión llevan rápidamente al estado ganador. La capacidad de esta estrategia de resolver es para un alto número de tripulantes, pero require potencialmente mucha energía.

Por otro lado, aplicar en profundidad resulta impráctico dado que consume una alta cantidad de memoria. Como son muchas decisiones a tomar (nodos a expandir) y se requiren muchos pasos para llegar a la solución, esta estrategia no sería conveniente casi en ningún

caso. Sólo sería viable como un prototipo para una simulación reducida en habitaciones y tripulantes.

Estrategias informadas

La primera estrategia informada es la de **costo uniforme**, que require definir el costo de un nodo. Esto se detalló con una clase AgentStateCost que implementa IStepCostFunction.

```
public class AgentStateCost implements IStepCostFunction {

    /**
     * This method calculates the cost of the given NTree node.
     */
    @Override
    public double calculateCost(NTree node)
    {
        ImpostorAgentState agentState = (ImpostorAgentState) node.getAgentState();

        return agentState.getDecisionCost();
    }
}
```

Como se ve, el estado del agente posee un "costo de decisión" asociado al atributo *decision_cost* que representa el costo de todas las acciones tomadas, desde el principio, para llegar a ese estado. Es decir, este valor se calcula como el costo de decisión del estado anterior más el de la acción que llevó al estado actual. Cada acción entonces tiene un costo, que será mayor cuanto más desfavorable sea para alcanzar rápidamente el estado final.

Acción	Costo
Asesinar	1
Sabotear	2
Desplazarse	5

En la próxima sección se verá que con esta estrategía se puede resolver (con un uso estándar de memoria) un juego con hasta 6 tripulantes.

La siguiente estrategia utiliza fue la **Avara** que require una función heurística o estimativa. Se muestra la clase AgentHeuristic que implementa IestimatedCostFunction.

```java
public class AgentHeuristic implements IEstimatedCostFunction
{
    /*
        Es un número en formato VVDDSS (Asume que el valor máximo de c/u es 99)
        Donde:
        -   VV : Tripulantes vivos
        -   DD : Sabotajes restantes
        -   SS : Distancia más cercana a un tripulante
        Al finalizar, el número quedaría 000000 → 0;

    */
    @Override
    public double getEstimatedCost(NTree node)
    {
        ImpostorAgentState agentState = (ImpostorAgentState) node.getAgentState();

        if(agentState.getCurrentRoom() == null) return Integer.MAX_VALUE;

        int sabotages = agentState.getRequiredSabotages().size() - agentState.getDoneSabotages().size();
        int alive = agentState.getAliveCrew().size();
        int distance = alive > 0 ? closest(agentState.getCurrentRoom(), agentState.getKnownRooms(), new ArrayList<>()) : 0;

        return (alive*10000 + sabotages*100 + distance);

    }
```

Se define un número de 6 digitos VVDDSS donde los 2 primeros digitos más significativos (VV) cuentan el número de tripulantes vivos, los 2 siguientes (DD) el número de sabotajes restantes y los últimos (SS) la distancia del agente al tripulante más cercano, medido en número de habitaciones y pasillos de distancia.

Con este arreglo, encontrar una acción que reduzca la distancia disminuye el número, pero encontrar primero un sabotaje lo disminuye más rapido, y todavía más rápido si se asesina. Esto significa que, como en esta estrategia se busca minimizar el valor estimativo, siempre se optará por realizar la acción más óptima en cualquier momento, saltándose ramas enteras y disminuyendo el tamaño del arbol.

Los resultados dados en la sección 3 demuestran que se puede mejorar mucho la capacidad de cálculo del agente usando esta función.

Por último, tenemos la estrategia $\mathbf{A}^*$ que combina tanto el costo de decisión como la función estimativa y trata de minimizar este valor. Los resultados demuestran que no se mejora sustancialmente la capacidad en comparación con Avara.

3 RESULTADOS

3.1 • Introducción

En esta sección se presentan los resultados de la implementación definida en la sección anterior para las 5 estrategias de búsqueda mencionadas:

- En profundidad
- En amplitud
- De costo uniforme
- Avara
- A*

Todas las ejecuciones mostradas se realizan con un tamaño máximo de memoria (heap) de 2 GB y 4 núcleos a 4 GHZ de CPU (aunque se utiliza una menor capacidad en la VM).

3.2 • Estrategias no informadas

En profundidad

Esta estrategia puede encontrar las soluciones para árboles de juegos con muchos tripulantes. Para aplicar esta búsqueda es muy importante considerar el orden de acciones del agente. Elegir un órden equivocado (por ejemplo, poniendo *desplazarse* primero) haría que el árbol se expandiese infinitamente hacia abajo, ya que siempre expande el nodo más a la izquierda o primero en la lista.

Si las acciones del agente se ordenan de forma apropiada (*asesinar*, *sabotear*, *desplazarse)* es posible encontrar una solución con poco gasto de CPU y memoria. Sin embargo, como no se tiene información acerca de qué tanto se acerca la elección al objetivo, se require mucha energía para poder aplicarse y puede sufrir de bucles que tornan su ejecución inútil.

Por ejemplo, abajo se enseña el último estado generado en un juego con las siguientes condiciones (el tiempo se mide en ciclos de percepción):

- **Tripulantes** : 12
- **Energía inicial** : 245
- **Tiempo entre información extrasensorial**: Max 8, Min 5
- **Tiempo entre movimiento de tripulantes**: Max 3, Min 1

```
==Juego avanza==
ERROR: The simulation has finished, but the agent has not reached his goal.

--□ltimo estado--
--Mundo | Tiempo: 245--
□D□nde est□ cada tripulante?:
Me llamo: Tripulante #0 y estoy en: Pasillo este inferior <-- MUERTO
Me llamo: Tripulante #1 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #2 y estoy en: Cafeter□a
Me llamo: Tripulante #3 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #4 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #5 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #6 y estoy en: Pasillo este centro <-- MUERTO
Me llamo: Tripulante #7 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #8 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #9 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #10 y estoy en: Pasillo este centro <-- MUERTO
Me llamo: Tripulante #11 y estoy en: Pasillo este centro <-- MUERTO

--Estado | Tiempo: 244--
Me encuentro en: Pasillo este centro
Vengo de: Armas
Tripulantes presentes: 0
Tripulantes asesinados: 11
Puedo sabotear?: NO
Sabotajes hechos: 1
Sabotajes totales: 3
Mi energ□a restante: 0
Tengo capacidad extrasensorial? : NO
```

Ilustración 3: Resultado con profundidad

Vemos que el agente ha logrado eliminar a 11 tripulantes, quedando el **Tripulante #2** vivo en Cafetería. También se ha realizado un sabotaje con éxito. Sin embargo, el juego termina abruptamente debido a que el impostor se ha quedado sin energía para continuar.

Debido a que el uso de memoria crece linealmente, la cantidad de nodos creados no es un impedimento y el programa es capaz de encontrar acciones hasta con más de 30 tripulantes!

En amplitud o anchura

Esta estrategia, como se explico en la sección anterior, resulta complicada por su alto consumo de memoria. Esto se debe a que primero se expanden todos los nodos de un mismo nivel antes de continuar con los siguientes. Como esto implica un crecimiento exponencial en el uso de memoria, se requiere que la solución puede encontrarse en solo unos cuántos niveles o movimientos, en escenarios con muchos tripulantes este no es el caso porque tenemos más de 14 habitaciones y 50 interconexiones entre ellas. Sin embargo, en juegos reducidos funciona razonablemente bien gracias a que encuentra la solución más óptima en términos de acciones.

Por ejemplo, abajo se enseña el último estado generado en un juego con las siguientes condiciones (el tiempo se mide en ciclos de percepción):

- **Tripulantes** : 5
- **Energía inicial** : 112
- **Tiempo entre información extrasensorial**: Max 8, Min 5
- **Tiempo entre movimiento de tripulantes**: Max 3, Min 1

```
==Juego avanza==
Agent has reached the goal!

--□ltimo estado--
--Mundo | Tiempo: 34--
□□□nde est□ cada tripulante?:
Me llamo: Tripulante #0 y estoy en: Motor inferior <-- MUERTO
Me llamo: Tripulante #1 y estoy en: Pasillo este inferior <-- MUERTO
Me llamo: Tripulante #2 y estoy en: Pasillo central <-- MUERTO
Me llamo: Tripulante #3 y estoy en: Cafeter□a <-- MUERTO
Me llamo: Tripulante #4 y estoy en: Dep□sito <-- MUERTO

--Estado | Tiempo: 33--
Me encuentro en: Pasillo central
Vengo de: Cafeter□a
Tripulantes presentes: 0
Tripulantes asesinados: 5
Puedo sabotear?: NO
Sabotajes hechos: 3
Sabotajes totales: 3
Mi energ□a restante: 78
Tengo capacidad extrasensorial? : NO
```

Ilustración 4: Resultado con anchura

Vemos que el tripulante completa su objetivo en 34 pasos. Al principio se detectó una demora de unos 10-15 segundos en obtener la primera acción, que luego fue disminuyendo conforme el agente se acercaba al objetivo.

Este resultado diferencia a esta estrategia de la anterior, que si bien sólo funciona en escenarios pequeños, es capaz de ganar con muy poco uso de energía y sin entrar en bucles.

Podemos realizar un cálculo estimativo de cuántos nodos se deberían mantener en memoria para encontrar el primer movimiento.

Considerando un juego con 5 tripulantes y una distancia promedio a cada objetivo de 3, se requerirían al menos 32 niveles en el arbol (5*3 acciones para alcanzar a cada tripulante, 5 para matarlos, 3*3 para alcanzar cada habitación saboteable y 3 para sabotearlas). Dadas 3 acciones para el tripulante, eso implican hasta 3^{32} nodos en el estado final.

En la práctica este número es mucho menor debido a caminos cerrados o nodos duplicados. Como se mostró en el ejemplo, gracias a un uso inteligente del método *equals()*

se puede llegar a obtener una solución factible. Pero incluso en este caso bastante reducido vemos que la cantidad potencial de nodos a crear es inviable. Además, recordemos que la acción *desplazarse* en realidad se divide en una acción por cada destino alcanzable desde la habitación actual.

Las ejecuciones del programa con esta estrategia para juegos con más de 5 tripulantes no arrojan ningún resultado y terminan con el cierre abrupto y un mensaje de **OutOfMemoryError**.

3.3 • Estrategias informadas

Como se vió, las estrategias no informadas funcionan bastante bien en escenarios reducidos, o en grandes donde el uso de energía del agente no es un problema. Las estrategias informadas tratarán de enmendar ambos casos y ofrecer soluciones para escenarios de tamaño intermedio con un buen uso de energía.

De costo uniforme

En la sección anterior se definió la función de costo como un *costo de decisión* asociado a cada nodo. A continuación se muestra una ejecución con las condiciones:

- **Tripulantes** : 6
- **Energía inicial** : 141
- **Tiempo entre información extrasensorial**: Max 8, Min 5
- **Tiempo entre movimiento de tripulantes**: Max 3, Min 1

```
==Juego avanza==
Agent has reached the goal!

--□ltimo estado--
--Mundo | Tiempo: 25--
□□□nde est□ cada tripulante?:
Me llamo: Tripulante #0 y estoy en: Reactor <-- MUERTO
Me llamo: Tripulante #1 y estoy en: Motor inferior <-- MUERTO
Me llamo: Tripulante #2 y estoy en: Pasillo oeste centro <-- MUERTO
Me llamo: Tripulante #3 y estoy en: Pasillo central <-- MUERTO
Me llamo: Tripulante #4 y estoy en: Pasillo central <-- MUERTO
Me llamo: Tripulante #5 y estoy en: Pasillo oeste inferior <-- MUERTO

--Estado | Tiempo: 24--
Me encuentro en: Armas
Vengo de: Cafeter□a
Tripulantes presentes: 0
Tripulantes asesinados: 6
Puedo sabotear?: NO
Sabotajes hechos: 3
Sabotajes totales: 3
Mi energ□a restante: 116
Tengo capacidad extrasensorial? : SI
```

Ilustración 5: Resultado con costo uniforme

El agente alcanza el objetivo en solo 25 pasos. Esto era esperaba ya que cada acción tiene un costo asociado.

Si bien esta estrategia presenta una mejora sobre la búsqueda en amplitud, el cambio no es significativo. Aumentar la cantidad de tripulantes a 7 arroja **OutOfMemoryError** en la mayoría de situaciones.

La razón detrás es que este algoritmo debe realizar cálculos adicionales para obtener el nodo con el menor coste de **todos** los nodos actuales. Sumado lo anterior al hecho de que el número de nodos creados con esta estrategia está entre los de profundidad y anchura, se require bastante CPU y memoria.

La conclusión es que este algoritmo es preferible sobre anchura pero no es el definitivo.

Avaro o codiciosa

La búsqueda avara usa un número estimativo de 6 dígitos definido en la sección anterior. De entrada esta estrategia debería ser más optima que la anterior puesto que se cumple que el valor de la heurística en el nivel **k** debe ser menor al de cualquiera en el nivel **k – 1**, siempre que se haya eligido expandir el nodo con el menor valor en **k – 1**. Esto simplifica la búsqueda del siguiente nodo y lleva a árbol más pequeños y rápidos.

Por ejemplo, abajo se enseña el último estado generado en un juego con las siguientes condiciones (el tiempo se mide en ciclos de percepción):

- **Tripulantes** : 12
- **Energía inicial** : 88
- **Tiempo entre información extrasensorial**: Max 8, Min 5
- **Tiempo entre movimiento de tripulantes**: Max 3, Min 1

```
==Juego avanza==
Agent has reached the goal!

--□ltimo estado--
--Mundo | Tiempo: 55--
□D□nde est□ cada tripulante?:
Me llamo: Tripulante #0 y estoy en: Pasillo central <-- MUERTO
Me llamo: Tripulante #1 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #2 y estoy en: Pasillo oeste superior <-- MUERTO
Me llamo: Tripulante #3 y estoy en: Cafeter□a <-- MUERTO
Me llamo: Tripulante #4 y estoy en: Pasillo oeste superior <-- MUERTO
Me llamo: Tripulante #5 y estoy en: Pasillo oeste superior <-- MUERTO
Me llamo: Tripulante #6 y estoy en: Pasillo central <-- MUERTO
Me llamo: Tripulante #7 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #8 y estoy en: Dep□sito <-- MUERTO
Me llamo: Tripulante #9 y estoy en: Pasillo este centro <-- MUERTO
Me llamo: Tripulante #10 y estoy en: Pasillo este centro <-- MUERTO
Me llamo: Tripulante #11 y estoy en: Armas <-- MUERTO

--Estado | Tiempo: 54--
Me encuentro en: Reactor
Vengo de: Pasillo oeste centro
Tripulantes presentes: 0
Tripulantes asesinados: 12
Puedo sabotear?: NO
Sabotajes hechos: 3
Sabotajes totales: 3
Mi energ□a restante: 33
Tengo capacidad extrasensorial? : NO


BUILD SUCCESSFUL (total time: 30 seconds)
```

Ilustración 6: Resultado con avaro

Vemos que esta estrategia aumenta considerablemente el número de tripulantes manejables por el agente.

Para analizar la cantidad de movimientos mínimos necesarios en un juego de este estilo, vamos a considerar que la distancia promedio a cada objetivo es de 3. Entonces los movimientos son:

- 12*3 para alcanzar a cada tripulante.
- 12 para asesinarlos
- 3*3 para llegar a las habitaciones saboteables.

- 3 para sabotear.

Para un total de **60** movimientos. Como el agente necesitó **55** para ganar, la conclusión es que el algoritmo también es muy bueno encontrando la solución óptima. Aún asi, recordemos que esta estrategia no garantiza optimalidad.

A* o estrella

Para finalizar, tenemos una estrategia que combina avaro con costo uniforme. Debido a que el número de la heurística es mucho mayor al de cualquier coste de decisión, no se espera que este algoritmo mejore mucho la situación sobre el anterior.

Abajo se enseña el último estado generado en un juego con las siguientes condiciones (el tiempo se mide en ciclos de percepción):

- **Tripulantes** : 12
- **Energía inicial** : 126
- **Tiempo entre información extrasensorial**: Max 8, Min 5
- **Tiempo entre movimiento de tripulantes**: Max 3, Min 1

```
==Juego avanza==
Agent has reached the goal!

--□ltimo estado--
--Mundo | Tiempo: 57--
□D□nde est□ cada tripulante?:
Me llamo: Tripulante #0 y estoy en: Seguridad <-- MUERTO
Me llamo: Tripulante #1 y estoy en: Pasillo oeste centro <-- MUERTO
Me llamo: Tripulante #2 y estoy en: Motor superior <-- MUERTO
Me llamo: Tripulante #3 y estoy en: Motor superior <-- MUERTO
Me llamo: Tripulante #4 y estoy en: Pasillo oeste superior <-- MUERTO
Me llamo: Tripulante #5 y estoy en: Pasillo oeste inferior <-- MUERTO
Me llamo: Tripulante #6 y estoy en: Cafeter□a <-- MUERTO
Me llamo: Tripulante #7 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #8 y estoy en: Armas <-- MUERTO
Me llamo: Tripulante #9 y estoy en: Motor inferior <-- MUERTO
Me llamo: Tripulante #10 y estoy en: Seguridad <-- MUERTO
Me llamo: Tripulante #11 y estoy en: Pasillo oeste centro <-- MUERTO

--Estado | Tiempo: 56--
Me encuentro en: Reactor
Vengo de: Pasillo oeste centro
Tripulantes presentes: 0
Tripulantes asesinados: 12
Puedo sabotear?: NO
Sabotajes hechos: 3
Sabotajes totales: 3
Mi energ□a restante: 69
Tengo capacidad extrasensorial? : SI


BUILD SUCCESSFUL (total time: 7 seconds)
```

*Ilustración 7: Resultado con A**

Vemos que el resultado es prácticamente igual al caso anterior. Se resolvió el problema con 56 movimientos, que, de acuerdo con este algoritmo, es el número mínimo de movimientos necesarios.

Sin embargo, el comportamiento se vuelve errático en juegos con 12 tripulantes en adelante. A veces el algoritmo encuentra rápidamente la solución y otras veces se queda colgado. Esto depende de las condiciones iniciales, particularmente de donde esté posicionado el agente y cada tripulante, porque ello influye en el tamaño del árbol de búsqueda. Cuando existe una distancia muy grande entre estos, se empieza a tener problemas de memoria.

3.4 • Implementación de la Interfaz Gráfica

Con el objetivo de proporcionar una visualización clara y dinámica del comportamiento del agente, su ambiente y la evolución de su estado, se diseñó e implementó una interfaz gráfica de usuario (GUI). Esta interfaz fue desarrollada utilizando Java Swing, una biblioteca que ofrece un conjunto de componentes gráficos para interfaces de usuario, lo cual ayudó a respaldar la lógica existente del proyecto.

Para una gestión eficiente de los estados, se optó por almacenar cada uno en una lista (**gameHistory**). De esta manera, al concluir la simulación, se puede acceder al historial completo de estados. El resultado presenta una ventana principal que incluye el panel del mapa, elementos interactivos del juego, indicadores de acciones y estado, así como la posibilidad automatizar o controlar manualmente la simulación.

Ilustración 8: Representación gráfica de la simulación

Esta implementación de la GUI facilita la comprensión y el análisis paso a paso del comportamiento del agente a lo largo de la simulación.

3.5 • Conclusión

Dadas las corridas mostradas anteriormente, podemos llegar a las siguientes conclusiones.

- Las estrategias no informadas funcionan bien para casos pequeños (**anchura**) o en juegos grandes sin límite de energía (**profundidad**).
- Las estrategias informadas son adecuadas para juegos de tamaño intermedio (4-12 tripulantes) donde es importante que el agente respete un conjunto de reglas.
- La búsqueda **Avara** representa la mejor opción de todas las vistas en el trabajo práctico y es la que decidimos recomendar como estrategia final.