Práctica 1: Desarrollo de un agente basado en búsqueda heurística para el entorno GVGAI

Juan Manuel Castillo Nievas

Descripción general de la solución

El comportamiento reactivo y deliberativo se dividen de la siguiente manera

- **Comportamiento deliberativo:** se utiliza para buscar las gemas y el portal y además para mover las rocas en caso de que alguna gema esté encerrada
- **Comportamiento reactivo:** se utiliza para situaciones concretas:
  - Para esquivar una roca cuando el agente ha destapado una casilla con una roca encima
  - Para picar una casilla de tierra cuando hay una piedra encima de dicha casilla
  - Cuando el agente tiene enemigos cerca y tiene que huir

Atributos importantes a tener en cuenta:

```
boolean esquivarRoca;
```

True si va ir hacia una casilla de tierra hacia arriba y una roca va a caer sobre él

```
boolean picar;
```

True si el agente va a ir hacia una casilla de tierra a la derecha o izquierda y hay una roca encima

```
boolean estoyHuyendo;
```

True si el agente tiene enemigos cerca y tiene que huir

```
boolean estoyAtrapado;
```

True si el agente no encuentra un camino hacia ninguna gema o portal

```
boolean moviendoPiedra;
```

True si el agente va hacia una gema pero tiene que mover alguna roca para alcanzarla

Agent.java

El archivo principal es el Agent.java. Este archivo contiene los métodos que controlan el comportamiento reactivo y que calculan un plan para el agente. Los métodos más importantes son:

```
public void taparCasillasMonstruos(StateObservation stateObs)
```

Este método coge la posición de todos los enemigos del mapa y hace una búsqueda en anchura desde el enemigo hasta las casillas de tierra que lo rodean. De esta forma se consigue saber las casillas que hacen

que los enemigos se liberen.

```
public boolean hayPeligroMonstruos(StateObservation stateObs, Vector2d gema)
```

Este método hace una búsqueda A* desde una gema hasta cada enemigo del mapa. Si hay un camino posible, significa que alcanzar esa gema supone un peligro para el agente.

```
public ArrayList<Node> calcularNuevoCamino(nGemas, avatar, stateObs)
```

Este método calcula un camino desde el avatar hasta una gema o un portal. Para saber qué gema es más fácil de seguir se sigue el siguiente razonamiento:

1. Primero se recorren todas las gemas que sean **inalcanzables por los enemigos**
    - Si **existe** un camino desde el avatar hasta la gema, termina la búsqueda y se devuelve el camino
    - Si **no existe** un camino desde el avatar hasta la gema, significa que la gema está encerrada, entonces:
        1. Se calcula la piedra más cercana a la gema y que se puede desprender cavando la tierra
        2. Se calcula un camino desde el avatar hasta la piedra para que se desprenda
2. Si en el paso 1) no se ha encontrado un camino, se recorren todas las gemas **alcanzables por los enemigos**
    - Si **existe** un camino desde el avatar hasta la gema, termina la búsqueda y se devuelve el camino
    - Si **no existe** un camino desde el avatar hasta la gema, significa que la gema está encerrada, entonces:
        1. Se calcula la piedra más cercana a la gema y que se puede desprender cavando la tierra
        2. Se calcula un camino desde el avatar hasta la piedra para que se desprenda
3. Si en el paso 2) no se ha encontrado un camino, significa que el avatar está **atrapado**
    1. El avatar activa la variable *estoyAtrapado* y se añaden acciones *ACTION_NIL*

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer)
```

Este método decide la siguiente acción del avatar. Acciones más importantes que se realizan por orden:

1. Se calculan las casillas que hacen que los enemigos sean liberados
2. Si hay un plan activado, eliminar la última acción que ha hecho el avatar
3. Calcular un camino si el avatar no tiene activado un comportamiento reactivo (variables *estoyAtrapado*, *esquivarRoca*, *picar* ó *estoyHuyendo*) y no hay un plan definido
4. Decidir la acción siguiente:
    - Si el avatar tiene activado un comportamiento reactivo, coger la acción programada correspondiente (se explicará en el apartado del comportamiento reactivo)
    - En caso contrario; si el avatar tiene enemigos cerca, construye un comportamiento reactivo para huir hacia el lado contrario de los enemigos y deshace el plan
    - En caso contrario; si el avatar tiene un plan, se mira la siguiente acción de acuerdo al plan:
        - Si el agente tiene que ir hacia la derecha:
            - Si la casilla siguiente es tierra y encima de ella hay una roca, se construye un comportamiento reactivo para picar esa piedra y deshace el plan
            - Si la casilla siguiente está vacía y hay una roca encima de esa casilla, deshacer plan y quedarse quieto
            - Si no ocurre nada de lo anterior, avanzar a la derecha y seguir con el plan
        - Si el agente tiene que ir hacia la izquierda:
            - Se hace lo mismo de forma análoga al punto anterior
        - Si el agente tiene que ir hacia abajo

        - El avatar avanza hacia abajo y sigue con el plan
    - Si el agente tiene que ir hacia arriba
        - Si la casilla siguiente es tierra y encima de ella hay una roca, construir un comportamiento reactivo para esquivar la roca de forma que después de esquivarla el agente haya ido al mejor lado según su objetivo; y deshace el plan
        - En caso contrario, el agente va hacia arriba y sigue con el plan
- En caso contrario, el agente no hace nada

Comportamiento reactivo

Tal y como se ha explicado brevemente anteriormente, el comportamiento reactivo se utiliza para situaciones concretas.
El comportamiento reactivo se maneja mediante una pila de acciones. Para ello se utiliza el siguiente atributo:

```
Stack<Types.ACTIONS> accionesProgramadas;
```

Dependiendo de lo que se quiera hacer, se almacenarán en la pila unas acciones u otras que el agente irá haciendo hasta que la pila quede vacía.

Huir de enemigos

El agente comprueba si tiene enemigos cerca usando la siguiente función:

```
public boolean hayMonstruoCasillaCercana(stateObs, avatar)
```

Este método comprueba si hay enemigos en las 2 próximas casillas al agente ya sean a la izquierda, derecha, arriba o abajo. Dependiendo de dónde estén los enemigos, el agente huirá hacia un determinado lado usando las siguientes funciones:

```
public void huirDerecha(StateObservation stateObs, Vector2d avatar)
```

Método que se utiliza cuando se han detectado enemigos en la derecha. Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia la izquierda si la casilla de la izquierda está libre para poder huir
- En caso contrario, comprobar la casilla de arriba
- En caso contrario, comprobar la casilla de abajo
- En caso contrario, el agente está atrapado y deshace el plan

```
public void huirIzquierda(StateObservation stateObs, Vector2d avatar)
```

Método que se utiliza cuando se han detectado enemigos en la izquierda. Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia la derecha si la casilla de la izquierda está libre para poder huir
- En caso contrario, comprobar la casilla de arriba
- En caso contrario, comprobar la casilla de abajo
- En caso contrario, el agente está atrapado y deshace el plan

```
public void huirArriba(StateObservation stateObs, Vector2d avatar)
```

Método que se utiliza cuando se han detectado enemigos abajo. Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia abajo si la casilla de la izquierda está libre para poder huir
- En caso contrario, comprobar la casilla derecha
- En caso contrario, comprobar la casilla izquierda
- En caso contrario, el agente está atrapado y deshace el plan

```
public void huirAbajo(StateObservation stateObs, Vector2d avatar)
```

Método que se utiliza cuando se han detectado enemigos arriba. Se sigue el siguiente razonamiento:

- Añadir acciones para ir hacia arriba si la casilla de la izquierda está libre para poder huir
- En caso contrario, comprobar la casilla derecha
- En caso contrario, comprobar la casilla izquierda
- En caso contrario, el agente está atrapado y deshace el plan

Esquivar roca

Para esquivar una roca se va a utilizar la siguiente función:

```
public void buscarMejorHuecoParaEsquivar(stateObs, avatar)
```

Este método calcula la localización de la gema más próxima y se va a seguir el siguiente razonamiento:

- Si la gema más próxima está a la izquierda:
  - Se comprueba si ir a la izquierda no supone un peligro, es decir, que no haya rocas o posibles desprendimientos de rocas y que no haya enemigos
  - En caso contrario, se comprueba si ir a la derecha no supone un peligro de la misma forma que el punto anterior
  - En caso contrario, se comprueba la casilla de abajo. Si se puede ir hacia abajo, se vuelven a mirar las posibilidades para ir hacia la izquierda o derecha.
  - En caso contrario, el agente está atrapado
- Si la gema más próxima está a la derecha:
  - De forma análoga al punto anterior, pero mirando primero la posibilidad de ir a la derecha

Picar

Cuando el agente va a ir hacia una casilla con tierra y encima de esta hay una roca, el agente va a proceder a usar la pala. Para ello se utilizan las siguientes funciones dependiendo de si la casilla está a la derecha o izquierda:

```
public void picarRocaIzquierda(boolean girarParaPicar)
```

```
public void picarRocaDerecha(boolean girarParaPicar)
```

Si el parámetro *girarParaPicar* es **true**, significa que el avatar no está orientado hacia el lado en el que se va a cavar y primero se añadirá la acción de girar. Después se añadirá la acción de usar la pala y por último se van a añadir acciones de no hacer nada para dar un tiempo a que la roca se desprenda.

Estar atrapado

Cuando el agente está atrapado se utiliza la siguiente función:

```
public void esperar()
```

Este método simplemente añade acciones de no hacer nada al agente. De esta forma, el agente se queda esperando hasta que se encuentre un plan.

Comportamiento deliberativo

Para calcular un camino desde una posición *p1* hasta una posición *p2* se utilizan los ficheros PathFinder.java, AStar.java , Anchura.java y Node.java.

AStar.java

Este archivo contiene la implementación del algoritmo clásico A* con unas pequeñas modificaciones. La heurística que se utiliza es la **distancia de Manhattan**. Para generar los vecinos de un nodo, se utilizan 3 modos distintos:

- Modo 0: este modo se utiliza cuando se calcula un camino hacia un portal. Con este modo los nodos que sean **muro**, **gema**, o **roca** no se generan como vecinos.
- Modo 1: este modo se utiliza cuando se calcula un camino desde el avatar hacia una gema. Con este modo los nodos que sean **muro** o **roca** no se generan como vecinos.
- Modo 2: este modo se utiliza cuando se calcula un camino desde un enemigo hacia una gema. Con este modo los nodos que sean **muro**, **tierra** o **roca** no se generan como vecinos. Con este modo se pretende comprobar si un enemigo puede llegar hasta una gema. Si el enemigo puede llegar a la gema, esa gema supone un peligro para el avatar.

Los modos 0 y 1 son casi iguales, pero si se calcula un camino hacia un portal significa que el avatar ya tiene todas las gemas y si el avatar coge una gema de camino al portal, la reglas del juego cambian y es recomendable no cogerla. Estos modos tienen en cuenta las siguientes situaciones:

- Si un vecino es un nodo que contiene tierra y encima de ese nodo hay una roca, se incrementará el costo de dicho nodo.
- Si un nodo es una casilla que abre un habitáculo en el que se encuentra un enemigo (estas casillas se calcularon previamente, tal y como se explica en la descripción general), el costo estimado de dicha casilla se incrementará de forma disparatada.

Anchura.java

Este archivo contiene la implementación del algoritmo búsqueda en anchura. Dispone de 2 modos diferentes en el cálculo de vecinos:

- Modo 0: este modo se utiliza para calcular las casillas de tierra que están en el límite de los habitáculos de los enemigos. Este modo genera vecinos que son **casillas vacías** o **gemas**. Los vecinos que se vayan generando y que sean **tierra**, **portal**, **murciélago**, o **escorpión** se meterán en un vector, indicando que esos nodos son el límite de los habitáculos, es decir, las casillas que liberan a los

enemigos.

- Modo 1: este modo se utiliza para calcular un camino desde una gema hasta una piedra que esté a la misma altura o por debajo de ella. Se generan vecinos que estén a la izquierda, derecha o abajo del nodo inicial. Los vecinos que se tienen son todas menos el muro y la piedra. Cuando se llegue a una **piedra**, se almacenará en un vector que contiene *x* número de piedras que están por debajo de la gema. Este modo sirve para calcular las piedras que se pueden mover para alcanzar una gema.

PathFinder.java

Este archivo contiene los generadores de vecinos de los modos previamente explicados.

```
ArrayList<Node> getNeighboursPortal(currentNode, casillasQueAbrenHabitaculos)
```

Modo 0 en *AStar.java*

```
ArrayList<Node> getNeighbours(currentNode, casillasQueAbrenHabitaculos)
```

Modo 1 en *AStar.java*

```
ArrayList<Node> getNeighboursHabitaculos(currentNode)
```

Modo 2 en *AStar.java*

```
ArrayList<Node> getNeighboursAnchuraCalculo0(currentNode, habitaculosEnemigos)
```

Modo 0 en *Anchura.java*

```
ArrayList<Node> getNeighboursAnchuraCalculo1(currentNode, caminoGemaAPiedra)
```

Modo 1 en *Anchura.java*

Node.java

Esta clase contiene la representación de un nodo. Un nodo está formado por:

```
public double totalCost;
```

El costo total del nodo

```
public double estimatedCost;
```

El costo estimado del nodo

```
public Node parent;
```

El padre del nodo

```
public Vector2d position;
```

La posición en el mapa del nodo