UNIVERSIDAD DE MÁLAGA

Laboratorio - Genéticos con Robocode

José A. Montenegro

Dpto. Lenguajes y Ciencias de la Computación
ETSI Informática. Universidad de Málaga
jmmontes@uma.es

30 de marzo de 2021

Ejemplo JGap

Genéticos en Robocode

Ejercicio

Ejemplo JGap

JGap

- Para entender cómo funciona los algoritmos genéticos vamos a trabajar con la librería de algoritmos genéticos disponible en `https://sourceforge.net/projects/jgap/`
- En el campus virtual está disponible un problema que es resuelto mediante algoritmos genéticos, HelloGeneticos.zip .
- Este problema es conocido como Hello Word, donde partimos de una cadena de texto aleatoria y mediante aplicación de un algoritmo genético debemos obtener la cadena deseada.

Definición Cromosoma

- El cromosoma utilizado en el problema viene determinado por la siguiente sentencias.
- Inicialmente, creamos un gen cadena de longitud 1, con el alfabeto definido.
- Posteriormente, creamos un cromosoma con los genes creados previamente de la longitud de la cadena objetivo que queremos conseguir.

```
String ALPHABET =   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ,
 !áéíóú";
Gene stringGene = new StringGene(conf, 1, 1, ALPHABET);
IChromosome sampleChromosome
   = new Chromosome(conf, stringGene, MESSAGE.length());
```

Función de evaluación

La función de evaluación heredad de la clase *FitnessFunction*:

```
public class CadenaFitnessFunction extends FitnessFunction {
```

La evaluación del cromosoma, se realiza comparando el cromosoma actual con la solución, cada vez que coinciden un carácter, se le sumará 1 a la función.

```
    protected double evaluate(IChromosome aSubject) {
        int fitnessValue = 0;
        int length = aSubject.size();
        for (int i = 0; i < length; i++) {
            String actual = (String) aSubject.getGene(i).getAllele();
            if (actual.toCharArray()[0] == _expected[i]) {
                fitnessValue += 1;
            }
        }
        return fitnessValue;
    }
```

Estudio de la Población

- Ejecute el programa y realice un estudio sobre los resultados obtenidos con distinto número de poblaciones. Sólo es necesario modificar el valor de la variable POBLACION al inicio de la clase.

```
public class CadenaGeneticos{
    private static final int POBLACION = 10;//<---- Población
```

- Podemos observar que a medida que incrementamos la población reducimos el número de generaciones para que finalice, con respecto al tiempo observamos que subir la población de 1 a 10 decrementa el tiempo, pero seguir incrementando la población repercute en el tiempo de cómputo.

Estudio de la Población

Población 1		
Ejecución	**Generación Finaliza**	**Tiempo milisegundos**
1	8663	719
2	15489	1152
3	10333	764
3	13844	1020
4	23439	1389
5	27418	1497
Promedio	**16531**	**1090,2**

Población 10		
Ejecución	**Generación Finaliza**	**Tiempo milisegundos**
1	625	351
2	833	481
3	964	509
3	1017	499
4	764	440
5	1108	555
Promedio	**885,2**	**472,5**

Estudio de la Población

Población 100		
Ejecución	**Generación Finaliza**	**Tiempo milisegundos**
1	235	944
2	123	587
3	189	695
3	134	581
4	129	532
5	154	715
Promedio	**160,7**	**675,7**

Población 1000		
Ejecución	**Generación Finaliza**	**Tiempo milisegundos**
1	47	1411
2	44	1430
3	50	1552
3	43	1328
4	41	1322
5	45	1409
Promedio	**45,0**	**1408,7**

Eliminar la Mutación

- Estableciendo una población de 10, ejecute el algoritmo manteniendo el operador *CrossOver* y eliminando el operador de *Mutación*. Utilice el método *modifyConfiguration* para establecer la configuración, como muestra el código adjunto.

```
// org.jgap.impl.MutationOperator
    MutationOperator mutation  = new MutationOperator(conf);
//  conf.addGeneticOperator(mutation);
```

- En un caso de ejecución observamos que en la Generación 7 todos los cromosomas son iguales.

```
************************
Generacion 7: Obtenido ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
************************
Chromosoma 0 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
Chromosoma 1 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
Chromosoma 2 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
Chromosoma 3 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
Chromosoma 4 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
Chromosoma 5 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
Chromosoma 6 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
Chromosoma 7 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
Chromosoma 8 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
Chromosoma 9 : ajRavkáqvfaélQFPaYgOrAplWkC fitness: 2
```

Eliminar el Crossover

- Estableciendo una población de 10, ejecute el algoritmo manteniendo el operador *Mutacion* y eliminando el operador de *CrossOver*. Utilice el método *modifyConfiguration* para establecer la configuración, como muestra el código adjunto.

```
// org.jgap.impl.CrossoverOperator;
        CrossoverOperator cross = new CrossoverOperator(conf);
        //conf.addGeneticOperator(cross);
```

Población 10		
Ejecución	**Generación Finaliza**	**Tiempo milisegundos**
1	1851	642
2	1917	634
3	1183	597
3	1994	789
4	1623	651
5	1311	599
Promedio	**1646,5**	**652,0**

- Observamos que el algoritmo encontraría la solución, pero el promedio subiría de 885,2 a 1646,5 generaciones y los milisegundos de 472,5 a 652,0.

Genéticos en Robocode

Prioridades de los eventos en Robocode

https://robocode.sourceforge.io/docs/robocode/robocode/AdvancedRobot.html

Hasta el momento hemos utilizado la clase Robot.

```
public class BusquedaRobot extends Robot
```

Ahora vamos a heredar de la clase AdvancedRobot,

```
public class EvolvableRobot extends AdvancedRobot
```

que nos proporciona más funcionalidades, una de ellas el método *setEventPriority*:

```
public void setEventPriority(String eventClass,int priority)
```

- Los eventos con mayor prioridad pueden interrumpir a los eventos con menor prioridad.
- En caso de empate en la prioridad, los eventos con la misma prioridad, los eventos nuevos son ejecutados primeros.
- Las prioridades válidas van desde 0-99, siendo 80 la prioridad por defecto.

Prioridades por defecto

Las prioridades por defecto, de mayor a menor son:

- WinEvent: 100 (reserved)
- SkippedTurnEvent: 100 (reserved)
- StatusEvent: 99
- CustomEvent: 80
- MessageEvent: 75
- RobotDeathEvent: 70
- BulletMissedEvent: 60
- BulletHitBulletEvent: 55
- BulletHitEvent: 50
- HitByBulletEvent: 40
- HitWallEvent: 30
- HitRobotEvent: 20
- ScannedRobotEvent: 10
- PaintEvent: 5
- DeathEvent: -1 (reserved)

Nos centraremos en los eventos marcados en rojo.

Cromosoma

El cromosoma está formado por 84 enteros, divididos en tres bloques:

Bloques del cromosoma	Número de Genes	Descripción	Cuadro
Prioridad de los Eventos	7	La prioridad de los eventos. Un evento con mayor prioridad puede interrumpir la prioridad de un evento con menor prioridad.	1
Naturaleza de cada Comportamiento	11	Modificar la naturaleza original de cada comportamiento. Un valor mayor de 50 significa que se eliminará la lista de eventos pendientes del robot.	2-3
Acciones para cada Comportamiento	6*11	6 Acciones para los 11 comportamientos (0-10)	4

Cromosoma - Prioridades de los Eventos

id	Prioridad Eventos
0	BulletHitEvent
1	BulletHitBulletEvent
2	BulletMissedEvent
3	HitByBulletEvent
4	HitRobotEvent
5	HitWallEvent
6	ScannedRobotEvent

Cuadro 1: Prioridades de los Eventos

Comportamiento Robot

Id	Comportamiento	Descripción
0	Por defecto	Ninguno de los eventos son lanzados.
1	Bala alcanza robot	Una bala alcanza a otro robot.
2	Bala alcanza bala	Una bala alcanza otra bala.
3	Bala fallada	Una bala no alcanza el objetivo.
4	Alcanzado por una bala	El robot es alcanzado por una bala.
5	Golpea a un robot	El robot choca con otro robot.
6	Golpeado por un robot	Otro robot choca con el nuestro.
7	Choca con una pared	El robot choca con la pared.
8	Robot detectado a poca distancia	El radar detecta otro robot a una distancia menor que 100 pixels.
9	Robot detectado a distancia media	El radar detecta otro robot a una distancia entre 100 y 300 pixels.
10	Robot detectado a larga distancia	El radar detecta otro robot a una distancia mayor que 300 pixels.

Cuadro 2: Comportamiento del Robot y sus descripciones

Comportamiento Robot

Id	Comportamiento	Método
0	Por defecto	Run method.
1	Bala alcanza robot	*BulletHitEvent*
2	Bala alcanza bala	*BulletHitBulletEvent*
3	Bala fallada	*BulletMissedEvent*
4	Alcanzado por una bala	*HitByBulletEvent*
5	Golpea a un robot	*HitRobotEvent*
6	Golpeado por un robot	*HitRobotEvent*
7	Choca con una pared	*HitWallEvent*
8	Robot detectado a poca distancia	*ScannedRobotEvent* $<$ 100 pixels.
9	Robot detectado a distancia media	*ScannedRobotEvent* 100 $>$ & $<$ 300 pixels.
10	Robot detectado a larga distancia	*ScannedRobotEvent* $>$ 300 pixels.

Cuadro 3: Comportamiento del Robot y sus métodos

Acciones Robot

Las acciones que componen cada comportamiento son:

id	**Nomenclatura**	**Acciones**	**Descripción de las acciones**
0	*fba*	Movimiento Avanza/Retrocede	Valores positivos hace que el robot avance y los negativos hacen que el robot retroceda.
1	*tba*	Gira cuerpo	0 Ninguna acción 1 Gira el cuerpo respecto "bearing" del objeto detectado 2 Gira el cuerpo respecto "bearing" del objeto detectado sin girar el arma 3 Gira el cuerpo respecto "heading" del robot 4 Gira el cuerpo respecto "heading" del robot sin girar el arma 5 Gira el cuerpo del robot como si el objeto detectado fuera un espejo 6 Gira el cuerpo del robot como si el objeto detectado fuera un espejo sin girar el arma
2	tbp	Angulo giro cuerpo	El ángulo de giro del cuerpo utilizado en tba
3	*tga*	Giro arma	0 Ninguna acción 1 Gira el arma con respecto al "bearing" del objeto detectado 2 Gira el arma con respecto al "heading" del robot
4	*tgg*	Angulo giro del arma	El ángulo de giro del arma utilizado en tga
5	*fa*	Disparo	Intensidad del disparo de 0 a 3. El valor 0 no producirá disparo

Cuadro 4: Comportamiento del Robot y sus eventos asociados

EvolvableRobot - Prioridad Eventos

La lectura del cromosoma se realiza en el método *init* del Robot. Además se establece la prioridad de los 7 eventos, determinada por el cuadro 1.

```
private boolean init() {
 try {
   leerCfg();                                    //Leo la configuracion
   } catch (Exception e) {
                out.println(e);
  }
  eventPriority = new int[numeroDeEventos];
  behaviourSubsumption = new boolean[numeroDeComportamientos];
  behaviourActions = new int[numeroDeComportamientos][numeroDeAcciones];
  if (!decodeGenome()) {//Leo el genoma que pone la funcion de fitness
   return false;
 }
 setEventPriority("BulletHitEvent", eventPriority[0]);
 setEventPriority("BulletHitBulletEvent", eventPriority[1]);
 setEventPriority("BulletMissedEvent", eventPriority[2]);
 setEventPriority("HitByBulletEvent", eventPriority[3]);
 setEventPriority("HitRobotEvent", eventPriority[4]);
 setEventPriority("HitWallEvent", eventPriority[5]);
 setEventPriority("ScannedRobotEvent", eventPriority[6]);
 return true;
}
```

EvolvableRobot - Comportamiento

- El código siguiente muestra la lectura de la naturaleza de los 11 comportamientos, dónde el primero que se corresponde por defecto no se modifica y los otros dependen del valor del cromosoma es mayor 50 serán puesto a verdadero.

```
behaviourSubsumption[0] = false;
  for (i = 1; i < numeroDeComportamientos; i++)
   behaviourSubsumption[i] = scan.nextInt() > 50;
```

- El método *executeBehaviour* ejecutará las 6 acciones para el comportamientos escogido, recogidos en la tabla 4. Al finalizar las acciones se ejecutará el método de Robocode *executes*, que permite que sigan ejecutándose las acciones que están pendientes.
- Al final del método se observa que si el comportamiento está a verdadero se borrarán los eventos pendientes con el método de Robocode *clearAllEvents*.

```
private void executeBehaviour(int behaviour, double bearing) {
 prepareMoveAction(behaviourActions[behaviour][0]);
 prepareTurnRobotAction(behaviourActions[behaviour][1], behaviourActions[behaviour][2], bearing);
 prepareTurnGunAction(behaviourActions[behaviour][3], behaviourActions[behaviour][4], bearing);
 fireAction(behaviourActions[behaviour][5]);
 execute();
 if (behaviourSubsumption[behaviour])  clearAllEvents();
```

EvolvableRobot - Eventos Robocode

- Cada evento ejecutará su comportamiento determinado, por ejemplo el método *onScannedRobot*, ejecutará su comportamiento dependiendo la distancia a la que se encuentre su enemigo.

```
public void onScannedRobot(ScannedRobotEvent event) {
    if (event.getDistance() < 100) {            // Enemy is at close distance
        executeBehaviour(scannedCloseDistRobotBehaviour, event.getBearing());
        if (verbose > 1) {
            out.print("Target near\t");
            out.println(event.getBearing());
        }
    }
    else if (event.getDistance() < 300) {       // Enemy is at mid distance
        executeBehaviour(scannedMidDistRobotBehaviour, event.getBearing());
        if (verbose > 1) {
            out.print("Target mid\t");
            out.println(event.getBearing());
        }
    }
    else {                                      // Enemy is at long distance
        executeBehaviour(scannedLongDistRobotBehaviour, event.getBearing());
        if (verbose > 1) {
            out.print("Target far\t");
            out.println(event.getBearing());
        }
    }
}
```

EvolvableRobot - Eventos Robocode

- El método Run de Robocode ejecutará el comportamiento por defecto.

```
public void run() {
    setColors(Color.BLACK, Color.BLUE, Color.YELLOW);
    init();
    if (verbose > 0) {
        out.println(Arrays.toString(eventPriority));
        out.println(Arrays.toString(behaviourSubsumption));
        for (int i = 0; i < numeroDeComportamientos; i++) {
            out.println(Arrays.toString(behaviourActions[i]));
        }
        out.flush();
    }
    while (true) {
        executeBehaviour(defaultBehaviour, 0);
        if (verbose > 1) {
            out.flush();
        }
    }
}
```

clase RobotEngine

RobotEngine : Constructor que establece las variables:

- *poplacionSize*. Tamaño de la población
- *maxIteraciones*. Número de evoluciones de la población
- *numeroBatallas*. Número de batallas en cada evolución

StartEngine : Método que invoca al entorno de Robocode para comenzar las batallas. Mediante la variable booleana *ejecucionVisible*, las batallas pueden ejecutarse en segundo plano o ser visualizada. Además de invocar el entorno crea el algoritmo genético *RobocodeGenetic* invocando a los métodos *setGeneticConfiguration* y *evolve*.

clase BattleObserver

Esta clase es un adaptador que es ejecutado al finalizar una batalla. Hereda de la clase BattleAdaptor de la librería Robocode. Actualmente solamente almacena la puntuación de nuestro robot y del enemigo, implementando la función *onBattleCompleted*.

```
public void onBattleCompleted(BattleCompletedEvent e) {
 if (e.getIndexedResults().length > 1) {
  scoreRobot = e.getIndexedResults()[0].getScore();  // cero es el genético
  scoreEnemy = e.getIndexedResults()[1].getScore();
  //  Explorar la variable para más información del fitness
 //         e.getSortedResults()[0].getBulletDamage();
 }
 else {
  System.out.println("Error. Robocode did not send results.");
  scoreRobot = 0;
  scoreEnemy = 0;
  }
 }
```

clase RobocodeGenetic

RobocodeGenetic : Constructor que establece los parámetros para el tamaño de la población, número de evoluciones de la población y número de batallas en cada evolución.

- archivoCfg. Almacena en un archivo (cfg.dat) la configuración del cromosoma. La utilizará el robot para saber cómo leer el cromosoma.

setGeneticConfiguration : Configura la librería genética jgap, estableciendo todos los parámetros necesarios, cromosoma, población, función fitness y operadores genéticos

evolve : Es el método encargado de realizar la evolución. Invoca a los siguientes métodos privados.

- *getPoblacionInicial*: Obtiene la población inicial del archivo EvolvingRobocode.xml de una ejecución anterior o crea una población aleatoria si el archivo no existe.
- *setPoblacionFinal*: Almacena la última población en el archivo EvolvingRobocode.xml para que sirva de inicio en otra ejecución.
- *storeBestChromosoma*: Almacena el mejor cromosoma en el archivo genome. El mejor comportamiento obtenido en las batallas.

clase RobocodeFitnessFunction

Clase encargada de implementar la función de fitness para el genético. Hereda de la clase FitnessFunction de la librería jgap.

RobocodeFitnessFunction : Es el constructor de la clase que permite establecer los parámetros de configuración de la batalla y cromosoma.

evaluate : Es la función encarga de evaluar cada cromosoma.

Inicialmente almacena el cromosoma que se está evaluando en el fichero genome para que el Robot aplique el contenido del cromosoma a su comportamiento.

El procedimiento invocará al entorno de Robocode para ejecutar el número de batallas configuradas con los robots disponibles.

Devolverá un valor double, que actualmente es un porcentaje de la puntuación de nuestro robot menos un porcentaje de la puntación del enemigo.

```
 for (i = 0; i < numberOfBattles; i++) {
  engine.runBattle(battleSpecs, true);
  double miScore                  = battleObserver.getScoreRobot();
  double enemigoScore = battleObserver.getScoreEnemy();
  tempFitness += miScore /300 - enemigoScore/500;
}
 tempFitness /= numberOfBattles;
 fitness += tempFitness;
```

Ejercicio

Ejercicio

Realice las siguientes modificaciones para evaluar el aprendizaje del Robot:

- Modifique la configuración del sistema:
  - *poplacionSize*. Tamaño de la población
  - *maxIteraciones*. Número de evoluciones de la población
  - *numeroBatallas*. Número de batallas en cada evolución
- Modifique la función de evaluación. Podemos tener en cuenta otros parámetros además de la puntuación de los robots implicados en la batalla.
- Modifique la configuración del cromosoma, bien añadiendo más elementos al cromosoma o cambiando el significado de los elementos.

José A. Montenegro Montes
Dpto. Lenguajes y Ciencias de la Computación
ETSI Informática. Universidad de Málaga
jmmontes@uma.es

UNIVERSIDAD
DE MÁLAGA