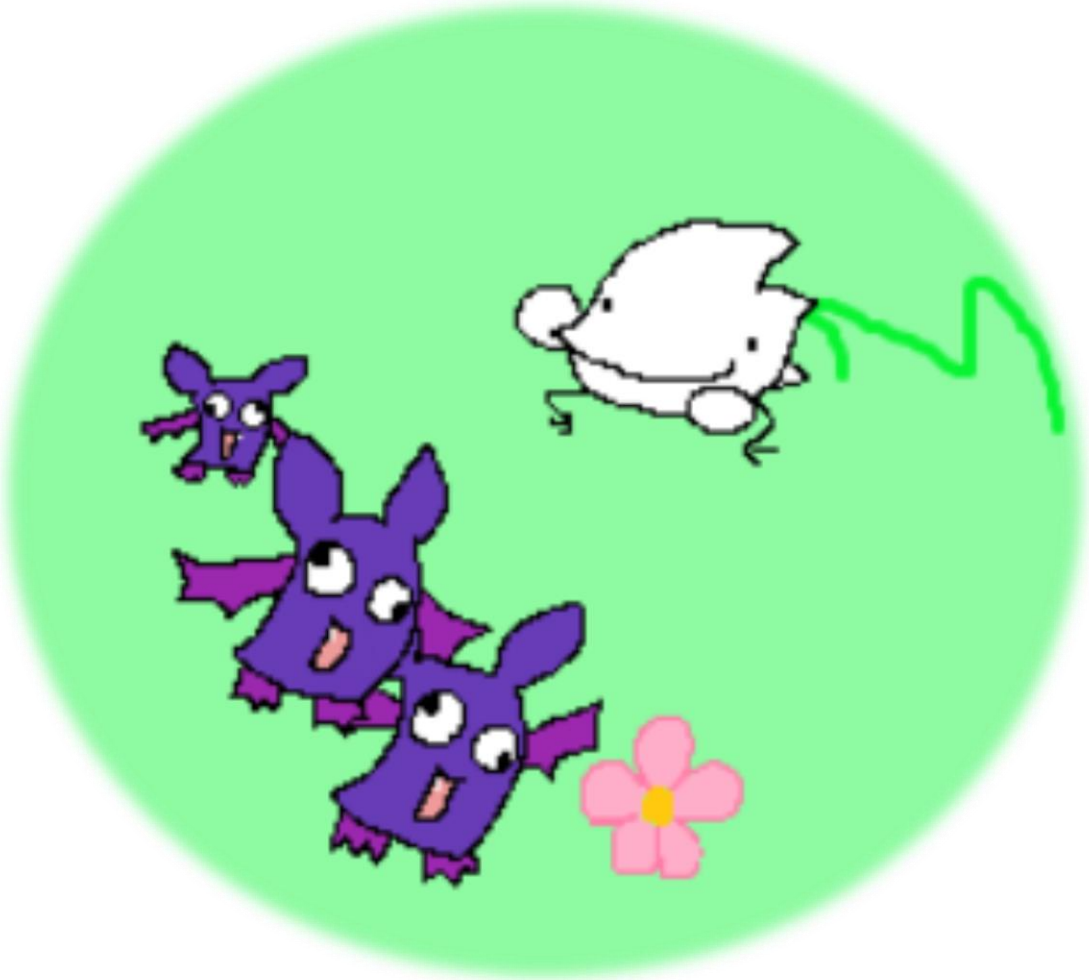
Aplicación de un algoritmo genético a un videojuego



María Ruiz Molina

Universidad de Valladolid

Escuela de Ingeniería Informática

Índice:

Descripción detallada del problema a resolver. El incremento de la dificultad en un videojuego:

La industria del videojuego se encuentra en apogeo. Cada vez son más las sagas de juegos que se unen al nuevo concepto de e-sport o deporte electrónico donde equipos de todo el mundo se enfrentan usando sus habilidades en un videojuego. Otras tratan de proponer nuevos retos a jugadores con diseños de niveles detallados hasta el último píxel o simplemente buscan la forma de mantener entretenidos a largo plazo a un público meramente casual que juega desde las pantallas de sus móviles.

Este problema involucra múltiples cuestiones, como gráficos atractivos (que no tienen por qué ser realistas), historias que resulten interesantes, mecánicas que controlen la jugabilidad o el diseño de niveles y de las curvas de dificultad que estos poseen.

Es aquí donde encontramos el problema a la hora de definir la "curva de dificultad perfecta" para cada videojuego, pues cada jugador es un mundo y lo que para uno puede resultar más sencillo, para otro puede volverse un pico en ese incremento de complejidad dependiendo de experiencias anteriores con juegos similares u otros factores. Es por ello por lo que las empresas dedicadas a este sector de desarrollo emplean parte del tiempo de pruebas y testeo en hacer probar sus productos a individuos ajenos a los diseñadores antes de sacar el juego al mercado.

Para incrementar la dificultad de un videojuego cada empresa posee su propia estrategia, y, además, cada tipo de juego (plataformero, de lucha, shooter…) puede requerir o verse beneficiado más o menos de una técnica u otra, sea uso de tutoriales, manuales, diseños cada vez más complejos, inteligencias artificiales cada vez más dotadas...

Por poner un ejemplo de hasta dónde es capaz de llegar la complejidad de dicho diseño voy a recurrir al juego de Super Mario Bros para la consola NES (Nintendo Entertainment System) diseñado por Shigeru Miyamoto, lanzado el 13 de septiembre de 1985 y producido por la compañía Nintendo. En concreto haré un pequeño análisis de la primera pantalla del primer nivel que nos encontramos en este juego.

Imagen 1. Comienzo del nivel

En la Imagen 1 podemos ver cómo la primera pantalla nos proporciona ya información sobre el juego. El personaje aparece situado mirando hacia la derecha lo que invita al jugador a moverse hacia ese lado de la pantalla, y en caso de equivocarse de dirección pronto tocará el borde imaginario situado a la izquierda de la pantalla para impedir su avance en dirección contraria. Además, el hecho de que no aparezca ningún elemento interactivo permite al jugador experimentar con los controles y pulsar todos los botones del mando de la consola.

Imagen 2. Primer enemigo

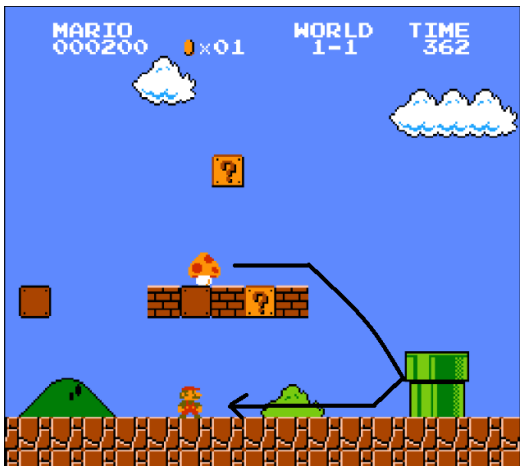


Imagen 3. Primer “Power Up”

Al poco de avanzar en la pantalla vemos cómo aparecen los dos primeros elementos. Uno brilla y posee un símbolo de interrogación, invitando al jugador a tocarlo. El otro posee un movimiento sencillo, viene hacia nosotros y parece tener cara de enfadado. El jugador puede entenderlo como un enemigo y dados los controles del juego optar por saltar por encima de él evitándole, sobre él eliminándole o bien no hacer nada y perder en cuanto este le toque. Esto haría que el jugador volviese a la posición en la Imagen 1 y pudiera volver a intentar superar este pequeño obstáculo sin haber perdido apenas tiempo en el nivel.

Tras ello vendría la Imagen 3, donde tras haber saltado debajo del bloque con el símbolo de interrogación habríamos obtenido una moneda, las cuales sirven para puntuar al jugador al final del nivel. Seguido hay otro bloque de la misma forma que tras tocarlo sale de él otro nuevo elemento: Un champiñón que es un Power Up, es decir un objeto que potencia al jugador. Este se moverá siguiendo la flecha dibujada en la Imagen 3. De esta forma, el diseñador del nivel se asegura de que el jugador obtenga el objeto y no pueda evitarlo, pues si trata de soltar rebotará con los bloques situados encima de él, y a su vez aprenda la mecánica de este “Power Up”, rebotando en todo lo que tocan. Además, tras ello se encuentra la tubería verde, la cual obliga al jugador a aprender a saltar algo más alto sobre ella para poder avanzar.

El nivel da para mucho más y podría servir para dar toda una “master class” en diseño de niveles para juegos plataformeros, pero espero con este ejemplo haber plasmado la complejidad que se encuentra detrás de algo tan simple como estas tres pantallas.

Este diseño se va volviendo más y más complicado según se avanza en el juego y cada vez toca puntos más específicos y concretos de las mecánicas, donde entra la cuestión de cómo cada

jugador puede llegar a afrontar esos retos y cuánto puede costarle según las dificultades que haya tenido en los anteriores niveles.

Por ello, una posible solución es adaptar el juego a cada jugador, haciendo que este aprenda de sus movimientos, fallos… en determinados puntos. Dicha información el juego la usaría para mejorar aquellas características en las que el jugador parece flaquear más, como comenzar a hacerle luchar contra más enemigos de un tipo concreto contra el que parece tener dificultades; adaptar los ataques de un rival concreto en un juego de lucha a las tácticas que ha estado usando el jugador en anteriores pantallas o incluso a lo largo del propio combate; o establecer una estrategia en un juego por turnos que posea ventaja ante la que el jugador parece haber empleado más a lo largo del juego.

Es aquí donde entraría en acción la construcción de un algoritmo genético que fuese capaz de ir adaptando y evolucionando la estrategia usada por el juego contra el jugador.

Justificación de estrategia utilizada: Adaptación a cada jugador de manera individualizada.

Empleando un algoritmo genético el juego se adaptaría a cada jugador, haciendo evolucionar a las mecánicas hacia aquellas características que parecen oponer más resistencia a las habilidades del usuario. Una vez que este pareciese dominar esa nueva estratagema el juego comenzaría un nuevo proceso de adaptación buscando el nuevo punto flaco del jugador, seleccionando aquellas mecánicas con características más aptas contra la forma de jugar del usuario.

Aplicando esta estrategia son los siguientes puntos los correspondientes a cada parte del algoritmo:

- **Problema para optimizar:** Aumentar la dificultad del juego.
- **Soluciones factibles:** El conjunto de enemigos, estrategias, mecánicas… que dispone el juego en cuestión.
- **Evaluación de las soluciones:** Cuánta resistencia oponen al jugador a la hora de avanzar este en el juego.
- **Operadores estocásticos:** Distintas mutaciones o variaciones que pueden darse dentro de cada estrategia o mecánica con el fin de explorar nuevas posibilidades.
- **Aplicación iterativa de los operadores sobre las soluciones:** La constante evolución de las mecánicas a la forma de jugar del usuario.

De esta manera la evolución sería constante. El algoritmo podría combinarse con machine learning para que en cada estado determinado incrementara las posibilidades de que unos parámetros u otros sean los seleccionados a la hora de evolucionar hacia una solución.

Este algoritmo presenta las ventajas ya vistas, y de hecho existen juegos que hacen uso de esta técnica como por ejemplo Warning Forever para la plataforma Microsoft Windows por la compañía indie Hikware fundada por el desarrollador Hikoza T Ohkubo, donde los enemigos finales reaccionan y te atacan según cómo hayas jugado ante los anteriores rivales.

Sin embargo, no todo iban a ser ventajas y como bien dije en la introducción, cada juego necesita de unas mecánicas y características. Así pues, el uso de un algoritmo genético aplicado a un videojuego dependerá ampliamente de las características y necesidades de este.

Para ilustrar un caso donde puede emplearse esta estrategia procedo a exponer un juego simple donde el jugador tiene que eliminar a una serie de enemigos hasta limpiar la pantalla para ganar y en el caso de que los enemigos le toquen un número determinado de veces, este perderá.

Estos enemigos se multiplicarán cada cierto tiempo y las nuevas generaciones estarán adaptadas tales que hereden las características de aquellos individuos que lleven más tiempo sobreviviendo al jugador, dando además cabida a posibles mutaciones o variaciones en dichos parámetros. Dichas características son tamaño del enemigo, velocidad de movimiento y amplitud del desplazamiento producido en cada iteración.

En concreto el algoritmo procede de la siguiente manera:

Se genera una población inicial de 7 enemigos, de los cuales 2 son completamente aleatorios y para los otros 5 se establecen unos valores lo suficientemente distintos para asegurar una mayor riqueza a la población. Los tamaños de los enemigos en todo caso serán aleatorios.

Tras ello comienza el juego y con ello la interacción con el jugador. Cada ciclo del juego se suma 1 al factor de tiempo de supervivencia de cada enemigo y si alguno quita alguna vida al jugador, le suma 1 a su factor de peligro. Los valores del enemigo con mayor factor de supervivencia se almacenan, así como los del enemigo con mayor factor de peligro.

En el momento de la generación de un nuevo individuo se tienen en cuenta dichos parámetros. Para ello, los nuevos atributos de este serán la media de aquellos que dicten dichos factores.

Tras ello se considera una posibilidad de mutación de forma individualizada para cada gen o parámetro. De esta manera se genera un número pseudoaleatorio que se suma o resta al valor anterior, pero siempre dentro de unos límites. Para el tamaño del enemigo se seleccionaría al azar uno de los 3 tamaños disponibles.

El proceso seguiría durante todo el juego, sustituyendo cada vez los parámetros empleados en la generación de nuevos enemigos por aquellos correspondientes a los nuevos enemigos que sobrevivan por más tiempo o dañen más al jugador.

Pseudocódigo del algoritmo desarrollado:

```
if tiempo_supervivencia_mayor > tiempo_supervivencia_nuevo_enemigo_generado:
    Asigna los valores de velocidad, evasión y tamaño del enemigo con mayor tiempo de
    supervivencia registrado al nuevo enemigo generado.

#En caso de que algún enemigo haya dañado al jugador, sus parámetros se tendrán en cuenta
también a la hora de generar un nuevo enemigo.
        if existe_velocidad_dominante:
          velocidad_enemigo_nuevo= (velocidad_enemigo_nuevo + velocidad_dominante)/2
        if existe_evasion_dominante:
          evasion_enemigo_nuevo= (evasion_enemigo_nuevo + evasion_dominante)/2
        if existe_tamaño_dominante:
              if (tamaño_enemigo_nuevo=='L' and tamaño_dominante == 'S') or
              (tamaño_enemigo_nuevo=='S' and tamaño_dominante == 'L'):
            tamaño_enemigo_nuevo = 'M'  #Equivale a una media entre tamaño L y S


    #El nuevo enemigo puede mutar de forma individualizada en cada característica
      if 30% de_los_casos:
        velocidad_enemigo_nuevo += número_aleatorio_entre_-1_y_1
              #Controla que no se salga de los límites
        if velocidad_enemigo_nuevo < 0.1:
          velocidad_enemigo_nuevo = 0.1
        if velocidad_enemigo_nuevo > 1:
          velocidad_enemigo_nuevo = 1

      if 30% de_los_casos:
        evasion_enemigo_nuevo += número_aleatorio_entre_-1_y_1
              #Controla que no se salga de los límites
        if evasion_enemigo_nuevo < 0:
          evasion_enemigo_nuevo = 0
        if evasion_enemigo_nuevo > 0.9:
          evasion_enemigo_nuevo = 0.9

      if 20% de_los_casos:
        tamaño_enemigo_nuevo= aleatorio entre S, M y L

.....
```

Cada ciclo del juego se suma 1 al tiempo de supervivencia de cada enemigo. Esta operación se encuentra dentro del bucle principal del juego y equivale a recorrer una lista de tamaño N donde N es el número de enemigos.

actualización_de_los_enemigos()

.....
Más adelante en el código se encuentra la parte del algoritmo que se encarga de determinar qué enemigo ha causado más daño al jugador a lo largo de la partida y de configurarlo para tenerlo en cuenta a la hora de generar nuevos enemigos en las futuras generaciones:
.....

#En concreto lo hace en la parte donde comprueba si algún enemigo ha colisionado con el jugador haciéndole perder una vida:
#Nota: La inmunidad al daño hace referencia a un intervalo de tiempo corto en el que el jugador tras recibir un golpe es inmune al daño para darle la oportunidad de separarse del enemigo. Esta se controla con un temporizador y un booleano.

```
for enemigo_actual in grupo_de_enemigos:
    if supervivencia_enemigo_actual>mayor_tiempo_supervivencia_registrado:
        mejor_velocidad=velocidad_enemigo_actual
        mejor_evasion=evasion_enemigo_actual
        mejor_tamaño=tamaño_enemigo_actual
        mayor_tiempo_supervivencia_registrado =tiempo_supervivencia_enemigo_actual

    if (enemigo_actual golpea jugador) and inmunidad_a_daño == False:
        enemigo_actual.fuerza += 1
        enemigo_mas_fuerte=enemigo_actual

        for enemigo in grupo_de_enemigos:
            if fuerza_de_enemigo>fuerza_del_mas_fuerte:
                enemigo_mas_fuerte=enemigo
                velocidad_dominante= velocidad_del_enemigo_mas_fuerte
                evasion_dominante= evasion_del_enemigo_mas_fuerte
                tamaño_dominante= tamaño_del_enemigo_mas_fuerte

        inmunidad= True
        if vidas== 0:
            bucle_juego=False
            gameover= True
```

Análisis de coste:

Para analizar el coste de este algoritmo vemos que posee tres puntos para tener en cuenta.

El primero, el cual asigna los valores al nuevo enemigo generado, poseerá un coste del orden de O(cte) pues simplemente se dedica a asignar los valores correspondientes obtenidos en la otra parte del algoritmo (que veremos a continuación). Así pues, hay diversas operaciones todas de orden O(1) tales que condicionales, sumas, asignaciones o generaciones de números pseudoaleatorios.

El segundo, donde se actualiza la lista de los enemigos tras comprobar si se ha eliminado a alguno, y donde se suma 1 al tiempo de supervivencia de cada enemigo, así pues, de orden O(n) siendo n el número de enemigos que quedan en pantalla.

Y finalmente donde se comprueba qué enemigo ha causado más daño al jugador y se establece este como el más fuerte, así como qué enemigo ha registrado el mayor tiempo de supervivencia hasta ahora. Esta parte necesita de dos bucles, uno para comprobar los tiempos de supervivencia y actualizarlos, y qué enemigos han hecho contacto con el jugador en ese momento, añadiéndolos al conjunto de enemigos que han quitado vidas al jugador en algún momento y otro dentro de este para determinar qué enemigo ha golpeado más veces al jugador, el cual determinará los segundos valores para tener en cuenta en la generación de nuevas generaciones. Esto implica que tenemos un bucle de tamaño O(n) siendo n el número de enemigos en total, y O(m) el bucle interno, siendo m el número de enemigos que han golpeado al jugador en algún momento. Así pues, este fragmento de código quedaría de orden O(n*m).

En su total tendríamos así pues $O(cte)+O(n)+O(n*m) \approx O(n) + O(n*m) = O(n + n*m)$

Además, aunque posee un orden de O(n*m), hay que tener en cuenta que tanto n como m no serán excesivamente grandes para este caso, pues llegaría un momento antes de que eso ocurriese en el que habría tantos enemigos en pantalla que al jugador le sería físicamente imposible evitarlos y por lo tanto terminaría dándose la condición de fin de juego.

Lenguaje de programación utilizado:

El juego se ha programado en Python 2.7.13 empleando las librerías de sys, os, random y pygame para llevar un control de tiempos, generación de números pseudoaleatorios y facilitar la programación de elementos referentes a videojuegos como “hitboxes” o manipulación de “sprites” y grupos de estos.

La decisión de emplear este lenguaje de programación se debe a la experiencia previa tanto con él como con la librería de Pygame que existe para este, lo cual dadas las características de juego del programa los volvían muy convenientes.

Bibliografía:

- Página de descargas del videojuego Warning Forever por Hikware: http://www.hikware.com/Prod/dlcount.cgi?product=wf

- Documentación de la librería Pygame para Python: https://www.pygame.org/docs/

- Design Club - Super Mario Bros: Level 1-1 - How Super Mario Mastered Level Design. Extra Credits https://www.youtube.com/watch?v=ZH2wGpEZVgE

- Genetic Algorithm in Artificial Intelligence - The Math of Intelligence. Siraj Raval https://www.youtube.com/watch?v=rGWBo0JGf50&list=PLYxupvX3vjuAZ5HeYl9qU1qC8wW0HMi6U&index=3&t=0s

- Canal de YouTube sobre diseño de videojuegos. Extra Credits: https://www.youtube.com/channel/UCCODtTcd5M1JavPCOr_Uydg

- Página empleada para diseñar los sprites y animaciones: PIXILART: https://www.pixilart.com/

- Página empleada para diseñar algunos de los audios del juego: BFXR: https://www.bfxr.net/

- Aplicación empleada para grabar los audios para el doblaje de los personajes: Audacity: https://audacity.es/

- Diapositivas sobre algoritmos genéticos de la asignatura Algoritmos y Computación de 3º de Grado de la mención de Computación de Ingeniería Informática de la Universidad de Valladolid