TC03 - Snake Algoritmo Evolutivo

Francisco Villanueva Quirós
Escuela de Ingeniería
en Computación
Tecnológico de Costa Rica
Paraíso, Cartago
franciscovq@estudiantec.cr

Sebastián Cruz Guzmán
Escuela de Ingeniería
en Computación
Tecnológico de Costa Rica
La Unión, Cartago
sebastiancruz@estudiantec.cr

***Abstract*—This paper presents the development of an autonomous agent capable of playing the classic Snake game using genetic algorithms as an evolutionary learning mechanism. The main objective is for the agent to improve its performance over multiple generations, learning more effective strategies through selection, crossing and mutation processes. A simplified version of the game is implemented in Pygame, and an aptitude function is designed that evaluates the agent's performance in amusement of the amount of food consumed, the duration of the game and penalties for temporary deaths. The agent's genetic representation was based on a decision table, which allows efficient coding of complete behaviors. Experiments were performed by varying key parameters such as population size and mutation rate. This project demonstrates the viability of genetic algorithms as a tool for automatic generation of intelligent behaviors.**

I. Introducción

El desarrollo de agentes inteligentes es una parte muy importante en la Inteligencia Artificial (IA). Uno de los métodos utilizados para realizar dicho comportamiento, es mediante algoritmos evolutivos. Estos son algoritmos inspirados en el proceso de evolución natural. Para este proyecto, se nos solicitó implementar un agente autónomo que aprendiera a jugar el juego Snake mediante un algoritmo genético. Además, el agente debe mejorar su desepeño a medida que juega mas partidas, por ende, se van a crear generaciones de agentes y durante el tiempo, estas generaciones deberían ser mejores que las anteriores. Para esto, se va a utilizar los pilares principales de los procesos evolutivos como lo son: Selección, Cruce y Mutación.

Para iniciar con la implementación del agente, se utilizó una versión simple del juego de Snake utilizando la biblioteca Pygame, con las reglas estándar del juego. Se crearon todos los componentes del juego: un tablero bidimensional de un tamaño ajustable y el tamaño de los bloques por los que esta compuesto el tablero, la creación de un agente inicial (la serpiente) que se pueda mover solo en cuatro direcciones (arriba, abajo, izquierda, derecha), la aparición de comida de forma aleatoria y las reglas para perder en la partida como que no puede chocar con los bordes del tablero o consigo mismo.

La representación genética fue realizada con una tabla de decisión, una matriz de pesos que traduce el estado del entorno a una acción concreta, esto mediante sensores. Además, el algoritmo genético se encarga de realizar todo el proceso evolutivo de los agentes y sus generaciones, aplicando operadores de selección, cruce y mutación. Además, de aplicar una función fitness que evalua que tan buena fue la generación y en esta se otorgan la recompensas por buen rendimiento.

A lo largo de este paper, se van a evidenciar todos los componentes esenciales del proyecto para analizar con detalle su funcionamiento. Se presentarán apartados sobre el diseño del agente, el algoritmo genético, la configuración de los experimentos, la función fitness, un análisis de resultados obtenidos y por último, conclusiones y detalles finales.

II. Diseño del Agente

Para la implementación del agente, el equipo de trabajo decidió utilizar una tabla de desición. Esta consiste en un conjunto de reglas que dictan que acciones tomar según el entorno del agente en ese momento. Esto representa los movimientos del agente durante la partida, dicho comportamiento depende o esta determinado por una matriz de pesos con un algoritmo genético. El objetivo del agente es maximizar su puntuación al recolectar la comida y evitar las colisiones con los bordes o su propio cuerpo.

Dicho comportamiento se representa mediante una matriz de pesos, la cuál es optimizada mediante el Algoritmo Genético, del cual se hará un análisis posteriormente en la documentación. La tabla de decisión esta compuesta por 18 sensores que perciben el entorno del agente en los multiples estados de este. Dichos sensores van a indicar la dirección que va a tomar el agente: seguir directo, girar a la derecha, girar a la izquierda.

A. Objetivo del Agente

Como se mencionaba el objetivo del agente es conseguir la mayor cantidad de puntación posible. Esto se logra mediante la recolección de la mayor cantidad de comidad posible, sin colisionar con los bordes del tablero o consigo mismo. Para ello, es necesario que el agente comprenda su entorno, aprenda a moverse y sobre todo tomar decisiones para ver cual seria la mejor dirección que tomar. Todo esto se evalúa en la función fitness.

B. Tabla de Decisiones

El agente toma decisiones mediante los 18 sensores del entorno que dispone, para esto, se realiza la función ”get_state()” dentro de la clase del juego (class SnakeGame). Esta función,

permite al agente analizar el estado en que se encuentra el tablero y asi detectar peligros, comida y otras caracteristicas del entorno. La función presenta la siguiente estructura:

```
state = [
    # Peligro adelante [0]
    (dir_r and self.is_collision_at(head.x +
        BLOCK_SIZE, head.y)) or
    (dir_l and self.is_collision_at(head.x -
        BLOCK_SIZE, head.y)) or
    (dir_u and self.is_collision_at(head.x,
        head.y - BLOCK_SIZE)) or
    (dir_d and self.is_collision_at(head.x,
        head.y + BLOCK_SIZE)),

    # Peligro a la derecha [1]
    (dir_r and self.is_collision_at(head.x,
        head.y + BLOCK_SIZE)) or
    (dir_l and self.is_collision_at(head.x,
        head.y - BLOCK_SIZE)) or
    (dir_u and self.is_collision_at(head.x +
        BLOCK_SIZE, head.y)) or
    (dir_d and self.is_collision_at(head.x -
        BLOCK_SIZE, head.y)),

    # Peligro a la izquierda [2]
    (dir_r and self.is_collision_at(head.x,
        head.y - BLOCK_SIZE)) or
    (dir_l and self.is_collision_at(head.x,
        head.y + BLOCK_SIZE)) or
    (dir_u and self.is_collision_at(head.x -
        BLOCK_SIZE, head.y)) or
    (dir_d and self.is_collision_at(head.x +
        BLOCK_SIZE, head.y)),

    # Direcci n actual [3-6]
    dir_l,
    dir_r,
    dir_u,
    dir_d,

    # Ubicaci n de la comida [7-10]
    self.food.x < head.x,  # comida a la
        izquierda
    self.food.x > head.x,  # comida a la
        derecha
    self.food.y < head.y,  # comida arriba
    self.food.y > head.y,  # comida abajo

    # Distancia a los bordes (normalizadas)
        [11-14]
    head.x / self.w,           # distancia
        relativa al borde izquierdo
    (self.w - head.x) / self.w, # distancia
        relativa al borde derecho
    head.y / self.h,           # distancia
        relativa al borde superior
    (self.h - head.y) / self.h, # distancia
        relativa al borde inferior

    # Informaci n adicional sobre la comida
        [15-17]
    moving_toward_food,  # Si nos estamos
        moviendo hacia la comida
    1.0 - food_dist_x,   # Cercan a a la
        comida en X (1=cerca, 0=lejos)
    1.0 - food_dist_y,   # Cercan a a la
        comida en Y (1=cerca, 0=lejos)
]
```

Listing 1. Código de ejemplo en Python

Se puede observar que la variable state, contiene todos los 18 sensores del entorno. La explicación de la función de cada uno de los sensores es:

1) **Indices 0 - 2:** Se encargan de verificar si hay una colisión al frente, derecha o izquierda, de acuerdo con la dirección actual del agente.
2) **Indices 3 - 6:** Se encarga de la dirección actual, el agente puede moverse: izquierda, derecha, arricha o abajo.
3) **Indices 7 - 10:** Este sensor indica la posición en la que se encuentra la comida. Usa de referencia la posición de la cabeza del agente.
4) **Indices 11 - 14:** Estos se encanrgan de la distancia relativa con los bordes del tablero
5) **Índice 15:** Este es un sensor que indica si el movimiento que realizo el agente lo pone en dirección o esta cerca de la comida
6) **Indices 16 - 17:** Indican la distancia del agente con la comida en los ejes X y Y del tablero.

Posteriormente, la acción que realiza el agente se determina mediante un producto punto entre el estado y la matriz de pesos, dando un vector de 3 valores, los cuales son las posibles acciones que puede hacer el agente. Además, el agente posee ajustes para modificar y hacer mejor su comportamiento: Evitar colisiones, Preferencia por la Comida, Previención de Ciclos. Todas estas funciónes serán detalladas en la sección de funciones clave del agente. Esta estructuración hace que el agente sea capaz de adaptarse a los diferentes estados que va a entrar a lo largo de los multiples juegos y buscar el mejor rendimiento.

C. Inicialización y Bias

Cuando se inicia la partida con el agente, los pesos de la matriz se pesos se inicializan de forma aleatoria pero utilizando un Bias para favorecer comportamientos del agente. Este consta de tres puntos clave:

1) Dar mayor peso negativo a detectar peligros

```
    self.weights[0, :] = -5.0  #Peligro
        adelante - evitar esta direcci n
    self.weights[1, 1] = -5.0  #Peligro a la
        derecha - no girar a la derecha
    self.weights[2, 2] = -5.0  #Peligro a la
        izquierda - no girar a la izquierda
```

Listing 2. Código de ejemplo en Python

2) Dar peso positivo a ir en dirección de la comida

```
    self.weights[7:11, :] =
        np.random.uniform(0.5, 1.5, size=(4,
        3))  # Direcci n de la comida
```

Listing 3. Código de ejemplo en Python

3) Premiar el moverse hacia la comida o estar cerca

```
    self.weights[15, 0] = 3.0  # Si va hacia
        la comida, seguir recto
    self.weights[16:18, :] =
        np.random.uniform(1.0, 2.0, size=(2,
        3))
```

Listing 4. Código de ejemplo en Python

D. Funciones Clave del Agente

Se implementó una clase llamada DecisionTable, la cuál contiene las funciones mas relevantes relacionadas con el comportamiento y el diseño del agente. Estas son las funciones que cumplen en el esquema del proyecto:

1) **get_action():** Esta es la función principal en el proceso de toma de decisiones del agente. Primeramente, se realiza un producto punto entre el estado y la matriz de pesos, dando como resultado un vector con tres valores:
   a) valor de seguir recto
   b) valor de girar a la derecha
   c) valor de girar a la izquierda

   Existen diferentes ajustes que posee el agente para hacer que éste tenga mejores comportamientos. **Evitar colisiones:**

```
    if danger_ahead > 0.5:
        q_values[0] = -100  # Penalizar
            fuertemente ir hacia adelante
```

Listing 5. Código de ejemplo en Python

Preferencia por la comida:

```
    if best_dir_to_food >= 0 and
        random.random() < 0.8:  # 80% de ir
        hacia la comida
        # Verificar que esa direcci n no sea
            peligrosa
        if (best_dir_to_food == 0 and not
            danger_ahead) or \
           (best_dir_to_food == 1 and not
               state[1]) or \
           (best_dir_to_food == 2 and not
               state[2]):
            q_values[best_dir_to_food] += 5.0
                # Dar un bonus significativo
```

Listing 6. Código de ejemplo en Python

Prevención de Ciclos:

```
    if has_cycle:
        if random.random() < 0.6:  # 60% de
            probabilidad de romper ciclos
            # Ruido significativo para romper
                patrones
            q_values += np.random.normal(0,
                2.0, size=q_values.shape)
            # Intentar moverse hacia una
                direcci n completamente
                diferente
            if random.random() < 0.3:  # 30%
                chance de cambiar
                completamente
                current_action =
                    np.argmax(q_values)
                q_values[current_action] =
                    -10  # Penalizar acci n
                    actual
```

Listing 7. Código de ejemplo en Python

Para finalizar, se toma la acción que tenga un valor mayor en la variable q_values y se guarda en un historial de acciones para realizar un análisis.

2) **_update_action_history():** Almacena las acciones recientes del agente para permitir análisis de comportamiento repetitivo.

```
    self.recent_actions.append(action)
        if len(self.recent_actions) >
            self.max_history:
            self.recent_actions.pop(0)  #
                Eliminar la acci n m s
                antigua
```

Listing 8. Código de ejemplo en Python

3) **_check_for_cycles():** Esta función detecta si el agente esta repitiende un patron de movimientos. Importante ya que el agente debería tener un juego fluído y no realizar movimentos enciclados.
4) **crossover():** Realiza un cruce entre la tabla de decisión de un agente y la de uno nuevo. El cruce tiene un 50% de probabilidad de heredar los pesos de la matriz del padre:

```
    child_weights = np.copy(self.weights)

        # Cruce uniforme (50% de probabilidad
            de heredar de cada padre)
    rows, cols = self.weights.shape
    for i in range(rows):
        for j in range(cols):
            # 50% de probabilidad de heredar
                de cada padre
            if random.random() < 0.5:
                child_weights[i][j] =
                    other.weights[i][j]

        # Crear nueva tabla de decisi n con
            los pesos resultantes
    return DecisionTable(child_weights)
```

Listing 9. Código de ejemplo en Python

5) **mutate():** Aplicada una mutuación a la tabla con la taza específicada en los parámetros de los experimentos. De esta forma podemos ver que existe una distinción entros lo pesos entros los nuevos agentes:

```
    for i in range(rows):
        for j in range(cols):
            if random.random() <
                mutation_rate:
                # A adir ruido gaussiano con
                    desviaci n moderada
                self.weights[i][j] +=
                    np.random.normal(0, 0.5)

                # 10% de probabilidad de
                    mutaci n m s dr stica
```

```
                if random.random() < 0.1:
                    # Mutaci n  m s agresiva
                        (cambio de signo o
                        reemplazo total)
                    if random.random() < 0.5:
                        # Cambiar signo
                        self.weights[i][j] *=
                            -1
                    else:
                        # Valor completamente
                            nuevo
                        self.weights[i][j] =
                            np.random.normal(0,
                            1.5)

    return self
```

Listing 10. Código de ejemplo en Python

III. Diseño del Algoritmo Genético

Esta parte de la implementación del código es muy relevante, ya que, es la forma por la que el agente evoluciona y es seleccionado como candidato. Debido a requerimientos, este agente de optimizar su rendimiento de forma progresiva mejorar su puntación en el juego. Este algoritmo simula un proceso evolutivo, en el cual una población de agentes compite generación tras generación, mejorando sus habilidades a través de selección, cruce y mutación.

A. Objetivo del Algoritmo Genético

El objetivo del algoritmo genético consiste en continuar optimizando progresivamente el comportamiento del agente Snake de modo que este pueda aprender a jugar de manera autónoma de la manera más eficiente. A través de la evolución basada en la selección natural, el algoritmo intentará crear agentes que maximicen la recolección de alimentos, eviten los choques, exploren más el tablero y vivan lo más tiempo posible.

B. Componentes del Algoritmo Genético

1) ***Evaluación****:* Este es uno de los procesos mas importantes dentro del ciclo evolutivo. El objetivo principal de esta etapa, es medir el rendimiento de los agentes a lo largo de las partidas en los entornos del juego, este rendimiento debe ser reflejado con un valor numérico, de esta forma se puede realizar la comparación entre cuales agentes tienen un mejor rendimiento bajo un contexto otorgado. Para esto, se hace uso de la llamada función Fitness, que se analizará mas adelante.

Para la implementación en el codigo, cada agente realiza tres partidas con semillas predefinidas, para asegurar una medición exhaustiva y darle mayor oportunidad al agente de realizar una buena partida. Durante cada una de estas partidas, nuestra evaluación registra diferentes métrica que se pondrán a prueba para realizar los análisis:

1) Cantidad de comida recolectada
2) Número total de pasos
3) Eficiencia de la ruta tomada a la comida
4) Acercamientos a la comida
5) Exploración del tablero
6) Evitar muertes y colisiones (tiempo en partida)
7) Penalizaciones por malos comportamientos

Todas estas metricas con recogidas y aplicadas en una fórmula para indiciar cual fue el rendimiento del agente en la partida. Sobre esta formula se va a hacer un analisis mas adelante.

2) ***Selección****:* El proceso de selección es importante en el ciclo evolutivo para ver que las generación avances y haya mejora entre estas. Este consta de seleccionar cromosomas de acuerdo a la función fitness. Esta estapa es importante, ya que, como parte de la selección natural, solo los individuos mas aptos sobrevevien. Esto es lo que sucede en esta parte, se pueden a prueba los agentes y se seleccionan los mejores agentes para que estos sean los que se reproduzcan hacia nuevos agentes y nuevas generaciones.

Se implementó el método de selección por ruleta, en el que cada agente tiene una probabilidad de ser seleccionado proporcional a su fitness. Si algunos agentes tienen valores negativos, los fitness se ajustan para que todos sean positivos, permitiendo un reparto justo.

3) ***Cruce****:* Esta etapa consiste en cruzar cromosomas con posibilidad de cruzamiento. De esta forma garantizamos que la decendencia no sea una copia exacta de los padres y entre en juego este concepto de variabilidad entre las generaciones. Es uno de los componentes más importantes del algoritmo, ya que permite mezclar comportamientos exitosos para crear soluciones potencialmente mejores.

El algoritmo implementa tres tipos de cruce:

1) Cruce uniforme: Cada peso de la tabla del hijo se hereda aleatoriamente de uno de los dos padres.
2) Cruce de punto: Se selecciona un punto de corte en la matriz de pesos, y los pesos posteriores se toman del segundo padre.
3) Cruce de dos puntos: Se selecciona un segmento completo que se intercambia entre los padres.

Esto permite explorar diferentes tipos de cruce en los futuros agentes.

4) ***Mutación****:* Esta etapa consiste en mutar la decendencia de los padres, en alguna de sus características. Esto se logra de forma aleatoria, cambiando una variación en sus genética. El objetivo de esta etapa, es que los agentes sean diferentes entre ellos mismos y dar espacia a explorar nuevas combinaciones y con esto nuevas soluciones al juego abarcando diferentes estados.

En esta implementación, la mutación ha sido diseñada de forma adaptativa y diferenciada, es decir, su comportamiento se ajusta dinámicamente según el progreso del proceso evolutivo y el contexto de cada parte del cromosoma del agente:

1) Tasa Base Adaptativa
2) Factor Estancamiento
3) Tasa Diferenciada por Sección

5) ***Elitismo****:* El elitismo es una estrategia que garantiza que los mejores agentes de una generación se conserven intactos para la siguiente. Esto previene que el proceso evolutivo destruya soluciones óptimas por accidente durante el cruce o la mutación.

En esta implementación, se especifica un parámetro que indica cuántos agentes se deben transferir directamente a la siguiente generación

6) ***Ciclo Evolutivo****:* Este es el concepto general de como funciona el algoritmo evolutivo, son todos los pasos necesarios que hay que seguir para realizar la evolución natural. Este algoritmo se le establecen el numero de agentes que van a realizar el experimiento y durante cuanta gereraciones. Para cada generación se realizan los pasos mencionados anteriormente:

1) Evaluar los agentes con la función fitness
2) Guardar estadisticas
3) Aplicar elitismo
4) Completar la población con: selección, cruce y mutación
5) Reemplazar la población

Este ciclo evolutivo permite que los agentes mejoren progresivamente su desempeño en el entorno, guiados por la retroalimentación proporcionada por la evaluación.

C. Funciones Clave del Algoritmo Genético

Se implementó una clase llamada GeneticAlgorithm, la cuál contiene las funciones mas relevantes relacionadas con toda la parte evolutiva de los agentes. Estas son las funciones que cumplen en el esquema del proyecto:

1) **Inicialización:** En la función init, se establecen los parámetros configurables como por ejemplo: tamaño de la población, número de generaciones, tasa de mutación, tasa de cruce, tamaño de la élite. Además de un mecanismo para establecer diferentes tipos de cruce.
2) **fitness():** Como se ha mencionado anteriormente en el artículo, el objetivo de la función fitness es evaluar el rendimiento de un agente y en base a este rendimiento retornar una putación numérica. De esta forma se puede saber cuáles son los mejores agentes. El primer aspecto de implementación es la parte de semillas fijas:

```
base_seed = 42  # Semilla base arbitraria pero fija
seeds = [base_seed + i * 1000 for i in range(num_games)]  # Generar semillas espaciadas
```

Listing 11. Código de ejemplo en Python

De esta forma garantizamos que los agentes sean evaluados bajo el mismo entorno, haciendo la comparación mas justa. La función implementada hace recolección de métricas que evaluan el redimiento del agente como:

a) eficiencia de movimiento
b) acercamientos a la comida
c) movimientos diferentes
d) evitar colisiones
e) evitar ciclos
f) pasos hasta la comida

Estos componenetes importantes para hacer el cálculo de la función fitness. En la parte del verificar ciclos, se hace el uso de un historial de movimientos recientes y se comparan para verificar si el agentes esta siguiente un patrón:

```
if len(last_positions) >= 8:
    for cycle_len in [2, 3, 4]:
        if len(last_positions) >= cycle_len * 2:
            recent = last_positions[-cycle_len:]
            previous = last_positions[-2*cycle_len:-cycle_len]
            if recent == previous:
                repeated_cycles += 1
                break
```

Listing 12. Código de ejemplo en Python

Posteriormente, otro componente imporante es el análisis del moviemento, si estos movimientos lo acerca la comida o a una colisión.

```
    # Analizar si se acerca a la comida
    if curr_distance < prev_distance:
        movement_efficiency += 1
        direct_path_bonus += 0.3
        successful_food_approaches += 1
        consecutive_approach_food += 1
    else:
        # Penalizaci n por alejarse de la comida
        direct_path_bonus -= 0.1
        consecutive_approach_food = 0

    # Bonus exponencial por aproximaci n consistente
    if consecutive_approach_food >= 3:
        direct_path_bonus += consecutive_approach_food * 0.5

    prev_distance = curr_distance

    # Verificar si encontr  comida
    if score > prev_score:
        foods_reached += 1
        food_eaten_in_game += 1
        # Registrar cu ntos pasos tom llegar a esta comida
        avg_steps_per_food.append(steps_since_last_food)
        steps_since_last_food = 0

        # Bonus por comer comida (mayor bonus mientras m s crece)
        food_bonus_multiplier = 1 + (food_eaten_in_game * 0.5)
        direct_path_bonus += 10.0 * food_bonus_multiplier

        # Reiniciar distancia para pr xima comida
        prev_distance = None
```

Listing 13. Código de ejemplo en Python

Además, como el objetivo del agente es coseguir mayor puntación, se incentiva a este a conseguir comida de forma consecutiva, exploración, evitar peligros:

```
    for i in range(total_score):
```

```
        # Cada comida vale un 10% m s que la
            anterior
        food_value = base_food_value * (1 +
            (i * 0.1))
        incremental_food_points += food_value
```

Listing 14. Código de ejemplo en Python

```
    exploration_bonus = unique_positions * 0.5

    if max_snake_length > 5 and
        unique_positions < total_steps * 0.5:
        confined_space_bonus = 30
```

Listing 15. Código de ejemplo en Python

```
    basic_avoidance = wall_avoidance_count *
        4.0
    critical_avoidance = near_death_avoidance
        * 10.0
```

Listing 16. Código de ejemplo en Python

```
    if total_score == 0:
        early_death_penalty = 200 * (1 -
            min(1.0, total_steps/100))
```

Listing 17. Código de ejemplo en Python

Por último, se hace una recolección de todas las métricas obtenidas por las partidas del agente y se devuelve el valor númerico de su rendimiento. La fórmula utilizada es la siguente:

```
    fitness = (
        food_points +                    #
            Valor de comida (ahora m s
            equilibrado)
        survival_points +                #
            Valor de supervivencia
            (incrementado)
        route_efficiency * 2.0 +         #
            Eficiencia de ruta (reducido de
            4.0 a 2.0)
        food_points_per_step * 2.0 +     #
            Nueva m trica de eficiencia
        food_approach_bonus +            #
            Aproximaci n a comida
        food_consistency_bonus +         #
            NUEVO: Consistencia entre juegos
        direct_path_bonus * 2.0 +        #
            Direcci n hacia comida
            (ligeramente reducido)
        exploration_bonus +              #
            Exploraci n
        avoidance_bonus +                #
            Evitaci n de obst culos
            (significativamente aumentado)
        (movement_efficiency * 1.0) -    #
            Eficiencia de movimiento
        early_death_penalty -            #
            Penalizaci n gradual por muerte
            temprana
        repetition_penalty               #
            Penalizaci n por ciclos (ahora
            exponencial)
        )
```

Listing 18. Código de ejemplo en Python

3) **selection():** En esta etapa, se seleccionan dos agentes como padres para realizar un cruce y generar decendencia en la siguiente generación. Para esta implementación se decidió utilizar un formato torneo para evitar la dominancia de individuos repetitivos. Este consiste en seleccionar un grupo aleatorio de agentes y de ellos seleccionar el que tenga el mejor fitness, de esta forma se obtienen los padres. La selección del torneo de hace de la siguiente manera:

```
    if self.stagnation_counter > 5:
    self.tournament_size = max(3,
        self.tournament_size - 1)
    elif len(self.improvement_rate_history) >
        2 and
        sum(self.improvement_rate_history[-2:])
        > 0.05:
    self.tournament_size = min(8,
        self.tournament_size + 1)
```

Listing 19. Código de ejemplo en Python

Esta implementación hace que si no hay muchos avances se reduce la presión selectiva y si hay mejoras se aumenta. Además, existe la probabilidad de que los padres sean completamente aleatorios, de esta forma no se cruzan los mismo agentes. Por otra parte, exisiten penalizaciones que ocurren durante el proceso de selección:

a) Candidad de descendencia ya generada
b) Similitud genética con el otro padre

Se decidió realizar una forma para garantizar la diversidad entre los padres, para que no sean iguales y el hijo sea una producto de una mayor combinación:

```
    # Asegurar que no se seleccione dos veces
        el mismo individuo
        if selected_indices[0] ==
            selected_indices[1]:
            # Reemplazar el segundo  ndice
                con otro individuo aleatorio
                distinto
            candidates = [i for i in
                range(len(self.population))
                if i != selected_indices[0]]
            if candidates:
                selected_indices[1] =
                    random.choice(candidates)
        return selected_indices
```

Listing 20. Código de ejemplo en Python

4) **crossover():** La siguiente etapa necesaria para crear nuevas generaciones es la del cruce. El objetivo de esta etapa es generar un nuevo agente hijo combinando dos agentes padres mediante el cruce genético. Este proceso es muy importante, ya que, nos permite experimentar nuevos comportamientos y nuevos resultados. Para ello, fue necesario la aparición de un parámetro que indica la tasa de cruce, la probabilidad de que el cruce ocurra

y si no se da esa posibilidad, el hijo es una copia del primer padre. Por otro lado, existen tres tipos de cruces que se implementaron:

```
crossover_type = getattr(self,
    'crossover_type', 'one_point')
```

Listing 21. Código de ejemplo en Python

El cruce uniforme:

```
# 50% de probabilidad de heredar de cada
    padre
if random.random() < 0.5:
    child_weights[i][j] =
        parent1.weights[i][j]
else:
    child_weights[i][j] =
        parent2.weights[i][j]
```

Listing 22. Código de ejemplo en Python

El cruce de 2 puntos:

```
 # Seleccionar dos puntos de cruce
total_genes = rows * cols
point1 = random.randint(0, total_genes -
    2)
point2 = random.randint(point1 + 1,
    total_genes - 1)
```

Listing 23. Código de ejemplo en Python

El cruce de un punto:

```
crossover_point = random.randint(0, rows
    * cols - 1)
row_idx = crossover_point // cols
col_idx = crossover_point % cols

if i > row_idx or (i == row_idx and j >=
    col_idx):
    child_weights[i][j] =
        parent2.weights[i][j]
```

Listing 24. Código de ejemplo en Python

Independientemente del tipo de cruce realizado, el hijo resultantate siempre crea su tabla de decisión. Al utilizar diferentes tipos de cruce, hace que haya mas diversidad en los resultados y no sigan siempre un patrón establecido.

5) **mutate():** El objetivo principal de esta etapa del proceso de evolución, modificar aleatoriamente los pesos del agente para introducir variabilidad genética. La tasa de mutación no es fija, sino que se reduce gradualmente a medida que avanzan las generaciones, permitiendo alta exploración al inicio y mayor refinamiento al final:

```
# 1. Tasa base que disminuye gradualmente
    con las generaciones
# Calculamos el progreso de evoluci n
    normalizado (0 al inicio, 1 al final)
if hasattr(self, 'evolve_progress') and
    self.num_generations > 0:
    evolution_progress =
        self.evolve_progress /
        self.num_generations
else:
    evolution_progress = 0.5  # Valor
        intermedio por defecto

# La tasa base comienza alta y disminuye
    gradualmente
base_rate = self.mutation_rate * (1.0 -
    evolution_progress * 0.7)
```

Listing 25. Código de ejemplo en Python

Posteriormente, si hay algun tipo de estancamiento, hay que aumentar la tasa de mutación para hacer una variabilidad en una de sus caracteristicas para cambiar este funcionamiento:

```
# 2. Aumentar significativamente la tasa
    cuando hay estancamiento
stagnation_factor = min(0.5,
    self.stagnation_counter * 0.1)
```

Listing 26. Código de ejemplo en Python

Se calcula un promedio de las mejoras recientes. Si han sido positivas, se reduce la mutación; si ha habido retroceso, se aumenta:

```
recent_improvement = 0
if len(self.improvement_rate_history) > 0:
    # Promedio de mejoras recientes
        (negativo si empeor )
    recent_improvement =
        sum(self.improvement_rate_history
    [-3:]) / min(3,
        len(self.improvement_rate_history))

# Si hay mejoras recientes, reducir
    mutaci n. Si hay deterioro,
    aumentarla.
improvement_factor = max(-0.2, min(0.2,
    -recent_improvement))

# Combinar factores - el estancamiento
    siempre aumenta la mutaci n
# mientras que las mejoras pueden
    reducirla
adjusted_rate = base_rate * (1.0 +
    stagnation_factor +
    improvement_factor)

# Asegurar que la tasa est  dentro de
    l mites razonables
adjusted_rate = max(0.01, min(0.8,
    adjusted_rate))
```

Listing 27. Código de ejemplo en Python

La otra forma de la tasa de mutación, es mediante los pesos obtenidos por los diferentes sensores del agente:

```
section_rates = {
    'danger': adjusted_rate * 0.5,
        # Menor tasa para pesos de
        detecci n peligro
    'direction': adjusted_rate * 0.6,
        # Menor tasa para pesos de
        direcci n
```

```
        'food_location': adjusted_rate * 1.2,
            # Mayor tasa para pesos de
            ubicaci n comida
        'borders': adjusted_rate * 1.0,
            # Tasa normal para distancias a
            bordes
        'food_advanced': adjusted_rate * 1.3
            # Mayor tasa para info avanzada
            de comida
    }
    # Aplicar mutaci n con tasas
        diferenciadas por secci n
    for i in range(rows):
        # Determinar la secci n a la que
            pertenece esta fila
        if i <= 2:
            section_rate =
                section_rates['danger']
        elif i <= 6:
            section_rate =
                section_rates['direction']
        elif i <= 10:
            section_rate =
                section_rates['food_location']
        elif i <= 14:
            section_rate =
                section_rates['borders']
        else:
            section_rate =
                section_rates['food_advanced']
```

Listing 28. Código de ejemplo en Python

6) **evolve():** Esta función se encarga de ejecutar el ciclo completo de evolución a un número fijo de generaciones establecidos por parámetro. Es el cerebro del algoritmo, controla la evaluación de la población, la selección de padres, la generación de descendencia, la mutación adaptativa y la aplicación del elitismo. Se inicia con la verficación de los parámetros y reiniciando todos los contadores y estructuras, verifica que ya exista la población con la que se va a trabajar. Posteriormente, todos los agentes son evaluados por la función fitness:

```
    fitnesses = [self.fitness(agent) for
        agent in self.population]

    best_fitness = fitnesses[max_fitness_idx]
    self.best_fitness_history
    .append(best_fitness)
```

Listing 29. Código de ejemplo en Python

Se guarda el mejor agente y actualizamos la métricas correspondientes. Posteriormente, se encarga de la detección de estancamiento, variables utilizadas en la parte de la mutación y selección:

```
    improvement_rate = (best_fitness -
        prev_best) / (prev_best + 0.01)
    if best_fitness > self.best_fitness_ever
        * 1.01:
    self.stagnation_counter = 0
    else:
        self.stagnation_counter += 1
```

Listing 30. Código de ejemplo en Python

Además, se encarga de la parte del elitismo, de cuidar a algunos agentes y colocarlos en la siguiente generación. De esta forma nos aseguramos que el algoritmo continue mejorando. El resto de la población se crea mendiante las demás etapas del algoritmo: selección, cruce y mutación.

```
    parent_indices = self.selection(fitnesses)
    child = self.crossover(parent1, parent2)
    child = self.mutate(child)
```

Listing 31. Código de ejemplo en Python

Una vez creada la nueva población, se reinicia el ciclo. Al terminar todas las generaciones, se reevalúa la población y se retorna el mejor agente final entrenado.

IV. Configuración Experimental

Como se mencionaba al inicio de la documentación, para probar el codigo evolutivo implementado, se iba a realizar tres experimentos distintos para comprobar el redimiento del algoritmo evolutivo.

Cada experimento consistió en un proceso de evolución completo de cien generaciones, con variaciones en el tamaño de la población (cantidad de agentes), la tasa de mutación y el elitismo, entre otros factores.

A continuación, se pueden observar los parámetros utilizados en los distintos proyectos:

1) **Experimento 1:**

```
    {
        'name': 'Experimento 1',
        'pop_size': 50,
        'generations': 100,
        'mutation_rate': 0.35,
        'elitism': 6,
        'base_seed': int(time.time()) % 10000
    },
```

Listing 32. Código de ejemplo en Python

Como se puede observar el total de la población en este experimento es de 50 agentes. Una tasa de mutación de 35% lo que es un tasa medianamente alta para que haya una buena tasa de cambios y por ende, diferente exploración. Además, un elitismo bajo para que no influya tanto la generación anterior de agentes en el futuro de las generaciones.

2) **Experimento 2:**

```
    {
        'name': 'Experimento 2',
        'pop_size': 60,
        'generations': 100,
        'mutation_rate': 0.45,
        'elitism': 4,
        'base_seed': (int(time.time()) %
            10000) + 5000
    },
```

Listing 33. Código de ejemplo en Python

Para el segundo experimento, se hizo un incremento en la problación, ahora son un total de 60 agentes. Por

otra parte, se incrementa la tasa de mutación a 45%, lo que hace hayan mas caracteristicas que cambian y por ende mayores posibles soluciones. Además, el elitismo seleccionado es mas bajo, lo que se prioriza a la nuevas generaciones para buscar nuevas soluciones.

3) **Experimento 3:**

```
    {
        'name': 'Experimento 3',
        'pop_size': 40,
        'generations': 100,
        'mutation_rate': 0.25,
        'elitism': 10,
        'base_seed': (int(time.time()) %
            10000) + 10000
    }
```

Listing 34. Código de ejemplo en Python

Como último experimento, la población es reducida en comparación a los otros dos experimentos, un total de 40 agentes. Además, se reduce la tasa de mutación a un 25% lo cuál hace que se trabajen con los movimientos y características exitosas de la población. Por último, el elitismo es aumento a 10, lo que hace que la generación anterior tengo un impacto mayor en el futuro de las generaciones haciendo que el conocimiento sea mas rápido.

V. Resultados y Visualización

En este apartado se van a presentar los resultados de la ejecución del codigo, con la implementación del algoritmo genético. Se comparan distintas métricas relevantes a lo largo de 100 generaciones de evolución:

1) **Evolución del Fitness Máximo:** Muestra el mejor valor de fitness alcanzado un agente en su generación.

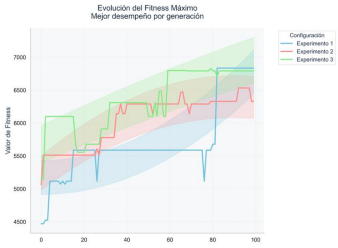

Fig. 1. Evolución Fitness Máximo

2) **Evolución del Fitness Promedio:** Este es el gráfico principal para ver la evolución del redimiento de los agentes a lo largo de las generaciones.

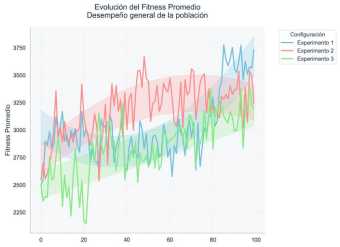

Fig. 2. Evolución Fitness Promedio

3) **Diversidad en la Población:** Muestra la variabilidad genética entre los agentes.

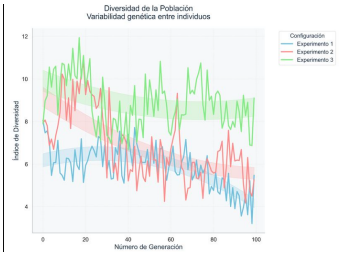

Fig. 3. Diversidad en la Población

4) **Evolución de la Capacidad de Alimentación:** Registra la cantidad de alimento recolectada en cada generación como desempeño.

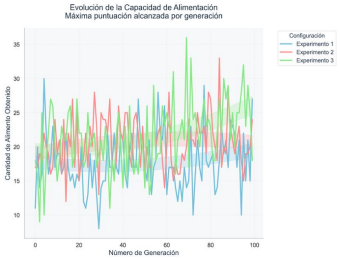

Fig. 4. Evolución Capacidad de Alimentación

VI. Discusión y Análisis

A continuación, se va a realizar una discusión y un análisis de los resultados mostrados anteriormente:

A. Evolución Fitness Máximo

Como se mencionó anteriormente, la figura 1, muestra la evolución del mejor fitness alcanzado en cada generación en cada uno de los tres experimentos. Haciendo un análisis se puede observar que en los tres experimientos se presenta una tendencia ascendente en el fitness máximo, lo que, inidica que a medida que pasan las generación se logra un mayor fitness, los agentes tieden a realizar una mejor puntación máxima. Sin embargo, se pueden evaluar las diferentes tendencias, por ejemplo, el experimento 3 alcanza una mayor cantidad de fitness máximo a traves de las generaciones. Esto puede ser debido a que es el experimiento que posee la mayor cantidad de elitismo y la menor cantidad de tasa de mutación, lo que

hace, que se exploren soluciones buenas. Por otro lado, el experimento 2, es el que presenta la tendencia mas progresiva y menos brusca. Se puede ver como va mejorando poco a poco conforme pasan las generaciones. Por lo que, mediante la tendencia a incrementar, se puede evidenciar que el algoritmo cumple su función de hacer que los agentes tengan mejores resultados conforme pasan las generaciones.

B. Evolución Fitness Promedio

Para este gráfico, puede ser el más significativo para el análisis de los resultados obtenidos durante la ejecución del algoritmo. Este nos permite ver en promedio, el fitness de cada generación. Esto es un claro indicador para ver que tan bien lo hace una generación con respecto a otra. A primera vista, se puede ver que el fitness promedio también aumenta a lo largo de las generaciones, lo que demuestra mejoras generacionales y no solo de individuos aislados. Haciendo un análisis mas detallado del gráfico, se puede observar que en promedio, el mejor experimento es el segundo. Se podría concluir que una mayor diversidad genética favorece el aprendizaje de la población en general y no solo de un individuo máximo. El experimento 3 tiene el promedio mas bajo de los tres experimentos, esto puede ser debido a que es la mayor elitismo tiene y menor tasa de mutación lo que hace que no exploren muchas soluciones. No obstante, de igual forma, se puede observar que la tendencia de la gráfica en general tiende a incrementar a excepción de algunas generaciones lo cuál es algo normal. Esto es una clara referencia a que las generaciones tienen a mejorar su rendimiento con el paso de futuras generaciones, lo cual, es un claro indicador de aprendizaje realizado durante el experimento.

C. Diversidad en la Población

El siguiente gráfico, como se mencionaba, muestra la variabilidad genética entre individuos durante las generaciones. Lo que nos indica, que tan diferente es una generación entre sus propios agentes. Para este caso, podemos ver un comportamiento insual o no esperado, ya que, el Experimento 3 es el que posee mayores indices de diversidad, lo cuál, no es esperado ya que, debido a su alto elitismo y baja tasa de mutación deberia tener menor índice de diversidad. Por otro lado, el experimento 2 y 3 tienes altos indices de diversidad al inicio de sus generación pero estos tienen a decaer. Esto puede deberse a que al final de las generaciones no hay tanta diversidad ya que, se poseen mejores soluciones que al inicio de los experimentos. Se puede concluir que la pérdida gradual de diversidad indica que la evolución selecciona y refuerza comportamientos exitosos, reduciendo las variaciones aleatorias.

D. Evolución Capacidad de Alimentación

Por último, la figura 4 nos muestra la cantidad de alimento recolectado por generación. Haciendo un análisis del gráfico, se observa una mejora general en la capacidad de alimentación en generaciones más avanzadas. Sin embargo, esta posee grandes variaciones durante todas las generaciones. El sentido de este, es que, la puntación fitness no solo toma en cuanta la comidad recolectada, si no, la eficiencia para realizar dicha recolección de comida. Por lo que, esos altibajos se puede deber a mayor dispersión debido a la aleatoriedad natural del entorno al momento de colocar la comida. Los experimentos 2 y 3 logran mayores puntuaciones máximas en alimentación, esto puede deberse a multiples causas como la selcción de parámetros utilizados. Sin embargo, se puede concluir, que los agentes aprenden a recolectar alimentos de manera más eficiente, pese a la variabilidad del entorno.

VII. Conclusiones y Trabajo Futuro

Al inicio del proyecto, se planteó como objetivo principal desarrollar un algoritmo en el cuál un agente puediera jugar el juego Snake y mediante la evolución de este, fuera aprendiendo a jugarlo cada vez mejor, mejorando su rendimiento. La meta, era lograr que el agente maximizara su tiempo de supervivencia en el juego, recolección de comida y movimiento inteligentes evitando obstáculos.

El análisis de gráficos realizado, fue de suma importancia para evidenciar el aprendizaje progresivo que tenían los agentes a través de sus generaciones. El aumento de fitness y su promedio funcionaron como punto de partida para realizar dichas conclusiones. Los agentes evolucionaron hacia estrategias más eficientes, logrando mejor rendimiento en términos de supervivencia y de recolección de alimentos. Esto confirma que el enfoque evolutivo fue exitoso. Durante los experimentos, se exploraron diferentes configuraciones de parámetros, contrastando enfoques posibles en las población, haciendo que unas tuvieran mas convergencia que otras.

El uso de un algoritmo adaptativo, que ajusta la tasa de mutación y los mecanismos de selección según el progreso evolutivo, demostró ser una buena estrategia para enfrentar problemas. Esta adaptabilidad permitió evitar estancamientos evolutivos y promover tanto la exploración de nuevas soluciones como la consolidación de comportamientos exitosos. Además, un concepto importante en el algoritmo fue la validación de movimientos ciclicos, usandolo como parámetro para calcular el fitness y de esta forma valir que el agente sobrevive no por que esta en un bucle. De esta forma, se evita que futores agentes tengan este comportamiento y se evoluciones hacia soluciones buenas y factibles.

Por último, como recomendaciones futuras, es importante considerar que la aleatoriedad del entorno introduce cierta variabilidad en las evaluaciones de fitness, lo que puede afectar la consistencia entre generaciones. Por lo que, es imporantes verificar la evaluación el fitness y posiblemente separar responsabilidades aplicando otras métricas a parte del fitness para realizar un análisis de resultados con mayor detalle. Por otra parte, se pueden hacer pruebas con generaciones que ya poseen un conocimiento del juego para ver que tanto pueden aprender en comparación a las que no tienen conocimiento previo.

No obstante, este proyecto nos ayudó a evidenciar que un agente puede aprender y mejorar su comportamiento en un entorno dinámico y cambiente como lo puede ser el juego

de Snake. Por lo qué, vemos interesante como aplicamos una teoría biológica a la programación para lograr soluciones efectivas, como lo son los algoritmos genéticos evolutivos.

REFERENCIAS

[1] Freecodecamp, *Python + PyTorch + Pygame Reinforcement Learning – Train an AI to Play Snake*, 2022. [Video]. Disponible en: https://www.youtube.com/watch?v=L8ypSXwyBds.