Universidad Autónoma del
Estado de Morelos

Universidad Autónoma del Estado de Morelos

Snake Inteligente

Autor:
Andrés López de Nava Cárdenas

Profesor:
Dr. Jorge Hermosillo

Proyecto presentado para la obtención del grado de
Licenciatura en inteligencia artificial
Facultad de ciencias (CInC)
Universidad Autónoma del Estado de Morelos

13/Junio/2024

Contents

1 Resumen

El proyecto aborda la optimización del camino en el juego Snake utilizando un algoritmo genético. El objetivo es minimizar los movimientos necesarios para que la serpiente recoja todas las frutas en un tablero donde todas las frutas aparecen al mismo tiempo. Se utilizó selección por torneo, cruce de dos puntos y mutación puntual. Los resultados mostraron que el algoritmo genético puede encontrar caminos eficientes, aunque su efectividad disminuye en problemas más grandes.

2 Introducción

En este proyecto, se crea un tablero del juego Snake donde todas las frutas aparecen simultáneamente y la serpiente requiere una entrada para cada movimiento. La posición inicial de la serpiente y las frutas es fija en cada ejecución. Este problema es relevante ya que demuestra cómo un algoritmo puede resolver tareas complejas diseñadas para seres humanos, ofreciendo una visión sobre la capacidad de optimización de algoritmos genéticos en problemas combinatorios.

Los algoritmos genéticos, inspirados en los principios de la evolución natural, son una poderosa herramienta para resolver problemas de optimización en áreas como la logística, la planificación de rutas y la robótica.

En logística, por ejemplo, encontrar la ruta óptima para recolectar y entregar bienes puede ser formulado de manera similar al problema de la serpiente que recolecta frutas. Los algoritmos genéticos pueden ayudar a reducir costos y mejorar la eficiencia en la cadena de suministro. De manera similar, en la robótica, planificar el movimiento de un robot para recoger objetos dispersos en un espacio confinado es crucial para aplicaciones industriales y de servicio.

Este proyecto, por tanto, no solo aborda un problema específico en un entorno de juego, sino que también sirve como modelo para entender cómo los algoritmos genéticos pueden ser aplicados a problemas reales de optimización. A través de la implementación y análisis del algoritmo genético en el juego Snake, se espera proporcionar una comprensión más profunda de su potencial y limitaciones, así como sugerencias para futuras investigaciones y aplicaciones.

3 Objetivos

3.1 Objetivo General

Encontrar el camino óptimo para que la serpiente recoja todas las frutas, maximizando la puntuación.

3.2 Objetivos Específicos

1. Implementar las reglas básicas del juego Snake en un entorno controlado.

    - **Descripción:** Se creó un tablero de juego y se codificaron las reglas básicas del juego Snake, incluyendo el movimiento de la serpiente y la generación de frutas en posiciones fijas.

2. Desarrollar y codificar un algoritmo genético para optimizar los movimientos de la serpiente.

    - **Descripción:** Se diseñó e implementó un algoritmo genético utilizando selección por torneo, cruce de dos puntos y mutación puntual. Los parámetros del algoritmo fueron ajustados a través de pruebas iterativas para maximizar su eficiencia.

3. Visualizar y analizar el comportamiento del algoritmo a través de gráficos y animaciones.

    - **Descripción:** Se desarrollaron scripts para generar gráficos que muestren el rendimiento del algoritmo a lo largo del tiempo, así como animaciones que visualizan el comportamiento de la serpiente durante el proceso de optimización.

4. Evaluar el desempeño del algoritmo en tableros de diferentes tamaños.

    - **Descripción:** Se realizaron experimentos con tableros de diversos tamaños para evaluar cómo afecta el tamaño del problema al rendimiento del algoritmo genético. Los resultados fueron analizados y comparados.

4 Metodología

4.1 Diseño del Tablero y Reglas del Juego

El tablero es una matriz de tamaño arbitrario (en este caso se utilizó una de 20x20 y otra de 40x40) inicializada con ceros. La serpiente se representa como una cola doblemente de doble extremo (*deque*) que contiene las coordenadas (x, y) de cada segmento de la serpiente. La serpiente comienza en el centro del tablero.

Las frutas se representan como una lista de coordenadas (x, y) y se colocan en el tablero en las posiciones especificadas, marcadas con un 1 en la matriz del tablero.

La lógica de movimiento de la serpiente se basa en las direcciones codificadas como números del 1 al 4, que representan arriba, abajo, izquierda y derecha, respectivamente. En cada movimiento, la serpiente agrega un nuevo segmento en la dirección especificada. Si la serpiente se mueve a una celda con una fruta, la fruta se come y se añade un segmento a la serpiente. Si la serpiente se mueve a una celda vacía, se añade un segmento en la nueva dirección y se elimina el último segmento, dando la impresión de que la serpiente se ha movido.

La función de aptitud evalúa un individuo (una secuencia de movimientos) basándose en cuántas frutas come la serpiente y cuántos movimientos realiza. Si la serpiente se come una fruta, recibe 5 puntos, además, recibe un bonus que va siendo exponencialmente mayor entre más rápido se coma la fruta. Hay una penalización de 100 puntos si la serpiente choca con las paredes o con su propio cuerpo.

4.2 Implementación del Algoritmo Genético

El algoritmo genético se utiliza para evolucionar una población de individuos, donde cada individuo es una secuencia de movimientos de la serpiente. Los parámetros del algoritmo genético incluyen el tamaño de la población, la longitud de la serpiente, la tasa de mutación y el número de generaciones.

La población inicial se crea con una función que genera una lista de individuos, cada uno de los cuales es una secuencia aleatoria de movimientos.

La selección de los padres para el cruce se realiza mediante la selección de torneo, donde se seleccionan 3 individuos al azar y se elige el mejor según su puntuación de aptitud.

El cruce se realiza con un cruce de dos puntos, donde se seleccionan dos puntos al azar en la secuencia de movimientos y se intercambian las secciones entre estos puntos para crear dos nuevos individuos.

La mutación se realiza recorriendo cada movimiento en la secuencia de un individuo y, con una probabilidad igual a la tasa de mutación, cambiando el movimiento por un movimiento aleatorio.

El algoritmo genético se ejecuta durante un número especificado de generaciones, en cada una de las cuales se evalúa la población, se seleccionan los padres, se realiza el cruce y la mutación, y se crea una nueva población. Se guarda la mejor puntuación y la puntuación media de cada generación.

Finalmente, se evalúa el mejor individuo de todas las generaciones y se imprime su secuencia de movimientos, el número de frutas que comió y su puntuación. Los datos de todas las generaciones y del mejor individuo se guardan en un archivo para su posterior análisis.

4.3 Visualización del Mejor Individuo y Jugabilidad Por Parte del Usuario

El tablero se imprime utilizando una función que toma el tablero y la serpiente como argumentos. Primero, crea una copia del tablero y luego marca las posiciones de la serpiente en este tablero temporal con el número 2. Luego, recorre cada celda del tablero e imprime un punto para las celdas vacías, un cuadrado rojo para las frutas y un cuadrado verde para la serpiente.

En la función del modo automático (visualización), se le da la serie de movimientos del mejor individuo, e itera sobre esta. Para cada movimiento:

1. Limpia la consola.
2. Imprime el estado actual del juego.
3. Aplica el movimiento actual a la serpiente y actualiza el tablero, la serpiente y las frutas. Si el movimiento resulta en el fin del juego (por ejemplo, la serpiente se choca consigo misma o con el borde del tablero), imprime la puntuación final y termina el juego.
4. Imprime la puntuación actual.
5. Hace una pausa de 0.1 segundos antes de pasar al siguiente movimiento.

Para que el usuario pueda jugar, se creó una función que permite jugar el juego de forma manual. Aquí están los pasos que sigue:

1. Inicializa el juego: se crea el tablero, la serpiente y las frutas.
2. Entra en un bucle infinito donde el estado del juego se imprime en cada iteración.
3. Solicita al usuario que introduzca un movimiento. Los movimientos posibles son 'w' para arriba, 's' para abajo, 'a' para la izquierda y 'd' para la derecha.

4. Si el movimiento introducido no es válido, imprime un mensaje de error y vuelve al principio del bucle.

5. Si el movimiento es válido, mueve la serpiente en la dirección especificada, actualiza el tablero y la lista de frutas, y devuelve False si el movimiento resulta en el fin del juego (por ejemplo, si la serpiente se choca consigo misma o con el borde del tablero).

6. Si el juego ha terminado, imprime la puntuación final y rompe el bucle.

7. Si el juego no ha terminado, imprime la puntuación actual y vuelve al principio del bucle.

4.4 Animación para visualizar una generación en específico

Se creó un script para poder ver el comportamiento de todas las serpientes de una generación al mismo tiempo utilizando la librería *matplotlib*. Primero, carga los datos de un archivo que contiene información sobre todas las generaciones de serpientes. Luego, inicializa un tablero y un diccionario para rastrear las frutas que cada serpiente ha comido.

Se crea una función que actualiza el tablero con las nuevas posiciones de las serpientes para cada movimiento en una generación dada. Esta función maneja la lógica del juego de la serpiente, incluyendo el movimiento de la serpiente, la detección de colisiones y el consumo de frutas.

Se crea una función que actualiza la visualización en cada cuadro de la animación. Esta función llama a la función anterior para obtener el estado actualizado del tablero, y luego dibuja el tablero en el eje de la figura.

Finalmente, el script crea una animación que muestra cómo evoluciona el tablero a lo largo de los movimientos de la serpiente en la generación especificada y muestra la animación.

4.5 Obtención de gráficas

Se utiliza la biblioteca *matplotlib* para visualizar dos gráficas: la evolución de los puntajes a lo largo de las generaciones en un juego de serpiente, y un histograma de las puntuaciones de la última generación.

5 Marco Teórico

5.1 Algoritmos Genéticos

Los algoritmos genéticos (AG) son una técnica de optimización basada en los principios de la selección natural y la genética, propuestos inicialmente por John Holland en los años 70 (Holland, 1975). Los AG son particularmente útiles para problemas de optimización donde el espacio de soluciones es grande y complejo. A través de mecanismos que emulan la evolución biológica, los AG pueden encontrar soluciones aproximadas a problemas difíciles en un tiempo razonable.

5.2 Componentes de los Algoritmos Genéticos

1. Codificación (Representación): Los individuos en un AG representan posibles soluciones al problema. En este caso, un individuo es una secuencia de movimientos (arriba, abajo, izquierda, derecha) que la serpiente debe seguir para recoger las frutas.

2. Población Inicial: La población inicial es un conjunto de individuos generados aleatoriamente. La diversidad en la población inicial es crucial para explorar efectivamente el espacio de soluciones.

3. Función de Aptitud (Fitness Function): La función de aptitud evalúa la calidad de cada individuo, asignando una puntuación basada en qué tan bien resuelve el problema. En este proyecto, la función de mide cuántas frutas recoge la serpiente y la eficiencia de su camino.

4. Selección: La selección se utiliza para elegir los individuos que se reproducirán para formar la siguiente generación. En este proyecto, se utiliza la selección por torneo, donde un subconjunto de individuos compite, y el mejor es seleccionado para reproducirse.

5. Cruzamiento (Crossover): El cruzamiento combina partes de dos individuos padres para crear uno o más hijos. En este proyecto, se utilizó el cruce de dos puntos, donde dos puntos de corte se eligen al azar y los segmentos entre estos puntos se intercambian entre los padres para generar nuevos individuos.

6. Mutación: La mutación introduce variabilidad adicional en la población al alterar aleatoriamente partes de un individuo. En este caso, se utiliza la mutación puntual, donde un movimiento en la secuencia puede cambiar aleatoriamente.

5.3 Ejemplos de Uso de Algoritmos Genéticos

Los algoritmos genéticos han sido aplicados exitosamente en una variedad de problemas, desde la optimización de rutas en la logística hasta el diseño de redes neuronales.

1. Logística y Planificación de Rutas: Los AG se utilizan para resolver problemas de rutas de vehículos, donde se debe encontrar la ruta óptima para que un vehículo entregue productos a varios destinos. Los AG pueden minimizar la distancia total recorrida y mejorar la eficiencia del transporte.

2. Optimización de Diseño en Ingeniería: En ingeniería, los AG se aplican para optimizar el diseño de estructuras y componentes mecánicos, encontrando configuraciones que minimizan el peso y maximizan la resistencia.

3. Evolución de Redes Neuronales: Los AG se emplean para evolucionar las arquitecturas de redes neuronales, buscando configuraciones que mejoren el rendimiento en tareas de aprendizaje automático.

5.4 Relevancia para el Proyecto

En este proyecto, los algoritmos genéticos se aplican para optimizar el camino que sigue una serpiente en el juego Snake. Este problema es una representación simplificada de problemas reales en logística, donde la planificación de rutas eficientes es crucial. La capacidad de los AG para encontrar soluciones aproximadas en espacios de búsqueda complejos los hace ideales para este tipo de problemas.

6 Implementación del Algoritmo Genético

En esta sección, se describe el desarrollo e implementación del algoritmo genético utilizado para optimizar los movimientos de la serpiente en el juego Snake. Se presentan fragmentos de código con explicaciones detalladas para cada una de las funciones principales.

6.1 Inicialización del Juego

Primero, se define el tamaño del tablero, la posición inicial de la serpiente y las posiciones de las frutas.

```
import numpy as np
from collections import deque
import random
import math
import pickle
import matplotlib.pyplot as plt
import json
import os

BOARD_SIZE = 40

board = np.zeros((BOARD_SIZE, BOARD_SIZE), dtype=int)

snake = deque([(BOARD_SIZE // 2, BOARD_SIZE // 2)])

fruits = [(5, 6), (7, 16), (13, 8), (14, 13), (3, 17), (20, 30),
          (35, 5), (10, 35), (30, 20), (21, 5)]

for fruit in fruits:
    board[fruit] = 1  # 1 representa una fruta
```

6.2 Visualización del Tablero

Se incluye una función para imprimir el tablero, donde la serpiente se representa con el valor 2 y las frutas con el valor 1.

```
# Función para imprimir el tablero
def print_board(board, snake):
    temp_board = np.copy(board)
    for x, y in snake:
```

```
        temp_board[x, y] = 2  # 2 representa la snake
    print(temp_board)

# Mostramos el tablero inicial
print_board(board, snake)
```

6.3 Parámetros del Algoritmo Genético

Se definen los parámetros clave del algoritmo genético, como el tamaño de la población, la longitud máxima de la serpiente, la tasa de mutación y el número de generaciones.

```
# Parámetros del algoritmo genético
POPULATION_SIZE = 2000
SNAKE_LENGTH = 500
MUTATION_RATE = 0.005
NUM_GENERATIONS = 20000

# Direcciones posibles (1: Arriba, 2: Abajo, 3: Izquierda, 4: Derecha)
DIRECTIONS = [1, 2, 3, 4]
```

6.4 Creación de Individuos y Población

Se crean individuos (secuencias de movimientos) y una población inicial de individuos.

```
# Función para crear un individuo
def create_individual():
    return [random.choice(DIRECTIONS) for _ in range(SNAKE_LENGTH)]

# Función para crear la población inicial
def create_population(size):
    return [create_individual() for _ in range(size)]
```

6.5 Evaluación de Individuos

La función `evaluate_individual` evalúa el desempeño de cada individuo en el tablero, calculando un puntaje basado en la cantidad de frutas recogidas y la eficiencia del camino.

```
# Función de evaluación
def evaluate_individual(individual, board, snake, fruits):

    snake_copy = deque(snake)
    score = 0
    moves = 0
    fruits_eaten = 0
    crash_penalty = 0

    # Mapeo de direcciones a movimientos en el tablero
    direction_map = {
        1: (-1, 0),  # Arriba
        2: (1, 0),   # Abajo
```

```
        3: (0, -1),  # Izquierda
        4: (0, 1)    # Derecha
    }

    fruits_copy = fruits[:]

    fruit_eating_moves = []

    for index, move in enumerate(individual):
        dx, dy = direction_map[move]
        head_x, head_y = snake_copy[0]
        new_head = (head_x + dx, head_y + dy)

        # Verificamos colisiones con las paredes o el cuerpo de la snake
        if (new_head[0] < 0 or new_head[0] >= BOARD_SIZE or
            new_head[1] < 0 or new_head[1] >= BOARD_SIZE or
            new_head in snake_copy):
            crash_penalty = -100
            break

        # Movemos la snake
        snake_copy.appendleft(new_head)
        if new_head in fruits_copy:
            score += 5
            fruits_copy.remove(new_head)
            fruits_eaten += 1
            fruit_eating_moves.append((index, move, new_head))
            score += math.exp(math.exp((len(individual) - index) / len(individual)))
            if len(fruits_copy) == 0:  # Ya no hay frutas
                break
        else:
            snake_copy.pop()

        moves += 1

    # Penalización por movimientos innecesarios
    penalty = moves / SNAKE_LENGTH

    return score + crash_penalty, fruits_eaten, fruit_eating_moves
```

6.6 Selección, Cruce y Mutación

Se implementan funciones para la selección de padres usando torneo, cruce de dos puntos y mutación puntual.

```
# Función para seleccionar a los padres usando torneo
def tournament_selection(population, scores, k=3):
    selected = random.sample(list(zip(population, scores)), k)
    selected = sorted(selected, key=lambda x: x[1], reverse=True)
```

```
    return selected[0][0]

# Función para cruzar dos individuos
def crossover(parent1, parent2):
    point1 = random.randint(1, SNAKE_LENGTH - 2)
    point2 = random.randint(point1, SNAKE_LENGTH - 1)
    child1 = parent1[:point1] + parent2[point1:point2] + parent1[point2:]
    child2 = parent2[:point1] + parent1[point1:point2] + parent2[point2:]
    return child1, child2

# Función para mutar un individuo
def mutate(individual, mutation_rate):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = random.choice(DIRECTIONS)
    return individual
```

6.7 Evolución de la Población

El algoritmo genético evoluciona la población a través de múltiples generaciones, evaluando, seleccionando, cruzando y mutando individuos.

```
# Crear la población inicial
population = create_population(POPULATION_SIZE)

all_generations = []

best_fruits_eaten = 0
best_score = float('-inf')
best_individual = None
best_scores = []
avg_scores = []

# Evolucionar la población
for generation in range(NUM_GENERATIONS):
    # Evaluar la población
    results = [evaluate_individual(ind, np.copy(board), deque(snake), fruits[:]) for
    scores, fruits_eaten, fruit_eating_moves = zip(*results)

    best_scores.append(max(scores))
    avg_scores.append(sum(scores) / len(scores))

    # Checar si el mejor individuo de la generación actual es mejor que el mejor indi
    max_index = scores.index(max(scores))
    if fruits_eaten[max_index] > best_fruits_eaten:
        best_fruits_eaten = fruits_eaten[max_index]

    all_generations.append(population[:])
```

```
    # Seleccionar los mejores individuos
    next_population = []
    for _ in range(POPULATION_SIZE // 2):
        parent1 = tournament_selection(population, scores)
        parent2 = tournament_selection(population, scores)
        child1, child2 = crossover(parent1, parent2)
        next_population.append(mutate(child1, MUTATION_RATE))
        next_population.append(mutate(child2, MUTATION_RATE))

    # Imprimir la mejor puntuación de cada generación
    current_best_score = max(scores)
    print(f"Generación {generation}: Mejor puntuación = {current_best_score}")

    if current_best_score > best_score:
        best_score = current_best_score
        best_individual = population[scores.index(current_best_score)]

    population = next_population
```

6.8 Resultados Finales

Se evalúa el mejor individuo encontrado y se guardan los datos relevantes en un archivo para análisis posterior.

```
# Evaluar el mejor individuo
results = evaluate_individual(best_individual, np.copy(board), deque(snake), fruits[:
best_score, best_fruits_eaten, best_fruit_eating_moves = results
print(f"Mejor secuencia de movimientos: {best_individual}")
print(f"Número de frutas comidas por el mejor individuo: {best_fruits_eaten}")
print(f"Puntuación del mejor individuo: {best_score}")
print(f"Movimientos que resultaron en comer una fruta (index, movimiento, (posición d

# Guardar los datos
directory = r'D:\UAEM\4to semestre\Busqueda bayesiana\Codes'
os.makedirs(directory, exist_ok=True)
file_path = os.path.join(directory, 'data.pkl')

data = {
    'all_generations': all_generations,
    'best_individual': list(best_individual),
    'best_scores': best_scores,
    'avg_scores': avg_scores,
    'best_score': best_score,
    'best_fruits_eaten': best_fruits_eaten,
    'results': (results[0], results[1], [(index, move, list(pos)) for index, move, po
    'scores': list(scores)
}

with open(file_path, 'wb') as f:
```

```
pickle.dump(data, f)
```

7 Resultados y Discusión

En esta sección se presentan y analizan los resultados obtenidos a través del algoritmo genético implementado para optimizar los movimientos de la serpiente en el juego Snake. Se incluyen diversas gráficas que muestran la evolución del rendimiento del algoritmo a lo largo de las generaciones.

7.1 Matriz de 20x20 con 5 frutas

En un tablero de 20x20 con 5 frutas, el algoritmo genético no requirió de mucho tiempo para encontrar un camino óptimo. El algoritmo pudo localizar todas las frutas de manera relativamente rápida. Los parámetros utilizados en este caso fueron:

- Longitud de individuo: 100
- Probabilidad de mutación: 0.001
- Tamaño de población: 1000
- Número de generaciones: 5000

7.1.1 Gráfica de la Mejor Puntuación y Puntuación Promedio por Generación

La Figura 1 muestra la evolución de la mejor puntuación y la puntuación promedio obtenida por los individuos a lo largo de 5000 generaciones. Se observa que el mayor salto de progreso ocurre durante las primeras 1000 generaciones, con pequeñas mejoras durante unas 2000 generaciones adicionales. Alrededor de la generación 2500 y nuevamente en la generación 3200, se observan aumentos significativos en el puntaje, los cuales pueden interpretarse como momentos en que la serpiente encontró una fruta adicional. Después de estos incrementos, el algoritmo converge y no se observan mejoras significativas durante más de 1000 generaciones.

7.1.2 Histograma de Puntuaciones de la Última Generación

La Figura 2 presenta un histograma de los puntajes de la última generación. En esta gráfica se observa que, aunque existe un pequeño grupo de individuos con puntajes negativos, la mayoría de los individuos tienen un puntaje superior a 60.

7.1.3 Resultados Detallados del Mejor Individuo

El algoritmo genético logró un puntaje máximo de **72.42**, indicando un alto grado de optimización en los movimientos de la serpiente. La serpiente logró consumir todas las frutas disponibles, evidenciando la eficacia del algoritmo. A continuación, se presentan los resultados detallados:

- **Mejor puntuación:** 72.42
- **Número de frutas consumidas:** 5

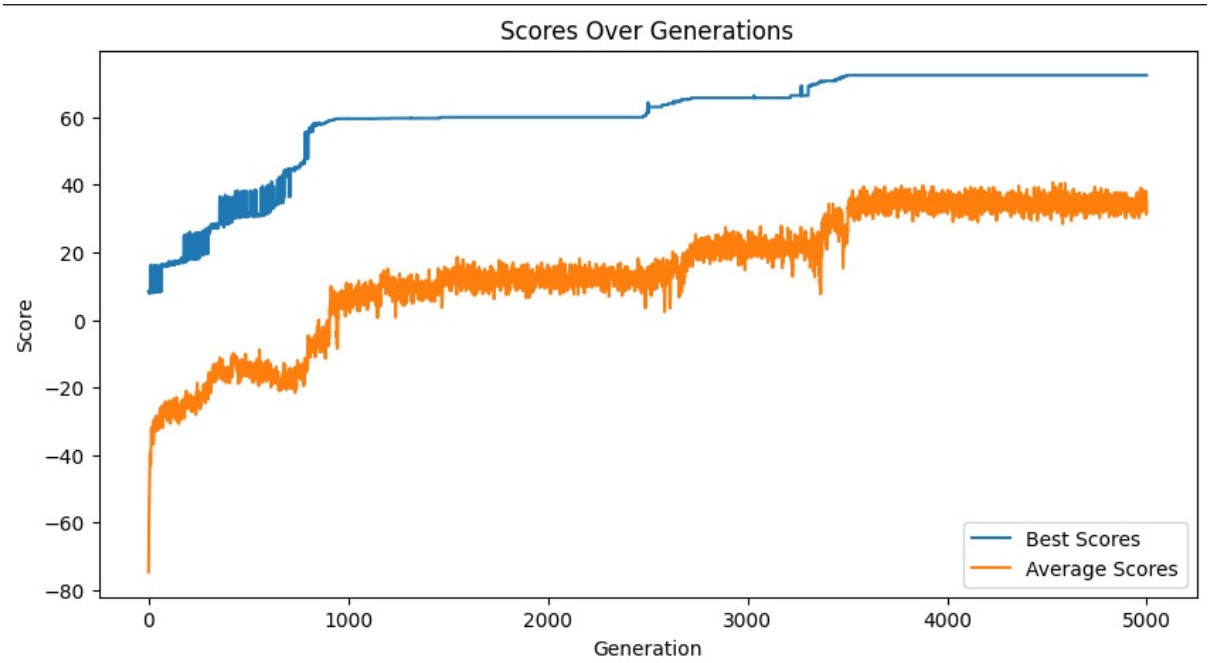

Figure 1: Puntuación promedio y mejor puntuación por generación para la matriz de 20x20 con 5 frutas.

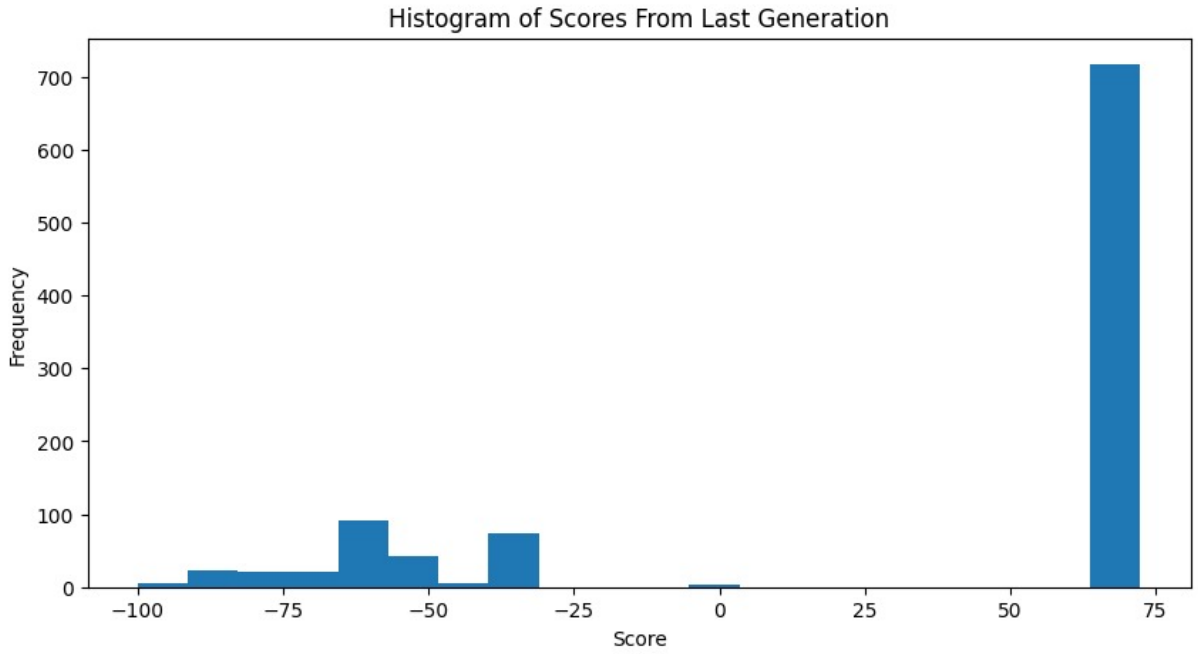

Figure 2: Histograma de puntuaciones de la última generación para la matriz de 20x20 con 5 frutas.

- **Detalles de los movimientos (índice, movimiento, posición de la fruta):**
  - (4, 2, [13, 8])
  - (10, 4, [14, 13])
  - (24, 4, [7, 16])
  - (29, 1, [3, 17])
  - (42, 3, [5, 6])

7.1.4 Interpretación de los Resultados:

Los resultados obtenidos son indicativos de un comportamiento típico de los algoritmos genéticos. La fase inicial del algoritmo muestra una rápida mejora, lo cual es esperado dado que las primeras generaciones exploran el espacio de soluciones de manera amplia. Los incrementos significativos en las generaciones 2500 y 3200 probablemente se deben a la optimización continua y al descubrimiento de frutas adicionales, que se refleja en los saltos en la puntuación.

La convergencia observada después de estos saltos sugiere que el algoritmo ha explorado exhaustivamente las soluciones óptimas disponibles en el espacio de búsqueda actual. Esto puede indicar que el algoritmo ha alcanzado un máximo local, donde la población ha explotado la mayoría de las buenas soluciones disponibles. La falta de mejoras adicionales en las generaciones siguientes también puede ser atribuida a una pérdida de diversidad genética dentro de la población, lo que impide una mayor exploración y optimización.

Para mejorar estos resultados, podrían considerarse técnicas como la reinicialización parcial de la población para introducir nueva diversidad genética, o la aplicación de operadores de mutación más agresivos para evitar el estancamiento en máximos locales.

7.2 Matriz de 40x40 con 10 frutas

En un tablero de 40x40 con 10 frutas, el espacio es significativamente más grande. Para abordar este problema, se incrementó la longitud de la serpiente a 200 movimientos y se realizaron 20000 generaciones. Sin embargo, la serpiente solo logró encontrar 6 frutas. Los resultados indican que se comió la primera fruta en el movimiento 12, la segunda en el movimiento 33, pero la tercera no fue consumida hasta el movimiento 118. Al observar la animación, se pudo notar que la serpiente realizó muchos movimientos innecesarios en este espacio, lo cual sugiere que el algoritmo no tuvo suficientes generaciones para alcanzar el camino óptimo o que se quedó atrapado en un máximo local. Lamentablemente, debido a limitaciones de hardware, no fue posible realizar tantas pruebas como se hubiera deseado. Los parámetros utilizados en este caso fueron los siguientes:

- Longitud de individuo: 200
- Probabilidad de mutación: 0.0005
- Tamaño de población: 1000
- Número de generaciones: 20000

7.2.1 Gráfica de la Mejor Puntuación y Puntuación Promedio por Generación

La Figura 3 muestra la evolución de la mejor puntuación y la puntuación promedio obtenida por los individuos a lo largo de 20000 generaciones. Se observa que el mayor salto de progreso ocurre durante las primeras 1000 generaciones, similar a la matriz anterior. Posteriormente, se observan pequeñas mejoras durante unas 5000 generaciones, con saltos ocasionales en los puntajes de los mejores individuos que rápidamente vuelven a bajar. Alrededor de la generación 6000, se observa otro salto en la puntuación, y este patrón se repite otras dos veces aproximadamente cada 2500 generaciones. Estos saltos pueden interpretarse como momentos en los que la serpiente encuentra una fruta adicional. A partir de la generación 12500, el algoritmo realiza pequeñas mejoras durante unas 2500 generaciones, para luego estabilizarse por unas 5000 generaciones.

7.2.2 Histograma de Puntuaciones de la Última Generación

La Figura 4 presenta un histograma de los puntajes de la última generación. A diferencia de la matriz anterior, se observa un grupo mucho mayor de individuos con puntajes negativos, y el grupo de individuos con puntajes positivos no está tan agrupado, abarcando desde 8 puntos hasta 56 puntos.

7.2.3 Resultados Detallados del Mejor Individuo

El algoritmo genético logró un puntaje máximo de **67.75**. La serpiente logró consumir 6 de las 10 frutas disponibles. A continuación, se presentan los resultados detallados:

- **Mejor puntuación:** 67.75
- **Número de frutas consumidas:** 6
- **Detalles de los movimientos (índice, movimiento, posición de la fruta):**
  - (12, 1, [14, 13])
  - (33, 1, [3, 17])
  - (118, 1, [7, 16])
  - (132, 3, [13, 8])
  - (168, 4, [10, 35])
  - (183, 2, [20, 30])

7.2.4 Interpretación de los Resultados:

Los resultados obtenidos son indicativos de las dificultades inherentes al aumento del tamaño del espacio de búsqueda. La serpiente logró encontrar 6 de las 10 frutas, pero mostró ineficiencias en sus movimientos, como lo indican los largos intervalos entre el consumo de algunas frutas. Los saltos observados en las gráficas de puntuación sugieren momentos de optimización significativos, pero la convergencia a puntajes más bajos indica que el algoritmo podría haberse estancado en máximos locales o que no se le proporcionaron suficientes generaciones para explorar completamente el espacio de búsqueda.

El histograma de puntajes de la última generación muestra una mayor dispersión en los puntajes, con un número significativo de individuos con puntajes negativos, lo que

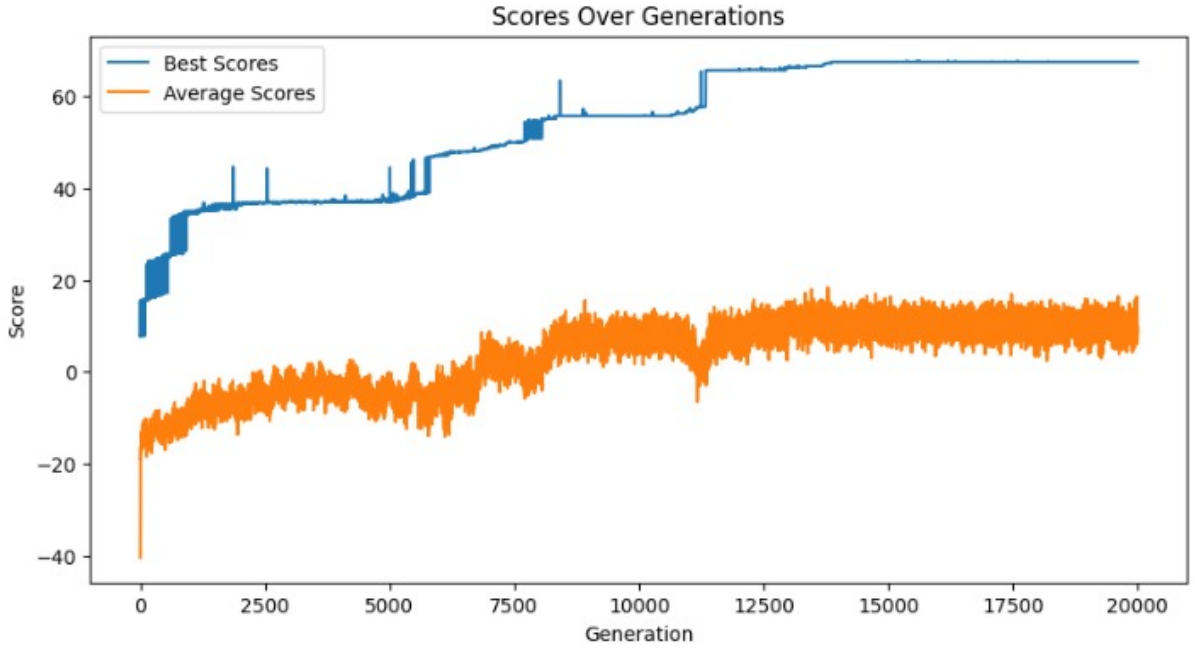

Figure 3: Puntuación promedio y mejor puntuación por generación para la matriz de 40x40 con 10 frutas.

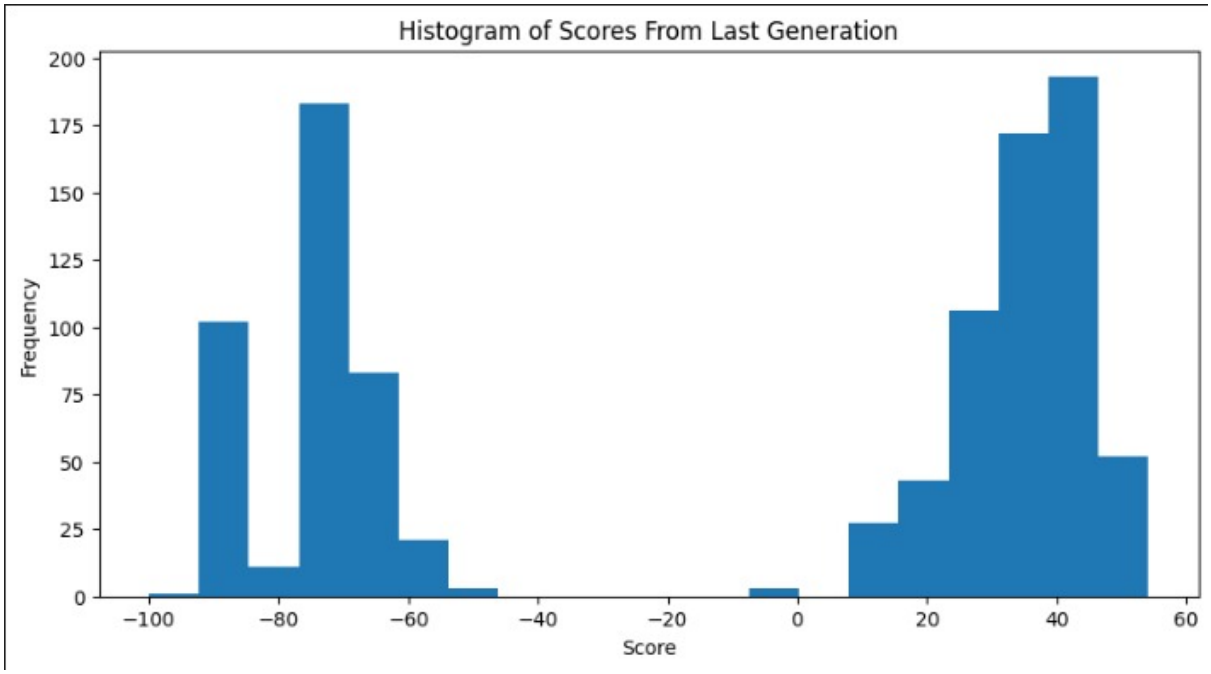

Figure 4: Histograma de puntuaciones de la última generación para la matriz de 40x40 con 10 frutas.

sugiere una variabilidad alta en la calidad de las soluciones dentro de la población. Este comportamiento podría mejorarse aumentando la diversidad genética o ajustando los parámetros del algoritmo genético para promover una exploración más efectiva.

Para mejorar estos resultados, se podrían aplicar las mismas mejoras mencionadas anteriromente: técnicas como la reinicialización parcial de la población para introducir nueva diversidad genética, o la aplicación de operadores de mutación más agresivos para evitar el estancamiento en máximos locales.

8 Conclusiones

En este proyecto se desarrolló e implementó un algoritmo genético para optimizar los movimientos de la serpiente en el juego Snake. A través de diversas pruebas y análisis, se lograron obtener resultados significativos que permiten evaluar la efectividad del algoritmo en distintos escenarios. Las conclusiones principales del proyecto se resumen a continuación:

8.1 Eficiencia del Algoritmo Genético

El algoritmo genético demostró ser eficaz en la optimización de los movimientos de la serpiente, especialmente en tableros de tamaño reducido. En el caso del tablero de 20x20 con 5 frutas, el algoritmo no requirió muchas generaciones para encontrar un camino óptimo, mostrando una rápida convergencia hacia soluciones de alta calidad. La mayor parte de las mejoras significativas ocurrieron durante las primeras 1000 generaciones, lo que indica una fase inicial de exploración efectiva seguida de una explotación de las mejores soluciones encontradas.

8.2 Desempeño en Problemas de Mayor Escala

Cuando se incrementó el tamaño del tablero a 40x40 con 10 frutas, el algoritmo enfrentó mayores desafíos. Aunque la serpiente logró encontrar 6 de las 10 frutas disponibles, se observó una mayor cantidad de movimientos innecesarios y una dispersión más amplia en las puntuaciones de la población. Esto sugiere que el algoritmo podría no haber tenido suficientes generaciones para explorar completamente el espacio de soluciones más grande o que se quedó atrapado en máximos locales. Es evidente que se requieren mas pruebas donde se juegue con los parametros para enocontrar mejores soluciones.

8.3 Visualización y Análisis de Resultados

Las gráficas de puntuación promedio y mejor puntuación por generación proporcionaron una visión clara de la evolución del algoritmo a lo largo del tiempo. En ambos tamaños de tablero, se identificaron saltos en las puntuaciones que se correlacionaron con momentos en que la serpiente encontró frutas adicionales. Los histogramas de las puntuaciones de la última generación mostraron diferencias en la dispersión de las puntuaciones, indicando variabilidad en la calidad de las soluciones dependiendo del tamaño del tablero y la complejidad del problema.

8.4 Limitaciones y Recomendaciones

Las limitaciones del hardware impidieron realizar tantas pruebas como se hubieran deseado, lo cual podría haber afectado la capacidad del algoritmo para alcanzar el camino óptimo en tableros más grandes. Para futuros trabajos, se recomienda:

- Incrementar el número de generaciones y el tamaño de la población para permitir una mayor exploración del espacio de soluciones.
- Implementar técnicas de reinicialización parcial de la población para introducir nueva diversidad genética y evitar el estancamiento en máximos locales.
- Experimentar con operadores de mutación más agresivos para promover una mayor variabilidad en las soluciones.
- Evaluar el desempeño del algoritmo en otros entornos de simulación y con diferentes configuraciones de parámetros para generalizar los resultados obtenidos.

8.5 Aplicaciones y Futuras Investigaciones

Los resultados obtenidos en este proyecto no solo son aplicables al juego Snake, sino que también tienen implicaciones en problemas de optimización del mundo real, como la planificación de rutas en logística y la navegación de robots en entornos complejos. La capacidad de los algoritmos genéticos para encontrar soluciones aproximadas en espacios de búsqueda grandes y complejos los hace una herramienta valiosa para diversas aplicaciones.

En futuras investigaciones, se podrían explorar combinaciones de algoritmos genéticos con otras técnicas de optimización, como algoritmos de enjambre de partículas o algoritmos de recocido simulado, para mejorar aún más la eficiencia y efectividad en la resolución de problemas complejos.

Conclusión Final:

El algoritmo genético desarrollado mostró un desempeño sólido en la optimización de los movimientos de la serpiente en el juego Snake, especialmente en escenarios menos complejos. Sin embargo, su efectividad disminuyó en problemas de mayor escala, destacando la necesidad de ajustar parámetros y mejorar la diversidad genética. Este proyecto subraya el potencial de los algoritmos genéticos en la resolución de problemas de optimización y su aplicabilidad a problemas reales, ofreciendo una base sólida para futuras investigaciones.

9 Notas del Autor

No incluí el codigo para mostrar al mejor individuo ni para que el usuario jugara, asi como tapoco incluí el código para ver las animaciónes. Esto se debe a que el enfoque principal del proyecto era la utilización de algorítmos genéticos, y agregar esto haria que me exendiera mucho. Sin embargo, son cosas que considero pueden ser utiles para tener una mejor idea de lo que esta pasando, por lo que cree un repositorio en github con los tres scripts para que quien guste pueda correrlos.

Cabe mencionar que tanto el script que implementa el algoritmo como el que genera las animaciones de una generación requieren muchos recursos, por lo que no es de preocuparse si el programa tarda mucho tiempo en correr (A mi me tomo unas 3-4 horas cada vez que

queria correr el codigo del AG de 40x40 con 20000 generaciones, si es que no crasheaba, y ni se diga de la animación).

Aquí está el repositorio `https://github.com/KixieLove/snake-inteligente`

References

[1] Genetic Algorithm. (n.d.). In *ScienceDirect*. Retrieved from `https://www.sciencedirect.com/topics/medicine-and-dentistry/genetic-algorithm`

[2] What are Genetic Algorithms? (n.d.). In *Spiceworks*. Retrieved from `https://www.spiceworks.com/tech/artificial-intelligence/articles/what-are-genetic-algorithms/`