Implementación de un Algoritmo Genético para la Evolución de una Cadena Binaria

Victor Gerardo Rodríguez Barragán

29 de Octubre de 2023

1 Descripcion del ejercicio y solucion

1.1 Ejericio

Implementar el algoritmo genético para evolucionar una cadena binaria hacia una cadena objetivo. La cadena objetivo, en este caso, es "1101101101". El algoritmo genético busca generar una población de cadenas que se asemejen lo más posible a la cadena objetivo a lo largo de varias generaciones.

1.2 Solucion

La solucion se implemento en Rust.

1. **Poblacion inicial:** Se genera una poblacion inicial de 10 cadenas binarias de 10 bits (en este caso).
2. **Evaluacion:** Se evalua cada cadena de la poblacion inicial para determinar su aptitud.
3. **Seleccion:** Se seleccionan las cadenas con mayor aptitud (10% de la poblacion inicial) como padres.
4. **Cruza:** Se cruzan las cadenas seleccionadas para generar descendientes.
5. **Mutacion:** Se mutan los descendientes con una probabilidad muy baja (0.00001) para cambiar un bit aleatorio.
6. **Iteracion:** Se repiten los pasos 2 a 5 hasta que se encuentre una cadena que sea igual a la cadena objetivo.

2 Codigo

```rust
use rand::Rng;
use std::collections::BinaryHeap;

const TARGET_STRING: &str = "1101101101";
const POPULATION_SIZE: usize = 10;
const MUTATION_RATE: f64 = 0.00001;

fn main()
{
    let rng = rand::thread_rng();
    let mut population = generate_initial_population();

    let mut generation = 0;
    let mut best_fitness = 0;
    let mut best_individual = String::new();
```

```
    loop
    {
        generation += 1;

        // Calculate the fitness of each individual
        let mut fitness_values: Vec<(usize, String)> = Vec::with_capacity(POPULATION_SIZE);
        for candidate in &population
        {
            let fitness = candidate
                .chars()
                .zip(TARGET_STRING.chars())
                .filter(|&(a, b)| a == b)
                .count();
            fitness_values.push((fitness, candidate.clone()));
        }

        // Sort by fitness
        fitness_values.sort_by(|a, b| b.0.cmp(&a.0));

        let (best_fitness_current, best_individual_current) = fitness_values[0].clone();

        println!("Generation {}: {}", generation, best_individual_current);

        if best_fitness_current >= best_fitness
        {
            best_fitness = best_fitness_current;
            best_individual = best_individual_current.clone();
        }

        if best_fitness >= TARGET_STRING.len()
        {
            println!("Target reached!");
            break;
        }

        let parents = select_parents(&fitness_values);

        let new_population = create_new_population(&parents);

        population = new_population;
    }
}

fn generate_initial_population() -> Vec<String>
{
    (0..POPULATION_SIZE)
```

```
        .map(|_| generate_random_string())
        .collect()
}

fn generate_random_string() -> String
{
    (0..TARGET_STRING.len())
        .map(|_| {
            if rand::random::<f64>() < 0.5 {
                '0'
            } else {
                '1'
            }
        })
        .collect()
}

fn select_parents(fitness_values: &Vec<(usize, String)>) -> Vec<String>
{
    let mut parents = Vec::new();

    for i in 0..fitness_values.len()
    {
        if i < POPULATION_SIZE / 10
        {
            parents.push(fitness_values[i].1.clone());
        }
        else
        {
            break;
        }
    }

    parents
}

fn create_new_population(parents: &Vec<String>) -> Vec<String>
{
    let mut new_population = Vec::with_capacity(POPULATION_SIZE);
    let mut rng = rand::thread_rng();

    new_population.push(parents[0].clone()); // Keep the best solution

    for _ in 1..POPULATION_SIZE
    {
        let parent1 = &parents[rng.gen_range(0..parents.len())];
```

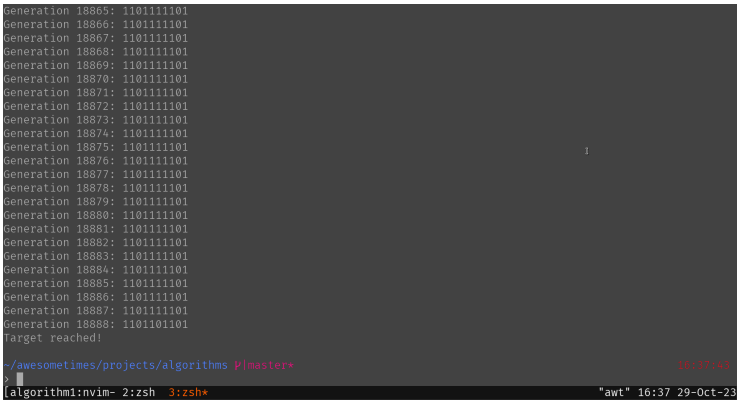
```
        let parent2 = &parents[rng.gen_range(0..parents.len())];
        let child = crossover(parent1, parent2);
        let child = mutate(&child);
        new_population.push(child);
    }

    new_population
}

fn crossover(parent1: &str, parent2: &str) -> String
{
    let mut rng = rand::thread_rng();
    let crossover_point = rng.gen_range(0..parent1.len());
    let mut child = parent1.chars().take(crossover_point).collect::<String>();
    child.push_str(&parent2[crossover_point..]);
    child
}

fn mutate(child: &str) -> String
{
    child
        .chars()
        .map(|c| {
            if rand::random::<f64>() < MUTATION_RATE
            {
                if c == '0' {
                    '1'
                } else {
                    '0'
                }
            }
            else
            {
                c
            }
        })
        .collect()
}
```

3 Evidencia



4 Numero de iteraciones

El algoritmo genético se ejecutó durante varias generaciones hasta alcanzar la cadena objetivo. El número exacto de iteraciones depende de factores como la aleatoriedad de la mutación y la selección de padres, pero el algoritmo se detiene cuando la cadena objetivo se iguala o cuando se alcanza un límite de generaciones.

5 Conclusion

5.1 Aprendizajes

Se adquirió una comprensión más profunda de cómo funciona un algoritmo genético y su aplicación para evolucionar soluciones. Además, se exploraron conceptos clave como selección, cruzamiento y mutación.

5.2 Implementacion del algoritmo

Lo veo util para futuras implementaciones en cuestion a rendimiento y velocidad de convergencia.

5.3 Retos y dificultades

Lo que mas se complico en la practica es tener la taza de convergencia adecuada para que no se quede en un minimo local y no se pueda salir de ahi.