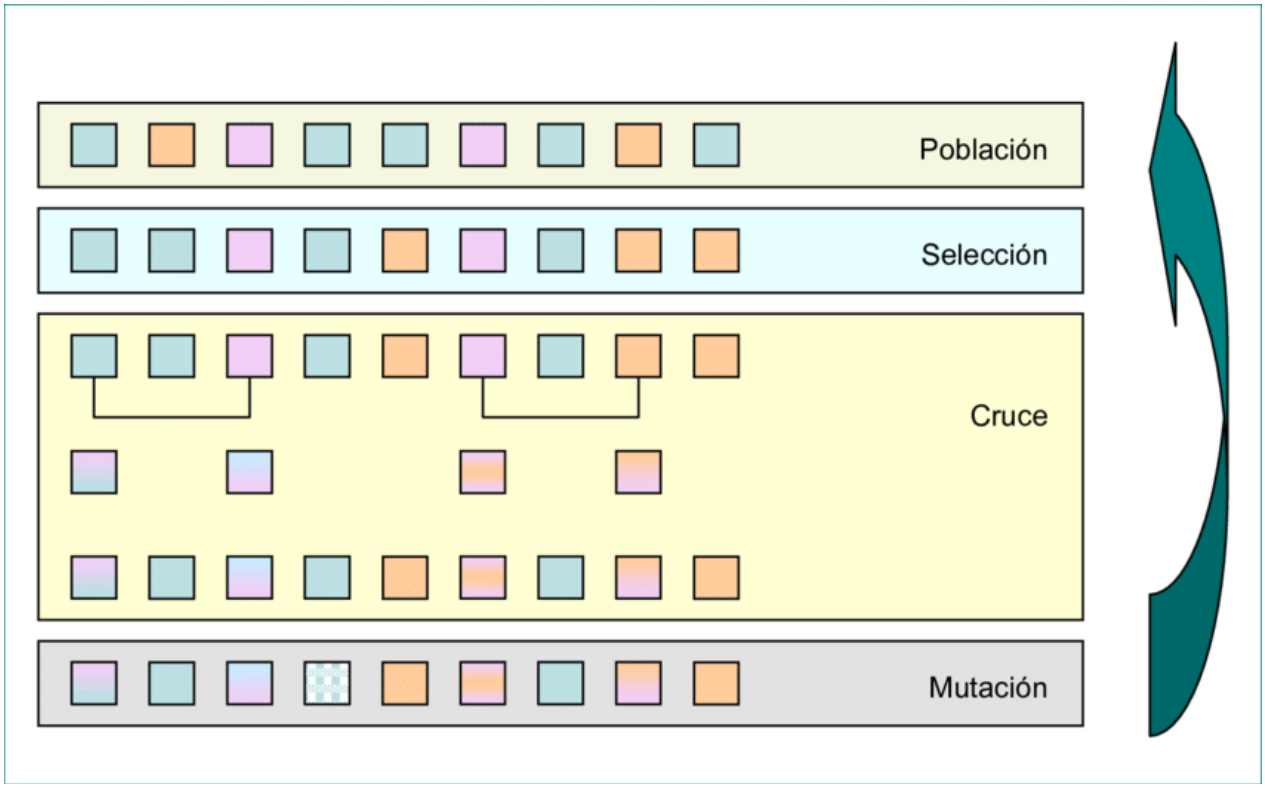
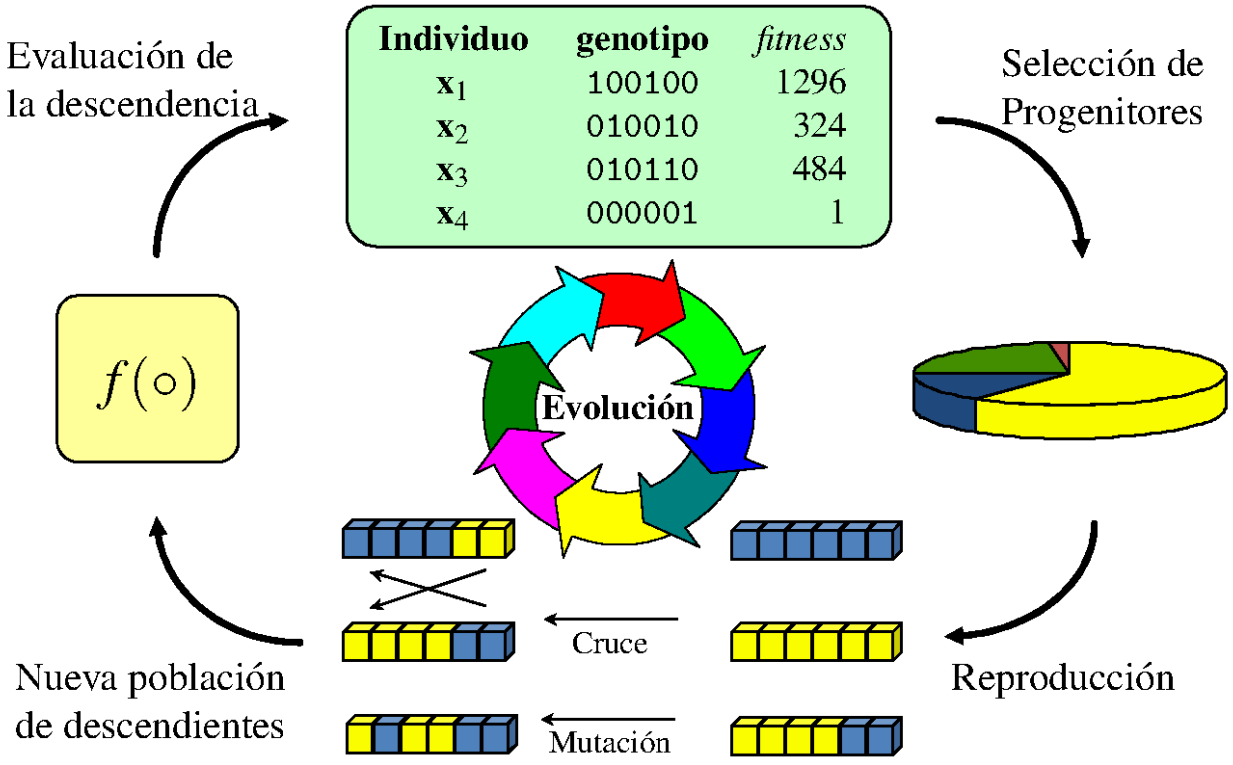
PRACTICA 2: APRENDIZAJE REFORZADO





Arturo Vilar Carretero,

Dario Muñoz Lanoza

ÍNDICE

DESCRIPCIÓN GENÉTICO

Los algoritmos genéticos (AG) funcionan entre el conjunto de soluciones de un problema llamado genotipo, y el conjunto de individuos de una población natural, codificando la información de cada solución en una cadena, generalmente binaria, llamada cromosoma.

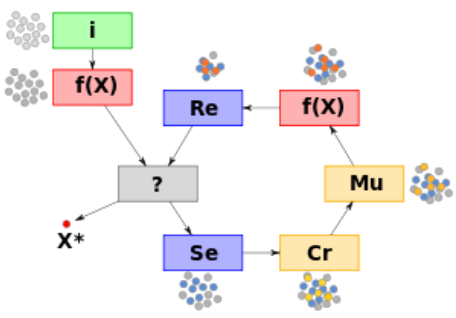
Los símbolos que forman la cadena son llamados genes. Cuando la representación de los cromosomas se hace con cadenas de dígitos binarios se le conoce como genotipo.

Los cromosomas evolucionan a través de iteraciones, llamadas generaciones. En cada generación, los cromosomas son evaluados usando alguna medida de aptitud. Las siguientes generaciones (nuevos cromosomas), son generadas aplicando los operadores genéticos repetidamente, siendo estos los operadores de selección, cruzamiento, mutación y reemplazo.

- Inicialización: Se genera aleatoriamente la población inicial, que está constituida por un conjunto de cromosomas los cuales representan las posibles soluciones del problema. En caso de no hacerlo aleatoriamente, es importante garantizar que dentro de la población inicial, se tenga la diversidad estructural de estas soluciones para tener una representación de la mayor parte de la población posible o al menos evitar la convergencia primigenia.

- Evaluación: A cada uno de los cromosomas de esta población se aplicará la función de aptitud para saber cómo de "buena" es la solución que se está codificando.

- Condición de término: El AG se deberá detener cuando se alcance la solución óptima, pero esta generalmente se desconoce, por lo que se deben utilizar otros criterios de detención. Normalmente se usan dos criterios: correr el AG un número máximo de iteraciones (generaciones) o detenerlo cuando no haya cambios en la población. Mientras no se cumpla la condición de término se hace lo siguiente:
  - Selección: Después de saber la aptitud de cada cromosoma se procede a elegir los cromosomas que serán cruzados en la siguiente generación. Los cromosomas con mejor aptitud tienen mayor probabilidad de ser seleccionados.
  - Recombinación o cruzamiento: La recombinación es el principal operador genético, representa la reproducción sexual, opera sobre dos cromosomas a la vez para generar dos descendientes donde se combinan las características de ambos cromosomas padres.
  - Mutación: Modifica al azar parte del cromosoma de los individuos, y permite alcanzar zonas del espacio de búsqueda que no estaban cubiertas por los individuos de la población actual.
  - Reemplazo: Una vez aplicados los operadores genéticos, se seleccionan los mejores individuos para conformar la población de la generación siguiente.



IMPLEMENTACIÓN

```
public class GeneticAlgorithm
{

    public List<Individual> population;
    private int _currentIndex;

    public int CurrentGeneration;
    public int MaxGenerations;

    public MutationOperator mutationOperator;
    public CrossoverOperator crossoverOperator;

    public string Summary;
    1 referencia | dariusml, Hace 16 horas | 1 autor, 1 cambio
    public GeneticAlgorithm(int numberOfGenerations, int populationSize, MutationOperator mutationOperator, CrossoverOperator crossoverOperator)
    {
        CurrentGeneration = 0;
        MaxGenerations = numberOfGenerations;

        this.mutationOperator = mutationOperator;
        this.crossoverOperator = crossoverOperator;

        GenerateRandomPopulation(populationSize);
        Summary = "";
    }
```

```
public void GenerateRandomPopulation(int size)
{
    population = new List<Individual>();
    for (int i = 0; i < size; i++)
    {
        population.Add(new Individual(Random.Range(0f, 90f), Random.Range(-60f, 60f), Random.Range(0f,12f)));
    }
    StartGeneration();
}
```

```
public void StartGeneration()
{
    _currentIndex = 0;
    CurrentGeneration ++;
}
```

Se crea el algoritmo genético y una población del tamaño estipulado en editor, esta población se genera incialmente completamente random para tener una primera tanda que pruebe múltiples soluciones y que luego evoluciones de las mejores de las mismas.

IMPLEMENTACIÓN

Tras esto en el ShotgunConfiguration se corre el bucle de juego que itera por cada individuo y evalua su fitness en base a la distancia de la bola al objetivo

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Time.timeScale = 1f;
        CurrentIndividual = Genetic.GetFittest();
        ShooterConfigure(CurrentIndividual.degreeX, CurrentIndividual.degreeY, CurrentIndividual.strength);
        Shot();
    }

    if (_ready)
    {
        CurrentIndividual = Genetic.GetNext();
        if (CurrentIndividual != null)
        {
            ShooterConfigure(CurrentIndividual.degreeX, CurrentIndividual.degreeY, CurrentIndividual.strength);
            Shot();
        }
        else
        {
            CurrentIndividual = Genetic.GetFittest();
            _ready = false;
        }
    }
}
```

```
public void Shot()
{
    _ready = false;

    transform.eulerAngles = new Vector3(XDegrees, YDegrees,0);
    var shot = Instantiate(ShotSpherePrefab, ShotPosition);
    shot.gameObject.GetComponent<TargetTrigger>().Target = Target;
    shot.gameObject.GetComponent<TargetTrigger>().OnHitCollider += GetResult;
    shot.isKinematic = false;
    var force = transform.up * Strength;
    shot.AddForce(force,ForceMode.Impulse);
}
```

Tras esto en el ShotgunConfiguration se corre el bucle de juego que itera por cada individuo y evalua su fitness en base a la distancia de la bola al objetivo.

```
public Individual GetFittest()
{
    population.Sort();
    return population[0];
}
```

IMPLEMENTACIÓN

Volviendo al script de genetic el endgeneration evalua todos los individuos y ejecuta dependiendo de la configuración del editor Arithmetic o Combined para Crossover y Uniform o Trading en la mutation

```
public void EndGeneration()
{
    population.Sort();
    Summary += $"{GetFittest().fitness};";
    if (CurrentGeneration < MaxGenerations)
    {
        massacre();
        switch (crossoverOperator)
        {
            case CrossoverOperator.ARITHMETIC:
                CrossoverArithmetic();
                break;
            case CrossoverOperator.COMBINED:
                CrossoverCombined();
                break;

        }
        switch (mutationOperator)
        {
            case MutationOperator.UNIFORM:
                mutationUniform();
                break;
            case MutationOperator.TRADING:
                mutationTrading();
                break;

        }
    }
}
```

```
public void CrossoverArithmetic()
{
    //SELECCION
    var ind1 = population[0];
    var ind2 = population[1];


    float r = 0.6f;
    float rMinus = 1 - r;

    float x1 = r * ind1.degreeX + rMinus * ind2.degreeX;
    float x2 = r * ind1.degreeY + rMinus * ind2.degreeY;
    float x3 = r * ind1.strength + rMinus * ind2.strength;
    var new1 = new Individual(x1, x2, x3);

    float y1 = r * ind2.degreeX + rMinus * ind1.degreeX;
    float y2 = r * ind2.degreeY + rMinus * ind1.degreeY;
    float y3 = r * ind2.strength + rMinus * ind1.strength;
    var new2 = new Individual(y1, y2, y3);

    population.Add(new1);
    population.Add(new2);
}
```

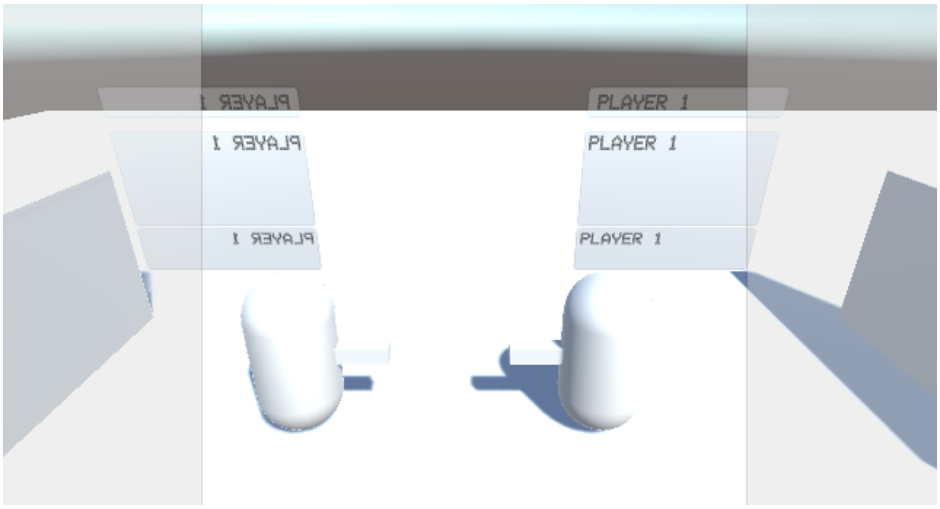
```
public void mutationUniform()
{
    foreach(var individual in population)
    {
        int genToModify = Random.Range(0, 3);
        if (Random.Range(0f, 1f) < 0.02f)
        {
            if (genToModify == 0)
            {
                individual.degreeX = Mathf.Clamp(Random.Range(individual.degreeX - 15, individual.degreeX + 15), 0, 90);
            }
            else if (genToModify == 1)
            {
                individual.degreeY = Mathf.Clamp(Random.Range(individual.degreeY - 15, individual.degreeY + 15), -60, 60);
            }
            else if (genToModify == 2)
            {
                individual.strength = Mathf.Clamp(Random.Range(individual.strength - 3, individual.strength + 3), 0, 12);
            }
            else
            {
                Debug.Log("Not existing gen to modify");
            }
        }
    }
}
```

```
public void CrossoverCombined()
{
    //SELECCION
    var ind1 = population[0];
    var ind2 = population[1];
    float h;

    float alpha = 0.1f;
    float[] values = new float[2];

    for(int i = 0; i<2; ++i)
    {
        h = Mathf.Abs(ind1.degreeX - ind2.degreeX) * alpha;

        values[0] = ind1.degreeX - h; values[1] = ind2.degreeX + h;

        int index = Random.Range(0, values.Length);
        float x1 = values[index];

        h = Mathf.Abs(ind1.degreeY - ind2.degreeY) * alpha;
        values[0] = ind1.degreeY - h; values[1] = ind2.degreeY + h;
        index = Random.Range(0, values.Length);
        float x2 = values[index];

        h = Mathf.Abs(ind1.strength - ind2.strength) * alpha;
        values[0] = ind1.strength - h; values[1] = ind2.strength + h;
        index = Random.Range(0, values.Length);
        float x3 = values[index];
        var new1 = new Individual(x1, x2, x3);

        population.Add(new1);
    }
```

```
public void mutationTrading()
{
    foreach (var individual in population)
    {

        if (Random.Range(0f, 1f) < 0.02f)
        {
            float cacheY = individual.degreeY;
            individual.degreeY = individual.degreeX;
            individual.degreeX = cacheY;
        }
    }
}
```

IMPLEMENTACIÓN

Figthing




```
11 references
public class IndividualOneOne : IComparable<IndividualOneOne>
{


    public float fitness;

    public float energy;
    public float health;

    public float Eenergy;
    public float Ehealth;

    public float[] moveChances;

    3 references
    public IndividualOneOne(float chance1, float chance2, float chance3, float chance4, float health, float energy)
    {
        this.health = health;
        this.energy = energy;
        Eenergy = energy;
        Ehealth = health;
        moveChances = new float[4] { chance1, chance2, chance3, chance4 };
        fitness = 10f;
    }
    0 references
    public int CompareTo(IndividualOneOne other)
    {
        return fitness.CompareTo(other.fitness);
    }
}
```

El individuo tiene recolección de su vida y la vida del enemgo para que a la hora de cargar el estado del enemigo, se cambie satisfactoriamente

IMPLEMENTACIÓN

Figthing

```
public class geneticAlgorithmOneOne
{

    public List<IndividualOneOne> population;
    private int _currentIndex;

    public int CurrentGeneration;
    public int MaxGenerations;

    float initialHealth;
    float initialEnergy;
    1 reference
    public geneticAlgorithmOneOne(int numberOfGenerations, int populationSize,float initialHealth, float initialEnergy)
    {
        CurrentGeneration = 0;
        MaxGenerations = numberOfGenerations;
        this.initialHealth = initialHealth;
        this.initialEnergy = initialEnergy;
        GenerateRandomPopulation(populationSize, initialHealth, initialEnergy);
    }

    1 reference
    public void GenerateRandomPopulation(int size, float initialHealth, float initialEnergy)...

    1 reference
    public IndividualOneOne GetFittest()...

    2 references
    public void StartGeneration()...

    1 reference
    public IndividualOneOne GetNext()...

    1 reference
    public void EndGeneration()...


    1 reference
    public void massacre()...


    1 reference
    public void Mutation()...

    1 reference
    public void CrossoverArithmetic()...
}
```

La implementación del algoritmo comparte casi en su totalidad el modelo y arquitectura del algoritmo de disparo

IMPLEMENTACIÓN

Figthing

```
protected override void Think()
{

    CurrentIndividual = Genetic.GetNext();

    _player.HP = CurrentIndividual.health;
    _player.Energy = CurrentIndividual.energy;

    GameState.ListOfPlayers.Players[_player.EnemyId].HP = CurrentIndividual.Ehealth;
    GameState.ListOfPlayers.Players[_player.EnemyId].Energy = CurrentIndividual.Eenergy;

    geneticConfigure(
CurrentIndividual.moveChances[0],
CurrentIndividual.moveChances[1],
CurrentIndividual.moveChances[2],
CurrentIndividual.moveChances[3],
_player.HP,
_player.Energy,
GameState.ListOfPlayers.Players[_player.EnemyId].HP,
GameState.ListOfPlayers.Players[_player.EnemyId].Energy
);

    if (CurrentIndividual != null)
    {
        _attackToDo = ScriptableObject.CreateInstance<Attack>();


        _attackToDo.AttackMade = _player.Attacks[action()];
        _attackToDo.Source = _player;
        _attackToDo.Target = GameState.ListOfPlayers.Players[_player.EnemyId];

        attacking = true;
    }
    else
    {
        CurrentIndividual = Genetic.GetFittest();
        finished = true;
    }

}
```

La unica diferencia es el think hace un proceso en el que recoge los datos necesarios y se los aplica a la vida del personaje y la del enemgio para que se pueda replicar el estado a como estaban en su momento.

RESULTADOS OBTENIDOS

Mapa 1 (Apuntado de catapulta)

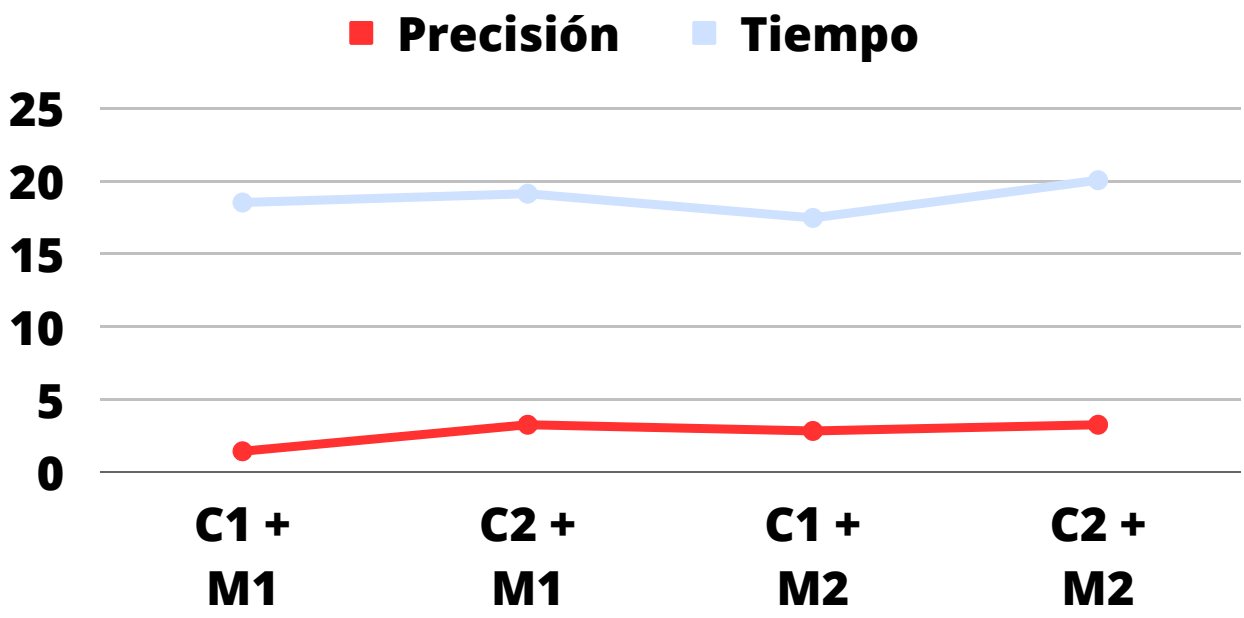
Los siguientes resultados del algoritmo genético usan los siguientes datos:

Generaciones máximas = 30
Veces probado por cada uno por el factor mutación = 10 o 5

Link a la hoja de google sheets usada para ver los resultados y las medias:
Archivo de texto extra en la entrega.

	Población 20	**Población 30**	**Población 40**	**Población 50**
C1 + M1	Precisión: 1,457 Tiempo: 18,524s	Precisión: 1,793 Tiempo: 25,746s	Precisión: 1,733 Tiempo: 31,553s	Precisión: 1,363 Tiempo: 42,985s
C2 + M1	Precisión: 3,277 Tiempo: 19,128s	Precisión: 2,126 Tiempo: 26,63s	Precisión: 2,612 Tiempo: 35,957s	Precisión: 2,213 Tiempo: 37,301s
C1 + M2	Precisión: 2,851 Tiempo: 17,468s	Precisión: 2,246 Tiempo: 26,783s	Precisión: 2,322 Tiempo: 35,576s	Precisión: 2,657 Tiempo: 40,314s
C2 + M2	Precisión: 3,277 Tiempo: 20,055s	Precisión: 2,261 Tiempo: 23,380s	Precisión: 2,005 Tiempo: 35,206s	Precisión: 1,610 Tiempo: 42,663s

C1 = Cruce Combinado / M1 = Mutación Uniforme
C2 = Cruce Artirmético / M2 = Mutación Intercambio



CONCLUSIÓN

Esta práctica ha sido una que en la parte de la catapulta se podía llegar una solución relativamente rápido, mientras que la de el sistema de combate ha sido significativamente mas complicada debido a como esta configurado el sistema de combate.

Esto ha sido una parte bastante dura del desarrollo de la segunda parte al tener que heredar el fitness del otro jugador mezclado con unos % de golpear, lo cual hacían los números bastante más complicados de debuggear y finalmente nos ha dejado sin ser capaces de completar de manera satisfactoria este segundo trozo.