

## Proyecto II: Genetic Kingdom

Deiler Morera Valverde: 2023115868  
Curso: Algoritmos y Estructuras de Datos II  
I Semestre 2025

# Contents

<b>1</b>	<b>Introducción</b>	<b>3</b>
<b>2</b>	<b>Breve Descripción del Problema</b>	<b>3</b>
<b>3</b>	<b>Descripción de la Solución</b>	<b>3</b>
3.1	Implementación por Requerimiento . . . . .	3
3.1.1	Aparición de Enemigos por Oleadas . . . . .	3
3.1.2	Tipos de Enemigos y Atributos . . . . .	3
3.1.3	Pathfinding A* . . . . .	3
3.1.4	Sistema de Torretas . . . . .	3
3.1.5	Algoritmo Genético . . . . .	4
3.2	Alternativas Consideradas . . . . .	4
3.3	Limitaciones y Problemas Encontrados . . . . .	4
<b>4</b>	<b>Diseño General</b>	<b>5</b>
4.1	Diagrama de Clases UML . . . . .	5
4.2	Principales Patrones de Diseño Aplicados . . . . .	5
<b>5</b>	<b>Enlace al Repositorio de Github</b>	<b>6</b>

# 1 Introducción

Genetic Kingdom es un videojuego tipo tower defense donde los enemigos evolucionan mediante un algoritmo genético para superar las defensas del jugador. El juego fue desarrollado en C++ utilizando SFML para el manejo de gráficos y eventos.

## 2 Breve Descripción del Problema

Se busca desarrollar un tower defense donde:

- Los enemigos evolucionan mediante un algoritmo genético.
- Cada enemigo posee atributos de vida, velocidad y resistencias.
- Los enemigos utilizan A\* para encontrar su camino hacia el castillo.
- El jugador puede construir diferentes tipos de torres.
- Se muestran estadísticas en tiempo real de generaciones y mutaciones.

## 3 Descripción de la Solución

### 3.1 Implementación por Requerimiento

#### 3.1.1 Aparición de Enemigos por Oleadas

- Se implementó la clase `WaveManager` que administra enemigos en oleadas.
- Cada oleada genera enemigos en una *queue* y los lanza al mapa a intervalos de 1 segundo.

#### 3.1.2 Tipos de Enemigos y Atributos

- Se implementaron las clases `Ogre`, `DarkElf`, `Harpy` y `Mercenary` derivadas de `WalkingEnemy`.
- Cada tipo tiene diferentes valores de vida, velocidad y resistencias.

#### 3.1.3 Pathfinding A\*

- Se utilizó el algoritmo A\* adaptado para trabajar sobre una matriz de enteros que representa el mapa.
- Los enemigos calculan su camino al momento de ser creados.

#### 3.1.4 Sistema de Torretas

- Se implementó la clase `TowerManager` que maneja la creación, disparo y dibujo de torres.
- Cada torre dispara proyectiles ("flecha", "fuego", "bala") con un intervalo de disparo y daño propio.

### 3.1.5 Algoritmo Genético

- Los enemigos más exitosos de cada oleada son seleccionados.
- Se realiza cruce de atributos y mutaciones aleatorias controladas.
- La población de la siguiente oleada incorpora individuos "híbridos".

### 3.2 Alternativas Consideradas

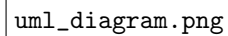
- Se consideró usar Dijkstra en lugar de A\*, pero A\* es más eficiente al tener un objetivo definido (y que también era requisito).
- Inicialmente se pensó en manejar disparos desde `Tile`, pero se optó por `TowerManager` para mantener la responsabilidad aislada.

### 3.3 Limitaciones y Problemas Encontrados

- El ajuste fino de los parámetros genéticos fue desafiante (probabilidad de mutación, presión de selección).
- El manejo de las texturas y su correcta escala para cada tipo de enemigo requirió varias iteraciones.
- Encontrar imágenes aptas para el juego también fue algo tedioso, pero se logró encontrar buenos sprites en:  
<https://craftpix.net/freebies/free-archer-towers-pixel-art-for-tower-defense/>

## 4 Diseño General

### 4.1 Diagrama de Clases UML

A large rectangular box with a thin black border, intended for a UML class diagram. The text 'uml\_diagram.png' is located in the lower-left corner of this box.

uml\_diagram.png

### 4.2 Principales Patrones de Diseño Aplicados

- **Factory Method:** para la creación flexible de enemigos en cada oleada.
- **Strategy:** enemigos pueden cambiar la estrategia de pathfinding si fuese necesario.
- **Singleton (parcial):** `TowerManager` centraliza control de torres.

## 5 Enlace al Repositorio de Github

- <https://github.com/Deiler2024/Proyecto-II>

**Notas Finales:**