Universidad Autónoma de Madrid
Escuela Politécnica Superior

Trabajo de Fin de Grado

INTELIGENCIA COMPUTACIONAL APLICADA A LA GENERACIÓN Y VALIDACIÓN AUTOMÁTICA DE PANTALLAS DE VIDEOJUEGOS

Grado en Ingeniería Informática

Gonzalo Guadalix Arribas
Julio 2013

INTELIGENCIA COMPUTACIONAL APLICADA A LA GENERACIÓN Y VALIDACIÓN AUTOMÁTICA DE PANTALLAS DE VIDEOJUEGOS

AUTOR: Gonzalo Guadalix Arribas
TUTOR: Antonio González Pardo
PONENTE: David Camacho Fernández

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio 2013

Resumen

Resumen

El sector de los videojuegos ha evolucionado considerablemente en los últimos años. Los jugadores no solo buscan buenos gráficos y sonido en un videojuego sino que van más allá, valorando otros aspectos como la rejugabilidad del mismo. La rejugabilidad es un concepto que hace referencia al entretenimiento de jugar a un videojuego más de una vez, ya sea porque al jugador le gustan ciertos aspectos del mismo o porque ofrece contenido adicional. Este aspecto ha presentado un problema, ya que los videojuegos disponen de un número limitado de escenarios en los cuales jugar y éstos no siempre se ajustan a un nivel de dificultad concreto. Como solución a este problema, han surgido diferentes trabajos basados en la generación procedimental del contenido (*Procedural Content Generation*, PCG). Gracias a estos sistemas, la rejugabilidad de los videojuegos ha sido mejorada notablemente. El trabajo presentado en este documento proporciona una nueva solución al problema expuesto. Para ello, se ha desarrollado un algoritmo genético que permite generar un escenario aleatorio a partir de un conjunto de escenarios iniciales, a los cuales llamaremos población. Estos escenarios son creados mediante la utilización de diferentes algoritmos de generación aleatoria. Además, se utiliza un algoritmo de colonia de hormigas como función de *fitness*, que permite la comparación entre escenarios. De esta forma, el algoritmo genético implementado permite crear una gran variedad de escenarios adaptados a un nivel de dificultad específico. Los resultados experimentales permiten extraer diferentes conclusiones. En primer lugar, el sistema propuesto puede ser empleado como una guía para parametrizar correctamente los diferentes algoritmos realizados. En segundo lugar, la utilización de un algoritmo genético complejo resulta eficaz en la generación de escenarios acordes a un nivel de dificultad. Por último, la función de *fitness* basada en una colonia de hormigas proporciona buenos resultados como método de evaluación.

Palabras Clave

Algoritmos genéticos; Algoritmos de optimización mediante colonia de hormigas; Generación de escenarios de videojuegos; Generación procedimental del contenido

Abstract

The gaming industry has evolved considerably in recent years. Players do not just look for nice graphics and sound in a video game, but go beyond, assessing other aspects like the replayability of itself. The replayability is a concept that refers to entertainment of playing a game more than once, either because the player like certain aspects of it or because it provides additional content. This has presented a problem, since the games have a limited number of scenarios in which to play and they do not always conform to a particular difficulty. To solve this problem, there have been various works based on Procedural Content Generation (PCG). Thanks to these systems, the replayability of the video games has been greatly improved. The work presented in this document provides a new solution to the exposed problem. To do this, we have developed a genetic algorithm in order to generate a random level from the evolution of an initial set of levels, called population. Different random algorithms have been used to create the initial levels. Furthermore, the fitness function of the genetic algorithm uses an ant colony algorithm, which allows the comparison between levels. Thus, the implemented genetic algorithm can generate a wide variety of scenarios tailored to a particular difficulty. From the experimental phase different conclusions can be extracted. First of all, the proposed system can be used as a guide to correctly parameterize the different implemented algorithms. Secondly, the use of a complex genetic algorithm is effective to generate scenarios that fit a difficulty level. And finally, the fitness function based on an ant colony algorithm provides good results as an evaluation method.

Key words

Genetic Algorithms; Ant Colony Optimization Algorithms; Game Level Generation; Procedural Content Generation

Índice general

Índice de figuras

1

Introducción

El mundo de los videojuegos ha sufrido una gran evolución a lo largo del tiempo y, más concretamente, en los últimos años. Esta evolución ha afectado a diferentes aspectos de los mismos, como los gráficos, el sonido o el modo de juego. No obstante, otros también han cobrado gran importancia, como la rejugabilidad. En este sector, el término rejugabilidad se emplea para describir el entretenimiento de jugar a un videojuego más de una vez, ya sea porque al jugador le gustan ciertos aspectos del mismo o porque ofrece contenido adicional. El hecho de que un juego no sea rejugable ha supuesto un problema para muchos jugadores. Para combatir este problema se han ofrecido modos y sistemas de juego basados en partidas de corta duración. Sin embargo, este tipo de sistemas a menudo ofrecen un número limitado de escenarios donde jugar, muchos de los cuales no están adaptados a un nivel de dificultad, por lo que pueden resultar tan fáciles que no ofrecen ningún reto o tan difíciles que se pierde el interés por jugar. Como solución a este problema surgen sistemas que emplean diferentes algoritmos de inteligencia artificial para generar escenarios de forma automática y aleatoria. Este tipo de sistemas se basan en la generación procedimental del contenido (*Procedural Content Generation*, PCG). En este aspecto se han realizado diversos trabajos, como el de Cardamone et al. [1], en el cual se generan pistas para un videojuego de carreras de coches, el de Ashlock et al. [2] en donde se opta por el desarrollo de laberintos, o el de Sorenson et al. [3] que hace uso de algoritmos genéticos y programación basada en restricciones (*Constraints Satisfaction Problems*, CSPs) con el fin de generar escenarios entretenidos para diferentes juegos.

El trabajo realizado pretende mostrar una solución más al problema expuesto. Para ello, se desarrolla un sistema PCG basado en la implementación de un algoritmo genético. La población inicial del mismo es generada mediante el uso de diferentes algoritmos de generación aleatoria y la función de *fitness* se lleva a cabo a través de un algoritmo de colonia de hormigas. El sistema propuesto permite generar escenarios para diversos géneros de videojuego en donde el objetivo principal sea movernos con nuestro personaje desde un punto a otro del escenario. De la misma forma, está principalmente orientado a entornos 3D, sin embargo, se podría emplear también en entornos 2D.

Para generar un escenario de manera automática, en primer lugar se genera la población inicial del algoritmo genético mediante los algoritmos de generación aleatoria implementados. Posteriormente, la función de *fitness* cobra gran importancia, dado que se emplea el algoritmo

de colonia de hormigas para evaluar los escenarios que componen la población inicial. Luego, mediante el uso de los operadores de selección, cruzamiento y mutación, se combinan los dos mejores escenarios, dependiendo del nivel de dificultad seleccionado, generando una nueva población. Este proceso se repite tantas veces como sea necesario hasta obtener un escenario ajustado al nivel de dificultad requerido.

En el algoritmo genético, el algoritmo de colonia de hormigas resulta realmente importante, ya no solo como mecanismo de evaluación de escenarios, sino como método para obtener información valiosa acerca de qué caminos son más o menos transitados en los mismos. Gracias a esto, se pueden establecer recursos o enemigos en las diferentes rutas del escenario, que ayuden o perjudiquen al personaje según el nivel de dificultad.

A partir de este momento, el trabajo se estructura como sigue. En la sección 2 se dará una introducción sobre algoritmos genéticos y algoritmos basados en colonias de hormigas y se comentarán los trabajos relacionados existentes y su aplicación en el entorno de los videojuegos. La sección 3 explicará todo lo referente a la implementación del algoritmo genético y su utilización para generar escenarios de manera automática adaptados a un nivel de dificultad. Más concretamente, se explicará cómo funcionan los algoritmos de generación de la población inicial, así como la adaptación del algoritmo de colonia de hormigas como función de *fitness*. Además, se explicará la manera en que se emplean los operadores de selección, cruzamiento y mutación para generar una población mediante la combinación de los mejores escenarios. En la sección 4 se detallarán las pruebas realizadas sobre el algoritmo de colonia de hormigas y el algoritmo genético y los resultados obtenidos de las mismas. Por último, en la sección 5 se comentarán las conclusiones derivadas del proyecto realizado y los aspectos que quedan abiertos del mismo para un posible trabajo futuro.

2

Estado del arte

Esta sección comienza con una pequeña introducción sobre algoritmos genéticos [4]. Posteriormente, se explicarán las bases de los algoritmos basados en inteligencia colectiva [5] y, más concretamente, los algoritmos basados en colonia de hormigas [6]. En ambos casos, se proporcionará una perspectiva orientada a diferentes aspectos relacionados con los videojuegos [7] [8].

2.1. Algoritmos genéticos

A lo largo de los años, los algoritmos genéticos (*Genetic Algorithm*, GA) han sido utilizados en una gran variedad de campos para encontrar soluciones a problemas de optimización [9][10][4]. Este tipo de algoritmos son encontrados en la resolución de problemas clásicos, como el problema de enrutamiento de vehículos (*Vehicle Routing Problem*, VRP) [11][12] o el problema del viajante de comercio (*Travelling Salesman Problem*, TSP) [13][14].

En el campo de la inteligencia artificial, los GA tratan de simular el comportamiento de la evolución natural de las especies. Para ello, parten de un conjunto de individuos lo más variado posible, al cual llamaremos población [1]. Cada individuo de la población representa una posible solución al problema propuesto. A su vez, cada individuo consiste en un conjunto de características o genes [2] que pueden tomar diferentes valores. Los valores de estos genes constituyen el genotipo de la solución y su representación física el fenotipo. El objetivo del GA será evolucionar las mejores soluciones con el fin de encontrar una solución óptima al problema. Esta evolución se lleva a cabo mediante la aplicación de tres operadores: selección, cruzamiento y mutación.

Selección

La selección es un operador genético cuyo objetivo es determinar qué individuos de la población resultan más apropiados para evolucionar. Para realizar esta tarea, utiliza una función de *fitness*, que le permite evaluar a los individuos basándose en diferentes criterios y asignarles un valor en función de la calidad de la solución que representan. Gracias a la función de *fitness*, los mejores individuos pueden ser evolucionados. Una vez se han

[1] La población inicial en los algoritmos genéticos se genera habitualmente de manera aleatoria.
[2] Cuando se forman largas cadenas de genes se habla de cromosomas.

evaluado los individuos, y dependiendo del tipo de selección que se desea realizar, el procedimiento a seguir es diferente. Existen diferentes tipo de selección, como la selección de ruleta, selección por torneo, selección por truncamiento, etc. Por ejemplo, en la selección de ruleta (*roulette wheel selection*, también conocida como *fitness proporcionate selection*) se calcula la probabilidad de que un individuo sea seleccionado como

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{2.1}$$

donde

N Representa el número de individuos de la población.

f_i Representa el valor *fitness* calculado para la solución i.

Mediante este tipo de selección, los individuos con mejor valor de *fitness* tienen más probabilidad de ser seleccionados. La idea de esta selección radica en que incluso los individuos con peor valor de *fitness* pueden tener genes que sean valiosos para la evolución y no deben ser descartados sin tenerlos en cuenta.

Cruzamiento

El cruzamiento constituye la operación básica del algoritmo genético. Consiste en la generación de un nuevo individuo (sucesor) mediante la combinación de dos o más individuos de la población (progenitores). Existen diferentes tipos de combinaciones, de las cuales la más sencilla es el corte en un punto. Si representamos los individuos progenitores como cadenas de valores, consiste en escoger una posición aleatoria a partir de la cual intercambiar los progenitores para formar dos sucesores. El primer sucesor surge de la unión de la primera parte del primer progenitor y la segunda parte del segundo progenitor, mientras que el segundo sucesor surge de la unión de la segunda parte del primer progenitor y la primera parte del segundo progenitor.

Figura 2.1: Cruzamiento mediante corte en un punto (*single-point crossover*).

Mutación

Una vez generado un sucesor mediante el cruzamiento se lleva a cabo la mutación. La mutación consiste en la modificación de uno o varios genes de las cadenas de cromosomas de los sucesores. Estos genes tienen una probabilidad de mutar y ésta debe ser baja. Si se establece muy alta, los individuos cambiarán demasiado y se terminará realizando una búsqueda aleatoria. Gracias a la mutación se consigue mantener una población de individuos variada.

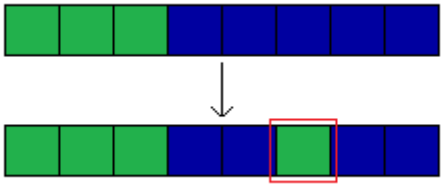


Figura 2.2: Mutación de un sucesor.

Algorithm 1: Algoritmo genético simple

```
1  Generar la población inicial aleatoriamente de tamaño N
2  Evaluar la población mediante la función de fitness
3  while not fin do
4  |  while Tamaño nueva población < N do
5  |  |  Seleccionar los progenitores
6  |  |  Realizar el cruzamiento de los individuos seleccionados
7  |  |  Realizar la mutación de los sucesores generados
8  |  |  Introducir dichos sucesores en la nueva población
9  |  end
10 |  Evaluar la nueva población mediante la función de fitness
11 |  if población ha convergido then
12 |  |  fin ← true
13 |  end
14 end
```

El GA termina cuando se cumple su condición de parada o se han realizado un número determinado de iteraciones. Esta condición suele tener que ver con la convergencia a una solución concreta. La función de *fitness* es la que determina a qué tipo de solución converge el algoritmo, ya que es la que realiza la evaluación de los individuos de la población. Por ejemplo, si la función de *fitness* evalúa mejor los individuos con cierta característica, estos individuos tendrán más probabilidad de ser seleccionados para generar la nueva población. Así mismo, los individuos de la nueva población que mejoren dicha característica serán nuevamente seleccionados para el cruzamiento. De esta forma, tras varias iteraciones se habrá conseguido maximizar esa característica. Por otro lado, llegará un momento en el cual la diferencia entre los individuos de una generación o ente los individuos de una generación y otra anterior será mínima. En este momento se podrá considerar que el algoritmo ha convergido a una solución. No obstante, si no se atiende a la parametrización del algoritmo (probabilidad de mutación, probabilidad de selección, tamaño de la población, etc.) se podrá incurrir en una convergencia prematura. Esto ocurrirá cuando los sucesores no consigan mejorar las características de sus predecesores tras pocas o casi ninguna iteración del GA. Suele producirse por la pérdida de información valiosa de una generación a otra. Algunas técnicas que previenen de la convergencia prematura son la prevención de incesto (*incest prevention* [15]) o el cruzamiento uniforme (*uniform crossover*). Por todo lo mencionado, se puede concluir que la importancia del GA radica principalmente en su función de *fitness* y la parametrización que se utilice.

Cabe destacar que algunos de los operadores del GA pueden resultar costosos computacionalemente. Por ello han surgido algunos trabajos que tienen como objetivo optimizar el tiempo y los recursos invertidos en dichas operaciones [16].

En el entorno de los videojuegos, se han empleado algoritmos genéticos para mejorar ciertos aspectos de los mismos. Estas mejoras han influido en la inteligencia artificial de los enemigos y personajes no jugables (*Non-Playable Characters*, NPCs), en la generación automática de escenarios, etc. Por ejemplo, el trabajo de Miles et al. [17] se basa en la evolución de una inteligencia artificial para juegos de estrategia en tiempo real capaz de tomar decisiones sobre dónde y cómo atacar, defender u obtener recursos basado en grafos de dependencias y mapas de influencia. Otros trabajos de interés son el de Cole et al. [18], basado en la evolución de la inteligencia artificial de los enemigos en juegos de disparos en primera persona (*First-Person Shooters*, FPSs) o el trabajo de Kendall y Spoerer [19], el cual utiliza algoritmos genéticos para resolver escenarios del videojuego *Lemmings*. También se pueden encontrar trabajos relacionados con la generación automática de escenarios, como el de Sorenson y Pasquier [7], en el cual se utilizan los algoritmos genéticos para generar escenarios compatibles con una gran variedad de géneros de videojuegos.

2.2. Algoritmos de inteligencia colectiva

El término inteligencia colectiva o de enjambre[3] fue introducido por Gerardo Beni y Jin Wang en el año 1989 en su artículo sobre sistemas robóticos móviles [5].

Los algoritmos que emplean inteligencia colectiva se caracterizan por la utilización de múltiples agentes[4] que interactúan entre ellos y con su propio entorno para la realización de un objetivo común. Estos algoritmos se basan en el comportamiento que tienen ciertas especies animales en la naturaleza, como las hormigas o las abejas.

Entre estos algoritmos podemos distinguir la optimización de enjambre de partículas (*Particle Swarm Optimization*, PSO) [20], la optimización de colonia de hormigas (*Ant Colony Optimization*, ACO) [6], la optimización de colonia de abejas (*Bee Colony Optimization*, BCO) [21], la optimización de alimentación bacteriana (*Bacterial Foraging Optimization*, BFO) [22], principalmente.

A continuación nos centramos en los algoritmos basados en colonias de hormigas. Los ACO tratan de simular el comportamiento que presentan las hormigas en la naturaleza [6]. Resultan de gran utilidad en la búsqueda de soluciones a problemas de optimización [23] y, al igual que los GA, han sido empleados en la resolución de problemas clásicos, como el VRP [24][25] o el TSP [26] [27], entre otros.

En estos algoritmos, los problemas suelen representarse en forma de grafos y, como otros algoritmos de la rama de la inteligencia colectiva, basan su comportamiento en la utilización de agentes. En el caso de los ACO, dichos agentes son las hormigas, las cuales realizan una función básica, como moverse de un nodo a otro del grafo. Gracias a la interacción entre las hormigas se consigue que desempeñen una tarea más importante: la búsqueda de caminos óptimos entre dos puntos del grafo. Cada uno de los caminos encontrados representa una solución al problema tratado.

La interacción entre las hormigas se consigue mediante la utilización de feromonas [5]. Las feromonas representan información sobre lo positivo o negativo que fue para una hormiga realizar

[3]Se hace referencia al término inteligencia de enjambre como traducción directa del inglés del concepto *swarm intelligence*.

[4]Este tipo de sistemas son denominados sistemas multi-agente.

[5]El intercambio de información mediante la utilización de feromonas en las colonias de hormigas se conoce como estigmergia

un movimiento hacia un nodo u otro en cierto momento, por lo tanto, se entienden como un histórico de movimientos de las hormigas a través del grafo. Las feromonas se depositan en las aristas del grafo, de esta forma, cuando se encuentra una solución se actualizan las aristas por las cuales pasó la hormiga desde el nodo inicial hasta el final. El valor de la feromona establecido en las aristas depende de la calidad de la solución. Si la solución obtenida es buena, el valor depositado sera mayor que si la solución se considera mala. Por otro lado, la calidad de la solución no se puede definir *a priori* y depende del dominio de la aplicación, aunque generalmente se suelen evaluar mejor los caminos más cortos.

Además de la información proporcionada por las feromonas, las hormigas se sirven de una función, denominada función de evaluación, para determinar la arista por la cual se moverán hacia el siguiente nodo. Esta función de evaluación representa lo bueno que resulta el movimiento hacia un determinado nodo y suele calcularse como $\eta_{x\to y} = \frac{1}{d_{x\to y}}$, donde $d_{x\to y}$ representa la distancia entre el nodo x e y.

Como ya se ha mencionado, las feromonas y la función de evaluación permiten a las hormigas seleccionar la arista a través de la cual se moverán. Para realizar esta selección calcularán la probabilidad de moverse de un nodo a otro como

$$p_{x\to y} = \frac{\tau^{\alpha}_{x\to y}\,\eta^{\beta}_{x\to y}}{\sum_{y\epsilon[nodos\ vecinos]}\tau^{\alpha}_{x\to y}\,\eta^{\beta}_{x\to y}} \tag{2.2}$$

donde

x Representa el nodo del cual parte la hormiga.

y Representa el nodo al cual se dirige la hormiga.

$\tau_{x\to y}$ Representa el valor de las feromonas en la arista entre el nodo x e y.

$\eta_{x\to y}$ Representa el valor que la función de evaluación asigna al movimiento del nodo x al nodo y.

α Representa la importancia de las feromonas. Su valor teórico habitual es $\alpha \geq 0$.

β Representa la importancia de la función de evaluación. Su valor teórico habitual es $\beta \geq 1$.

Una vez que las hormigas llegan al nodo final y, por lo tanto, han encontrado una solución al problema tratado, se encargan de depositar feromonas en las aristas que interconectan los nodos por los cuales han pasado. El valor de feromonas, como ya se ha comentado, depende de la calidad de la solución. Algunos criterios que se suelen tener en cuenta son la longitud de la solución encontrada o el número de giros realizados (una solución directa se evaluaría mejor que otra que realice muchos cambios de dirección). Gracias a estas feromonas, las aristas se actualizarán y otras hormigas utilizarán la información de las soluciones encontradas por sus compañeras para mejorar su decisión de movimiento.

En la línea 2 del algoritmo 2 se habla de eliminar los nodos adyacentes que no son accesibles desde el nodo actual. Esta operación puede ser necesaria en ciertos casos en los que el problema tratado lo requiera o para satisfacer ciertas condiciones (e.g. evitar la selección de un nodo ya visitado). En segundo lugar, cabe destacar que la selección proporcional que se realiza en la línea 7 es similar a la selección de ruleta de los GA. Por último, en la línea 15 se establece que es necesario reiniciar la hormiga. Esto consiste en situar la hormiga en el nodo inicial y vaciar la lista de nodos visitados.

Algorithm 2: Funcionalidad de una hormiga

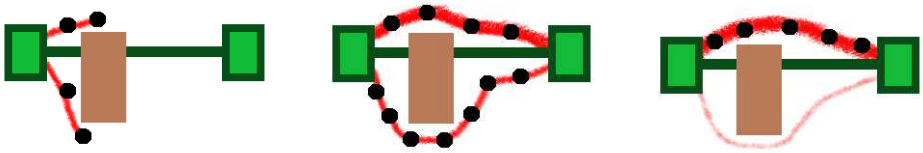
```
1  Obtener los nodos adyacentes al nodo actual
2  Eliminar los nodos adyacentes inaccesibles
3  n ← número de nodos adyacentes accesibles
4  for i ← 0 to n do
5  |  Calcular probabilidad de movimiento hacia el nodo i
6  end
7  Elegir el nodo al cual moverse mediante selección proporcional
8  Mover la hormiga al nuevo nodo
9  Agregar el nuevo nodo a la lista de nodos visitados
10 if nuevo nodo = nodoFinal then
11 |  ph ← evaluarsolucion
12 |  for x, yϵ nodos visitados do
13 |  |  Actualizar feromonas de la arista x → y mediante ph
14 |  end
15 |  Reiniciar hormiga
16 end
```

Generalmente, cada agente del ACO realiza la operación descrita en el algoritmo 2 durante una serie de iteraciones o *steps*. En concreto, un *step* consiste en la realización de una única iteración por cada uno de los agentes que constituyen la colonia de hormigas. De esta forma, si tuviésemos 3 hormigas y realizásemos 10 *steps* en total, el pseudo-código detallado en el algoritmo 2 se ejecutaría 30 veces. Aunque este tipo de algoritmos suelen ejecutarse durante una serie de *steps*, también cabe la posibilidad, al igual que ocurría con los GA, de que se ejecuten hasta alcanzar un criterio de convergencia determinado (e.g. encontrar una solución con unas características determinadas).

Algorithm 3: Algoritmo de optimización mediante colonia de hormigas

```
1  Generar hormigas
2  while i < número de steps do
3  |  for i < número de hormigas do
4  |  |  Realizar una iteración de la hormiga i
5  |  end
6  |  Actualizar feromonas
7  end
```

En la línea 6 del algoritmo 3 se observa que es necesario actualizar las feromonas. Esta actualización consiste en la evaporación de las feromonas con el paso del tiempo. De esta forma, los caminos más largos hacia la fuente de alimento tenderán a desaparecer y los más cortos se harán mas intensos debido a la acumulación de feromonas.



(a) Etapa de exploración (b) Se encuentran soluciones (c) Solución óptima

Figura 2.3: Inicialmente 2.3(a), las hormigas exploran el terreno en busca de comida. Según se van encontrando soluciones 2.3(b), el rastro de feromonas que dejan las hormigas se hace más intenso. Finalmente 2.3(c), las feromonas acumuladas en los caminos más largos se acaban evaporando, mientras que las de los caminos más cortos se intensifican permitiendo a las hormigas llegar rápidamente hasta la comida.

Los algoritmos basados en colonia de hormigas han sido aplicados en multitud de campos, como la minería de datos [28] o las redes sociales [29]. Un ejemplo de la aplicación de estos algoritmos a la minería de datos es la resolución de problemas de *clustering* [30].

En los videojuegos, han sido utilizados en la creación de jugadores para juegos como *Ms. Pac-Man* [31] o *Tetris* [32]. También se han empleado en la búsqueda de caminos [33] y en la generación de escenarios [8].

3

Dominio de la aplicación

En esta sección se detallará en primer lugar el caso de estudio al que se dirige este trabajo. Posteriormente, se mostrará una adaptación del GA al caso de estudio concreto.

3.1. Caso de estudio

Este trabajo está dirigido a todo videojuego en donde el objetivo sea movernos con nuestro personaje desde un punto a otro del escenario. Por lo tanto, es aplicable a diferentes géneros de videojuegos y puede utilizarse para entornos, tanto en dos como en tres dimensiones. No obstante, este trabajo ha estado orientado desde un inicio a la generación de escenarios aleatorios para el videojuego *Kill'em All*.

Kill'em All es un videojuego de disparos en tercera persona (*Third-Person Shooter*, TPS) en donde el objetivo principal es mover al personaje seleccionado desde un punto a otro del escenario. Sin embargo, este movimiento se ve dificultado por los enemigos que atacarán al jugador. Por su parte, el jugador cuenta con una serie de armas y recursos que le permiten eliminar a los enemigos y abrirse camino hacia el objetivo.

Este videojuego está desarrollado íntegramente en Java mediante el motor de juego jMonkeyEngine (jME). Este motor es libre y ha sido diseñado específicamente para la realización de videojuegos en tres dimensiones. Hace uso de la librería de código abierto LWJGL (*Lightweight Java Game Library*), la cual ha sido diseñada para proporcionar a los desarrolladores acceso a tecnologías que actualmente no existen o no están debidamente implementadas en la plataforma de Java.

Cabe mencionar, además, la integración de la librería MASON con el proyecto para realizar simulaciones de una colonia de hormigas. Esta librería está implementada completamente en Java y permite la realización de simulaciones de sistemas multi-agente.

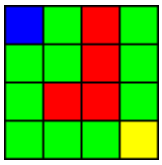
Como ya se ha comentado, el objetivo es generar escenarios para videojuegos. Un escenario se define como un mapa o matriz bidimensional formado por un conjunto de casillas. Por lo tanto, una casilla representa la unidad mínima de un escenario. Existen cuatro tipos de casillas diferentes:

Secciones de suelo Representan partes del escenario por las cuales el personaje puede moverse libremente.

Secciones de pared Representan obstáculos a través de los cuales el personaje no puede pasar.

Posición de inicio Representa la posición de la cual parte el jugador.

Posición objetivo Representa la posición a la cual debe llegar el jugador.



(a) Mapa bidimensional

(b) Representación tridimensional

Figura 3.1: Estructura de un escenario sencillo. Se representan las secciones de suelo de color verde, las secciones de pared de color rojo, la posición de inicio de color azul y la posición objetivo de color amarillo.

Como se puede observar en la figura 3.1, los mapas bidimensionales son interpretados para representar los escenarios tridimensionales. Esta interpretación consiste en la asignación de modelos 3D a cada tipo de casilla del escenario.

Cabe mencionar que a lo largo de este trabajo se generan las posiciones de inicio y fin en esquinas opuestas de la diagonal principal por simplicidad y constancia. No obstante, dichas posiciones podrían ser generadas en cualquier punto del escenario. De la misma forma, se establece una dimensión máxima de 100x100 y una dimensión mínima de 30x30 para los escenarios.

3.2. Implementación realizada

Como se vio en la sección 2, mediante los GA se puede dar solución a problemas de optimización. El problema en este caso se define como la búsqueda de un escenario que cumpla con ciertas condiciones de dificultad. Esta búsqueda terminará cuando se hayan generado un número determinado de poblaciones de individuos (especificadas por parámetro). Para realizar esta implementación ha sido necesario adaptar los conceptos de los GA al problema a tratar.

Los individuos de la población son los escenarios. Esto implica que cada escenario es una posible solución al problema de búsqueda planteado. Así mismo, cada casilla representa un gen. De esta forma, podemos definir el genotipo de un escenario como la matriz de casillas que lo conforma. Por otro lado, el fenotipo será la representación física del escenario en un entorno de dos o tres dimensiones.

A continuación, se detalla la manera en que se genera la población inicial del GA y la función de *fitness* implementada, a través de la cual se convergerá a un tipo de escenarios concretos.

3.2.1. Generación de la población inicial

La población inicial se define como un conjunto de escenarios aleatorios. Dado que cada escenario representa una solución al problema, la población debe ser lo más variada posible con el fin de

encontrar un individuo óptimo. A continuación se presentan diferentes algoritmos de generación de escenarios que tratan de cumplir este objetivo.

NOTA: Dependiendo del algoritmo utilizado en su generación, un escenario será de tipo *paredes*, *caminos* o *clusters*.

Algoritmo basado en generación de paredes

Este algoritmo parte de un escenario inicial vacío y lo genera estableciendo cada casilla como una sección de pared o suelo en función de una serie de probabilidades especificadas como parámetro. En concreto, una casilla será una sección de pared con una cierta probabilidad según cumpla unas determinadas condiciones. Las probabilidades y condiciones que se han tenido en cuenta son las siguientes:

- Probabilidad de formar cuadrados. Si junto con otras tres casillas del escenario, la casilla a la que se le quiere asignar un valor fuese una sección de pared y éstas formasen un cuadrado, la probabilidad de esta casilla de ser una pared se ve modificada por el parámetro *squareProbability*.

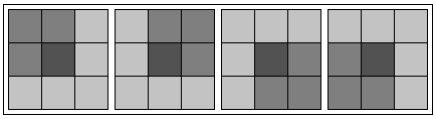


Figura 3.2: Formación de cuadrados en un escenario

- Probabilidad de formar esquinas. Si junto con otras dos casillas del escenario, la casilla a la que se le quiere asignar un valor fuese una sección de pared y éstas formasen una esquina, la probabilidad de esta casilla de ser una pared se ve modificada por el parámetro *cornerProbability*.

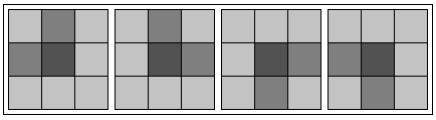


Figura 3.3: Formación de esquinas en un escenario

- Probabilidad de formar una pared perpendicular a otra. Si ya existe una pared y la casilla a la que se le quiere asignar un valor fuese una sección de pared perpendicular a la misma, la probabilidad de esta casilla de ser una pared se ve modificada por el parámetro *newWallProbability*.

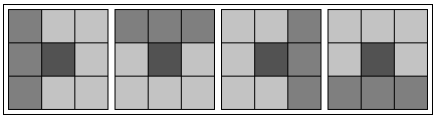


Figura 3.4: Formación de nuevas paredes en un escenario

- Probabilidad de continuar una pared. Si ya existe una sección de pared antes de la casilla a la que se le quiere asignar un valor y ésta fuese otra sección de pared, la probabilidad de esta casilla de ser una pared se ve modificada por el parámetro *continuousWallProbability*. Esta probabilidad disminuye según aumenta la longitud de la pared.

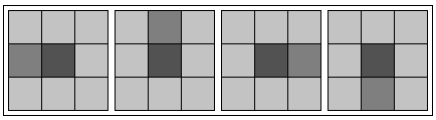


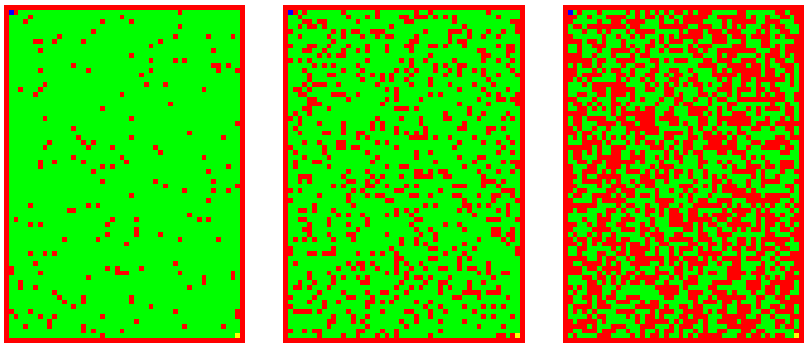
Figura 3.5: Continuación de paredes en un escenario

- Probabilidad por defecto. Si no se cumplieron ninguna de las condiciones anteriores, la probabilidad de esta casilla de ser una pared se ve modificada por el parámetro *defaultProbability*. Consecuentemente, la probabilidad de una casilla de ser una sección de suelo sería *1 - defaultProbability*.

Algorithm 4: Algoritmo de generación de paredes

```
1  for i ← 0 to numero de casillas do
2  |   c ← casilla i-ésima
3  |
4  |   if c forma un cuadrado then
5  |   |   p ← squareProbability
6  |   else if c forma una esquina then
7  |   |   p ← cornerProbability
8  |   else if c forma una nueva pared then
9  |   |   p ← newWallProbability
10 |   else if c forma una continuación de pared then
11 |   |   p ← continuousWallProbability
12 |   else
13 |   |   p ← defaultProbability
14 |   end
15 |
16 |   r ← generar numero aleatorio entre [0, 1]
17 |   if r < p then
18 |   |   c es sección de pared
19 |   else
20 |   |   c es sección de suelo
21 |   end
22 end
```

El algoritmo permite la creación de escenarios variados dado que posee un alto grado de parametrización. Por otro lado, dado que se basa en probabilidades, se observa que los escenarios generados son principalmente formaciones de nubes de puntos donde cabe destacar la mayor o menor densidad de las mismas.



(a) Densidad baja (b) Densidad media (c) Densidad alta

Figura 3.6: Diferentes escenarios creados mediante el algoritmo de generación de paredes

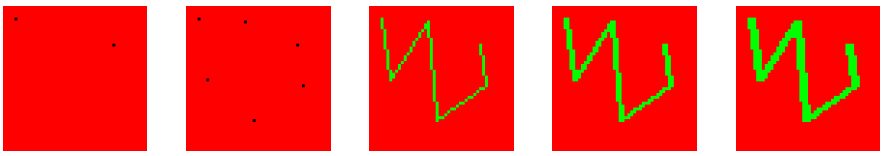
Algoritmo basado en generación de caminos

Este algoritmo parte de un escenario donde todas las casillas son secciones de pared. A partir de esta situación, genera varios caminos con secciones de suelo de un punto a otro del escenario. Para ello calcula dos puntos aleatorios que serán el comienzo y fin del camino, luego calcula otros n puntos aleatorios, que representan los tramos del camino, y los une mediante líneas rectas de un grosor nuevamente aleatorio. Se repite esta operación hasta haber completado el número de caminos requerido. En este algoritmo intervienen los siguientes parámetros principalmente:

- Número de caminos. Indica el número mínimo y máximo de caminos principales que tendrá el escenario.

- Número de tramos del camino. Indica el número mínimo y máximo de tramos que pueden tener los caminos principales.

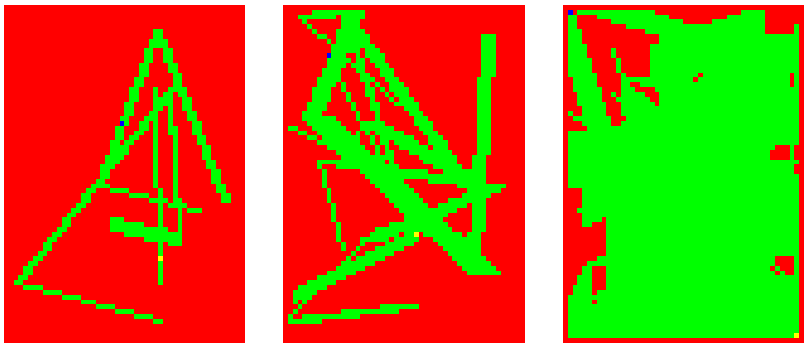
- Anchura de los caminos. Indica la anchura mínima y máxima que tendrán los tramos y caminos.



(a) Paso 1 (b) Paso 2 (c) Paso 3 (I) (d) Paso 3 (II) (e) Paso 3 (III)

Figura 3.7: Como se puede observar, en 3.7(a) se escogen dos puntos aleatorios del escenario. A continuación, en 3.7(b) se escogen otros n puntos aleatorios. Por último, en 3.7(c) se unen los puntos mediante líneas rectas. En 3.7(d) y 3.7(e) se muestra cómo hubiese quedado el camino empleando un grosor de 2 o 3 unidades, respectivamente.

Los escenarios generados mediante este algoritmo resultan variados. Además, presentan formaciones laberínticas de diversa complejidad, por lo que resulta sencillo diferenciar los escenarios por su dificultad.



(a) Pocos caminos y tramos de poco grosor

(b) Algunos caminos y tramos de grosor medio

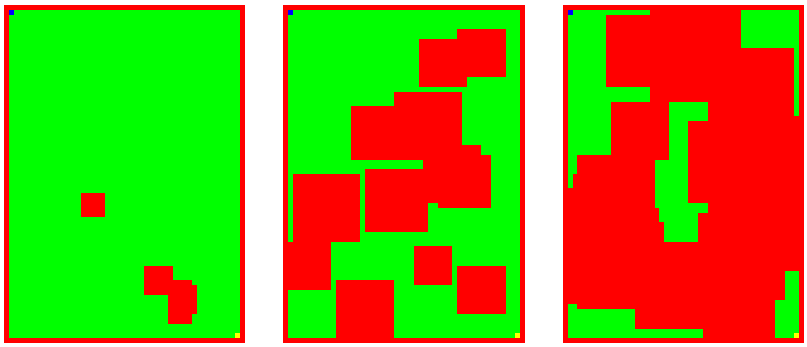
(c) Muchos caminos y tramos de gran grosor

Figura 3.8: Diferentes escenarios creados mediante el algoritmo de generación de caminos

Algoritmo basado en generación de *clusters*

Este algoritmo parte de un escenario donde todas las casillas son secciones de suelo. A partir de esta situación, se genera un número aleatorio de agrupaciones de paredes o *clusters* de una cierta dimensión. Los parámetros que intervienen en la generación de este tipo de escenarios son:

- Número de *clusters*. Indica el número mínimo y máximo de *clusters* que tendrá el escenario.
- Tamaño de los *clusters*. Indica el tamaño mínimo y máximo que tendrán los *clusters*.



(a) Escenario con pocos *clusters*

(b) Escenario con algunos *clusters*

(c) Escenario con muchos *clusters* (sin solución)

Figura 3.9: Diferentes escenarios creados mediante el algoritmo de generación de *clusters*

No existe ningún tipo de restricción respecto a la situación y/o tamaño de los *clusters*, no obstante, si se establecen parámetros con valores demasiado bajos o altos se obtendrán escenarios prácticamente sin paredes o con tantas que no existirá una solución a los mismos, respectivamente.

Ventajas e inconvenientes de cada algoritmo

Como ya se ha mencionado previamente, el objetivo principal de estos algoritmos es generar escenarios variados, dado que serán combinados posteriormente entre sí y se desea una población

lo más diversa posible. Sin embargo, no todos los algoritmos consiguen siempre este objetivo. A continuación se comentan las ventajas e inconvenientes de cada algoritmo.

Algoritmo	Ventajas	Inconvenientes
Basado en generación de paredes	Alta tasa de escenarios con solución. Escenarios variados en densidad.	Escasa diversidad al combinarlos mediante el GA.
Basado en generación de caminos	Alta tasa de escenarios con solución. Gran variedad de escenarios.	Alta tasa de escenarios sin solución al combinarlos mediante el GA.
Basado en generación de *clusters*	Alta tasa de escenarios con solución. Gran variedad de escenarios.	Dependiendo de los parámetros establecidos, la tasa de escenarios sin solución puede aumentar considerablemente al combinarlos mediante el GA.

3.2.2. Adaptación de los operadores de cruzamiento y mutación

Dado que los individuos del GA son escenarios, es necesario adaptar los operadores de cruzamiento y mutación que se vieron de manera teórica en la sección 2.

Cruzamiento de escenarios

Como ya se ha visto, los escenarios se representan mediante matrices de casillas. De esta forma, podemos definir la operación de cruzamiento como la combinación de dos matrices mediante corte en un punto. Para ello se realizan los siguiente pasos:

- Escoger tipo de corte. Se debe escoger el tipo de corte a realizar (horizontal o vertical). Esta selección se realiza de manera equitativa, con un 50 % de probabilidad para cada opción.
- Escoger posición de corte. Una vez escogido el tipo de corte se debe escoger la fila o columna donde cortar, dependiendo de si se realizó un corte horizontal o vertical, respectivamente.
- Combinación. El sucesor será generado como una combinación de los dos progenitores. Si se realizó un corte horizontal, se obtendrá la parte superior de un progenitor y la parte inferior del otro. Si se realizó un corte vertical, se obtendrá la parte izquierda de un progenitor y la parte derecha del otro.

Como se ha explicado previamente, cada escenario es de un tipo determinado dependiendo del algoritmo con el que fue generado. Así, cada progenitor tendrá su propio tipo, al igual que el sucesor. Si los progenitores son del mismo tipo, el sucesor lo será también. Por otro lado, si son de diferentes tipos, el sucesor tendrá un 50 % de probabilidad de ser de un tipo o de otro.

La dimensión del sucesor se calcula a partir de la media de altura y anchura de sus progenitores

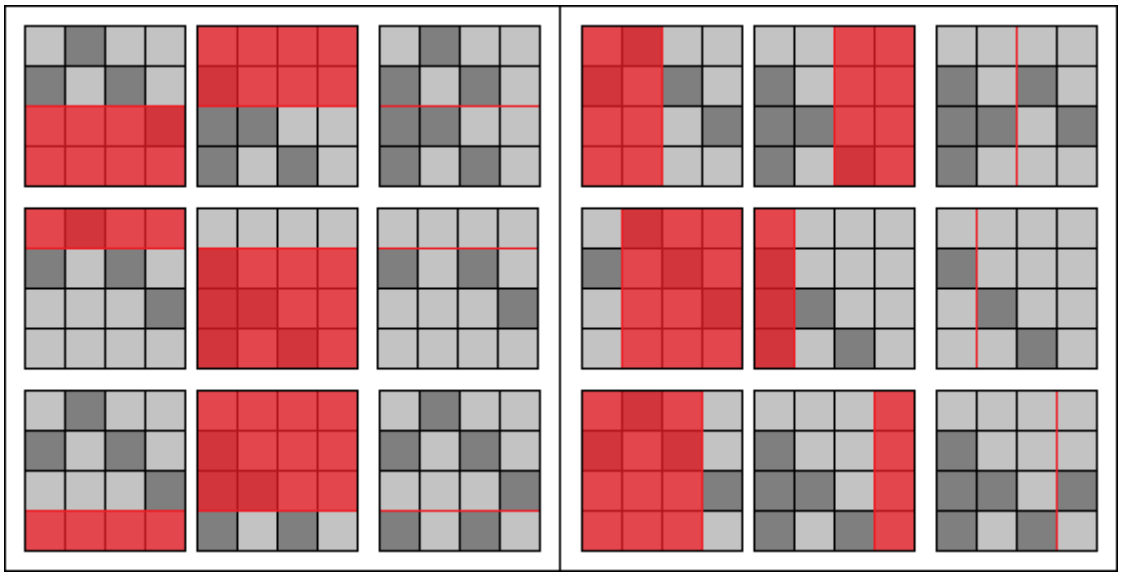


Figura 3.10: Cruzamientos entre escenarios

variando dichas medidas en como mucho un 20 %. De esta forma, se garantiza que tendrá relación con la dimensión de sus progenitores, pero no tanto como para que no exista diversidad en las dimensiones de los diferentes escenarios de la población.

Por último, cabe mencionar que dado que los escenarios son de dimensiones variables, es posible que se intente cortar un progenitor en una posición que supere su dimensión. Por ello, se establece cada corte de tal manera que el sucesor obtenga al menos una fila o columna de cada uno de sus progenitores.

Mutación de escenarios

Mediante el cruzamiento se obtiene un sucesor a partir de la combinación de dos escenarios. La mutación consiste en modificar este sucesor para generar variaciones en la población de escenarios. Estas modificaciones, no sólo por su dimensión sino también por su cantidad, deben ser lo suficientemente notorias como para crear una población variada, pero no tanto como para aleatorizar la generación de sucesores.

Dependiendo del tipo de escenario (*paredes*, *caminos* o *clusters*) se realiza una mutación diferente, aunque en todo caso se modifica un 1 % del escenario como máximo. En concreto, se distinguen tres tipos de mutaciones:

- Mutación de paredes. Se aplica a escenarios creados mediante el algoritmo de generación de paredes. Las casillas que eran secciones de suelo pasarán a ser secciones de pared y viceversa.
- Mutación de caminos. Se aplica a escenarios creados mediante el algoritmo de generación de caminos. Se utilizan las secciones de suelo para crear nuevos caminos de grosor aleatorio.
- Mutación de *clusters*. Se aplica a escenarios creados mediante el algoritmo de generación de *clusters*. Se utilizan secciones de pared y de suelo para generar nuevos *clusters* de tamaño aleatorio.

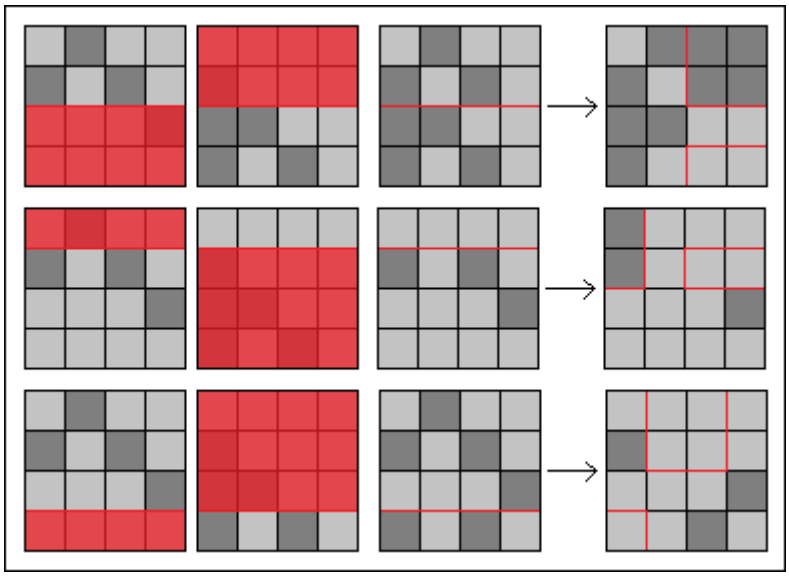


Figura 3.11: Mutación de sucesores

3.2.3. Función de *fitness* y selección de individuos

Como se vio en la sección 2, el ACO es utilizado para problemas de optimización. En este caso, se ha realizado una implementación del ACO para utilizarla como función de *fitness* en el GA. Para ello, en primer lugar se explicará en detalle el ACO implementado y, posteriormente, se mostrará la adaptación realizada para su utilización como función de *fitness*. Por último, se detallará el proceso de selección que se utiliza en el GA.

Implementación de una colonia de hormigas

El problema a tratar se define como la búsqueda de caminos en un grafo no dirigido. Cada escenario representa un grafo y cada casilla un nodo del mismo. A su vez, cada nodo está conectado con otros ocho nodos, exceptuando los que se sitúan en las esquinas y bordes del escenario que están conectados con tres y cinco nodos, respectivamente. Por otro lado, dado que existen diferentes tipos de casillas, también se deben definir distintos tipo de nodos. En concreto, cada casilla se representa de la siguiente forma:

- Secciones de pared. Representan nodos no visitables del grafo.
- Secciones de suelo. Representan nodos visitables del grafo. Se pueden depositar feromonas sobre los mismos.
- Posición de inicio. Nodo visitable del grafo del cual parten las hormigas. Representa el hormiguero. Se pueden depositar feromonas sobre el mismo.
- Posición de fin. Nodo visitable del grafo al que deben dirigirse las hormigas. Representa la fuente de alimento. Se pueden depositar feromonas sobre el mismo.

El objetivo de las hormigas es llegar al nodo final partiendo del nodo inicial por el camino más corto posible. El movimiento de las hormigas hacia uno u otro nodo viene determinado por una función de evaluación (heurística) y las feromonas acumuladas en cada uno de los nodos. Una buena función de evaluación permitirá obtener soluciones más rápido, mientras que las feromonas permitirán que otras hormigas puedan obtener las mismas u otras soluciones

similares.

Calcularemos la probabilidad de una hormiga de moverse a un determinado nodo como

$$P_{x \to y} = \frac{\tau_y^{\alpha} \, \eta_y^{\beta}}{\sum_{y \in visitables} \tau_y^{\alpha} \, \eta_y^{\beta}} \tag{3.1}$$

donde

- x Representa el nodo desde el cual se mueve la hormiga.
- y Representa el nodo hacia el cual se mueve la hormiga.
- τ_y Representa el valor de feromona acumulado en el nodo y.
- η_y Representa el valor devuelto por la función de evaluación en el nodo y.
- α Determina la importancia de τ_y. Su valor teórico habitual es $\alpha \geq 0$.
- β Determina la importancia de η_y. Su valor teórico habitual es $\beta \geq 1$.

Como se ha podido observar, la fórmula empleada no es la misma que la vista en la sección 2. Mientras que en la fórmula original se tiene en cuenta el valor de feromonas y función de evaluación en las aristas que unen los nodos, en este caso se opta por considerar estos valores en los nodos hacia los que se puede dirigir la hormiga. Esta adaptación es resultado de la experiencia obtenida en la realización del trabajo.

A continuación se procede a detallar los aspectos más importantes de las feromonas y la función de evaluación implementada.

- Feromonas (τ_y). Tras encontrar una primera solución al problema planteado, las feromonas cobran gran importancia facilitando la obtención de nuevas soluciones. No obstante, un uso abusivo de feromonas puede hacer que el algoritmo converja rápidamente a una solución no óptima e impida obtener otra solución mejor. Esto es debido a que si el valor de feromona es muy alto, la función de evaluación pierde importancia, por lo que una vez que se ha encontrado una solución será difícil para una hormiga desviarse del camino marcado por las feromonas. Para evitar este comportamiento, el valor de las feromonas y el parámetro α deben ser establecidos a un valor adecuado.

  El valor de feromona utilizado se calcula como

$$\tau_{y \in solucion} = \frac{(distancia_{\,inicio-fin})^f}{longitud\ de\ la\ solucion} \tag{3.2}$$

  donde f es un factor que permite nivelar el valor de las feromonas respecto al devuelto por la función de evaluación. Es importante que estos valores sean consecuentes entre sí. Si el valor de feromona es excesivamente alto en comparación con el de la función de evaluación, este último pierde importancia, y viceversa. Además, al igual que el parámetro α de la ecuación 3.1, el factor f afecta directamente a la convergencia del algoritmo: un valor alto hará converger rápidamente el algoritmo, mientras que un valor bajo puede hacer que no converja nunca.

- Función de evaluación (η_y). Como ya se ha mencionado, el objetivo de la función de evaluación es agilizar la obtención de una solución. Se desea evitar que el comportamiento de las hormigas sea muy aleatorio, ya que dificultaría la obtención de una solución, o demasiado guiado, ya que podría impedir el avance de las hormigas.

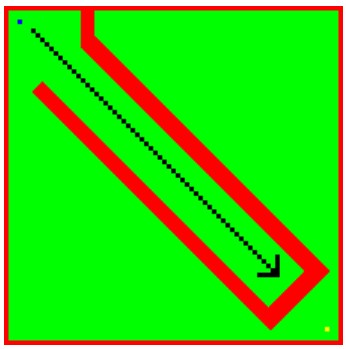


Figura 3.12: Comportamiento demasiado guiado de las hormigas

Inicialmente, se pensó en una heurística sencilla basada en la distancia euclídea desde la posición inicial del escenario hasta la posición que ocupaba el nodo al cual podía moverse la hormiga. De esta forma, los nodos más alejados de la posición inicial eran mejor evaluados. No obstante, esto presentó varios problemas. En primer lugar, para escenarios de cierta dimensión, la diferencia entre viajar a un nodo o a otro es ínfima, por lo que la función de evaluación tiende a aleatorizar el movimiento de las hormigas. Además, la suposición de que cuanto más alejado se esté de la posición inicial más cerca se estará de la final es errónea y únicamente válida para escenarios donde la distancia entre la posición inicial y final es la mayor entre dos puntos cualesquiera del escenario.

Aunque la idea inicial fue errónea, permitió que con un pensamiento similar se hallase una mejor solución. Esta solución se basó en la idea de que cuanto más cerca se esté de la posición final, mejor se debe puntuar dicho nodo. Además, era necesario diferenciar de alguna manera la posibilidad de ir a un nodo o a otro, dado que la diferencia entre distancias era muy pequeña. La heurística que finalmente se implementó tiene parte de este pensamiento con algunas modificaciones que permiten agilizar la búsqueda de soluciones. Esta heurística se calcula como

$$\eta_y = factor\,de\,cercania_y * factor\,de\,evasion_y * factor\,de\,impulso_y \tag{3.3}$$

* Factor de cercanía al objetivo. Los nodos más cercanos a la posición final del escenario tendrán mayor probabilidad de ser elegidos. Este factor se calcula como

$$factor\,de\,cercania_y = \frac{distancia_{\,inicio-fin} + 1}{distancia_{\,y-fin} + 1} * (1 - k) \tag{3.4}$$

  donde k representa un valor que aumenta según lo hace la distancia a la posición final del escenario. Esto permite realizar una mayor diferenciación entre los nodos accesibles para la hormiga.

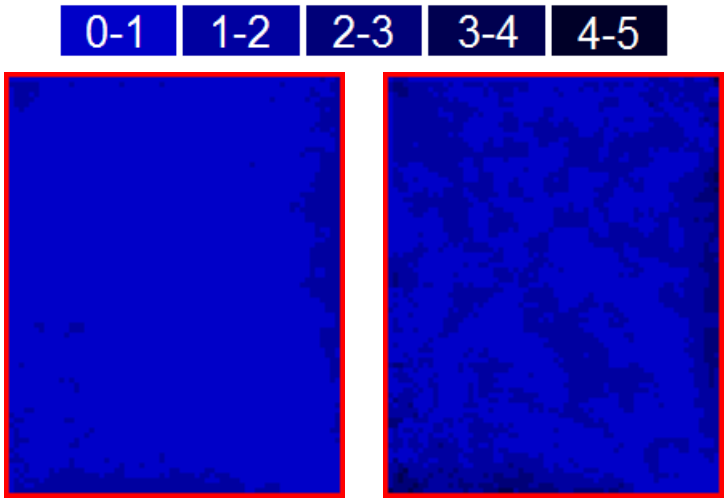
  El factor de cercanía tiene poco peso en la decisión de movimiento de la hormiga inicialmente, no obstante, según se acerca a la posición final los valores tienden a aumentar. De esta forma, la hormiga se dejará guiar por las feromonas al comienzo de su camino, pero tenderá a poder mejorar dicho camino gracias a los valores en incremento de la función de evaluación.

* Factor de evasión de nodos ya visitados. Para evitar que la hormiga deambule indefinidamente por los mismos nodos, la probabilidad de que vuelva a moverse hacia un nodo ya visitado disminuye con cada pasada de la hormiga por el mismo. De esta forma, este factor se calcula como

$$factor\,de\,evasion_y = \frac{1}{frecuencia_y} \tag{3.5}$$

donde $frecuencia_y$ representa el número de veces que la hormiga ha pasado por el nodo y. Si esta frecuencia es 0, no se tiene en cuenta este factor.

* Factor de impulso. Para evitar que la hormiga vuelva a moverse hacia el nodo del cual viene, se penaliza fuertemente el regreso a la posición anterior. En concreto, este factor disminuye el valor de la función de evaluación en un 80 %.



(a) Con factor de evasión (b) Sin factor de evasión

Figura 3.13: Frecuencia de movimiento de las hormigas con y sin factor de evasión, respectivamente. Como se puede observar, con el factor de evasión el movimiento hacia nodos ya visitados disminuye considerablemente. Por otro lado, si no se considera el factor de evasión, las frecuencias de movimiento hacia nodos ya visitados tienden a aumentar y se distribuyen de la misma forma en que lo hace la acumulación de feromonas en los caminos hacia la fuente de alimento.

Adaptación de la colonia de hormigas como función de *fitness*

Como se vio en la sección 2, la función de *fitness* es la encargada de evaluar la calidad de los diferentes individuos de la población con el fin de facilitar la posterior selección de los mismos. Gracias a esta función, el GA converge a un tipo de individuos cada vez más parecidos. En este caso, se trata de adaptar la funcionalidad del ACO a la función de *fitness*.

Como ya se ha mencionado, mediante el ACO implementado se obtienen soluciones a escenarios. Gracias a esto, es posible asignar a cada escenario un valor de *fitness* basado en la calidad de la mejor solución obtenida (entendiendo como mejor solución la que representa el camino más corto entre el nodo inicial y el nodo final).

$$fitness_{escenario} = longitud\,de\,mejor\,solucion_{escenario} * proporcion\,de\,pared_{escenario} \tag{3.6}$$

donde

- Longitud de mejor solución. La longitud de la mejor solución de un escenario medida en número de casillas.
- Proporción de pared. El número de casillas que son secciones de pared entre el número de casillas que son secciones de suelo.

Por lo tanto, esta función de *fitness* asigna valores a los individuos basándose en la longitud de la solución y la concentración de secciones de pared respecto a secciones de suelo. Esto permite ordenarlos de manera descendente para poder seleccionar los que convengan en cada situación.

NOTA: Cabe mencionar que en los escenarios generados mediante el algoritmo de generación de caminos se toma la proporción de pared con un valor de 1. Esto es debido a que en dicho algoritmo, por lo general, los escenarios con soluciones más cortas son también los escenarios con mayor proporción de secciones de pared, lo cual acaba falseando los resultados y dando lugar a escenarios poco adecuados al nivel de dificultad seleccionado.

Selección de individuos

La selección de individuos se realiza obteniendo aquellos individuos que sean más adecuados para el nivel de dificultad requerido por el usuario. En concreto, se distinguen los siguientes niveles de dificultad:

- Nivel de dificultad fácil. Se escogen los dos individuos con menor valor de *fitness*.
- Nivel de dificultad medio. Se escogen los dos individuos que estén en la mitad del *ranking* de valores de *fitness*.
- Nivel de dificultad difícil. Se escogen los dos individuos con mayor valor de *fitness*.

NOTA: Si se añadiese cualquier otro nivel de dificultad se podrían obtener los individuos apropiados a ese nivel mediante una selección proporcional al número de niveles de dificultad.

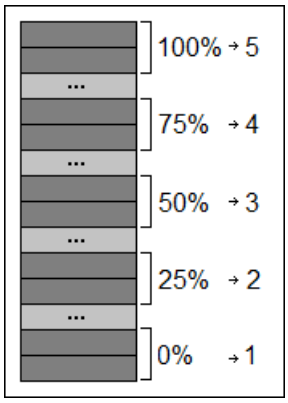


Figura 3.14: Selección proporcional de individuos con cinco niveles de dificultad

Dado que los individuos mejor puntuados son los escenarios con soluciones más largas y más proporción de pared, un nivel de dificultad difícil tenderá a generar escenarios grandes, con soluciones largas y una alta concentración de secciones de pared. Por otro lado, un nivel de

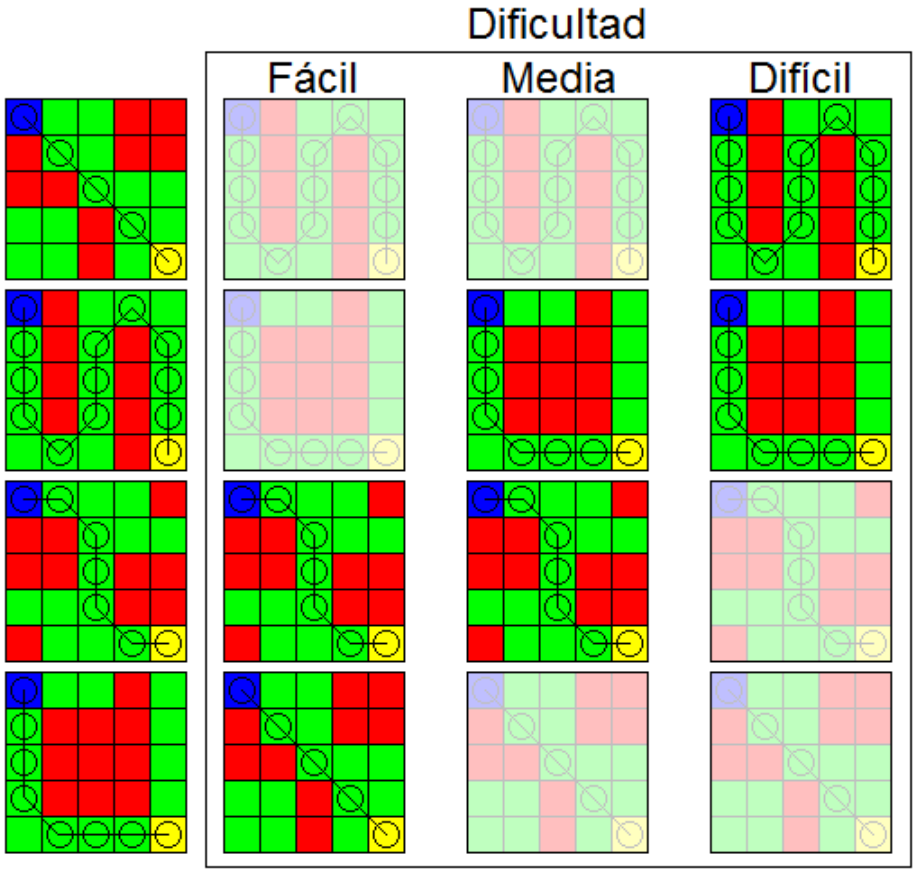


Figura 3.15: Ejemplo de la función de *fitness* implementada y su repercusión en la selección de individuos de la población

dificultad fácil generará escenarios pequeños, con soluciones cortas y pocas secciones de pared.

4

Resultados experimentales

En esta sección se detallarán las diferentes pruebas realizadas y las conclusiones derivadas de las mismas. Estas pruebas tienen el objetivo de mostrar cómo funciona el GA visto en la sección anterior y su potencial a la hora de generar escenarios aleatorios ajustados a un nivel de dificultad concreto. Para ello, inicialmente se profundizará en la colonia de hormigas implementada y en la generación de la población inicial. Posteriormente, se validará el comportamiento de los operadores del GA. Por último, se mostrarán los resultados obtenidos al evolucionar escenarios mediante el GA en forma de mapas bidimensionales y su representación en un entorno tridimensional.

4.1. Algoritmo de colonia de hormigas

Estas pruebas permiten determinar qué parámetros aumentan la probabilidad de generar una población inicial de escenarios con solución, disminuyen el tiempo de ejecución empleado en la búsqueda de soluciones y aumentan la calidad de las mismas.

4.1.1. Generación de escenarios con solución

Como se ha visto en la sección 2, cada algoritmo de generación de escenarios tiene sus propios parámetros y características. Las siguientes pruebas muestran qué parametrización se debe aplicar a cada algoritmo para aumentar considerablemente la probabilidad de generar una población inicial de escenarios con solución. En la realización de las diferentes simulaciones del ACO se han empleado 100 hormigas durante 10000 *steps* con parámetros $\alpha = 0{,}75$ y $\beta = 1$. Más adelante se verá que esta configuración no es óptima, no obstante, el objetivo de estas pruebas no es comprobar el número de soluciones obtenidas sino el comportamiento de los diferentes algoritmos de generación aleatoria. Cabe mencionar, que las gráficas mostradas son resultado de la media aritmética de 25 simulaciones cada una.

Algoritmo de generación de paredes

En la figura 4.1 se muestra cómo la proporción de pared que especifiquemos afecta directamente al número de soluciones encontradas. Como se puede observar, el número de soluciones encontradas

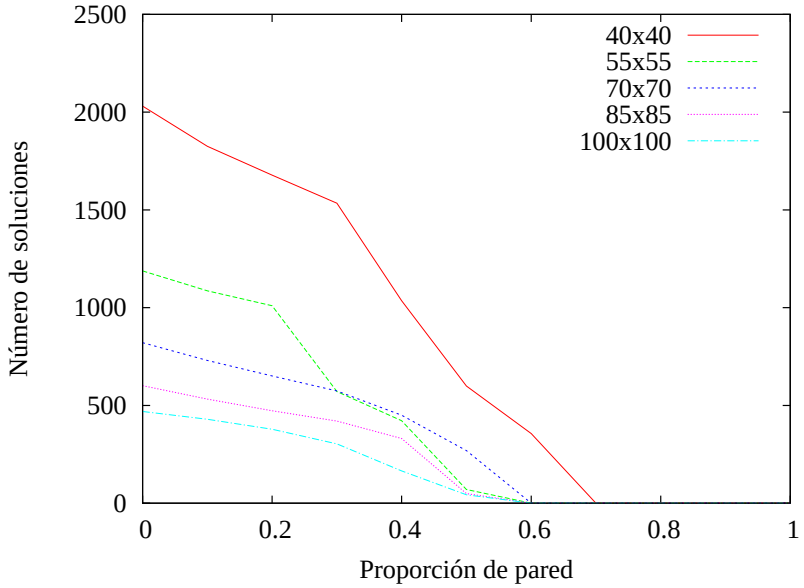


Figura 4.1: Generación de escenarios con solución (I). Número de soluciones encontradas para diferentes parametrizaciones del algoritmo de generación de paredes y escenarios de dimensiones variadas.

desciende notablemente según aumenta la dimensión del escenario. Esto se debe a que la distancia recorrida por las hormigas hasta encontrar una solución es mayor, por lo que el número de soluciones que encuentran en el mismo tiempo es menor. De la misma forma, el número de soluciones también desciende según aumenta la proporción de pared del escenario. En concreto, para la mayoría de los casos con un valor de 0.6 el número de soluciones tenderá a valer 0. Para los casos estudiados se ha establecido una proporción de pared de 0.5 como máximo.

Algoritmo de generación de caminos

En la figura 4.2 se muestra cómo el número de caminos y tramos que especifiquemos afectan directamente al número de soluciones encontradas. En la realización de esta prueba se ha tomado una anchura de 5 unidades para los caminos y tramos que se generan.

Al igual que en el algoritmo de generación de paredes, la dimensión de los escenarios afecta al número de soluciones encontradas. Por otro lado, se observa que una vez generado un número mínimo de caminos es indiferente si se continúan generando más caminos o no, dado que el número de soluciones se mantiene prácticamente estable. Este comportamiento se debe a que cuando hay pocos caminos la distancia entre la posición inicial y final del escenario es similar a cuando hay una gran cantidad de ellos. La diferencia radica en que cuando hay pocos caminos la solución no se obtiene de manera directa, ya que hay gran cantidad de secciones de pared que lo impiden, mientras que cuantos más caminos hay más despejada de secciones de pared se encuentra la ruta hacia el objetivo, por lo que más directa es la solución. Dado que la distancia es similar, el número de soluciones encontradas en el mismo tiempo también lo es. No obstante, se ha de puntualizar que, a pesar de que no se obtienen más soluciones, si se desea generar una población de escenarios variada mediante este algoritmo, el número de caminos deberá ser igualmente variado, ya que, aunque no afecte en el número de soluciones encontradas, si lo hace en la dificultad y calidad de las mismas.

Cabe mencionar que, si el número de caminos es demasiado bajo, se puede dar el caso en

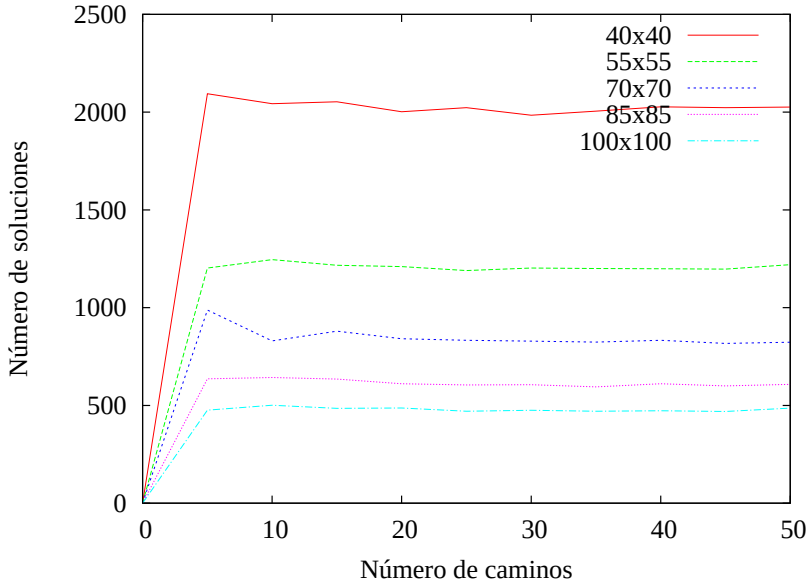


Figura 4.2: Generación de escenarios con solución (II). Número de soluciones encontradas para diferentes parametrizaciones del algoritmo de generación de caminos y escenarios de dimensiones variadas.

el que el número de soluciones aumente enormemente debido a la proximidad de las posiciones de inicio y fin.

Algoritmo de generación de *clusters*

En la figura 4.3 se muestra cómo el número de *clusters* que especifiquemos afecta directamente al número de soluciones encontradas (cuantos más *clusters* se generen menos soluciones se encontrarán). Por otro lado, como en los algoritmos anteriores, el número de soluciones disminuye según aumenta la dimensión de los escenarios.

Cabe destacar aquellos casos poco habituales en los que el número de soluciones aumenta de manera exagerada o disminuye hasta valer cero. El primer caso se puede llegar a dar cuando el número de *clusters* es lo suficientemente alto como para que las posiciones de inicio y fin se generen muy próximas entre sí. El segundo caso es algo más habitual y, al igual que el primer caso, ocurre al establecer un número de *clusters* demasiado alto, originando escenarios sin solución. En los casos estudiados se ajusta dicho parámetro como un valor proporcional a la dimensión del escenario, lo que habitualmente da lugar a escenarios con solución.

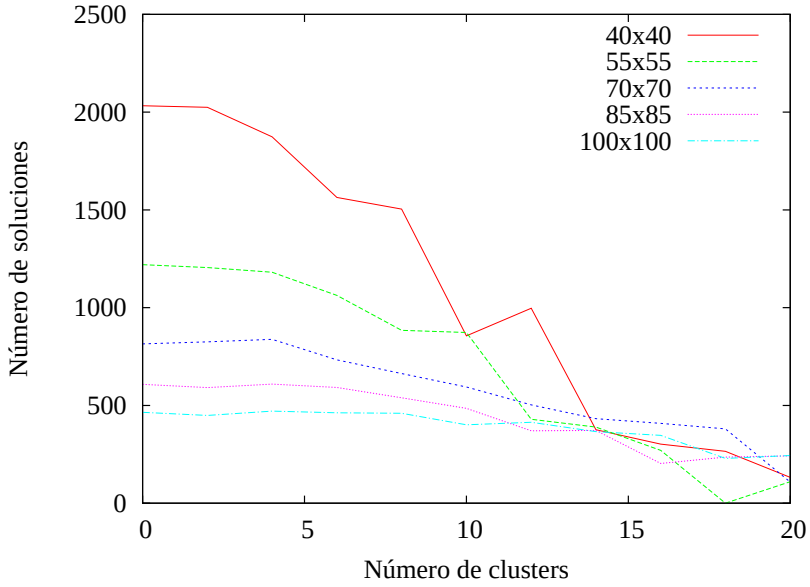


Figura 4.3: Generación de escenarios con solución (III). Número de soluciones encontradas para diferentes parametrizaciones del algoritmo de generación de *clusters* y escenarios de dimensiones variadas.

4.1.2. Configuración del algoritmo de colonia de hormigas

A continuación, se estudian los diferentes parámetros que afectan al ACO en la obtención de soluciones en escenarios de diferentes dimensiones y se muestra la evolución del algoritmo respecto al número de *steps* realizados.

Parámetros α y β

Como ya se ha visto, los parámetros α y β determinan la importancia de las feromonas y la función de evaluación, respectivamente. La prueba que se muestra a continuación determina qué combinación de estos parámetros es más adecuada de cara a la búsqueda de soluciones. Cabe mencionar que las simulaciones han sido lanzadas con 100 hormigas durante 10000 *steps*.

Tal y como se ha visto, se podrían obtener mejores soluciones con valores de α y β más altos, no obstante, la diferencia en la calidad de solución entre el valor comentado y otro valor superior no es significativo y, sin embargo, sí lo es el tiempo de ejecución empleado [1].

A continuación se muestra la calidad de las mejores soluciones obtenidas mediante las mejores y peores combinaciones de alfa y beta. En las mejores combinaciones la solución tiene un trazo fino desde el punto de inicio hasta el punto final, lo que implica que la hormiga que encontró dicha solución lo hizo de manera directa. Por otro lado, en las peores combinaciones se observa un movimiento mucho más errático, provocado por el poco peso que tiene la función de evaluación en la fase de exploración inicial y las feromonas en la fase de optimización de soluciones.

[1] El mayor valor probado es de $\alpha = 25$ y $\beta = 25$. Con esta combinación, la simulación tarda una cantidad considerable de tiempo, en comparación con otros valores inferiores, y el resultado obtenido no es significativamente mejor que $\alpha = 3{,}75$ y $\beta = 4$.

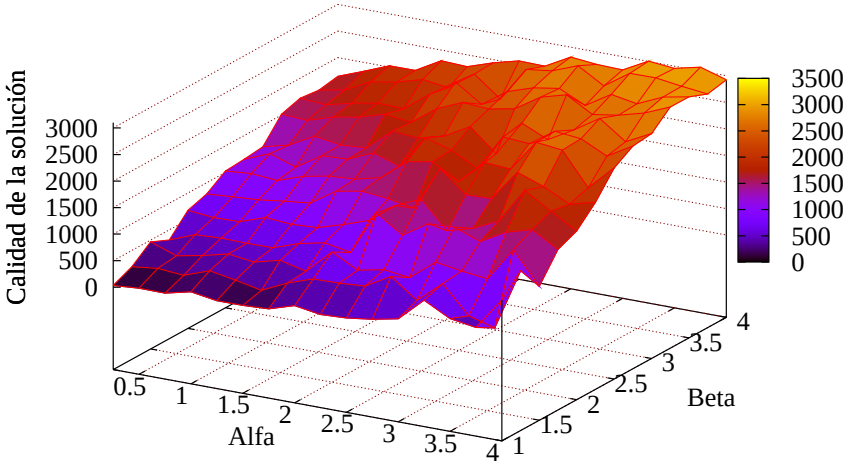


Figura 4.4: Como se puede observar, se obtienen mejores soluciones con valores de α y β más altos. Por otro lado, unos valores altos implican operaciones más costosas computacionalmente. Una buena combinación de calidad/coste de estos valores es $\alpha = 3{,}75$ y $\beta = 4$.

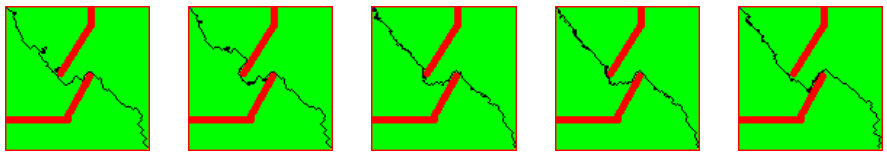


Figura 4.5: Resultado obtenido de las mejores combinaciones de los parámetros alfa y beta.

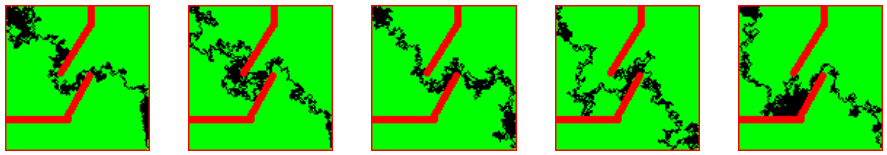


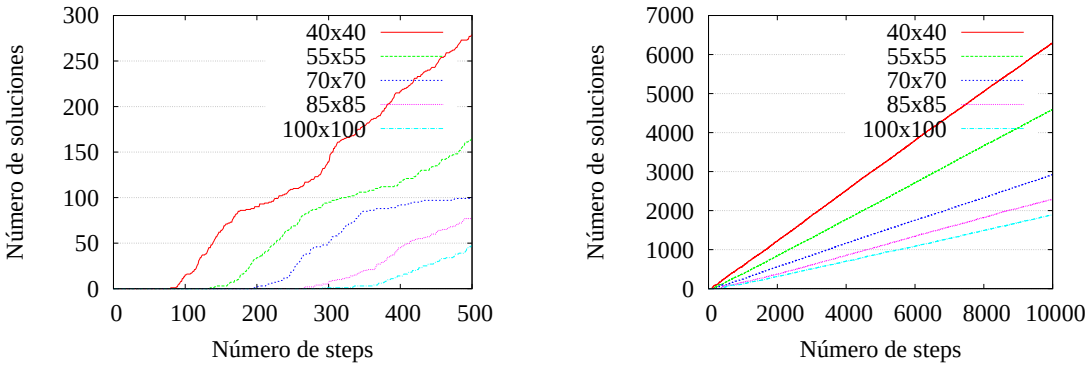
Figura 4.6: Resultado obtenido de las peores combinaciones de los parámetros alfa y beta.

Se puede observar que en esta prueba se ha empleado un escenario en donde el camino directo hacia el objetivo se ve dificultado por secciones de pared. Además, si no se equilibran correctamente los valores de α y β, puede derivar en que las hormigas deambulen aleatoriamente o queden atrapadas durante un tiempo en las zonas con "picos"del escenario, lo cual produciría malas soluciones. Es por ello, que los valores calculados en esta prueba resultan sólidos.

Obtención de soluciones

Es importante determinar el número de *steps* necesarios para comenzar a obtener soluciones y a partir de qué momento el número de soluciones obtenidas por *step* se mantiene constante.

Gracias a esto podremos ajustar la duración de la simulación. En las simulaciones de esta prueba se han utilizado 100 hormigas durante 10000 *steps* con $\alpha = 3{,}75$ y $\beta = 4$.



(a) Visión de los primeros 500 *steps*

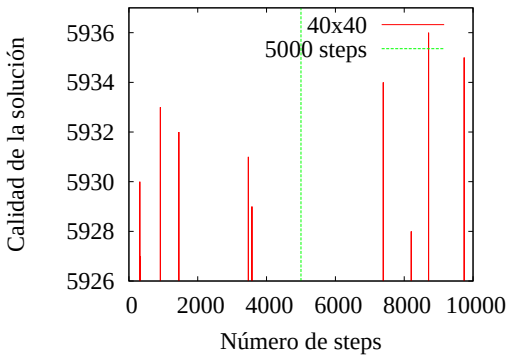
(b) Visión general

Figura 4.7: Como se puede observar en la figura 4.7(a), cuanto mayor sea la dimensión del escenario mayor número de *steps* serán necesarios para comenzar a obtener soluciones. Por otro lado, en la figura 4.7(b), se proporciona una visión general de las simulaciones realizadas, en donde se demuestra que a mayor dimensión menor número de soluciones se encontrarán en total.

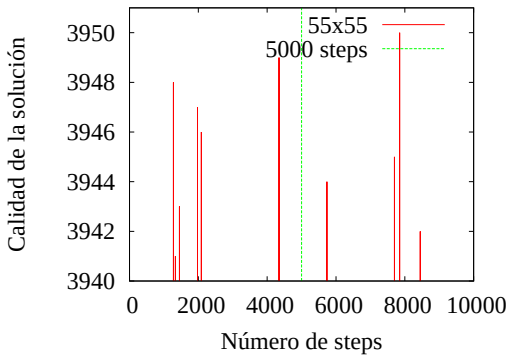
Calidad de las soluciones

No solo es importante tener en cuenta a partir de qué momento se obtienen las primeras soluciones. Mediante las siguientes gráficas se pretende mostrar qué número aproximado de *steps* es aceptable para obtener las mejores soluciones de un escenario. Para ello se muestra el momento en el que se obtuvieron las diez mejores soluciones y se marca con una línea vertical verde el momento en el que se alcanzan los 5000 *steps*. Como se puede observar, generalmente la mayor parte de las mejores soluciones ocurren antes de los 5000 *steps*. Esto es debido, principalmente, a que la configuración del ACO utilizada permite que el rastro de feromonas se fortalezca rápidamente y, al alcanzar dicho *step*, sea suficientemente intenso como para que se hayan obtenido algunas de las mejores soluciones. Las simulaciones se han realizado con 100 hormigas durante 10000 *steps* con $\alpha = 3{,}75$ y $\beta = 4$.

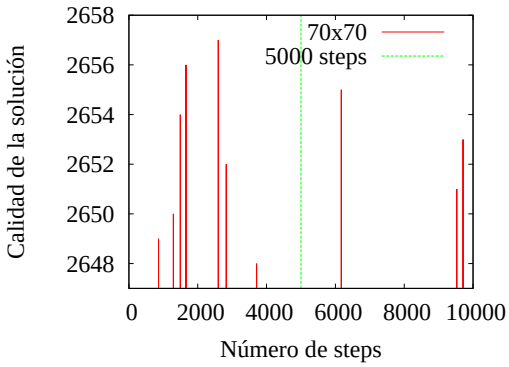
NOTA: Las soluciones obtenidas mediante el ACO son evaluadas según su longitud (medida en número de casillas). A partir de ahí, se determina la calidad de la solución como la diferencia entre la longitud de la peor solución y la longitud de la solución a evaluar.



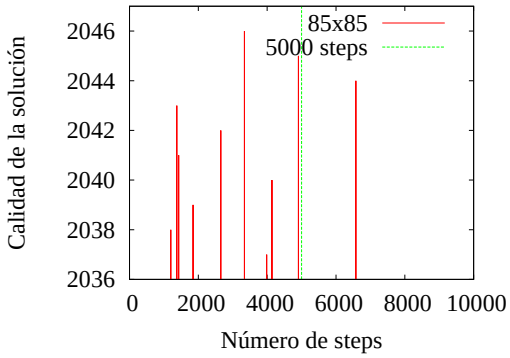
(a) 40x40



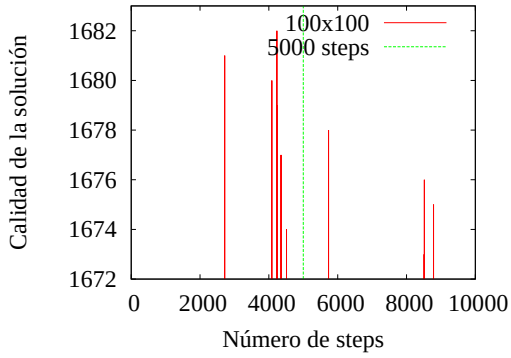
(b) 55x55



(c) 70x70



(d) 85x85



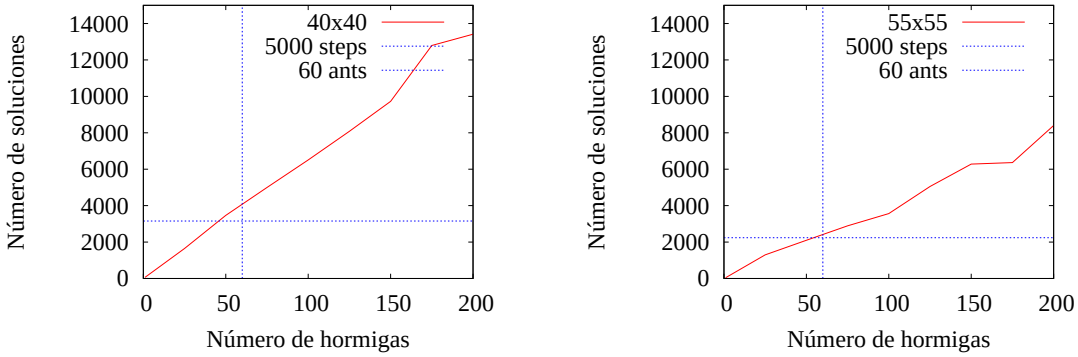
(e) 100x100

Figura 4.8: En las gráficas se pueden observar los *steps* en los que se obtuvieron las diez mejores soluciones en escenarios de diferentes dimensiones.

Número de hormigas

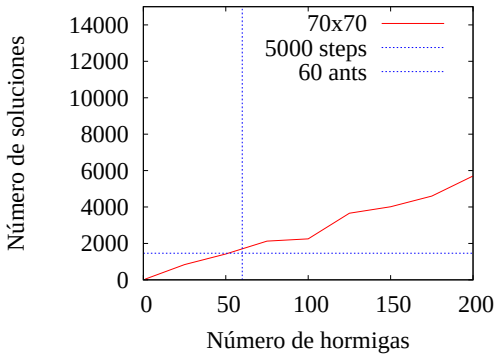
El número de hormigas que se utilizan en las simulaciones también es importante de cara a la rapidez con la que se obtienen las mejores soluciones, ya que un número mayor de hormigas explorarán más terreno en menos tiempo y dejarán un mayor número de feromonas en total. Por otro lado, cuantas más hormigas se utilicen en una simulación más cantidad de recursos y tiempo de ejecución se necesitarán para una tarea que quizá con un número menor de hormigas podría haberse realizado de manera más eficiente. El objetivo de esta prueba es determinar qué número aproximado de hormigas es suficiente para obtener las mejores soluciones de un

escenario. Estas simulaciones se han lanzado a lo largo de 10000 *steps* con $\alpha = 3{,}75$ y $\beta = 4$.

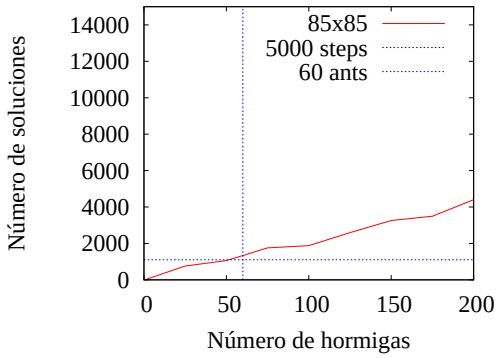


(a) 40x40

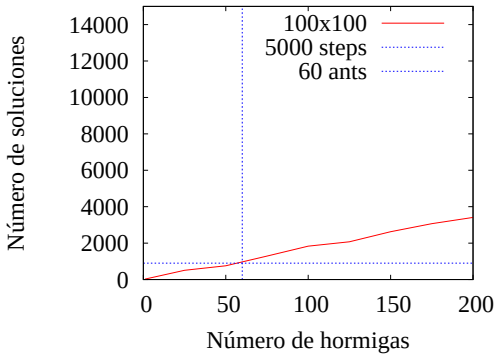
(b) 55x55



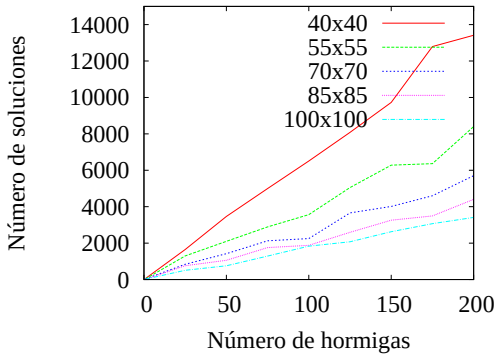
(c) 70x70



(d) 85x85



(e) 100x100



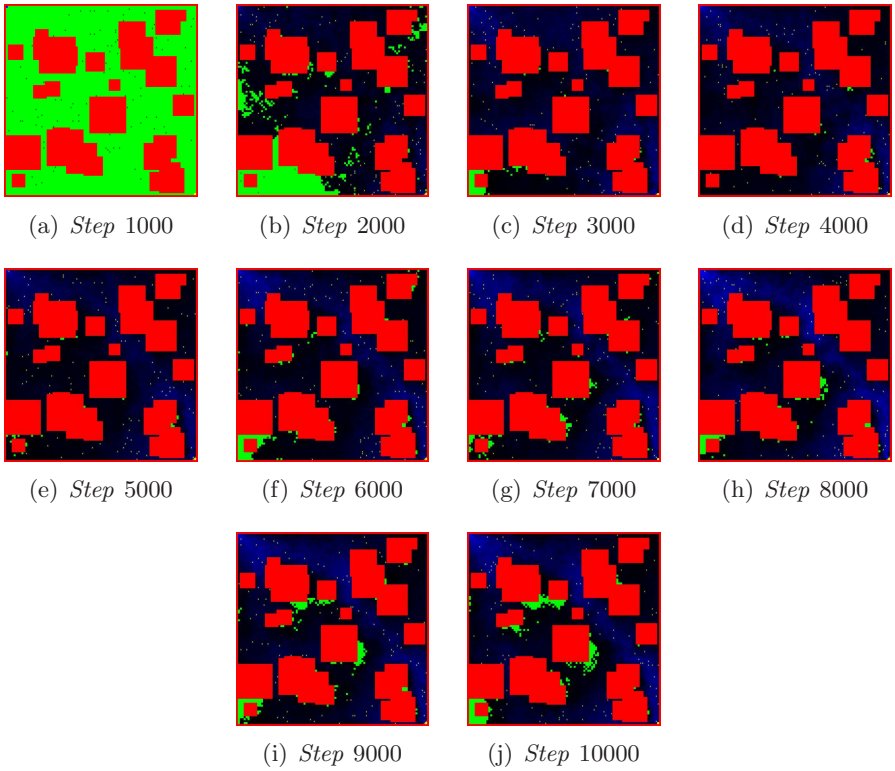
(f) 85x85

Figura 4.9: La línea vertical azul representa la utilización de 60 hormigas, mientras que la línea horizontal azul representa el número de soluciones obtenidas para dicha dimensión a los 5000 *steps*, según los datos obtenidos de la figura 4.7. Como se puede observar, las diferentes curvas cortan con la línea horizontal siempre a la izquierda de la línea vertical. Esto quiere decir que, para las dimensiones de escenarios estudiadas, 60 hormigas son suficientes para encontrar las mejores soluciones, dado que éstas se obtenían en los primeros 5000 *steps*. Por otro lado, en la figura 4.9(f), se ofrece una visión comparativa general del número de soluciones obtenidas por número de hormigas para diferentes dimensiones.

Evolución del algoritmo

A continuación se muestra la evolución del algoritmo de optimización mediante colonia de hormigas en un escenario de tipo *clusters* con el fin de visualizar cómo influyen la función de evaluación y, sobre todo, las feromonas en la búsqueda de las mejores soluciones. Para ello se han realizado dos simulaciones que pretenden verificar de nuevo la importancia de los parámetros α y β. En la primera simulación, se lanzan 100 hormigas durante 10000 *steps* con $\alpha = 0{,}75$ y $\beta = 1$. En la segunda simulación, se modifican los valores α y β, que pasarán a valer 3.75 y 4, respectivamente.

NOTA: En las siguientes figuras se representa el rastro de feromonas dejado por las hormigas mediante diferentes tonalidades de azul. Un color oscuro indica que el rastro de feromonas es débil, mientras que un color claro representa un rastro de feromonas intenso.



(a) *Step* 1000 (b) *Step* 2000 (c) *Step* 3000 (d) *Step* 4000

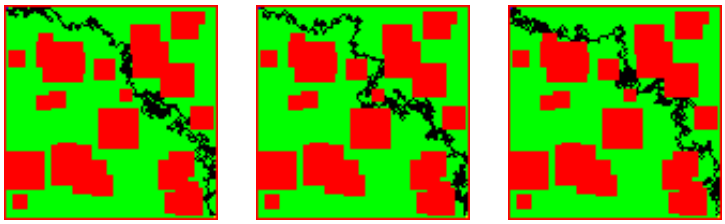
(e) *Step* 5000 (f) *Step* 6000 (g) *Step* 7000 (h) *Step* 8000

(i) *Step* 9000 (j) *Step* 10000

Figura 4.10: Evolución del algoritmo de optimización mediante colonia de hormigas con valores $\alpha = 0{,}75$ y $\beta = 1$. Como se puede observar, el camino más frecuentado por las hormigas es visible a partir del *step* 5000 y se refuerza a partir del *step* 8000.

Como se ha visto, el parámetro α permite a las hormigas seguir el rastro de las mejores soluciones, sin embargo, dado que su valor no es alto, el rastro de feromonas se ve difuminado y no tiene el poder de influencia que debería. Por otro lado, el parámetro β seleccionado permite obtener soluciones, no obstante, dado que su valor es bajo, hace que la función de evaluación pierda importancia. Esto causa que se tarde más tiempo en encontrar las primeras soluciones y, gráficamente, las mejores soluciones presenten un trazo más grueso (debido a la conducta errática de las hormigas, como se comentó previamente).

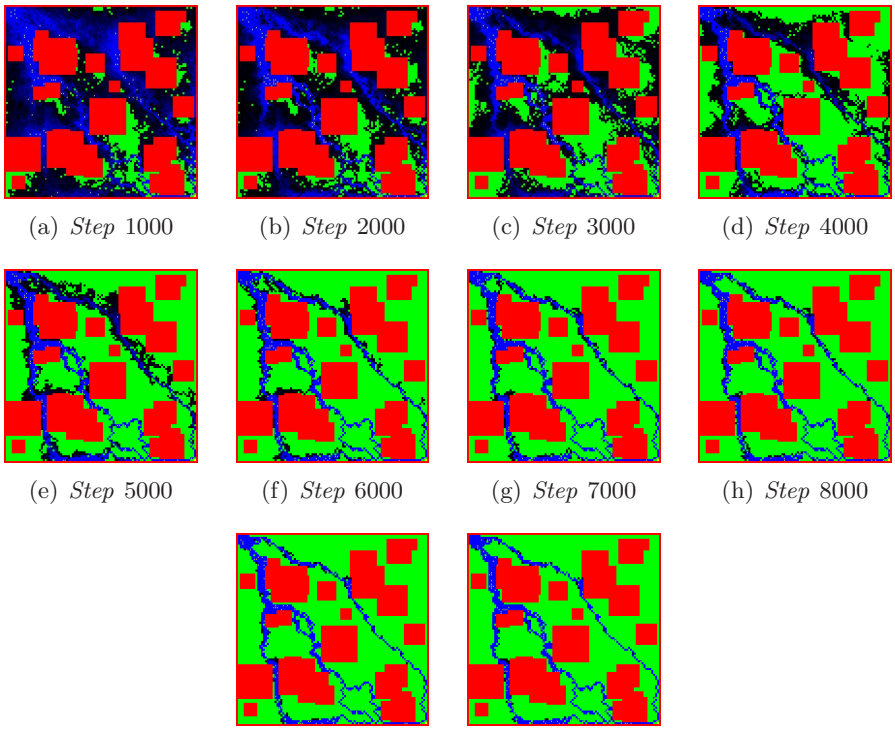
En la siguiente figura se muestran las mejores soluciones obtenidas mediante esta combinación de α y β.



(a) Solución 1 (b) Solución 2 (c) Solución 3

Figura 4.11: Como ya se ha mencionado, las soluciones presentan una trayectoria poco directa y, consecuentemente, un trazo grueso.

A continuación se muestran los resultados obtenidos en la segunda simulación.



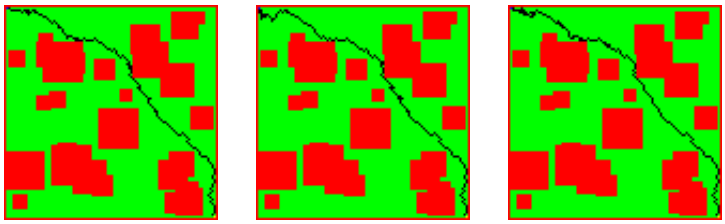
(a) *Step* 1000 (b) *Step* 2000 (c) *Step* 3000 (d) *Step* 4000

(e) *Step* 5000 (f) *Step* 6000 (g) *Step* 7000 (h) *Step* 8000

(i) *Step* 9000 (j) *Step* 10000

Figura 4.12: Evolución del algoritmo de optimización mediante colonia de hormigas con valores $\alpha = 3{,}75$ y $\beta = 4$. A partir del *step* 1000 se comienzan a obtener las primeras soluciones. Entre los *steps* 2000 y 4000 se marcan los mejores caminos. Posteriormente, se evaporan las feromonas de los caminos menos frecuentados. Finalmente, se visualizan de forma clara los mejores caminos hacia el objetivo.

Como se ha visto, en esta ocasión los valores de α y β escogidos permiten obtener soluciones bastante precisas. La siguiente figura muestra las mejores soluciones obtenidas mediante esta combinación de α y β.



(a) Solución 1 (b) Solución 2 (c) Solución 3

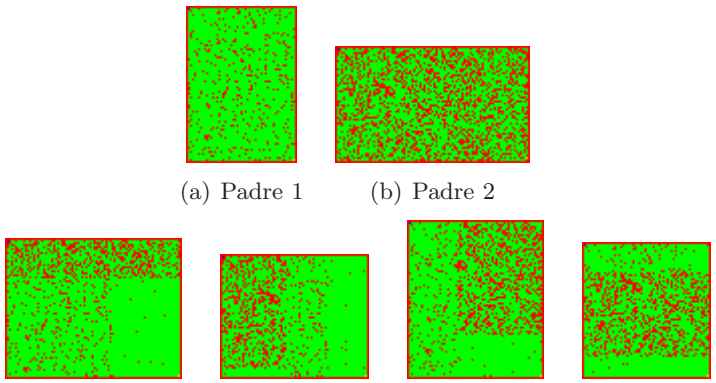
Figura 4.13: Se muestran soluciones directas al objetivo y de trazo fino.

4.2. Operadores del algoritmo genético

Las pruebas realizadas en este apartado permiten validar el funcionamiento de los operadores de cruzamiento y mutación del GA mediante la combinación de escenarios de tipos y dimensiones variadas.

Combinación de escenarios tipo paredes

Como se vio en la sección 3, el sucesor obtenido de la combinación de dos escenarios de tipo paredes es otro escenario de tipo paredes. Este sucesor se caracteriza por formar nubes de puntos. Esto da lugar a escenarios distinguibles principalmente por su densidad.



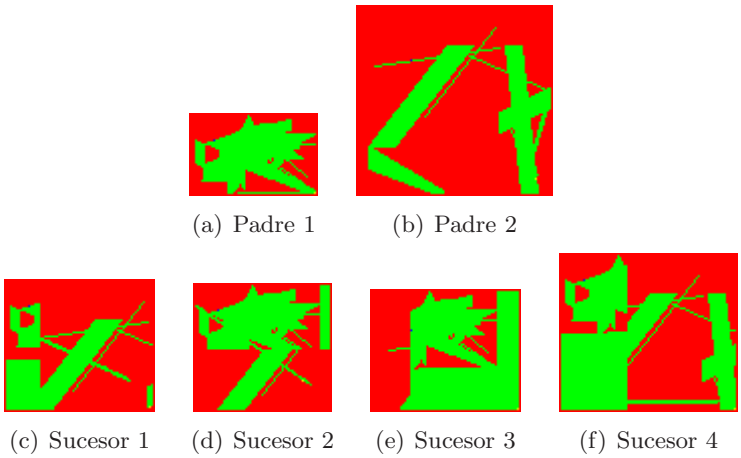
(a) Padre 1 (b) Padre 2

(c) Sucesor 1 (d) Sucesor 2 (e) Sucesor 3 (f) Sucesor 4

Figura 4.14: Generación de sucesores a partir de dos escenarios de tipo paredes.

Combinación de escenarios tipo caminos

Como se vio en la sección 3, el sucesor obtenido de la combinación de dos escenarios de tipo caminos es otro escenario de tipo caminos. Cabe mencionar que muchos escenarios generados a través de esta combinación no tienen solución.



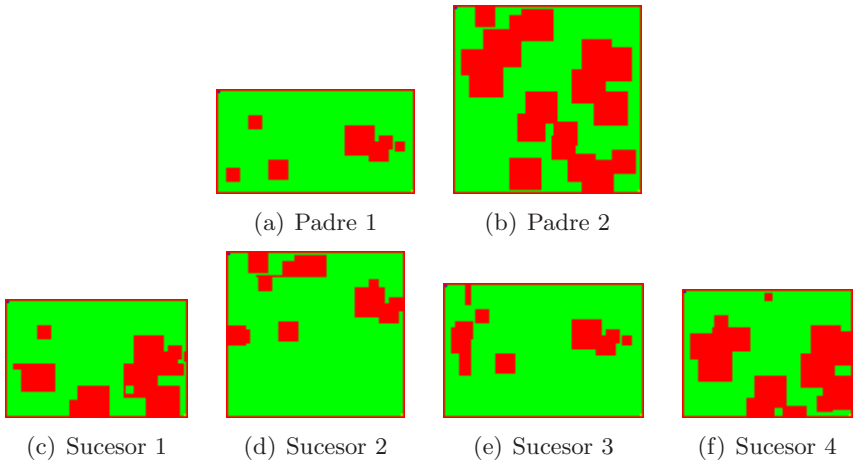
(a) Padre 1 (b) Padre 2

(c) Sucesor 1 (d) Sucesor 2 (e) Sucesor 3 (f) Sucesor 4

Figura 4.15: Generación de sucesores a partir de dos escenarios de tipo caminos.

Combinación de escenarios tipo *clusters*

Como se vio en la sección 3, el sucesor obtenido de la combinación de dos escenarios de tipo *clusters* es otro escenario de tipo *clusters*. Este tipo de sucesores se caracterizan por su diversidad.



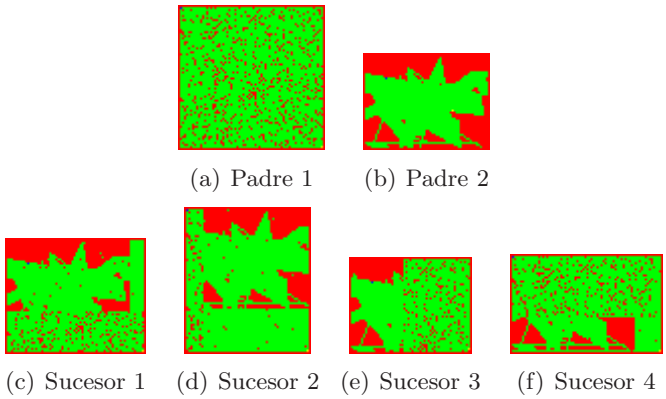
(a) Padre 1 (b) Padre 2

(c) Sucesor 1 (d) Sucesor 2 (e) Sucesor 3 (f) Sucesor 4

Figura 4.16: Generación de sucesores a partir de dos escenarios de tipo *clusters*.

Combinación de escenarios tipo paredes con tipo caminos

El sucesor tiene una probabilidad de un 50 % de ser de tipo paredes o caminos. De esta combinación resultan escenarios variados y con una alta tasa de solución, en donde se pasa de una zona de escenario abierto (tipo caminos) a una zona con gran cantidad de obstáculos (tipo paredes).



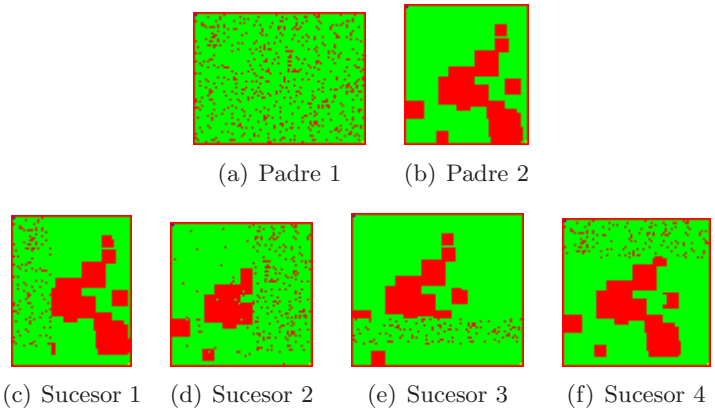
(a) Padre 1 (b) Padre 2

(c) Sucesor 1 (d) Sucesor 2 (e) Sucesor 3 (f) Sucesor 4

Figura 4.17: Generación de sucesores a partir de un escenario tipo paredes y otro escenario tipo caminos.

Combinación de escenarios tipo paredes con tipo *clusters*

El sucesor tiene una probabilidad de un 50 % de ser de tipo paredes o *clusters*. Al igual que en el caso anterior, resultan escenarios variados, con alta tasa de solución y con zonas visualmente diferenciables.



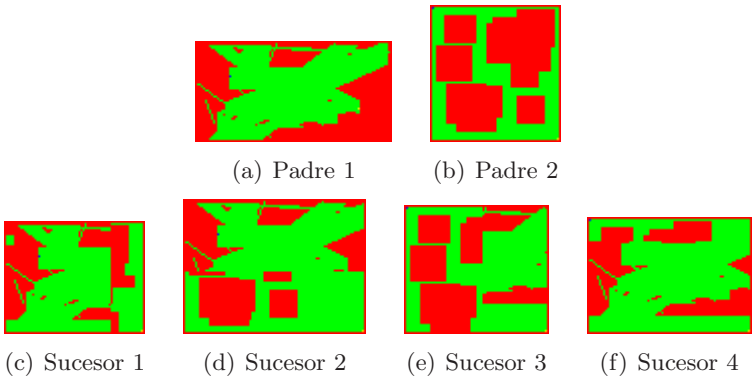
(a) Padre 1 (b) Padre 2

(c) Sucesor 1 (d) Sucesor 2 (e) Sucesor 3 (f) Sucesor 4

Figura 4.18: Generación de sucesores a partir de un escenario tipo paredes y otro escenario tipo *clusters*.

Combinación de escenarios tipo caminos con tipo *clusters*

El sucesor tiene una probabilidad de un 50 % de ser de tipo caminos o clusters. También en este caso resultan escenarios variados, no obstante, en ciertas ocasiones se producen escenarios sin solución.



(a) Padre 1 (b) Padre 2

(c) Sucesor 1 (d) Sucesor 2 (e) Sucesor 3 (f) Sucesor 4

Figura 4.19: Generación de sucesores a partir de un escenario tipo caminos y otro escenario tipo *clusters*.

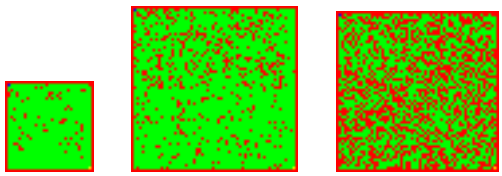
4.3. Generación de escenarios aleatorios

Como ya se ha explicado en secciones anteriores, mediante el algoritmo genético es posible generar escenarios aleatorios ajustado a un nivel de dificultad. Las pruebas que se realizan a continuación tratan de mostrar los resultados obtenidos al generar escenarios de diferentes tipos y para varias dificultades. Las simulaciones del ACO han sido lanzadas con 100 hormigas durante 15000 *steps* con $\alpha = 3{,}75$ y $\beta = 4$. Para el GA, se ha establecido un tamaño de población de 10 escenarios y una duración de 10 iteraciones.

4.3.1. Generación de escenarios de tipo paredes

En esta prueba se parte de una población inicial de tipo paredes y se evoluciona hasta obtener un escenario acorde a diferentes niveles de dificultad. Cabe mencionar que, dado que este tipo de escenarios no presentan gran diversidad en su combinación mediante el algoritmo genético, el escenario obtenido tampoco lo hará. Sin embargo, si se podrá diferenciar una mayor o menor densidad de los escenarios dependiendo del nivel de dificultad.

NOTA: Se omiten las poblaciones de escenarios generadas en cada iteración. No obstante, estas son accesibles en el Anexo A, Anexo B y Anexo C.



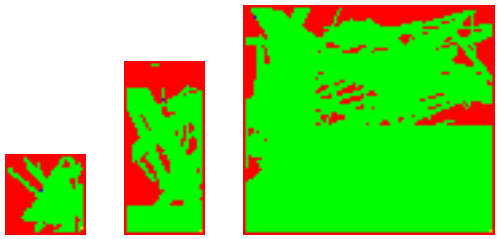
(a) Fácil (b) Media (c) Difícil

Figura 4.20: Escenarios resultantes de la generación de tipo paredes en dificultad fácil, media y difícil. Cabe mencionar como, según varía el nivel de dificultad, los escenarios cambian iteración tras iteración su dimensión y proporción de pared respecto a suelo (que en este tipo de escenarios se representa por la densidad de nubes de puntos).

4.3.2. Generación de escenarios de tipo caminos

En esta prueba se parte de una población inicial de tipo caminos y se evoluciona hasta obtener un escenario acorde a diferentes niveles de dificultad. Cabe destacar que se han generado cierta cantidad escenarios sin solución, no obstante, estos han sido omitidos ya que no influyen en la generación del escenario final.

NOTA: Se omiten las poblaciones de escenarios generadas en cada iteración. No obstante, estas son accesibles en el Anexo D, Anexo E y Anexo F.



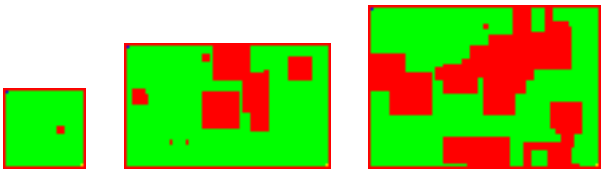
(a) Fácil (b) Media (c) Difícil

Figura 4.21: Escenarios resultantes de la generación de tipo caminos en dificultad fácil, media y difícil. Al igual que en el caso anterior, según varía el nivel de dificultad, los escenarios cambian iteración tras iteración su dimensión. Por otro lado, como ya se vio anteriormente, la proporción de pared respecto a suelo no se tiene en cuenta, lo que produce que no se modifique de manera lógica en las iteraciones. No obstante, esto no resulta ningún problema y, como se puede observar, los escenarios generados son consecuentes con el nivel de dificultad seleccionado.

4.3.3. Generación de escenarios de tipo *clusters*

En esta prueba se parte de una población inicial de tipo *clusters* y se evoluciona hasta obtener un escenario acorde a diferentes niveles de dificultad. Cabe destacar que se han generado una pequeña cantidad escenarios sin solución, no obstante, estos han sido omitidos ya que no influyen en la generación del escenario final.

NOTA: Se omiten las poblaciones de escenarios generadas en cada iteración. No obstante, estas son accesibles en el Anexo G, Anexo H y Anexo I.



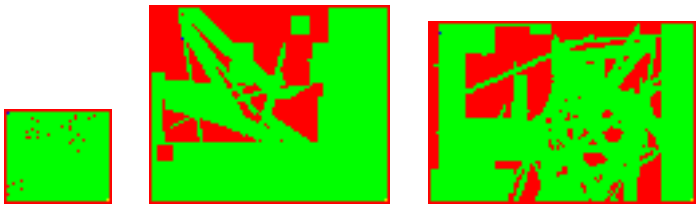
(a) Fácil (b) Media (c) Difícil

Figura 4.22: Escenarios resultantes de la generación de tipo *clusters* en dificultad fácil, media y difícil. Como anteriormente, según varía el nivel de dificultad, los escenarios cambian iteración tras iteración su dimensión. Además, también se produce un cambio en la proporción de pared y cantidad de *clusters*.

4.3.4. Generación de escenarios de tipo equiprobable

En esta prueba se parte de una población inicial con equiprobabilidad de pertenecer a cualquiera de los tipos existentes y se evoluciona hasta obtener un escenario acorde a diferentes niveles de dificultad. Según el tipo de escenarios que han evolucionado en las diferentes pruebas se han generado más o menos escenarios sin solución. En cualquier caso, estos han sido omitidos ya que no influyen en la generación del escenario final.

NOTA: Se omiten las poblaciones de escenarios generadas en cada iteración. No obstante, estas son accesibles en el Anexo J, Anexo K y Anexo L.



(a) Fácil (b) Media (c) Difícil

Figura 4.23: Escenarios resultantes de la generación de tipo equiprobable en dificultad fácil, media y difícil. Como ya se ha visto, según sea cada escenario de un tipo u otro, se produce una evolución diferente. Es por esto que este tipo de generación es la menos predecible y depende en gran medida de la población inicial aleatoria. No obstante, sigue cumpliendo los mismos criterios de evolución de siempre: variación en la dimensión y proporción de pared (excepto en escenarios de tipo caminos). Cabe mencionar que el escenario de dificultad fácil se ha generado debido a la evolución de un escenario tipo paredes y otro tipo *clusters*, mientras que en dificultad media y difícil ha sido debido a la evolución de un escenario tipo caminos y otro tipo *clusters*.

4.4. Representación tridimensional

Las pruebas anteriores detallaban los resultados obtenidos al generar escenarios aleatorios. A continuación se muestra la representación tridimensional de estos escenarios.

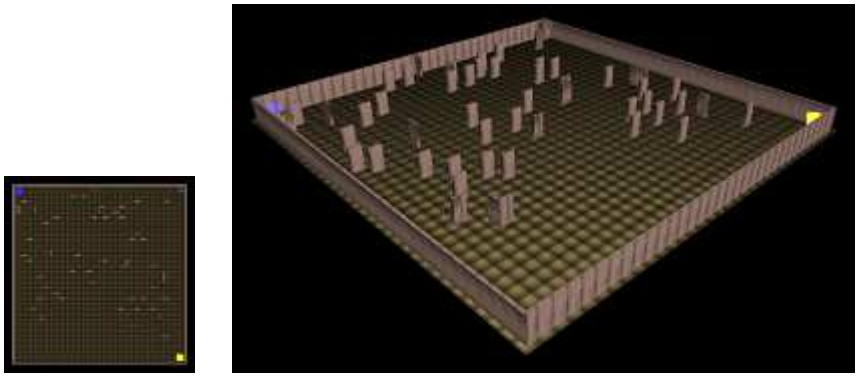


Figura 4.24: Representación 3D de un escenario tipo paredes en dificultad fácil.

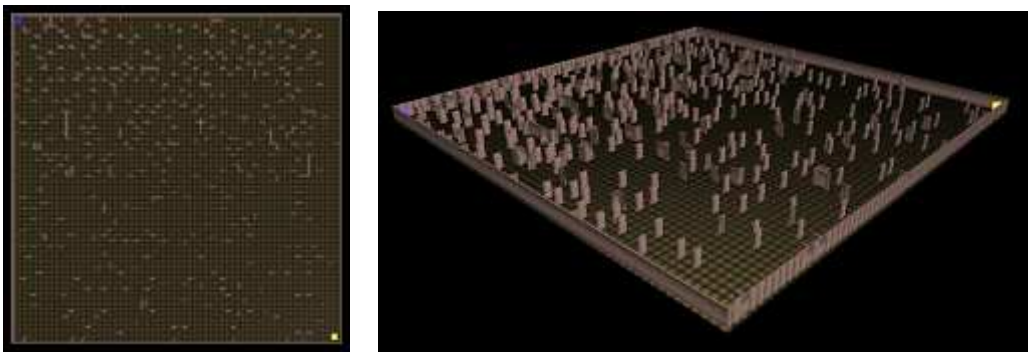


Figura 4.25: Representación 3D de un escenario tipo paredes en dificultad media.

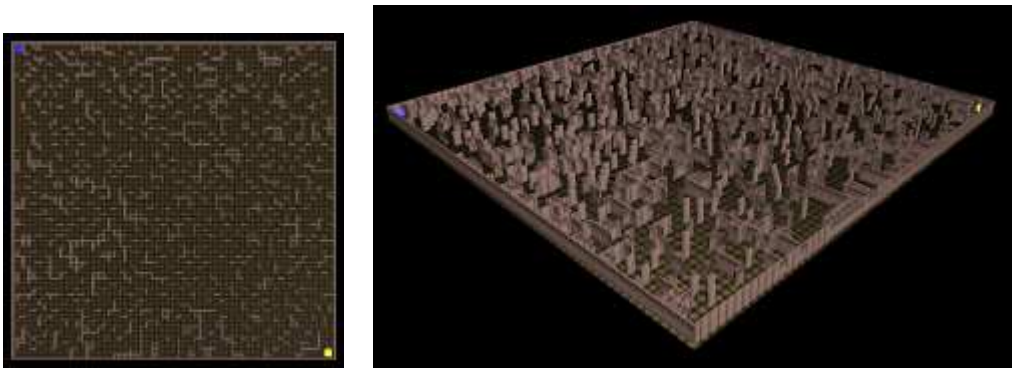


Figura 4.26: Representación 3D de un escenario tipo paredes en dificultad difícil.

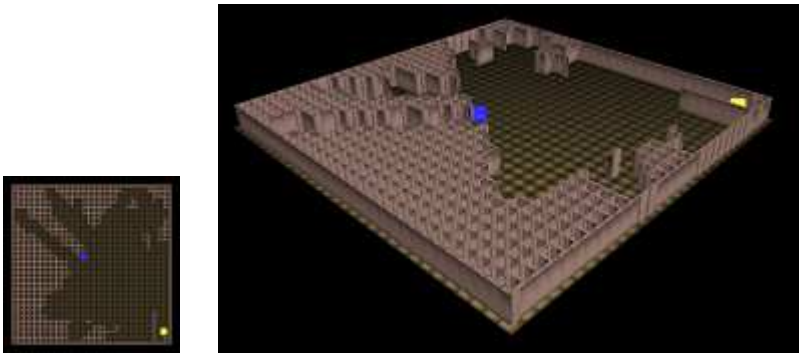


Figura 4.27: Representación 3D de un escenario tipo caminos en dificultad fácil.

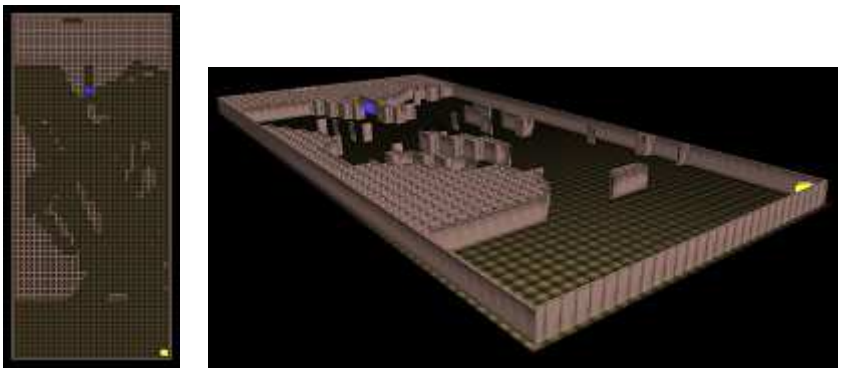


Figura 4.28: Representación 3D de un escenario tipo caminos en dificultad media.

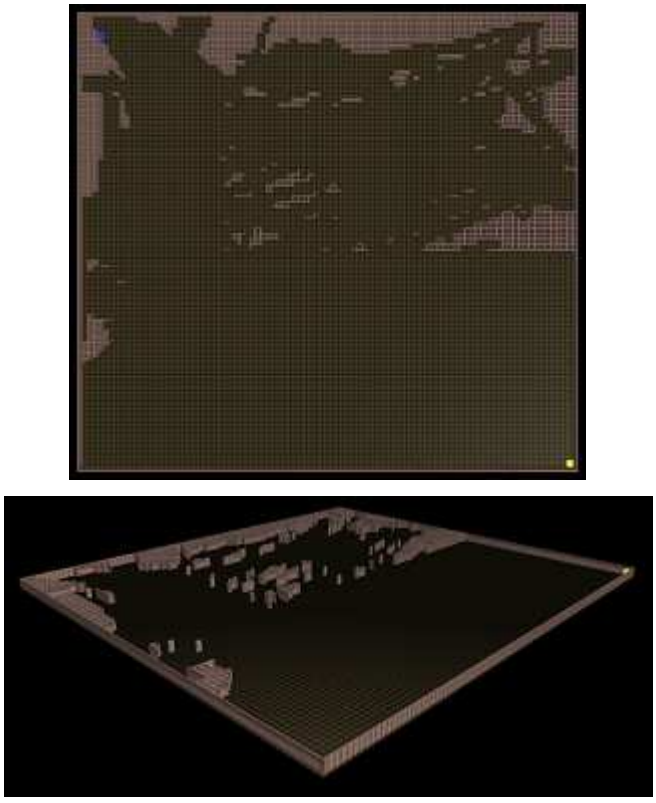


Figura 4.29: Representación 3D de un escenario tipo caminos en dificultad difícil.

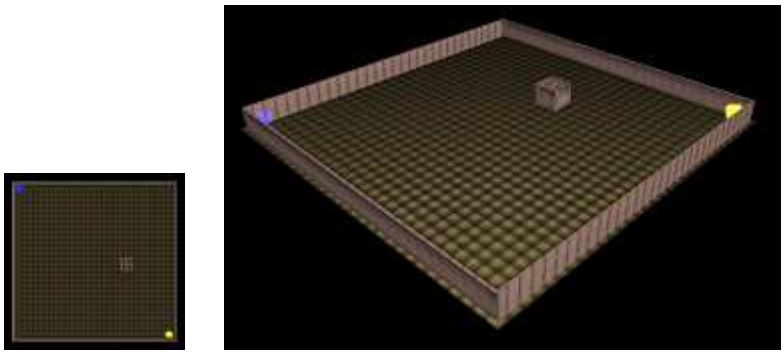


Figura 4.30: Representación 3D de un escenario tipo *clusters* en dificultad fácil.

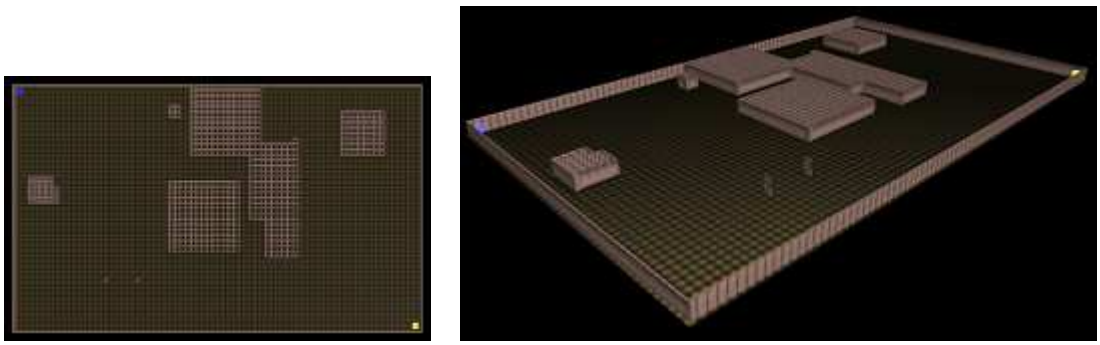


Figura 4.31: Representación 3D de un escenario tipo *clusters* en dificultad media.

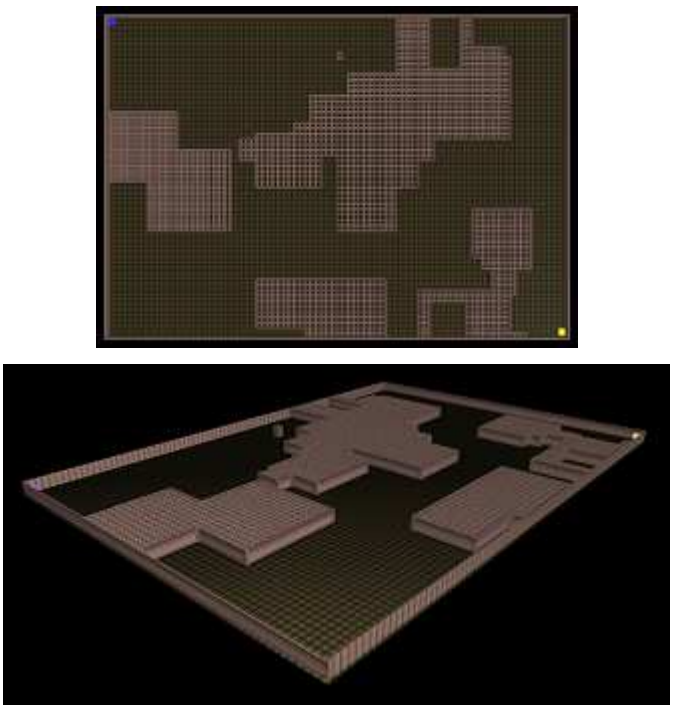


Figura 4.32: Representación 3D de un escenario tipo *clusters* en dificultad difícil.

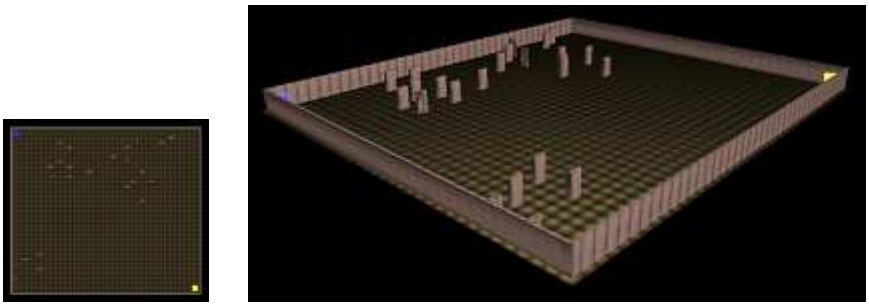


Figura 4.33: Representación 3D de un escenario tipo equiprobable en dificultad fácil.

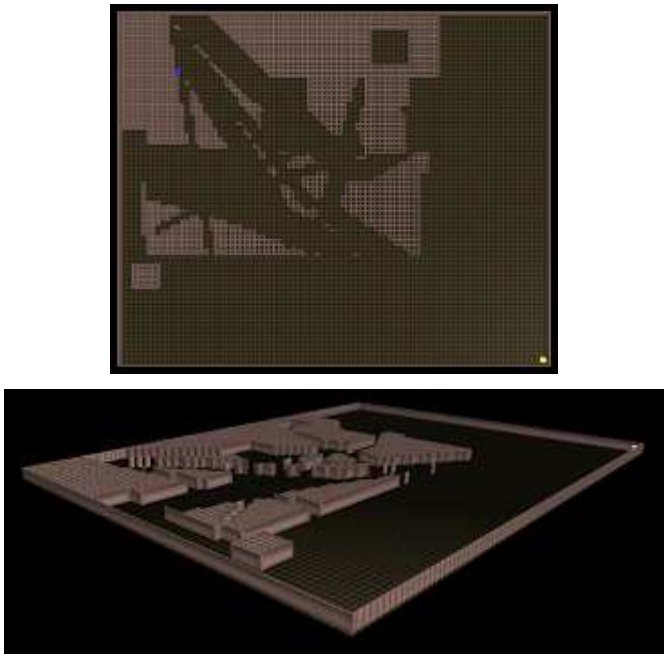


Figura 4.34: Representación 3D de un escenario tipo equiprobable en dificultad media.

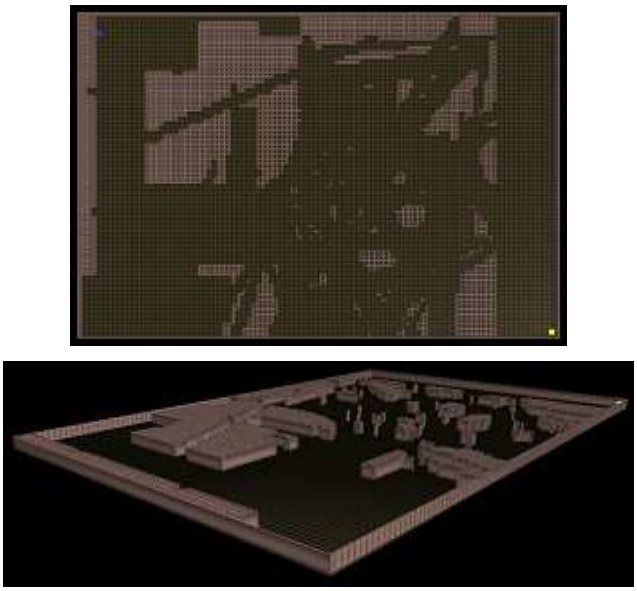


Figura 4.35: Representación 3D de un escenario tipo equiprobable en dificultad difícil.

5

Conclusiones y trabajo futuro

En esta sección se detallarán las conclusiones del proyecto realizado, así como el trabajo futuro que se podrá llevar a cabo con el fin de mejorar ciertos aspectos del mismo.

Como se ha visto tras las pruebas realizadas, mediante la utilización de un algoritmo genético es posible generar pantallas de videojuegos adaptadas a un nivel de dificultad concreto. Además, se ha demostrado que la utilización de un algoritmo de colonia de hormigas como función de *fitness* da buenos resultados para lograr este mismo objetivo. Por otro lado, los diferentes algoritmos implementados para generar la población inicial han dado también buenos resultados. No obstante, quizá con otro tipo de algoritmos u otro planteamiento en los mismos se podrían haber mejorado los escenarios que constituyen la población inicial, haciéndolos más variados y complejos.

De cualquier forma, el trabajo realizado se considera satisfactorio, ya que se han logrado los objetivos que se pretendían cumplir inicialmente. Además, se han adquirido nuevos conocimientos, principalmente en la rama de la inteligencia artificial y, más concretamente, sobre algoritmos genéticos y algoritmos basados en colonia de hormigas.

Durante la realización del trabajo, se destaca la dificultad para implementar de forma satisfactoria el algoritmo basado en colonia de hormigas principalmente debido a la complejidad del problema a tratar y la parametrización a establecer. Se tuvieron que realizar muchas pruebas hasta conseguir una buena implementación de este algoritmo. Por otro lado, se destaca también la dificultad para combinar los diferentes algoritmos y crear un algoritmo genético complejo. Por último, es necesario enfatizar el hecho de que inicialmente no se tenía ningún conocimiento previo acerca de algoritmos genéticos o algoritmos basados en colonia de hormigas y que el conocimiento necesario para la realización del trabajo se ha ido adquiriendo durante su realización.

Como trabajo futuro, se propone implementar un mecanismo de evaluación de escenarios más avanzado basado en la medición de ciertos aspectos, como la distribución de las secciones de pared en el escenario, o en la detección de cambios bruscos en la trayectoria de la mejor solución del escenario. Gracias a este mecanismo sería posible evaluar los escenarios de manera más precisa y hacer más diferencia entre los mismos.

También, se propone investigar la convergencia del algoritmo de colonia de hormigas de forma que al encontrar una solución determinada se detuviese la simulación. Igualmente, se podría implementar un proceso similar en el algoritmo genético para detectar cuándo la diferencia entre escenarios de una generación y otra es mínima y evitar generar poblaciones de individuos innecesarias.

Por último, cabe mencionar que este trabajo no sólo ha servido para obtener nuevos conocimientos sino también como aplicación de otros adquiridos a lo largo de la carrera. Como la utilización de una base de datos para guardar la información referente a las simulaciones de colonia de hormigas realizadas.

Glosario de acrónimos

- **ACO**: Ant Colony Optimization
- **BFO**: Bacterial Foraging Optimization
- **BCO**: Bee Colony Optimization
- **CSP**: Constraint Satisfaction Problem
- **FPS**: First-Person Shooter
- **GA**: Genetic Algorithm
- **NPC**: Non-Playable Character
- **PCG**: Procedural Content Generation
- **PSO**: Particle Swarm Optimization
- **TPS**: Third-Person Shooter
- **TSP**: Travelling Salesman Problem
- **VRP**: Vehicle Routing Problem

Anexos

Anexo A. Generación de un escenario tipo paredes en nivel fácil

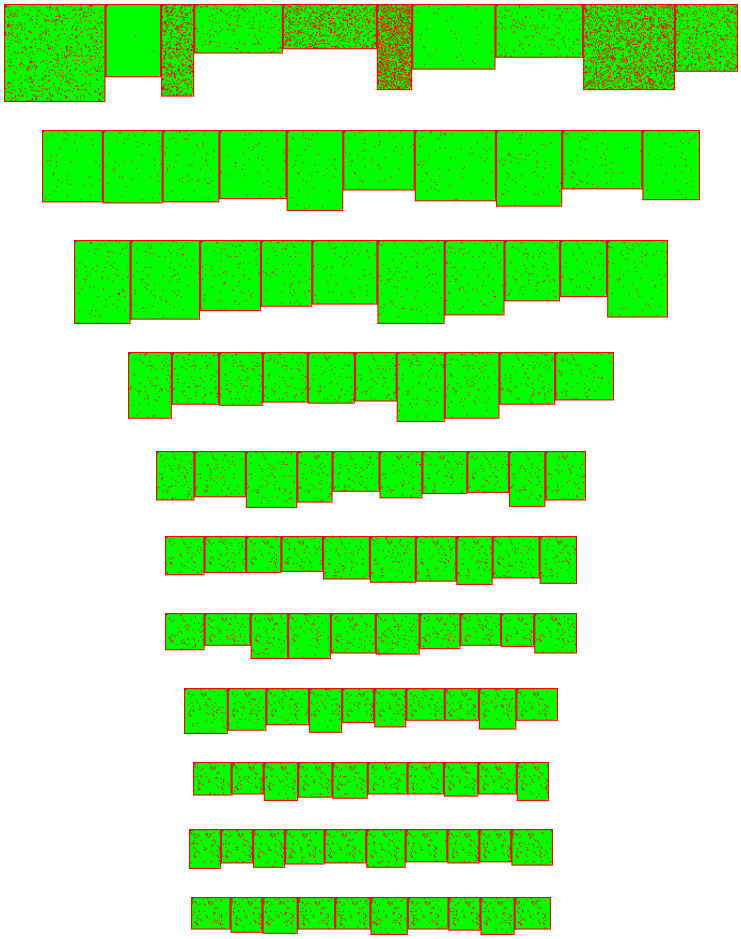


Figura 5.1: Generación de un escenario tipo paredes en dificultad fácil.

Anexo B. Generación de un escenario tipo paredes en nivel medio

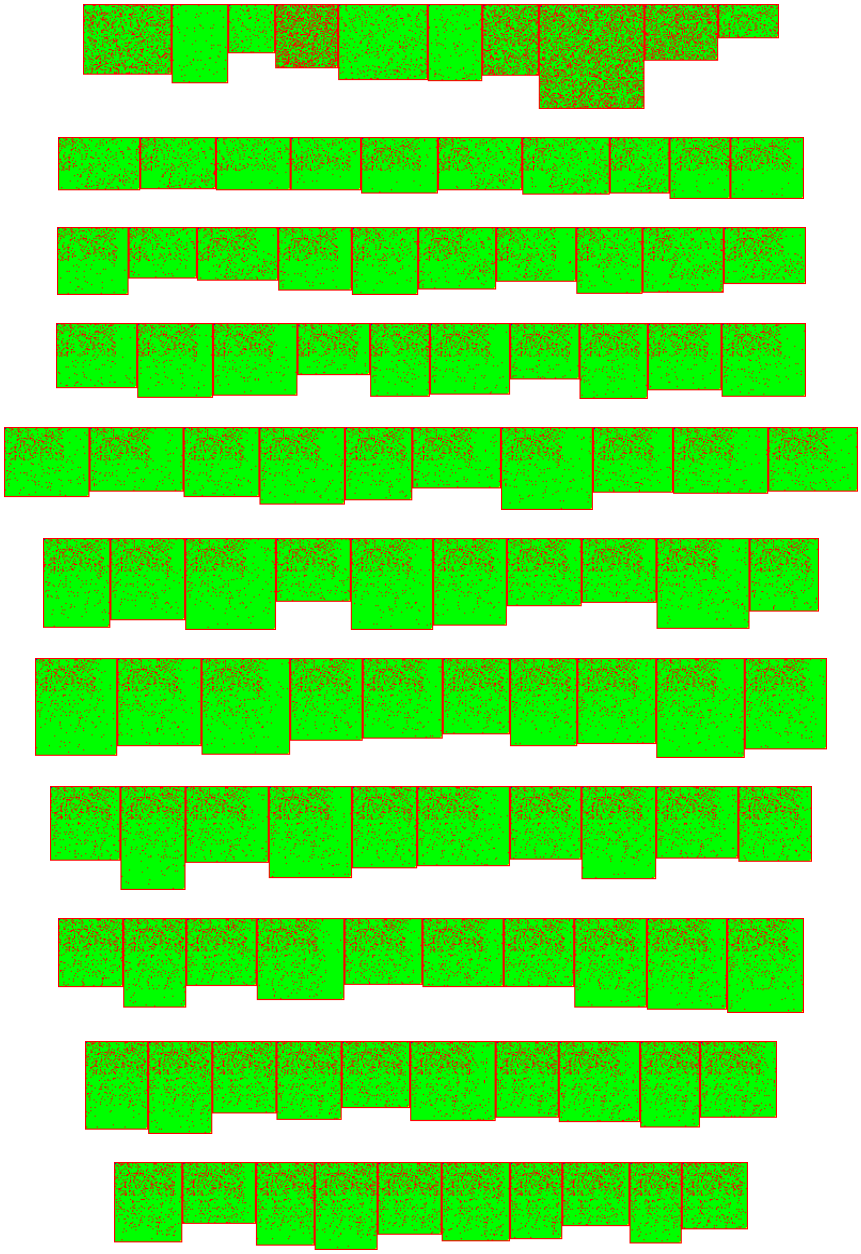


Figura 5.2: Generación de un escenario tipo paredes en dificultad media.

Anexo C. Generación de un escenario tipo paredes en nivel difícil

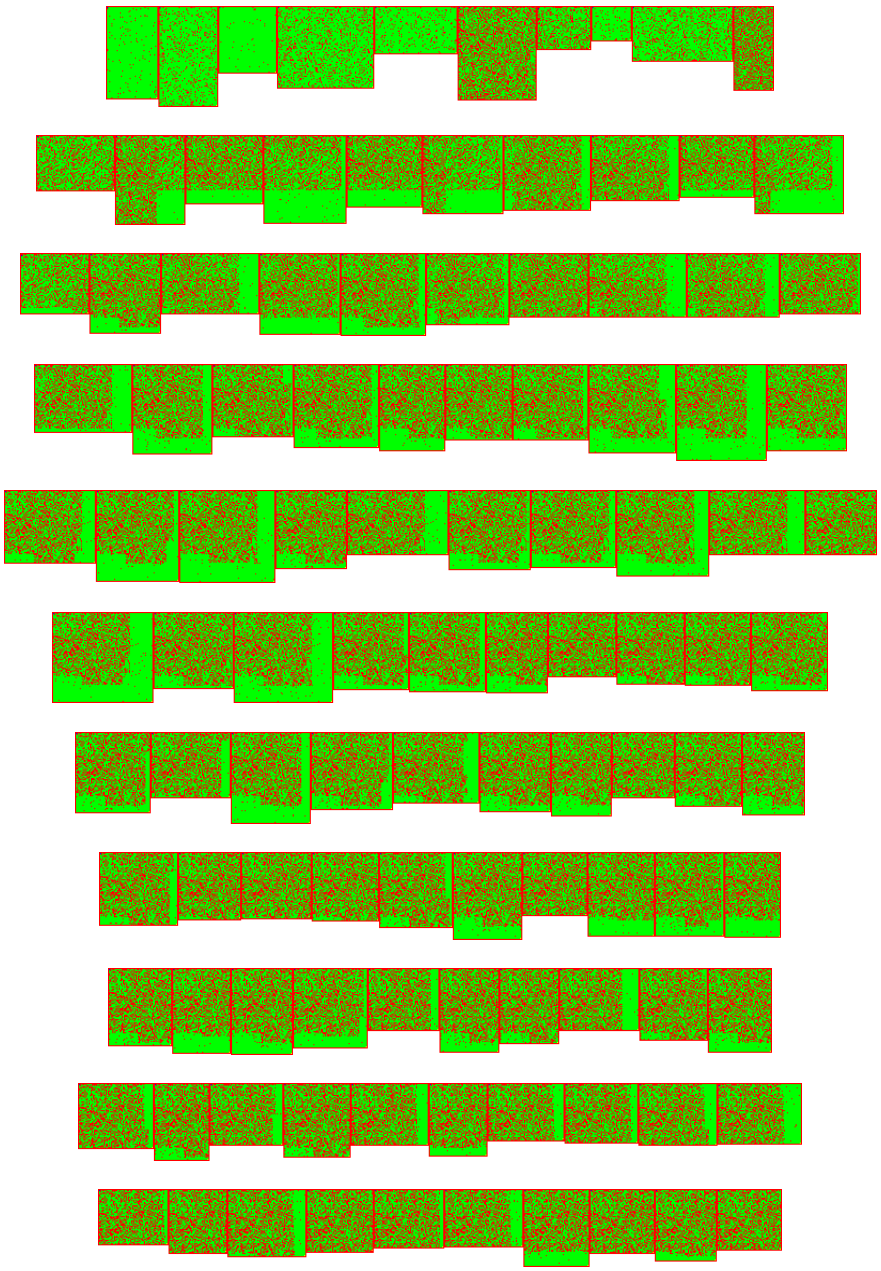


Figura 5.3: Generación de un escenario tipo paredes en dificultad difícil.

Anexo D. Generación de un escenario tipo caminos en nivel fácil

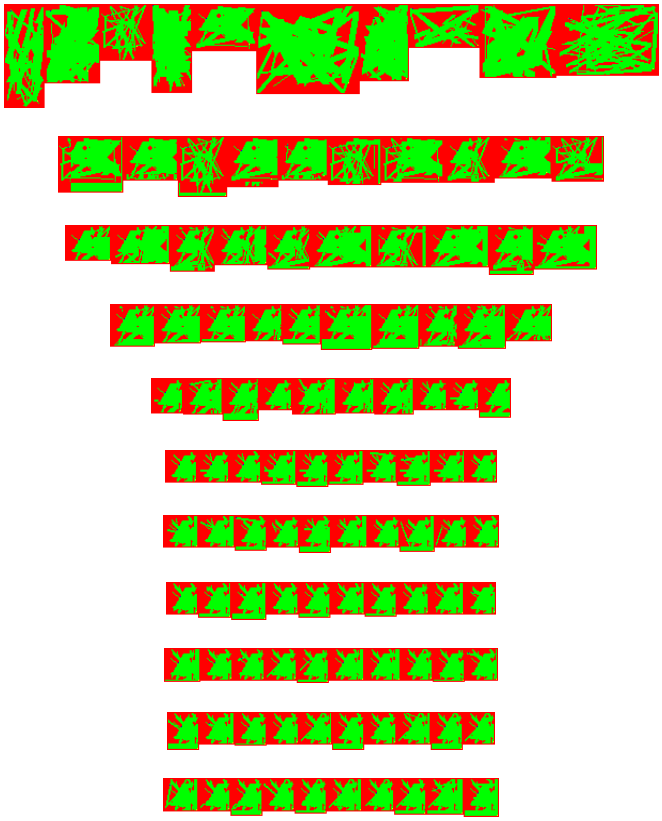


Figura 5.4: Generación de un escenario tipo caminos en dificultad fácil.

Anexo E. Generación de un escenario tipo caminos en nivel medio

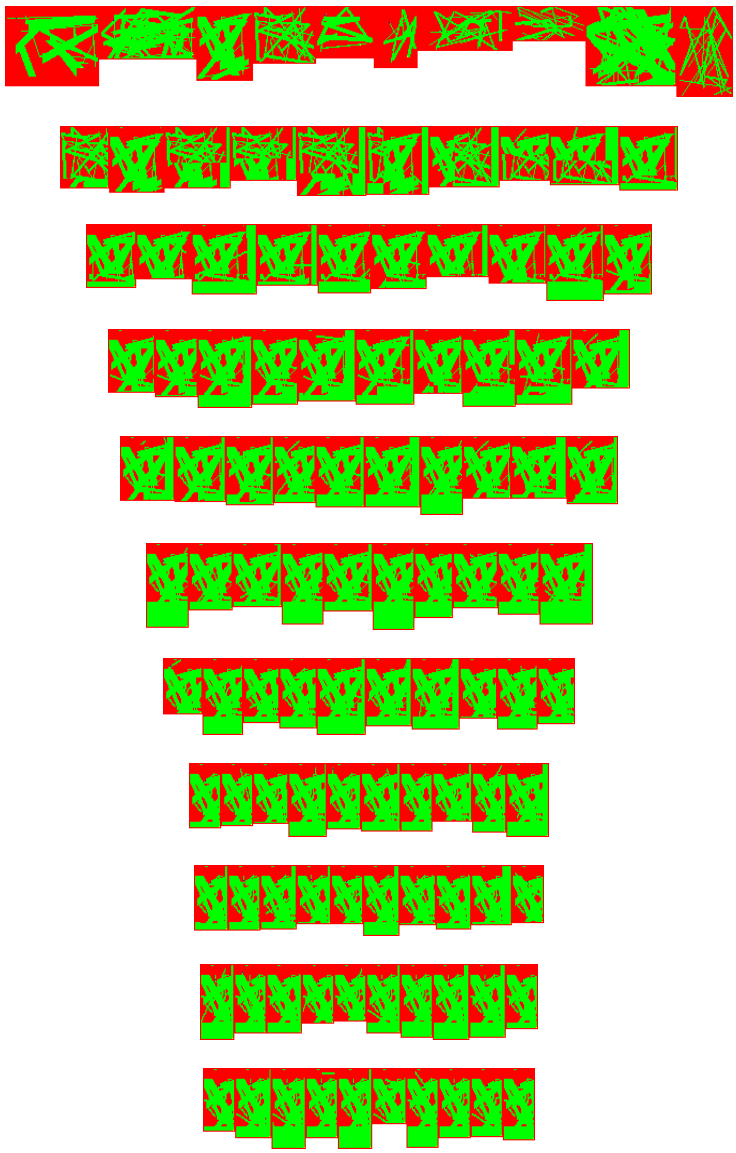


Figura 5.5: Generación de un escenario tipo caminos en dificultad media.

Anexo F. Generación de un escenario tipo caminos en nivel difícil

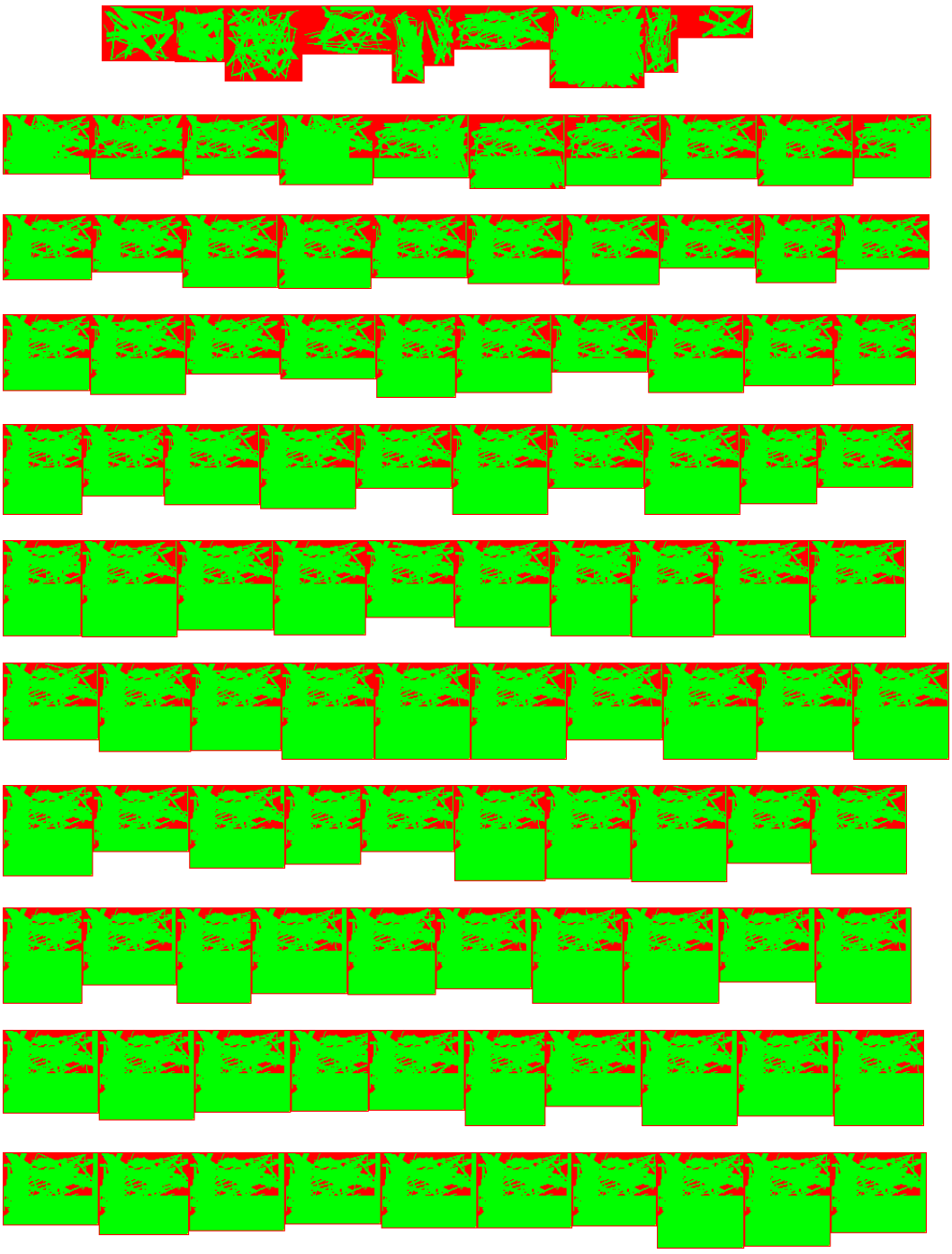


Figura 5.6: Generación de un escenario tipo caminos en dificultad difícil.

Anexo G. Generación de un escenario tipo *clusters* en nivel fácil

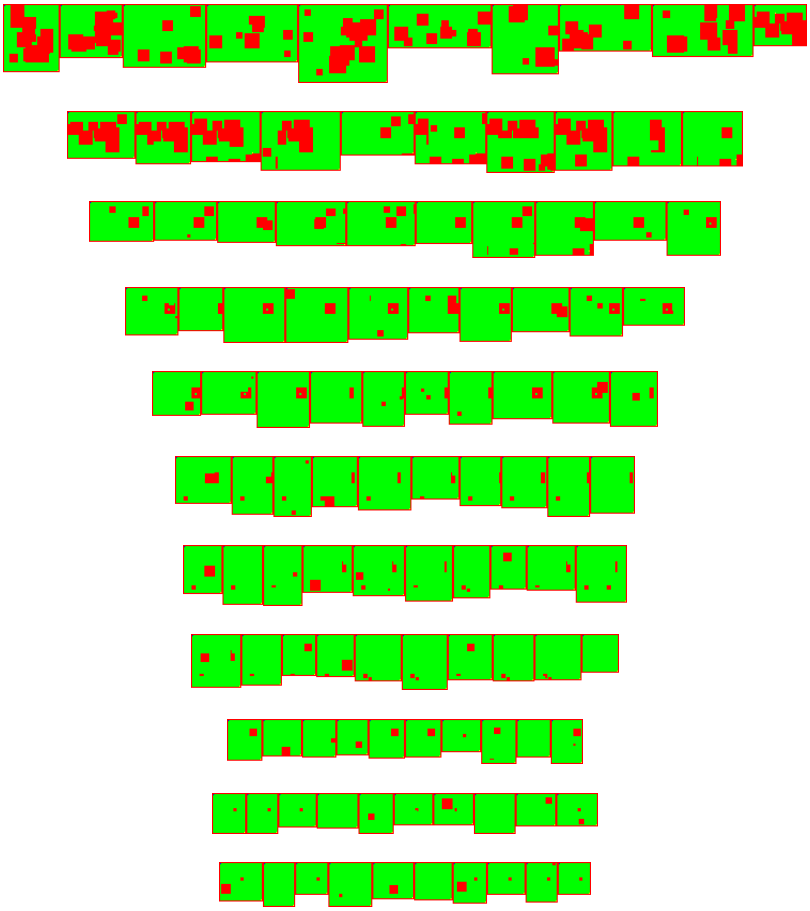


Figura 5.7: Generación de un escenario tipo *clusters* en dificultad fácil.

Anexo H. Generación de un escenario tipo *clusters* en nivel medio

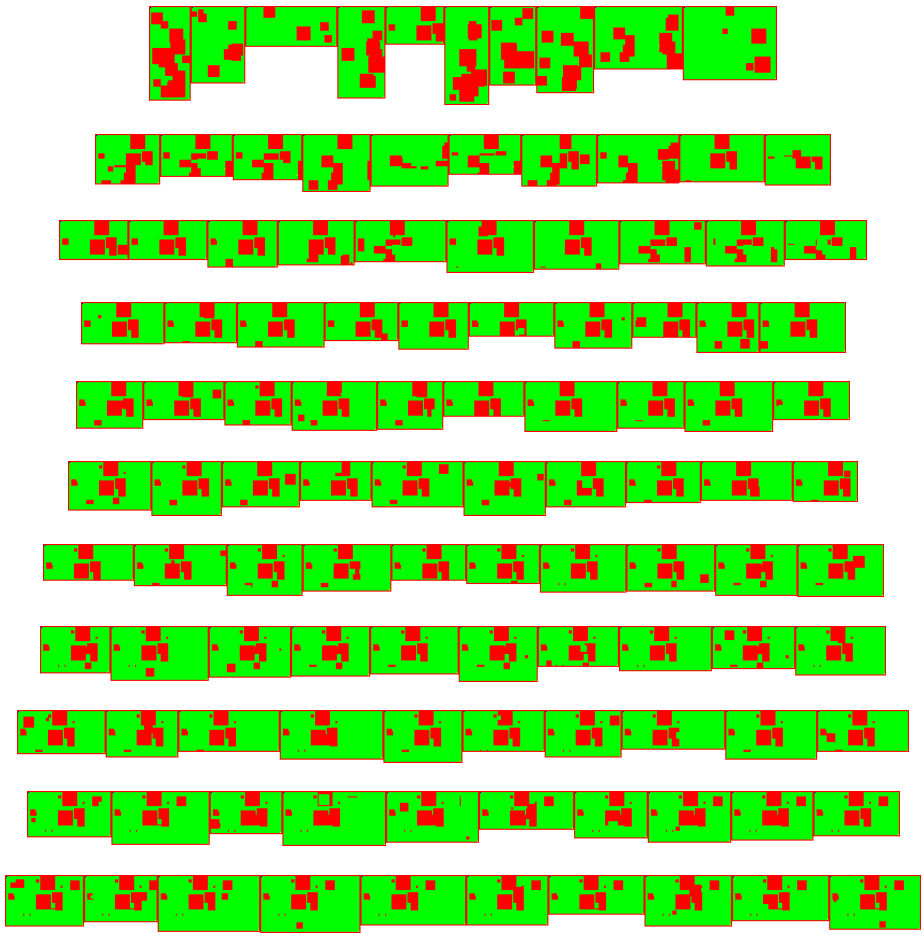


Figura 5.8: Generación de un escenario tipo *clusters* en dificultad media.

Anexo I. Generación de un escenario tipo *clusters* en nivel difícil

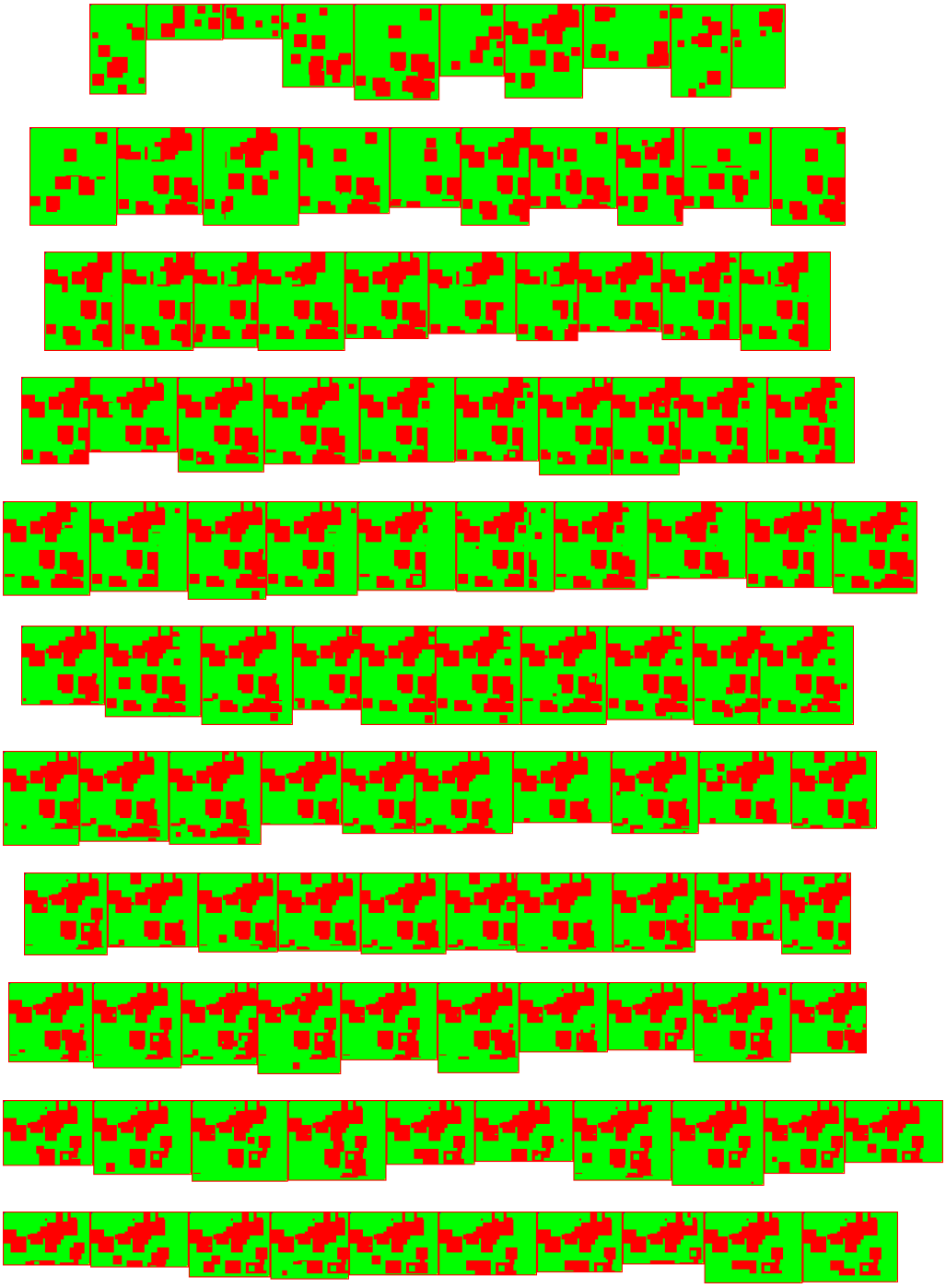


Figura 5.9: Generación de un escenario tipo *clusters* en dificultad difícil.

Anexo J. Generación de un escenario tipo equiprobable en nivel fácil

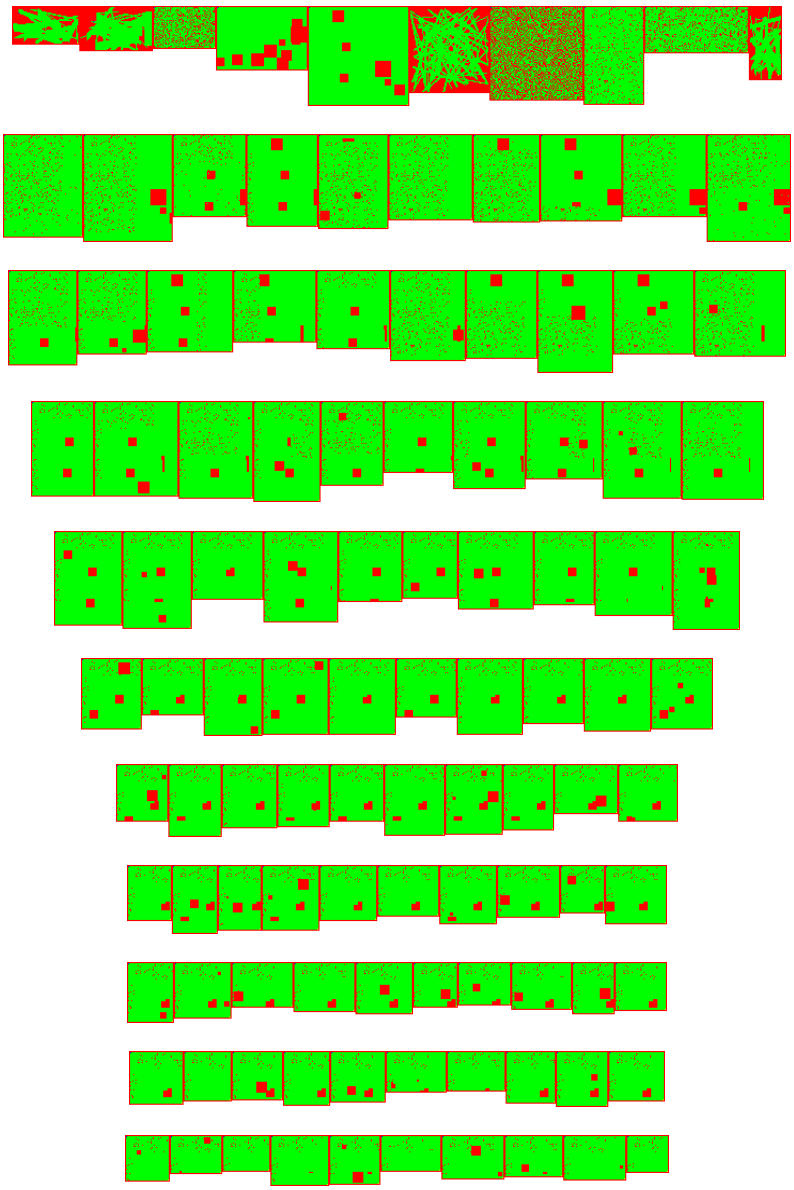


Figura 5.10: Generación de un escenario tipo equiprobable en dificultad fácil.

Anexo K. Generación de un escenario tipo equiprobable en nivel medio

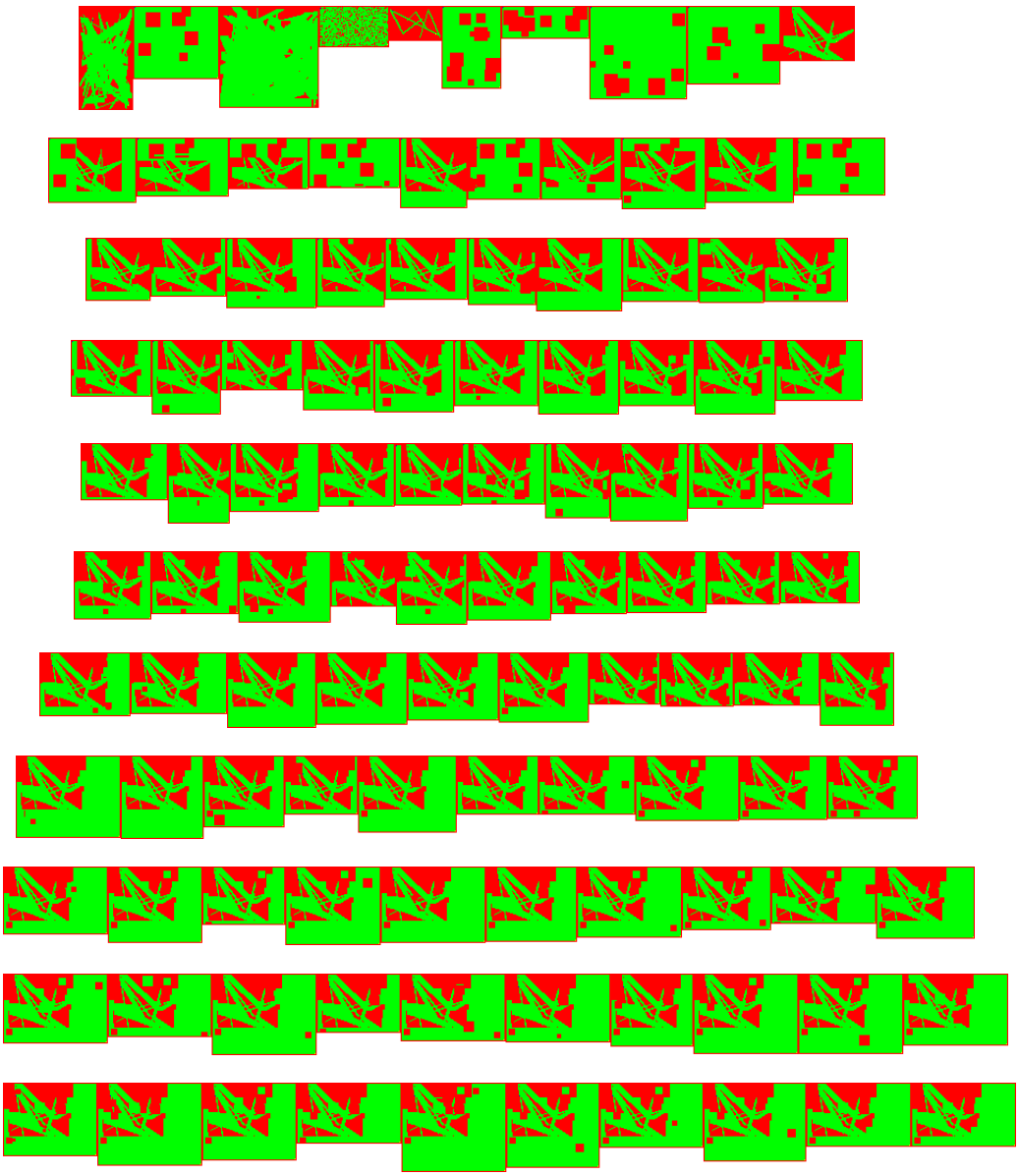


Figura 5.11: Generación de un escenario tipo equiprobable en dificultad media.

Anexo L. Generación de un escenario tipo equiprobable en nivel difícil

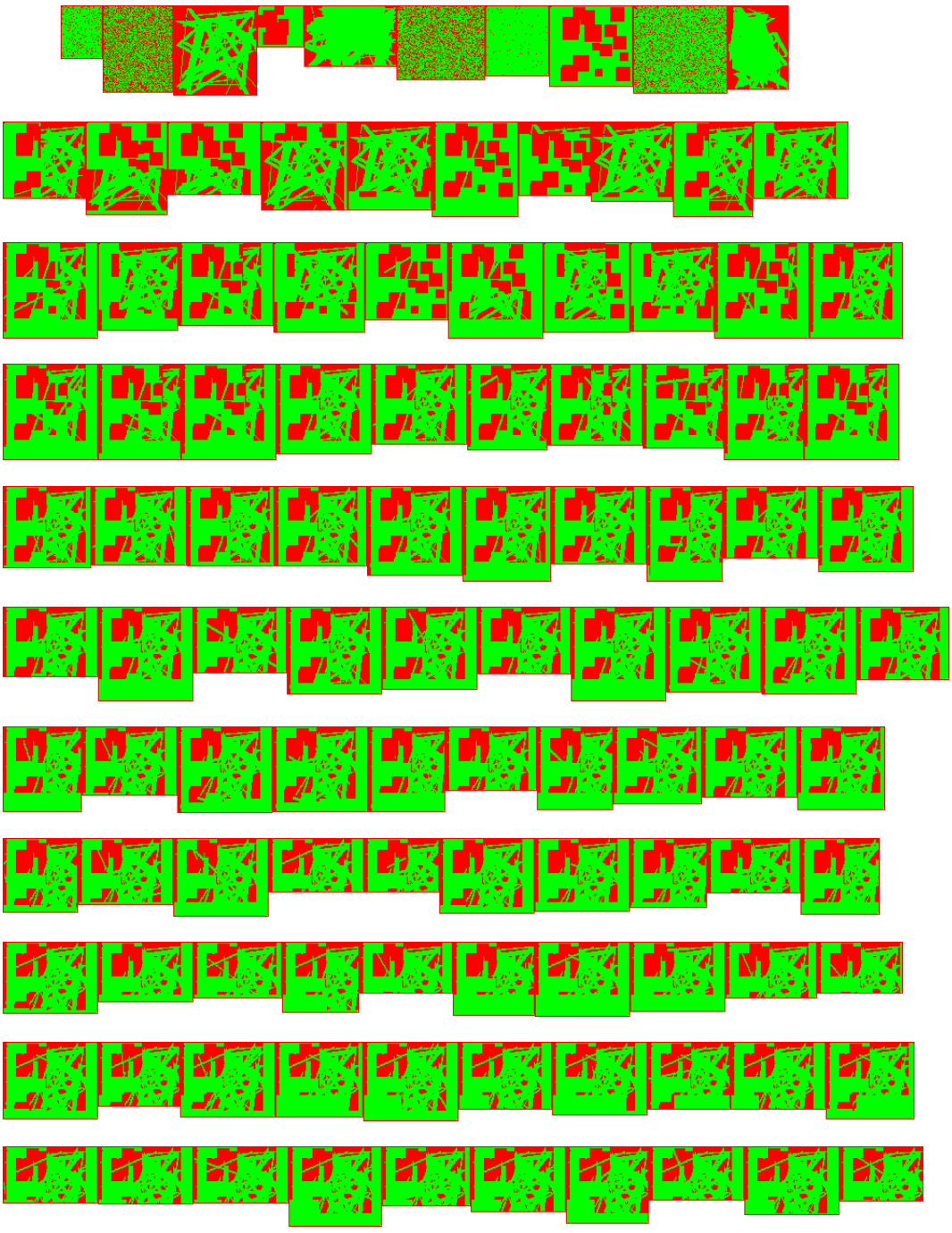


Figura 5.12: Generación de un escenario tipo equiprobable en dificultad difícil.

Bibliografía

[1] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Interactive evolution for the procedural generation of tracks in a high-end racing game. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 395–402, New York, NY, USA, 2011. ACM.

[2] D. Ashlock, C. Lee, and C. McGuinness. Search-based procedural generation of maze-like levels. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):260–273, 2011.

[3] N. Sorenson, P. Pasquier, and S. DiPaola. A generic approach to challenge modeling for the procedural creation of video game levels. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):229–244, 2011.

[4] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[5] G. Beni and J. Wang. Swarm intelligence in cellular robotic systems. In *NATO Advanced Workshop on Robotics and Biological Systems*, volume 102, pages 703–712. Springer Berlin Heidelberg, June 1989.

[6] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, 2006.

[7] Nathan Sorenson and Philippe Pasquier. Towards a generic framework for automated video game level creation. In *Proceedings of the 2010 international conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pages 131–140. Springer-Verlag, 2010.

[8] Alexis Sepchat, Romain Clair, Nicolas Monmarché, and Mohamed Slimane. Artificial ants and dynamical adaptation of accessible games level. In *Proceedings of the 11th international conference on Computers Helping People with Special Needs*, ICCHP '08, pages 593–600, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1 edition, January 1989.

[10] David A. Coley. *An introduction to genetic algorithms for scientists and engineers*. World Scientific Publishing, 1 edition, January 1999.

[11] Ziauddin Ursani. *Localized Genetic Algorithm for the Vehicle Routing Problem*. Lambert Academic Publishing, May 2010.

[12] Christian Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers and Operations Research*, 31(12):1985–2002, 2004.

[13] Prasanna Jog, Jung Y. Suh, and Dirk van Gucht. The effects of population sizeheuristic crossover and local improvement on a genetic algorithm for the traveling salesman problem. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 110–115, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[14] Heinrich Braun. On solving travelling salesman problems by genetic algorithms. In Hans-Paul Schwefel and Reinhard Männer, editors, *PPSN*, volume 496 of *Lecture Notes in Computer Science*, pages 129–133. Springer, 1990.

[15] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (2nd, extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1994.

[16] Riccardo Poli and William B. Langdon. Backward-chaining evolutionary algorithms. *Artif. Intell.*, 170(11):953–982, August 2006.

[17] Chris Miles, Juan Quiroz, Ryan Leigh, and Sushil J. Louis. Co-evolving influence map tree based strategy game players. In *in Proceedings of the IEEE Symposium on Computational Intelligence and Games. IEEE*, pages 88–95. Press, 2007.

[18] Nicholas Cole, Sushil J. Louis, and Chris Miles. Using a genetic algorithm to tune first-person shooter bots. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 139–145, 2004.

[19] Graham Kendall and Kristian Spoerer. Scripting the game of lemmings with a genetic algorithm. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 117–124, 2004.

[20] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[21] Chin Soon Chong, M.Y.H. Low, A.I. Sivakumar, and Kheng Leng Gay. A bee colony optimization algorithm to job shop scheduling. In *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, pages 1954–1961, 2006.

[22] Arijit Biswas, Sambarta Dasgupta, Swagatam Das, and Ajith Abraham. Synergy of pso and bacterial foraging optimization: A comparative study on. In *Numerical Benchmarks, Second International Symposium on Hybrid Artificial Intelligent Systems (HAIS 2007), Advances in Soft computing Series*, pages 255–263. Springer Verlag, 2007.

[23] M. Dorigo, G. D. Di Caro, and L. Gambardella. Ant colony optimization: a new meta-heuristic. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2, pages 1470–1477, Mayflower Hotel, Washington D.C., USA, Jun-Sep 1999. IEEE Press.

[24] Luca Maria Gambardella, Éric Taillard, and Giovanni Agazzi. Macs-vrptw: A multiple colony system for vehicle routing problems with time windows. In *New Ideas in Optimization*, pages 63–76, Maidenhead, UK, England, 1999. McGraw-Hill Ltd., UK.

[25] John E. Bell and Patrick R. McMullen. Ant colony optimization techniques for the vehicle routing problem. *Advanced Engineering Informatics*, 18(1):41 – 48, 2004.

[26] T. Stützle and M. Dorigo. ACO Algorithms for the Traveling Salesman Problem. In K. Miettinen, M. Mäkelä, P. Neittaanmäki, and J. Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science: Recent Advances in Genetic Algorithms, Evolution Strategies, Evolutionary Programming, Genetic Programming and Industrial Applications*. John Wiley & Sons, 1999.

[27] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The ant system: Optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41, 1996.

[28] R.S. Parpinelli, H.S. Lopes, and A.A. Freitas. Data mining with an ant colony optimization algorithm. *Evolutionary Computation, IEEE Transactions on*, 6(4):321–332, 2002.

[29] Carolin Kaiser, Johannes Krockel, and Freimut Bodendorf. Swarm intelligence for analyzing opinions in online communities. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, volume 0, pages 1–9, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[30] P.S Shelokar, V.K Jayaraman, and B.D Kulkarni. An ant colony approach for clustering. *Analytica Chimica Acta*, 509(2):187–195, 2004.

[31] M. Emilio, M. Moises, R. Gustavo, and S. Yago. Pac-mant: Optimization based on ant colonies applied to developing an agent for ms. pac-man. In Georgios N. Yannakakis and Julian Togelius, editors, *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 458–464. IEEE, 2010.

[32] Xingguo Chen, Hao Wang, Weiwei Wang, Yinghuan Shi, and Yang Gao. Apply ant colony optimization to tetris. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1741–1742, New York, NY, USA, 2009. ACM.

[33] Jose A. Mocholi, Javier Jaen, Alejandro Catala, and Elena Navarro. An emotionally biased ant colony algorithm for pathfinding in games. *Expert Systems with Applications*, 37(7):4921–4927, 2010.