APRENDIZAJE AUTOMÁTICO APLICADO A JUEGOS DE ESTRATEGIA EN TIEMPO REAL: UN ENFOQUE GENÉTICO

MACHINE LEARNING APPLIED TO REAL-TIME STRATEGY GAMES: A GENETIC APPROACH

RAFAEL HERRERA TROCA
RUBÉN RUPERTO DÍAZ

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Grado en Ingeniería Informática
Curso 2019/2020

26 de junio de 2020

Directores:

Antonio Alejandro Sánchez Ruiz-Granados
Pedro Pablo Gómez Martín

En esta memoria se describe el desarrollo de una inteligencia artificial capaz de jugar a un juego de estrategia en tiempo real. El código correspondiente puede encontrarse en el repositorio de *GitHub* `https://github.com/TFG-Informatica/Aprendizaje-automatico-aplicado-a-juegos-RTS`.

Autoría de la imagen de la portada:

Resumen en castellano

La aplicación de técnicas de inteligencia artificial a juegos de estrategia en tiempo real es un campo de investigación amplio con numerosos problemas abiertos. Para probar estas técnicas antes de aplicarlas a dominios complejos, se creó el juego μ-RTS, una versión simplificada del género de estrategia en tiempo real. Nosotros proponemos el uso de comportamientos precodificados para cada tipo de unidad del juego, que se seleccionan formando estrategias globales que determinan las decisiones de la IA durante una partida. En un paso mayor de complejidad, utilizamos varias estrategias durante la misma partida, lo que permite a la IA cambiar su comportamiento en función de la etapa de la partida en la que se encuentre. Para confeccionar las estrategias y seleccionar las más adecuadas para una partida, utilizamos un algoritmo genético. Finalmente, incluimos un análisis de cómo se desenvuelve nuestro planteamiento frente a otros bots desarrollados para el juego.

Palabras clave

Inteligencia artificial, estrategia en tiempo real, juegos RTS, microRTS, aprendizaje automático, algoritmo genético, estrategias precodificadas.

Abstract

The use of artificial intelligence techniques to play real-time strategy games is a broad field of research with many open problems. The game μ-RTS, which is a simplified version of an RTS game, was created to test this techniques before using them in more complex domains. We present a bot that uses hard-coded behaviors designed for each different type of unit in μ-RTS. Some of these behaviors are selected to create a strategy that the bot can use during the game. Moreover, several strategies can be used in the same game, which lets the bot change its way to play depending on the stage of the game. We use a genetic algorithm to create these strategies and choose the best ones for a particular game. Finally, we include a study about how this approach performs against other state-of-the-art bots created for μ-RTS.

This document is mainly written in Spanish. However, a brief introduction in English can be found at page 5, and some conclusions of the work can be found at page 79.

Keywords

Artificial intelligence, real-time strategy, RTS games, microRTS, machine learning, genetic algorithm, hard-coded strategies.

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

Hay muchas personas sin cuya colaboración la realización de este Trabajo de Fin de Grado no habría sido posible o no sería lo que ha llegado a ser.

En primer lugar, nos gustaría agradecer a nuestros tutores, Antonio y Pedro Pablo, su trabajo a la hora de guiarnos en la realización de este proyecto, en especial por sus ideas en los momentos más críticos y sus exhaustivas correcciones, siempre hechas de forma constructiva y tratando de ayudarnos a mejorar.

En segundo lugar, también nos gustaría agradecer a Manuel Freire sus indicaciones sobre concurrencia en Java y sobre el uso correcto de hilos mediante una *ThreadPool* tras oír lo perdidos que estábamos en este aspecto.

También queremos agradecer su trabajo a los profesores que nos han dado clase durante la carrera y que, de una forma u otra, han contribuido a nuestra formación y a nuestro crecimiento personal.

Por último, pero no menos importante, también queremos agradecer a nuestras familias su apoyo incondicional en los buenos y malos momentos, que nos ha permitido seguir adelante con nuestra carrera hasta ponerle culmen con este trabajo.

A todos ellos nuevamente, gracias.

Dedicatoria

Este trabajo es el punto culminante de nuestro esfuerzo a lo largo de los últimos años, por eso nos gustaría dedicárselo a nuestros amigos, que nos han ayudado resolviendo nuestras dudas en incontables ocasiones y nos han apoyado para sacar adelante la carrera. Sin ellos, nuestro paso por la universidad habría sido mucho más duro.

Capítulo 1

Introducción

En la actualidad, la inteligencia artificial es un campo de estudio en auge y se busca aplicarla a numerosos aspectos de nuestra vida. En particular, su desarrollo orientado a los juegos, ya sean estos juegos de mesa clásicos o videojuegos, resulta especialmente interesante y relevante por el alto nivel de complejidad que pueden llegar a ofrecer y por las aplicaciones que pueden tener los resultados obtenidos al llevarlos a otros campos.

Un género particularmente interesante de juegos a los que se puede tratar de aplicar la inteligencia artificial son los juegos de estrategia en tiempo real o RTS por sus siglas en inglés. Estos juegos se caracterizan por la ausencia de turnos entre sus jugadores, al contrario que otros juegos en los que se ha aplicado con éxito la inteligencia artificial como el *go* (con el famoso jugador automático *AlphaGo*) o el ajedrez. Esta ausencia de turnos provoca que el juego sea mucho más complejo de interpretar a nivel computacional, ya que los jugadores pueden actuar en cualquier momento. Esto, unido a la gran cantidad de unidades que puede llegar a manejar a la vez cada jugador, hace que la aplicación de la inteligencia artificial resulte especialmente compleja. Tanto es así, que se han probado gran cantidad de técnicas con resultados muy variados pero sin llegar a encontrar una que sea claramente superior a las demás.

En la primera parte de este trabajo se hace una recopilación de estas técnicas, basada en la realizada por Santiago Ontañón y otros autores en [1], para adquirir una visión general del trabajo hecho hasta el momento y los resultados obtenidos. Este estudio nos aporta una amplia perspectiva de las opciones que se han utilizado, su funcionamiento e incluso de la posibilidad de combinarlas. Estos conocimientos nos aportan una sólida base para comenzar a trabajar a la hora de desarrollar nuestro propio bot en los capítulos 4 y 5.

A continuación se dedica el capítulo 3 al estudio del juego μ-RTS, una versión simplificada de código abierto de los juegos de estrategia en tiempo real, diseñado para ser una mesa de pruebas sobre la que poder trabajar cómodamente. Al estar especialmente diseñado para la investigación de técnicas de inteligencia artificial

aplicadas a los juegos RTS, resulta sencillo crear bots capaces de jugar, ya que el propio juego incluye facilidades tanto para su desarrollo como para su acoplamiento al juego.

Además, este juego se utiliza todos los años desde el 2017 para realizar una competición en la *Conference of Games* (CoG) del IEEE. Los resultados de este concurso, así como sus reglas, están disponibles online [2], y nos han resultado útiles para poder enfrentar los bots que desarrollamos en los capítulos 4 y 5 con rivales más realistas que los que incluye μ-RTS.

Posteriormente, en el capítulo 4 se procede al desarrollo de un bot capaz de jugar a μ-RTS. Después de haber estudiado en la primera parte las técnicas usadas hasta el momento en la aplicación de la inteligencia artificial a este tipo de juegos, disponemos de un amplio abanico de opciones que utilizar para desarrollar nuestro propio bot. Proponemos el uso de comportamientos fijos precodificados para cada tipo de unidad del juego, que se combinan para una estrategia global que determina el comportamiento del bot durante la partida.

Usando esta aproximación probamos a enfrentar todos los bots resultantes de las posibles combinaciones de estos comportamientos contra los proporcionados en el propio juego y algunos de los bots mejor clasificados en los concursos de la CoG de años anteriores. Tras un exhaustivo análisis de los resultados de estos experimentos, que se detalla en la sección 4.4, conseguimos obtener las mejores estrategias para jugar en mapas de distintos tamaños.

En el capítulo 5 pretendemos llegar un paso más lejos que en el anterior y pasamos a utilizar varias estrategias durante la partida. De esta forma, el bot es capaz de ir cambiando su comportamiento a medida que avanza la partida y se vuelve mucho más flexible y adaptable. No obstante, esto también tiene un inconveniente, que es el drástico aumento del número de bots posibles.

Para resolver este problema hacemos uso del estudio que hemos hecho en la primera parte y nos inspiramos en el bot Tiamat [3], basado en el artículo [4], y ganador del concurso de μ-RTS de la CoG en 2018. En el artículo en que se basa, se emplea un algoritmo genético para resover un problema parecido al nuestro, por lo que decidimos adaptar su planteamiento a nuestro caso, si bien el algoritmo que usamos dista ligeramente del genético habitual y tiene algunas modificaciones detalladas en la sección 5.3.

Por último, hacemos un profundo análisis de los resultados obtenidos por el genético, evaluándolo al enfrentarse tanto a los bots proporcionados en μ-RTS como a algunos de los participantes en los concursos anteriores de la CoG. Además, probamos dos aproximaciones distintas del algoritmo genético: una en la que el cálculo del *fitness* de cada individuo se realiza enfrentándolo contra una selección de bots de referencia y otra en la que dicho cálculo se hace compitiendo contra el resto de bots de su generación.

Todo el código que hemos desarrollado para este Trabajo de Fin de Grado se encuentra disponible en el repositorio de *GitHub* `https://github.com/TFG-Informatica/Aprendizaje-automatico-aplicado-a-juegos-RTS`.

1.1. Objetivos

Una vez hemos dado una vista general sobre el contenido del trabajo, pasamos a enumerar brevemente los objetivos que se pretende cubrir:

- Realización de un estudio de la investigación actual en el campo de la inteligencia artificial aplicada a los juegos de estrategia en tiempo real.
- Desarrollo de un bot sencillo capaz de desenvolverse en el juego μ-RTS y análisis de los resultados obtenidos por dicho bot.
- Desarrollo de un bot para el juego μ-RTS capaz de enfrentarse a los competidores del concurso de dicho de juego organizado por el IEEE en su CoG [2], así como el análisis de los resultados obtenidos.

En el capítulo 6 se revisarán estos objetivos para evaluar si se han cumplido y qué se ha realizado dentro de cada uno.

Introduction

Nowadays, artificial intelligence, and its application to numerous aspects of our lives, is a growing field of study. In particular, its development oriented to games, be they classic board games or video games, is especially interesting and relevant due to the high level of complexity they can offer and the applications the results may have when taken to others fields.

A particularly interesting genre of games to which artificial intelligence can be applied are real-time strategy games, or RTS. These games are characterizzed by the absence of turns between their players, unlike other games in which artificial intelligence has been successfully applied, such as go (with the famous automatic player *AlphaGo*) or chess. This absence of turns makes the game much more complex to interpret at a computational level, since players can act at any time. This, together with the large number of units that each player may have to manage at the same time, makes the application of artificial intelligence especially complex. So much so, that a large number of techniques have been tried with very varied results, but without finding one that is clearly superior to the others.

In the first part of this work, a compilation of this techniques is made, based on the one carried out by Santiago Ontañón et al. in [1], to acquire an overview of the work done so far and the results obtained. This study gives us a broad perspective of the options that have been used, how do they operate and even the possibility of combining them. These insights provide us with a solid foundation to start working on when developing our own bot in chapters 4 and 5.

Chapter 3 is then devoted to studying the game μ-RTS, a simplified open source version of real-time strategy games designed to be a benchtable on which research can be done comfortably. Being especially designed for the investigation of artificial intelligence techniques applied to RTS games, it is very easy to create bots capable of playing, since the game itself includes facilities both for its development and for its coupling to the game.

In addition, this game has been used every year since 2017 to hold a competion at the IEEE Conference of Games (CoG). The results of this contest, as well as it rules, are available online [2], and have been useful for us so we could face the bots that we developed in chapters 4 and chapters 5 with more realistic rivals than those included in μ-RTS.

Later, in chapter 4, we proceed to the development of a bot capable of playing μ-RTS. After having studied in the first part the techniques used so far in the application of artificial intelligence to this type of games, we have a wide range of options to use when developing our own bot. We propose the use of hard-coded fixed behaviors for each type of unit in the game, which are combined for an overall strategy that determines the behavior of the bot during the match.

Using this approach we confront all the bots resulting from the possible combinations of these behaviors against those provided in the game itself and some of the top ranked bots in the CoG contests of previous years. After an exhaustive analysis of the results of these experiments, which is detailed in the section 4.4, we managed to obtain the best strategies for playing on maps of different sizes.

In chapter 5 we try to go one step further that we did in the last one, and we use various strategies during the game. This way, the bot is able to change its behavior as the game progresses and becomes much more flexible and adaptable. However, this also has a drawback, which is the enormous increase in the number of possible bots.

To solve this problem, we make use of the study we made in the first part and take inspiration in Tiamat [3], based on the article [4] and winner of the contest for μ-RTS of the 2018 CoG. In the article on which it is based, an genetic algorithm is used to solve a problem similar to ours, so we decided to adapt its approach to our case, although the algorithm we use is slightly different from the usual one and has some modifications detailed in section 5.3.

Finally, we do an in-depth analysis of the results obtained by the genetic, evaluating it by facing both the bots provided in μ-RTS and some of the participants in previous CoG contest. In addition, we tested two different approaches to the genetic algorithm: one in which the calculation of the *fitness* of each individual is carried out by confronting it against a selection of reference bots and the other, in which said calculation is done by competing against the rest of the bots of its generation.

All the code we have developed for this work is available in the GitHub repository `https://github.com/TFG-Informatica/Aprendizaje-automatico-aplicado-a-juegos-RTS`.

Objectives

Once we have given an overview of the content of the work, we will briefly list the objectives we intend to cover:

- Carrying out a study of current research in the field of artificial intelligence applied to real-time strategy games.

- Development of a simple bot capable of playing μ-RTS and analysis of the results obtained by said bot.
- Development of a bot for μ-RTS capable of taking on the competitors of the said game competition organized by the IEEE in its CoG[2], as well as the analysis of the results obtained.

In chapter 6.2, we will review these objectives to assess whether they have been met and what has been accomplished within each.

Capítulo 2

Inteligencia artificial aplicada a juegos RTS

Para ponernos en contexto, vamos a hablar de los juegos de estrategia en tiempo real o juegos RTS, por sus siglas en inglés, de qué los caracteriza y de cómo se ha abordado el problema de crear bots capaces de jugar a este tipo de juegos.

2.1. Juegos de estrategia en tiempo real

Los juegos RTS son un género de videojuegos en el que varios jugadores se enfrentan en un mapa que puede imitar un entorno natural o fantástico. Los jugadores controlan unas unidades que pueden moverse por el terreno e interactuar con los elementos del mapa y deben explotar los recursos que les ofrece el entorno para construir una economía capaz de sustentarse y de conquistar a sus enemigos.

Algunos ejemplos se este género son la saga *Age of Empires* [5], en la que los jugadores deben construir una civilización ambientada en distintas épocas de la historia y vencer al resto de jugadores, o *StarCraft* [6], donde el desarrollo de la partida sucede en una sección distante de la Vía Láctea en el siglo XXVI y los jugadores deben escoger una raza entre tres y tratar de hacerse con la victoria.

2.1.1. Características de los juegos RTS

Los RTS se caracterizan porque no existen los turnos, de modo que los jugadores pueden tomar decisiones y ejecutar acciones en cualquier momento. Es por este motivo por el que se les conoce como *estrategia en tiempo real*. Sin embargo, en su implementación interna puede existir algún tipo de ciclo que se repite continuamente y que sincroniza los movimientos de los jugadores.

Además, las acciones pueden ser *durativas*, es decir, pueden no tomar efecto inmediatamente, requiriendo cierto tiempo para ser completadas. Por ejemplo, desde el momento en que se comienza la construcción de un edificio hasta que acaba, pasará un tiempo durante el cual al menos un constructor estará ocupado ejecutando esa tarea y el edificio no será funcional hasta estar terminado.

Debido a que las acciones se ejecutan continuamente, puede ocurrir que varios jugadores distintos ejecuten acciones *simultáneamente*, e incluso que dos acciones sean incompatibles, como que dos jugadores intenten construir dos edificios en el mismo lugar exactamente en el mismo instante. El juego debe implementar alguna política para resolver estos conflictos, como anular ambas acciones, o elegir al azar cuál se descarta.

También pueden existir acciones *no deterministas*, esto es, que no siempre obtengan el mismo resultado. Por ejemplo, el daño que hace un ataque de una tropa en cierto instante puede ser elegido pseudo-aleatoriamente dentro de un rango de valores.

El mapa suele estar cubierto por la *niebla de guerra*, que oculta a los jugadores todo lo que esté ocurriendo allí donde el rango de visión de sus unidades no llegue. Además de esto, el terreno suele ser desconocido al inicio de la partida, por lo que los jugadores deben dedicar algunas de sus unidades a explorar el entorno para cartografiarlo y encontrar recursos.

El gran número de unidades de todos los jugadores que puede haber en el tablero junto con la gran variedad de acciones que pueden tener lugar en cada instante de tiempo hacen que el espacio de estados de un juego RTS sea muchos órdenes de magnitud mayor que en cualquier juego por turnos y que el factor de ramificación de un árbol de búsqueda en este espacio sea inabarcable incluso para una máquina. La gran complejidad de estos juegos ha conseguido que sea un interesante campo para la investigación en inteligencia artificial, y que se hayan propuesto multitud de técnicas para tratarlo.

2.1.2. Desarrollo de una partida

En los juegos RTS, los jugadores normalmente tienen que desarrollar una economía para sustentarse y mejorar su capacidad para enfrentarse a sus rivales. Esto provoca que se puedan distinguir una serie de fases o etapas más o menos diferenciadas durante la partida.

Al inicio de la partida, el jugador tiene que conseguir los primeros recursos que encuentre en su entorno más cercano y usarlos para construir los edificios básicos y producir algunas unidades. En este punto, el jugador tiene acceso sólo a las unidades más básicas, con poco ataque, vida y velocidad de movimiento, y suele centrase en la producción de muchos trabajadores para obtener recursos más rápido y así tratar de conseguir una ventaja sobre el resto de jugadores.

En una segunda etapa tienen lugar los primeros contactos entre los jugadores como consecuencia de explorar regiones más alejadas para hacerse una idea de la ubicación de los enemigos y para encontrar nuevas fuentes de recursos antes de que se agoten las más cercanas. Se producen los primeros enfrentamientos militares, por lo que es necesario investigar mejoras que den a las tropas aliadas una ventaja. El jugador también debe conseguir mejoras económicas, para explotar sus recursos de forma más eficiente, encontrando un equilibrio entre economía y defensa.

Si el jugador resiste, llegará a una etapa de apogeo en la que sus tropas han alcanzado su máximo potencial gracias a las investigaciones y mejoras y la producción está en su máxima capacidad. Los jugadores han acumulado gran cantidad de recursos y han construido una ciudad con numerosos edificios capaces de producir grandes batallones rápidamente. En esta fase, la población de trabajadores disminuye en favor de una mayor cantidad de tropas.

Si la partida se alarga, los jugadores que hayan sobrevivido pueden entrar en una fase de decadencia, en la que los recursos se han agotado y no se pueden permitir perder las tropas que quedan. Las tropas más avanzadas son caras, por lo que el jugador se puede ver obligado a producir tropas peores pero más baratas para defenderse. Un jugador que haya adoptado una estrategia más defensiva y haya tratado de ahorrar parte de sus recursos puede esgrimir ahora cierta ventaja para atacar a sus enemigos, si es que ha conseguido resistir a sus ataques en su momento de máxima fuerza.

Por supuesto, estas fases son orientativas y la transición de una a otra es progresiva y no repentina, por lo que se pueden solapar en cierta medida. Además, muchos factores pueden influir en el desarrollo de una partida, por ejemplo, si el mapa consta de islas en la que cada jugador está separado de los otros, las primeras fases de la partida serán mucho más tranquilas, ya que los jugadores tienen que desarrollar tecnologías más avanzadas para mover sus tropas a través del mar. De este modo, los jugadores estarán mucho más desarrollados cuando ocurran las primeras batallas.

Otro factor muy influyente es la relación entre los jugadores durante la partida. Cuando hay más de dos jugadores, es posible que se establezcan alianzas entre ellos, permanentes o temporales, que pueden cambiar el transcurso del juego o la forma de enfrentarse a los rivales. Por ejemplo, dos jugadores pueden sincronizar sus ataques para hostigar a un tercero o compartir sus recursos para defenderse.

También la distancia entre los jugadores puede influir en el desarrollo de la partida. Si los jugadores están muy cerca, existe la posibilidad de hacer un ataque rápido con tropas básicas para tratar de derrotar al enemigo en una etapa inicial. Una estrategia así puede tener éxito si el enemigo ha descuidado su defensa al principio de la partida y las tropas del jugador lo atacan antes de estar preparado. En general, no hay una estrategia que sea siempre superior a las demás. En ciertas situaciones, una estrategia puede ser más adecuada que otra, pero en otras circunstancias puede ocurrir lo contrario.

2.2. Técnicas de inteligencia artificial aplicadas a RTS

Se han utilizado gran variedad de técnicas de inteligencia artificial para crear sistemas capaces de jugar a los juegos RTS. Como son un género de juego muy complejo, se pueden aplicar varias técnicas de IA en una misma partida para tomar las decisiones de distintos ámbitos. En líneas generales, podemos distinguir tres grandes ámbitos dentro de un juego RTS:

- Estrategia: consiste en el conjunto de decisiones a mayor escala dentro de la partida, que afectan a todas las unidades de un jugador. Por ejemplo, un jugador puede adoptar una estrategia agresiva o defensiva, o puede centrarse en la economía y la obtención de recursos al inicio de la partida, y tras un tiempo atacar al enemigo.
- Táctica: se refiere a cómo aplicar la estrategia en cada momento mediante el posicionamiento de las construcciones o los movimientos, entre otros. Se aplica a un grupo concreto de unidades. Por ejemplo, para aplicar una estrategia de obtención de recursos que permita a un jugador desarrollarse al inicio de la partida, es necesario poner a los trabajadores a recolectar los recursos y distribuir las tropas en grupos para cubrir a los trabajadores o defender yacimientos.
- Control reactivo: son las decisiones que afectan a una sola unidad, es decir, a pequeña escala. Abarcan cómo aplicar las tácticas, decidiendo, por ejemplo, cuándo disparar o huir en un combate.

Otra clasificación de los ámbitos del RTS muy usada entre jugadores profesionales es la que diferencia entre *microjuego* y *macrojuego*. El *micro* se refiere a las acciones a pequeña escala que buscan un objetivo inmediato en la partida, lo que esencialmente es el control individual de las unidades. El *macro* comprende las acciones y decisiones que buscan un objetivo a más largo plazo, por ejemplo, la planificación de la economía. Se puede establecer una correspondencia entre la división en tres ámbitos que se hace anteriormente (estrategia, táctica y control reactivo) y la división en *macro* y *micro*. El *macro* se corresponde a grandes rasgos con la estrategia y la parte de táctica que tiene efecto a medio o largo plazo (por ejemplo, la vigilancia de puntos clave y la exploración del mapa), mientras que el *micro* se corresponde con el control reactivo y la parte táctica con efecto a corto plazo (por ejemplo, el posicionamiento del ejército en una batalla).

Volviendo a la división en estrategia, táctica y control reactivo, pasamos ahora a resumir algunos de los planteamientos propuestos en cada uno de estos tres ámbitos para crear bots capaces de jugar a juegos RTS, siguiendo la recopilación realizada por Ontañón y otros autores en [1].

2.2.1. Estrategia

Dentro de este ámbito existen muchos problemas que se han de resolver durante una partida. En general, se debe utilizar la información disponible sobre el mapa para tratar de predecir el comportamiento de los enemigos. En base a esa información, hay que planificar el desarrollo de la economía, es decir, elegir el orden en que se construyen los edificios y se investigan las mejoras, y decidir la composición del ejército.

La primera aproximación es usar estrategias precodificadas o *hard-coded* que guíen cómo debe actuar la IA a lo largo de la partida. Tienen la ventaja de que son sencillas y permiten incorporar conocimiento experto previo sobre el dominio del juego. Sin embargo, tienen poca capacidad de adaptación a distintos escenarios, aspecto en el que son superadas por técnicas de aprendizaje automático. Suelen consistir en máquinas de estados finitos [7], con estados que representan formas de actuar muy generales como atacar o recolectar recursos.

Otra opción es el razonamiento basado en casos, que se ha aplicado para secuencias de construcciones [8], para modelos de comportamiento de los oponentes [9], para selección automática de objetivos y para construcción del ejército en base al del enemigo [10].

El aprendizaje automático ha sido muy explorado en este ámbito, pudiendo mencionarse el uso de minado de datos y redes neuronales en el juego *StarCraft* para predecir estrategias enemigas [11] y aprender secuencias de orden de construcciones. Encontramos también modelos bayesianos para aprendizaje a partir de repeticiones de partidas previas y predicción durante la partida [12]. Además, se han usado algoritmos evolutivos para tratar de determinar la prioridad de las tareas de alto nivel de la partida [13].

2.2.2. Tácticas

Dentro de esta categoría incluimos el razonamiento espacial o análisis del terreno, para el que se han empleado técnicas como mapas de influencia y áreas con información de conectividad [14]. También se ha estudiado la importancia de tener información cualitativa acerca del tablero (posición de muros y otros elementos) para la búsqueda de caminos (*pathfinding*) y el análisis geométrico [15]. Se han utilizado además mallas generadas a partir de una semilla para encontrar regiones convexas del mapa [16], así como descomposición de Voronoi (*clustering*) para detectar las zonas clave [17]. Finalmente, otro aspecto relacionado con el razonamiento espacial que se ha estudiado es el uso de edificios como muralla para bloquear el paso de las tropas enemigas [18].

En cuanto a la toma de decisiones a nivel táctico, se han probado diversas técnicas de aprendizaje automático y árboles de búsqueda. Una de ellas son los modelos ocultos de Markov, empleados para rastrear unidades y para tratar de identificar lo que planea el jugador enemigo [19]. Otro ejemplo es la combinación de razonamiento basado en casos con aprendizaje por refuerzo para reutilizar partes de estrategias [20]. También se ha probado la idea de dividir el tablero en regiones abstractas a las que se asigna una puntuación según su importancia a nivel económico o militar, en función de lo aprendido en repeticiones de jugadores profesionales [21]. Finalmente, se ha propuesto el uso de algoritmos evolutivos para aprender a tomar decisiones en función de la situación del tablero [22].

Por último, en el uso de árboles de búsqueda destacan el algoritmo ABCD (*Alpha-Beta Considering Durations*) [23] o los árboles de búsqueda de Monte-Carlo (MCTS) [24]. Es necesario aplicar algún tipo de abstracción sobre la representación del estado del juego para reducir la complejidad y que el árbol de búsqueda sea factible computacionalmente. También es posible la simplificación de las acciones en lugar del estado del juego, utilizando un conjunto predefinido y limitado de *scripts* de acciones de alto nivel a explorar en el árbol.

2.2.3. Control reactivo

Para abordar el control reactivo de unidades se han empleado campos potenciales (*potential fields*), concretamente para evitar obstáculos y ataques enemigos [25], así como mapas de influencia, para buscar movimientos con los que flanquear al enemigo [26]. Estas técnicas tienen la desventaja de que requieren ajustar gran cantidad de parámetros, por lo que se han combinado con técnicas como el aprendizaje por refuerzo para encontrar configuraciones óptimas automáticamente.

El aprendizaje automático también forma parte de las técnicas empleadas en este campo, por ejemplo, se han empleado modelos bayesianos en combinación con campos potenciales para conseguir que las decisiones de control reactivo también conduzcan a cumplir objetivos tácticos [27], además de algoritmos de aprendizaje por refuerzo para control reactivo descentralizado, es decir, en el que se toman decisiones para cada unidad por separado [28]. Debido a la enorme cantidad de acciones posibles a explorar, se ha propuesto suministrar algo de conocimiento previo sobre el dominio a los algoritmos de aprendizaje para acelerarlo [29]. También encontramos trabajos que utilizan algoritmos evolutivos para determinar los mejores parámetros de control reactivo [30], por ejemplo para una batalla. Por último, quedan por mencionar los árboles de búsqueda, como la búsqueda alfa-beta aplicada al control de unidades [23].

Para acabar con la sección, mencionamos las técnicas de búsqueda de caminos o *pathfinding*, ya que se suele aplicar a nivel de unidad, aunque el movimiento y distribución de unidades fuera de combate encaja más dentro de la categoría de tácticas. El algoritmo de *pathfinding* dominante es A*, pero requiere gran cantidad

de recursos de procesador y memoria. Por ello, se puede integrar con técnicas de simplificación del mapa, como dividirlo en cuadrantes, o agrupar unidades para moverlas a todas a la vez [31].

2.3. Algoritmos genéticos

De entre todas las técnicas descritas, vamos a desarrollar un poco más la de algoritmos genéticos, ya que la utilizaremos en nuestros experimentos en el capítulo 5.

En este tipo de algoritmos se busca resolver un problema aplicando el mismo proceso que guía la evolución de los seres vivos. Por lo general, estos algoritmos se aplican para resolver problemas en los que se busca una configuración óptima de una serie de parámetros. Así, cada posible configuración se denomina *individuo*, y se utiliza una población de individuos que se van reproduciendo y evolucionando generación a generación para alcanzar una solución lo mejor posible. Cada parámetro se denomina, con una nomenclatura inspirada en la evolución natural, *gen*, mientras que el conjunto de todos ellos se denomina *cromosoma*.

Un algoritmo genético estándar comienza con la elección de la población inicial. Por lo general, esta población inicial se elige de forma aleatoria, de forma que se van tomando valores aleatorios para cada uno de los genes de cada individuo. También se puede usar una población inicial predeterminada, pero no es lo habitual.

La población inicial constituye la primera generación del algoritmo, que como hemos mencionado antes, va iterando generación a generación en imitación de lo que sucede con una población de individuos de una cierta especie en la naturaleza. En cada una de las generaciones se realiza el siguiente proceso:

- Evaluación: se calcula la puntuación de cada individuo de la población en base a una cierta función de evaluación o *fitness* definida. Esta función de *fitness* debe representar cómo de bueno es el individuo si lo tomamos como solución al problema que pretendemos resolver, de forma que sea mayor cuanto mejor es dicho individuo.
- Elitismo: se toman los individuos con mayor puntuación de *fitness* y se pasan directamente a la siguiente generación. La cantidad de individuos tomados para elitismo debe ser siempre menor que el tamaño de la población, ya que si no el algoritmo no avanzaría.
- Selección: se eligen los individuos que serán los progenitores de la siguiente generación. Si la cantidad de individuos elegidos por el elitismo es cero, este paso puede parecer trivial, ya que cada pareja genera dos hijos y se necesita que la siguiente generación tenga el mismo número de individuos que la actual. Sin embargo esto no es así, dado que existe la posibilidad de que algunos individuos entren varias veces a la lista de progenitores mientras que otros no

sean elegidos. Lo más habitual para que la elección de estos progenitores esté basada en la evaluación y que los individuos mejores tengan mayor probabilidad de reproducirse es organizar un torneo para escoger cada progenitor. Un *torneo* consiste en tomar cierta cantidad de individuos de forma aleatoria, con la posibilidad de que estos estén repetidos, y se considera ganador del torneo al individuo con mayor *fitness*. El ganador de cada torneo sería el que entraría en la lista de progenitores, haciendo que los individuos con mayor *fitness* tengan mayor probabilidad de entrar pero sin eliminar la posibilidad de que los peores sean elegidos.

- Cruce: se toman dos individuos aleatorios de la lista de progenitores procedente de la selección y se les aplica la función de cruce elegida. La función de cruce puede variar según el problema que se pretenda resolver, pero debe mezclar de alguna forma los dos progenitores para obtener dos descendientes. Una de las funciones de cruce más comunes es el llamado cruce de un punto. En éste, se elige un gen del cromosoma y se intercambian todos los genes posteriores.

- Mutación: se recorren los genes de los individuos obtenidos mediante el cruce y se cambian por otro gen aleatoriamente con una cierta probabilidad.

Este proceso se repite una cantidad de veces fijada de antemano, o hasta que la diferencia entre el *fitness* de una generación y la anterior es inferior a cierto umbral.

Los algoritmos genéticos se utilizan para explorar el espacio de soluciones formado por todas las posibles configuraciones de los cromosomas buscando la que maximiza la función de *fitness*. Se basan en que al cruzar dos individuos con una buena puntuación se obtendrá la mayoría de las veces otro con una buena puntuación, de forma que el cruce va guiando la búsqueda hacia los máximos. En el caso de que esto no se cumpla, puede que el algoritmo genético funcione algo peor, aunque se pueden buscar alternativas para resolver este problema. Por otro lado, para evitar que el algoritmo se quede en un máximo local, se permite que los individuos con un *fitness* bajo también se crucen, de forma que también se puede empeorar ligeramente al pasar a la siguiente generación. La mutación ofrece también una ayuda para salir de los máximos locales, ya que hace que los individuos puedan pasar rápidamente de un punto del espacio de búsqueda a otro quizá muy alejado.

Conclusiones

En este capítulo hemos visto un resumen sobre los juegos RTS y las técnicas que se han empleado hasta ahora para intentar crear bots capaces de jugar a este tipo de juegos.

En capítulo 3, pasamos a explicar en profundidad μ-RTS, un juego sencillo desarrollado íntegramente para probar la aplicación de distintas técnicas de inteligencia artificial a los juegos RTS, que luego podrían exportarse a juegos más complejos.

Capítulo 3

Descripción del juego μ-RTS

El juego μ-RTS [32] es un juego de RTS extremadamente simplificado para dos jugadores, programado en Java y utilizado para investigar sobre la aplicación de técnicas de aprendizaje automático a esta categoría de juegos. Su baja cantidad de unidades y su sencillez en cuanto a objetivo y tablero hacen que sea muy útil para probar diversas técnicas que posteriormente podrían exportarse a juegos más complejos si los resultados son buenos y se desea continuar esa línea de investigación. Además, como el juego pretende ser un banco de pruebas, permite la configuración de varios parámetros clave. Estas configuraciones, que se irán explicando posteriormente a medida que se vayan presentando, se fijan al principio de la partida y se mantienen a lo largo de ésta.

3.1. Características generales

Tal y como se explicó en el capítulo anterior, los juegos de RTS se caracterizan por la ausencia de turnos entre los jugadores, esto es, todos los jugadores pueden realizar acciones en cualquier momento sin tener que esperar a que el otro jugador actúe (como sí sucede en otros juegos por turnos como el ajedrez, por ejemplo). En una ejecución en un ordenador el tiempo no es continuo, por lo que es necesario buscar una forma de simular esta ejecución de acciones "en cualquier momento". Para ello, el juego discretiza el tiempo internamente en ciclos, en los que pide acciones a ambos jugadores. Por el mismo motivo, las acciones dadas en un mismo ciclo se ejecutan en un orden determinado, pero como son durativas se entrelazan a lo largo de los ciclos siguientes, de forma que finalmente el resultado es como si se hubieran realizado "a la vez".

En el caso de que dos acciones contradictorias tengan efecto a la vez, μ-RTS ofrece varias estrategias de resolución configurables. Por un lado se tiene la opción de cancelación aleatoria, que cancela una de las dos acciones de forma aleatoria, lo cual hace que el juego no sea determinista. Por otro lado, se ofrecen las opciones de

cancelar ambas acciones o la de ir cancelándolas de forma alternante. Es importante notar que, si se opta por cancelar ambas acciones, se pueden producir situaciones en las que ciclo tras ciclo dos unidades tratan de moverse a la misma casilla y se cancelan las acciones una y otra vez.

Respecto a las unidades, también existen varias configuraciones. Nuevamente, una de ellas es no determinista, de forma que el daño causado por las tropas varía de forma aleatoria dentro de unos límites. Las otras dos, llamadas *versión original* y *versión afinada*, sí son deterministas y varían ligeramente en algunas características de las unidades, como el daño que hacen al atacar o el tiempo de producción. Estas características se explicarán más detenidamente en la sección 3.3.

Finalmente, ofrece la opción de jugar o no con *niebla de guerra*, que como se explicó en el capítulo anterior consiste en ocultar parcialmente el tablero a los jugadores, de forma que sólo ven lo que las tropas en el juego podrían ver.

Además de utilizarse para la investigación, μ-RTS se usa también para realizar una de las competiciones de la Conference On Games (CoG) organizada por el IEEE todos los años desde el 2017. En esta competición [33], se conceden $100\,ms$ en cada ciclo a cada uno de los bots para tomar su decisión, y en caso de que no respondan en ese tiempo, se interpreta que no hacen nada que no estuvieran haciendo en el ciclo anterior. Además, también se concede una hora a cada bot por cada mapa para hacer posibles pruebas previas y guardar los resultados, facilitando así el uso de técnicas que requieran un estudio previo del tablero. En cuanto a las configuraciones de las que hablábamos anteriormente, en el concurso se utiliza la configuración afinada para las unidades y la estrategia de resolución de conflictos en la que se cancelan las dos acciones. Se realiza, no obstante, una versión de la competición con visión absoluta del tablero y otra con *niebla de guerra*.

3.2. Objetivo del juego

En la mayoría de juegos de RTS, existen varias condiciones de victoria que pueden cambiar según la partida o el criterio de los jugadores. En el caso de μ-RTS, la única condición de victoria que hay es la destrucción total del adversario, esto es, destruir todas las unidades enemigas. Es especialmente importante porque en caso de que se acabe el tiempo de juego no se hace distinción entre el número de tropas que le queden a cada uno o de que tipo sean éstas, lo único que se tiene en cuenta es si alguno de los jugadores no tiene ninguna unidad. Otro caso en el que es importante que la condición de victoria sea esta es, por ejemplo, aquel en el que un jugador sólo tiene una unidad que no puede hacer nada, por ejemplo una base en el caso de que no le queden recursos. Pese a que este jugador no puede actuar, si el enemigo no destruye esta unidad antes del límite de tiempo, el resultado de la partida sería de empate. Lo mismo puede suceder si a un jugador sólo le queda una tropa con vida y todas las demás han sido destruidas, incluida la base. En algunos juegos de RTS

la destrucción de la base podría suponer directamente la derrota, sin embargo, en el caso de μ-RTS, si la única tropa que le queda al jugador consiguiera destruir todas las unidades enemigas, se alzaría con la victoria.

3.3. Unidades controladas por los jugadores

Además del bloque de recursos (u oro), del que hablaremos en la sección 3.4, se tienen otros seis tipos de unidades con las que cada jugador desarrolla la partida. Estas unidades se pueden dividir en dos grupos. Por un lado están los edificios, que no pueden moverse ni atacar, y cuya única función es crear otras unidades. Hay dos tipos de edificios: la base y el cuartel. Por otro lado se tienen las tropas, que pueden atacar y desplazarse por el tablero y normalmente no pueden crear otras unidades. Hay cuatro tipos de tropas: el trabajador, el soldado ligero, el soldado pesado y el soldado a rango. Anteriormente mencionamos que las tropas tienen una serie de características configurables. Pasamos ahora a detallar cuáles son estas características para luego mostrar en la tabla 3.1 el valor que tiene cada unidad respecto a cada característica.

- Coste: el número de unidades de oro que cuesta producir la unidad en cuestión.
- Salud: la cantidad de daño que puede resistir la unidad antes de ser destruida.
- Daño: los puntos de salud que la unidad le inflige a otra cuando ataca.
- Rango: la distancia, tomando ésta como distancia euclídea, a la que la unidad puede atacar a otra.
- Tiempo de producción: el número de ciclos que tarda la unidad en ser producida.
- Tiempo de movimiento: el número de ciclos que tarda la unidad en moverse a una casilla adyacente.
- Tiempo de ataque: el número de ciclos que tarda la unidad en atacar a otra situada dentro de su rango.

Unidad	Coste	Salud	Daño	Rango	T. prod.	T. mov.	T. ata.
Base	10	10	-	-	200	-	-
Cuartel	5	4	-	-	100	-	-
Trabajador	1	1	1	1	50	10	5
Soldado ligero	2	4	2	1	80	8	5
Soldado pesado	3	8	4	1	120	10	5
Soldado a rango	2	1	1	3	100	10	5

Tabla 3.1: Valor de las características para cada tipo de unidad.

Se pasa ahora a explicar brevemente las conclusiones que se pueden sacar de la tabla, así como algunas características cualitativas de cada unidad.

- Base: en este edificio es donde se almacena el oro recolectado. Su única función aparte de ésta es la de producir trabajadores. Puesto que es, con diferencia, la unidad más cara del juego y el tiempo de producción de los trabajadores es relativamente rápido, no es frecuente tener más de una base.
- Cuartel: en este edificio se producen las tropas militares, es decir, todas las tropas a excepción de los trabajadores. Son la segunda unidad más cara del juego y no es frecuente construir más de uno, aunque según la estrategia que se siga se podría hacer para producir una mayor cantidad de tropas en menos tiempo.
- Trabajador: es la tropa más básica y débil, pero presenta la ventaja de ser también muy barata. Además, es la única unidad que puede recolectar oro, lo que la convierte en una pieza central en la partida para evitar el desabastecimiento. Aunque a veces se puede emplear para atacar, sobre todo en una estrategia que consista en abrumar al enemigo con un ataque rápido y numeroso, normalmente se utiliza para recoger oro de los bloques de recursos y transportarlo hasta la base. Para extraer los recursos de forma eficiente, las bases suelen construirse cerca de algún bloque de recursos, lo cual puede reducir el espacio en torno a estos y limitar la libertad de movimiento de las unidades. En general, no es recomendable tener más de dos o tres trabajadores recolectando el mismo yacimiento por la facilidad con que pueden obstruirse el camino entre ellos.
- Soldado ligero: se trata de una tropa a melé muy útil por ser la que menos tarda en producirse y en desplazarse de entre los soldados. Además, aunque es superada por el soldado pesado en el ataque y salud, la diferencia en velocidad de producción, de movimiento y en coste hacen que sea utilizada con frecuencia.
- Soldado pesado: esta tropa tiene una gran cantidad de salud y de daño. Esto obliga a que sean necesarias varias tropas enemigas para vencerla, ya que con un solo ataque puede eliminar a cualquier otra unidad con excepción de la base. Sin embargo, su alto tiempo de producción y movimiento y su coste permiten al jugador contrario producir una defensa fuerte antes de que una de estas unidades pueda llegar a atacarle.
- Soldado a rango: esta tropa es extremadamente débil en comparación con los otros dos soldados, y tampoco destaca su tiempo de producción ni de desplazamiento. Sin embargo, tiene la gran ventaja de ser la única unidad que puede atacar sin acercarse, ya que su rango es de tres casillas. Esto la convierte también en la única unidad capaz de atacar a través de obstáculos como pueden ser muros, bloques de recursos u otras unidades aliadas o enemigas.

La configuración de las tropas que se muestra en la tabla 3.1 es la versión afinada, que es la que se utiliza en el concurso de la CoG.

3.4. Tablero

El tablero de μ-RTS consiste en una cuadrícula de tamaño arbitrario sobre la que se sitúan las unidades y que representa el mapa de juego. Al contrario que en otros juegos de RTS, no hay distintos tipos de terreno ni agua, lo cual simplifica mucho el mapa y evita la necesidad de unidades específicas como los barcos. No obstante, para que el mapa no esté completamente vacío, existen muros cuya única función es impedir el paso de las unidades por aquellas casillas en las que están situados. En cada casilla en la que no haya un muro sólo puede haber una unidad. Aquellas unidades que tienen la capacidad de moverse pueden pasar de una casilla a otra adyacente en las cuatro direcciones cardinales siempre y cuando la nueva casilla esté libre. El tiempo necesario para hacer esta acción depende de cada unidad, como se detalló en la sección 3.3. Por último, es necesario hablar aquí de un tipo particular de unidad que no pertenece a ninguno de los jugadores y que se podría considerar parte del tablero. Esta unidad es el bloque de recursos, que consiste en una casilla especial con una cantidad determinada de recursos que los trabajadores pertenecientes a ambos jugadores pueden extraer. Hay que destacar que sólo existe un tipo de recurso a extraer, al contrario que en otros juegos de RTS donde suele haber más (por ejemplo comida, metal, piedra, madera,...) y puesto que este único recurso se utiliza tanto para construir edificios como para crear tropas, de ahora en adelante lo llamaremos oro.

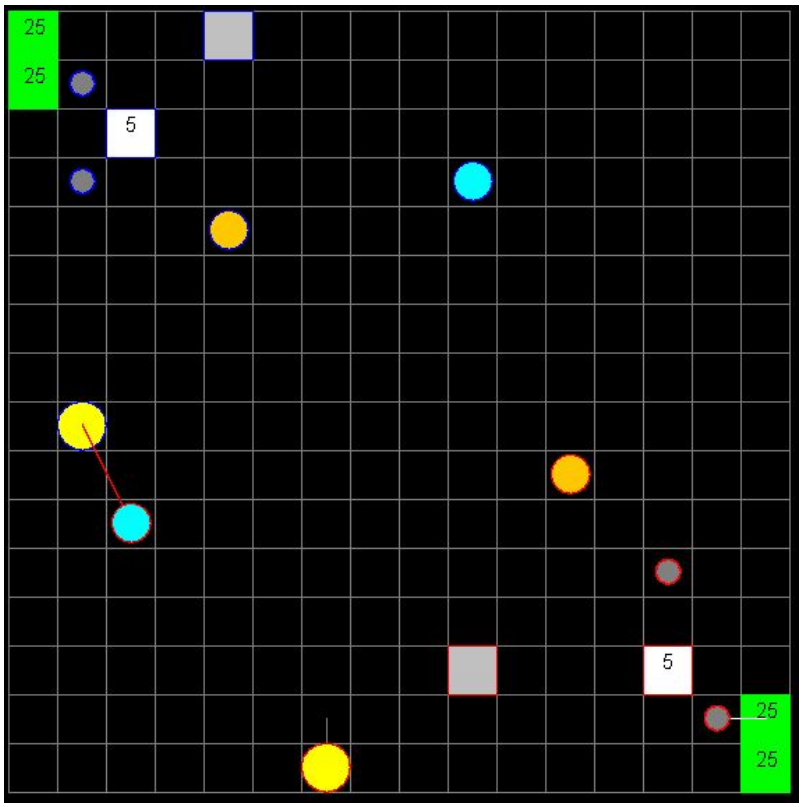

Figura 3.1: Tablero de μ-RTS durante una partida.

En la figura 3.1 se muestra un ejemplo del tablero en un cierto instante de tiempo durante una partida de μ-RTS. Por lo general, el tablero es simétrico para que la situación sea lo más justa posible para ambos jugadores. Las casillas verdes son los bloques de recursos, y el número escrito sobre ellas indica cuántas unidades de oro se pueden recolectar en ellas. Como se observa, se encuentran repartidas en la esquina noroeste y la esquina suroeste. En estas dos esquinas es donde están colocadas, en la mayoría de tableros, las primeras unidades de cada jugador. Por lo general, el jugador 1, cuyas unidades se muestran bordeadas en rojo, empieza en la esquina sudeste y el jugador 2, cuyas unidades se muestran bordeadas en azul, en la noroeste. En cuanto a estas unidades, vienen representadas por un código de formas y colores. Así, las que tienen forma cuadrada son edificios, siendo la de color blanco la base y la de color gris el cuartel. El número que se muestra sobre la base indica la cantidad de recursos que tiene el jugador correspondiente. Por su parte, las unidades con forma redonda son las tropas. Las pequeñas de color gris son los trabajadores, las naranjas los soldados ligeros, las amarillas los soldados pesados y las azules los soldados a rango. A la vista de las dependencias entre la creación de los distintos tipos de unidades, explicadas en la sección 3.3, es claro que al comienzo de la partida cada jugador debe tener una de las siguientes configuraciones mínimas de unidades: una base y al menos una unidad de oro con la que producir el primer trabajador, o un trabajador y al menos diez unidades de oro con las que construir una base. Lo más habitual es contar con una mezcla de ambas configuraciones y tener tanto un trabajador como una base con una cantidad variable de oro ya almacenada.

3.5. Desarrollo de la partida

Los juegos RTS suelen tener varios tipos de unidades con distintos niveles de mejora, además de investigaciones que dan a los jugadores bonificaciones de ataque, resistencia o eficiencia para gestionar los recursos. Además, suele existir un sistema similar al juego *piedra, papel o tijeras*, en el que se da ventaja a ciertos tipos de unidades al enfrentarse a otros tipos. Por ejemplo, los arqueros pueden tener bonificación de ataque sobre la infantería, la infantería sobre la caballería, y la caballería a su vez sobre los arqueros. Además existen otras unidades como curanderos, que restauran vida, o maquinaria de asedio, que hace más daño a los edificios. Todas estas reglas constituyen variables que incrementan enormemente la complejidad de este tipo de juegos y que μ-RTS no tiene. En μ-RTS no hay mejoras, ni investigaciones y tan sólo existen seis tipos de unidades, incluyendo edificios, que tienen estadísticas básicas que no se alteran por ninguna bonificación. Por tanto, μ-RTS es un juego extremadamente simplificado dentro del género RTS. Aún así, la complejidad de μ-RTS a nivel computacional sigue siendo muy alta, ya que la gran variedad de acciones posibles para todas las unidades sobre el tablero provocan que el espacio de estados del juego sea enorme y que un árbol de búsqueda en ese espacio tenga un gran factor de ramificación.

Como se describió en la sección 2.1.2, en la mayor parte de juegos RTS la partida se puede dividir en varias fases a lo largo de las cuales ambos jugadores se van desarrollando. A grandes rasgos, lo habitual es comenzar recolectando recursos para crear tropas, realizar investigaciones para desbloquear nuevas tropas o mejorar las ya existentes y construir edificios. Después, una vez que el jugador ya tiene un gran ejército y se ve capacitado para derrotar al oponente, es cuando pasa a la fase de ataque y trata de vencerlo. Por supuesto, estas fases son sólo lo habitual y no tienen por qué ser fijas. Por ejemplo, un jugador podría tratar de atacar nada más empezar con unas pocas tropas básicas para coger desprevenido al rival o impedirle desarrollarse. El tamaño del tablero también influye en estas fases. En un tablero pequeño, un ataque rápido tendría mayor probabilidad de éxito que en uno grande. La razón es que en un tablero de gran tamaño, las tropas tardan más tiempo en llegar hasta el rival, con lo que éste tendría más tiempo para desarrollar tropas mejores capaces de vencer a las atacantes.

Como en μ-RTS no existen investigaciones ni mejoras para las tropas, y hay una cantidad muy pequeña de tipos de edificios y unidades, la división en fases de la partida se vuelve muy sencilla. Se pueden distinguir principalmente dos fases: primero una fase en la que se recolectan recursos y se crean tropas y después otra fase en la que se ataca. La duración de estas fases, dada la sencillez del juego, depende únicamente del tamaño del tablero. Un tablero más pequeño da pie a una primera fase más corta, o quizá incluso inexistente, por el mismo motivo que la estrategia alternativa que poníamos de ejemplo en el párrafo anterior podría funcionar en un tablero pequeño. Cuando la distancia que separa a los dos jugadores es muy corta, atacar rápidamente suele ser la mejor opción porque el rival no tendrá tiempo de preparar tropas más poderosas mientras llegamos. Además, en caso de no hacerlo y tratar de comenzar con una primera fase de recolección, nos exponemos a ser nosotros los atacados antes de estar listos. Si el tablero es más grande, nuevamente sí que tiene sentido dividir la partida en fases y tratar de generar tropas más fuertes antes de atacar, ya que en el tiempo que tardamos en llegar hasta el rival éste tendrá tiempo de crear soldados fuertes con los que defenderse de nuestro ataque. En el capítulo 4 se analizarán los resultados obtenidos con bots que juegan asumiendo la existencia de una única fase, mientras que en el 5, se hará lo propio con bots que dividen el juego en múltiples fases.

3.6. Ciclo de juego

Como ya se ha mencionado, el sistema utilizado por μ-RTS para simular un desarrollo de la partida en tiempo real es el de realizar ciclos muy cortos en los que se piden acciones a ambos jugadores, y hacer que estas acciones puedan tardar varios ciclos en completarse. Una partida, por tanto, se desarrolla a base de ejecutar en bucle ciclos de juego hasta que uno de los dos jugadores gane o hasta que se haya jugado un cierto máximo de ciclos establecido.

Cada uno de estos ciclos realiza el siguiente proceso:

- Pedir al jugador 1 el conjunto de acciones que quiere realizar para cada una de sus unidades.
- Pedir al jugador 2 el conjunto de acciones que quiere realizar para cada una de sus unidades.
- Procesar las acciones dadas por el jugador 1 para convertirlas en acciones sobre el tablero.
- Procesar las acciones dadas por el jugador 2 para convertirlas en acciones sobre el tablero.
- Aplicar las acciones al juego en el orden en que se han procesado para completar el ciclo.

La forma en que está implementada la ejecución de las acciones durativas hace que su ejecución sea justa y no se produzcan cancelaciones indebidas. Esto es, si dos acciones se ejecutan exactamente en el mismo ciclo, por ejemplo, dos tropas atacándose mutuamente, ambas se llegan a ejecutar en cualquier caso, y aunque una se aplique antes y mate a la otra, recibirá igualmente el daño del ataque enemigo. Es importante destacar esto porque, de no ser así, el jugador 1 tendría una gran ventaja porque sus acciones siempre se procesan, y por tanto se aplican, en primer lugar.

3.7. Desarrollo de bots en μ-RTS

Uno de los aspectos en los que más destaca μ-RTS es en la facilidad con la que se pueden crear y acoplar nuevos bots. Tiene sentido que esto sea así, ya que la finalidad principal de μ-RTS es, precisamente, la de facilitar la investigación de diferentes técnicas de inteligencia artificial aplicadas al desarrollo de bots para juegos RTS.

El juego está programado en Java, y el código es público y puede ser descargado por cualquiera desde su repositorio de Github [32]. El proceso para crear un bot y jugar a μ-RTS con él es tan sencillo como crear las clases correspondientes en Java y ejecutarlas junto con el juego.

Para crear un nuevo bot, es obligatorio extender la clase `AI`, ya que ésta indica las funciones que se deben implementar para poder jugar, y es a la que μ-RTS llamará. La más importante de estas funciones es `getAction()`, que es la función que invoca el juego para preguntar al bot qué quiere hacer en cada ciclo. También se ofrecen funciones auxiliares que se pueden sobreescribir de forma opcional, como `preGameAnalysis()`, que da la oportunidad a los bots de hacer un procesamiento previo del tablero en caso de que lo requieran.

No obstante, crear un bot como se ha explicado arriba resulta algo enrevesado, ya que implementar incluso las acciones más sencillas requiere bucear en el código del juego y utilizar una gran cantidad de clases distintas. Por ejemplo, para atacar a otra unidad, se necesitaría buscar la ruta hasta la unidad objetivo e ir dando las acciones individuales para moverse casilla a casilla y, finalmente, atacar una vez esté suficientemente cerca. Para ahorrar todo este trabajo al programador y que éste pueda centrar sus esfuerzos en el funcionamiento de la inteligencia artificial en sí, se tiene la clase `AbstractionLayerAI`. Esta clase hereda de `AI` y ofrece un paso más de abstracción con un array de las acciones que se quieren realizar en un cierto ciclo y funciones para añadir dichas acciones a este array fácilmente. Estas funciones resultan intuitivas y sencillas de utilizar, y hacen que no sea necesario aprender todos los detalles de las distintas clases que el juego utiliza para cada tipo de acción. Así, por ejemplo, para atacar a otra unidad, basta llamar a la función `attack()` dándole la unidad con la que vamos a atacar y nuestro objetivo, y la propia función se encarga de atacar directamente si está suficientemente cerca o de calcular la ruta para llegar hasta nuestro objetivo y mover la unidad en caso contrario. Además, la clase `AbstractionLayerAI` hace que sólo sea necesario indicar acciones para las tropas que se desea utilizar, asignando de forma automática acciones vacías para aquellas tropas sobre las que no se ha dado ninguna indicación.

Sin embargo, usar la clase `AbstractionLayerAI` presenta también una desventaja, y es que estas funciones de las que hablábamos no tienen en cuenta la posición del jugador para el que son las acciones. Recordemos que, como se comentó en la sección 3.4, el tablero de μ-RTS suele ser simétrico y los jugadores suelen comenzar en dos esquinas opuestas, por lo general la noroeste y la sudeste. El problema de la clase `AbstractionLayerAI` es que el orden en que explora las casillas circundantes a una unidad a la hora de, por ejemplo, crear otra unidad, es siempre el mismo: norte, este, sur, oeste. Esto hace que las acciones puedan ser más beneficiosas para uno de los jugadores que para el otro.

Debido a esto, en los experimentos realizados en los capítulos 4 y 5 repetiremos todas las partidas actuando tanto como el jugador 1 como el 2, para así diluir la posible desventaja.

3.8. Bots predefinidos en μ-RTS

Dentro del código del juego se incluyen también algunos bots relativamente sencillos ya programados. A continuación se explica el comportamiento de algunos de estos bots, que utilizaremos en los capítulos 4 y 5 para nuestros experimentos.

En primer lugar, se tiene el grupo de bots *rush*. Estos bots basan su estrategia en ir creando unidades de un cierto tipo y atacar con ellas en cuanto son producidas. Utilizaremos cuatro bots distintos de este tipo, *LightRush*, que genera soldados

ligeros, *HeavyRush*, que genera soldados pesados, *RangedRush*, que genera soldados a rango, y *WorkerRushPlusPlus*, que ataca directamente con trabajadores y tiene la peculiaridad de fijar primero como objetivos las bases y cuarteles enemigos.

Por otro lado se encuentra el grupo de bots *defense*. Estos bots también basan su estrategia en un único tipo de tropa, pero en lugar de atacar con ellas forman una línea defensiva a una cierta distancia de la base. Cuando las unidades están colocadas en sus posiciones, permanecen estáticas a no ser que alguna tropa enemiga se acerque a ellas, en cuyo caso todas las unidades atacan a dicha tropa y, una vez la eliminan, vuelven a pararse hasta que otro enemigo esté cerca. Es importante notar que, si el enemigo no ataca, esta estrategia nunca llegaría a ganar, ya que las tropas permanecen estáticas hasta que se acerca algún enemigo. También utilizaremos cuatro bots distintos de este tipo, *LightDefense*, que defiende con soldados ligeros, *HeavyDefense*, que lo hace con pesados, *RangedDefense*, que utiliza soldados a rango, y *WorkerDefense*, que construye la línea defensiva directamente con trabajadores.

Se tiene también un grupo de bots algo más complejos, los de tipo *economy*. Estos bots tratan de construir una cantidad cada vez mayor de bases y cuarteles a la vez que entrenan trabajadores y soldados, de forma que van desarrollando cada vez más dominio sobre el tablero. La forma más sencilla de explicar su funcionamiento es hacerlo por separado para cada tipo de unidad. Por un lado, las bases generan trabajadores hasta que hay N por cada base construida. Esta N varía según el bot, y es la principal diferencia entre todos los que forman este grupo. En cuanto a los trabajadores, construyen un primer cuartel y, una vez éste ha generado al menos dos unidades, siguen construyendo cuarteles y bases si tienen recursos suficientes. Por su parte, los cuarteles producen un soldado ligero, uno a rango y uno pesado, en ese orden, y mientras siga habiendo uno de cada tipo producen más soldados de forma aleatoria. Finalmente, las tropas militares de todos los tipos atacan directamente al enemigo. Los bots de este grupo que utilizaremos son *SimpleEconomyRush*, con $N = 3$, *EconomyRush*, con $N = 4$, *EconomyRushBurster*, con $N = 6$, y *EconomyMilitaryRush*, con $N = 8$. Se tiene un último bot, *EMRDeterministico*, también con $N = 6$, que se diferencia de los demás en que los cuarteles, en lugar de producir tropas de forma aleatoria una vez se tiene una de cada tipo, siguen produciéndolas en el orden inicial: ligero, a rango, pesado.

Por último, se tienen dos bots aleatorios: *RandomAI* y *RandomBiasedAI*. Ambos realizan acciones completamente aleatorias, pero se diferencian en que el segundo tiene mayor probabilidad de que la acción escogida sea atacar o recolectar. De este grupo solo utilizaremos el último.

Conclusiones

A lo largo de este capítulo hemos visto cómo funciona el juego μ-RTS y todas las herramientas que pone al alcance de los usuarios para que estos puedan desarrollar y probar sus propios bots.

En el siguiente capítulo, pasaremos a experimentar con estas herramientas para intentar desarrollar un bot sencillo a la vez que competente en el juego. Asimismo, probaremos a enfrentar nuestro bot tanto con los expuestos en la sección 3.8, con el fin de comprobar cómo se desenvuelve contra estos jugadores sencillos, como con otros bots más complejos que se han presentado al concurso de la CoG.

Capítulo 4

Jugando a μ-RTS con una sola estrategia

En este capítulo se describe el desarrollo y prueba de varios bots capaces de jugar a μ-RTS. Todos ellos tienen una característica común: utilizan un único estilo de juego que no varía a lo largo de la partida. Por lo tanto, consideran que el juego ocurre en una única fase, y no en varias como ocurre en muchos RTS (sección 2.1.2). Utilizar siempre la misma estrategia supone una limitación para estos bots, especialmente en partidas largas, pero nos sirven como punto de partida para las versiones más elaboradas que se describirán en el capítulo 5.

Para determinar el estilo de juego que utilizará el bot, se elige un comportamiento para cada uno de los seis tipos de unidad que existen en μ- RTS: Base, Cuartel, Trabajador, Ligero, Pesado y A rango. Un comportamiento para cierto tipo de unidad consiste en una función que, a partir del estado de juego, devuelve la acción a realizar. Recuperando la descomposición en ámbitos del juego que hicimos en la sección 2.2, podemos considerar que la elección de los seis comportamientos determina la estrategia con la que va a jugar el bot, y que los comportamientos implementan las tácticas que deben emplear las unidades para aplicar esa estrategia.

La estrategia utilizada por el bot permanecerá fija durante toda la partida, de modo que cada unidad estará dirigida siempre por el mismo comportamiento. Esto no quiere decir que cada tipo de unidad vaya a ejecutar siempre la misma acción, ya que un comportamiento puede devolver acciones distintas dependiendo de la situación de la partida. Por ejemplo, se puede implementar un comportamiento para la base que produzca un trabajador si el jugador tiene menos de tres, y que no haga nada si ya se ha alcanzado esa cantidad.

4.1. Descripción de los comportamientos básicos

Los comportamientos básicos son diferentes formas de actuar que hemos desarrollado para cada tipo de unidad en base a nuestro conocimiento del dominio de los juegos RTS. A continuación explicamos las conductas implementadas en cada comportamiento.

4.1.1. Comportamientos de la base

La función de la base es almacenar recursos y producir trabajadores. Almacenar recursos en la base es algo intrínseco del juego que no se puede modificar, por tanto, lo único que puede determinar que una base se comporte de una u otra forma es cuándo produce los trabajadores y en qué cantidad.

Por un lado, los comportamientos OneWorker, TwoWorker y ThreeWorker producen trabajadores mientras que la población sea menor que la cantidad determinada de uno, dos o tres trabajadores. Si uno muere y como consecuencia la población es menor que la deseada, la base generará otro, pero si ya se ha alcanzado la población deseada se mantendrá inactiva. El motivo por el que no consideramos poblaciones superiores a tres trabajadores en estas estrategias se debe a las limitaciones del tablero del juego. Al no poder haber varias unidades en la misma casilla, si se juntan demasiados trabajadores a recolectar en el mismo yacimiento es muy probable que se produzcan interbloqueos que les impidan moverse. Además, es habitual que los recursos se encuentren en las esquinas del mapa y que por tanto haya poco espacio por el que desplazarse a su alrededor.

Por otro lado, RushWorker produce trabajadores continuamente, sin más limitación que la disponibilidad de recursos y el tiempo de espera requerido para producirlos. En principio, esta estrategia no funcionará bien si todos los trabajadores se ponen a recolectar en una zona pequeña, ya que acabarán bloqueándose.

4.1.2. Comportamientos del cuartel

El cuartel es la unidad encargada de producir tropas, por lo que sus comportamientos deben determinar en qué circunstancias se producen. Los que hemos implementado producen continuamente, siempre que haya recursos suficientes, un tipo concreto de tropa, ligera en el caso del comportamiento Light, pesada en el caso Heavy y a rango en el caso Ranged. Además, está el comportamiento Less, que intenta producir constantemente una unidad del tipo de tropa de la que menos tenga el jugador.

4.1.3. Comportamientos de los trabajadores

Implementar el comportamiento de los trabajadores es más complicado, ya que hay que tener en cuenta si el jugador necesita construir una base o un cuartel, si hay recursos que recolectar o si es preferible que algunos trabajadores ataquen en lugar de centrarse en la economía. Además, hay que sincronizar a todos los trabajadores y repartir correctamente las tareas, asegurándose de que cada uno haga lo que debe sin entorpecer al resto. Por tanto, la implementación de un comportamiento para los trabajadores debe contemplar que pueda haber trabajadores con acciones muy distintas en el mismo ciclo. Debido a la variedad de conductas que pueden adoptar, estos son los que tienen un mayor número de comportamientos posibles implementados.

El primero de ellos, HARVESTER, consiste en que todos los trabajadores que tenga el jugador en el mapa se dediquen por completo a la economía, es decir, a construir edificios y a recolectar recursos. En primer lugar se comprueba si hay al menos una base construida. En caso de no haberla, se ordena al primer trabajador que la construya, si dispone de los recursos necesarios, y asegurándonos que ningún otro trabajador se ponga a construir una segunda base a la vez. A continuación, se hace la misma comprobación con el cuartel, tomando las mismas precauciones que al construir la base. Por último, los trabajadores a los que no se les haya asignado la construcción de un edificio, bien por no ser necesario, bien por no disponer de suficientes recursos, son enviados a recolectar al yacimiento de recursos más cercano.

Otro comportamiento es AGGRESSIVE, pensado para producir trabajadores lo más rápido posible y enviarlos a atacar antes de que el rival esté preparado para defenderse. En este comportamiento no se contempla la construcción de un cuartel, pues el objetivo es que los trabajadores destruyan al enemigo antes de que produzca tropas, pero sí se comprueba que el jugador tenga una base construida. En caso negativo, los trabajadores intentarán construirla, incluso recolectando recursos si es necesario, pero una vez haya una base los trabajadores irán a atacar al enemigo más cercano sin preocuparse de recolectar ningún recurso.

Además, hemos incluido algunos comportamientos en los que cierto número de trabajadores recolectores se centran en la economía como en la estrategia HARVESTER, mientras que el resto se dedican al ataque como en la estrategia AGGRESSIVE. Estos son ONEHARVAGGR, TWOHARVAGGR y THREEHARVAGGR, en función de si hay uno, dos o tres trabajadores recolectores. En ellos se comprueba la presencia de una base y un cuartel como en HARVESTER, luego se envían trabajadores a recolectar y por último, si ya hay la cantidad de recolectores deseada y quedan trabajadores sin una acción asignada, se envían a atacar a la unidad enemiga más cercana. Como hemos indicado en 4.1.1, limitamos el número de recolectores para evitar que se obstruyan el paso unos a otros al transportar los recursos.

Por último, tenemos algunos comportamientos similares a los anteriores pero que descartan el desarrollo militar. En ellos, cierto número de trabajadores se dedican a la recolección, pero los recursos obtenidos se dedicarán por completo a la producción

de más trabajadores, construyendo una base si es necesario pero nunca un cuartel. El resto de trabajadores se envían a luchar. Los nombres que identifican a estos comportamientos son OneHarvNoBar, TwoHarvNoBar y ThreeHarvNoBar, en función del número de recolectores.

En la tabla 4.1 se muestra un resumen de los comportamientos que pueden tener los trabajadores.

Comportamiento	Cuartel	Recolectores	Atacantes
Harvester	Sí	Todos	Ninguno
OneHarvAggr	Sí	1	Todos excepto 1
TwoHarvAggr	Sí	2	Todos excepto 2
ThreeHarvAggr	Sí	3	Todos excepto 3
Aggressive	No	Ninguno	Todos
OneHarvNoBar	No	1	Todos excepto 1
TwoHarvNoBar	No	2	Todos excepto 2
ThreeHarvNoBar	No	3	Todos excepto 3

Tabla 4.1: Comportamientos de los trabajadores.

4.1.4. Comportamientos de las tropas

En cuanto a los comportamientos que pueden adoptar las tropas militares, la implementación es igual para los tres tipos (soldados ligeros, pesados y a rango), por lo que los describiremos de forma genérica.

En primer lugar, hemos implementado comportamientos que atacan al enemigo más cercano a la unidad y que dan distinta prioridad a unos objetivos u otros cuando hay varios enemigos en rango de ataque. El comportamiento Closest simplemente busca la unidad enemiga a menor distancia y, en caso de empate, ataca a la primera que ha encontrado. La estrategia LessHp se acerca al enemigo más cercano y, en caso de tener varios enemigos en rango de ataque, ataca primero al que menos vida tenga. Otra estrategia es LessPercHp, similar a la anterior, pero dando prioridad a atacar a unidades con menor porcentaje de vida.

En el caso del comportamiento ClosestBuild, la unidad atacará al enemigo más cercano a alguno de los edificios del jugador. El propósito de esta conducta es que las tropas sean capaces de reaccionar a un ataque a la base del jugador aunque se encuentren lejos de ésta, ya que los comportamientos descritos en el párrafo anterior pueden provocar que las tropas se obstinen en un ataque tras las filas enemigas dejando indefensa la base, situación en la que algún enemigo puede separarse del resto y atacarla sin encontrar resistencia.

Otro comportamiento incluido es Wait, en el que las tropas esperan en una posición determinada cercana a la base formando un grupo. Si un enemigo se acerca demasiado, el grupo reacciona atacándolo.

Por último, hay un comportamiento particular que sólo tienen los soldados a rango, la estrategia KITE. Consiste en que la unidad ataca al enemigo manteniendo cierta distancia con él, es decir, la unidad dispara a un enemigo que entre en su rango de ataque y se aleja si el enemigo se acerca demasiado. Esto puede ser útil al enfrentarse contra unidades a melé (con un rango de ataque de sólo una casilla), ya que al mantener la distancia no reciben daño. Sin embargo, puede ser contraproducente porque si el enemigo se acerca lo suficientemente rápido, la unidad a rango intenta alejarse permanentemente sin efectuar ningún disparo y puede llegar a verse arrinconada.

Unidad	Comportamientos
Base	ONEWORKER, TWOWORKER, THREEWORKER, RUSHWORKER
Cuartel	LIGHT, HEAVY, RANGED, LESS
Trabajador	HARVESTER, ONEHARVAGGR, TWOHARVAGGR, THREEHARVAGGR, AGGRESSIVE, ONEHARVNOBAR, TWOHARVNOBAR, THREEHARVNOBAR
Soldado ligero	CLOSEST, LESSHP, LESSPERCHP, CLOSESTBUILD, WAIT
Soldado pesado	CLOSEST, LESSHP, LESSPERCHP, CLOSESTBUILD, WAIT
Soldado a rango	CLOSEST, LESSHP, LESSPERCHP, CLOSESTBUILD, WAIT, KITE

Tabla 4.2: Resumen de los comportamientos básicos de las unidades

4.2. Construcción de bots a partir de comportamientos

Ahora pretendemos construir bots capaces de utilizar una combinación de los comportamientos básicos que acabamos de describir. Como hemos explicado, cada comportamiento se aplica a todas las unidades de un mismo tipo de entre los seis que existen en μ-RTS. Nuestros primeros bots utilizarán durante toda la partida un único comportamiento para cada uno de los tipos de unidades, elegido de la colección de comportamientos básicos disponibles, que se resume en la tabla 4.2.

La elección de los comportamientos que se van a emplear durante la partida es una decisión dentro del ámbito estratégico o del *macrojuego* (sección 2.2). Es una decisión a largo plazo, dirigida a lograr el objetivo final de vencer al oponente, y que no va a cambiar en toda la partida. Además, determinará aspectos como la composición del ejército del jugador, en qué medida se va a desarrollar su economía o qué edificios se van a construir, todos ellos propios del ámbito de la estrategia. Por otro lado, las decisiones tácticas y de control reactivo de unidades, es decir, el *microjuego*, vienen codificadas dentro de los propios comportamientos, que especifican dónde construir los edificios, cómo moverse por el tablero (*pathfinding*) o si durante una batalla se debe esquivar un ataque.

En definitiva, la forma de jugar de estos bots quedará caracterizada por los seis comportamientos elegidos para cada tipo de unidad, que constituyen una estrategia para toda la partida. Es por eso que los llamaremos bots *SingleStrategy*. Siguiendo las pautas descritas en la sección 3.7, el bot *SingleStrategy* tendrá un "cerebro" capaz de comunicarse con el núcleo de μ-RTS y que, para cada unidad del jugador, preguntará a los comportamientos básicos escogidos cuál es la acción que debería ejecutar.

Como se puede comprobar en la tabla 4.2, hemos implementado 4 comportamientos para la base, otros 4 para el cuartel, 8 para los trabajadores y 5 para cada tropa militar, excepto los soldados a distancia que tienen un comportamiento adicional. Esto hace que un *SingleStrategy* pueda tener $4 \times 4 \times 8 \times 5 \times 5 \times 6 = 19\,200$ posibles configuraciones, una cifra que no es demasiado grande y que nos permite probarlas todas y evaluar su rendimiento.

Hay que mencionar que muchas de las configuraciones actúan exactamente de la misma manera, por ejemplo, un bot que tenga el comportamiento AGGRESSIVE para los trabajadores no construirá cuartel ni producirá tropas, por tanto, la elección de un comportamiento u otro para esas unidades no influirá en absoluto en el desarrollo de la partida. A continuación, vamos a estudiar exactamente cuántas estrategias distintas hay entre las 19 200 configuraciones de *SingleStrategy*.

4.3. Estrategias que puede adoptar *SingleStrategy*

En primer lugar observamos que de los seis comportamientos que se deben elegir para configurar una estrategia dos influyen claramente en la forma de actuar de los trabajadores. Estos son el de la base (que los produce) y, claro está, el de los propios trabajadores. Los otros cuatro afectan a cómo actúan las tropas, y no afectan en absoluto a lo que hagan los trabajadores. Sin embargo, el comportamiento de los trabajadores sí influye en el de las tropas, ya que pueden construir o no un cuartel. Por tanto separaremos los 8 posibles comportamientos de los trabajadores en 4 que construyen cuartel y 4 que no.

La forma de actuar de los trabajadores depende esencialmente de cuántos se dedican a la economía (es decir, a recolectar y construir) y cuántos se dedican a atacar. Esto depende de los comportamientos que tengan la base y los propios trabajadores. Supongamos que la base se comporta según ONEWORKER, entonces el jugador nunca tendrá más de un trabajador. Si el comprtamiento para los trabajadores es HARVESTER, el trabajador realizará las siguientes acciones por orden de prioridad: construirá base si no hay, construirá cuartel si no hay o recolectará. Si el comportamiento es ONEHARVAGGR, TWOHARVAGGR o THREEHARVAGGR ocurrirá lo mismo, ya que ninguno de estos tres últimos envía trabajadores a atacar cuando sólo hay uno. En el caso de que la base mantenga una población de dos trabajadores, los comportamientos HARVESTER, TWOHARVAGGR y THREEHARVAGGR serán iguales y mantendrán los dos trabajadores recolectando, mientras que ONEHARVAGGR usará uno como recolector y otro como atacante. Siguiendo este

razonamiento, podemos completar la tabla 4.3, en la que se muestra la distribución de tareas de los trabajadores en función de sus comportamientos y los de la base. Cada casilla de la tabla se corresponde con una combinación de comportamientos de base y trabajadores y tiene dos cifras separadas por una barra. La cifra de la izquierda indica cuántos trabajadores de entre toda la población se dedicarían a la economía con esa configuración, y la de la derecha cuántos trabajadores estarían atacando. El símbolo * indica que los trabajadores centrados en la economía no construyen cuartel, y por tanto su comportamiento es distinto de aquellos que sí lo construyen.

	Comportamiento de la base			
Comportamiento de trabajadores	One	Two	Three	Rush
Construyen cuartel				
Harvester	1/0	2/0	3/0	N/0
OneHarvAggr	1/0	1/1	1/2	1/(N-1)
TwoHarvAggr	1/0	2/0	2/1	2/(N-2)
ThreeHarvAggr	1/0	2/0	3/0	3/(N-3)
No construyen cuartel				
Aggressive	0/1	0/2	0/3	0/N
OneHarvNoBar	1*/0	1*/1	1*/2	1*/(N-1)
TwoHarvNoBar	1*/0	2*/0	2*/1	2*/(N-2)
ThreeHarvNoBar	1*/0	2*/0	3*/0	3*/(N-3)

Tabla 4.3: Distribución de trabajadores según su comportamiento y su población.

De la tabla 4.3 extraemos que de las 16 configuraciones que construyen cuartel, sólo 10 son realmente distintas. En el caso de aquellas que no construyen cuartel, de las 16 posibilidades, son distintas 13.

Pasemos ahora a considerar los otros cuatro tipos de unidad: cuartel, ligero, pesado y a rango. Si los trabajadores no construyen cuartel, la elección del comportamiento para éste y las tropas es irrelevante, ya que no entran en escena. Hemos explicado que no construyen cuartel 4 de los 8 comportamientos de los trabajadores, por lo que el número de configuraciones de *SingleStrategy* en esta situación son $4 \times 4 \times 4 \times 5 \times 5 \times 6 = 9\,600$. En todos ellos, sólo influye la forma de actuar de los trabajadores, que queda determinada por aquellos de sus 4 comportamientos que no construyen cuartel y los 4 comportamientos de la base. En total, son 16 posibilidades de las que sólo 13 definen estrategias distintas, como hemos visto en la tabla 4.3.

Además, cuando el cuartel produce un solo tipo de tropa se vuelven irrelevantes los comportamientos elegidos para las otras dos. Para analizarlo, supongamos que fijamos el comportamiento del cuartel. Para cada comportamiento del cuartel, tenemos $4 \times 4 \times 5 \times 5 \times 6 = 2\,400$ configuraciones posibles eligiendo los comportamientos del resto de unidades, donde al elegir el de los trabajadores consideramos sólo los 4 que sí construyen cuartel.

Ahora, supongamos que el cuartel tiene el comportamiento LIGHT. Entonces el comportamiento de soldados pesados y a rango no afecta a la partida. Así, se pueden elegir 4 opciones para la base y 4 para los trabajadores que construyen cuartel, y hemos visto en la tabla 4.3 que de las 16 posibilidades sólo 10 son distintas. De los comportamientos de las tropas, sólo cuentan las 5 posibilidades de los soldados ligeros, que son los únicos que se producen. Por tanto, fijado el comportamiento del cuartel a LIGHT, de las 2 400 configuraciones posibles, sólo $10 \times 5 = 50$ definen estrategias distintas. Exactamente lo mismo ocurre cuando la tropa producida son soldados pesados, las 2 400 opciones se reducen a 50. Para el caso de soldados a rango la cifra es distinta, pero el razonamiento es análogo: hay $10 \times 6 = 60$ estrategias distintas de las 2 400 combinaciones teóricas. Sin embargo, cuando el comportamiento del cuartel es LESS, las elección para todos los soldados importa, por lo que las 2 400 posibilidades definen $10 \times 5 \times 5 \times 6 = 1\,500$ estrategias diferentes.

	Configuraciones teóricas	Estrategias distintas
No construyen cuartel	9 600	13
Sólo producen Ligeros	2 400	50
Sólo producen pesados	2 400	50
Sólo producen A rango	2 400	60
Producen todas las tropas	2 400	1 500
TOTAL	19 200	1 673

Tabla 4.4: Estudio del número de estrategias distintas para utilizar en *SingleStrategy*.

La tabla 4.4 recoge todos los cálculos realizados, que nos llevan a concluir que de las 19 200 configuraciones posibles de comportamientos básicos en un bot *SingleStrategy*, tan sólo 1 673 definen estrategias distintas.

4.4. Resultados de los bots *SingleStrategy*

Para comprobar cuales de nuestros bots obtienen mejores resultados, los enfrentamos a un conjunto de 15 bots de referencia, 14 de ellos incluidos en código de μ-RTS y cuyo comportamiento se explica en la sección 3.8. Estos bots son: *EconomyMilitaryRush*, *EconomyRush*, *EconomyRushBurster*, *EMRDeterministico*, *HeavyDefense*, *HeavyRush*, *LightDefense*, *LightRush*, *RandomBiasedAI*, *RangedDefense*, *RangedRush*, *SimpleEconomyRush*, *WorkerDefense* y *WorkerRushPlusPlus*. Además, hemos introducido el bot *Droplet* [34], que participó en el concurso de μ-RTS de la Conference of Games (CoG) de 2019, resultando en tercera posición en la categoría estándar con mapas conocidos.

En este experimento hemos utilizado la misma configuración que en el concurso de la CoG [33], por lo que utilizamos la configuración afinada para las tropas con cancelación de ambas acciones en caso de conflicto. En cuanto a la *niebla de guerra*, optamos por no utilizarlo puesto que añade una dimensión de dificultad adicional más allá de lo que es la aplicación del aprendizaje automático a los juegos RTS.

4.4.1. Mapa de 8x8

En primer lugar, realizamos el experimento en el mapa *BasesWorkers8x8*, que se muestra en la figura 4.1. Cada jugador tiene una base y un trabajador, además de un bloque de recursos cercano. Cada uno de los 19200 *SingleStrategy* jugó 2 partidas contra cada uno de los 15 bots de referencia. En la segunda partida, ambos bots se intercambiaron las posiciones de inicio para garantizar la igualdad de condiciones. El límite de tiempo de una partida antes de que el resultado se considerara empate fue de 3000 ciclos, como se establece en las reglas del concurso de μ-RTS de la CoG [33] para mapas de 8x8. Para estudiar el desempeño de cada *SingleStrategy*, contamos el número de victorias que consiguieron, de manera que la máxima puntuación posible es 30.

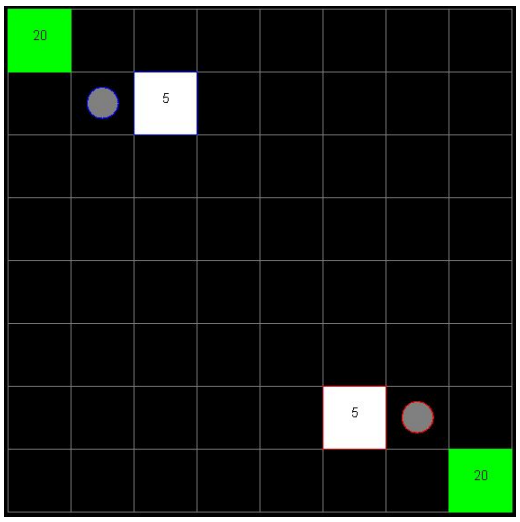

Figura 4.1: Mapa *BasesWorkers8x8* incluido en el juego.

El archivo *All8x8.csv*[1] recoge los resultados obtenidos durante esta prueba. Consta de 19 200 filas correspondientes a cada configuración probada, además de 6 columnas con los comportamientos usados para cada tipo de unidad, 15 columnas con los resultados obtenidos contra cada rival y una última columna con la suma de todas las puntuaciones. A partir de estos datos, destacamos las siguientes observaciones:

- La puntuación más alta obtenida por un *SingleStrategy* es de 28 sobre 30.
- Ninguno de los 19 200 bots consiguió ganar en ambas partidas a *WorkerDefense* ni a *WorkerRushPlusPlus*.
- Aquellos que alcanzaron 28 puntos ganaron todas las partidas salvo una contra *WorkerDefense* y otra contra *WorkerRushPlusPlus*.

[1] `https://github.com/TFG-Informatica/Aprendizaje-automatico-aplicado-a-juegos-RTS/blob/master/data/Exhaustivo/All8x8/All8x8.csv`

Todos los *SingleStrategy* que alcanzaron la puntuación de 28 sobre 30 utilizaron una de estas tres posibles configuraciones:

- RushWorker para la base y Aggressive para los trabajadores. Esta estrategia consiste en producir trabajadores sin parar y enviarlos al ataque.
- ThreeWorker para la base y Aggressive para los trabajadores. En este caso, el mapa es tan pequeño que, aunque la base esté produciendo trabajadores continuamente, al enviarlos a todos al ataque no llega a haber más de tres con vida a la vez. Por ello, en el mapa 8x8, esta estrategia es esencialmente igual que la anterior, aunque en un mapa más grande sí se apreciaría la diferencia.
- RushWorker para la base y TwoHarvNoBar para los trabajadores. Esta estrategia consiste en producir trabajadores sin parar, de los que dos se ponen a recolectar mientras que el resto atacan al enemigo. Poner a tres trabajadores a recolectar es contraproducente en este mapa tan pequeño, ya que en el tiempo que tardan en producirse, al enemigo le da tiempo a empezar a atacar la base del jugador. Por otro lado, poner a sólo un trabajador a recolectar no da suficiente ventaja sobre los oponentes.

Ninguna de estas tres configuraciones construye un cuartel, por lo que el comportamiento elegido para éste y para los soldados es irrelevante. Como hay 4 comportamientos básicos de cuarteles y 5, 5 y 6 para las tropas ligeras, pesadas y a rango, respectivamente, tenemos 600 bots dentro de cada configuración que tienen idéntico comportamiento. Sin embargo, no siempre ganaron las mismas partidas, debido a la presencia de bots no deterministas. Las puntuaciones obtenidas por los *SingleStrategy* con estas tres configuraciones oscilan entre los 25 y los 28 puntos, ya que en las 4 partidas contra los bots *RandomBiasedAI* y *Droplet* obtuvieron 1, 2, 3 o 4 victorias. La tabla 4.5 muestra las posibles puntuaciones y cuántos de los 600 bots de cada configuración obtuvieron dichas puntuaciones.

	Puntuación				
Bot	28	27	26	25	Media
RushWorker y Aggressive	146	365	82	7	27,08
ThreeWorker y Aggressive	147	385	79	9	27,05
RushWorker y TwoHarvNoBar	16	183	401	0	26,35

Tabla 4.5: La tabla muestra el número de bots que consiguió cada puntuación y la puntuación media

A la vista de los datos de la tabla 4.5, parece que el bot con RushWorker y TwoHarvNoBar tuvo un rendimiento peor que los otros dos. Su moda fue de 26 puntos, y tan sólo alcanzó los 28 en 16 de las 600 ocasiones en las que jugó, lo que supone que consiguió ganar las 4 partidas a *RandomBiasedAI* y *Droplet* en uno

de cada 38 enfrentamientos. En cuanto a los otros dos bots, obtuvieron resultados muy similares debido a que en el mapa de 8x8 desarrollan esencialmente la misma estrategia. Sus modas fueron de 27 puntos, y alcanzaron los 28 puntos en alrededor de 150 de las 600 veces que jugaron. Esto significa que consiguieron ganar las 4 partidas a *RandomBiasedAI* y *Droplet* en uno de cada 4 enfrentamientos.

Figura 4.2: Resultados obtenidos en función del comportamiento de los trabajadores.

La figura 4.2 presenta las puntuaciones obtenidas por los bots *SingleStrategy* en función del comportamiento adoptado por los trabajadores. Es importante notar que en los resultados que muestra la gráfica pueden haber influido los comportamientos de otro tipo de unidades, como la base. Por ejemplo, un gran número de bots con el comportamiento ThreeHarvNoBar han obtenido resultados muy malos, por debajo de 10 puntos, pero esto no se debe exclusivamente al comportamiento ThreeHarvNoBar, sino a su combinación con estrategias de la base que limitan la población de trabajadores a 1, 2 o 3. En estas situaciones, ningún trabajador ataca ni construye un cuartel para producir tropas, por lo que no tiene manera de defenderse, lo que explica la baja puntuación. En contraposición a esto, cuando se combina ThreeHarvNoBar para los trabajadores con RushWorker para la base se logran puntuaciones en torno a 25 puntos que se corresponden con los puntos superiores de la columna de ThreeHarvNoBar.

No obstante, la figura 4.2 nos sirve para confirmar que los comportamientos que construyen cuartel (los 4 de la izquierda) tienen un potencial menor que los que no lo construyen (los 4 de la derecha). Para afirmar esto, nos basamos en que los *SingleStrategy* que construyen cuartel no superan los 25 puntos, mientras que cual-

quier estrategia que prescinde de cuartel consigue superar esa barrera. Puede parecer que conseguir 25 de 30 puntos es un buen resultado, pero en realidad, muchos de los 15 bots básicos que hemos utilizado para la evaluación no están diseñados para un mapa de 8x8. Están programados para construir cuartel, a pesar de no ser la estrategia óptima en este mapa, y, por tanto, nuestros *SingleStrategy* que también construyen cuartel tienen la oportunidad de ganarles y acumular muchas victorias. Los bots básicos que realmente destacan en este escenario son *WorkerRushPlusPlus* y *WorkerDefense*, que sólo producen trabajadores. También *Droplet* es capaz de adoptar una estrategia basada en trabajadores. Por tanto, en este experimento, de los 30 puntos que puede llegar a conseguir un *SingleStrategy*, son 6 los que realmente marcan la diferencia entre unos u otros. Todos los *SingleStrategy* que construyeron cuartel perdieron las 4 partidas contra *WorkerRushPlusPlus* y *WorkerDefense*, mientras que los mejores que prescindieron del cuartel lograron ganar contra ellos en 2 de las 4 partidas. Esto es causado porque en mapas pequeños, como se explicó en la sección 3.5, las estrategias ganadoras consisten en hacer un ataque rápido con trabajadores antes de que el contrario tenga tiempo de generar unidades más fuertes.

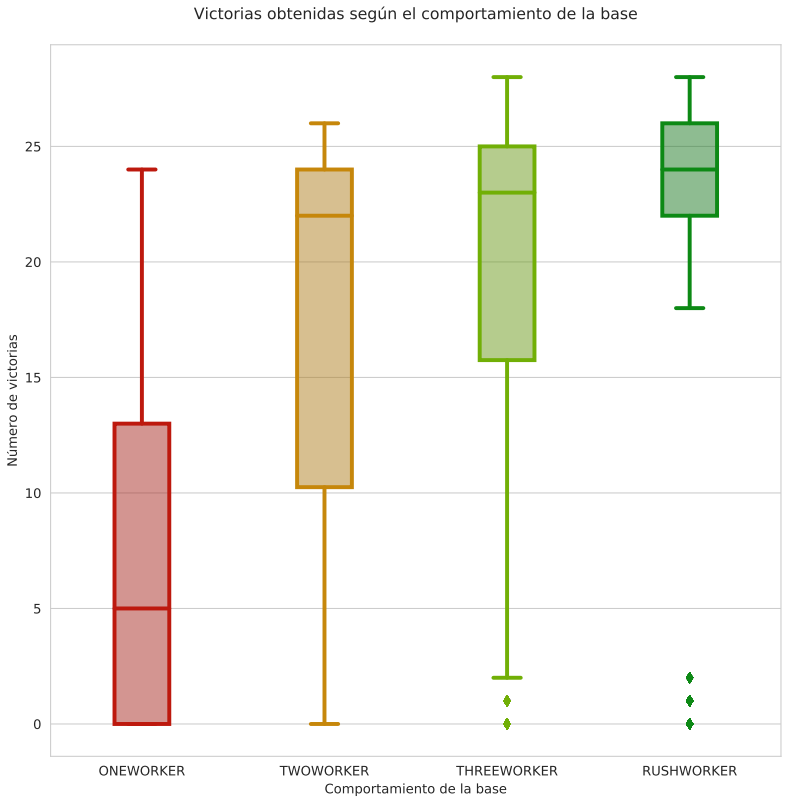

Figura 4.3: Resultados obtenidos en función de la estrategia de la base.

También queda clara en la figura 4.3 la importancia de producir muchos trabajadores. Esto se corresponde con el hecho de que las estrategias que emplean a los trabajadores para atacar tienen mejores resultados en este mapa. Cuantos más trabajadores se sumen al ataque, más posibilidades de éxito tendrá. Además, en la gráfica se pone de manifiesto la relación entre la población de trabajadores,

controlada por la base, y el comportamiento que adoptan. Muchos de los comportamientos de los trabajadores los dividen en un grupo dedicado a la recolección y otro dedicado al ataque. Si la población está limitada a 1, 2 o 3 trabajadores, y su comportamiento establece que la cantidad que debe dedicarse a recolectar es mayor que la población, entonces ninguno atacará, aunque el comportamiento contemple enviar a los trabajadores sobrantes al ataque. Al no atacar con los trabajadores, se obtienen puntuaciones más bajas. Un análisis más detallado de cómo influye la combinación de los comportamientos de la base y los trabajadores en el reparto de tareas de estos últimos se puede ver en la tabla 4.3

4.4.2. Mapa de 24x24

En este segundo experimento pusimos a prueba a los 19 200 *SingleStrategy* en el mapa *BasesWorkers24x24*, que se muestra en la figura 4.4. Cada jugador dispone una base con un par de bloques de recursos al lado y un trabajador. Las condiciones fueron las mismas que en el caso del mapa de 8x8, se jugaron 2 partidas contra el conjunto de bots de referencia alternándose las posiciones. Sin embargo, en este caso tuvimos que descartar el bot *Droplet*, ya que requiere un tiempo de cómputo de 100 ms por ciclo, lo que provocaba que la prueba se alargase demasiado. Con una estimación optimista de 1 000 ciclos por partida, se tardaría un total de 1 066 horas en enfrentarlo a los 19 200 bots *SingleStrategy*, lo cual nos parece una cantidad de tiempo excesiva. Tras eliminar al *Droplet* del conjunto de rivales, el máximo número de victorias que se puede conseguir es 28. También cambia el límite de ciclos de la partida, que establecimos en 5 000 ciclos, al igual que en la competición de μ-RTS de la CoG [33].

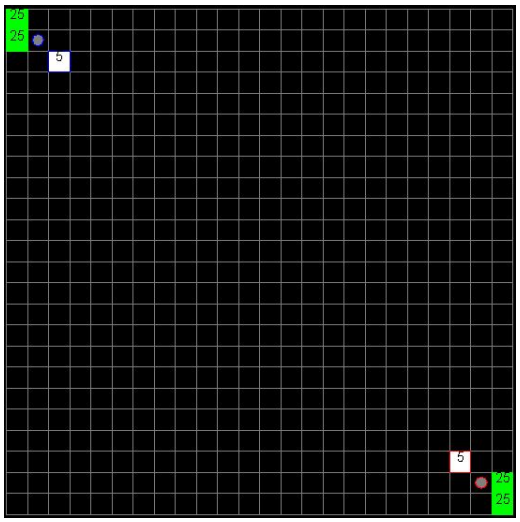

Figura 4.4: Mapa *BasesWorkers24x24* incluido en el juego.

Los resultados de los enfrentamientos están recogidos en el archivo *All24x24.csv*[2]. El archivo consta de 19 200 filas correspondientes a cada configuración probada, además de 6 columnas con los comportamientos usados para cada tipo de unidad, 14 columnas con los resultados obtenidos contra cada rival y una última columna con la suma de todas las puntuaciones. Estos datos muestran que las siguientes tres estrategias consiguieron ganar las 28 partidas jugadas:

[2] `https://github.com/TFG-Informatica/Aprendizaje-automatico-aplicado-a-juegos-RTS/blob/master/data/Exhaustivo/All24x24/All24x24.csv`

- La primera estrategia utiliza dos trabajadores que construyen un cuartel y se dedican por completo a la recolección. En la tabla 4.3 se muestra que hay varias combinaciones de comportamientos de la base y los trabajadores para conseguirlo: TwoWorker con Harvester, TwoWorker con TwoHarvAggr y TwoWorker con ThreeHarvAggr. El cuartel produce exclusivamente soldados ligeros, que atacan a la unidad enemiga que tienen más cerca. Recordemos que hay varios comportamientos que atacan a la unidad enemiga más cercana y que se diferencian en a quien priorizan en caso de haber varios enemigos en rango de ataque (sección 4.1). Estos son Closest, LessHp y LessPercHp.
- La segunda estrategia emplea dos trabajadores recolectores de la misma manera que lo hacía la primera configuración. Sin embargo, se diferencia de la anterior en que el cuartel produce exclusivamente soldados pesados, que de nuevo atacan a la unidad enemiga más cercana.
- La tercera estrategia utiliza también un cuartel que produce exclusivamente soldados pesados, que atacan a la unidad enemiga más cercana, pero utiliza tres trabajadores recolectores en lugar de dos.

Estas configuraciones de *SingleStrategy* ejecutan estrategias similares a *LightRush* y *HeavyRush* (ver 3.7) que tratan de mantener una población de 2 o 3 trabajadores dedicados a recolectar y producen un ejército formado por un único tipo de tropa, bien soldados ligeros o bien soldados pesados. Además, consiguen ganar a las versiones de *LightRush* y *HeavyRush* incluidas en μ-RTS, ya que en ellas sólo se pone a un trabajador a recolectar, por lo que obtienen recursos más lentamente.

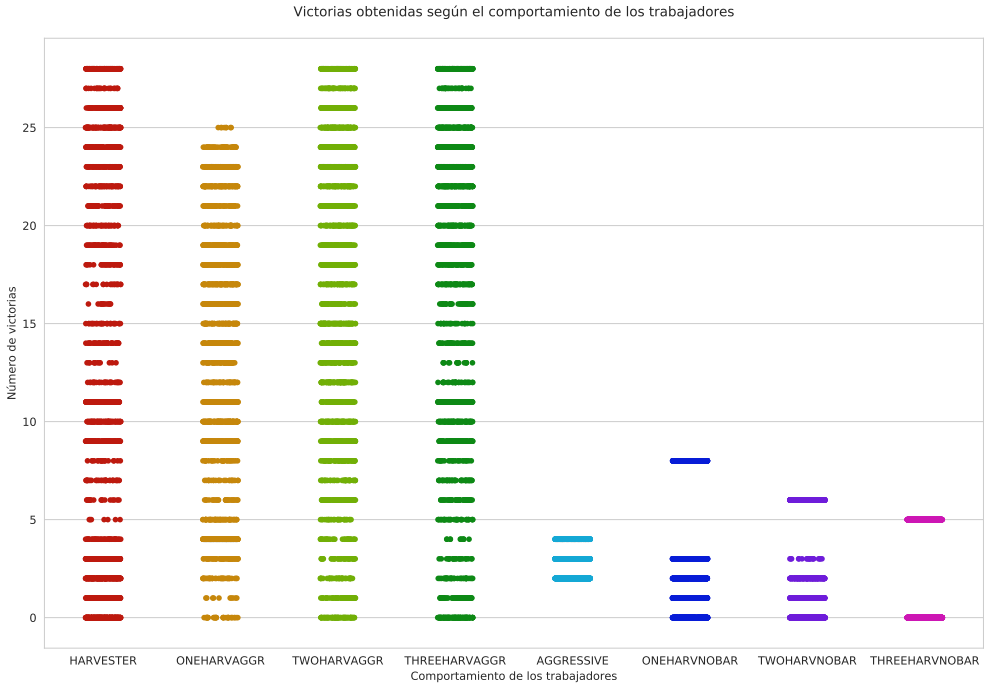

Figura 4.5: Resultados obtenidos en función de la estrategia de los trabajadores en el mapa de 24x24.

En la figura 4.5 queda claro que las estrategias que mejor funcionan en el mapa de 24x24 se invierten con respecto a las del mapa de 8x8. En este caso, los comportamientos que no construyen cuartel (los 4 de la derecha) y que funcionaban en el mapa más pequeño no alcanzan ni siquiera 10 puntos, mientras que aquellos que desarrollan unidades militares (los 4 de la izquierda) obtienen los mejores resultados. En la sección 4.4.1 estudiábamos el desempeño de *SingleStrategy* en el mapa de 8x8 y hablamos de que muchos de los 14 bots de referencia usados en la evaluación no estaban pensados para el mapa de 8x8. Por ello, los *SingleStrategy* que construían cuartel, a pesar de ser peores para un mapa tan pequeño, obtenían puntuaciones altas de hasta 25 sobre 30 puntos. En el mapa de 24x24, la mayoría de bots se desenvuelven bien, lo cual explica que los *SingleStrategy* que usan estrategias mal adaptadas al mapa obtengan puntuaciones mucho más bajas.

Las estadísticas obtenidas por los comportamientos del cuartel, que se pueden observar en la figura 4.6, reflejan que al producir solo soldados a rango se obtienen peores resultados que al hacerlo con ligeros o pesados. Esto se debe a que los soldados a rango son tropas frágiles, que mueren de un solo golpe contra cualquier oponente. Para compensar esta debilidad, pueden atacar a una distancia de hasta tres casillas, pero esto no es suficiente para hacer frente a unidades como los ligeros o los pesados, que tienen suficiente vida como para resistir sus disparos hasta alcanzarlos. Los detalles de las características de estas unidades se pueden consultar en la sección 3.3.

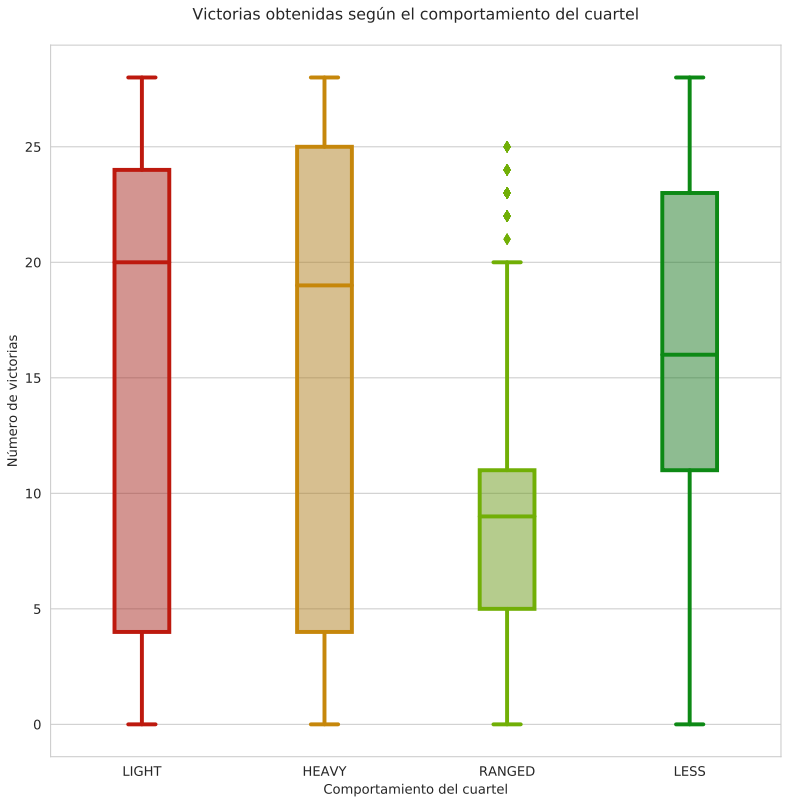

Figura 4.6: Estadísticas en función del comportamiento del cuartel.

Para eliminar ruido de la figura 4.6, sólo se tienen en cuenta los resultados de bots cuyos trabajadores construyen cuartel, pero aun así hay que insistir en que en los resultados mostrados influyen otros comportamientos aparte del elegido para el cuartel. A pesar de esto, los factores externos al cuartel afectan por igual a todos los comportamientos, por lo que sí se pueden extraer conclusiones fiables. El hecho de que las tropas ligeras o pesadas alcancen los 28 puntos pero los soldados a rango no, refleja que no tienen el mismo potencial.

A partir de las estadísticas de los comportamientos de las tropas ligeras, visibles en la figura 4.7, se extrae que la estrategia WAIT es pésima para el bot *SingleStrategy*, algo lógico ya que tener a las tropas en espera durante toda la partida difícilmente nos puede llevar a la victoria. También se observa que el comportamiento CLOSESTBUILD, que ataca al enemigo más cercano a la base, logra resultados ligeramente peores que aquellos comportamientos que atacan al enemigo más cercano a la unidad que va a ejecutar el ataque. La gráfica que se obtiene al estudiar los comportamientos de soldados pesados es muy similar y conduce a las mismas conclusiones.

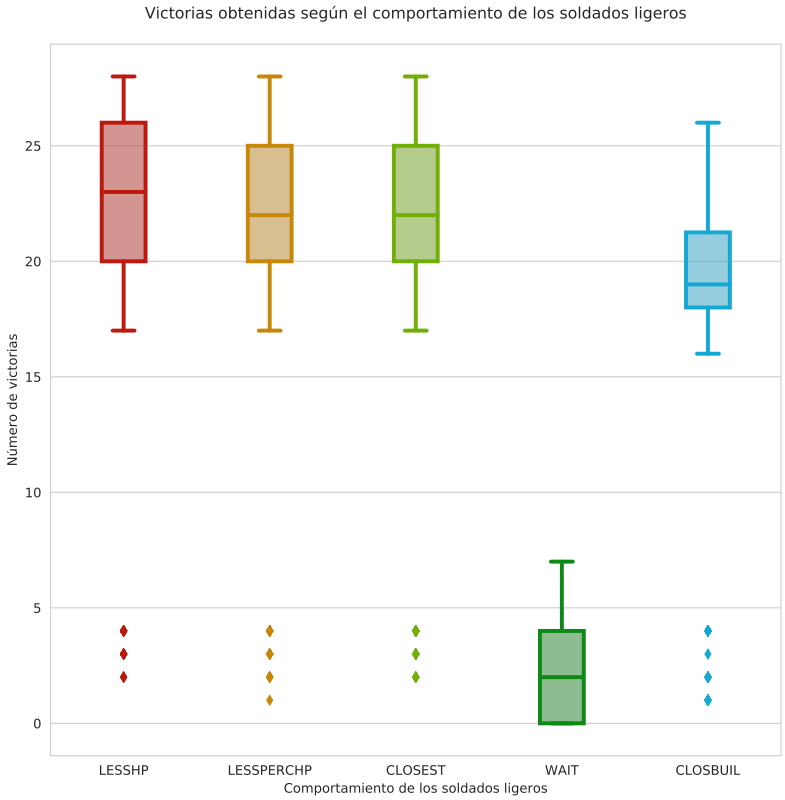

Figura 4.7: Estadísticas en función del comportamiento de las tropas ligeras.

Nuevamente, para eliminar ruido del diagrama, sólo se tienen en cuenta los resultados de bots cuyos trabajadores construyen cuartel y que utilizan el comportamiento LIGHT para el cuartel, pero aún así en los resultados pueden influir el resto de comportamientos. Por ejemplo, hemos comprobado que los puntos extremos que se acumulan por debajo de 5 puntos en todos los comportamientos distintos de WAIT

se deben al uso simultáneo de RushWorker para la base y Harvester para los trabajadores (ver sección 4.1). Esto provoca que una población cada vez mayor de trabajadores se acumule alrededor de los bloques de recursos, obstruyéndose el paso unos a otros y generando situaciones de interbloqueo entre las unidades.

Finalmente, aunque no hayamos incluido el bot *Droplet* en la prueba con los 19 200 bots, hemos tomado tres bots representativos de las tres configuraciones que alcanzaron la puntuación máxima de 28 puntos durante la prueba y los hemos enfrentado a *Droplet* en 10 partidas, para ver si también son capaces de vencerlo.

- El primer bot, formado por TwoWorker, Light y Harvester como comportamientos para la base, el cuartel y los trabajadores, respectivamente, y Closest como criterio de ataque para las tropas no ganó ninguna partida.
- El segundo, con el comportamiento Heavy para el cuartel y los mismos que el anterior para el resto de unidades, obtuvo 10 victorias.
- Finalmente, el tercero usó ThreeWorker, Heavy y Harvester en la base, el cuartel y los trabajadores, respectivamente, y también Closest en las tropas, y obtuvo 8 victorias.

Estos resultados indican que a la hora de enfrentarse a *Droplet*, usar tropas pesadas es una opción considerablemente mejor que el uso de tropas ligeras.

Conclusiones

En este capítulo hemos explicado el desarrollo del bot *SingleStrategy*, que utiliza una estrategia fija durante toda la partida construida a partir de comportamientos básicos precodificados. Hemos estudiado su desempeño en dos mapas de μ-RTS. En el mapa de 8x8 dominan las estrategias similares a *WorkerRush*, mientras que en el de 24x24 las mejores estrategias son similares a *LightRush* y *HeavyRush*. Esto refleja que el tamaño del mapa es determinante a la hora de jugar una partida, de tal forma que estrategias ganadoras en el mapa de 8x8 no son útiles en el de 24x24 y viceversa.

A la vista de esto, una de las ventajas de nuestro bot *SingleStrategy* es que ofrece la capacidad de utilizar estrategias adaptadas al tablero en el que va a jugar. Para determinar qué estrategia es mejor para cierto mapa, es necesario realizar algún tipo de prueba como la que hemos llevado a cabo a lo largo de la sección 4.4. En estas pruebas hemos enfrentado a *SingleStrategy* contra algunos bots básicos incluidos en el juego. Hemos descubierto que es importante la elección de los rivales con los que vamos a evaluarlo, ya que si lo enfrentamos a bots que no saben jugar bien en el mapa que estamos estudiando no obtendremos resultados representativos.

En el capítulo 5 probaremos el rendimiento de un bot más flexible que pueda tener varias estrategias por partida en lugar de una sola. Como la cantidad de bots posibles aumentará drásticamente al aumentar el número de estrategias por partida, utilizaremos un algoritmo genético para buscar la mejor combinación de comportamientos para cada tablero.

Capítulo 5

Jugando a μ-RTS con varias estrategias

Llegados a este punto, hemos comprobado que es factible crear bots razonablemente buenos con las estrategias que hemos diseñado. Sin embargo, los bots resultantes son bastante simples y limitados, ya que siempre actúan igual a lo largo de toda la partida. Esta clase de planteamientos estáticos provocan una gran desventaja al enfrentarse a bots más adaptativos, capaces de cambiar la estrategia que siguen según las acciones del contrario o el desarrollo de la partida. En este capítulo se persigue mejorar este aspecto, aumentando la complejidad de nuestros bots para mejorar su comportamiento.

5.1. Bot *MultiStrategy*

La aproximación más natural a este aumento de la complejidad en los bots es dejar de jugar con una única estrategia para cada tipo de unidad durante toda la partida y pasar a utilizar varias. De forma intuitiva, querremos ir pasando por distintas fases (como se mencionaba en la sección 2.1.2) e ir adaptando nuestras estrategias a medida que avanzamos de fase. Así, por ejemplo, podríamos empezar con una estrategia no agresiva, en la que recolectemos recursos y construyamos los cuarteles, para en la siguiente fase producir tropas y reservarlas, y finalmente atacar con todas a la vez en la última fase. Para implementar esto hemos hecho un nuevo bot, al que hemos llamado *MultiStrategy*, que considera varias estrategias como las utilizadas por *SingleStrategy* de forma que en cada fase aplica la que corresponde. Para escoger la estrategia correspondiente en cada momento, dividimos el máximo de ciclos que se permite que dure la partida entre la cantidad de fases, utilizando cada estrategia durante esa cantidad de ciclos.

Para elegir el máximo de ciclos que permitimos que dure la partida tomaremos los valores utilizados en el concurso de la CoG [33], del mismo modo que se hizo en la sección 4.4. No obstante, como este límite de ciclos es muy holgado, lo más frecuente es que la partida no se extienda tanto y que, por tanto, haya algunas de las estrategias que no se lleguen a utilizar. Esto no supondría ningún problema en nuestro planteamiento, ya que si la partida se divide en suficientes fases sí que se llegaría a cambiar la estrategia. Sin embargo, es importante tenerlo en cuenta para no llegar al caso en el que se utiliza una única estrategia y, por tanto, se está trabajando de la misma forma que en el capítulo 4.

Con este planteamiento se consiguen bots más flexibles cuantas más fases tengan, pero se presenta un grave problema. Recordemos que había 19 200 combinaciones de comportamientos posibles para el bot *SingleStrategy*. Entonces, si ahora tenemos varias de estrategias en cada bot, digamos n, habría un total de $19\,200^n$ bots distintos.

En la sección 4.3 calculamos que no todas las posibles combinaciones de comportamientos conducían a estrategias distintas. Esto podía ocurrir, por ejemplo, si en una estrategia nunca se producía cierto tipo de unidad y por tanto su comportamiento se volvía irrelevante. Sin embargo, al considerar varias fases el cálculo de la sección 4.3 no es válido, debido a que en una fase anterior se pudieron producir unidades de ese tipo, y por tanto su comportamiento tiene efecto en la partida, aunque en la fase actual no se produzcan más. Por tanto, en esta ocasión es necesario considerar que hay 19 200 estrategias distintas en cada fase.

Queremos, igual que en el capítulo anterior, encontrar el bot que mejor funciona para un cierto tablero, pero ahora hay demasiadas posibilidades distintas como para probarlos todos como hacíamos en el capítulo 4.

5.2. Búsqueda de las mejores configuraciones

Queremos encontrar la mejor estrategia para cada fase, estando cada una de estas estrategias compuesta por un comportamiento para cada tipo de unidad. Para resolver el problema nos basamos en el procedimiento utilizado por el bot *Tiamat* [3], explicado detalladamente en el artículo de Mariño y otros autores [4]. A grandes rasgos, en dicho artículo se plantea el uso de un algoritmo genético para encontrar la mejor combinación de estrategias de entre una amplia colección para un determinado tablero.

En nuestro caso, el problema es similar, por lo que decidimos plantear una forma parecida de resolverlo, también mediante un algoritmo genético, aunque con algunas peculiaridades. Cada cromosoma estará compuesto por los seis comportamientos (uno para cada unidad) en cada una de las fases, de forma que se tendrán $6n$ genes

en cada uno. A continuación detallamos el funcionamiento del algoritmo genético que hemos implementado, que como ya se ha indicado, es ligeramente distinto del habitual (explicado en la sección 2.3).

5.3. Descripción del algoritmo genético

El principal inconveniente que presenta la utilización de un algoritmo genético tal y como está explicado en la sección 2.3 para resolver el problema que hemos planteado es que puede haber poca relación entre el *fitness* de los padres y el de los hijos, es decir, que tomando dos padres con una puntuación muy alta no tiene por qué generarse un hijo con una puntuación similar.

Pongamos un ejemplo con bots de una sola fase, es decir, *SingleStrategy* como los que hemos utilizado en el capítulo 4. Supongamos que estamos trabajando en el tablero *BasesWorkers24x24*, utilizado en la sección 4.4.2. Tomaremos dos de los bots que obtenían las puntuaciones más altas, cuyos comportamientos concretos para cada tipo de unidad se muestran en las dos primeras filas de la tabla 5.1. En dicha tabla, hemos abreviado el comportamiento de las bases TwoWorker por "TwoW." y el comportamiento de los trabajadores Harvester por "Harv.".

Base	Trab.	Cuartel	Ligero	Pesado	A rango	Victorias
TwoW.	Harv.	Light	Closest	Wait	Wait	28
TwoW.	Harv.	Heavy	Wait	Closest	Wait	28
TwoW.	Harv.	Light	Wait	Closest	Wait	0
TwoW.	Harv.	Heavy	Closest	Wait	Wait	2

Tabla 5.1: Ejemplo de cruce en el que los hijos empeoran el resultado de los padres.

Tras aplicar el algoritmo de la forma habitual (como se explica en la sección 2.3), con el cruce de un punto sobre el gen correspondiente al comportamiento del cuartel y sin ninguna mutación, se obtendrían los dos bots cuyos comportamientos se muestran en las dos últimas filas de la tabla 5.1. En la tabla se muestra también el número de victorias que obtenían todos estos bots en la sección 4.4.2. Como vemos, los progenitores obtenían la puntuación más alta, 28 victorias en 30 partidas, mientras que los dos hijos obtienen 0 y 2 victorias.

Esto está provocado por la sinergia que puede haber entre las fases de un cromosoma, que es lo que hace que un bot funcione bien, y que se rompe completamente al cortarlo. En el ejemplo esta relación se daba entre el comportamiento del cuartel y los comportamientos de las tropas. En el primer progenitor, el cuartel producía soldados ligeros y el comportamiento de éstos era Closest, que ofrece muy buenos resultados. Por otro lado, el comportamiento de los soldados pesados es Wait, que ofrece malos resultados, pero no influye porque no se generan soldados de este tipo gracias al comportamiento del cuartel. En cuanto al otro progenitor, sucede casi

lo mismo, solo que el comportamiento CLOSEST es para los soldados pesados y el WAIT para los ligeros. Como en su caso el cuartel genera soldados pesados, esta combinación funciona perfectamente. No obstante, al combinar ambos cromosomas deja de coindidir el comportamiento del cuartel con el comportamiento adecuado de las tropas, y se pasa a tener dos bots en los que los soldados que se generan en cada uno tienen el comportamiento WAIT.

Para solventar este problema hemos modificado el genético para tener una mayor variedad en cada generación entre la que elegir la siguiente. Así, en cada generación tenemos un conjunto en el que almacenamos la generación anterior, los hijos obtenidos tras la selección y el cruce y varias mutaciones de estos hijos. De entre los individuos que tenemos en este conjunto escogemos a los que tienen mayor puntuación de *fitness* para formar la siguiente generación.

Este proceso aumenta el coste computacional del algoritmo, ya que es necesario evaluar una cantidad considerablemente superior de individuos en cada generación, pero es una solución efectiva. Al haber almacenado la generación anterior en el conjunto de donde tomamos la siguiente, nos aseguramos de que en el peor de los casos, si todos los individuos nuevos generados durante el proceso son peores, tendremos exactamente la misma población. Además, esto elimina completamente la utilidad de un método de elitismo como el que se utiliza generalmente en algunos algoritmos genéticos, por el que los mejores individuos de la generación anterior pasan directamente a la siguiente. En nuestro caso, pasarán aquellos que sean mejores que los que se han generado, ya sea este conjunto la totalidad de la generación anterior si no se ha producido nada mejor, o ninguno en el caso de que hayan quedado completamente superados por sus descendientes.

No obstante, esta técnica presenta también algunos problemas. En primer lugar, al conservar los individuos de la generación anterior junto con sus hijos y sus mutaciones, cabe la posibilidad de que comiencen a aparecer individuos duplicados (por ejemplo por no haber mutado) y, en caso de que su *fitness* sea mayor que el de los demás, podrían acabar por conformar ellos solos la población, con lo que el algoritmo se estancaría. Para solucionar esto hemos obligado a que haya una cierta diversidad en cada generación. Así, hemos definido la distancia entre dos individuos como el número de diferencias entre sus genes y hemos establecido un mínimo de distancia entre cada individuo que se va a añadir a la población y los que ya se han escogido.

Una vez se ha descrito de forma general el funcionamiento y las peculiaridades de nuestro algoritmo genético, pasamos a explicar en detalle el proceso que va siguiendo.

En primer lugar, generamos la población inicial de forma aleatoria, obligando a que haya la distancia requerida entre los individuos como se indicó anteriormente. Para ello simplemente vamos generando estrategias aleatorias y uniéndolas para formar un individuo y, si al comprobar la distancia con el resto de individuos ya generados resulta no ser lo suficientemente distinto, lo descartamos. Llamaremos N al tamaño de esta población.

Una vez hecho esto comienza el bucle que se repite cada generación. En primer lugar, en la fase de selección, se escogen los individuos que serán los progenitores de la siguiente generación. Cada pareja de progenitores dará lugar a dos hijos, por lo que habrá tantos hijos como progenitores. Como se quieren tantos descendientes como individuos había en la generación previa, será necesario elegir a N progenitores. Aun así, esta elección no consiste simplemente en escoger a todos los individuos de la generación anterior, ya que permitimos que un mismo individuo sea escogido varias veces como progenitor.

Para escoger a los progenitores organizamos N torneos. Cada torneo consiste en seleccionar una cierta cantidad de individuos, fijada previamente en una constante, y tomar el que mayor *fitness* tiene de cada torneo. De esta forma, los individuos con una buena puntuación tienen mayor probabilidad de ser escogidos, pero no se elimina por completo la posibilidad de que aquellos con puntuaciones más bajas también sean progenitores.

Cuando hemos acabado la selección, pasamos al cruce. Aquí, los progenitores se van tomando de forma aleatoria por parejas y cada una da lugar a dos hijos. Es importante destacar que, en el caso de que N sea impar, se generarán $N - 1$ hijos, ya que quedará un progenitor sin pareja. Esto no tiene mayor importancia y no hace que la siguiente generación descienda, como veremos más adelante. Para el cruce, empleamos el de un punto clásico, esto es, tomamos un punto aleatorio del cromosoma e intercambiamos los genes a partir de éste.

A continuación pasamos a la mutación. Para favorecer la diversidad de la siguiente generación, repetimos el proceso de mutación una cierta cantidad de veces fijada previamente en una constante. Este proceso consiste en ir pasando por todos los genes de cada uno de los hijos que hemos generado en la fase anterior, y, bajo una cierta probabilidad, ir cambiándolos, o no, por un gen nuevo.

Una vez hemos hecho todo esto, pasamos a la evaluación de los individuos. Primero, formamos un conjunto con los individuos pertenecientes a la generación anterior, los hijos generados mediante el cruce y todas sus mutaciones obtenidas. Tener todos estos individuos aumenta drásticamente el coste computacional, ya que estamos evaluando como mínimo $3N$ individuos (generación anterior, hijos y al menos una mutación) cuando lo habitual sería evaluar sólo N por generación. Para aliviar ligeramente este aumento, almacenamos el *fitness* de cada individuo en un diccionario, de forma que si posteriormente se tuviera que volver a evaluar, no sería necesario volver a hacer el proceso y bastaría con tomar el valor almacenado. Una vez tenemos el conjunto, aplicamos la función de evaluación a los individuos cuyo *fitness* aún no conocíamos para averiguarlo. Los detalles específicos sobre las funciones de evaluación que hemos empleado se pueden encontrar en la sección 5.4.

Finalmente, una vez que conocemos el *fitness* de cada individuo, pasamos a seleccionar los individuos que formarán la siguiente generación. Para ello, vamos tomando a aquellos con mayor puntuación y que son suficientemente "distintos", en el sentido que se ha explicado anteriormente, a los que ya hemos escogido, hasta

completar los N necesarios. Puesto que entre los individuos que estamos ofreciendo para tomar están los de la generación anterior, que sí cumplían la restricción de distancia, es casi seguro que conseguiremos seguir manteniéndola. Aun así, podría suceder que uno de los hijos se pareciese demasiado a sus dos progenitores y que superase a ambos. Esto haría que se escogiera antes por tener mayor *fitness* pero impediría a su vez que cualquiera de los dos padres fuera escogido, llevando en el peor de los casos a la imposibilidad de mantener la restricción. En el caso de que esto suceda, se añadirá el individuo con mayor *fitness* de entre los que no se han añadido aún.

5.4. Funciones de fitness

Para dirigir el aprendizaje del algoritmo genético utilizamos funciones de *fitness* que evalúan el desempeño de un bot al acabar una partida, devolviendo siempre un valor positivo. El valor de *fitness* es más alto cuanto mejor haya sido el resultado del bot, por lo que el algoritmo genético trata de maximizarlo seleccionando en cada generación a aquellos individuos con mayor *fitness*.

La función más sencilla que se puede utilizar y que cumple con estas especificaciones es la que devuelve 1 cuando el bot ha logrado la victoria y 0 en otro caso. También se puede imitar el reparto de puntos de una liga de fútbol, asignando 3 puntos en caso de victoria, 1 en caso de empate y 0 en caso de derrota. El problema de estas funciones de *fitness* es que en caso de que varios bots hayan perdido al ser evaluados, no hay manera de distinguir si unos se han desenvuelto mejor que otros durante la partida, ya que todos tendrían un *fitness* igual a 0. Esto dificulta el aprendizaje del algoritmo, especialmente si al empezar el entrenamiento los bots de la población inicial no consiguen ganar a nadie. En este caso el algoritmo no puede discernir que bots son los mejores candidatos para pasar a la siguiente generación.

Para resolver este problema podemos utilizar el tiempo que ha durado la partida en la función de *fitness* para diferenciar, de entre los ganadores, cuáles han ganado más rápido y por lo tanto son potencialmente mejores y, de entre los perdedores, cuáles han aguantado más tiempo en la partida antes de ser derrotados. Basándonos en esta idea y en las propiedades de la función sigmoide $P(t) = \frac{1}{1+e^{-t}}$, hemos diseñado una función de *fitness* que asigna a cada bot un valor real entre 0 y 1, dando siempre mayor puntuación a un bot ganador que a uno perdedor. Además, entre los ganadores da más puntuación cuanto más corta haya sido la partida, mientras que entre los perdedores da mayor *fitness* cuanto más haya durado la partida. La fórmula es la siguiente:

$$F(t) = \begin{cases} \dfrac{L}{1 + e^{c\frac{t-M}{M}}} + (1 - L) & \text{si el bot gana} \\ \\ \dfrac{L}{1 + e^{-c\frac{t-M}{M}}} & \text{en otro caso} \end{cases}$$

En esta fórmula, t representa el ciclo en el que acaba la partida. El parámetro $0 < L \leq 0{,}5$ determina el rango de valores que puede obtener un bot, que está contenido en $(0,\ L)$ para el perdedor y $(1 - L,\ 1)$ para el ganador. M es el valor de t en el que la gráfica tiene el punto de inflexión. Por último, c determina la curvatura, de modo que para valores mayores de c la función es más pronunciada, mientras que para valores menores, es más progresiva.

Para la elección del parámetro M, como queremos que el punto de inflexión quede en el punto medio de la partida, determinamos que $M = \frac{T}{2}$, donde T es el límite de ciclos establecido para la partida.

En cuanto al parámetro L, hay que tener en cuenta que si se toma $L = 0{,}5$, un bot que pierda casi al límite de tiempo tendría prácticamente la misma puntuación que uno que haya ganado en la misma situación. Esto no es incorrecto *per se*, ya que el ganador tendrá siempre un *fitness* ligeramente mayor, pero puede que nos interese que la diferencia de puntuación entre ganar y perder sea más importante. Por ejemplo, imaginemos que para evaluar una serie de individuos los enfrentamos a cuatro bots incluidos en μ-RTS y calculamos su *fitness* total como la suma de los *fitness* obtenidos en cada enfrentamiento, con $L = 0{,}5$. El individuo 1 gana a los cuatro bots en partidas largas, por lo que consigue un *fitness* de 0,55 en cada partida, haciendo un total de 2,2. El individuo 2 gana a un bot rápidamente, obteniendo un *fitness* de 0,9, pero pierde contra los otros tres tras largas partidas, obteniendo 0,45 puntos en cada una y quedando el total en 2,25. En esta situación, un bot que gana todas las partidas puede obtener menos puntuación que uno que ha perdido en tres ocasiones. Si queremos reducir esta posibilidad, podemos tomar un L más pequeño, de forma que aparezca un umbral entre la máxima puntuación posible para un bot perdedor y la mínima para un bot ganador que premie a los ganadores.

En nuestros experimentos tomaremos $L = 0{,}45$ y $c = 5$. En la figura 5.1 se muestra la función de *fitness* con estos valores. En color verde se indica la puntuación cuando la partida acaba en victoria y, en azul, cuando acaba en derrota o empate.

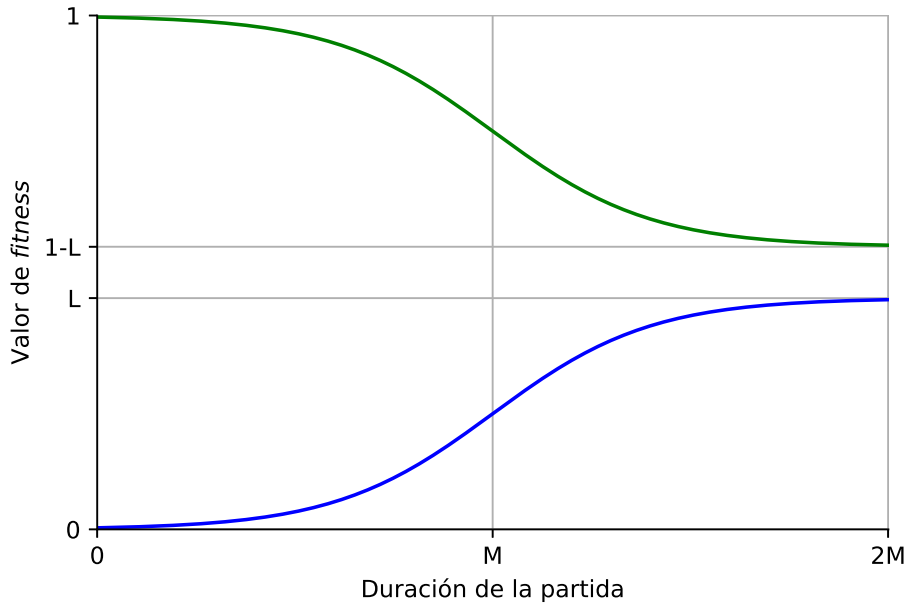

Figura 5.1: Función de evaluación.

Para evaluar a un bot con estas funciones de *fitness* en un algoritmo genético hay que simular un cierto número de partidas que se pueden jugar bien contra un conjunto fijo de bots de referencia o bien entre los propios individuos de la generación. Si se elige lo segundo, hay que tener en cuenta que las puntuaciones obtenidas por bots de distintas generaciones no son comparables, ya que se han evaluado enfrentándolos a rivales distintos.

5.5. Elección de los parámetros

Dentro del algoritmo genético hay multitud de parámetros que ajustar para lograr una configuración que encuentre los mejores individuos de forma eficaz. Para elegir los valores de estos parámetros hemos considerado diversos factores que se explicarán a continuación, en algunos casos incluso probando a ejecutar el algoritmo varias veces para comprobar qué funciona mejor.

En primer lugar debemos elegir el número de individuos que tendrá cada generación. Por la forma en que está diseñado nuestro algoritmo, es mejor que esta población no sea demasiado grande, ya que en cada generación, como se explicó anteriormente, se evalúa tanto a ésta como a sus hijos y sus mutaciones. Finalmente nos hemos decantado por tomar poblaciones de 10 individuos, ya que es un valor suficientemente bajo como para que el coste computacional no aumente demasiado, pero permite la diversidad necesaria para que el algoritmo pueda avanzar.

Otro parámetro que influye en este aumento del coste es el número de veces que se aplica el proceso de mutación a los hijos obtenidos a partir de una generación. En un genético normalmente sólo se aplica la mutación una vez, pero en este dominio nos interesa que el algoritmo analice una gran variedad de posibilidades, por lo que repetimos este proceso tres veces, de forma que en total, a la hora de escoger la siguiente generación, tendremos cinco veces más individuos de los requeridos entre los que elegir. Por ejemplo, si tenemos nuestra población de 10 individuos, para escoger a los miembros de la siguiente generación tendremos disponibles 50 individuos: los 10 de la generación anterior, sus 10 hijos sin mutar y 3 mutaciones de cada hijo, es decir, otros 30 individuos.

También se debe considerar la probabilidad de mutación de cada gen. Hemos decidido tomar para este parámetro un valor de 0,3. Aunque puede parecer que este valor es demasiado alto, lo cierto es que nos interesa que los hijos muten fácilmente, ya que también hemos introducido los hijos sin mutar en el conjunto del que luego tomaremos la siguiente generación. De esta forma, la probabilidad de que no mute ninguno de los genes de un cromosoma es de $0{,}7^{6N}$, siendo N el número de fases. Podemos observar que para el menor valor posible de N, $N = 1$, esta probabilidad es tan solo de 0,11.

En cuanto al tamaño del torneo, teniendo en cuenta que trabajaremos con poblaciones de 10 individuos, hemos decidido que su tamaño sea de dos. Es decir, para escoger a cada progenitor tomaremos de forma aleatoria dos bots, pudiendo elegir el mismo, y nos quedaremos con el mejor de ellos. Así, la probabilidad de que el bot con la puntuación más baja de la generación sea escogido al menos una vez para formar parte del grupo de los progenitores de la siguiente generación es de sólo un 0,01, lo cual lo hace difícil pero no imposible.

Otro parámetro a determinar es el número de partidas que jugaremos para resolver cada enfrentamiento. Como vimos cuando explicábamos el funcionamiento de μ-RTS, los movimientos de ambos jugadores tal y como los ofrece la clase *AbstractionLayerAI*, que estamos utilizando, no son simétricos, por lo que no es igual jugar como jugador 1 que como jugador 2. Por tanto, lo ideal es que el número de enfrentamientos sea par, con el fin de que se hayan jugado las mismas partidas en ambas posiciones, y tomaremos el mínimo valor adecuado, es decir 2 partidas por enfrentamiento, para mantener esta justicia sin aumentar demasiado el coste.

Para ajustar el resto de parámetros hemos ejecutado el algoritmo genético utilizando con los parámetros que ya hemos especificado y 20 generaciones sobre el mapa *BasesWorkers24x24*. La configuración por defecto para el número de fases será 5, y para calcular el *fitness* de cada individuo utilizaremos la función que tiene en cuenta la duración de la partida enfrentándolo con los catorce bots básicos incluidos en el juego que usábamos en el capítulo anterior.

En primer lugar, hemos comprobado que las variaciones en el número de diferencias requeridas no afectan al *fitness* del resultado final del genético. Tanto eliminando esta restricción como exigiendo una diferencia de al menos 4 de los 6 genes de cada

fase se obtiene un fitness máximo de 27,08 en el mapa *BasesWorkers24x24*. Cuando el número de diferencias requeridas es 0, la generación final está compuesta por 10 bots exactamente iguales. Al exigir cierta diferencia entre los individuos de cada generación se logra una mayor variedad, sin embargo, una restricción mayor también provoca que alcanzar este máximo requiera más generaciones. Esto podría utilizarse, por ejemplo, para tener un conjunto de bots más variado entre los que pudiera elegir después otro tipo de inteligencia. En los siguientes experimentos, estableceremos el número de diferencias requeridas en 4 veces el número de fases del cromosoma. Este número puede parecer demasiado alto teniendo en cuenta que el número total de genes es 6 veces el número de fases. Sin embargo, como se explicó detalladamente en la sección 4.3, hay muchas combinaciones de comportamientos que pueden llevar a la misma estrategia, por lo que es necesario exigir una gran diferencia para que sea más probable que los dos cromosomas jueguen realmente de formas diferentes.

También hemos comprobado que el algoritmo aprende tanto con la función que sólo cuenta las victorias como con la que considera la duración de la partida, es decir, que en cada generación se produce una mejora de los resultados. En cualquier caso, considerar la duración de la partida permite una evaluación de los bots más precisa, por lo que seguiremos utilizando esta función.

Por último, hemos estudiado la influencia del número de fases en el aprendizaje. De nuevo, el genético es capaz de encontrar bots con un fitness de 27,08 para el mapa *BasesWorkers24x24* tanto eligiendo 5 fases como eligiendo 10 o 20, aunque se requieren más generaciones para alcanzar el máximo al incrementar el número de fases. En principio, cuantas más fases tenga el bot más versátil podrá llegar a ser, pero tampoco tiene sentido elegir un número excesivo de fases, del mismo modo que no tiene sentido cambiar de estrategia continuamente durante una partida. No obstante, como se indicó en la sección 5.1, normalmente no se llegará a utilizar todas las fases, ya que es muy raro que la partida se extienda hasta el límite de ciclos. Es necesario tener esto en cuenta a la hora de elegir el número de fases para no terminar jugando con menos estrategias de las que queremos.

5.6. Resultados

A continuación, probaremos el algoritmo genético en diversos mapas utilizando bots con varias fases. Como oponentes para calcular el *fitness*, probaremos tanto a utilizar la selección de 14 bots incluidos en el código del juego que ya empleamos en el capítulo anterior como a enfrentar a cada individuo contra el resto de su generación. Es importante tener en cuenta que la opción elegida influirá en el tiempo de ejecución, ya que cambia el número de partidas que es necesario jugar para evaluar la generación. Cuando se usa la selección de bots, se juegan 28 partidas para cada individuo de la generación, lo que hace un total de 1400 partidas. En el caso en el que calculemos el *fitness* enfrentando a cada individuo con el resto, como hay 50 individuos en cada generación, cada uno de ellos jugaría 2 partidas contra los 50, de modo que en total se jugarían 5000 partidas.

Hemos prescindido de bots que consuman todo el tiempo de cómputo en cada ciclo, debido a que alargan excesivamente la duración del entrenamiento genético. Los bots de este tipo desarrollados para μ-RTS requieren un tiempo de cómputo de $100\,ms$ por ciclo, lo que alarga cada partida durante varios minutos, especialmente en mapas grandes. Si estimamos la duración de una partida en 3 000 ciclos, ésta se puede alargar $300\,000\,ms$, que son exactamente 5 minutos. Si tenemos en cuenta que en cada generación se evalúan como mínimo 50 bots y hay que jugar en ambos lados del tablero, las partidas contra uno sólo de estos bots de una generación se alargarían 500 minutos, que equivalen a más de 8 horas. Esto hay que multiplicarlo por el número de generaciones, por lo que aún empleando paralelismo se sale de nuestras posibilidades.

Una vez terminado el genético, tomaremos el bot vencedor y lo enfrentaremos en 10 partidas contra cada uno de los 5 bots de referencia para dar una conclusión final sobre su rendimiento. Estos 5 bots serán: *Droplet* [34] (basado en el artículo [35]), que alcanzó el tercer puesto en el concurso de μ-RTS de la CoG de 2019, *Tiamat* [3] (basado en los artículos [4] y [36]), ganador de la edición de 2018 del mismo concurso, y los bots básicos *NaiveMCTS* (basado también en el artículo [35]), *LightRush* y *WorkerRushPlusPlus* incluidos en el juego (ver sección 3.8). Hemos elegido a estos tres últimos ya que se incluyen habitualmente como referencia junto a *RandomBiased* en el concurso de μ-RTS de la CoG [2].

5.6.1. Mapa de 8x8

Comenzaremos con el mapa *BasesWorkers8x8* en el que entrenaremos el genético con bots de 5 fases. Al igual que en la sección 4.4.1, tomaremos un máximo de 3000 ciclos para cada partida basándonos en las reglas de la CoG [33]. Como ya hemos indicado, durante el genético probaremos a evaluar los individuos primero enfrentándolos a los 14 bots básicos y luego enfrentándolos entre sí. Al utilizar los bots incluidos en el juego para calcular el *fitness*, el mejor de los bots en la última generación obtiene una puntuación de 26,68 y se obtienen los valores de *fitness* medio mostrados en la figura 5.2.

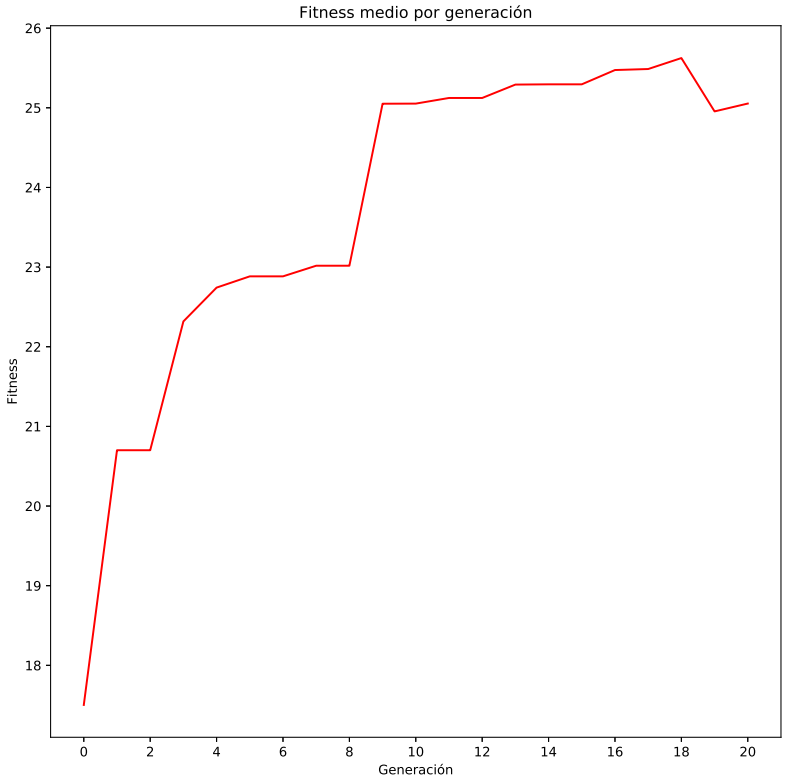

Figura 5.2: *Fitness* medio obtenido en cada generación para el mapa de 8x8.

En la gráfica puede llamar la atención que en las últimas generaciones el *fitness* medio baje. Esto se debe a la restricción de diversidad que hemos impuesto en nuestro algoritmo genético (sección 5.3). Si dos bots que deberían pasar a la siguiente generación se parecen demasiado, se descarta el peor de ellos, pudiendo pasar algún bot con menor *fitness* en su lugar.

Nótese que cuando calculamos el *fitness* enfrentando a cada individuo con el resto de su generación no tiene sentido representar una gráfica con estos valores, ya que un mismo bot podría tener mayor o menor *fitness* en la generación siguiente según qué individuos se hayan obtenido al hacer el cruce y las mutaciones.

Respecto al tiempo de ejecución, jugar 100 partidas en este mapa requieren una media de 77 ms. Esto supone que, cuando se utiliza la selección de bots, cada generación tarda aproximadamente 1 072 ms en evaluarse y por tanto las 20 generaciones requieren un total de unos 20 segundos. Por otro lado, si usamos al resto de la generación para evaluar, se necesitarán 3 828 ms por generación y total de 76,58 s para terminar las 20 generaciones que utilizamos.

A continuación, enfrentamos a los *MultiStrategy* elegidos tras el genético contra los 5 bots de referencia. Los resultados se muestran en la tabla 5.2. Las columnas con el rótulo "Evaluación Externa" corresponden al *MultiStrategy* ganador del genético en el que el fitness se ha evaluado con los bots básicos del juego, es decir, con bots

ajenos al algoritmo genético. Las columnas marcadas como "Evaluación Interna" se corresponden con la ejecución del genético en el que se han evaluado los individuos enfrentándolos al resto de su propia generación.

	Evaluación Externa		Evaluación Interna	
Rival	Victorias	Puntuación	Victorias	Puntuación
WorkerRushPlusPlus	10	9.63	0	3.36
LightRush	10	9.92	10	9.92
NaiveMCTS	10	9.77	10	9.92
Droplet	4	4.27	4	5.25
Tiamat	10	9.63	0	3.36

Tabla 5.2: Resultados obtenidos por *MultiStrategy* en el mapa *BasesWorkers8x8*

Observamos que al flexibilizar nuestros bots dotándolos de varias fases existen configuraciones que logran ganar a *WorkerRushPlusPlus* siempre, algo que ninguna configuración de una sola fase logró en el mapa de 8x8 (ver sección 4.4.1). Esto se consigue en el genético en el que se ha utilizado el método de evaluación externa para calcular el fitness, es decir, en el que cada individuo se ha evaluado enfrentándolo a los 14 bots básicos.

Sin embargo, los resultados logrados con la evaluación interna, entrenando sólo con los bots de la generación, son claramente inferiores a los obtenidos al entrenar con los 14 bots prefijados. Vemos que no ha sido capaz de ganar ni una sola vez a *WorkerRushPlusPlus* ni a *Tiamat*, que en este mapa desarrollan estrategias exactamente iguales. Esto de debe a que entre los individuos generados en el genético no ha aparecido ninguno lo suficientemente similar a *WorkerRushPlusPlus* como para aprender a vencerlo y la evolución se ha podido estancar en un máximo local.

Además, al enfrentar en 10 partidas al bot obtenido con evaluación externa contra el de evaluación interna, resulta que el de evaluación externa gana en las 10 ocasiones.

Todo ello nos lleva a pensar que, aunque existen configuraciones de *MultiStrategy* capaces de derrotar a *WorkerRushPlusPlus* en el mapa de 8x8, al evaluar a los individuos del genético mediante evaluación interna el aprendizaje se puede estancar en un máximo local con mayor facilidad. Esto puede ocurrir porque en un determinado momento los individuos de una generación pierden diversidad, de forma que ninguno destaca sobre los demás. En esta situación, aquellos individuos con la mayor puntuación solo consiguen vencer a algunos de sus rivales, ya que las estrategias que usan no son lo suficientemente sofisticadas. Otra posibilidad es que dentro de una generación haya individuos que destaquen y sean capaces de ganar a la mayoría de sus rivales, pero que las estrategias que les han servido para lograrlo no sean lo suficientemente buenas como para ganar a *WorkerRushPlusPlus*. Si esta situación se prolonga en el tiempo y no aparece algún individuo mejor, el aprendizaje se estancará.

En contraposición a esto, incluir a *WorkerRushPlusPlus* en la evaluación permite identificar si alguno de los individuos aguanta más tiempo frente a él aunque pierda, obteniendo así una mayor puntuación. De esta forma, se va mejorando la capacidad para enfrentarse a *WorkerRushPlusPlus* poco a poco, generación tras generación, guiando el aprendizaje hacia un bot capaz de vencerlo.

5.6.2. Mapa de 24x24

Nuestro segundo experimento se desarrolla en el mapa *BasesWorkers24x24*, de nuevo con bots de 5 fases. El número máximo de ciclos por partida será, como se establece en las reglas de la CoG [33], de 5000 ciclos. Como en el caso anterior, empezamos evaluando el *fitness* enfrentando a los individuos contra los 14 bots del juego. El *fitness* máximo obtenido en cada generación viene representado en la gráfica 5.3. El mejor individuo obtenido en la última generación consiguió 26,94 puntos.

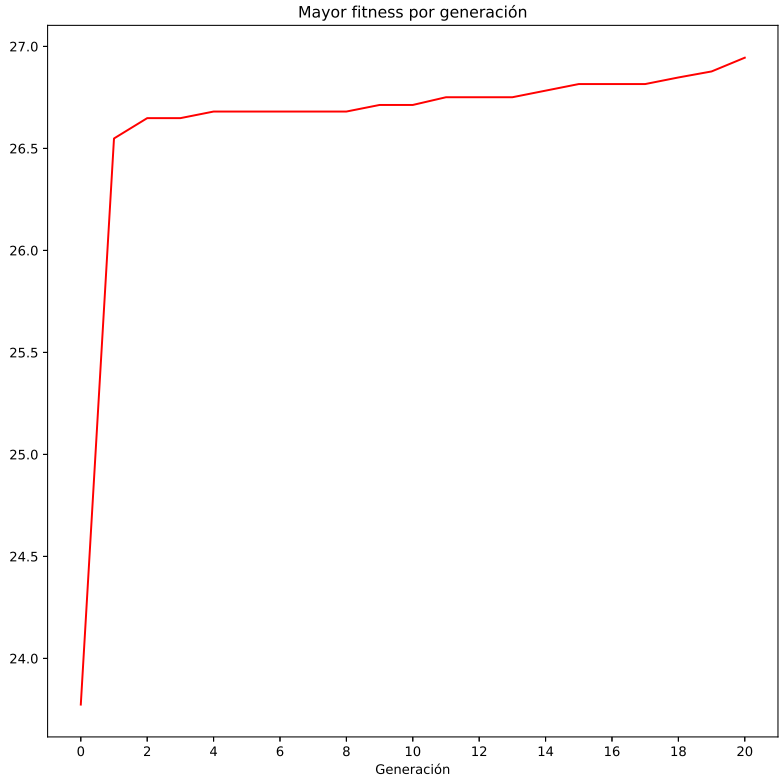

Figura 5.3: *Fitness* máximo obtenido en cada generación para el mapa de 24x24.

Repetimos el proceso pero evaluando el *fitness* contra los otros individuos de la generación. Al enfrentar a los *MultiStrategy* resultantes contra los 5 bots en la evaluación final el resultado se muestra en la tabla 5.3.

	Evaluación Externa		Evaluación Interna	
Rival	Victorias	Puntuación	Victorias	Puntuación
WorkerRushPlusPlus	10	9.79	10	9.46
LightRush	10	9.46	10	9.63
NaiveMCTS	10	9.56	10	9.4
Droplet	7	7.06	9	8.3
Tiamat	8	7.78	6	6.02

Tabla 5.3: Resultados obtenidos por *MultiStrategy* en el mapa *BasesWorkers24x24*

A diferencia de lo que sucedía en el mapa de 8x8, cuando calculamos el *fitness* mediante evaluación interna, se obtienen unos resultados finales tan buenos como al utilizar evaluación externa. El motivo es que en el otro mapa, la estrategia ganadora debía ser sumamente parecida a *WorkerRush*, y al ser tan específica era más difícil encontrarla sin una guía más directa como la que ofrecía enfrentarse a ese mismo bot. En este caso, hay muchas más estrategias con potencial para ganar, es decir, muchas más formas de conseguir un bot bueno, por lo que no es necesario ofrecerle un aprendizaje tan guiado.

Además, al enfrentar en 10 partidas al bot obtenido con evaluación externa contra el de evaluación interna, resulta que el de evaluación interna gana en las 10 ocasiones. Esto no significa que el genético con evaluación interna sea siempre mejor al de evaluación externa. En el caso del tablero de 24x24, hay una gran cantidad de posibles *MultiStrategy* que pueden dar muy buenos resultados. Tal y como se observaba en la tabla 5.3, tanto con evaluación interna como externa se puede alcanzar un bot capaz de desenvolverse razonablemente bien y, por tanto, es posible que en otra ejecución del algoritmo genético se obtengan dos bots tales que el de evaluación externa gane al de interna.

En cuanto a los tiempos de ejecución, jugar 100 partidas en este tablero requiere aproximadamente $580\,ms$, por lo que evaluar una generación completa cuando se utiliza evaluación externa requerirá aproximadamente 8 segundos. De este modo, ejecutar las 20 generaciones que utilizamos en nuestro genético requerirá aproximadamente 160 segundos. Si utilizamos evaluación interna, cada generación tardará unos $30\,s$, y el total del genético necesitará 15 minutos.

5.6.3. Mapa de 64x64

Terminamos el estudio en el mapa más grande incluido en μ-RTS, *GardenOfWar64x64*, que se puede observar en la figura 5.4. En este mapa, los jugadores disponen de una base con tres bloques de recursos cercanos y un trabajador. Estas unidades se ubican en dos de las esquinas del tablero, una para cada jugador, pero en la zona central hay también una gran cantidad de casillas de recursos sobre las

que ningún jugador tiene el control. Para este mapa el límite de ciclos establecido en las reglas del concurso de la CoG es de 8000 ciclos, mucho mayor que en los mapas que hemos empleado hasta ahora, por lo que entrenaremos un bot con 10 fases.

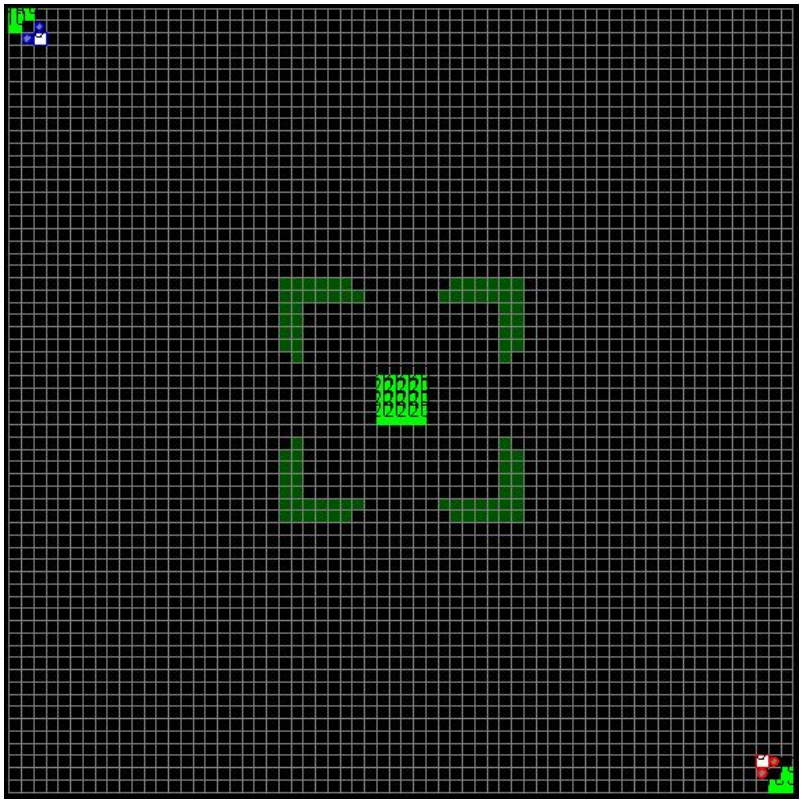

Figura 5.4: Mapa *GardenOfWar64x64* incluido en el juego.

En primer lugar, ejecutamos el genético que calcula la función de *fitness* con los 14 bots de μ-RTS, cuyo vencedor consiguió un *fitness* de 25,82. En la figura 5.5 se muestra el *fitness* medio a medida que pasan las generaciones.

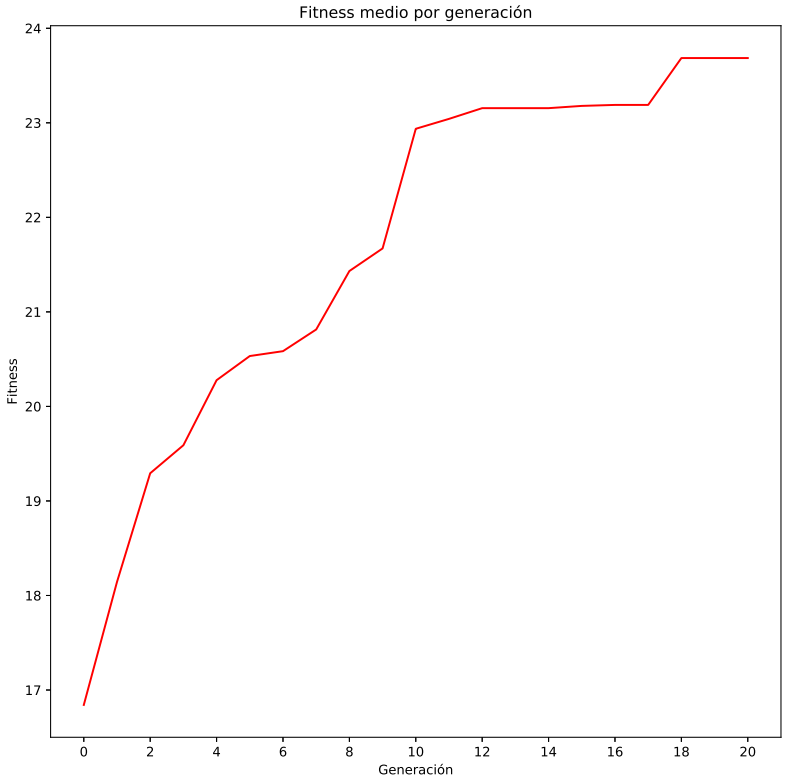

Figura 5.5: *Fitness* medio obtenido en cada generación para el mapa de 64x64.

Repetimos la ejecución evaluando los individuos de la generación enfrentándolos entre sí y tomamos al ganador. Los resultados obtenidos por ambos al enfrentarse a los 5 bots de referencia están recogidos en la tabla 5.4.

	Evaluación Externa		Evaluación Interna	
Rival	Victorias	Puntuación	Victorias	Puntuación
WorkerRushPlusPlus	10	9.13	10	9.13
LightRush	9	8.34	10	8.79
NaiveMCTS	10	8.89	10	8.27
Droplet	9	8.17	10	8.86
Tiamat	10	8.92	10	9.13

Tabla 5.4: Resultados obtenidos por *MultiStrategy* en el mapa *GardenOfWar64x64*

En cuanto a los tiempos de ejecución, el gran tamaño de este mapa hace que las partidas se vuelvan considerablemente más largas. Así, cien partidas tardan en ejecutarse aproximadamente 7,25 s, lo que implica que cada generación tardará 101 s en evaluarse con la evaluación externa y 362 s con la interna. Por tanto, la ejecución completa de las 20 generaciones requerirá unos 33 minutos al usar la evaluación externa y aproximadamente 2 horas al usar la interna.

Podemos observar que los resultados obtenidos son muy buenos tanto al usar evaluación externa como al usar evaluación interna. En el primer caso se ganan todas las partidas excepto dos, y en el segundo caso se vence en todas las partidas. Recordemos que en el tablero de 8x8 la evaluación interna funcionaba peor porque necesitábamos que los bots se parecieran lo más posible al *WorkerRush*, y por tanto enseñarlos a vencer a éste era la mejor forma de conseguirlo. Ahora, con un tablero tan grande, la estrategia ganadora no está tan definida, y dejar que los bots se entrenen libremente compitiendo contra otros bots generados durante el genético funciona mejor que en el mapa de 8x8. Además, es de esperar que el bot resultante de la evaluación interna sea más genérico, mientras que con evaluación externa puede quedar más sesgado, en el sentido de que se especializa en vencer a un conjunto prefijado de bots. Si los bots elegidos no son lo suficientemente representativos, la evaluación externa puede no ser tan eficaz, por lo que si hubiera que entrenar el genético en un mapa completamente nuevo, en el que no supiéramos qué bots se desenvuelven mejor, la evaluación interna sería la mejor opción.

En este caso, al enfrentar a los dos bots entre sí en diez partidas, comprobamos que el bot obtenido con evaluación interna consigue ganar ocho de ellas. Del mismo modo que en el tablero de 24x24, hay una gran cantidad de *MultiStrategy* capaces de obtener muy buenas puntuaciones, por lo otra ejecución del genético podría haber conducido a estrategias diferentes en las que el bot entrenado con evaluación externa ganase al de evaluación interna.

En definitiva, estos resultados nos conducen a pensar que un mapa muy pequeño restringe la libertad de los jugadores a la hora de confeccionar estrategias que los conduzcan a la victoria. Concretamente en el mapa de 8x8 la estrategia dominante emplea exclusivamente trabajadores, y cualquier cosa que se salga de esto empeora el rendimiento del jugador. A medida que el mapa es más grande, las estrategias viables se diversifican, permitiendo a los jugadores utilizar otros tipos de tropa y desarrollar estrategias distintas pero con igual potencial. Hemos comprobado también que la evaluación interna funciona mejor en mapas grandes que en el pequeño de 8x8, precisamente gracias a la existencia de una mayor variedad de estrategias con potencial ganador. Además, tiene la ventaja de no depender de bots externos para la evaluación. La evaluación externa obtiene buenos resultados en todos los mapas, siempre y cuando se le suministren bots que sepan jugar bien en ellos.

Observamos que en los mapas más grandes no hay una estrategia dominante que gane siempre, sino que existen estrategias adecuadas para enfrentarse a ciertos rivales, pero que pueden perder contra otros. Además, puede ocurrir que al modificar la estrategia para conseguir ganar a algún rival nuevo, se deje de ganar contra otros. Por tanto, en mapas en los que ocurra esto, no basta sólo con encontrar estrategias adaptadas al mapa, sino que es necesario adaptarse también al rival y ser capaz de identificar lo que hace para responder adecuadamente. Esto está fuera de las capacidades de *MultiStrategy*, aunque no impide que sea capaz de aprender una estrategia con potencial ganador durante el entrenamiento en el mapa.

5.7. Viabilidad de una búsqueda aleatoria

Antes de mostrar los resultados obtenidos en esta sección, es necesario recordar que μ-RTS es una versión muy simplificada de los juegos RTS, como se explicó en la sección 3.5. Esto hace que, aunque computacionalmente el juego siga siendo muy complejo, la jugabilidad esté muy limitada. Por tanto, teniendo unos comportamientos adecuados para cada unidad, como los que hemos definido nosotros, es relativamente sencillo combinarlos para generar un bot capaz de jugar razonablemente bien.

Además, en el cálculo del *fitness* para evaluar a los bots *MultiStrategy*, empleamos los bots básicos del juego, explicados en la sección 3.8, que no utilizan estrategias más elaboradas que poner algún trabajador a recolectar y a las tropas o al resto de trabajadores a atacar. Nuestros *MultiStrategy* construyen estrategias a partir de comportamientos que ya contemplan de por sí usar trabajadores para recolectar o atacar y llevar a las tropas a atacar. Por ello, salvo que hayan tomado comportamientos incompatibles entre sí, la mayoría de configuraciones consiguen plantar cara a los bots básicos y muchas de ellas los vencen por usar un tipo de tropa mejor, distribuir sus trabajadores de forma más eficiente o ambas cosas. Un ejemplo de comportamientos incompatibles es el uso simultáneo de RushWorker en la base y Harvester en los trabajadores, que provoca situaciones de bloqueo alrededor de los yacimientos de recursos (ver sección 4.1). No obstante, la mayoría de combinaciones de comportamientos no presentan esta clase de incongruencias, por lo que llegan a obtener fácilmente puntuaciones superiores a los 24 puntos sobre 28 cuando se evalúan con los 14 bots que usábamos para calcular el *fitness* del genético.

Vamos a comparar ahora los resultados obtenidos con el algoritmo genético con los que se podrían obtener de evaluar la misma cantidad de bots pero tomándolos de forma aleatoria.

Recordemos que hacíamos 20 generaciones y elegíamos los 10 individuos de cada una de entre un total de 50 por generación. Así pues, el genético habría llegado a evaluar 1010 individuos al añadir los 10 de la generación inicial. Por tanto, vamos a generar 1010 individuos de forma completamente aleatoria y a evaluar su rendimiento. Para hacer esta evaluación, los compararemos con los 14 bots que usábamos para calcular el *fitness* de cada generación y, finalmente, enfrentaremos al que mayor puntuación tenga con los 5 bots de referencia que utilizábamos.

En las gráficas, mostramos cuál ha sido el mayor fitness obtenido tras evaluar cierto número de bots tanto usando el genético como usando la búsqueda aleatoria. En azul mostramos el *fitness* máximo obtenido por el genético y en rojo, el obtenido con la búsqueda aleatoria.

5.7.1. Mapa de 8x8

Comenzaremos por el tablero *BasesWorkers8x8* utilizando bots de 5 fases. La elección aleatoria llega a obtener un *fitness* de 26,16, algo más bajo que el 26,68 obtenido con el genético, aunque con muy poca diferencia. En la figura 5.6 podemos observar como aumentan ambos valores a medida que se evalúan más bots.

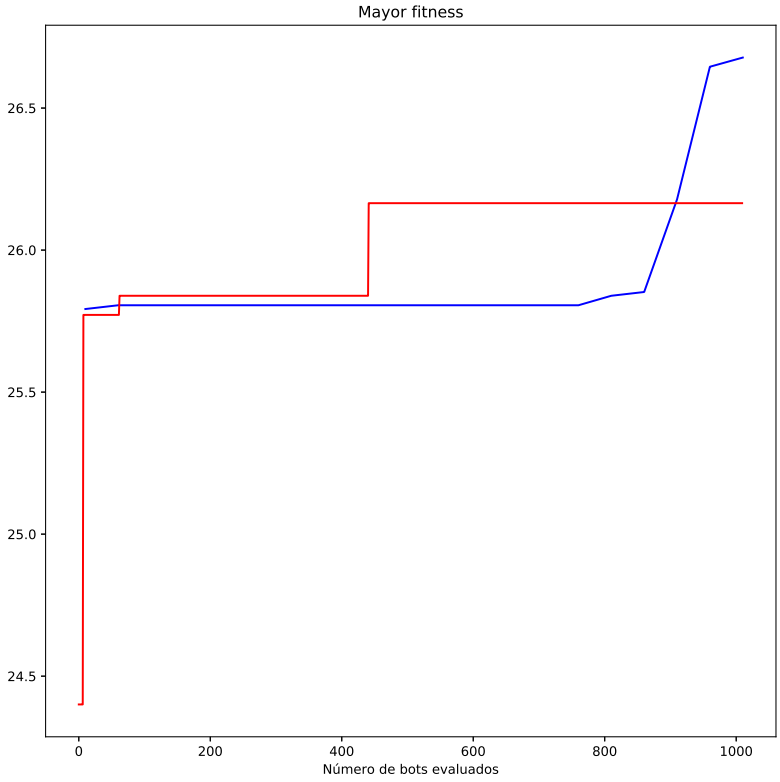

Figura 5.6: Comparación del *fitness* máximo en el tablero *BasesWorkers8x8*.

Cuando lo evaluamos contra los bots de referencia obtenemos los resultados mostrados en la tabla 5.5. En dicha tabla se muestran nuevamente los resultados obtenidos por el genético con evaluación externa para facilitar su comparación.

	Genético		Aleatorio	
Rival	Victorias	Puntuación	Victorias	Puntuación
WorkerRushPlusPlus	10	9.63	5	7.13
LightRush	10	9.92	10	9.92
NaiveMCTS	10	9.77	10	9.85
Droplet	4	4.27	1	1.34
Tiamat	10	9.63	5	7.13

Tabla 5.5: Resultados obtenidos en el mapa *BasesWorkers8x8*

Como vemos, son algo peores que los obtenidos por el genético. La razón es probablemente la misma que explicaba por qué entrenar el genético enfrentando a cada individuo al resto de su generación no funcionaba demasiado bien. En el caso del tablero de 8x8, la estrategia ganadora es muy específica, por lo que se necesita una búsqueda muy guiada en su dirección para encontrarla. Aun así, los resultados de la búsqueda aleatoria resultan bastante satisfactorios, ya que consigue ganar al menos la mitad de las partidas contra todos los rivales.

Si examinamos las puntuaciones obtenidas por todos los bots evaluados durante la búsqueda aleatoria, resulta que tan solo 2 (0,19 %) obtuvieron una puntuación superior a 26, y ninguno logró 27 puntos. Esto refuerza nuestra teoría, que las estrategias buenas en este mapa son pocas y por tanto más difíciles de encontrar eligiéndolas aleatoriamente.

5.7.2. Mapa de 24x24

En el tablero *BasesWorkers24x24* obtenemos un *fitness* máximo de 27,08, algo más alto que el 26,94 obtenido con el genético. En la figura 5.7 se observa la evolución del valor máximo obtenido por ambos métodos.

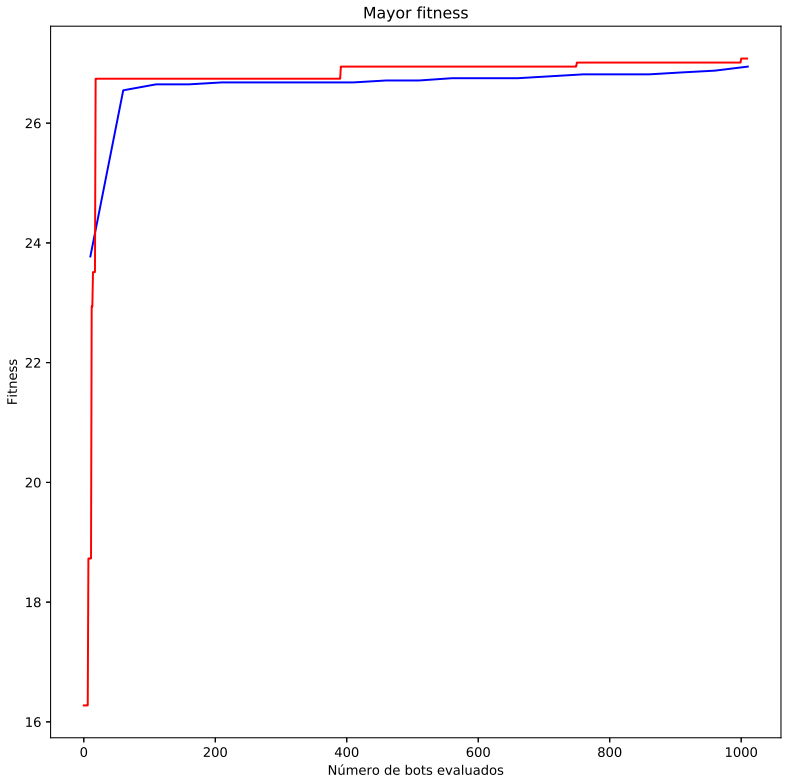

Figura 5.7: Comparación del *fitness* máximo en el tablero *BasesWorkers24x24*.

Al enfrentar el mejor de estos bots contra los 5 de referencia, no obstante, se obtienen unos resultados muy parecidos a los del genético, como se muestra en la tabla 5.6.

	Genético		Aleatorio	
Rival	Victorias	Puntuación	Victorias	Puntuación
WorkerRushPlusPlus	10	9.79	10	9.79
LightRush	10	9.46	10	9.46
NaiveMCTS	10	9.56	10	9.63
Droplet	7	7.06	8	7.85
Tiamat	8	7.78	7	6.69

Tabla 5.6: Resultados obtenidos en el mapa *BasesWorkers24x24*

El éxito de la búsqueda aleatoria en este caso se debe a que a pesar de que el espacio de búsqueda sea enorme, hay muchas combinaciones de estrategias que funcionan bien. Esto permite que probando al azar la probabilidad de dar con un buen bot sea relativamente alta. En el mapa de 24x24, de los 1010 bots evaluados, 29 (2,87 %) obtuvieron una puntuación superior a 26, de los que 2 bots consiguieron una puntuación mayor que 27. Si aumentáramos la complejidad de los bots, añadiendo nuevas estrategias o aumentando el número de parámetros que afectan a un comportamiento, es probable que la proporción de bots sobresalientes se redujese, haciendo así más necesaria una búsqueda guiada como la que nos proporciona el algoritmo genético.

5.7.3. Mapa de 64x64

Por último probamos este método en el mayor de los mapas que ofrece el juego, *GardenOfWar64x64*. Utilizaremos, del mismo modo que hacíamos con el genético, bots de 10 fases para que puedan adaptarse mejor a la larga duración que previsiblemente tendrá una partida en un mapa tan grande. El mayor *fitness* obtenido con la búsqueda aleatoria es de 24,55, mientras que para el algoritmo genético se obtenía 25,82. En la figura 5.8 observamos, como es habitual, el avance del máximo *fitness* según el número de bots evaluados tanto en el genético como en la búsqueda aleatoria.

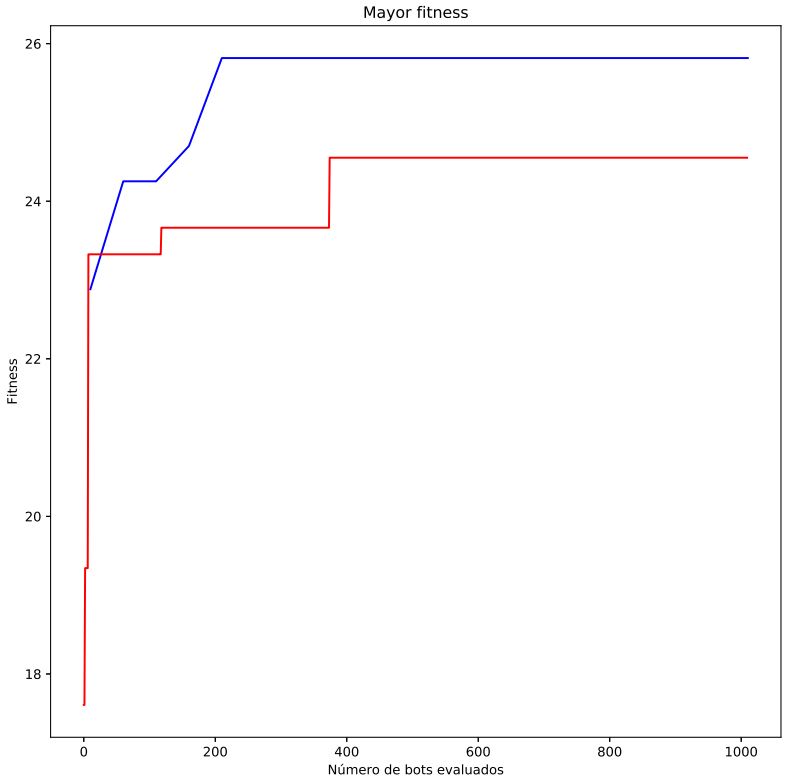

Figura 5.8: Comparación del *fitness* máximo en el tablero *GardenOfWar64x64*.

Los resultados obtenidos cuando enfrentamos el mejor bot contra los de referencia se muestran en la tabla 5.7.

	Genético		Aleatorio	
Rival	Victorias	Puntuación	Victorias	Puntuación
WorkerRushPlusPlus	10	9.13	10	9.12
LightRush	9	8.34	10	8.79
NaiveMCTS	10	8.89	9	7.11
Droplet	9	8.17	10	8.99
Tiamat	10	8.92	10	9.02

Tabla 5.7: Resultados obtenidos en el mapa *GardenOfWar64x64*

Estos resultados vuelven a mostrar que el espacio de búsqueda tiene muchas estrategias buenas, por lo que aun eligiendo aleatoriamente se obtiene un bot capaz de ganar a los 5 bots de referencia. Además de esto, observamos que el mejor bot obtenido en la búsqueda aleatoria alcanza un *fitness* algo más bajo que el genético contra los 14 bots, lo cual no quiere decir que no les gane, sino que tarda más tiempo, pero tiene un resultado igual de bueno contra los 5 bots de referencia que el obtenido por el bot ganador del genético. Esto nos lleva a concluir que, para el mapa de 64x64,

los 14 bots elegidos para evaluar el *fitness* no son los óptimos, ya que hemos visto que un bot con peor puntuación para estos bots consigue unos resultados igual de buenos al enfrentarse a los 5 bots de referencia.

Conclusiones

En este capítulo hemos conseguido desarrollar un bot mucho más complejo, *MultiStrategy*, capaz de ir cambiando de estrategia a medida que avanza la partida. Tras el consecuente aumento del espacio de búsqueda, ya que ahora hay muchas más combinaciones de comportamientos posibles, hemos utilizado un algoritmo genético para buscar los bots más prometedores, y hemos comprobado que en efecto estos bots consiguen ganar a algunos de los participantes del concurso de la CoG de otros años.

Hemos comprobado que el tamaño del mapa influye en cómo son las estrategias más exitosas. En el mapa más pequeño, las posibilidades están más restringidas y las estrategias basadas en atacar con los trabajadores dominan con claridad. Sin embargo, a medida que crece el mapa crecen también la libertad para usar las tropas y la variedad de estrategias con potencial. Existen multitud de formas de componer el ejército que se desenvuelven bien y no hay una estrategia que domine sobre las demás, sino estrategias adecuadas para enfrentarse a ciertos rivales pero que pierden contra otros y viceversa.

Esto hace que emplear bots externos durante la evaluación en mapas más grandes pueda sesgar el aprendizaje, haciendo que el *MultiStrategy* resultante se especialice en ganar rápidamente a esos bots, pero obteniendo quizá un peor resultado contra otros rivales. Por otro lado, no tener ningún bot externo en la evaluación hace que no exista una referencia fija para comparar la evolución del *fitness* entre generaciones y puede desembocar en un estancamiento durante el aprendizaje si los individuos analizados son demasiado uniformes. Una posibilidad sería encontrar un equilibrio entre las dos formas de evaluar, en la que se enfrente a los individuos de la generación entre sí pero añadiendo algunos bots externos que se consideren representativos. De este modo el genético se beneficiaría del aprendizaje más genérico de la evaluación interna y de la guía que ofrecen los bots externos para comprobar el progreso entre generaciones.

Por otro lado, en la sección 5.7 hemos comprobado que una búsqueda aleatoria también consigue buenos resultados. Esto indica que la mayoría de combinaciones de nuestros comportamientos son suficientemente buenos como para ganar a muchos de los bots predefinidos que hemos usado para evaluar los resultados. Podría parecer pues el algoritmo genético no es necesario, sin embargo también hemos visto en esta última sección que al utilizarlo se consigue la pequeña mejora necesaria para que el bot obtenido pueda vencer a los bots más difíciles que se le resisten al aleatorio. Esto nos lleva a pensar que, de aumentar la complejidad de los bots a los que nos enfren-

tamos, la utilidad del algoritmo genético quedaría más que demostrada. Al hacer esto, la diferencia entre los resultados del genético y los del aleatorio aumentaría, lo que haría que el genético destacara mucho más.

Capítulo 6

Conclusión

Como se explicó al principio y se ha comprobado a lo largo del trabajo, la aplicación de la inteligencia artificial a los juegos de estrategia en tiempo real resulta especialmente interesante por la gran complejidad que ofrecen éstos. La gran cantidad de unidades que puede llegar a controlar cada jugador a la vez y el amplio abanico de acciones a realizar en cada turno hacen que resulte especialmente complejo encontrar una técnica adecuada o que destaque sobre las demás.

En este trabajo, hemos desarrollado bots capaces de jugar al RTS simplificado μ-RTS. Este juego, especialmente desarrollado para la investigación, resulta ideal para poner a prueba técnicas de inteligencia artificial en un escenario relativamente sencillo antes de aplicarlas a otros RTS más complejos, como pueden ser *StarCraft* [6] o *Age of Empires* [5].

Para desarrollar nuestros bots hemos optado por utilizar comportamientos codificados a mano en base al conocimiento experto del dominio del RTS. Estos comportamientos básicos definen las acciones que deben ejecutar las unidades de un cierto tipo durante la partida. Eligiendo un comportamiento para cada tipo de unidad de μ-RTS, podemos construir estrategias que permitan a un bot jugar una partida. Esto se pone de manifiesto en nuestro bot *SingleStrategy*, que juega durante toda la partida usando la misma estrategia. Además, este planteamiento se puede combinar con el uso de algún tipo de aprendizaje automático que determine qué combinaciones de comportamientos son más adecuadas para la partida que se va a jugar.

Por otro lado, una de las características principales de los juegos RTS es que en función del momento de la partida los jugadores necesitan centrarse en unos aspectos u otros del juego, como la recolección, la formación de un ejército o el ataque a un enemigo. Esto provoca que los jugadores no jueguen igual durante toda partida, sino que sus decisiones se adapten al momento de la partida en el que se encuentran. Basándonos en esta idea, hemos creado el bot *MultiStrategy*, que dispone de varias estrategias construidas a partir de comportamientos básicos para ir cambiando su forma de jugar a medida que avanza la partida.

Este bot utiliza un algoritmo genético que manipula las estrategias mediante cruces y mutaciones y evalúa el rendimiento de cada una de ellas jugando varias partidas. Gracias a esto, es capaz de adoptar una forma de jugar compleja que combina el uso de distintas estrategias en función del momento de la partida, y de las características particulares del mapa en el que va a jugar. Hemos estudiado el desempeño de *MultiStrategy* contra algunos participantes de los concursos más recientes de μ-RTS celebrados en la CoG durante los últimos años, mostrando que es capaz de ganarles.

Entre las principales ventajas que ofrece *MultiStrategy* está su capacidad de adaptarse al mapa en el que se va a desarrollar la partida. Esto le permite desenvolverse bien en escenarios muy distintos y que requieren comportamientos completamente antagónicos, como los que hemos estudiado en las secciones de resultados 5.6.

Asimismo, puede cambiar de estrategia en momentos clave de la partida, adaptando su juego en función de si se encuentra en una etapa temprana, intermedia o avanzada, lo que le aporta la versatilidad necesaria para tratar de obtener una ventaja sobre sus rivales.

Sin embargo, *MultiStrategy* también presenta limitaciones. Aunque puede adaptarse al mapa de juego, no puede adaptarse a su rival ni puede identificar o predecir su comportamiento para tratar de contrarrestar sus acciones. Además, para aprender estrategias adaptadas al mapa de juego requiere un entrenamiento previo de al menos unos minutos que hacemos por medio de un algoritmo genético.

Este entrenamiento esconde también algunos problemas. El más relevante es cómo evaluar las estrategias para determinar las más adecuadas para el tablero. La opción empleada por *MultiStrategy* consiste en jugar varias partidas usando esas estrategias y puntuarlas en función del resultado obtenido. El problema llega a la hora de elegir un rival para jugar estas partidas. En nuestro trabajo, hemos probado tanto a utilizar un conjunto fijo de bots de referencia como a enfrentar a cada individuo con el resto de su generación.

Por un lado, si se usan bots de referencia fijos como rivales en las partidas, es necesario que los bots elegidos sean lo suficientemente buenos y variados como para que el resultado del entrenamiento conduzca a buenas estrategias. Sin embargo, los bots más sofisticados que emplean técnicas avanzadas suelen requerir un tiempo de cómputo que alarga el entrenamiento excesivamente.

Por otro lado, al usar las propias estrategias a evaluar como rivales haciendo que se enfrenten entre ellas en una especie de torneo puede ocurrir que el entrenamiento se estanque en un máximo local en el se prueben continuamente estrategias parecidas sin que ninguna destaque. Para tratar de reducir esta posibilidad, nuestro algoritmo realiza más mutaciones de lo habitual y además trata de mantener un mínimo de variedad entre las estrategias que forman cada generación. Aun así, rea-

lizar el entrenamiento de este modo puede requerir más generaciones para aprender estrategias que se encontrarían más rápido con enfrentando al genético directamente contra otros bots más especializados en el tablero.

6.1. Revisión de los objetivos

Pasamos ahora a explicar el desglose del trabajo que hemos realizado dentro de los objetivos que habíamos establecido en la introducción:

- En primer lugar, en cuanto al estudio de la investigación actual en el campo de la inteligencia artificial aplicada a los juegos RTS, hemos desarrollado un resumen de las técnicas utilizadas, que se puede encontrar en el capítulo 2. También entraría dentro de este objetivo, aunque en combinación con el siguiente, el estudio del juego μ-RTS realizado en el capítulo 3, ya que es un juego RTS desarrollado especialmente para la investigación de la inteligencia artificial aplicada a este campo.
- Dentro del segundo objetivo, el desarrollo de un bot sencillo, hemos diseñado los comportamientos basándonos en nuestro conocimiento del dominio, para luego combinarlos en la creación del bot *SingleStrategy*. Este primer bot nos sirvió para familiarizarnos con el entorno ofrecido por μ-RTS, y supuso una base sobre la que apoyarnos a la hora de desarrollar el bot más complejo que proponíamos en el tercer objetivo. Todo el proceso de desarrollo y el análisis de los resultados conseguidos con *SingleStrategy* se puede encontrar en el capítulo 4 de este trabajo.
- Finalmente, en cuanto al tercer objetivo, hemos desarrollado el bot *MultiStrategy*, en el que combinamos los comportamientos desarrollados para *SingleStrategy* con la división en fases de la partida, haciendo que el bot cambie de estrategia a medida que ésta avanza. Además, para poder encontrar la combinación de estrategias más adecuada para cada tablero, hemos utilizado un algoritmo genético inspirado por los conocimientos adquiridos en el estudio realizado en la primera parte. El proceso de desarrollo de *MultiStrategy* y el análisis de los resultados obtenidos se puede encontrar detallado en el capítulo 5.

6.2. Posibles vías de desarrollo futuro

Es posible aumentar la complejidad de *MultiStrategy* de diversas formas para tratar de mejorar su capacidad de adaptación y desarrollo de estrategias prometedoras. En los experimentos que hemos llevado a cabo, el algoritmo genético encuentra bue-

nas configuraciones en pocas generaciones, lo que muestra que aún hay margen para aumentar la complejidad sin provocar que el algoritmo deje de funcionar. Algunas de las posibilidades que se pueden abordar para mejorar *MultiStrategy* son:

- Implementar nuevos comportamientos básicos para las unidades. Por ejemplo se podrían implementar nuevos criterios de ataque para las tropas, como atacar primero a los soldados a rango o a otro tipo concreto de unidad, o establecer varios niveles de prioridad. Otra posibilidad para mejorar los comportamientos es añadir nuevos parámetros configurables. En nuestras implementaciones, muchos parámetros están prefijados, como la población de trabajadores generados, la cantidad de bases o cuarteles que se construyen o la posición de espera de las tropas. Se podrían añadir nuevos genes al cromosoma que permitan al genético elegir estos parámetros como si se tratara de un comportamiento más.

- Valorar varias estrategias a la vez. *MultiStrategy* puede cambiar de estrategia al cambiar de fase, pero durante los ciclos que dure esa fase no considerará ninguna otra estrategia que la que venga fijada en su cromosoma. Es posible hacer que el algoritmo genético devuelva varias estrategias para permitir que el bot tenga más de una estrategia posible para cada fase, y que se elija cuál de ellas aplicar en cada momento realizando algún tipo de simulación como la que se utiliza, por ejemplo, en los árboles de Monte Carlo. También se podrían combinar las distintas estrategias con otras divisiones además de la temporal que utilizamos hasta ahora, por ejemplo según el sector del tablero, o dividir las unidades que tiene el jugador en grupos y asignar una estrategia distinta a cada uno.

- Tener en cuenta la distribución espacial de las unidades en el tablero. Continuando con la idea del punto anterior donde se proponía dividir las unidades en grupos, se podría procesar la posición relativa de las unidades para crear formaciones, por ejemplo, colocando los soldados a distancia detrás de los pesados para que éstos los protejan. También se pueden utilizar edificios para bloquear el camino a las unidades contrarias.

- Combinar *MultiStrategy* con otras técnicas de inteligencia artificial. Por ejemplo, en lugar de utilizar comportamientos precodificados para diseñar las estrategias, se podría acoplar otro sistema de inteligencia artificial que se encargase de crear estos comportamientos, como aprendizaje por refuerzo, para que luego el algoritmo genético las combinase de la mejor forma posible. También se podrían acoplar otras técnicas de inteligencia artificial para añadir funcionalidades completamente nuevas, como modelado dinámico del oponente para adaptarnos a él durante la partida.

- Aprender a partir de trazas de partidas. Se podría buscar la forma de procesar las trazas de partidas de jugadores expertos y combinarlas con el aprendizaje automático a la hora de buscar las estrategias más adecuadas.

- Aprender a jugar con conocimiento incompleto del estado de juego. Al principio del trabajo descartamos tener en cuenta la *niebla de guerra* a la hora de desarrollar nuestros bots, ya que se añade una dimensión más de dificultad a la que ya ofrecen los juegos RTS. Ahora que tenemos una base sobre la que trabajar, se podría buscar la forma de hacer que *MultiStrategy* pueda jugar también con este nivel de dificultad añadido.

En definitiva, los juegos RTS son un dominio apasionante con multitud de problemas abiertos, con muchas posibles vías de investigación sobre cómo aplicar técnicas de inteligencia artificial para resolverlos. Tanto es así que incluso grandes empresas como Google han mostrado interés en desarrollar inteligencias artificiales capaces de jugar a este tipo de juegos, en particular para *StarCraft II* [6]. Este es el caso de *AlphaStar* [37], que utiliza una red neuronal profunda entrenada directamente con los datos de las partidas que juega, y que ha conseguido vencer a jugadores profesionales de este juego.

Conclusion

As explained throughout the work, the application of Artificial Intelligence to real-time strategy games is especially interesting due to the great complexity that these offer. The large number of units that each player can control at the same time and the wide range of actions to be carried out each turn make it especially difficult to find a suitable technique or one that stands out from the rest.

In this work, we have developed bots capable of playing μ-RTS, a simplified RTS game. Specifically developed for research, this game is ideal for testing artificial intelligence techniques in a relatively simple scenario before applying them to other more complex RTS, such as *StarCraft* [6] or *Age of Empires* [5].

To develop our bots we have chosen to use hard-coded behaviors based on expert knowledge of the RTS domain. These basic behaviors define the actions that units of a certain type must take during the game. By choosing a behavior for each unit type in μ-RTS, we can build strategies that allow the bot to play a game. This is what we do in our bot *SingleStrategy*, which plays throughout the game using the same strategy. In addition, this approach can be combined with the use of some type of machine learning that determines which combinations of behaviors are most suitable for the game to be played.

On the other hand, one of the main characteristics of RTS games is that, depending on the time of the game, players need to focus on some aspects or other of the game, such as gathering resources, forming an army or attacking an enemy. This means that players do not play the same during the whole game, but rather that their decisions are adapted to the moment of the game they are in. Based on this idea, we have created the bot *MultiStrategy*, which uses various strategies built on basic behaviors to change the way it plays as the game progresses.

MultiStrategy uses a genetic algorithm that manipulates the strategies by means of crossovers and mutations and evaluates the performance of each of them by playing several games. This allows the bot to adopt a complex way to play that uses different strategies depending on the stage of the game and the characteristics of the game map. We studied the performance of *MultiStrategy* against some of the particiants in the μ-RTS competitions held at the CoG in recent years, showing that it can win them.

One of the advantages of *MultiStrategy* is its ability to adapt its behavior to the map where the game will take place. This allows it to get good results in very different scenarios that may require antagonistic behaviors.

Furthermore, it can change its strategy at some points of the game, adapting its gameplay depending on whether it is in an early, intermediate or advanced stage of the game. This gives *MultiStrategy* the versatility to gain an advantage over its rivals.

However, *MultiStrategy* also has limitations. It may be able to adapt its gameplay to the game map, but it cannot learn from its rival, nor try to predict his next movements or actions. Moreover, the genetic approach needs some previous training in order to learn stategies suitable for the game map, which can take from several minutes to hours.

This training also hides some problems. The most relevant one is how to evaluate the strategies to determine the most suitable one for the board. The option used by *MultiStrategy* is to play multiple games using these strategies and rate them based on the result obtained. The problem comes when choosing an opponent to play these games. In our work, we have tried both using a fixed set of baseline bots and confronting each individual with the rest of their generation.

On the one hand, if fixed baseline bots are used as rivals in games, the chosen bots need to be good and varied enough, so the training result leads to good strategies. However, the most sophisticated bots that use advanced techniques often requier a computational time that lengthens the training excessively.

On the other hand, when using the strategies to be evaluated as rivals, making them face each other in kind of a tournament, it may happen that the training get stuck at a local maximum in which similar strategies are continuously tested without any standout. To try to reduce this possibility, our algorithm performs more mutations than usual and also tries to maintain a minimum of variety among the strategies that form each generation. Still, training in this way may require more generations to learn strategies that would be found faster by training the genetic against other bots more speciallized it that specific board.

Review of objectives

We now turn to explain the breakdown of the work that we have done within the objectives that we had established in the introduction:

- Firstly, regarding the study of current research in the field of artificial intelligence applied to RTS games, we have developed a summary of the techniques used, which can be found in chapter 2. It would also fall within this objecti-

ve, although in combination with the following, the study of the game μ-RTS carried out in chapter 3, since it is an RTS game developed especially for the investigation of artificial intelligence applied to this field.

- Within the second objective, the development of a simple bot, we have designed the behaviors based on our knowledge of the domain, and then combined them in the creation of the bot *SingleStrategy*. This first bot was useful to familiarize us with the environment offered by μ-RTS, and provided a basis on which to build on the development of the more complex bot that we proposed in the third objective. The entire development process and the analysis of the results achieved with *SingleStrategy* can be found in chapter 4 of this work.
- Finally, regarding the third objective, we have develped the bot *MultiStrategy*, in which we combine the behaviors developed for *SingleStrategy* with the division into phases of the game, causing the bot to change strategy as it advances. Furthermore, in order to find the most suitable combination of strategies for each board, we have used a genetic algorithm inspired by the knowledge acquired in the study carried out in the first part. The development process of *MultiStrategy* and the analysis of the results obtained can be found in detail in chapter 5.

Possibilities for future work

It is possible to increase the complexity of *MultiStrategy* in various ways to try to improve its skill to adapt and develop more promising strategies. In the experiments we have carried out, the genetic algorithm finds good configurations in a few generations, which shows that there is still room to increase complexity without causing the algorithm to stop working. Some of the possibilities that can be addressed to improve *MultiStrategy* are:

- The implementation of new basic behaviors for units. For example, new attack criteria could be implemented for troops, such as prioritizing ranged soldiers or another specific type of unit or establishing various priority levels. Another possibility is to add new configurable parameters. In our implementations, many parameters are preset, such as the population of workers generated, the number of bases or barracks that are built, or the waiting position of the troops. We could allow the genetic algorithm to choose these parameters as if it were just another behavior.
- Using several strategies at the same time. *MultiStrategy* can change its strategy when changing phases, but during the cycles that this phase lasts, it will not consider any other strategy than the one fixed in its chromosome. It is possible to make the genetic algorithm return various strategies to allow the bot to have more than one possible strategy for each phase, and to choose which of them to apply at any time by performing some kind of simulation such as the one

used, for example, in Monte Carlo tree search. The different strategies could also be assigned according to the sector of the board, or we could divide the units of the player in groups and assign a different strategy to each one.

- Taking into account the spatial distribution of the units on the board. Continuing with the idea of the previous point, we could divide the units into groups and rearrange their units to create formations, for example, placing the soldiers at a distance behind the heavy ones to protect them. Also, we could use buildings to block the way for opposing units.

- Combining *MultiStrategy* with other artificial intelligence techniques. For example, instead of using precoded behaviours to design the strategies, another artificial intelligence system could be coupled so it was in charge of creating these behaviors, such as reinforcement learning, so later the genetic algorithm could combine them in the best possible way. Other artificial intelligence techniques could also be coupled to add completely new functionalities, such as dynamic modeling of the opponent to adapt the gameplay to its actions.

- Learning from game traces. We could find a way to process the game traces of expert players and combine them with machine learning to find the most suitable strategies.

- Learning to play with incomplete knowledge of the state of play. When creating our bots, we did not take into account the *fog of war*, as it adds even more difficulty to RTS games. Now that we have a base to work on, we could develop a way to make *MultiStrategy* play with this added level of difficulty as well.

In short, RTS games are an exciting domain with multiple open problems and many possible ways of research on how to apply artificial intelligence techniques to solve them. So much so that even large companies like Google have shown interest in developing arficial intelligences capable of playing these kind of games, in particular for *StarCraft II* [6]. This is the case of *AlphaStar* [37], which uses a deep neural network trained directly with the data of the games it plays, and which has managed to beat professional players of this game.

Contribución: Rafael Herrera Troca

La primera parte del trabajo comprende la investigación previa sobre el juego μ-RTS y las técnicas de inteligencia artificial que se han aplicado hasta el momento a los juegos de estrategia en tiempo real.

Comenzamos esta investigación estudiando el artículo [38], que habla sobre la competición de μ-RTS organizada por el IEEE en su *Conference on Games* (CoG). Ambos nos leímos tanto este artículo, ya que la idea del trabajo era desarrollar un bot que se pudiera presentar a dicho concurso, como otro artículo del mismo autor [1] que hace una recopilación de las técnicas de inteligencia artificial que se han aplicado a los juegos RTS.

Una vez hecho esto pasamos al estudio de los bots que se habían presentado al concurso en años anteriores y comenzamos por el ganador del concurso de 2017, *StrategyTactics* [39]. En este bot se combinan dos técnicas de inteligencia artificial: *PuppetSearch* y árboles de Monte Carlo, por lo que decidimos investigarlas por separado.

Yo me encargué de investigar el funcionamiento de PuppetSearch, un algoritmo de búsqueda especialmente diseñado para jugar a juegos RTS. Los dos artículos en los que se detalla el funcionamiento del algoritmo y los resultados obtenidos al aplicarlo son [40] y [41].

Posteriormente, en vista de que el ganador del concurso 2018, *Tiamat* [3] había superado ampliamente a *StrategyTactics*, decidimos basar nuestro bot en las técnicas utilizadas por éste. Dichas técnicas consisten en estrategias precodificadas y un algoritmo genético, como se explica en el artículo [4] que nos leímos ambos.

En esta primera parte del proyecto también estudiamos el código de μ-RTS [32], ya que era el entorno sobre el que debía funcionar nuestro bot, y nos repartimos este trabajo. Por mi parte, me dediqué a desentrañar el funcionamiento interno del juego, que consta principalmente de un bucle que se repite mientras dura la partida, y de cómo interacciona éste con los bots. Posteriormente pusimos en común lo aprendido y creamos nuestro primer bot.

La segunda parte del proyecto consiste en el desarrollo de un bot capaz de jugar a μ-RTS. Para ello nuestro bot utiliza estrategias precodificadas y un algoritmo genético que lo ayuda a elegir la mejor combinación de dichas estrategias.

En la implementación del bot, nos repartimos las clases a realizar. Por mi parte, me encargué de programar el algoritmo genético que debía seleccionar los comportamientos más prometedores para el tablero sobre el que se iba a jugar. Tras una

serie de pruebas iniciales, hubo que corregir el algoritmo genético haciéndolo algo distinto del habitual porque de otro modo no funcionaba correctamente por las características del problema al que nos enfrentábamos.

Por último redactamos la memoria, que recoge tanto la investigación inicial sobre las técnicas utilizadas y el μ-RTS como el posterior desarrollo de nuestro bot y un análisis exhaustivo de los resultados obtenidos.

Sobre la primera parte de la memoria, me encargué de la redacción del capítulo 3, que trata sobre μ-RTS. En éste, se explica el juego, pasando por los distintos tipos de unidades y sus características, los elementos que componen el tablero y el desarrollo de la partida. También se explican otros elementos de más bajo nivel, como el procedimiento a seguir para desarrollar un bot y acoplarlo a μ-RTS o el funcionamiento de los bots ya incluidos en el juego.

En cuanto a la segunda parte, desarrollé el capítulo 5 en el que se explica el funcionamiento del bot *MultiStrategy*. En este capítulo se desarrolla en detalle el algoritmo genético que hemos utilizado, que tiene algunas peculiaridades que lo distinguen del habitual, la configuración de sus parámetros y el análisis de los resultados obtenidos por este bot. Además, al final del capítulo también se realiza una comparación de los resultados de *MultiStrategy* con los que se podrían obtener realizando una búsqueda aleatoria para encontrar la mejor estrategia en lugar de utilizar un algoritmo genético.

También dentro de esta segunda parte, me encargué de desarrollar el código en *Python* que utilizamos para procesar los datos obtenidos al probar los bots y generar las gráficas que se muestran en las secciones de resultados, si bien mi compañero añadió algunos cambios posteriores enfocados a eliminar ruido de los diagramas y obtener imágenes vectoriales de mejor calidad.

Finalmente, me encargué también de redactar la introducción y su traducción al inglés, en la que se da una visión general del contenido del trabajo, los antecedentes y objetivos y los resultados finales obtenidos.

Una vez terminado el primer borrador, lo leímos entero para retocarlo y corregir errores que se hubieran pasado por alto, tanto en las secciones propias como en las realizadas por el otro. A la hora de incorporar los comentarios y correcciones que nos aportaron nuestros tutores nos repartimos el trabajo nuevamente siguiendo la distribución que habíamos hecho a la hora de redactar la memoria.

Contribución: Rubén Ruperto Díaz

El proyecto arrancó con una primera investigación del dominio de μ-RTS y de las técnicas de inteligencia artificial que se han aplicado a los juegos RTS. En esta primera parte, leímos varios artículos relacionados con el tema. Ambos nos leímos los artículos que hablan de la competición de μ-RTS de la CoG de 2017 [38] y la recopilación de técnicas aplicadas a RTS realizada por Ontañón y otros autores en [1]. El ganador de la competición, *StrategyTactics* [39], combinaba el uso de dos técnicas: árboles de Monte Carlo y *PuppetSearch*. Estas técnicas estaban explicadas en varios artículos, cuya lectura nos repartimos.

Yo me encargué de los artículos relacionados con árboles de Monte Carlo, o MCTS por las siglas de Monte Carlo tree search. Entre estos artículos están el que describe el problema *combinatorial multi-armed bandit* (CMAB) y su aplicación en juegos RTS [35] mediante el algoritmo *NaiveMCTS*. Otro artículo que trata este tipo de árboles y su aplicación a los RTS es [42].

Finalmente descartamos basar nuestro bot en estas técnicas y nos decantamos por los algoritmos genéticos que utilizaba *Tiamat* [3] el ganador de la competición de 2018. Ambos nos leímos el artículo que lo describe [4].

Durante esta primera parte del proyecto, también analizamos el código del juego μ-RTS [32]. Cada uno se centró en una parte del código, yo me centré en ver cómo se utilizaban las acciones básicas en el juego para crear bots, así como las clases de Java que era necesario extender para crear un primer bot muy simple que tan sólo producía trabajadores que corrían siempre hacia la izquierda. Posteriormente pusimos en común lo aprendido para empezar a programar un bot más elaborado.

Nuestro bot utiliza comportamientos precodificados que se combinan usando un algoritmo genético. Yo me encargué de diseñar e implementar algunos comportamientos básicos, para lo que me basé en mi experiencia previa jugando a juegos RTS, principalmente a *Age Of Empires I*, tratando de simplificarlas y aplicarlas a las unidades de μ-RTS. Posteriormente probamos y corregimos estos comportamientos y comprobamos que el genético fuera capaz de aprender cuáles eran más adecuados para jugar una partida.

En cuanto a la función de *fitness*, necesitábamos una capaz de evaluar a los individuos durante el genético de forma más precisa que simplemente contando el número de victorias que lograban. Para solucionarlo, utilicé las propiedades de la función *sigmoide* y diseñé una fórmula que asigna puntuaciones a los bots en función del tiempo transcurrido hasta obtener la victoria o sufrir una derrota. Esta función está descrita en la sección 5.4.

A la hora de redactar la memoria, nos hemos repartido los capítulos equitativamente para escribir una primera versión. En este momento, redacté el capítulo 2, en el que se explican algunas nociones básicas de los juegos RTS y las técnicas que se han aplicado a estos juegos recopiladas en [1], realizando una división de las decisiones que se han de tomar durante el juego en tres ámbitos: estrategia, tácticas y control reactivo.

También me encargué del capítulo 4, donde se describen los comportamientos precodificados y su combinación para formar estrategias que nuestros bots pueden utilizar para tomar decisiones durante una partida. En este capítulo se explican también los resultados obtenidos por los bots al enfrentarlos a algunos bots incluidos de forma predeterminada en el juego.

Como evaluamos varios miles de bots, para poder representar gráficamente estos resultados propuse usar diagramas de cajas y bigotes que nos permitieron analizar estadísticamente el rendimiento de cada comportamiento. Aunque no escribí todo el código, si realice algunos cambios para eliminar ruido de los diagramas y para que las imágenes obtenidas fueran vectoriales, mejorando su visualización en la memoria.

Finalmente, escribí el breve resumen inicial que condensa las ideas principales del trabajo en un párrafo y me ocupé de redactar el capítulo de conclusiones, en el que se recogen las ventajas y limitaciones de nuestra propuesta, así como posibles ideas para aumentar su complejidad, buscando aumentar su potencial. También traduje a inglés estas secciones, ya que la normativa de la facultad así lo pide.

Una vez completado el primer borrador, ambos miembros del equipo lo leímos por completo y rectificamos los posibles errores en los capítulos que redactó el otro. También incorporamos los muchos comentarios y correcciones que nos hicieron nuestros tutores, repartiendo de nuevo el trabajo entre ambos integrantes siguiendo la misma división que empleamos durante la primera redacción.

Bibliografía

[1] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill y M. Preuss, *RTS AI Problems and Techniques.* 2019.

[2] *microRTS AI Competition.* dirección: `https://sites.google.com/site/micrortsaicompetition/home` (visitado 14-06-2020).

[3] *TiamatBot*, nov. de 2019. dirección: `https://github.com/jr9Hernandez/TiamatBot` (visitado 04-06-2020).

[4] J. R. Marino, R. O. Moraes, C. Toledo y L. H. Lelis, «Evolving action abstractions for real-time planning in extensive-form games», en *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, págs. 2330-2337.

[5] *Age of Empires Franchise - Official Web Site*, en-US. dirección: `https://www.ageofempires.com/` (visitado 09-06-2020).

[6] *Juego - StarCraft II - Página oficial*, es. dirección: `https://starcraft2.com/es-es/game` (visitado 09-06-2020).

[7] D. Fu y R. Houlette, «The ultimate guide to FSMs in games», *AI game programming Wisdom*, vol. 2, págs. 283-302, 2004.

[8] J.-L. Hsieh y C.-T. Sun, «Building a player strategy model by analyzing replays of real-time strategy games», en *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, IEEE, 2008, págs. 3106-3111.

[9] F. Schadd, S. Bakkes y P. Spronck, «Opponent Modeling in Real-Time Strategy Games.», en *GAMEON*, 2007, págs. 61-70.

[10] M. Certický y M. Certický, «Case-based reasoning for army compositions in real-time strategy games», en *Proceedings of Scientific Conference of Young Researchers*, 2013, págs. 70-73.

[11] B. G. Weber y M. Mateas, «A data mining approach to strategy prediction», en *2009 IEEE Symposium on Computational Intelligence and Games*, IEEE, 2009, págs. 140-147.

[12] G. Synnaeve y P. Bessiere, «A Bayesian model for opening prediction in RTS games with application to StarCraft», en *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, IEEE, 2011, págs. 281-288.

[13] J. Young y N. Hawes, «Evolutionary learning of goal priorities in a real-time strategy game», en *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[14] D. C. Pottinger, «Terrain analysis in realtime strategy games», en *Game Developers Conference 2000*, 2000.

[15] K. D. Forbus, J. V. Mahoney y K. Dill, «How qualitative spatial reasoning can improve strategy game AIs», *IEEE Intelligent Systems*, vol. 17, n.º 4, págs. 25-30, 2002.

[16] D. H. Hale, G. M. Youngblood y P. N. Dixit, «Automatically-generated Convex Region Decomposition for Real-time Spatial Agent Navigation in Virtual Worlds.», *AIIDE*, vol. 8, págs. 173-8, 2008.

[17] L. Perkins, «Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition», en *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2010.

[18] F. Richoux, A. Uriarte y S. Ontanón, «Walling in strategy games via constraint optimization», en *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[19] S. Hladky y V. Bulitko, «An evaluation of models for predicting opponent positions in first-person shooter video games», en *2008 IEEE Symposium On Computational Intelligence and Games*, IEEE, 2008, págs. 39-46.

[20] M. Sharma, M. P. Holmes, J. C. Santamaría, A. Irani, C. L. Isbell Jr y A. Ram, «Transfer Learning in Real-Time Strategy Games Using Hybrid CBR/RL.», en *IJCAI*, vol. 7, 2007, págs. 1041-1046.

[21] G. Synnaeve y P. Bessiere, «Special tactics: A bayesian approach to tactical decision-making», en *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2012, págs. 409-416.

[22] C. Miles y S. J. Louis, «Co-evolving real-time strategy game playing influence map trees with genetic algorithms», en *Proceedings of the International Congress on Evolutionary Computation, Portland, Oregon*, IEEE Press, 2006, págs. 88-95.

[23] D. Churchill, A. Saffidine y M. Buro, «Fast heuristic search for RTS game combat scenarios», en *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[24] M. Chung, M. Buro y J. Schaeffer, «Monte Carlo Planning in RTS Games.», en *CIG*, Citeseer, 2005.

[25] A. Uriarte y S. Ontanón, «Kiting in RTS games using influence maps», en *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[26] H. Danielsiek, R. Stuer, A. Thom, N. Beume, B. Naujoks y M. Preuss, «Intelligent moving of groups in real-time strategy games», en *2008 IEEE Symposium On Computational Intelligence and Games*, IEEE, 2008, págs. 71-78.

[27] G. Synnaeve y P. Bessiere, «A Bayesian model for RTS units control applied to StarCraft», en *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, IEEE, 2011, págs. 190-196.

[28] B. G. Weber, M. Mateas y A. Jhala, «A particle model for state estimation in real-time strategy games», en *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[29] C. A. Madeira, V. Corruble y G. Ramalho, «Designing a Reinforcement Learning-based Adaptive AI for Large-Scale Strategy Games.», en *AIIDE*, 2006, págs. 121-123.

[30] M. Ponsen y P. Spronck, «Improving adaptive game AI with evolutionary learning», Tesis doct., Citeseer, 2004.

[31] C. W. Reynolds, «Steering behaviors for autonomous characters», en *Game developers conference*, Citeseer, vol. 1999, 1999, págs. 763-782.

[32] S. Ontañón Villar, *microRTS*, jun. de 2020. dirección: `https://github.com/santiontanon/microrts` (visitado 06-06-2020).

[33] *Rules - microRTS AI Competition*. dirección: `https://sites.google.com/site/micrortsaicompetition/rules` (visitado 06-06-2020).

[34] *Droplet*, oct. de 2019. dirección: `https://github.com/zuozhiyang/Droplet` (visitado 05-06-2020).

[35] S. Ontanón, «The combinatorial multi-armed bandit problem and its application to real-time strategy games», en *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[36] L. H. Lelis, «Stratified Strategy Selection for Unit Control in Real-Time Strategy Games.», en *IJCAI*, 2017, págs. 3735-3741.

[37] *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*, ALL. dirección: `https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii` (visitado 24-06-2020).

[38] S. Ontañón, N. A. Barriga, C. R. Silva, R. O. Moraes y L. H. Lelis, «The first microrts artificial intelligence competition», *AI Magazine*, vol. 39, n.º 1, págs. 75-83, 2018.

[39] N. A. Barriga, *nbarriga/microRTSbot*, nov. de 2019. dirección: `https://github.com/nbarriga/microRTSbot` (visitado 15-06-2020).

[40] N. A. Barriga, M. Stanescu y M. Buro, «Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games», en *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[41] N. A. Barriga, M. Stanescu y M. Buro, «Game tree search based on nondeterministic action scripts in real-time strategy games», *IEEE Transactions on Games*, vol. 10, n.º 1, págs. 69-77, 2017.

[42] S. Ontanón, «Informed monte carlo tree search for real-time strategy games», en *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2016, págs. 1-8.