# JUGADOR AUTOMÁTICO DE HEARTHSTONE USANDO ÁRBOLES DE MONTE CARLO Y ALGORÍTMOS GENÉTICOS

# A HEARTHSTONE AGENT USING MONTE CARLO TREES AND GENETIC ALGORITHMS

Mateo García Pérez y Jorge Sainero Valle

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

---


Trabajo Fin de Grado en Ingeniería informática

26-06-2020

Directores:

Pedro Pablo Gómez Martín
Antonio Alejandro Sánchez Ruiz-Granados

# Resumen en castellano

La inteligencia artificial y los juegos han estado fuertemente ligados desde los orígenes de esta. El principal objetivo en sus inicios era crear un agente que obtuviese un buen rendimiento en un determinado juego.

El objetivo de este trabajo es crear un agente capaz de jugar al juego de cartas coleccionables Hearthstone intentando maximizar el número de victorias que este obtiene contra otros agentes. Para ello se utilizarán técnicas para explorar el espacio de estados como los árboles de búsqueda de Monte Carlo. También se utilizarán algoritmos genéticos para buscar una buena función capaz de evaluar como de bueno es un determinado estado de una partida.

## Palabras clave

Inteligencia artificial, árboles de búsqueda de Monte Carlo, algoritmos genéticos, Hearthstone.

# Abstract

Artificial intelligence and games have been strongly linked since the origins of this. The main objetive in the beginning was build an agent that would perform well in a given game.

The objetive of this project is to build an agent capable of playing the collectible card game called Hearthstone triying to maximize the number of victories that it obtains againts other agents. This will be done using techniques to explore the space of states such as Monte Carlo tree search. Genetic algorithms will also be used to search for a good function capable of evaluating how good of a given state of a game is.

## Keywords

Artifical intelligence, Monte Carlo tree search, genetic algorithms, Hearthstone.

# Índice general

# Capítulo 1

# Introducción

## 1.1. Motivación

El desarrollo de programas capaces de jugar a juegos de forma automática siempre ha sido un tema de especial interés en el campo de la inteligencia artificial. Aspectos como la necesidad de adaptarse a variables externas (como pueden ser las jugadas de los oponentes) o la naturaleza competitiva, que permite la comparación entre diferentes jugadores o programas, han hecho que juegos como el ajedrez, el *Go*, el poker o videojuegos como *StarCraft* o *Dota 2* hayan sido objeto de estudio durante años. Cada uno de ellos presenta diferentes retos que permiten la especialización en ciertos aspectos.

Entre estos juegos destacamos *Hearthstone: Heroes of Warcraft* un videojuego de cartas coleccionables desarrollado por *Blizzard Entertainment*. En este juego, dos jugadores, cada uno con su mazo de cartas, se van alternando el turno con el objetivo de reducir los puntos del rival a cero mediante el uso de esas cartas y de sus efectos. Como veremos en la sección 3.2, *Hearthstone* presenta una serie de características que hacen de él un juego único. La gran variedad de cartas distintas lo separa de otros juegos clásicos como el Póker, haciendo que sea muy difícil predecir lo que va a hacer el contrario; y la aleatoriedad de algunos efectos lo separa de otros juegos de cartas coleccionables similares como *Magic: The Gathering* [6] o *Gwent* [5]. Es por esto que Alexander Dockhorn y Sanaz Mostaghim organizan de forma anual una competición llamada *The Hearthstone-AI Competition* en la que varios jugadores automáticos de *Hearthstone* se enfrentan entre sí.

La implementación de un jugador automático o agente capaz de jugar a *Hearthstone* será el tema de este trabajo. Para ello nos centraremos en técnicas de inteligencia artificial basadas en búsquedas que, dadas un estado de juego, sean capaces de establecer, evaluando estados posteriores, cual sería la mejor acción posible.

## 1.2. Objetivos

A la hora de desarrollar una inteligencia artificial basada en búsqueda capaz de jugar a un juego hay dos aspectos a tener en cuenta: el ser capaz de tomar la decisión que, dado

un estado del juego, lleve a un mejor resultado y obtener información de calidad sobre estos estados para poder compararlos entre sí .

El agente automático debe, por lo tanto, poder establecer cuál es el plan de acción que le llevará a un mejor resultado. Una aproximación muy extendida a este problema es el usar un método voraz que, siempre que haya que escoger entre varias acciones, seleccione aquella que lleve a un estado considerado más favorable. Como veremos en la sección 3.2 *Hearthstone* es un juego complejo y a veces llevar a cabo una acción inmediata considerada peor puede dar lugar a un mejor resultado a largo plazo. Por ello, y como veremos en la sección 2.1, se van a emplear árboles de búsqueda de Monte Carlo con el objetivo de mantener un balance entre la exploración de estados futuros y el tiempo empleado en ello.

Para obtener información sobre un estado entran en juego las funciones de evaluación. Estas funciones otorgan información decisiva a la hora de establecer cuál de los dos jugadores lleva ventaja o a quién resulta más favorable cada estado. No obstante y como veremos en las secciones 3.2 y 4.2 existen una gran variedad de estrategias a la hora de jugar a *Hearthstone* y, cada una de ellas, le da una importancia distinta a las diferentes variables de cada estado. Sin embargo, el hacer una buena función de evaluación para cada estrategia requiere disponer de cierto conocimiento sobre el juego. Con el fin de obtener una buena función de evaluación independiente de ese conocimiento de forma automática, se usarán algoritmos genéticos capaces de optimizar el número de partidas ganadas.

## 1.3. Plan de trabajo

En primer lugar se hará un estudio sobre los árboles de búsqueda de Monte Carlo y de su aplicación en el campo de la inteligencia artificial aplicada a juegos. Una vez hecho eso se procederá a implementar un agente capaz de consultar las jugadas posibles dentro de un turno y de tomar la que considere más adecuada.

También se estudiarán algunas de las distintas posibilidades para elaborar una función de evaluación del estado del juego con el fin de mejorar la calidad del conocimiento que el agente tendrá sobre este. Tras esto se implementará un algoritmo genético para obtener una nueva función de evaluación de estados.

Para terminar se harán una serie de experimentos comparando las diferentes implementaciones de estos agentes y de las funciones de evaluación para determinar cuál es la que nos proporciona un mejor resultado.

## 1.4. Estructura de la memoria

En el ***Capítulo 2*** se habla sobre la base teórica que se va a utilizar basándose en dos pilares: los árboles de búsqueda de Monte Carlo sobre los diferentes estados de un juego para obtener la mejor jugada y el uso de algoritmos genéticos para optimizar la función de evaluación de estos estados.

El ***Capítulo 3*** trata sobre el funcionamiento del juego *Hearthstone* así como del simulador sobre el que se va a trabajar.

En el ***Capítulo 4*** se desarrollan y explican los pasos seguidos durante la implementación de los agentes y los problemas que han surgido. Distinguimos dos partes: una sobre la implementación de un agente de inteligencia artificial encargado de explorar el espacio de estados posibles y seleccionar la jugada más favorable, y otra que examina las diferentes funciones de evaluación de estos estados y emplea un algoritmo genético para obtener una mejor. También se analizan los resultados que se han obtenido.

El ***Capítulo 5*** resume el trabajo realizado, los resultados obtenidos en los experimentos y las conclusiones que se pueden extraer de ellos. Además se tratan los diferentes caminos que se podrían seguir para mejor los agentes implementados.

# Capítulo 1

# Introduction

## 1.1. Motivation

The development of programs capable of playing games automatically has always been a topic of special interest in the field of artificial intelligence. Aspects such as the need to adapt to external variables (such as the plays of the opponents) or its competitive nature, which allows comparison between different players or programs, have led to games such as chess, *Go*, poker or video games such as *StarCraft* or *Dota 2* having been under study for years.

One of these games is *Hearthstone Heroes of Warcraft* a digital collectible card game developed by *Blizzard Entertainment*. In this game, two player, each with their own deck of cards, alternate turns with the goal of of reducing the opponent's points to zero by using those cards and their effects. As we will show in section 3.2, *Hearthstone* has a number of features that make it a unique game. Having a great variety of different cards separates it from other classic games such as Poker, making very difficult to predict what the other player will do; and the randomness of some effect separate it from other collectible card games like *Magic: The Gathering* [6] and *Gwent* [5]. For these reasons Alexander Dockhorn and Sanaz Mostaghim organize an annual competition called *The Hearthstone-AI Competition* where several automatic players compete against each other.

The implementation of an automatic player capable of playing *Hearthstone* will be the subject of this work. To do this, we will focus on search-based artificial intelligence techniques

## 1.2. Objectives

When developing a search-based artificial intelligence capable of playing a game there are two aspects to consider: being able to make a decision that, given a game state, leads to a better result and obtaining reliabale information so they can be compared between them.

The automatic player should be able to decide which action plan will result on the better outcome. A popular approach to this problem is to use a greedy method in which, whenever there is a choice between several actions, the one that leads to a state considered to be

more favorable is selected. As we will see in section 3.2 *Hearthstone* is a complex game and sometimes choosing a suboptimal action could lead to a better outcome on the long term. For that reason, as we will show in section 2.1, we will use Monte Carlo tree search to keep a balance between exploring future game states and the time needed for it.

To obtain information about a game state, the evaluation functions come into play. These functions provide decisive information when it comes to establishing which player has advantage or who benefits the most from each state. However, as we will see in sections 3.2 and 4.2, there is a wide variety of strategies when it comes to playing *Hearthstone* and each of them gives a different importance to the different variables of each state. Nevertheless, doing a good evaluation function for each strategy requires some game knowledge. In order to obtain a knowledge-independent evaluation function we will use genetic algorithms capable of optimizing the number of games won.

## 1.3. Work plan

Firstly, a study of Monte Carlo tree search and its application in the field of artificial intelligence on games will be made. Once this is done, we will implement an automatic agent capable of checking the possible moves within a turn and taking the one with the best outcome.

We will also explore some of the alternatives for developing a game state evaluation function in order to improve the quality of the agent's game . After that, a genetic algorithm will be implemented to obtain a new gam state evaluation function.

Finally, a series of experiments will be conducted comparing the different implementations of these agents and evaluation functions to determine which one will provide the best results.

## 1.4. Work structure

In ***Chapter 2*** we will cover the theoretical basis to be used: the Monte Carlo search trees on the different game states to obtain the best play and the use of genetic algorithms to optimize the evaluation function of these states.

***Chapter 3*** explains how the game *Hearthstone* works as well as the simulator that will be worked on.

***Chapter 4*** describes the steps followed during the implementation of the agents and the problems that have arisen. We distinguish two parts: one about the implementation of an agent in charge of exploring the space of possible game states and selecting the most advantageous move, and another one that examines the different functions of evaluation of these states and uses a genetic algorithm to obtain a better one. The results obtained are also analysed.

***Chapter 5*** summarizes the work done, the results obtained in the experiments and the conclusions that can be drawn from them. It also discusses the different approaches that could be taken in order to improve the implemented agents.

# Capítulo 2

# Árboles de búsqueda de Monte Carlo y algoritmos genéticos

Como se ha comentado en la sección 1.2 nuestro objetivo es realizar una IA capaz de jugar a un juego por turnos de dos jugadores como es el caso del Hearthstone. Generalmente el espacio de estados de estos juegos se suele representar con estructura de árbol donde la raíz es el estado actual y los hijos de esta son los estados que se alcanzan aplicando las acciones que permita este estado. Este árbol se desarrollaría aplicando este razonamiento a los hijos obtenidos y sucesivamente hasta llegar a estados terminales correspondientes a las hojas del árbol.

Lo más habitual en los juegos por turnos es que después de cada acción se produzca un cambio de turno entre jugadores y que los niveles pares del árbol representen el turno de un jugador y los impares, el del contrario. Con esta dinámica de juego y con total conocimiento del estado de juego (información perfecta) el algoritmo más habitual es el algoritmo de búsqueda minimax [32].

El algoritmo minimax empieza a tener problemas cuando el factor de ramificación del árbol es muy elevado porque la expansión completa del árbol es muy cara. Cuando no es factible expandir el árbol completo por su tamaño, se suelen utilizar métodos probabilísticos que eviten expandir el árbol en su totalidad.

Este es el caso del Hearthstone donde las acciones disponibles en un turno son muchas y además es un juego que no dispone de información completa ni siquiera dentro del mismo turno. Por este motivo, veremos en este capítulo otra estrategia para explorar el espacio de estados como son los árboles de búsqueda de Monte Carlo.

## 2.1. Árboles de búsqueda de Monte Carlo

Los árboles de búsqueda de Monte Carlo (MCTS) son un método de búsqueda que combina la precisión de los árboles de búsqueda con la generalidad del muestreo aleatorio. Es un algoritmo para encontrar las decisiones óptimas en un dominio dado tomando muestras aleatorias y construyendo un árbol de búsqueda con la información obtenida de estas

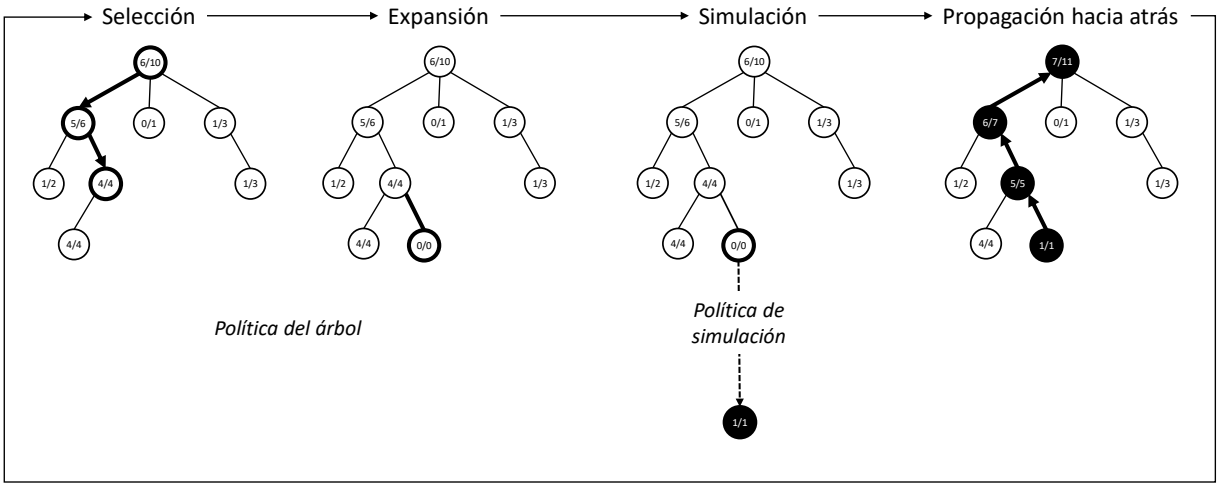


Figura 2.1: Una iteración del algoritmo general

muestras [11]. Llevándolo al terreno de los juegos, se construiría un árbol con los posibles futuros estados del juego lanzando una gran cantidad de simulaciones y aprendiendo de los resultados de estas.

Para explorar el espacio de estados, la idea de los MCTS es tener un árbol parcialmente expandido (de manera irregular, no tiene porque estar expandido completamente un nivel para poder expandir nuevos nodos en el siguiente) y en cada paso se decide el nodo a expandir añadiendo un hijo al padre. Este hijo no es más que una acción en el estado del padre que no ha sido aplicada antes. Una vez realizada esta expansión se lanza una simulación desde el estado del hijo añadido al árbol hasta llegar al final del juego lo que da un éxito o un fracaso dependiendo del resultado. El resultado obtenido repercute positiva o negativamente al hijo añadido y a la acción que llevaba desde el padre al hijo y, de forma indirecta, a todos los ancestros de ese nodo.

Este proceso se repite una y otra vez y surge un dilema. El nodo nuevo que se expande va a manipular la percepción que tenemos de todos sus predecesores, de modo que cuando decimos qué expandimos estamos implícitamente diciendo si buscamos "confirmar que un nodo predecesor es bueno" (simulando otra vez un hijo suyo aleatoriamente hasta el final para ver si nos lleva a un éxito), o si preferimos dar una oportunidad a los nodos que no han demostrado ser tan buenos pero que podrían serlo y haber sido cosa de la mala suerte que las simulaciones hayan resultado en fracaso.

Esto nos lleva al dilema exploración vs. explotación. La exploración consistiría en explorar los nodos de los cuáles tenemos menos información porque no hemos realizado muchas simulaciones desde ellos y las que hemos hecho no nos han devuelto muchos éxitos. Por su parte, la explotación consistiría en seguir explorando las ramas más prometedoras con el fin de confirmar que son buenas las acciones que llevan a esos nodos.

En la figura 2.1 podemos ver una iteración de como se va construyendo el árbol iterativamente.

El primer valor del interior de cada nodo indica el número de éxitos obtenidos en simu-

laciones que pasaban por el estado del nodo y el segundo valor indica el número total de simulaciones que pasaban por él. Por la construcción del árbol, podemos apreciar que el valor de estos parámetros es la suma de los de sus hijos y por ello cada subárbol es considerado un árbol de búsqueda de Monte Carlo.

Aunque el algoritmo básico que se describirá continuación en la sección 2.1.1 sirva para un gran rango de problemas, en la práctica se suele modificar el algoritmo adaptándolo al dominio concreto donde lo vamos a aplicar. Gran parte de la investigación sobre árboles de búsqueda de Monte Carlo consiste en determinar que mejoras y variaciones se pueden aplicar a una situación concreta y como estas mejoras se pueden extender a otros dominios [11].

### 2.1.1. Algoritmo básico de expansión de un MCTS

Como hemos comentado previamente, el algoritmo básico implica una construcción incremental e iterativa de un árbol de búsqueda. Vamos expandiendo un nodo nuevo en cada iteración hasta que se alcanza un determinado presupuesto computacional. Este presupuesto suele ser una limitación de tiempo, iteraciones o memoria. En el árbol, cada nodo representa un estado del dominio y los enlaces a los hijos representan acciones que conducen a estados posteriores.

Cada iteración del algoritmo consiste en cuatro pasos:

1. *Selección*: Comenzando en el nodo raíz se aplica la política del árbol para elegir cuál es el nodo que va a ser elegido. Un nodo es expansible si no es un estado terminal y tiene hijos que no pertenecen al árbol (no han sido visitados todavía). La política se aplica de forma descendente, esto significa que en la raíz se selecciona uno de los hijos, si este hijo ya ha sido totalmente expandido se aplica la política al subárbol con el nodo del hijo como raíz.

2. *Expansión*: Uno o más hijos del nodo seleccionado se añaden al árbol de búsqueda, de acuerdo con las acciones disponibles.

3. *Simulación*: Se ejecuta una simulación desde los nuevos nodos de acuerdo con la política de simulación hasta producir un resultado.

4. *Propagación hacia atrás*: El resultado de la simulación sube por el árbol por todos los antecesores del nodo expandido actualizándolos.

Como hemos visto existen dos políticas a la hora de seleccionar el siguiente nodo:

1. *Política del árbol*: Es la que se emplea en la selección y la expansión. Es la que se aplica con los nodos que pertenecen al árbol.

2. *Política de simulación*: Es la que se emplea en la simulación para escoger la siguiente acción cuando estamos fuera del árbol de búsqueda. La política que se usa por defecto suele ser escoger una acción aleatoria.

Estos pasos están resumidos en el pseudocódigo en 1. La función recibe un estado $s_0$ y $v_0$ es el nodo raíz correspondiente al estado $s_0$, $v_l$ es el último nodo añadido al árbol de búsqueda mediante la política del árbol y se corresponde con el estado $s_l$ y $r$ es el resultado de haber aplicado la política por defecto en la simulación desde el estado $s_l$. Una vez acabado el presupuesto computacional se selecciona el mejor hijo de $v_0$ y se devuelve la acción que se aplica al estado $s_0$ para obtener el estado del mejor hijo. El significado de mejor hijo depende de la implementación.

**Algoritmo 1** Pseudocódigo del algoritmo MCTS

```
1: function MCTS(s_0)
2:     crear nodo raíz v_0 con s_0 como estado
3:     while haya presupuesto computacional do
4:         v_l = PoliticaArbol(v_0)
5:         r = PoliticaDefecto(s_l)
6:         Propagacion(v_l, r)
7:     return a(MejorHijo(v_0))
```

Cuando se termina el presupuesto computacional, se debe seleccionar cual va ser la acción devuelta. Para ello se suele utilizar uno de los siguientes criterios descritos en [12]:

1. *Max child*: Se selecciona el hijo de la raíz con la mayor recompensa.
2. *Robust child*: Se selecciona el hijo de la raíz más visitado.
3. *Max-Robust child*: Se selecciona el hijo de la raíz con mayor recompensa y mayor número de visitas. Si no existe este hijo, se ejecutan más simulaciones hasta que esto suceda [13].
4. *Secure child*: Se selecciona el hijo que maximice el "límite de mínima confianza". Este límite se define como $r + \frac{A}{\sqrt{n}}$ dónde $A$ es un parámetro, $r$ es la recompensa del nodo y $n$ el número de visitas recibidas.

Elegir el siguiente nodo a expandir tiene que ver con el dilema de la exploración y la explotación. Por ello, en las siguientes secciones trataremos sobre dos posibles políticas que se suelen aplicar como política del árbol y que buscan el equilibrio entre explotación y exploración.

## 2.2. Políticas del árbol

Como primera aproximación empezaremos con una política muy sencilla. Esta política resuelve el dilema exploración vs. explotación de una manera muy simple, dedica parte del presupuesto computacional a la exploración de todos los nodos por igual y el resto del presupuesto a la explotación de los nodos más prometedores.

Esta política tiene como parámetro un cierto $k \in [0, 1]$ que indica que parte del presupuesto computacional se dedica a la exploración. Cuando hay que aplicar esta política se pueden dar dos situaciones: el porcentaje del presupuesto computacional restante es menor que $k$ o que sea mayor o igual que $k$.

En el primer caso estaríamos en la parte del presupuesto dedicada a la exploración. Se reparten equitativamente las visitas a cada uno de los hijos, siguiendo un cierto orden para evitar que ninguno de los hijos reciba más visitas que el resto. En el segundo caso estaríamos en la parte del presupuesto dedicada a la explotación. Para seguir obteniendo información de los hijos más prometedores podemos utilizar diferentes estrategias:

- Seleccionar al hijo con mayor cociente de éxitos por simulaciones.
- Seleccionar un hijo aleatorio dentro de los $N$ mejores siendo $N$ un parámetro independiente o dependiente del número total de hijos (por ejemplo, podría decidir elegir un hijo que pertenezca al 20 % de los mejores hijos).

### 2.2.1. UCB: Límite de confianza superior

El dilema exploración vs. explotación es típico de otros dominios como el aprendizaje por refuerzo, de modo que se aplican en los MCTS técnicas importadas de él, por lo que para entenderlas merece la pena detenerse un momento y comprenderlas, aunque sean de otro dominio. En realidad, en ambos contextos surge el mismo problema que se puede resumir como "el problema del bandido multibrazo".

Los problemas de bandido multibrazo son una clase bien conocida de los problemas de decisión secuenciales, en los que es necesario elegir entre $K$ acciones para maximizar la recompensa acumulada tomando consistentemente la acción óptima [11]. El ejemplo más común es aquel en el que un jugador se encuentra frente a una serie de máquinas tragaperras (bandidos de un brazo) tiene que decidir con qué máquinas juega y en qué orden lo hace. Cuando juega, cada máquina devuelve una recompensa aleatoria derivada de la distribución de probabilidad de la máquina. El objetivo del jugador es maximizar la suma de las recompensas dada una secuencia de máquinas. Este es un claro ejemplo de aprendizaje por refuerzo ya que el jugador va obteniendo información de las recompensas devueltas por las máquinas antes de tomar la decisión final.

La elección de la acción es difícil ya que se desconocen las distribuciones de las recompensas y las recompensas potenciales deben estimarse en función de las observaciones anteriores. Esto nos lleva al dilema de exploración vs. explotación: es necesario equilibrar la explotación de la acción que en el momento se cree óptima con la exploración de otras acciones que actualmente parecen peores pero que pueden resultar mejores a largo plazo.

Un bandido con K-brazos se define mediante las variables aleatorias $X_{i,n}$ con $1 \leq i \leq K$ y $n \geq 1$ dónde $i$ indica el brazo del bandido. Las jugadas sucesivas del bandido $i$ tienen como resultado $X_{i,1}$, $X_{i,2}$,... que están idéntica e independientemente distribuidas de acuerdo a una ley desconocida con una cierta "expectativa" $\mu_i$. Los valores de $X_{i,n}$ se entienden dentro del intervalo [0,1]. El problema del bandido K-brazos puede abordarse mediante una

política que determine qué bandido se debe jugar, basándose en las recompensas obtenidas anteriormente.

El objetivo de esta política tiene que ser minimizar la pérdida esperada al no jugar el brazo óptimo del bandido, lo que se entendería como jugar el brazo más seguro. La primera política llamada *"Upper Confidence Indices"* propuesta en [29] permitía estimar la recompensa esperada de cada brazo del bandido después de cada ejecución. Sin embargo estos índices eran difíciles de computar y en [9] se propone una modificación en la que la recompensa de cada bandido puede expresarse mediante una sencilla función. Esta será la base para que luego se propongan varias variantes, una de las más importante es UCB1 de la que hablaremos a continuación.

Como hemos comentado antes, para los problemas de bandido multibrazo, es útil conocer la cota superior de confianza (UCB) de cualquier hijo dentro del árbol de exploración. La política más simple fue propuesta en [10], se llama *UCB1* y la pérdida tiene un crecimiento logarítmico respecto al número de jugadas cuando no se tiene ningún conocimiento previo de las distribuciones de las recompensas de los brazos del bandido. Esta política selecciona al brazo $j$ que maximice la siguiente fórmula:

$$UCB1_j = \bar{X}_j + \sqrt{\frac{2\ln n}{n_j}}$$

donde $\bar{X}_j$ es la media de la recompensa obtenida por el brazo $j$, $n_j$ es el número de veces que se ha seleccionado el brazo $j$ y $n$ es el número de total de jugadas (que es la suma de todos los brazos). Un bandido con una recompensa media $\bar{X}_j$ alta fomenta la explotación de este. Por su parte el término de la derecha $\sqrt{\frac{2\ln n}{n_j}}$ aumenta cuando un brazo es poco explorado en relación con el resto por lo que este término fomenta la exploración de los brazos menos visitados.

En la siguiente sección veremos el algoritmo UCT que es el algoritmo resultante de aplicar la fórmula UCB como política de árbol en un MCTS.

### 2.2.2. UCT: Límite de confianza superior para árboles

El objetivo de los árboles de búsqueda de Monte Carlo es aproximar el valor de las acciones que se pueden tomar desde el estado inicial. Esto se consigue construyendo iterativamente un árbol de búsqueda parcial como mostramos en la figura 2.1. La forma en la que se va construyendo el árbol depende de cómo se van seleccionando los nodos del árbol, es decir, la política del árbol. El éxito de los algoritmos MCTS, especialmente en el juego Go, se debe principalmente a esta política de árbol [24]. Los autores de [27] y [28] proponen utilizar la fórmula *UCB1* como política del árbol. Al tratar la elección del nodo hijo como un problema de bandido multibrazo, el valor del nodo hijo es la recompensa acumulada de las simulaciones de Monte Carlo y el número de simulaciones realizadas desde este.

La fórmula $UCT$ se define para cada hijo $j$ como:

$$UCT_j = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

donde el hijo $j$ se selecciona para maximizar la expresión anterior donde $n$ es el número de veces que se ha visitado el nodo actual (nodo padre), $n_j$ el número de veces que se ha visitado el hijo $j$ y $C_p > 0$ es una constante. Si varios nodos tienen el mismo valor el empate se suele romper aleatoriamente [28].

En general, se entiende que cuando $n_j = 0$ el valor de $UCT$ es $\infty$, de modo que a los hijos que no han sido visitado previamente se les asigna el mayor valor posible para asegurar que todos los hijos de un nodo seleccionado sean considerados una vez antes de que cualquiera de sus hermanos se expanda más. Esto resulta en un poderoso *método de búsqueda local iterada* [11].

Al igual que en la fórmula UCB, existe un equilibrio esencial entre el primer (explotación) y segundo (exploración) término de la fórmula $UCT$. A medida que el nodo va recibiendo visitas, el denominador del segundo término aumenta, lo que hace que disminuya su contribución. Por otro lado, si se visita otro hijo del nodo padre, el numerador aumenta y los valores de los hermanos no visitados aumentan. El término de exploración asegura que cada nodo hijo tiene una probabilidad no nula de ser seleccionado, algo esencial dada la naturaleza aleatoria que tienen los juegos. Esto también otorga una propiedad inherente de reinicio al algoritmo, ya que los hijos con baja recompensa tienen garantizada la elección final (con suficiente tiempo) y, por tanto, la exploración de diferentes opciones y líneas de juego.

La constate del segundo término $C_p$ puede ajustarse para disminuir o aumentar la importancia del factor de exploración. En [28] los autores muestran que el valor de $C_p = \frac{1}{\sqrt{2}}$ es aquel que satisface la desigualdad de Hoeffding cuando el valor de las recompensas está en el intervalo [0,1]. Cuando las recompensas están fuera de este rango se necesitan valores diferentes de $C_p$ aunque también hay usos en los que las recompensas están en ese rango y el valor de $C_p$ es diferente.

### Algoritmo de expansión UCT

El algoritmo $UCT$ pertenece a la familia de algoritmos MCTS y por ello es similar al MCTS general que hemos visto en la sección 2.1.1. En este, se va aplicando la fórmula $UCB$ hasta que se llega a un nodo que no ha sido totalmente expandido. Se selecciona un hijo que no forme parte del árbol de búsqueda aleatoriamente y se añade a este. Después se aplica la política por defecto (aleatoria en el caso más sencillo) hasta que se llega a un estado terminal y se obtiene una recompensa $r$. El valor de $r$ se propaga hacia atrás por todos los antecesores hasta llegar a la raíz.

Cada nodo $v$ almacena dos valores, $N(v)$ que es el número de veces que ha sido visitado ese nodo y $Q(v)$ que es la recompensa acumulada de todas las iteraciones que han pasado por el nodo. El cociente $\frac{Q(v)}{N(v)}$ es una aproximación del valor teórico del nodo ya que corresponde a la media de las recompensas de todas las simulaciones.

Para que todos los subárboles mantengan la estructura de los MCTS, cada vez que un nodo forma parte de una iteración desde la raíz sus valores se actualizan. Una vez terminado el presupuesto computacional, el algoritmo devuelve la acción del hijo de la raíz con el mayor valor teórico.

En el algoritmo 2 tenemos el pseudocódigo del algoritmo UCT. Cada nodo $v$ almacena cuatro atributos: su estado asociado $s(v)$, la acción que aplicada al estado de su padre y que lleva hasta él $a(v)$, $Q(v)$ que es la recompensa acumulada y $N(v)$ el número total de visitas. La función $f(s, a)$ devuelve el estado $v'$ resultante de aplicar la acción $a$ al estado $s$. Por eficiencia en términos de memoria el estado $s(v)$ a veces no se almacena y se va calculando mientras se baja por el árbol en cada iteración.

**Algoritmo 2** Pseudocódigo del algoritmo UCT

**function** UCT($s_0$)
  crear nodo raíz $v_0$ con $s_0$ como estado
  **while** haya presupuesto computacional **do**
    $v_l = PoliticaArbol(v_0)$
    $r = PoliticaDefecto(s(v_l))$
    $Propagacion(v_l, r)$
  **return** $a(MejorHijo(s_0, 0))$

**function** PoliticaArbol($v$)
  **while** $v$ no sea un estado terminal **do**
    **if** $v$ no está totalmente expandido **then**
      **return** $Expandir(v)$
    **else**
      $v = MejorHijo(v, C_p)$
  **return** $v$

**function** Expandir($v$)
  seleccionamos una acción no expandida $a$ de las posibles de $A(s(v))$
  añadimos un nuevo hijo $v_0$ de $v$
  con $s(v_0) = f(s(v), a)$
  y $a(v_0) = a$
  **return** $v_0$

**function** MejorHijo($v$,$c$)
  **return** $\underset{v_0 \in hijo\ de\ v}{\text{máx}} \frac{Q(v_0)}{N(v_0)} + 2c\sqrt{\frac{2\ln N(v)}{N(v_0)}}$

**function** PoliticaDefecto($s$)
  **while** $v$ no sea un estado terminal **do**
    elegimos $a \in A(s)$ aleatoriamente
    $s = f(s, a)$
  **return** recompensa de $s$

**function** Propagación($v, r$)
  **while** $v$ no sea nulo **do**
    $N(v) = N(v) + 1$
    $Q(v) = Q(v) + r$
    $v =$ padre de $v$

## 2.3. Algoritmos genéticos

Los árboles de búsqueda de Monte Carlo nos pueden ofrecer una buena aproximación para construir un jugador automático que tenga en cuenta estados futuros. Sin embargo, seguimos dependiendo de una buena función de evaluación de esos estados que nos indique cuán favorables son para cada jugador y así poder compararlos entre sí. Para solucionar esto una posible alternativa es el calcular la importancia que se le debe dar a cada variable del estado de forma que se optimice el número de partidas ganadas al disponer de información de mayor calidad. Por ello, vamos a ver a continuación como funcionan los algoritmos genéticos y de que forma nos pueden ser de utilidad.

El uso de algoritmos genéticos es de especial importancia cuando el problema sobre el que se está trabajando puede tener varias soluciones o directamente no hay una solución óptima y se quiere estudiar cuál es una alternativa lo suficientemente buena.

Los algoritmos genéticos fueron inventados por John Holland en los años sesenta y desarrollados en mayor profundidad por él, sus estudiantes y colegas de la Universidad de Michigan durante los años sesenta y setenta. El objetivo de estos estudios era estudiar el proceso de adaptación que sufren las poblaciones de individuos en la naturaleza y buscar formas de llevar estos procesos a sistemas informáticos. Uno de sus estudiantes, David E. Goldberg, define en [21] (p.1) los algoritmos genéticos como: "algoritmos de búsqueda basados en las mecánicas de selección natural y genética que combinan la supervivencia del mejor con un intercambio de información estructurado pero basado en la aleatoriedad".

El funcionamiento de un algoritmo genético es el siguiente: partiendo de una población de individuos se seleccionan los que se consideren mejores y se cruzan para obtener la siguiente generación. Un porcentaje de estos individuos obtenidos se someten a mutaciones, es decir, se alteran algunas de sus características, con el fin de obtener una mayor diversidad y conseguir que las nuevas generaciones sean mejores que las anteriores.

Debido a estar basados en las mecánicas de selección natural se toman prestados muchos términos de la biología para hablar sobre algoritmos genéticos. Dado un problema determinado, se denomina **cromosoma** a una solución candidata de este problema y hablamos de **genes** para referirnos a cada uno de los valores que toman las variables en esta solución. Estos genes pueden ser tanto valores discretos (los más usados son los binarios) o continuos, como cuando se quiere obtener un valor extremo de una función sobre $\mathbb{R}$. Cada uno de los cromosomas determinan a un **individuo** que son los operandos sobre los que se trabaja. El conjunto de todos los individuos recibe el nombre de **población**.

En la naturaleza, la idoneidad de un individuo está directamente relacionada con su supervivencia pero en los algoritmos genéticos informáticos esto no es posible. Para ello se definen las llamadas funciones de ***fitness*** que indican como de bueno es un individuo. De esta forma podemos otorgar una mayor probabilidad de transmitir sus genes a un individuo con una puntuación de *fitness* alta.

Para elaborar un algoritmo genético lo primero que se debe hacer es tener claro que variable constituye cada uno de los genes y determinar la estructura de los cromosomas. Con ello se genera la población inicial, que generalmente está formada por individuos donde los genes toman unos valores aleatorios dentro de unos permitidos. Una vez se ha hecho esto

se evalúa el *fitness* de cada individuo para ver cuales aportan una mejor solución. Es en este momento cuando entran en juego los tres operadores principales que definen los algoritmos genéticos: selección, cruce y mutación.

### 2.3.1. Selección

Una vez se tiene a los individuos de una población con su valor de *fitness* se debe elegir cuales de ellos se van a usar como padres para una nueva generación, es decir, se debe elegir que individuos van a transmitir su información genética. La selección abarca esos algoritmos que nos llevan a la elección de los individuos padre. En [33] se nos describen algunos de los algoritmos de selección más conocidos y sus características. De esto vamos a hablar de los siguientes: el método de selección por ruleta, el método de selección por rango y el método de selección por torneo.

#### Selección por ruleta

En este método de selección la posibilidad de ser elegido de cada individuo es proporcional a su valor de *fitness*. La probabilidad de cada individuo $i$ de ser elegido en una población de tamaño $n$ con una función de *fitness* $f$ viene dada por la siguiente fórmula:

$$p(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)}$$

Este método es el más rápido y simple, pero tiene el inconveniente de que puede converger prematuramente, dejándonos en un extremo local de la función objetivo.

#### Selección por rango

Es similar al método de selección por ruleta. En el método de selección por rango se le asigna a cada individuo un rango tras ordenarlos según su valor de *fitness*. El mejor recibe el rango $N$ (siendo $N$ también el tamaño de la población) y el menor el rango 1. Una vez se tiene el rango asignado a cada individuo se hace un método de selección por ruleta pero sobre el rango, no sobre el *fitness*. De esta forma los individuos mejores siguen teniendo una mayor probabilidad, pero por ejemplo un individuo en la posición $k$ tiene la misma probabilidad sea el doble o diez veces peor que el que está en la posición $k+1$. Así la probabilidad de ser elegido de cada individuo $i$ en una población de tamaño $n$ es:

$$p(i) = \frac{rango(i)}{n \times (n-1)}$$

Este método es fiable y robusto, pero tiene una convergencia muy lenta.

#### Selección por torneo

En este método se escoge de forma aleatoria una subpoblación de individuos de un tamaño $k$ menor a la total. Estos individuos son comparados entre sí y se selecciona el que

tiene un *fitness* mayor. Este método garantiza la diversidad ya que todos los individuos tienen la misma posibilidad de participar en el torneo pero esto también lleva a que su velocidad de convergencia sea lenta. Además se puede ajustar la selección según el tamaño del torneo. La probabilidad de ser seleccionado de cada individuo $i$ es:

$$p(i) = \begin{cases} \frac{\binom{k-1}{n-1}}{\binom{k}{n}} & \text{si } i \in [1, n-k-1] \\ 0 & \text{si } i \in [n-k, n] \end{cases}$$

Otro factor a tener en cuenta en la selección de individuos es el **elitismo**. Debido a que en todos los métodos de selección es posible que los mejores individuos no sean elegidos es posible hacer un número determinado de ellos pasen por defecto a la siguiente generación.

### 2.3.2. Cruce

Una vez hemos elegido los individuos la pareja de individuos *padre* mediante el método de selección que hemos considerado adecuado llega el momento de generar a los individuos *hijos* a partir de estos. Mediante el cruce se obtienen generalmente dos individuos *hijo* cuyos genes pertenecerán a uno de sus dos *padres* obteniendo así una nueva combinación que pueda mejorar su *fitness*. Existen muchos métodos de cruce pero aquí nos vamos a centrar en tres, mostrados en [22]: cruce de $n$-puntos, cruce uniforme y cruce con combinación lineal.

#### Cruce de $n$-puntos

Este es uno de los métodos de cruce más conocidos y usados. Dados los cromosomas que representan a cada uno de los individuos *padre* seleccionados, con $k$ genes cada uno de ellos, se escogen de forma aleatoria $n$ puntos de 1 a $k$ y se marcan como puntos de corte, delimitando así $n-1$ trozos para cada padre. Una vez se ha hecho eso, se crean dos individuos *hijo*. El primero se obtiene concatenando estos trozos obtenidos manteniendo el orden pero alternando en cada punto de corte el padre del que se selecciona. El segundo se crea con los trozos que no se han seleccionado de la misma forma. Veamos un ejemplo con 2 puntos de cruce: dados dos individuos padre que hemos seleccionado de la población, colocamos de forma aleatoria dos puntos de cruce (en este caso en la posición 2 y 4). Así se generan los dos hijos que toman información genética de un padre hasta la posición 2, cambian el padre hasta la posición 4 y en la posición 4 vuelven al padre que tenían antes.

Generalmente se utilizan uno o dos puntos de corte aunque según el problema puede ser interesante utilizar más. Podemos ver un ejemplo de cruce de dos puntos en la figura 2.2.

#### Cruce uniforme

Otra alternativa es la que viene dada por escoger por separado "tirando una moneda" de que padre se escoge cada gen. Esto recibe el nombre de cruce uniforme. Así, dados dos individuos padre seleccionados y los dos hijos que buscamos generar se selecciona de forma aleatoria e individual para cada gen el individuo hijo que lo va a heredar. Por ejemplo,

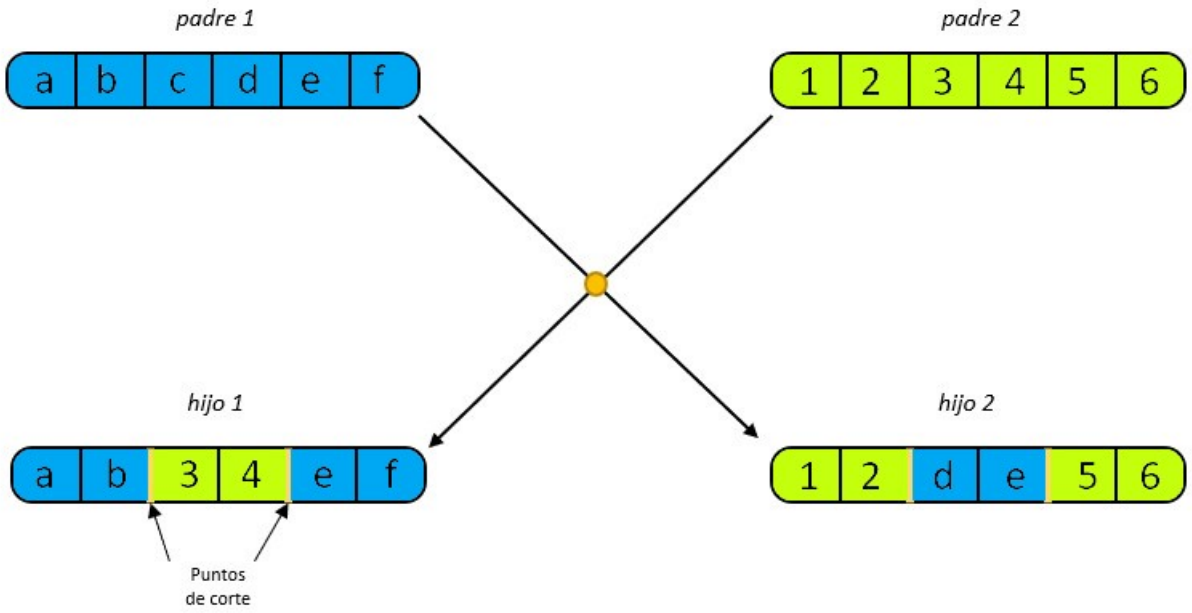


Figura 2.2: Cruce de dos puntos

si llamamos $gp1$ a los genes de un padre y $gp2$ a los genes del otro podemos obtener por ejemplo los siguientes individuos hijos:

Hijo 1: $gp1$ $gp1$ $gp2$ $gp1$ $gp2$ $gp2$

Hijo 2: $gp2$ $gp2$ $gp1$ $gp2$ $gp1$ $gp1$

### Cruce con combinación lineal

Como se puede apreciar, en los dos métodos anteriores el cruce no servía para añadir información nueva a los individuos hijo, sino que simplemente se combinaba de diferente forma los genes de los padres. Esto puede ser suficiente si se están trabajando con variables binarias, pero es posible que no funcione tan bien si se trabaja sobre variables continuas y sea necesario introducir otros mecanismos para garantizar una mayor diversidad.

Una posible alternativa a esto es hacer que los genes de los individuos hijo sean una combinación lineal de la de los padres. Así, sea un gen $i$ que uno de los padres toma el valor $g1$ y en el otro $g2$, generamos de forma aleatoria un valor $\alpha \in [0, 1]$ y calculamos el valor de ese gen en cada uno de los hijos de la siguiente forma:

$$g_{hijo1} = \alpha g1 + (1 - \alpha)g2$$

$$g_{hijo2} = (1 - \alpha)g1 + \alpha g2$$

Así obtenemos que tanto $g_{hijo1}$ como $g_{hijo2}$ están dentro del intervalo formado por $g1$ y $g2$ pudiendo generar posibles nuevos valores para el gen. Sin embargo, y aunque este método produce una mayor diversidad que los anteriores no es perfecto ya que como los valores del gen de los dos padres acotan a los dos hijos es posible acabar convergiendo demasiado rápido a un extremo local de la función objetivo.

### 2.3.3. Mutación

Cuando se cruzan dos individuos y se obtienen nuevos la información genética de estos depende directamente de la de sus padres. Por ello es posible, si no se introducen cambios, que la función de *fitness* se quede atascada en un extremo local. Por ello entra en juego el operador de mutación.

Una vez se genera un nuevo individuo existe la posibilidad de que este mute para introducir así mayor diversidad a la población. Existen muchos métodos de mutación como podemos ver en [41]. Por ejemplo el denominado *bit-flip* que se utiliza sobre variables binarias y transforma un gen aleatorio que tenga como valor 1 a 0 o al revés o el método de intercambio, que intercambia los valores de dos genes. Otra alternativa para generar diversidad en una población es introducir en ella un pequeño número de individuos generados aleatoriamente.

Durante el desarrollo de esta memoria nos centraremos en dos métodos: el método de mutación de *creep* y el método de mutación por *scramble*.

#### Mutación de *creep*

En este método se escoge uno o varios genes aleatorios del cromosoma y se les otorga un nuevo valor.

#### Mutación por *scramble*

En este método de mutación se selecciona un subconjunto de un tamaño determinado de los genes del individuo y se recolocan de una forma aleatoria. El método de mutación por cambio es una variedad de este método con un subconjunto de tamaño 2 y en el que se obliga a que los genes no terminen con el mismo valor.

Con esto ya disponemos de los conocimientos teóricos necesarios para implementar un jugador automático que tenga en cuenta los estados del juego futuros mediante árboles de búsqueda de Monte Carlo y un algoritmo genético capaz de aproximar una función de evaluación de estados. En el capítulo siguiente veremos como funciona *Hearthstone* y por qué su estudio supone un reto interesante.

# Capítulo 3

# Hearthstone

## 3.1. La Inteligencia Artificial y los juegos

La inteligencia artificial y los juegos han estado ligados prácticamente desde los inicios de esta. Gran parte de la investigación realizada sobre la IA aplicada en juegos se ocupa de construir agentes para jugar, con o sin un componente de aprendizaje [46]. Históricamente, crear agentes para jugar ha sido el primer y, durante mucho tiempo, el único enfoque para usar la IA en los juegos. Incluso desde antes de que se reconociera la inteligencia artificial como campo, los primeros pioneros de la informática escribieron programas de juego porque querían probar si los ordenadores podían resolver tareas que parecían requerir "inteligencia". Alan Turing implementó el algoritmo Minimax para jugar al ajedrez [45] aunque esta implementación solo fue teórica ya que en ese momento no existía el hardware capaz de ejecutarlo.

El primer software que logró jugar al nivel de las personas en un juego fue programado por A.S. Douglas en 1952 en una versión digital del juego "Tres en Raya" y unos años más tarde, fue el primero en inventar lo que ahora se conoce como **aprendizaje por refuerzo** usando un programa que aprendió a jugar a las damas jugando contra sí mismo [34]. La mayoría de las primeras investigaciones sobre la IA de los juegos se centraron en los juegos de mesa clásicos, como las damas y el ajedrez. Después de más de tres décadas de investigación sobre árboles de búsqueda, en 1992, Gerald Tesauro desarrolla el software de backgammon llamado TD-Gammon. TD-Gammon emplea una red neuronal artificial que se entrena mediante **aprendizaje por diferencias temporales** jugando al backgammon contra sí mismo varios millones de veces [43, 44]. TD-Gammon logró jugar al nivel de un jugador de backgammon humano de primer nivel.

En 1994, el agente Chinook Checkers logró vencer a la campeona mundial de damas Marion Tinsley [37]. En 2007 se probó que el mejor jugador no podría obtener un mejor resultado que un empate frente a Chinook [36]. Durante décadas el ajedrez fue el modelo sobre el que se probaron innumerables métodos nuevos de IA hasta se desarrolló un software capaz de jugar mejor que los humanos. El software que primero exhibió la capacidad sobrehumana de jugar al ajedrez, Deep Blue de IBM, consistía en un algoritmo Minimax con

modificaciones específicas para el ajedrez y una función de evaluación del tablero [25]. Deep Blue ganó a Garry Kasparov en 1997. Hoy en día, este software ejecutado en un ordenador actual ganaría a cualquier persona.

Después de Deep Blue, el siguiente éxito de IBM fue Watson, un sistema capaz de responder a las preguntas dirigidas en lenguaje natural. En 2011, Watson compitió en un concurso de televisión y ganó a los antiguos ganadores del juego [16].

Tras los éxitos de la IA en los juegos de mesa tradicionales, el último hito en juegos de mesa de la IA se alcanzó en 2016 en el juego Go. Poco después de Chinook y Deep Blue, este juego se convirtió en la nueva referencia de la IA aplicada a juegos con un factor de ramificación que se aproxima a 250 y un vasto espacio de búsqueda muchas veces mayor que el del ajedrez. Mientras que el hecho de alcanzar el nivel humano en el Go se esperaba en un futuro lejano [30], en 2016 Lee Sedol (jugador profesional de Go) perdió un partido al mejor de cinco rondas contra el software AlphaGo de Google que abordaba el problema mediante **aprendizaje por refuerzo profundo** [40]. En mayo de 2017, AlphaGo ganó un partido al mejor de tres rondas contra el número 1 del mundo, Ke Jie, corriendo en un solo ordenador. Con esta victoria, el Go fue el último gran juego de mesa clásico donde los ordenadores han alcanzado un rendimiento superior al humano.

Pero los juegos de mesa clásicos no son los únicos juegos objeto de estudio. En los últimos 20 años, se ha ido formando una comunidad científica dedicada a investigar las aplicaciones de la IA en juegos que no sean de mesa, en particular los videojuegos. Gran parte de la investigación en esta comunidad se centra en el desarrollo de la IA para jugar juegos (de la manera más efectiva, más humana o con cualquier otro requisito). En 2014, los algoritmos desarrollados por Google DeepMind aprendieron a jugar varios juegos de la clásica consola de videojuegos Atari 2600 mejor que cualquier persona únicamente con las entradas de píxeles en bruto [31].

Otras aplicaciones de la IA en los videojuegos son también muy importantes. Uno de ellos es el uso de la IA para generar contenidos dentro del videojuego como en los videojuegos de los años 80 Rogue (Toy y Wichmann, 1980) y Elite (Acornsoft, 1984). Más recientemente también se ha utilizado la IA para analizar juegos y modelar a los jugadores. Existen juegos como Nevermind (Flying Mollusk, 2016) que identifica los cambios emocionales del jugador y adapta el juego en consecuencia.

Recientemente, la investigación sobre agentes veraces (believable agents) en los juegos ha abierto nuevos horizontes en la IA aplicada a juegos. Una forma de cuantificar la credibilidad es programar agentes que puedan pasar las pruebas de Turing basadas en el juego. Este test es una variante del test de Turing en el que varios jueces deben adivinar si el comportamiento en un juego es el de un humano o un robot de juego controlado por una IA [23, 39]. Lo más remarcable es que dos bots controlados por una IA lograron pasar la prueba de Turing en el "Unreal Tournament 2004" (Epic Games, 2004) en el centenario de Turing en 2012 [38].


(a) Esbirro (b) Hechizo (c) Arma

Figura 3.1: Tipos de carta

## 3.2. ¿Qué es Hearthstone?

Hearthstone: Heroes of Warcraft es un juego de cartas coleccionables (CCG) por turnos online desarrollado por Blizzard Entertainment en el que se enfrentan dos jugadores usando mazos construidos por ellos mismos. Cada uno de los mazos gira alrededor de un *héroe* de una de las 9 clases disponibles. Los héroes cuentan con un total de 30 puntos de vida y el objetivo del jugador es reducir hasta 0 los puntos de vida del oponente para así ganar la partida.

Hearthstone es un juego por turnos en el que los jugadores se van alternando para jugar sus cartas. Cada una de estas cartas tiene asociado un coste, esto es, los puntos de maná que se deben pagar para poder usarlas. Cada jugador empieza la partida con un punto de maná y al comienzo de cada turno se recuperan los puntos de maná gastados en el turno anterior y se añade uno más.Los puntos de maná pueden llegar hasta un máximo de 10 puntos. Una mayor cantidad de puntos de maná da acceso a jugar cartas más poderosas y con más habilidades lo que aumenta la complejidad del juego con el paso de los turnos.

Como hemos dicho antes, los jugadores necesitan construir sus propios mazos para poder jugar una partida. Estos consisten en un total de 30 cartas que pueden ser elegidas entre más de 2000 cartas y únicamente puede haber 1 o 2 copias de la misma carta en un mazo. El juego está sometido a actualizaciones cada cierto tiempo en las que se añaden nuevas cartas y habilidades. Cada carta de la colección es única, es decir, no existe una carta que tenga un nombre diferente y el mismo efecto. La elección del *héroe* es muy importante a la hora de construir el mazo porque cada *héroe* da acceso a una serie de cartas que son especificas de su clase y no pueden ser utilizadas en un mazo que lleve un *héroe* de otra clase.

Las cartas pueden ser de tres tipos diferentes: esbirros, hechizos o armas. En la figura 3.1

se muestra un ejemplo de cada tipo de carta. Los esbirros son cartas que se colocan en el campo de batalla, hasta tener un máximo de 7 esbirros en tu campo (las posiciones en el campo de batalla son relativas, solo existe una forma de colocar una carta en un campo vacío). Cuentan con un número que indica su ataque, esto es el el daño que hacen, su defensa, que indica cuanto daño pueden soportar antes de dejar de estar en el campo y su coste, que son cuantos puntos de maná tienes que gastar para ponerlo en juego. Cada esbirro puede, al turno siguiente de ser invocado, atacar al héroe enemigo o a uno de sus esbirros haciéndole tanto daño como su ataque. Cuando un esbirro ataca a otro esbirro se produce un enfrentamiento entre ambos en el cuál ambos son dañados por el ataque del otro mientras que si atacan al héroe no reciben daño. Los esbirros pueden tener otras habilidades que crean efectos en el juego. Por ejemplo, un esbirro con viento furioso (o windfury) puede atacar dos veces en un turno.

Los hechizos son cartas que generan otros efectos en el juego. Estos son muy diversos y no tendría sentido intentar clasificarlos. Algunos de estos efectos pueden ser hacer daño a un esbirro o héroe, invocar a un esbirro aleatorio o robar un número de cartas del mazo. Los secretos son una clase especial de hechizos. Los efectos de un secreto no se aplican inmediatamente después de jugarlos si no que dependen de que una determinada condición se cumpla para que se activen (por ejemplo, que pasen 2 turnos). Una vez activados funcionan como cualquier otro hechizo.

Las armas son cartas que se equipan al héroe dotándole a este de capacidad de ataque como si fuera un esbirro. Aparte del coste, tienen asociados dos valores, el ataque que al igual que en los esbirros es el daño que hacen y la durabilidad que es el número de ataques que puede hacer el héroe con ella antes de que sea destruida y eliminada del campo. Un héroe solo puede tener equipado un arma simultáneamente.

Los mazos de Hearthstone suelen girar en general alrededor de una temática común. Los efectos de la mayoría de las cartas suelen influir positivamente en otras lo que hace que formen buenas combinaciones y aumente el valor de cada carta dentro del contexto del mazo. Por este motivo, el valor de una carta depende en gran medida de la mano del jugador, los elementos que hay en juego y del mazo. Un ejemplo de esta sinergia podrían ser los esbirros de la clase "Murloc" que por separado son esbirros muy débiles pero suelen tener habilidades de potencian al resto de esbirros de la misma clase.

Cuando empieza la partida cada jugador empieza con 3 cartas y roba otra de su mazo al comienzo de cada uno de sus turnos. Además el jugador que no empieza jugando tiene además una carta de coste 0 llamada *La Moneda* que permite disponer de un punto de maná adicional durante el turno en cuál sea utilizada.

A continuación vamos a describir como es un turno de un jugador pero primero vamos a ver una imagen del tablero y las zonas que tiene. En la figura 3.2 se muestra el tablero de juego de una partida en curso, el tablero es simétrico para ambos jugadores y tiene las siguientes zonas:

1. *Mano*: Zona donde están las cartas que podemos jugar pagando su respectivo coste de maná cuyo tamaño máximo es de 10 cartas. En la parte inferior central tenemos las cartas, esta zona está oculta para el oponente. A la derecha podemos ver los cristales

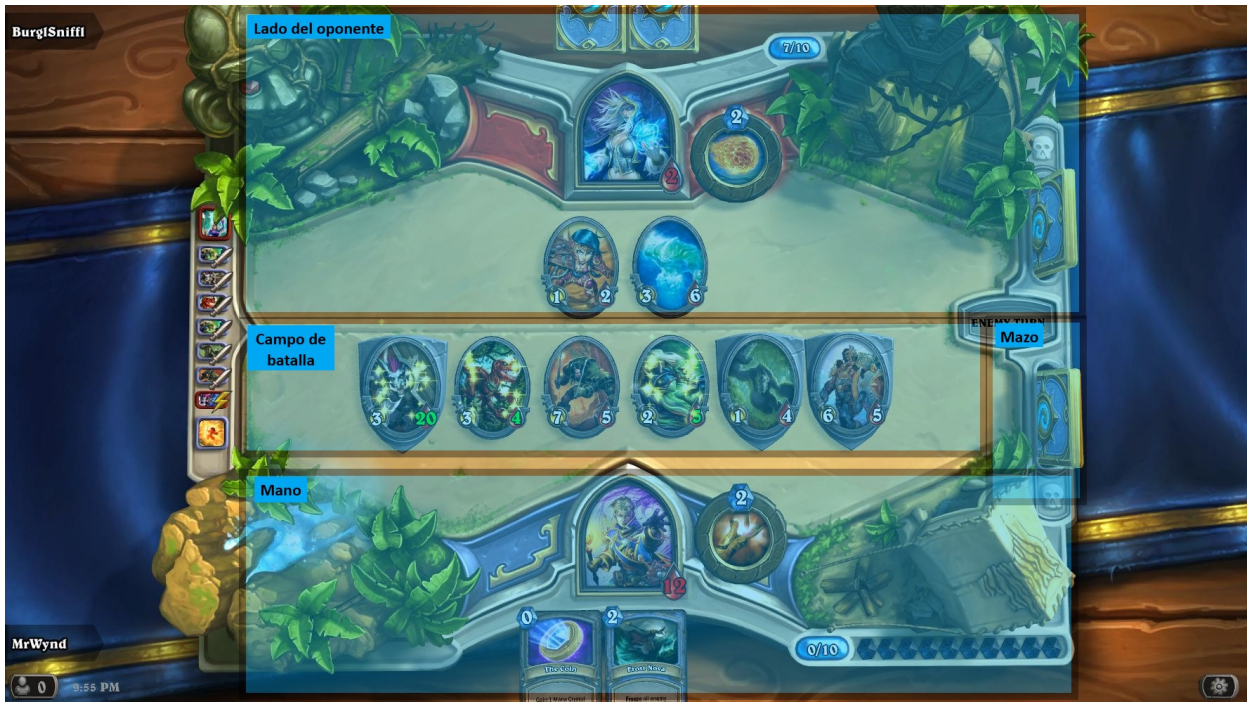


Figura 3.2: Tablero de una partida de Hearthstone

de maná que tenemos disponibles para jugar las cartas (en la figura 0/10). Encima de las cartas tenemos a nuestro héroe donde a su derecha tiene el poder característico de este y a su izquierda el arma equipada (en la figura no hay ninguna equipada) y en la esquina inferior derecha de su imagen los puntos de vida restantes.

2. *Campo de batalla*: Es la zona donde están las cartas de esbirro que ya hemos jugado y mantienen su vida por encima de 0.

3. *Mazo*: Es una zona oculta para ambos jugadores. Cada jugador sabe que cartas hay en su propio mazo pero desconoce el orden de estas. Si un jugador se queda sin cartas en el mazo pierde una cantidad de vida cada turno que se va incrementando con el paso de estos.

4. *Oponente*: El lado del oponente está constituido por las mismas zonas que el nuestro. Vemos el lado del oponente de la misma manera que el nuestro a excepción de las cartas de su mano.

Cuando el turno de un jugador comienza este recupera todos los cristales de maná gastados anteriormente como hemos comentado antes, y roba la primera carta de su mazo. En esta situación, el jugador tiene las siguientes opciones:

- Jugar una carta de la mano, solo podrá jugar las cartas de la mano si le quedan suficientes cristales de maná. En el caso de la imagen, el jugador solo podría jugar la carta de la izquierda con coste 0.

- Atacar con un esbirro, el ataque con esbirros puede ser de dos formas dependiendo de cuál sea el objetivo de este: atacar al héroe del oponente o atacar a un esbirro del oponente. Cuando se ataca al héroe del oponente el héroe pierde tantos puntos de vida como ataque tenga el esbirro. Cuando se ataca a un esbirro, se produce un enfrentamiento entre ambos donde el ataque de uno disminuye la defensa del otro y viceversa. En el caso de la figura, cada uno de los 6 esbirros del jugador podría atacar al héroe del oponente o a uno de los 2 esbirros que controla. Un esbirro solo puede realizar un ataque por turno a menos que tenga una habilidad especial.
- Atacar con el arma equipada al héroe, es como el ataque con un esbirro donde el ataque del esbirro es el poder del arma y la defensa del esbirro son los puntos de vida restantes del héroe. En la imagen el héroe no tiene ningún arma equipada por tanto no podría realizar esta acción.
- Utilizar el poder del héroe, el poder del héroe es una habilidad intrínseca de cada héroe que es equivalente a una carta que siempre tiene en la mano y puede jugar infinitas veces mientras pague su coste de maná. En el poder del héroe de la imagen el coste es 2.
- Pasar turno, el jugador finaliza su turno y empieza el del oponente.

Durante un turno, el jugador puede realizar tantas acciones como quiera hasta pasar turno. En el caso de la imagen, el jugador no tiene porque restringirse a un ataque con un único esbirro sino que puede realizar varios.

Otro aspecto a considerar a la hora de jugar al Hearthstone es el estilo de juego que vamos a seguir. Esto es porque generalmente cada mazo tiene uno o varios estilos de juego diferentes asociados y suelen estar fuertemente ligados con la clase del héroe. Es interesante crear agentes que sean capaces de aprender un determinado estilo de juego y aplicarlo. En el Hearthstone distinguimos tres estrategias principales aunque de estas pueden surgir diferentes modificaciones:

1. *Aggro*: En este estilo de juego se prioriza el derrotar al adversario lo antes posibles dedicando todos los recursos disponibles en atacar a este.
2. *Control*: El objetivo de este estilo es mantener el control de la partida con el objetivo de que tu oponente no pueda desarrollar su juego para en un turno tardío ganar.
3. *Mid-range*: Este es el estilo más genérico de todos y se encuentra a caballo entre *Aggro* y *Control*.

## 3.3. ¿Por qué el Hearthstone es un reto interesante para la IA?: trabajos relacionados con el Hearthstone

Como hemos visto en la sección anterior Hearthstone es un juego complejo en el que en muchas variables influyen en el resultado final. En cada momento de juego el número de

decisiones alternativas es muy elevado y muchas veces también depende el orden en el que se toman.

En [15], los autores presentan algunas características que hacen que Hearthstone (y en general los juegos de cartas coleccionables) sea un buen campo de estudio para la IA. Entre estas numeran:

- **Estados parcialmente observables:** A la hora de jugar, muchos aspectos del estado del juego están ocultos para el jugador (y por extensión al agente). Este desconocimiento de ciertos datos, como por ejemplo las cartas que forman el mazo del oponente, hacen que la forma de afrontar la construcción de un jugador automático sea distinto a cualquier otro juego cuyo estado sea completamente observable, como el ajedrez. Esto lleva a que sea imposible, o por lo menos muy complejo, el estimar como va a reaccionar o las jugadas que tiene disponible el adversario.

- **Alta complejidad:** En [3], base de datos de cartas de *Hearthstone* oficial, se recoge que actualmente el juego dispone 2496 cartas únicas. Esto lleva a que a pesar de ser un juego con reglas sencillas, el número de variables a tener en cuenta sea bastante elevado.

- **Aleatoriedad:** Además de la aleatoriedad intrínseca a cualquier juego de cartas, ya que se depende del orden en el estas estén en el mazo, Hearthstone dispone de más efectos aleatorios que otros juegos similares como puedan ser *Magic: The Gathering* [6] o *Gwent* [5]. Existen, por ejemplo, cartas con efectos aleatorios. Esta aleatoriedad dificulta la capacidad de decisión y obliga al jugador a irse adaptando a lo que sucede.

- **Construcción de mazos:** La gran variedad de cartas de las que se dispone y las diferentes formas de relacionarse entre ellas lleva a que la elección de las cartas a incluir en el mazo sea otra cuestión de estudio.

- **Meta-juego cambiante:** Aproximadamente cada tres meses, *Hearthstone* recibe una expansión que añade más cartas a su juego. Estas nuevas cartas introducen nuevos efectos y por lo tanto nuevas estrategias que pueden ser más efectivas. Así el meta-juego (donde el prefijo *meta* se corresponde con las iniciales de *Most Efficient Tactic Available*) cambia de forma constante y estrategias de juego que antes eran poco efectivas pasan a serlo más y al revés.

Se pueden encontrar similitudes entre Hearthstone y otros juegos como el Poker donde hay una gran cantidad de información oculta y el hecho de conocer las cartas de tu oponente te da una considerable ventaja y por ello los jugadores profesionales emplean gran parte de sus esfuerzos en predecir la mano del rival. Aunque hay que remarcar que en el Hearthstone esto es mucho más complejo ya que la cantidad de cartas disponibles es mucho mayor que en el Poker. Existen plug-ins como Hearthstone Deck Tracker que intentan predecir las cartas que juega el rival en tiempo real [4].

La mayoría de los trabajos publicados sobre Hearthstone tienen como temática central diferentes métodos de cómo jugar [35, 42, 47]. Dentro de estas publicaciones, gran parte

de ellas utilizan los árboles de búsqueda de Monte Carlo como algoritmo para estudiar el espacio de búsqueda y así crear un agente que sea capaz de jugar al juego [35, 42]. El principal problema de los árboles de búsqueda es la forma de lidiar con la información oculta y la imposibilidad de utilizar la búsqueda minimax a menos que se tenga información sobre la mano del oponente [14]. Otros agentes, como los que implementaremos en este trabajo, simplemente consideran el espacio de búsqueda hasta el final del turno sin intentar predecir el movimiento del oponente.

A raíz de los diferentes estilos de juego y los agentes de juego que no evalúan únicamente el resultado de la partida son necesarias unas funciones de evaluación del estado. Estas funciones de evaluación deben permitir al agente saber como de bueno es el estado al que se llega tras aplicar una determinada acción al estado actual. Es importante tener en cuenta que esta función de evaluación depende también del estilo de juego del agente. Por ejemplo, si un agente está jugando un mazo *Aggro* ponderará más a la hora de evaluar el estado el hecho de que el oponente tenga menos vida que si está jugando un mazo *Control*. En el caso contrario, un agente si está jugando un mazo *Control* ponderará más el hecho de que el oponente no tenga muchos esbirros en el Campo de batalla que el mazo *Aggro*. Estos podrían ser algunos de los parámetros a tener en cuenta a la hora de evaluar, otros ejemplos de parámetros podrían ser: el número de cartas en la mano, el número de cartas restantes en el mazo, el poder de los esbirros en el campo de batalla o las habilidades de estos. Algunos de los trabajos realizados sobre mejoras en las funciones de evaluación para así obtener agentes que jueguen mejor son mediante algoritmos evolutivos [18, 19].

Otros retos a la hora de crear un agente son crear un jugador que de la sensación de ser "humano" o agentes que propongan un mayor o un menor reto a la hora de enfrentarse un jugador humano a ellos. Entendemos un jugador "humano" como un agente que tiene tiempos de respuesta similares a los de un humano y la elección de sus acciones podría ser explicable. Los agentes que muestran una mayor o menor dificultad a la hora de enfrentarlos ya están implementados en el propio juego pero no se ha publicado ningún trabajo relativo a ello.

Aunque el hecho de jugar al juego sea el tema que más atención reciba, también se han realizado estudios sobre como crear un mazo para jugar [17, 20]. Los jugadores humanos suelen construir los mazos a través de su experiencia jugando y el "metagame" del momento. El "metagame" está formado por los mazos y las estrategias que en un momento puntual del juego ganan a la gran mayoría de los mazos y estrategias. El hecho de crear los mazos mediante una IA puede hacer que se abran caminos a la creación de nuevas estrategias de juego u otros "metagames" diferentes. La creación del mazo suele ir muy ligada al héroe ya que hay cartas que son especificas de cada héroe y al estilo de juego.

## 3.4. El framework: Hearthstone-AI framework

El framework *Hearthstone-AI framework* [15] está basado en el simulador de código abierto *Sabberstone* [8]. Este framework está escrito en C# y extiende el framework original de *Sabberstone* con varias clases auxiliares para proporcionar a un agente medios sencillos

para acceder al estado del juego limitado a las variables que serían observables por un jugador humano.

Entrando en detalle, cada agente tiene que heredar de la clase `AbstractAgent`. Los métodos `InicializateAgent()` y `FinalizeAgent()` pueden usarse para cargar y guardar información al principio y al final de cada sesión (una o varias partidas). Además, los métodos `InitializeGame()` y `FinalizeGame()` se llaman al principio y al final de cada partida. Estos métodos son para configurar o actualizar una estrategia basada en los resultados obtenidos en partidas anteriores de la misma sesión.

Durante una partida, cada vez que el agente tiene que elegir una jugada el framework llama al método `GetMove()` del agente, este es el método en la que implementaríamos la estrategia de búsqueda del agente. En él, el agente recibe un objeto `POGame` (Player Observable Game) que representa la observación parcial del estado actual de la partida. Este contiene información sobre la parte visible del tablero, el conjunto de cartas que le quedan a su mazo (no el orden), las cartas en su mano y el número de cartas en la mano del oponente. Además, el mazo del oponente, así como las cartas de su mano, se sustituyen por cartas ficticias para asegurar que la información permanezca oculta al agente.

El agente tiene hasta 60 segundos de tiempo de cálculo para devolver un lista de acciones y terminar su turno. En caso de que el agente no haya terminado su turno, se devuelve el estado de juego como estaba pero con el turno cambiado (es como si la acción aplicada fuera un "pasar de turno"). Las jugadas de ambos jugadores se aplican hasta que se pueda determinar un ganador o se supere el número máximo de turnos (por defecto, 50 turnos). En este último caso la partida termina en empate.

La clase `POGameHandler` controla la simulación de una o varias partidas e informa del resultado de estas con un objeto del tipo `GameStats`. Este resultado incluye: el número de victorias, empates y derrotas, así como el tiempo total y medio de respuesta del agente.

## 3.5. Agentes ya implementados en el simulador

En esta sección comentaremos los dos agentes que vienen ya implementados en el framework y cual es su funcionamiento.

### 3.5.1. Random Agent

Este agente es el más sencillo de todos y no tiene mayor utilidad que comprobar si otro agente sigue una estrategia que le lleve a la victoria, ya que es prácticamente imposible perder contra un jugador que tome las decisiones al azar. Su funcionamiento es muy sencillo, consulta la lista con las posibles acciones que puede aplicar al estado actual y selecciona una aleatoriamente.

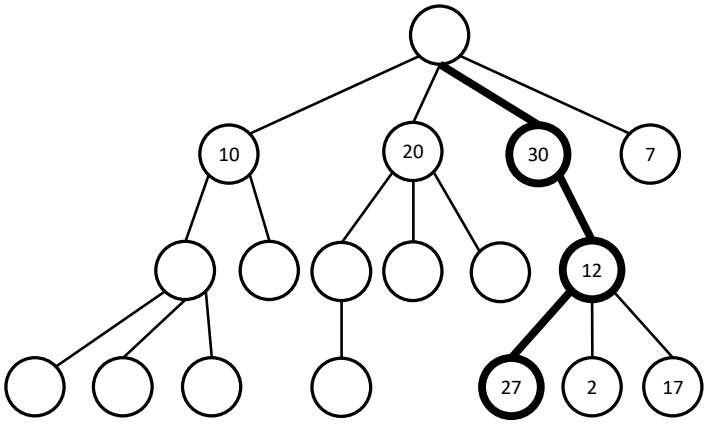


Figura 3.3: Ejemplo de búsqueda en el espacio de estados del Greddy Agent

### 3.5.2. Greedy Agent

El funcionamiento de este agente es el más sencillo de todos (obviando al *Random Agent*) aunque también es el primer agente que utiliza una estrategia de búsqueda y una función de evaluación de estado. El agente empieza con un estado del juego sobre el que tiene que aplicar una jugada para modificarlo y pasar al siguiente. Para ello el agente llama al método `Simulate()` con el estado actual y esta le devuelve un diccionario con los pares clave-valor donde cada clave es una de las posibles acciones y el valor es el estado resultante de aplicar la acción al estado actual.

Una vez generado este diccionario, el agente evalúa cada estado con la función de evaluación de estado `Score` que devuelve un número indicando como de buena es una acción. El agente selecciona la acción que tenga un valor de `Score` mayor.

Como hemos comentado en la sección 3.3, la función de evaluación de estado depende en gran medida del estilo de juego del mazo y por tanto del héroe. Para ello la función `Score` tiene 3 implementaciones diferentes: `AggroScore` (que premia el atacar directamente al héroe enemigo para acabar la partida cuanto antes) si el mazo usado es de la clase *WARRIOR*, `ControlScore` (que premia más que el oponente no tenga criaturas en el campo para así alargar la partida) si el mazo es de la clase *MAGE* y `MidrangeScore` (que es un término medio entre los dos anteriores) si es de cualquier otra.

En la figura 3.3 tenemos un ejemplo de como este agente visita los estados del espacio de estados de todo el turno del jugador. Los estados con una puntuación son aquellos que este agente evalúa. Como únicamente considera los hijos del estado en el que está, una vez selecciona el que tiene mayor puntuación descarta el resto de ramas.

## 3.6. La competición: Hearthstone-AI competition

La competición Hearthstone-AI competition es una de las múltiples competiciones de la conferencia anual sobre juegos organizada por el IEEE [1]. En esta competición existen diferentes categorías intentando cubrir el espectro de retos que presenta el Hearthstone.

Durante los primeros años de la competición [2] había dos categorías a las que poder inscribirse. Para las próximas competiciones, los organizadores planean ampliar la lista de categorías para, en palabras textuales de los organizadores, dar a los usuarios algo de tiempo a adaptarse al framework y al juego en sí [15]. Las categorías existentes en la competición del año 2019 fueron las siguientes:

- **Competición de mazos prehechos**: En esta categoría, los participantes recibirán una lista de seis mazos y jugarán todas las combinaciones posibles entre ellos. Únicamente tres de los seis mazos serán conocidos antes del envío de su participación. El objetivo a largo plazo de esta competición es desarrollar un agente que sea capaz de ganar jugando con cualquier mazo.

- **Competición de mazos creados por el competidor**: En esta categoría, se invita a los participantes a crear su propia baraja o elegir una de todas las que se pueden encontrar online. Encontrar combinaciones entre diferentes mazos y agentes que puedan vencer de forma consistente al resto es la clave de la competición. Además, como se tiene predefinido del mazo que va a utilizar el agente esta categoría permite optimizar el agente para adaptarse mejor a la baraja elegida.

Para determinar la clasificación, se utiliza un torneo que enfrenta a todos los agentes entre sí. Se determina el porcentaje de victorias y se les clasifica en función a estos resultados. En el caso de que este torneo fuera inviable se harían varios subtorneos que servirían como filtro para luego realizar un torneo con los mejores de cada subtorneo. Los enfrentamientos entre dos agentes se repetirán varias veces para así intentar evitar en gran medida la aleatoriedad del orden en el que se roban las cartas del mazo.

# Capítulo 4

# Creando un agente

En este capítulo vamos ver el trabajo que realizamos a la hora de crear un agente que sea capaz de jugar y ganar el mayor número posible de partidas contra otros agentes. Para ello dividiremos el capítulo en dos secciones: la primera dedicada al estudio de diferentes estrategias para la búsqueda en el espacio de estados posibles en el turno de un jugador y la segunda a la creación de una función de evaluación que le indique al agente como de bueno es un estado. Como la función de evaluación está fuertemente ligada al mazo y el estilo de juego, fijaremos el mazo al *MidrangeJadeShaman* que es uno de los mazos prehechos del framework cuyo estilo de juego es *Mid-range*.

## 4.1. Estrategias de búsqueda

En esta sección veremos diferentes estrategias de búsqueda para recorrer el espacio de estados. Como dado un estado se pueden aplicar diferentes acciones, el espacio de estados se puede ver como un árbol que nace del estado inicial y que sería el nodo raíz. Por ello, todas las estrategias de búsqueda son para espacios de estados con forma arbórea.

En la sección sobre los árboles de búsqueda de Monte Carlo 2.1 comentábamos que las simulaciones se ejecutaban hasta el final del juego produciendo un éxito o un fracaso. Nosotros reduciremos el espacio de estados únicamente a nuestro turno debido a la gran de cantidad de información oculta que supone desconocer las cartas de la mano y el mazo del rival. Los estados terminales serán los generados tras finalizar el turno en vez de los que indiquen el fin de la partida.

### 4.1.1. Greedy Tree Agent

El primer agente que hemos desarrollado extiende el *Greedy Agent* evaluando los estados terminales únicamente y elige aquel que lleva a una mejor situación final en lugar de hacer una única expansión por niveles y eligiendo en cada nivel el mejor nodo y expandiendo solo ese. Estos estados terminales son las hojas del espacio de estados como podemos ver en la figura 4.1. En ella, los nodos con un valor son las hojas que han sido evaluadas, se elige la

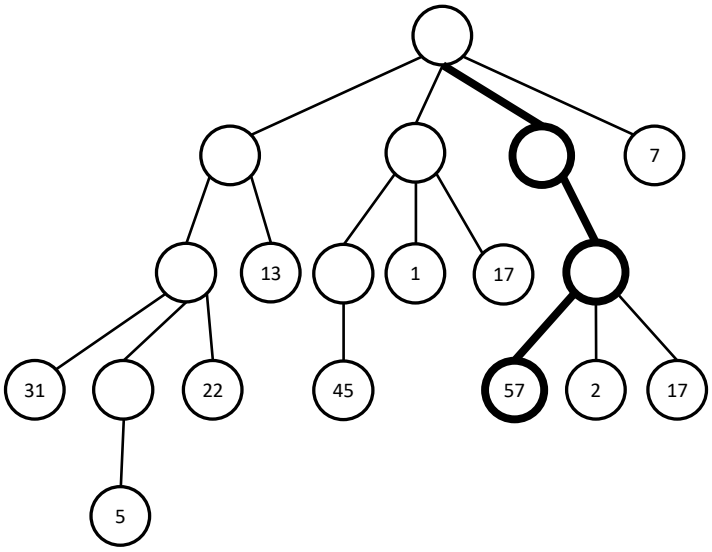


Figura 4.1: Ejemplo de búsqueda en el espacio de estados del Greddy Tree Agent

hoja con mayor valor y la secuencia de acciones que se aplican son las que permiten llegar hasta ella.

Para la implementación, recorremos el árbol en profundidad y almacenamos provisionalmente la secuencia de acciones para llegar a cada nodo desde el nodo raíz (el estado inicial del turno). Cuando llegamos a un estado terminal evaluamos el estado de este y si es mejor que el anterior guardado lo sustituimos. Cuando terminamos de recorrer todo el árbol devolvemos la secuencia de acciones de todo el turno. Únicamente guardamos en memoria la secuencia actual y la mejor en cada momento.

La principal ventaja es que al valorar más opciones que el *Greedy Agent* siempre se obtiene un estado al final del turno con mejor o igual valor que el obtenido por el *Greedy Agent* siempre y cuando utilicemos la misma función de evaluación de estado. La contraparte de esta estrategia de búsqueda es que como el factor de ramificación del espacio de estados es muy elevado es imposible explorar todos los nodos en un tiempo razonable. Es por este motivo que este agente únicamente lo utilizamos para unas pruebas iniciales contra el *Greedy Agent* con un mazo muy sencillo que no generaba espacios de estados muy grandes y por ello no lo tendremos en cuenta dentro de los resultados (sección 4.3).

### 4.1.2. Simulation Tree Agent

Este agente es el primero de los que hemos desarrollado basados en los árboles de búsqueda de Monte Carlo (MCTS). La idea de este agente es explorar parte del árbol en un tiempo razonable y no almacenar demasiada información en memoria. Para ello utiliza múltiples simulaciones con vistas a elegir la siguiente acción. Primeramente se inicializa el primer nivel del árbol que es el único que se va a guardar en memoria. La raíz consiste en el estado actual

del juego y cada uno de sus hijos es una de las posibles acciones. Los nodos hijos almacenan la acción que hay que aplicar al estado inicial para llegar a este y dos valores numéricos $v$ y $n$. El primero de ellos $v$ es el valor acumulado de todas las simulaciones lanzadas desde este nodo y devuelto por la función de evaluación en un estado terminal y $n$ es el número de simulaciones lanzadas desde este nodo. El valor $v$ sería comparable la recompensa de la fase de simulación en el algoritmo MCTS y por ello el nombre del agente. Ambos valores se inicializan a 0 en todos los nodos menos en el nodo que lleva la acción fin del turno. Como es un estado terminal, no podemos aplicarle ninguna acción y debido a ello se evalúa en este momento, añadiéndole además una visita.

Una vez inicializado el árbol, la simulación es igual que la explicada en la sección 2.1. Partiendo de un nodo y su correspondiente estado, se le aplica una acción seleccionada aleatoriamente al estado generando un nuevo estado. Este proceso se repite en el nuevo estado hasta llegar a un estado terminal. En este momento se evalúa el estado obtenido y se guarda la puntuación en el nodo seleccionado y se añade una visita. En la figura 4.2 se muestra un ejemplo de una iteración del agente:

1. *Selección*: La selección del nodo se hace en función de un parámetro llamado EXPLORATION_RATE (entre 0 y 1). Mientras el número de iteración respecto al número total de iteraciones sea menor que este parámetro se selecciona a cada nodo equitativamente y una visita a cada uno. Por ejemplo, si vamos a realizar un total de 1000 visitas y el valor de EXPLORATION_RATE es 0.7, las primeras 700 selecciones se realizan equitativamente. Una vez se supera la EXPLORATION_RATE se ordenan los nodos de mayor a menor valor promedio y se simula uno aleatorio de los siete mejores (si son menos de siete entonces es uno aleatorio simplemente).

2. *Simulación*: Al igual que en la etapa de *Simulación* de los MCTS, se aplica una política por defecto hasta llegar a un estado terminal y se devuelve el valor que resulta de aplicar la función de evaluación de estado a ese estado terminal.

3. *Actualización*: Se actualiza el nodo desde el cual se ha lanzado la simulación añadiendo una visita y el resultado devuelto por la simulación. Esta etapa sería equivalente a la de *Propagación hacia atrás* de los MCTS solo que al tener un único nivel el árbol no hace falta actualizar más que el propio nodo.

Cuando han terminado las simulaciones se selecciona el nodo con mejor promedio.

### 4.1.3. MCTS1 Agent

Este agente también basado en los árboles de búsqueda de Monte Carlo esta a caballo entre el *Simulation Tree Agent* (STA) y el MCTS teórico. La idea es construir un árbol de búsqueda incrementalmente en el espacio de estados de un turno almacenando más información del espacio de estados que el STA que se limitaba al primer nivel del árbol. Al igual que en el STA, inicializamos el árbol guardando en la raíz el estado actual del juego y en sus hijos los estados resultantes de aplicar cada una de las posibles acciones del estado actual. La información que guardamos en cada uno de los nodos es:

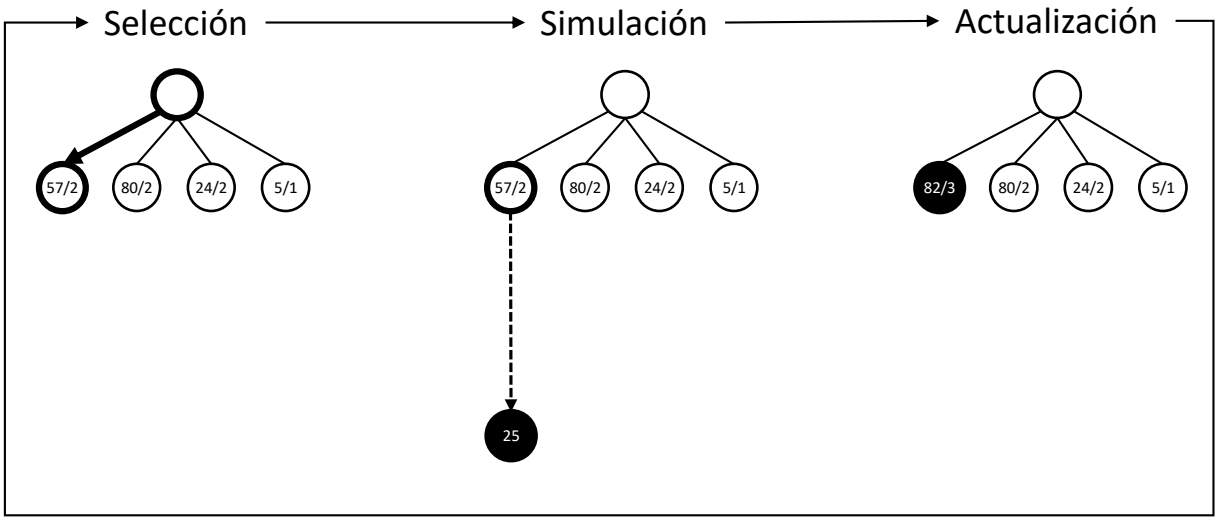


Figura 4.2: Ejemplo de una iteración del Simulation Tree Agent

- El estado correspondiente.
- La acción para llegar a él desde su padre.
- El nodo del padre
- Los nodos de los hijos
- Dos valores numéricos $v$ y $n$ que al igual que en el STA representan el valor de todas las simulaciones que pasan por el estado del nodo y el número de simulaciones que pasan por el nodo, respectivamente.

Una vez inicializado se realizan varias iteraciones del algoritmo hasta que se agota el presupuesto computacional. En nuestro caso ese presupuesto es el número de iteraciones que se van a realizar. Si en menos de ese número de iteraciones se consigue explorar todo el espacio de estados también se para el algoritmo.

Como se muestra en la figura 4.3 en cada iteración del algoritmo tenemos los siguientes pasos:

1. *Selección*: De todos los nodos que son expandibles seleccionamos el mejor según la fórmula *UCB1*. Los nodos expansibles son aquellos que pertenecen a la frontera del árbol de búsqueda en ese momento (en la figura los nodos remarcados) y no son estados terminales del espacio de estados.[1] A la hora de realizar la selección no vamos bajando por niveles sino que comparamos todos los nodos expandibles y elegimos el mejor. El valor del parámetro $C$ se discutirá en la sección de resultados 4.3 porque el valor óptimo no tiene porque ser el valor teórico $\sqrt{2}$ ya que este valor solo es óptimo cuando

[1]En el siguiente paso veremos que no puede haber nodos interiores que sean expansibles

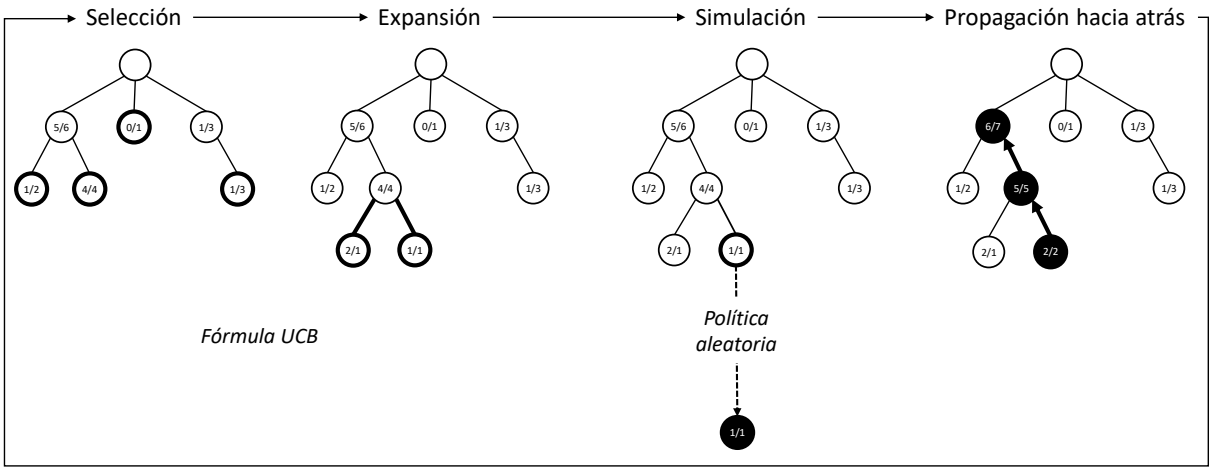


Figura 4.3: Ejemplo de una iteración del MCTS1 Agent

el valor de las recompensas está en el rango [0,1] (sección 2.2.1) mientras que en el Hearthstone el rango depende de la función de evaluación.

2. *Expansión*: Este paso y el anterior son los que más se diferencian de los del algoritmo general. Primero expandimos todos los hijos del nodo seleccionado, los evaluamos y los añadimos a la lista de nodos expansibles. Esta lista lo que representa es la frontera del árbol de búsqueda que hemos comentado en la *selección*. Después eliminamos de la lista el nodo seleccionado (pues ya hemos expandido todos sus hijos así que ya no es expandible) y devolvemos un hijo aleatorio que es desde donde empezará la simulación.

3. *Simulación*: Es exactamente igual que en el algoritmo MCTS teórico. La política para elegir la siguiente acción es aleatoria, el resultado que se produce es la evaluación del estado cuando se acaba el turno y sabemos que hemos llegado a un estado terminal cuando hemos aplicado la acción fin del turno.

4. *Propagación hacia atrás*: También es igual que en el algoritmo MCTS teórico. Se actualizan todos los antecesores del nodo a partir del cuál se ha simulado con la puntuación obtenida en la simulación a excepción de la raíz.

Una de las ventajas respecto al STA es que, aunque se hace un gasto mayor de memoria, se almacena más información de como es el espacio de estados del turno sin limitarnos únicamente a almacenar información del primer nivel. El uso de la fórmula *UCB1* permite equilibrar la explotación del mejor nodo con la exploración de los nodos menos visitados mientras que en el STA dependía de un parámetro fijo.

Como contrapartida, al tener en cuenta toda la frontera cada vez que hacemos la selección y esta crecer de manera exponencial, en ocasiones nos encontramos con fronteras muy grandes en las que se tarda demasiado en ordenar todos los nodos para saber cuál seleccionar. Otro inconveniente es que a la hora de expandir, añadimos todos los hijos al árbol

pero solo simulamos desde uno de ellos. Esto implica que se añaden nodos a la frontera sin haber sido simulados y que puede que no sean simulados en ningún momento de la ejecución. Además, el hecho de realizar más expansiones que simulaciones hace que tanto las visitas como la recompensa acumulada de un nodo sea menor que la suma de las de sus hijos. El agente descrito en la siguiente sección intenta paliar todos estos problemas.

### 4.1.4. MCTS2 Agent

Este agente lo programamos con intención de mejorar el *MCTS1 Agent* anterior y ser fieles a la estructura del algoritmo *UCT* que hemos visto en la sección 2.2.2. Los nodos del árbol guardan la misma información que los nodos del árbol que construye el agente anterior.

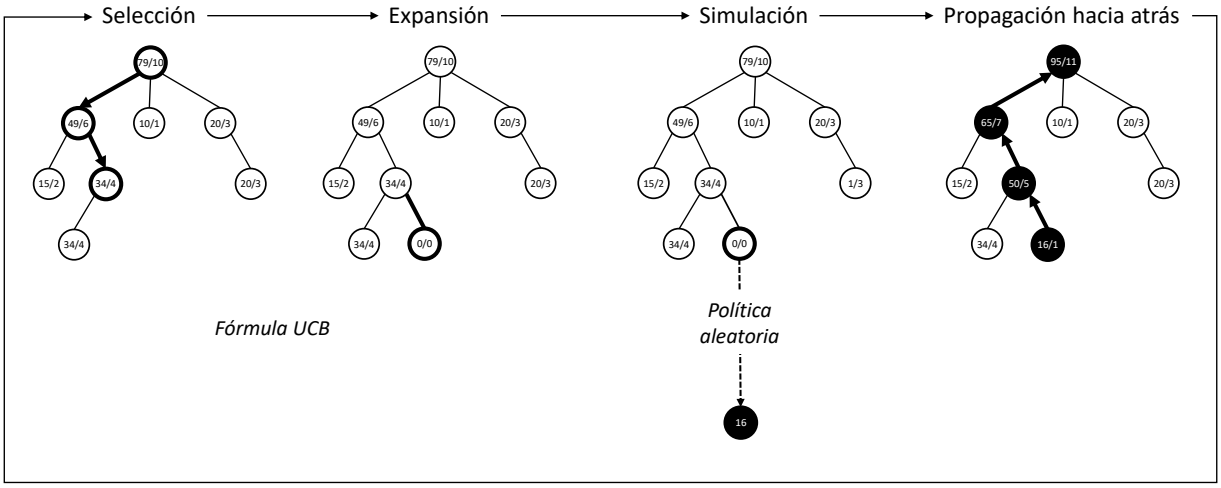


Figura 4.4: Ejemplo de una iteración del MCTS Agent 2

En la figura 4.4 se muestra la estructura de una de las iteraciones de búsqueda que es la siguiente:

1. *Selección*: Empezando desde la raíz del árbol de búsqueda si un nodo no ha sido totalmente expandido seleccionamos ese nodo, en caso de que ya hubiera sido totalmente expandido, aplicamos la fórmula *UCB* para elegir que hijo será el nodo seleccionado. Si este nodo también ya hubiera sido totalmente expandido volveríamos a aplicar la fórmula *UCB* y así sucesivamente. En la figura vemos como se va bajando por el árbol hasta llegar a un nodo no totalmente expandido.

   En un caso límite podríamos llegar a elegir un nodo hoja, este nodo está totalmente expandido pero no tiene hijos. Como es un nodo terminal, la función de evaluación de estado nos puede proporcionar una recompensa así que evaluamos el estado y actualizamos el árbol saltándonos los pasos de expansión y simulación en este caso. La fórmula *UCB1* podría llevarnos varias veces al mismo nodo pero como en cada

iteración se estaría actualizando el árbol, se fomentaría la exploración de todos los nodos que son hermanos de un nodo del camino que conecta la raíz con el nodo hoja.

2. *Expansión*: En la etapa de selección nos hemos asegurado de que el nodo seleccionado sea expansible. En este momento, elegimos aleatoriamente uno de los hijos que no forme parte del árbol y lo añadimos a este. Estos hijos representan acciones sobre el estado que no han sido exploradas aún.

   En este paso también estamos implícitamente aplicando la fórmula *UCB1* como política de selección porque los nodos no explorados tienen el número de visitas a cero y por tanto el valor de la fórmula aplicada al nodo es infinito. Ante el empate de todos los posibles nodos nuevos decidimos romperlo aleatoriamente como en [28].

3. *Simulación*: Empezando desde el estado del nodo añadido en el paso anterior elegimos una acción posible aleatoriamente y recursivamente hacemos lo mismo sobre el estado obtenido hasta llegar a un estado terminal. Cuando llegamos a este estado terminal llamamos a la función de evaluación de estado y devolvemos el resultado obtenido. En caso de que el nodo añadido corresponda a un estado terminal simplemente evaluamos ese nodo y devolvemos el resultado obtenido.

4. *Propagación hacia atrás*: Al igual que en el modelo teórico y en el *MCTS1 Agent*, actualizamos todos los nodos del camino que hay desde la raíz hasta el nodo expandido en esta iteración (ambos inclusive) añadiendo una visita y el resultado obtenido en esa simulación. La actualización se realiza de abajo a arriba y de forma recursiva llamando el hijo a la actualización del padre.

Este agente es más rápido a la hora de seleccionar el nodo en cada iteración que el *MCTS1 Agent* porque utiliza una estrategia voraz para ello a diferencia del *MCTS1 Agent* que tiene que buscar directamente en toda la frontera. Al seguir el esquema del algoritmo *UCT* cada subárbol siempre es un árbol de búsqueda de Monte Carlo.

## 4.2. Función de evaluación del estado

La capacidad de evaluar cuán favorable es un estado del juego para cada uno de los jugadores resulta fundamental. Sin ella sería imposible el saber que acción es mejor y darle mayor prioridad. Sin embargo, *Hearthstone* es un juego complejo en el que hay muchas variables en juego y no todas son observables. También resulta esencial a la hora de construir los árboles de búsqueda de Monte Carlo que hemos descrito en la sección anterior. Esto se debe a que, dada la complejidad del juego y la dificultad de estimar las acciones del adversario, sólo se están explorando las acciones disponibles para cada turno. Por ello es necesario ser capaz de comparar los estados entre sí para establecer cuál es más favorable para cada jugador.

A la hora de evaluar un estado nos encontramos con dos problemas principales. El primero es que puede haber situaciones en las cuales las variables observables del estado del juego no

aporten la suficiente información como para saber como de favorable es un estado para cada uno de los jugadores. Las cartas que tiene el oponente en la mano y la capacidad del jugador de responder a ellas son aspectos imposibles de averiguar así que tenemos que centrarnos en otros.

El segundo problema es que no todos los mazos se juegan igual. Como hemos visto en la sección 3.2, mientras que unos buscan establecer una fuerte ventaja sobre el oponente durante los primeros turnos otros pueden estar enfocados a sobrevivir durante el suficiente tiempo para poder reunir los recursos necesarios para poder jugar sus cartas más poderosas en los turnos posteriores. Por ello, resulta complicado establecer una función de evaluación de estado que indique como de bueno es ese estado dado que la importancia de cada una de las variables observables dependerá de que tipo de mazo se este jugando. No obstante, y recordando lo dicho en la sección 3.2 de forma general podemos definir tres tipos de mazo mayoritarios cada uno con su estrategia: *aggro*, *control* y *midrange*.

### 4.2.1. Funciones de evaluación predefinidas

Para poder adaptarse a los diferentes tipos de estrategias, el simulador dispone por lo tanto de 3 clases Score distintas por defecto que le permiten adaptarse a ellas, veamos cada una en detalle.

#### AggroScore

Orientado a los mazos *aggro* este Score valora mucho la diferencia entre la vida del jugador y la del oponente por lo que se da mayor puntación aquellos estados que hacen disminuir la vida del rival. Además se tiene en cuenta el ataque de las criaturas que se controla ya que son puntos de daño que se pueden hacer en el futuro. También se tiene en cuenta si el oponente controla a una criatura con la habilidad Provocar (o Taunt) ya que esta habilidad hace obligatorio atacarla a ella .

#### ControlScore

En esta estrategia se prioriza el controlar el juego, es decir, reducir los recursos del enemigo y cortar su estrategia. Al ser una estrategia más reactiva resulta más difícil el estimar la idoneidad de un estado. En este caso a lo que más prioridad se le da es a la diferencia entre el número de criaturas controladas por el jugador y las controladas por el rival. También se cuenta el ataque de las criaturas que controla el jugador, para dar prioridad a las criaturas más fuertes.

#### MidrangeScore

Buscando un término medio entre las dos estrategias anteriores, en esta función de evaluación se le da importancia tanto a la diferencia de vida entre un jugador y su oponente como a la diferencia entre el número de esbirros que controla cada uno. Además también se

compara el ataque y la resistencia de estos de forma que se premien estados en los que los esbirros de un jugador son más poderosos que los de su adversario.

### 4.2.2. UtilityScore

Estas tres funciones de evaluación vistas son bastante superficiales y en el momento en el que un mazo sea más complejo y se salga del esquema general comenzarán a fallar ya que hay muchos factores que no se tienen en cuenta. Por ejemplo, a veces puede ser mejor no jugar una carta en un momento y guardársela para mas tarde para poder tener más maneras de responder a una jugada del oponente. Por esto puede ser necesario tener en cuenta el número de cartas en la mano de cada jugador, variable que las funciones de evaluación anterior no tenían en cuenta. En [26] se propone una función de evaluación alternativa e independiente de la estrategia del mazo que adaptamos para reducir la importancia que se le atribuía a los esbirros. Esta función crea una puntuación de la siguiente forma:

- La victoria otorga la máxima puntación
- La derrota otorga la mínima
- Se hace la raíz cuadrada de la vida restante (ya que la vida es más importante cuanto más avanzada esté la partida) y se multiplica por dos.
- El número de cartas en mano se multiplica por 3 para las tres primeras y por 2 para las siguientes
- Se hace la raíz cuadrada del número de cartas restantes en el mazo menos el daño de fatiga, esto es el daño que se recibe cuando no quedan cartas en el mazo.
- Si el campo del oponente está vacío se suma $2 + min(t, 10)$ con $t$ siendo el número de turno.
- Los esbirros suman su ataque y vida.

Con esto se calcula la puntuación desde el punto de vista del propio jugador y del oponente y el resultado final se corresponde a la diferencia entre uno y otro.

### 4.2.3. Aproximación genética

Cada uno de los agentes anteriores establece una evaluación de los estados evaluando las variables que consideran importantes para ese estilo de juego o en general, les asignan un peso y las suman obteniendo como resultado final como de favorable es ese estado. Sin embargo, estas funciones dependen de cierto conocimiento experto para establecer de forma correcta el peso de cada variable o si se debe estudiar en primer lugar. Una alternativa a esto es calcular los valores de esos pesos mediante el uso de algún mecanismo de aprendizaje.

Para ello decidimos crear una nueva función de evaluación en la que se recibiese como parámetro los pesos que se le da a cada una de las variables observables. Estos pesos toman valores de 0 a 1 y multiplican a las siguientes variables, correspondiéndose las 4 primeras a aspectos generales del juego y las siguientes a información específica de cada criatura y se tienen en cuenta para cada una de ellas:

- La raíz cuadrada de los puntos de vida.
- El número de cartas que tiene el jugador en la mano.
- El número de cartas que quedan en el mazo y solo se comprueba cuando la partida ha pasado del décimo turno ya que antes no es un factor muy importante.
- Un valor que es 1 cuando el oponente no tiene criaturas en el campo y 0 en caso contrario.
- La vida de cada criatura controlada.
- El ataque de cada criatura, sólo se comprueba si la criatura puede atacar.
- Un valor que es 1 si la criatura tiene la habilidad de provocar, es decir, que debe ser objetivo obligatorio de los ataques del enemigo. Este peso se multiplica además por cada punto de vida que tenga la criatura ya que cuantos más tenga más daño podrá absorber antes de ser destruida.
- Un valor que es 1 si la criatura tiene último aliento, es decir a que se active un efecto al morir esta. Vale 0 en otro caso.
- Un valor que es 1 si la criatura tiene escudo divino, es decir que pueda recibir daño una primera vez sin que le baje la vida.Vale 0 en otro caso.
- Un valor que es 1 si la criatura tiene cargar, esto es, que pueda atacar la misma ronda que entra en juego. Vale 0 en otro caso.
- Un valor que es 1 si la criatura tiene grito de batalla lo que significa que cuando entra en juego genera un efecto. Vale 0 en otro caso.
- Un valor que es 1 si la criatura tiene viento furioso, es decir, que pueda atacar dos veces. Vale 0 en otro caso.

Estos pesos serán los genes de cada uno de los individuos de la población sobre la que apliquemos el algoritmo genético ya que queremos hallar los valores de estos que nos supongan que evaluamos correctamente el estado de juego y por lo tanto nos lleven a un mayor índice de victorias.

De esta forma podemos empezar a construir un algoritmo genético siguiendo las indicaciones dadas en la sección 2.3. Como ya hemos dicho, los cromosomas de cada uno de los

individuos estarán formados por cada uno de los pesos que se le da a las variables observables de cada estado, que serán los genes.

Otro aspecto importante es como vamos a comprobar la optimalidad de cada individuo, es decir, cuál va a ser la función de *fitness* de este algoritmo genético. Una mejor evaluación del estado debería suponer un mayor índice de victorias en partidas ya que el agente tendrá un conocimiento de mayor calidad sobre que estado es más favorable para él y menos para su oponente. Por ello decidimos que la mejor forma de determinar el fitness era jugar un número de partidas determinado (que fijamos en 200, 100 empezando un jugador y 100 empezando otro). Uno de los jugadores utilizara la función de evaluación dada por el uso de los pesos en los genes del individuo y la otra utilizará la función `MidrangeScore` que viene por defecto en el simulador. El mazo de los jugadores esta fijado siendo el mazo que hemos comentado antes que recibe el nombre de *Midrange Jade Shaman* y cuyas cartas se pueden encontrar en [7]. Además los bots que escogen las acciones son similares solo distinguiéndose en la función de evaluación de estados. En ambos casos los bots siguen una estrategia voraz escogiendo en todas las situaciones la jugada que lleva a un estado cuya función de evaluación le dice que es el mejor. El valor de *fitness* vendrá dado por el número de partidas que gane el jugador que utiliza los pesos del cromosoma partido por el total de partidas jugadas.

Una vez tenemos la función de *fitness* debemos implementar los operadores genéticos de selección, cruce y mutación. Se implementarán todos los métodos de selección, cruce y mutación vistos en la sección 2.3 para poder comparar los resultados obtenidos y establecer cual es más apropiado en este caso para cada una de las operaciones. Recordaremos su funcionamiento, visto en la sección 2.3, junto con algunos detalles de su implementación.

### Métodos de selección

Estos son los métodos que nos permiten elegir a los padres de la siguiente generación. Nos interesa darle prioridad a los mejores individuos pero al mismo tiempo debemos garantizar que los demás también tienen posibilidades de ser elegidos para que nuestra población no se estanque. Estos métodos son:

- *Ruleta*: Se asigna a cada uno de los individuos una probabilidad en función de su fitness de forma que los que sean mejores tengan más posibilidad de ser elegidos y además esta posibilidad sea proporcional a su *fitness*, por ejemplo, si el *fitness* de un individuo es el doble que el de otro deberá tener el doble de posibilidad de ser elegido.

- *Rango*: Se ordenan los individuos en función de su *fitness* y se le asigna una puntación $n$ al mejor, siendo $n$ el número de individuos totales en la población, $n-1$ al siguiente y así hasta llegar al último que debe tener una puntuación de 1. Con esta nueva puntuación se emplea el método de la ruleta de forma que en este caso lo único que importa es cuál es mejor que otro, no en cuanta cantidad lo es.

- *Torneo*: Dado un $k$ menor que $n$ siendo $n$ el tamaño total de la población se extrae una subpoblación de tamaño $k$ (admitiendo repetidos) y se selecciona el que tiene una mejor puntuación de *fitness* de entre ellos.

**Métodos de cruce**

Son los métodos que generan nuevos individuos a partir de los seleccionados. Nos interesa mezclar los genes de los individuos padre para obtener otros nuevos que puedan ser mejores pero también puede ser que introduzcamos pocos cambios y que la población se estanque. Los métodos de cruce usados son:

- *Corte*: Dado un número $k$ se emplea el método de cortes de $k$ puntos.
- *Uniforme*: Para cada uno de los genes se selecciona de forma aleatoria de que padre se escoge.
- *Combinación*: Generamos un $\alpha \in [0, 1]$ manera aleatoria de forma que los genes de los hijos están acotados por los de los padres.

**Métodos de mutación**

Los métodos de mutación son una de las principales formas que tenemos de introducir diversidad en la población y de evitar quedarnos en un óptimo local de la función de *fitness*. Vamos a usar tres aunque dos de ellos son una variación del mismo:

- *Total con cota*: Esta es una variación del método de mutación de *Creep* en la que para cada gen se decide de forma aleatoria e independiente si muta o no dada una probabilidad. Como es posible que en este caso muten muchos de los genes se introduce una cota de $[-0.1, 0.1]$ para no perder demasiada información.
- *Aleatorio no acotado*: También es una variación del método de mutación de creep. En este método se escoge un gen de forma aleatoria y se le otorga un nuevo valor también aleatorio.
- *Scramble*: Dado un tamaño se extrae un subconjunto de genes del cromosoma con ese tamaño, se mezcla y se vuelve a meter en el cromosoma en es nuevo orden.

Además de todo esto, hicimos que el algoritmo tuviese cierto grado de elitismo, es decir, que los $n$-mejores individuos de una generación pasasen directamente a la siguiente. Para poder introducir mayor diversidad a la población decidimos también que en cada generación se introducirían un número determinado de nuevos individuos generados de forma aleatoria.

## 4.3. Experimentos y resultados

En esta sección presentaremos los experimentos que hemos realizado con los agentes y el algoritmo evolutivo para buscar una buena función de evaluación de estado que hemos programado. Esta sección se divide en tres subsecciones: la primera de ella consistirá en los experimentos sobre los agentes, la segunda en los experimentos del algoritmo evolutivo y la tercera en los experimentos con el agente resultante de las dos secciones anteriores.

Como comentamos al comienzo de este capítulo, el mazo utilizado para todos los experimentos será el *MidRangeShaman* que viene dentro del simulador.

### 4.3.1. Experimentos y resultados de los agentes

Antes de comenzar con los experimentos vemos que cada uno de los agentes depende de 3 parámetros:

1. *Presupuesto computacional*: Es el tiempo que tiene el agente para decidir cuál va ser la siguiente acción que va a tomar en su turno. El rango de valores que consideraremos será: [1,1000] ms. A la hora de discretizar este intervalo continuo, tomaremos únicamente las potencias de 10.

2. *Constante de exploración*: Como los tres agentes (*SimTree Agent*, *MCTS1 Agent* y *MCTS2 Agent*) están inspirados en mayor o menor medida en los árboles de búsqueda de Monte Carlo esta constante permite encontrar el equilibrio entre explotación y exploración. En el caso del *SimTree Agent* este parámetro es EXPLORATION_RATE y toma valores entre [0,1] así que para dicretizarlo tomamos todos los valores del intervalo con un paso de 0,05.

   Tanto en el *MCTS1 Agent* como en el *MCTS2 Agent*, la constante de exploración es la C. Hemos comentado antes que el valor teórico era $\sqrt{2}$ cuando el valor de las recompensas estaba entre [0,1]. Como en nuestro caso las recompensas no se encuentran en ese rango consideraremos los valores de C en el intervalo [1,3] y para discretizarlo tomaremos los valores con un paso de 0,1.

3. *Función de evaluación*: A la hora de evaluar el estado, con el fin de no mezclar los resultados con las funciones obtenidas del algoritmo genético, solo consideraremos las funciones `MidrangeScore` y `UtilityScore`.

Para poder estimar cuales son los mejores valores de las constantes de exploración de cada agente enfrentaremos a los agentes con cada uno de los valores discretos contra el *Greedy Agent* con la función `MidrangeScore` que viene en el simulador. Los enfrentamientos consistirán en 100 partidas (50 empezando un agente y 50 empezando el otro) para intentar evitar en gran parte el factor aleatorio del juego y los agentes contarán con un presupuesto computacional que fijaremos a 100 ms para seleccionar la siguiente acción.

Los resultados de estos experimentos se muestran en los gráficos 4.5 y 4.6. La información de estos gráficos se puede ver en las tablas C.1, C.2 y C.3. En ellos se puede ver que el *SimTree Agent* y el *MCTS2 Agent* obtienen mejores resultados con la función de evaluación `UtilityScore` siendo bastante evidente en el caso del *MCTS2 Agent*. Por su parte, el *MCTS1 Agent* obtiene resultados muy similares con ambas funciones de evaluación. Un posible motivo de que haya diferencias en los resultados obtenidos por parte del *SimTree* y el *MCTS2* con respecto al *MCTS1* es que los dos primeros únicamente utilizan la función para evaluar estados terminales y el *MCTS1* también evalúa estados intermedios.

Una vez hemos realizado estos primeros experimentos para calibrar las constantes de exploración, seleccionamos los 2 valores de estas constantes teniendo en cuenta los valores similares y no escogiendo dos valores que estén muy próximos entre sí. El resumen de los 12 agentes elegidos se muestra en la tabla 4.1.

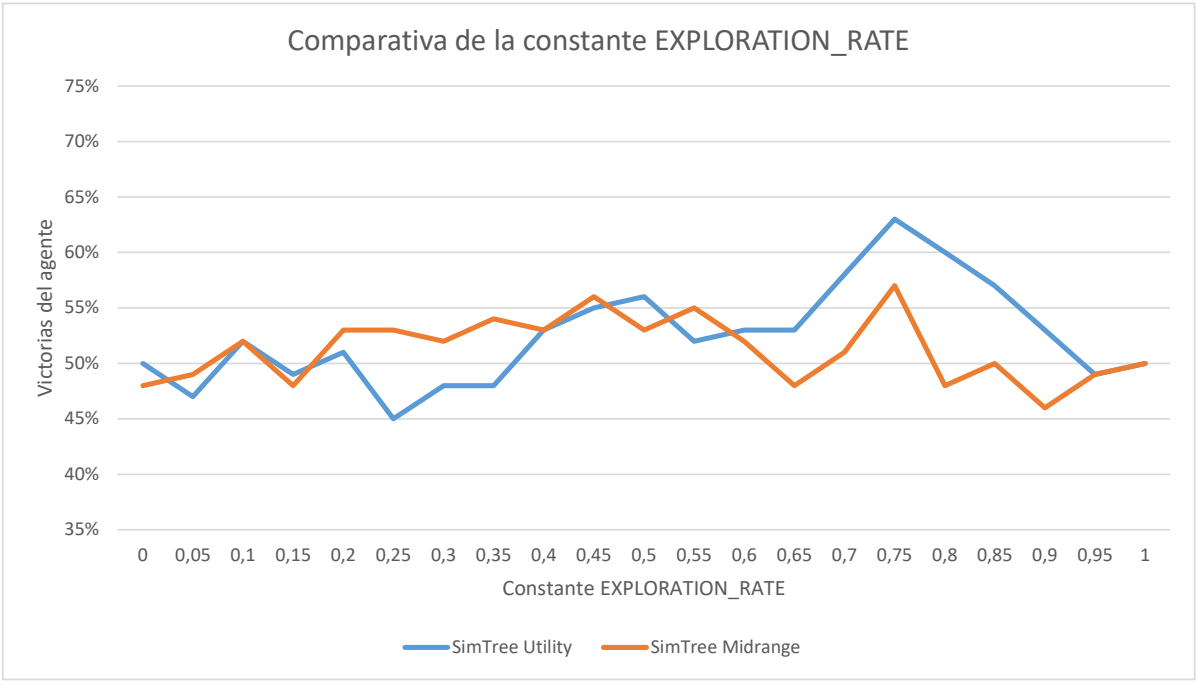


Figura 4.5: *SimTree Agent* vs *Greedy Agent* con diferentes valores de la constante EXPLORATION_RATE

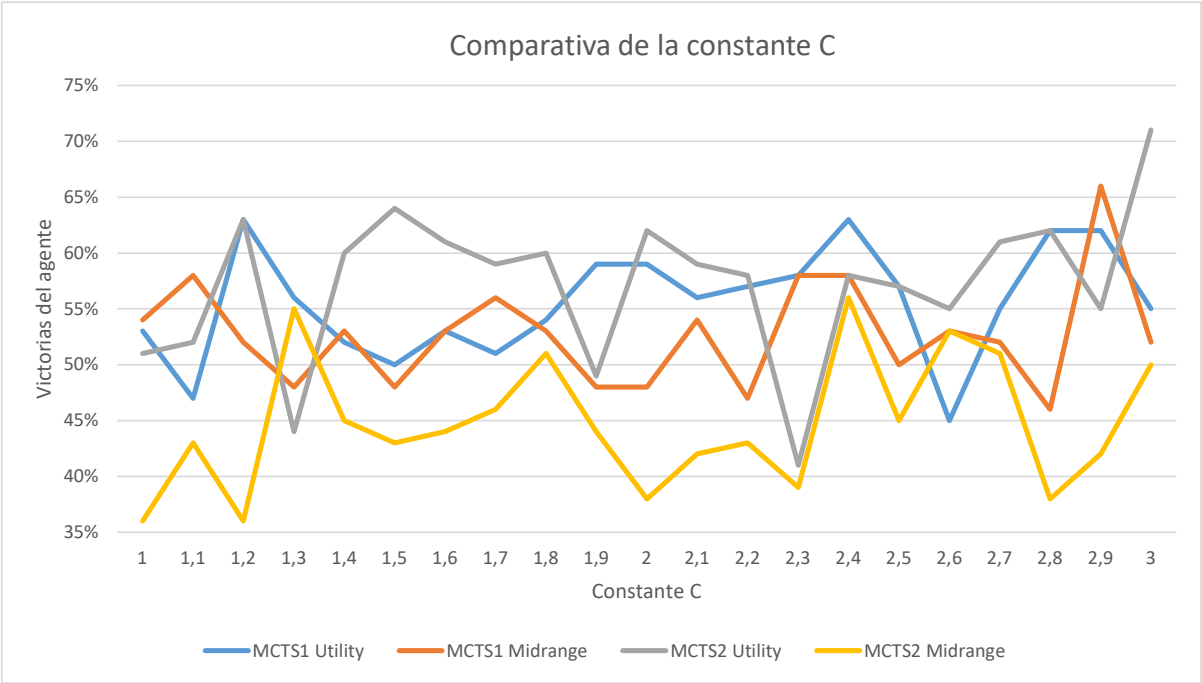


Figura 4.6: *MCTS1 Agent* o *MCTS2 Agent* vs *Greedy Agent* con diferentes valores de la constante C

Cuadro 4.1: Los 12 agentes seleccionados

| Agente | Función de evaluación | Constante 1 | Constante 2 |
|---|---|---|---|
| SimTree | Utility | 0,5 | 0,75 |
| | Midrange | 0,5 | 0,75 |
| MCTS1 | Utility | 2,4 | 2,4 |
| | Midrange | 2,9 | 2,9 |
| MCTS2 | Utility | 1,5 | 3,0 |
| | Midrange | 1,3 | 2,4 |

Al realizar los experimentos anteriores, en la mayoría de los casos el agente obtenía más victorias jugando primero. En las tablas C.4, C.5 y C.6 del anexo C, referentes a los siguientes experimentos, están divididas las victorias de cada agente en función de si empezaba él o empezaba el otro. Para comprobar que efectivamente sucedía esto y que para este mazo y esta estrategia en concreto era ventajoso empezar, enfrentamos 2 agentes iguales (*Greedy Agent*) en un total de 1000 partidas, la probabilidad de ganar siendo el agente que empezaba era de un 57 % frente a un 43. Esto confirma nuestra hipótesis del desequilibrio de este mazo y esta estrategia.

Los siguientes experimentos consistirán en enfrentar a los 12 agentes (tabla 4.1) contra el *Greedy Agent* esta vez variando el parámetro del presupuesto computacional. Los resultados obtenidos se muestran en los gráficos 4.7, 4.8 y 4.9.

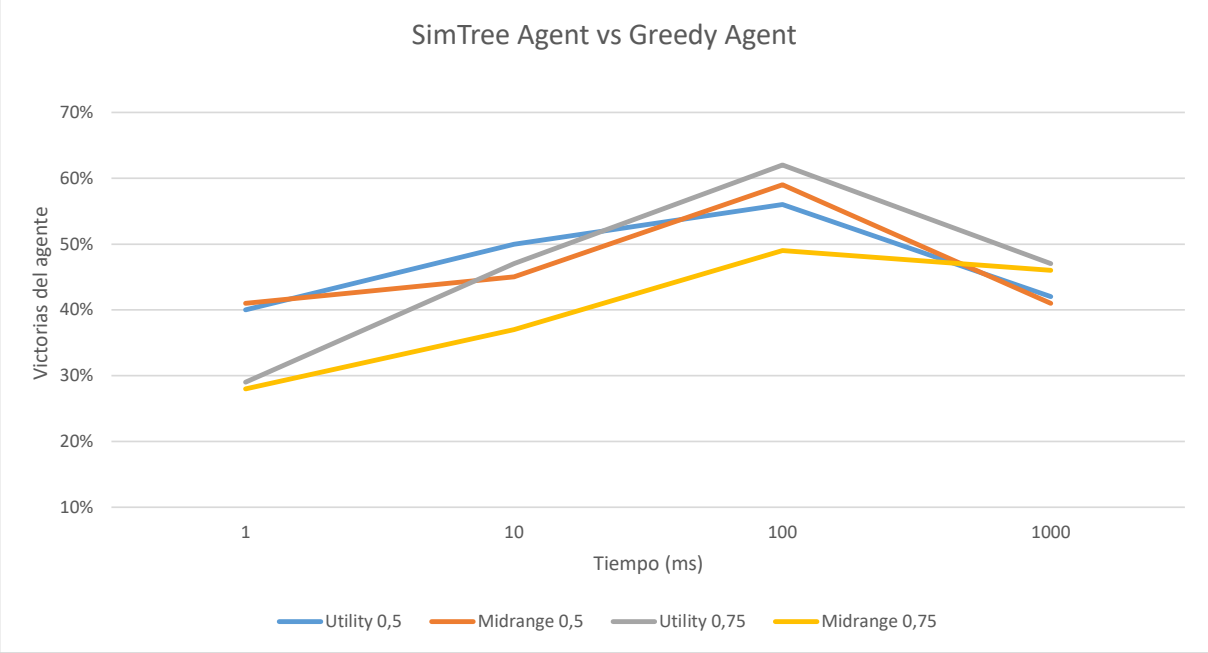


Figura 4.7: *SimTree Agent* vs *Greedy Agent* con diferentes tiempos de decisión

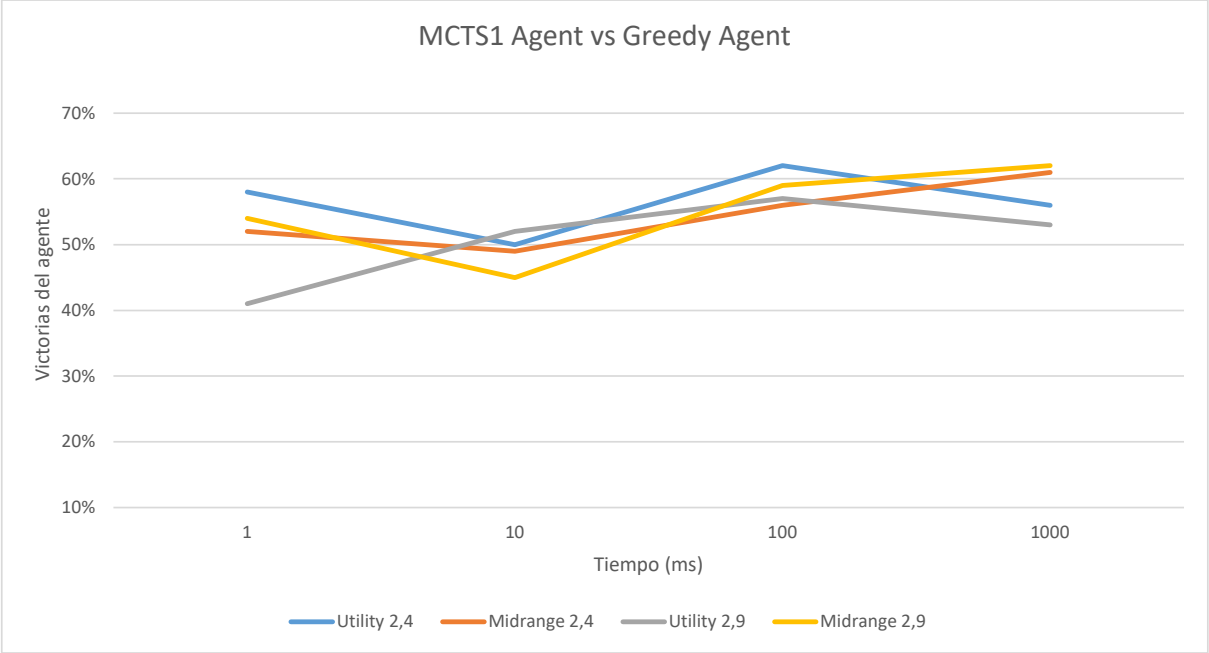


Figura 4.8: *MCTS1 Agent* vs *Greedy Agent* con diferentes tiempos de decisión

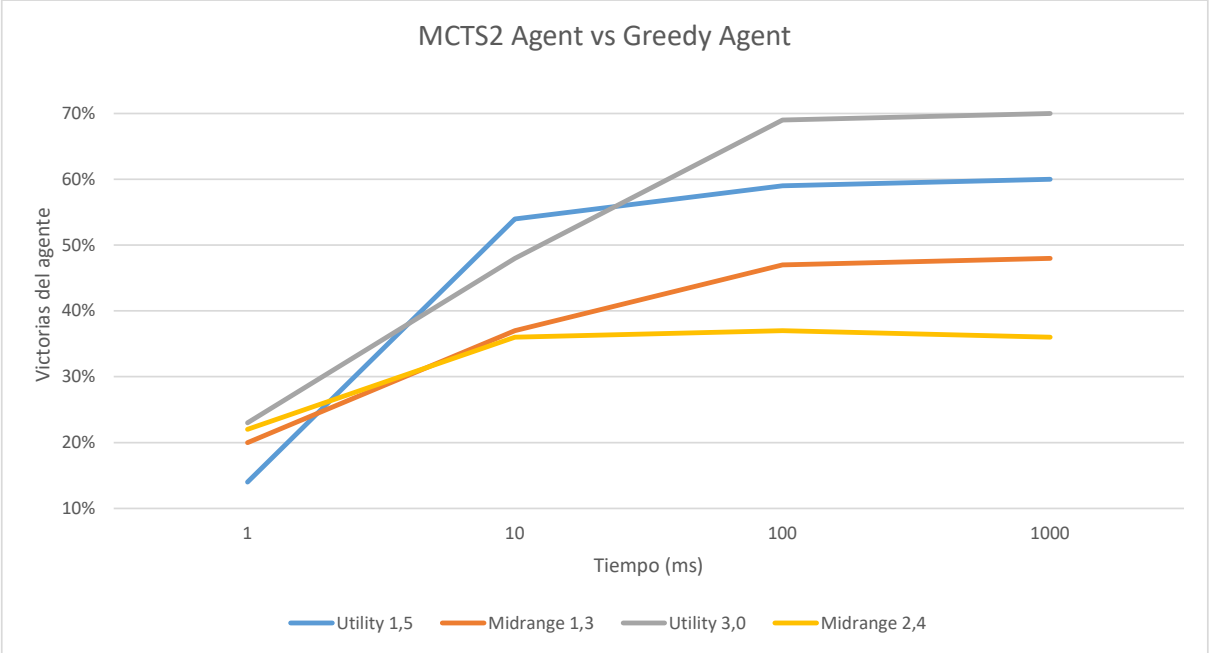


Figura 4.9: *MCTS2 Agent* vs *Greedy Agent* con diferentes tiempos de decisión

De estos gráficos y sus correspondientes tablas C.4, C.5 y C.6 podemos sacar las siguientes conclusiones:

- Efectivamente el jugador que empieza cuando se juega con este mazo tiene una cierta ventaja. Únicamente en 4/48 enfrentamientos ha ganado más partidas el agente cuando no empezaba.
- En el caso del *SimTree Agent* tener más tiempo para elegir la siguiente acción no tiene porque implicar mejores resultados. Como podemos ver en el gráfico 4.8 los resultados son mejores con 100 ms en vez de con 1000 ms. Esto puede ser porque la función de evaluación no sea muy buena e introduzca demasiado ruido lo que provoca que los resultados empeoren.
- Como era razonable pensar los peores resultados se obtienen cuando el tiempo disponible es 1 ms, pero llama especialmente la atención el caso del *MCTS2* que apenas llega a un 20 % de victorias (gráfico 4.9).
- Mientras que el *SimTree Agent* y el *MCTS1 Agent* funcionan de forma parecida con la función de evaluación `MidRangeScore` como con `UtilityScore`, el *MCTS2 Agent* funciona mucho mejor con `UtilityScore`. En los experimentos anteriores teníamos la sensación de que el SimTree Agent funcionaba un poco mejor con el `UtilityScore` pero en este caso no es tan claro aunque hay una pequeña diferencia.

Hasta este momento, únicamente hemos enfrentado a los diferentes agentes contra el *Greedy Agent*. Para enfrentarlos entre ellos realizaremos 4 torneos, uno para cada presupuesto computacional de los que consideramos. El torneo consistirá en que 7 agentes se enfrentarán en 100 partidas (50 empezando uno y 50 empezando el otro) contra cada uno de los otros agentes, jugando así un total 600 partidas cada agente. La agentes que formarán parte del torneo serán los dos mejores *SimTree Agent*, los dos mejores *MCTS1 Agent*, los dos mejores *MCTS2 Agent* y el *Greedy Agent*. El motivo de que el *Greedy Agent* forme parte del torneo es que al no depender del presupuesto computacional nos servirá para comparar si mejoran o no las prestaciones de los demás agentes al aumentar el presupuesto y en que medida lo hacen. Los agentes seleccionados se muestran en la tabla 4.2.

Cuadro 4.2: Agentes que participan en los torneos de 7

| **Agente** | **Función de evaluación** | **Constante** |
|---|---|---|
| Greedy | Midrange | |
| SimTree | Utility | 0,75 |
| | Midrange | 0,5 |
| MCTS1 | Utility | 2,4 |
| | Midrange | 2,9 |
| MCTS2 | Utility | 1,5 |
| | Utility | 3,0 |

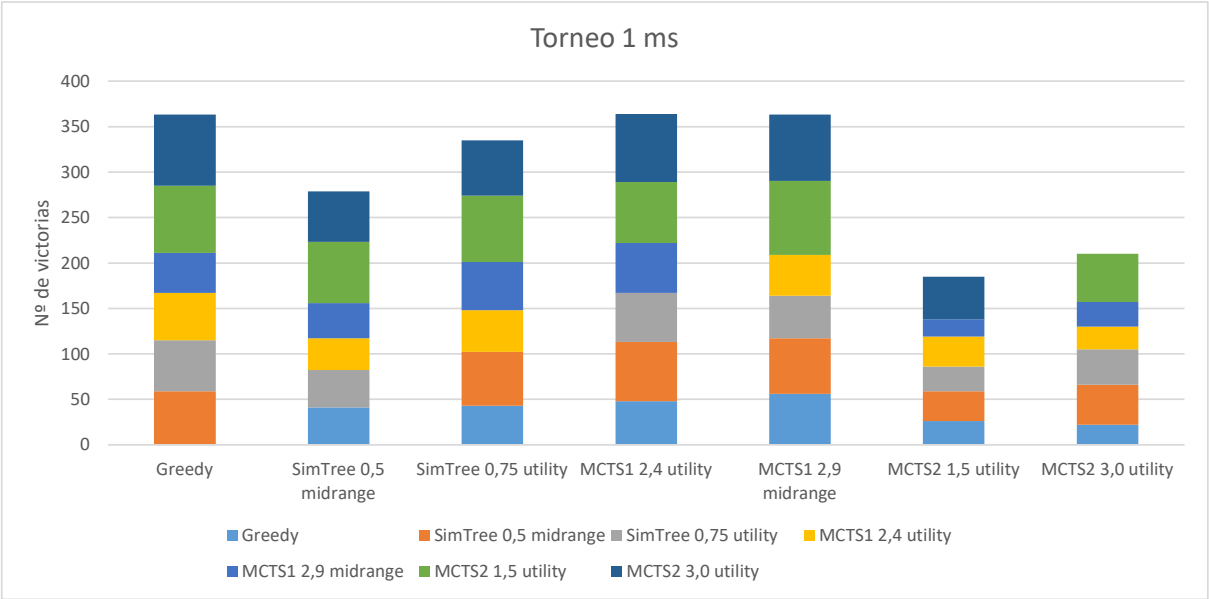


Figura 4.10: Torneo con 1 ms de tiempo de respuesta 100 partidas entre cada agente

Los resultados de estos torneos se presentarán en histogramas en los cuales la longitud total de cada barra representa el número de victorias de cada agente en ese torneo y los diferentes segmentos de cada barra indican cuántas se han conseguido contra cada agente. Adicionalmente añadiremos una tabla para cada torneo donde la posición $(i, j)$ de esta representa el número de victorias del agente $i$ contra el agente $j$. Los resultados de los diferentes torneos se muestran en los gráficos 4.10, 4.11, 4.12 y 4.13 y en las tablas C.7, C.8, C.9 y C.10.

Como podemos ver en la figura 4.10 en el torneo de 1 ms el *Greedy Agent* obtiene un gran número de victorias respecto al resto de agentes excepto al *MCTS1 Agent* que obtiene resultados similares. Esto puede ser porque el *MCTS1 Agent* expande muchos más nodos que el *MCTS2 Agent* aunque no simule todos.

En la figura 4.11, correspondiente al torneo de 10 ms, vemos que al tener más tiempo para elegir la jugada, los dos *MCTS2 Agent* mejoran en gran medida sus resultados. Comparando esta figura con la figura 4.10 vemos que el segundo *SimTree Agent* mejora y el segundo *MCTS1 Agent* empeora. Esto no significa que juegue peor el *MCTS1* ya que sus resultados contra el *Greedy Agent* son similares pero como el resto de agentes juegan mejor sus resultados empeoran.

Cuando el torneo ya es de 100 ms por acción (figura 4.12) vemos que el segundo *MCTS2 Agent* es el que mejores resultados obtiene y como todos los agentes menos el primer *SimTree Agent* mejoran su juego, lo que se refleja en el gran descenso de victorias del *Greedy Agent*.

En el torneo de 1000 ms (figura 4.13) los resultados son similares al torneo anterior. Como el número de victorias del *Greedy Agent* es el mismo que en el torneo anterior podemos decir que los agentes realmente no mejoran su juego al aumentar el presupuesto computacional hasta tener 1000 ms por acción.

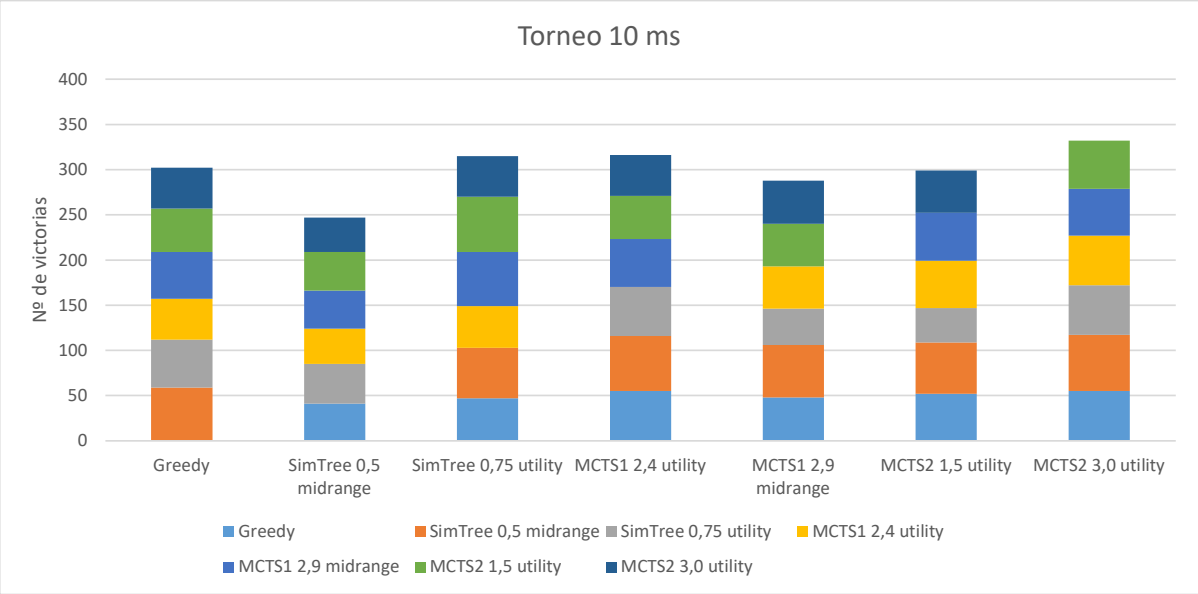


Figura 4.11: Torneo con 10 ms de tiempo de respuesta 100 partidas entre cada agente

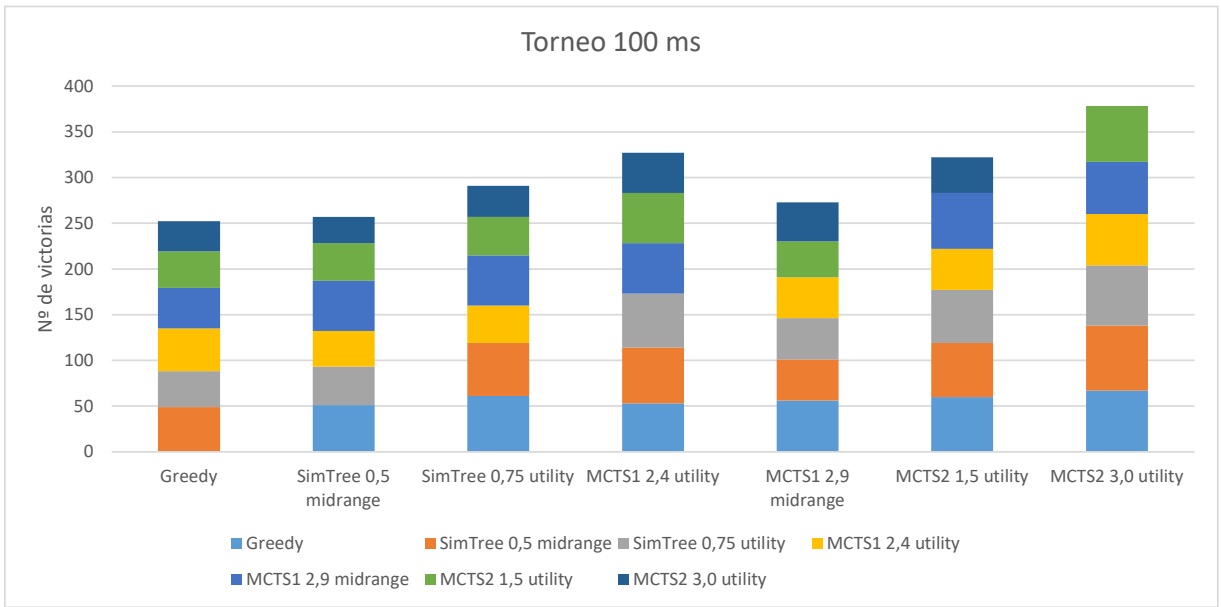


Figura 4.12: Torneo con 100 ms de tiempo de respuesta 100 partidas entre cada agente

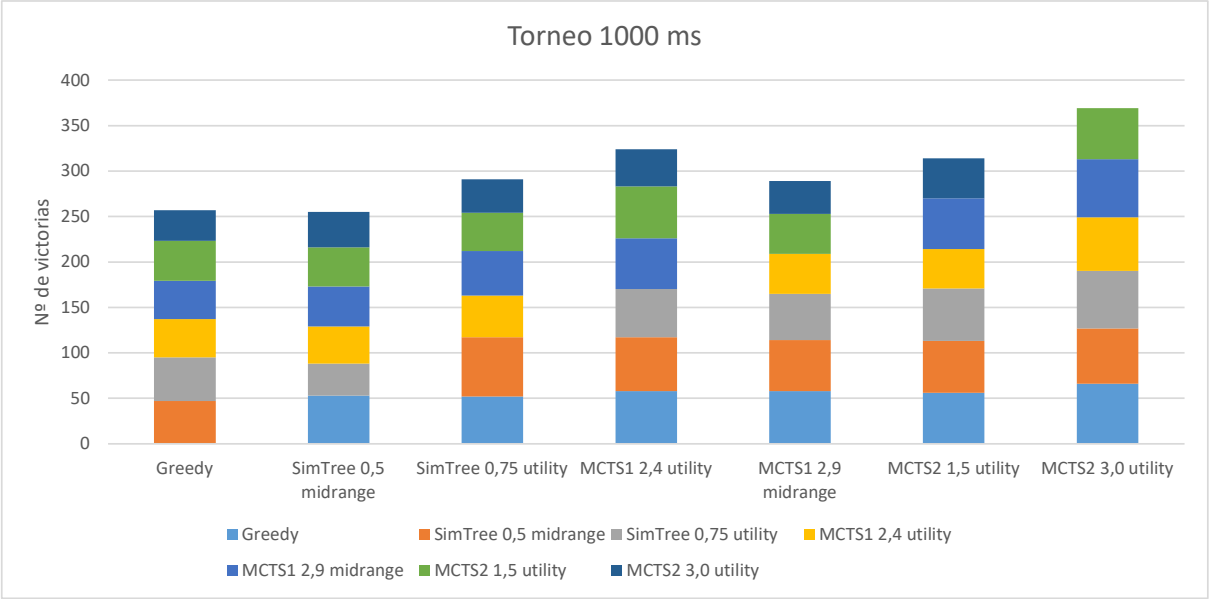


Figura 4.13: Torneo con 1000 ms de tiempo de respuesta 100 partidas entre cada agente

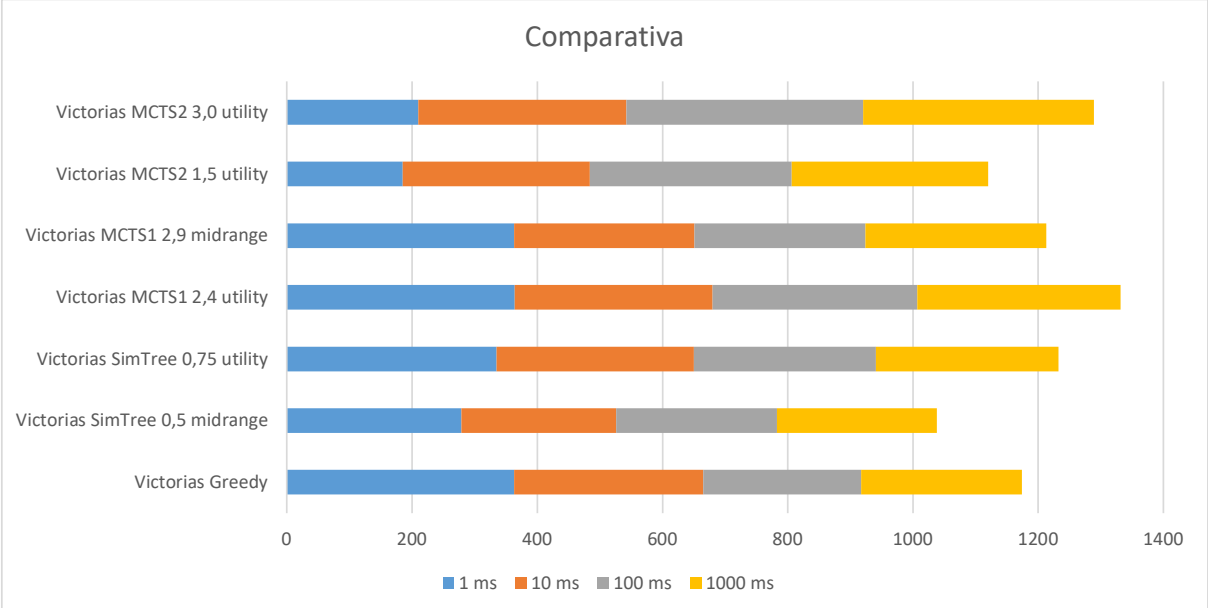


Figura 4.14: Comparativa del número de victorias de cada agente en todos los torneos

Para terminar en esta sección, podemos ver una comparativa del número de victorias totales de cada uno de los agentes a través de los diferentes torneos (figura 4.14). En esta comparativa podemos ver que el agente que más victorias ha obtenido ha sido el *MCTS1 Agent* con constante $C = 2,4$ y la función de evaluación `UtilityScore` seguido del *MCTS2 Agent* con constante $C = 3,0$ y la función de evaluación `UtilityScore`.

La principal diferencia es debida al torneo de 1 ms, en el que el *MCTS1 Agent* es claramente superior. En contextos con más presupuesto computacional, como la competición internacional [2] en la que cada turno puede alargarse hasta 60 segundos, el *MCTS2 Agent* con $C = 3,0$ tendrá más opciones de ganar.

### 4.3.2. Experimentos y resultados del algoritmo genético

Como ya hemos visto antes, se van a ejecutar diferentes experimentos sobre el algoritmo genético descrito en la sección 4.2.3. Variando los métodos de selección, cruce y mutación, así como los valores de elitismo o el número de individuos aleatorios nuevos generados en cada generación buscaremos un nuevo conjunto de pesos, dados por los genes de cada individuo, que nos permitan mejorar las funciones de evaluación de las que dispone el simulador por defecto. Recordando lo visto en esa sección para calcular el *fitness* de cada individuo se jugará un número de partidas fijo, entre dos jugadores con un mazo fijo e igual y uno de los jugadores usará como función de evaluación la dada por los genes del individuo y el otro usará la función de evaluación por defecto `MidrangeScore`.

Las variables que se estudiarán serán:

- **Grado de elitismo:** es decir, el número de individuos que pasan directamente a la siguiente generación por tener un *fitness* más alto.
- **Nuevos individuos aleatorios:** en cada generación una parte de la población no vendrá de la generación anterior sino que se producirán de forma aleatoria para favorecer la diversidad.
- **Probabilidad de mutación**
- **Método de selección:** además también se contará la variable dada por el tamaño del torneo en el método de selección por torneo.
- **Método de cruce:** en el caso del cruce por corte, la variable que lo acompaña es el número de puntos de corte considerados.
- **Método de mutación**: uno de los métodos de mutación requiere a su vez una variable. En el caso de la mutación total con cota esta variable es la probabilidad que se le da a cada gen para mutar una vez que se ha decidido que ese individuo debe mutarse. También se podría considerar como una variable el tamaño del subconjunto de genes a barajar en el método de *Scramble* pero por la complejidad del experimento decidimos fijarlo en 4.

Cuadro 4.3: Primera aproximación genética

| Núm. | Indv. Aleat | Prob. Mut. | Selección | Cruce | Mutación | Mejor Ind. |
|---|---|---|---|---|---|---|
| 1 | 65 | 0.35 | Ruleta | Uniforme | AleatorioNoCota | 0.455 |
| 2 | 65 | 0.4 | Torneo(25) | Corte(3) | AleatorioNoCota | 0.48 |
| 3 | 75 | 0.5 | Torneo(20) | Corte(2) | TotalCota(0.25) | 0.475 |
| 4 | 95 | 0.4 | Rango | Combinación | AleatorioNoCota | 0.4 |
| 5 | 95 | 0.5 | Ruleta | Uniforme | TotalCota(0.35) | 0.41 |

### Primera aproximación

En primer lugar y tras hacer varias pruebas decidimos que el tamaño de la población fuese de 200 individuos y que se calculasen 10 generaciones ya que a partir de la décima generación parecía que la población se estabilizaba y no había cambios. En la tabla 4.3 se muestran las variables de cada uno de estos experimentos (exceptuando el elitismo que en en este caso es 5 para todos) junto con el individuo con mejor *fitness* en la última generación La figura 4.15 muestra además la evolución de la puntuación de *fitness* más alta en cada generación.

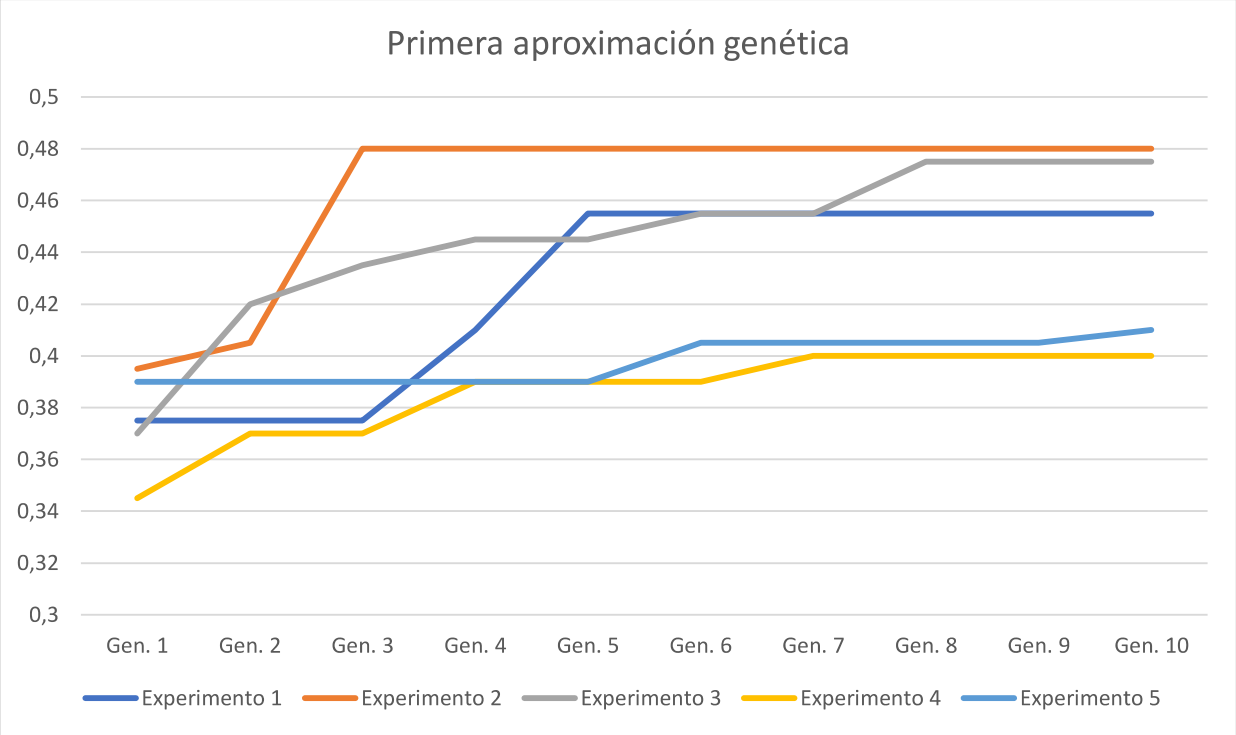


Figura 4.15: Evolución del mejor individuo en cada generación (primera aproximación)

Durante las pruebas notamos que la población tendía a estabilizarse y evolucionar poco. Por ello, utilizamos en algunos casos una probabilidad de mutación tan elevada con la

esperanza de intentar introducir más diversidad y de evitar el estancamiento. Observando tanto la tabla 4.3 como la figura 4.15 podemos ver que los casos en los que la población alcanza un mejor *fitness* y además presenta una mayor variación (el experimento 3) se corresponden con el uso del método de torneo como método de selección. En el resto de los casos la población solo mejora en dos o tres generaciones, mientras en el resto se estabiliza dando como resultado un valor de *fitness* constante.

Otro aspecto preocupante de esta aproximación es que ninguno de los experimentos lograban ganar más partidas de las que perdían, es decir, tener un *fitness* mayor que 0.5. El *fitness* parecía converger entre 0.4 y 0.5.

Por estos motivos decidimos implementar una serie de cambios en la función de evaluación del estado y repetir los experimentos con la esperanza de obtener unos resultados mejores.

### Segunda aproximación

En primer lugar, cambiamos el cuarto peso para que, en vez de sumarse simplemente cuando el contrario no tuviese criaturas en el campo, multiplicase a la diferencia entre el número de criaturas controladas por el jugador y las controladas por el oponente. De esta forma se recibe una bonificación si el jugador tiene más ya que se añade un valor positivo y una penalización si el que tiene más es el oponente, ya que el valor a añadir es negativo.

También añadimos otro nuevo peso. Al principio se calculaba la bonificación que otorgaba cada criatura (según su ataque, habilidades...) y se sumaban todas a la puntuación total. Decidimos introducir un nuevo peso que multiplicase la suma de las puntuaciones obtenidas de los esbirros para escalarlas más y que tuviesen el mismo valor (de 0 a 1) que por ejemplo el número de cartas en mano, no más.

Además implementamos el método de mutación por *Scramble* que sería capaz de variar en mayor medida los individuos mutados. También aumentamos el tamaño de la población a 400 individuos, lo cual supuso que el coste en tiempo se duplicase pero consideramos que podría ser una solución para evitar el estancamiento o convergencia temprana de la población, ya que al ser el número de individuos más elevados también sería posible que hubiese una mayor diversidad.

Tras introducir estos cambios hicimos otra serie de experimentos consiguiendo los resultados obtenidos en la tabla 4.4. Como hemos visto anteriormente que el método de selección que mejor funcionaba era el método de corte le hemos dado prioridad y hemos probado con diferentes tamaños. Es importante recordar además que, como hemos doblado el tamaño de la población la proporción de individuos introducidos por elitismo o generados de forma aleatoria es menor.

En las figuras 4.16 podemos observar la evolución del *fitness* del mejor individuo de cada generación. Fijándonos en los experimentos 5 y 7 vemos que son los que menos variación presentan, por lo que nos sirven para descartar el método de selección por ruleta ya que en nuestro experimento parece llevar a una convergencia excesivamente temprana.

Una de las primeras cosas que se puede observar además es como el haber cambiado los pesos nos permite obtener un valor más alto en las funciones de *fitness*. Antes, la puntuación de un individuo generado aleatoriamente era siempre menor que 0.4 mientras que ahora

Cuadro 4.4: Segunda aproximación genética

| Núm. | El. | Ind. Aleat | Prob. Mut. | Selección | Cruce | Mutación | Mejor Ind. |
|---|---|---|---|---|---|---|---|
| 1 | 5 | 55 | 0.35 | Torneo(25) | Combinacion | Scramble | 0.53 |
| 2 | 3 | 77 | 0.4 | Torneo(25) | Corte(3) | AleatorioNoCota | 0.57 |
| 3 | 3 | 77 | 0.35 | Torneo(60) | Uniforme | AleatorioNoCota | 0.555 |
| 4 | 3 | 77 | 0.5 | Rango | Corte(2) | UniformeAcotada(0.25) | 0.51 |
| 5 | 3 | 55 | 0.4 | Ruleta | Combinacion | Scramble | 0.51 |
| 6 | 3 | 67 | 0.2 | Rango | Uniforme | Scramble | 0.54 |
| 7 | 3 | 67 | 0.15 | Ruleta | Corte(4) | UniformeAcotada(0.25) | 0.5 |
| 8 | 3 | 67 | 0.45 | Torneo(50) | Corte(3) | AleatorioNoCota | 0.555 |
| 9 | 5 | 55 | 0.4 | Torneo(30) | Corte(3) | Scramble | 0.545 |
| 10 | 5 | 55 | 0.35 | Torneo(40) | Combinacion | AleatorioNoCota | 0.555 |
| 11 | 5 | 55 | 0.4 | Torneo(50) | Combinacion | Scramble | 0.56 |

suben hasta 0.52 y son valores más altos en general.

Observando los experimentos 6 y 7 se puede apreciar que un valor muy pequeño en la probabilidad de mutación nos lleva a que no se alcancen valores de *fitness* muy elevados y además pueden conducir al estancamiento. Sin embargo parece que el usar uno u otro método de mutación no produce mucha diferencia o al menos no tanta como el de selección. Por ejemplo, el experimento que da lugar a un mejor individuo obtiene un *fitness* de 0.57 usando método de mutación aleatoria sin cota, mientras que el siguiente mejor (el experimento 11) utiliza el método de *Scramble* obteniendo un 0.56. No obstante si que se puede ver que los experimentos que utilizan la mutación aleatoria sin cota parecen acabar todos en valores superiores a 0.55. No obstante al ser la diferencia entre uno y otros tan pequeña, el número de experimentos tan pequeño y la función de *fitness* no determinista es posible que este resultado fuese casual.

Aún más igualados que los métodos de mutación se encuentran los métodos de cruce sobre los cuales no se puede ver una diferencia clara.

Para los experimentos futuros vamos a extraer los pesos de los 3 mejores individuos para poder compararlos con las demás funciones de evaluación. Estos son el mejor individuo del experimento 2, el mejor del 11 y el mejor del 3, en ese orden. Los pesos de estos individuos se pueden observar en la tabla 4.5. Estos pesos se encuentran normalizados entre 0 y 1. Cada uno de ellos multiplica el valor de una variable descrita en la sección 4.2.3, siguiendo ese orden, y con los cambios indicados al principio de esta segunda aproximación. De estos pesos el mayor, o al menos el que coincide en tener un valor elevado en todos los individuos es el que sirve para escalar la vida, lo cual nos indica que es una variable importante a la hora de evaluar el estado. En general y exceptuando el peso 2 (relativo al número de cartas en el mazo) y el 9 (relativo a que una criatura pueda atacar en el mismo turno en que se juega), podemos ver como los valores de los pesos son similares lo cual nos puede indicar que la solución converge.

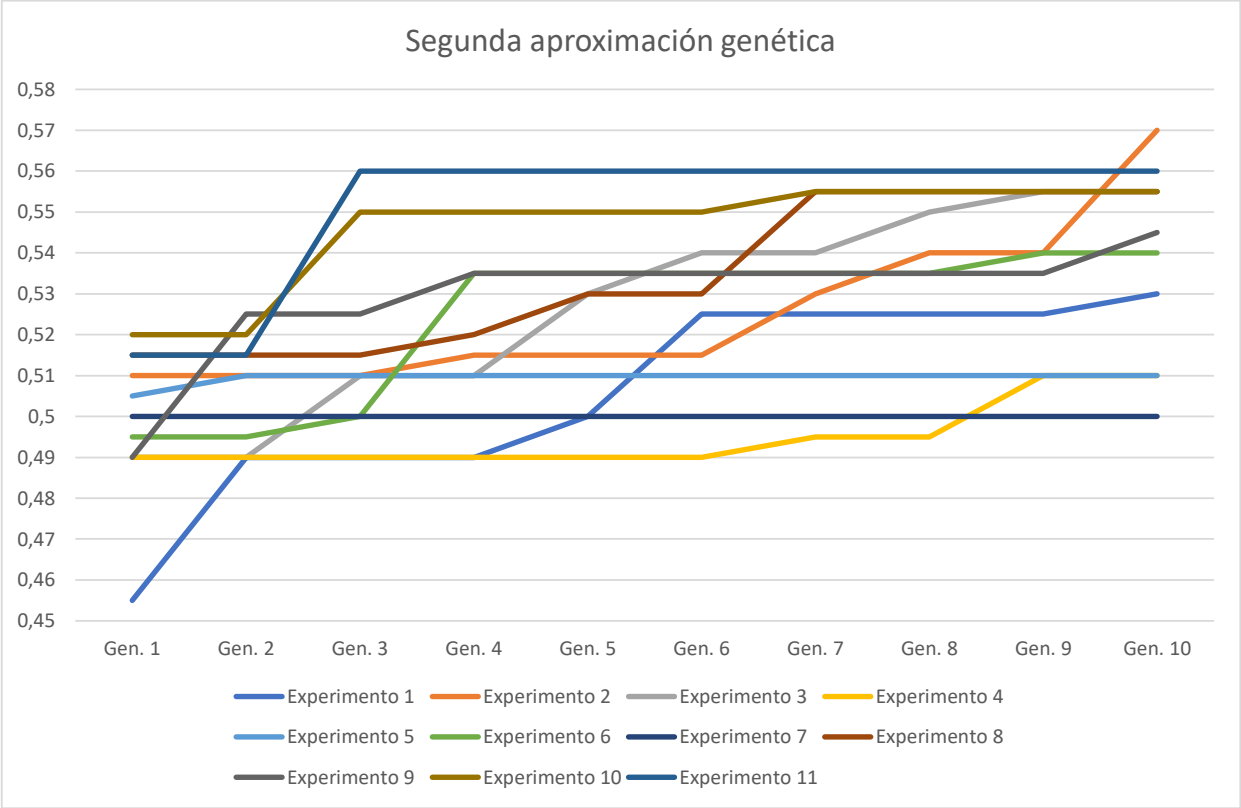


Figura 4.16: Evolución del mejor individuo en cada generación (segunda aproximación)

### 4.3.3. Experimentos del *MCTS Agent2* con la función de evaluación sacada a partir del algoritmo genético

En esta última sección de resultados mostramos como de buenas son las funciones de evaluación del estado obtenidas mediante el algoritmo genético en la sección anterior aplicadas al *MCTS2 Agent* que era la mejor estrategia de exploración del espacio de estados (sección 4.3.1). Las funciones de evaluación obtenidas mediante el algoritmo genético las nombraremos como `Evo1Score`, `Evo2Score` y `Evo3Score`.

Para ello lo primero que hacemos es ver cuál es el mejor valor de la constante $C$ para los *MCTS2 Agent* con cada una de las tres funciones de evaluación obtenidas. Esto lo hacemos de la misma forma que en la sección 4.3.1, enfrentamos en 100 partidas (50 empezando un agente y 50 empezando el otro) con 100 ms de tiempo para cada acción a cada *MCTS2 Agent* contra el *Greedy Agent* con el `MidrangeScore` como función de evaluación. Tomamos también los valores de $C$ en el intervalo $[1, 3]$ con un paso de $0,1$.

En el gráfico 4.17 vemos los resultados de enfrentar al *MCTS2 Agent* con las nuevas funciones de evaluación obtenidas mediante la aproximación evolutiva al *Greedy Agent*. Comparando este gráfico con el gráfico 4.6 vemos que la función de evaluación `Evo1` tiene un comportamiento parecido a la función `Utility` mientras que las otras dos funciones tienen un comportamiento similar a la función `Midrange`.

Cuadro 4.5: Mejores indiviudos

| **Número** | Individuo 1 | Individuo 2 | Individuo 3 |
|---|---|---|---|
| 0 | 0.8844 | 0.7244 | 0.70678 |
| 1 | 0.5722 | 0.4547 | 0.5479 |
| 2 | 0.3038 | 0.8004 | 0.7428 |
| 3 | 0.3258 | 0.1634 | 0.1632 |
| 4 | 0.4966 | 0.6000 | 0.6002 |
| 5 | 0.5702 | 0.8319 | 0.7774 |
| 6 | 0.1405 | 0.1144 | 0.1233 |
| 7 | 0.5138 | 0.5321 | 0.4165 |
| 8 | 0.6154 | 0.6199 | 0.6506 |
| 9 | 0.1260 | 0.7243 | 0.5872 |
| 10 | 0.6913 | 0.6057 | 0.5908 |
| 11 | 0.0299 | 0.1375 | 0.2126 |
| 12 | 0.2187 | 0.1277 | 0.1742 |
| *Fitness* | 0.57 | 0.56 | 0.55 |

Al igual que en la sección 4.3.1 vamos a realizar 4 torneos para comparar a los agentes. Estos torneos serán iguales que los otros pero cambiando los participantes. Los participantes se muestran en la tabla 4.6.

Cuadro 4.6: Agentes que participan en los torneos de 3

| **Agente** | **Función de evaluación** | **Constante** |
|---|---|---|
| MCST2 | Utility | 3,0 |
| | Evo1 | 1,9 |
| | Evo1 | 2,4 |

El objetivo de estos torneos era comparar a los mejores agentes del último experimento con el *MCTS2 Agent* al que consideramos ganador del torneo anterior. Como los resultados de las funciones de evaluación `Evo2Score` y `Evo3Score` no fueron buenos, decidimos no incluir ningún agente con estas funciones en el torneo.

En la figura 4.18 y las correspondientes tablas C.12 y C.13 podemos ver los resultados de los dos primeros torneos. En el primero de ellos es claramente superior el agente con la función de evaluación `UtilityScore` pero en el segundo torneo, con ya 10 ms para elegir la siguiente acción, el agente con constante $C = 1,9$ y función de evaluación `Evo1Score` obtiene resultados similares al ganador del primer torneo.

En la figura 4.19 y las correspondientes tablas C.14 y C.15 podemos ver en los resultados del tercer torneo que el ganador es el agente con `Evo1Score` aunque con un desempleo en el torneo muy similar al agente con `UtilityScore`. En el torneo de 1000 ms el ganador vuelve a ser el mismo y vemos que incluso el otro agente con `Evo1Score` supera al agente con `UtilityScore`.

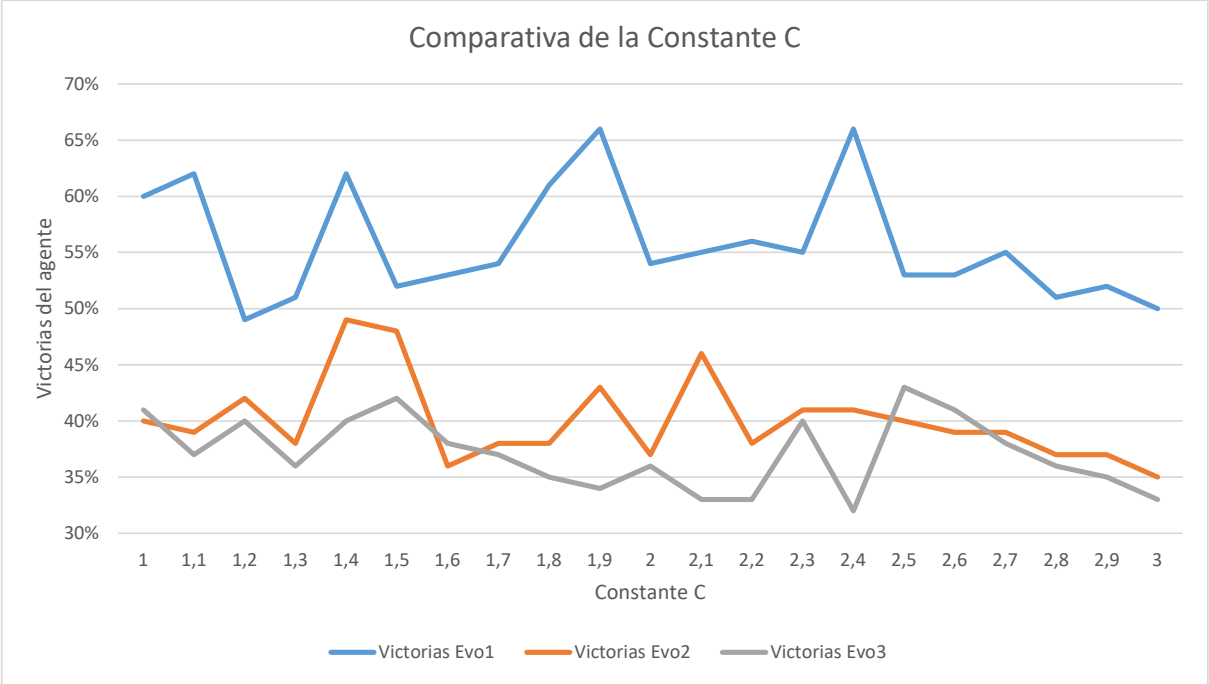


Figura 4.17: *MCTS2 Agent* vs *Greedy Agent* con diferentes valores de la constante C

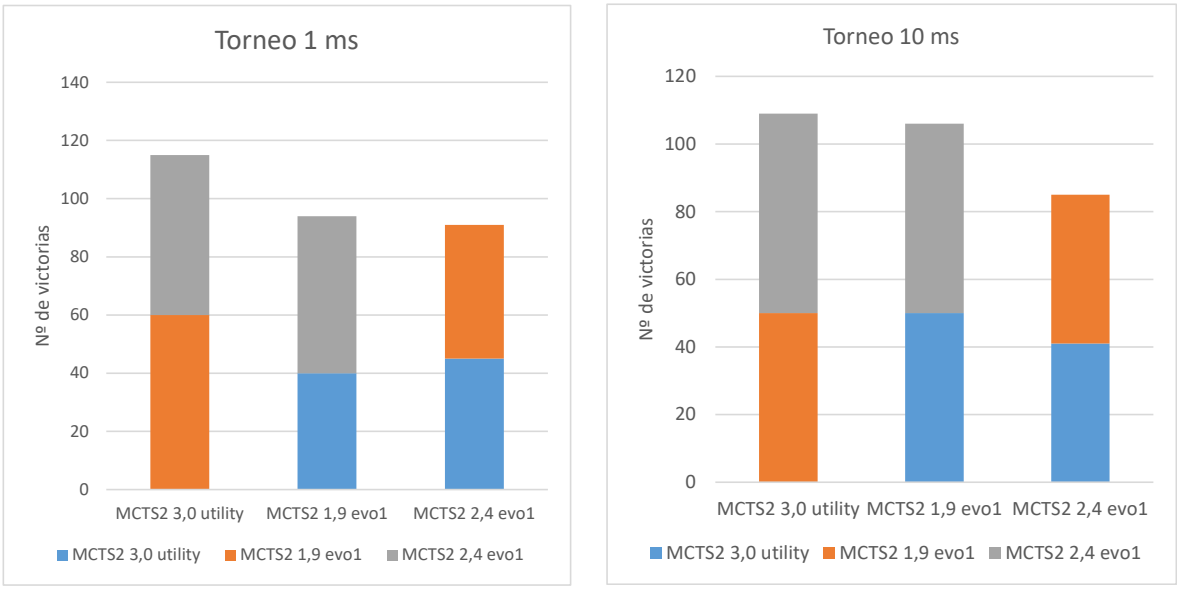


Figura 4.18: Torneos de 1 y 10 ms respectivamente

Para finalizar esta sección y el capítulo, vemos en 4.20 una comparativa del número de victorias en estos últimos torneos donde el ganador es el agente con función de evaluación `Evo1Score` y constante de exploración $C = 1,9$. Consideramos ganador a este agente porque

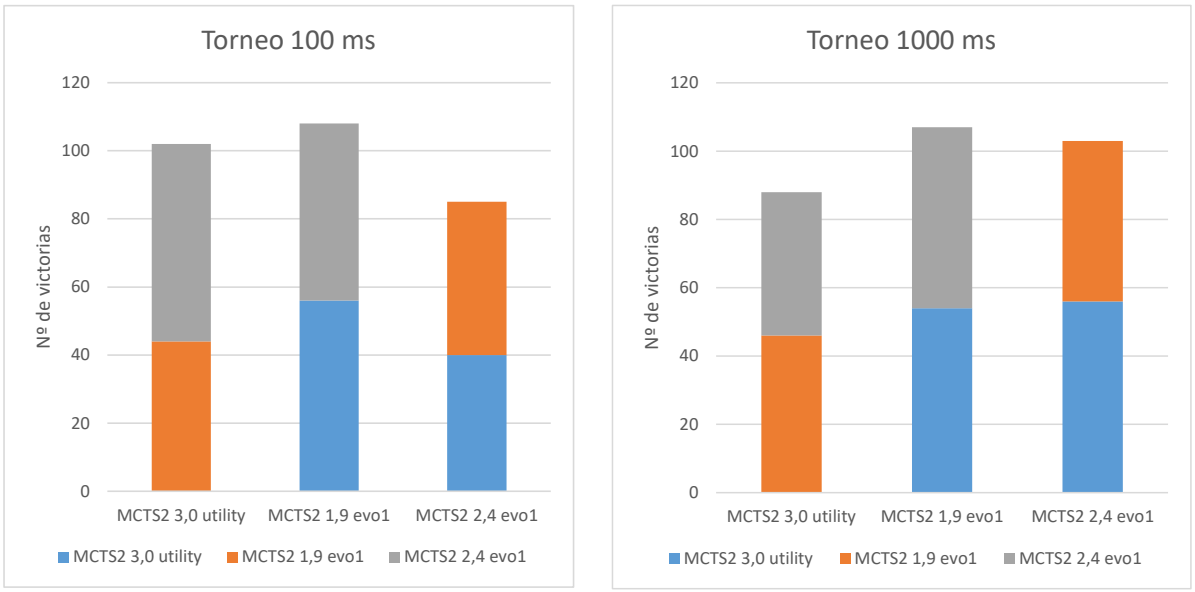


Figura 4.19: Torneos de 100 y 1000 ms respectivamente

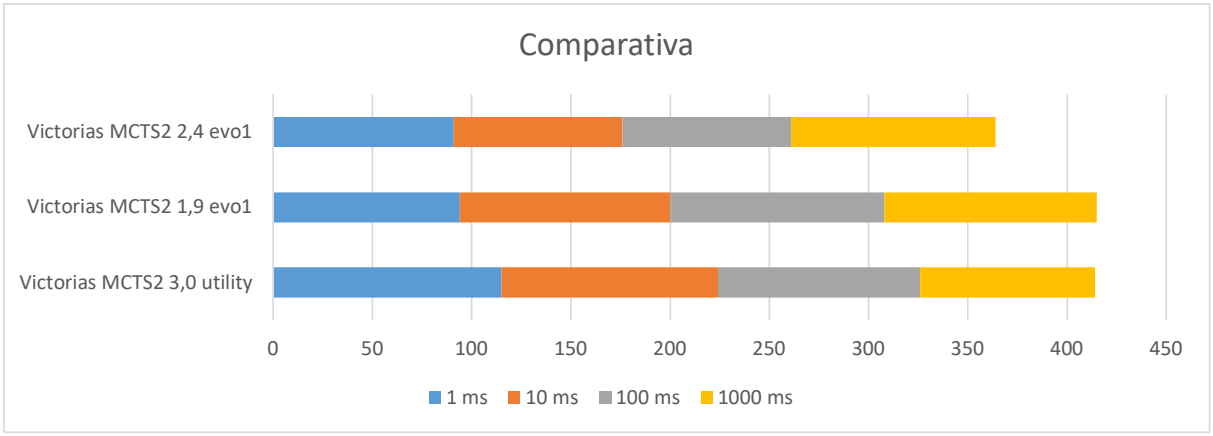


Figura 4.20: Comparativa del número de victorias de cada agente en todos los torneos

es el que mejor rendimiento consigue en los dos torneos más importantes que son el de 100 ms y el de 1000 ms.

### 4.3.4. Resumen de los resultados obtenidos

En los primeros experimentos (sección 4.3.1) hemos determinado con que constantes obtenían mejor rendimiento los agentes para un tiempo fijo y después con las mejores constantes hemos probado con distintos tiempos para seleccionar a 7 agentes. Estos 7 agentes se han enfrentado entre sí resultando ganador el *MCTS2 Agent* con constante $C = 3,0$ y

función de evaluación `UtilityScore`.

En los experimentos del algoritmo genético (sección 4.3.2) hemos estudiado los diferentes resultados en función de los métodos de selección, cruce y mutación con el fin de obtener una buena función de evaluación de forma automática. Tras una serie de ajustes a los parámetros del algoritmo hemos extraído las mejores funciones de evaluación obtenidas para poder compararlas con aquellas de las que ya disponíamos anteriormente.

Los últimos experimentos (sección 4.3.3) han sido prácticamente iguales que los primeros, primero hemos visto cuales son los mejores valores de la constante $C$ para las funciones de evaluación obtenidas del algoritmo genético y hemos enfrentado a los mejores agentes con el *MCTS2 Agent* con constante $C = 3,0$ y función de evaluación `UtilityScore`. Obteniendo como ganador el MCTS2 Agent con constante $C = 1,9$ y función de evaluación `Evo1Score`.

La conclusión final de los experimentos es que la mejor estrategia de búsqueda es la del *MCTS2 Agent* en cuanto hay algo de tiempo para explorar el espacio de estados. La función de evaluación obtenida en el algoritmo genético `Evo1Score` mejora a la función `UtilityScore` pero sin ser una mejora muy significativa.

# Capítulo 5

# Conclusiones

## 5.1. Resumen del trabajo realizado y los resultados obtenidos

Como hemos visto en la sección 3.3, *Hearthstone* es un juego de cartas muy complejo en el que el factor de aleatoriedad juega un papel decisivo. En todo momento cada jugador tiene que decidir entre muchas jugadas posibles de las cuales puede no saber con exactitud el resultado. Por este motivo, decidimos estudiar algunas de las diferentes técnicas de elaboración de jugadores o agentes automáticos capaces de jugar a *Hearthstone*.

Como hemos visto en la sección 1.2, en este trabajo nos hemos centrado en mejorar la calidad de la información de la que se dispone en cada momento mediante una función de evaluación de estados y en ser capaces de establecer que serie de acciones nos llevan a un mejor resultado.

Para simplificar los experimentos y debido al gran número de cartas distintas en el juego y la gran variedad de interacciones posibles, decidimos fijar el mazo del jugador y de su oponente de forma que todos los experimentos se hiciesen sobre el mismo conjunto de cartas. Como mazo de prueba nos quedamos con *Midrange Jade Shaman* [7] porque las estrategias *midrange* se encuentran a medio camino entre las *aggro*, donde lo único importante es la vida del oponente, y las *control* en las cuales hay que tener muchos factores en cuenta y el número de acciones posibles en cada momento es mucho más elevado.

Empezamos por crear un agente capaz de elegir las jugadas más favorables. En una primera aproximación creamos un jugador automático que examinase todas las jugadas posibles durante un turno y escogiese aquella serie de acciones que condujesen a un estado mejor, usando la función de evaluación por defecto del simulador `MidrangeScore` descrita en la sección 4.2.1. Tras implementarlo vimos que era inasumible en cuanto a tiempo (tardaba más de 10 minutos) y memoria (ocupaba 13 Gb aproximadamente). Con esto decidimos que era mejor simplificarlo y no guardar todo el árbol al mismo tiempo sino guardar sólo la rama que llevase a un mejor estado. Así reducimos el impacto en memoria del agente pero no en tiempo sin ofrecer un resultado mejor al que daba el *Greedy Agent* por defecto.

Así, decidimos emplear árboles de búsqueda de Monte Carlo ya que nos permitía ajustar-

nos a un presupuesto computacional dado además de ofrecer buenos resultados mejorando al *Greedy Agent* que en todo momento escoge la jugada que lleva a un estado mejor sin considerar estados más allá del siguiente.

De forma paralela, fuimos probando diferentes funciones de evaluación de estados y la función `UtilityScore` nos proporcionó unos buenos resultados tras hacerle algunas modificaciones. Sin embargo, todas estas funciones de evaluación dependían de conocimiento experto del dominio para ser efectivas y establecer de forma adecuada la importancia que se le daba a cada una de las variables observables del estado de juego. Como alternativa a esto y para no depender del conocimiento del dominio decidimos usar un algoritmo genético para crear de forma automática una función de evaluación de estados. De esta forma, el algoritmo asigna a cada variable el peso adecuado para poder permitir una estimación correcta del estado de la partida y obtener un mayor número de victorias.

Sin embargo, a la hora de elaborar el algoritmo genético era necesario jugar muchas partidas. Con una población de 200 individuos era necesario jugar 40000 partidas en cada generación. Por ello, para jugar estas partidas necesarias para el funcionamiento del algoritmo genético tuvimos que emplear un agente que siguiese una estrategia voraz en vez de usar árboles de búsqueda de Monte Carlo por ser el primero mucho más rápido.

En la sección de experimentos y resultados (sección 4.3) hemos hecho los experimentos de estrategias de exploración del espacio de estados y los experimentos con el algoritmo genético por separado. Después combinábamos los mejores resultados de ambos experimentos y veíamos si obteníamos un agente que produjese un buen rendimiento.

Las conclusiones que hemos sacado de todos estos experimentos son las siguientes:

- En el caso de las estrategias de búsqueda, cuando se tiene poco tiempo de computo para elegir una acción lo mejor es optar por una estrategia voraz como la del *Greedy Agent* o una estrategia como la del *MCTS1 Agent* que evalúa muchos de los estados del espacio de búsqueda y no simula tantos.

- Cuando el tiempo de cómputo es mayor la estrategia que mejores resultados produce es la del *MCTS2 Agent* aunque es muy importante la elección correcta de la constante de exploración.

- A la hora de buscar una buena función de evaluación hemos visto que el espacio de búsqueda es muy amplio y que a la hora de obtener mejores resultados es necesario meter algo de conocimiento experto.

- Cuando hemos combinado los resultados de los dos primeros experimentos los resultados obtenidos no han sido buenos en 2/3 casos. Suponemos que es porque la función de evaluación de un *Greedy Agent* evalúa todo tipo de estados mientras que la de un *MCTS2 Agent* solo evalúa estados terminales y toda la información que se almacena en el árbol es con estas evaluaciones. Por ello, en un caso extremo, una función que valorara siempre la mejor opción con una puntuación alta menos en los estados terminales que hiciera lo contrario, sería una función buena para el *Greedy Agent* y muy mala para el *MCTS2 Agent*.

## 5.2. Trabajo futuro

A la hora de elaborar un agente automático se pueden utilizar otras aproximaciones diferentes a las utilizadas en este trabajo. Por ejemplo, en [48] se propone un método alternativo en el cual se simplifican los estados mediante técnicas de extracción de parámetros y se aplican diferentes técnicas de aprendizaje automático como por ejemplo la construcción de un modelo de regresión. Este aprendizaje automático podría aplicarse sobre la evaluación de los estados de forma que, recurriendo a una base de datos con partidas guardadas, se pudiese establecer la idoneidad de cada estado en función del número de partidas en las que este estado conduce a una victoria.

Es importante recordar que, en nuestro caso, hemos limitado los experimentos a un sólo mazo. Resultaría interesante trabajar con más tipos de mazos con un estilo de juego radicalmente distinto o incluso generalizarlo todo más de forma que fuese posible jugar con cualquier mazo.

También hay que tener en cuenta que al utilizar el algoritmo genético se enfrentaban entre sí dos agentes que seguían una estrategia voraz. Si se utilizase en este caso uno de los agentes desarrollados utilizando MCTS podría ser posible que se mejorase mucho más. Sin embargo, esto supondría que el tiempo necesario para cada experimento fuese inabarcable para nosotros. Además, otra posible mejora sería probar el uso de diferentes heurísticas para elegir el siguiente nodo en un árbol de búsqueda de Monte Carlo de forma que se tuviesen en cuenta más factores de los que se tienen ahora.

A la hora de elaborar este trabajo se suponía que el agente no conocía las cartas que podría tener en el mazo el oponente, sin embargo, esto no es del todo cierto a la hora de jugar dos humanos entre sí. La mayor parte de los jugadores de *Hearthstone* son capaces de establecer aproximadamente que mazo está usando su oponente en función de su clase y de las primeras cartas que juegue. Esto es porque el usar una clase u otra limita el número de cartas disponibles y además, y por la naturaleza de los juegos competitivos se puede suponer que el oponente busca ganar, de forma que usará cartas que se complementen para obtener mayor ventaja. Por ejemplo, si el oponente tiene una carta que dice "Obtienes un punto de vida cada vez que descartas una carta" se puede asumir que más tarde usará cartas que le permitan descartar. Por ello puede resultar interesante la elaboración de un agente que disponga de este conocimiento, ya sea porque lo ha aprendido de forma automática al trabajar sobre una base de datos de partidas anteriores o porque se ha introducido ese conocimiento del dominio en él por parte de alguien que conozca el juego en gran profundidad.

# Capítulo 5

# Conclusions

## 5.1. Summary of the work done and the results obtained

As we have seen in section 3.3, *Hearthstone* is a very complex game where the randomness plays a key role. All the time each player has to decide between many possible moves of which the player might not know the result. For this reason, we decided to study some of the different techniques for building automatic agents capable of playing *Hearthstone*.

As we have seen in section 1.2, in this work we have focused on improving the quality of the information available at each moment through a status evaluation function and on being able to determine which sequence of actions give us a better result.

To make the experiments simpler and because of the large number of different cards in the game and the wide variety of possible interactions, we decided to fix the players' deck so that every experiment would be done withe the same cards. As test deck we stay with *Midrange Jade Shaman* [7] because *midrange* strategies are placed between *aggro* strategies, where the only important thing is the life of the opponent, and *control* strategies in which many factors must be taken in consideration and the number of possible actions at each moment is higher.

We start building an agent capable of choosing the most favorable plays. In a first approach we created an agent that would explore all the moves during a turn and choose the sequence of actions that would lead to a better state using the default evaluation function of the simulator *MidrangeScore* described in section 4.2.1. After the implementation we saw that it was unacceptable in terms of time (it took more than 10 minutes) and memory (it needed approximately 13 Gb). With this we decided that is better to simplify it and not to keep in memory the whole tree, but to keep only the branch that would lead to a better state. This way we reduce the impact in memory but not in time and we don't offer a better result than the one given by the *Greedy Agent*.

Therefore, we decide to use Monte Carlo tree search since it allows us to adapt to a given computational budget in addition to offering good results by improving the Greedy Agent that always chooses the move that leads to a better state without considering states beyond the next one.

At the same time, we tested different state evaluation functions and the `UtilityScore` function gave us good results after maiking some modifications. However, all these evaluation functions were dependent on expert knowledge of the domain to be effective and to properly establish the importance given to each of the obsarvable variables of the state of the game. As an alternative to this and in order not to depend on domain knowledge we decided to use a genetic algorithm to automatically build a state evaluation function. The algorithm sets the right weight to each variable to allow a good estimation of the game state and to obtain a higher win rate.

The execution of the genetic algorithm requiered many games to be played. With a population of 200 individuals it was necessary to play 40000 games in each generation. Therefore, to play these games we had to employ and agent that follow a greedy strategy instead of using Monte Carlo tree search because the first was much faster.

In the section of experiments and results (section 4.3) we have made the experiments of state space exploration strategies and the experiments with the genetic algorithm separately. Then we combined the best results of both experiments and saw if we get an agent that gives a good performance.

The conclusions we have extracted from all these experiments are the following:

- In the case of search strategies, when you have a little computational budget to choose an action the best option is a greedy strategy such as the *Greedy Agent* or a strategy such as the *MCTS1 Agent* that evaluates many of the states of the search space and does not simulate all.
- When the computational budget is larger, the strategy that produces the best results is *MCTS2 Agent's*, althought the correct choice of the exploration constant is very important.
- When looking for a good evaluation function we have seen that the search space is very wide and in order to get better results it is necessary to introduce some expert knowledge.
- When we have combined the results of the first two experiments the results obtained have not been good in 2/3 of the cases. We assume that it is because the evaluation of a *Greedy Agent* evaluates every type of state while the *MCTS2 Agent's* one only evaluates terminal states and all the information that is stored in the tree is with this evaluations. In an extreme situation, a function that always chooses the best option less in the terminal state where do the opposite would be a good function for the *Greedy Agent* and a very bad for the *MCTS2 Agent*.

## 5.2. Future work

Other approaches than those used in this work can be used when developing an automatic player. For example, in [48] an alternative approach is proposed in which the states are

simplified by parameter extraction techniques and different machine learning techniques are applied such as the construction of a regression model. These machine learning techniques could be used to the evaluate the states and, using a database with saved games, the fitness of each state could be established according to the number of games in which this state leads to a win.

It's important to remember that, in our case, we have restricted the experiments to a single deck. It would be interesting to work with more types of decks with a radically different play style or to generalise everything more so that it would be possible to play with any deck.

It should also be noted that when using the genetic algorithm, two *Greedy Agents* were playing against each other. If one of the agents developed using MCTS was used in this situation, it could be possible to improve it much more. However, this would mean that the time needed for each experiment would be impossible for us. In addition, another possible improvement would be to try using different heuristics to choose the next node in a Monte Carlo tree search.

At the time of preparing this work it was supposed that the agent did not know what cards the opponent might have in the deck, however, this is not completely true when playing two people against each other. Most Hearthstone players are able to establish approximately what deck their opponent is using based on the class of their hero and the first cards they play. This is because using one class or another limits the number of cards available and in addition, and by the nature of competitive games, it can be expected that the opponent will try to win, so he will use cards that complement each other to take advantage. For example, if the opponent has a card that says "You get one life point each time you discard a card" it can be expected that he will later use cards that allow him to discard cards. Therefore it can be interesting to build up an agent who has this knowledge, either because he has learned it automatically when working on a database of past games or because he has introduced this knowledge of the domain into it manually by someone who knows how to play the game properly.

# Apéndice A

# Material adicional

En este trabajo hemos utilizado el framework de la competición Hearthstone-AI competition (sección 3.6) que se puede descargar en el repositorio de Github https://github.com/ADockhorn/HearthstoneAICompetition.

Nuestro proyecto está en el repositorio de Github https://github.com/matgar03/TFGHS y nuestra contribución se encuentra en las carpetas Agent y Evolutivo.

En la primera carpeta se encuentran todos los agentes utilizados en el trabajo y en la segunda todas las clases necesarias para el algoritmo evolutivo. En la carpeta `src` que contiene a estas dos se encuentran las clases utilizadas para lanzar los experimentos que enfrentaban a los diferentes agentes.

# Apéndice B

# Contribuciones individuales al proyecto

## B.1. Contribución de Mateo García Pérez

En un primer lugar nos intentamos familiarizar en mayor medida con el juego *Hearthstone* y su funcionamiento. Para ello jugamos varias partidas fijándonos en cada una de las diferentes elecciones que había que hacer en cada momento, que variables nos indicaban qué jugador llevaba ventaja y hasta donde podíamos intentar estimar el comportamiento futuro del juego teniendo en cuenta las posibilidades del adversario y la aleatoriedad de algunos efectos del juego. También vimos partidas profesionales para intentar ampliar el conocimiento que teníamos nosotros sobre el juego prestando especial atención en qué variables del juego llevaban a los profesionales a tomar una decisión u otra.

Tras eso, empezamos a estudiar el funcionamiento del simulador sobre el que íbamos a desarrollar el trabajo así como de los diferentes agentes presentados en años anteriores de la competición. Fueron necesarios varios días estudiando el código base de este simulador para saber cómo se debía modificar y ampliar para que un agente desarrollado por nosotros pudiese funcionar dentro de este entorno. Tras estudiar agentes de otros años llegamos a la conclusión de que dos eran los aspectos más decisivos a la hora de implementar un jugador automático de *Hearthstone*, ser capaces de tomar decisiones considerando posibles estados futuros y poder estimar de forma acertada cómo de ventajoso era cada estado para cada jugador.

Con esto empezamos una fase de documentación leyendo mayoritariamente artículos que trataban sobre el uso de árboles de búsqueda de Monte Carlo en juegos, no necesariamente *Hearthstone*, aunque sin descartar otras alternativas como el uso de una red neuronal sobre una base de datos de partidas jugadas o la exploración en un grafo en el que cada nodo podría representar un estado del juego.

Mientras tanto empezamos a desarrollar nuestro primer agente. Este agente examinaba todas las posibles acciones que se podían tomar en un turno y escogía la que se consideraba mejor (usando en ese momento todavía las funciones de evaluación de estados predefinidas en el simulador). Sin embargo, el explorar todas las posibles jugadas de un turno demostró ser una mala aproximación porque suponía explorar cientos de miles de estados en cada

turno, lo que llevaba a un coste en tiempo y memoria muy elevado. Por esto nos decidimos definitivamente por el uso de árboles de búsqueda de Monte Carlo, ya que nos permitía ajustar el número de estados posibles explorados y el tiempo usado, e implementamos el primer agente que hacía uso de esta estrategia.

De forma paralela probamos diferentes funciones de evaluación alterando algunos aspectos de aquellas que ya incluía el simulador, pero sin lograr mejorar los resultados. Consultando diversos artículos llegamos a [26], que ofrecía un estudio sobre una función de evaluación de estados en un juego similar a *Hearthstone*. A partir de esta función y tras introducir unos cambios llegamos a `UtilityScore` que nos proporcionó unos resultados mejores.

Sin embargo, esta búsqueda nos dejó claro que para poder crear una buena función de evaluación era necesario disponer de gran nivel de conocimiento sobre el juego. Por ello decidimos que sería interesante la implementación de un método que pudiese calcular de forma automática una buena función de evaluación sin depender de un conocimiento anterior sobre el juego. Tras estudiar diferentes alternativas nos decidimos por el uso de algoritmos genéticos del que desarrollamos un primer prototipo y, tras ver que nos otorgaba resultados favorables, decidimos implementarlo con mayor detalle e incluirlo como parte del trabajo.

Mientras que hasta ese momento Jorge y yo estuvimos trabajando de forma conjunta a partir de aquí empezamos a centrarnos cada uno en un tema. Yo me dediqué a trabajar sobre el algoritmo genético e intentar mejorar su funcionamiento. Para ello estuve documentándome sobre algoritmos genéticos, su funcionamiento y su uso general para optimizar funciones. Tras un tiempo decidí reescribir todo el código relativo al algoritmo genético para poder dividirlo en partes y trabajar sobre ellas, mejorándolas de una en una y de tal forma que más tarde pudiesen ser ampliadas en el caso de ser necesario.

Tras esto, empecé a realizar los experimentos sobre el funcionamiento del algoritmo genético recogidos en la sección 4.3.2. Como se puede ver ahí, en un primer momento no se alcanzaba un resultado que ofreciese ventaja frente a las funciones de evaluación de las que se disponían antes. Por ello, tuve que hacer modificaciones a la forma de construir la función de evaluación en base a los individuos del algoritmo y ampliar el tamaño de la población. Con esto ya se lograron unos resultados que mejoraban las otras funciones de evaluación y además sin depender del conocimiento sobre el juego.

Paralelamente fuimos escribiendo la memoria. Yo me centré en la parte relativa a los algoritmos genéticos y al funcionamiento de las diferentes funciones de evaluación, esto son las secciones 2.3 y 4.2. También analicé los resultados del algoritmo genético en la sección 4.3.2. Junto con esto escribí la sección 1 con su traducción, la parte relativa al resumen del trabajo realizado en 5.1, el posible trabajo futuro en 5.2 y unos fragmentos sobre el funcionamiento del juego en 3.2 y por qué supone un objeto de estudio interesante en 3.3.

## B.2. Contribución de Jorge Sainero Valle

Como ha comentado Mateo, las primeras fases del proyecto las desarrollamos juntos. Estas fases son las relativas a conocer el juego y entender que factores eran más determinantes en una partida.

Después seguimos trabajando conjuntamente también y nos informamos de como funcionaba el simulador y la competición, así como estuvimos viendo los trabajos y agentes publicados en los años anteriores.

Una vez decidido que íbamos a desarrollar agentes basados en la exploración del espacio de estado, empezamos a desarrollar los agentes. Mientras desarrollábamos los agentes comenzamos a investigar sobre diferentes funciones de evaluación hasta que llegamos a la tesis que inspiró la función de evaluación `UtilityScore`.

Más adelante decidimos ver si eramos capaces de obtener una buena función de evaluación sin necesidad de tener conocimiento sobre el dominio. Por ello programamos un primer algoritmo genético que más adelante cambiaría Mateo.

Llegados a este punto es cuando empezamos a trabajar de forma separada y mientras Mateo se dedicaba a trabajar sobre el algoritmo genético, yo me dediqué a las estrategias de búsqueda e implemente el *MCTS2 Agent*. Todos los experimentos de la sección de resultados relativos a estrategias de búsqueda (secciones 4.3.1 y 4.3.3) los diseñé yo también.

A la hora de escribir la memoria si nos dividimos el trabajo. Yo escribí toda la parte sobre árboles de Monte Carlo del segundo capítulo (secciones 2.1 y 2.2). También escribí el tercer capítulo completo a excepción de la primera mitad de la sección relativa a explicar como funciona el Hearthstone (sección 3.2) y la parte final de la siguiente sección donde contábamos porque el Hearthstone era un reto interesante para la IA (sección 3.3).

Como yo me había dedicado más a los árboles de búsqueda de Monte Carlo y había implementado el *MCTS2 Agent*, en el cuarto capítulo, escribí todas las secciones sobre estrategias de búsqueda (sección 4.1). Por este mismo motivo y como yo había realizado los experimentos de las estrategias de búsqueda también escribí sus correspondientes secciones (secciones 4.3.1 y 4.3.3).

En el último capítulo, escribí la parte de la sección 5.1 que correspondía a las conclusiones obtenidas de los experimentos y sus resultados y realicé la traducción de este capítulo.

# Apéndice C

# Resultados adicionales

En este anexo incluimos las tablas con la información necesaria para construir los gráficos de la sección de resultados (sección 4.3).

Cuadro C.3: *MCTS2 Agent* vs *Greedy Agent*

| C | Victorias UtilityScore | Victorias MidrangeScore |
|---|---|---|
| **1,0** | 51 | 36 |
| **1,1** | 52 | 43 |
| **1,2** | 63 | 36 |
| **1,3** | 44 | **55** |
| **1,4** | 60 | 45 |
| **1,5** | **64** | 43 |
| **1,6** | 61 | 44 |
| **1,7** | 59 | 46 |
| **1,8** | 60 | 51 |
| **1,9** | 49 | 44 |
| **2,0** | 62 | 38 |
| **2,1** | 59 | 42 |
| **2,2** | 58 | 43 |
| **2,3** | 41 | 39 |
| **2,4** | 58 | **56** |
| **2,5** | 57 | 45 |
| **2,6** | 55 | 53 |
| **2,7** | 61 | 51 |
| **2,8** | 62 | 38 |
| **2,9** | 55 | 42 |
| **3,0** | **71** | 50 |

Cuadro C.1: *SimTree Agent* vs *Greedy Agent*

| **EXPLORATION_RATE** | **Victorias UtilityScore** | **Victorias MidrangeScore** |
|---|---|---|
| **0** | 50 | 48 |
| **0,05** | 47 | 49 |
| **0,10** | 52 | 52 |
| **0,15** | 49 | 48 |
| **0,20** | 51 | 53 |
| **0,25** | 45 | 53 |
| **0,30** | 48 | 52 |
| **0,35** | 48 | 54 |
| **0,40** | 53 | 53 |
| **0,45** | 55 | 56 |
| **0,50** | **56** | **60** |
| **0,55** | 52 | 55 |
| **0,60** | 53 | 52 |
| **0,65** | 53 | 48 |
| **0,70** | 58 | 51 |
| **0,75** | **63** | **57** |
| **0,80** | 60 | 48 |
| **0,85** | 57 | 50 |
| **0,90** | 53 | 46 |
| **0,95** | 49 | 49 |
| **1,00** | 50 | 49 |

Cuadro C.2: *MCTS1 Agent* vs *Greedy Agent*

| C | Victorias UtilityScore | Victorias MidrangeScore |
|---|---|---|
| **1,0** | 53 | 54 |
| **1,1** | 47 | 58 |
| **1,2** | 63 | 52 |
| **1,3** | 56 | 48 |
| **1,4** | 52 | 53 |
| **1,5** | 50 | 48 |
| **1,6** | 53 | 53 |
| **1,7** | 51 | 56 |
| **1,8** | 54 | 53 |
| **1,9** | 59 | 48 |
| **2,0** | 59 | 48 |
| **2,1** | 56 | 54 |
| **2,2** | 57 | 47 |
| **2,3** | 58 | 58 |
| **2,4** | **63** | **58** |
| **2,5** | 57 | 50 |
| **2,6** | 45 | 53 |
| **2,7** | 55 | 52 |
| **2,8** | 62 | 46 |
| **2,9** | **62** | **66** |
| **3,0** | 55 | 52 |

Cuadro C.4: *SimTree Agent* vs *Greedy Agent*

| Score | EXPLORATION_ RATE | Tiempo (ms) | Victorias Local | Victorias Visitante | Victorias |
|---|---|---|---|---|---|
| Utility | 0,5 | 1 | 52 % | 28 % | 40 % |
| Utility | 0,5 | 10 | 52 % | 48 % | 50 % |
| Utility | 0,5 | 100 | 70 % | 42 % | **56 %** |
| Utility | 0,5 | 1000 | 52 % | 32 % | 42 % |
| Midrange | 0,5 | 1 | 44 % | 38 % | 41 % |
| Midrange | 0,5 | 10 | 58 % | 32 % | 45 % |
| Midrange | 0,5 | 100 | 66 % | 52 % | **59 %** |
| Midrange | 0,5 | 1000 | 50 % | 32 % | 41 % |
| Utility | 0,75 | 1 | 30 % | 28 % | 29 % |
| Utility | 0,75 | 10 | 42 % | 52 % | 47 % |
| Utility | 0,75 | 100 | 68 % | 56 % | **62 %** |
| Utility | 0,75 | 1000 | 58 % | 36 % | 47 % |
| Midrange | 0,75 | 1 | 32 % | 24 % | 28 % |
| Midrange | 0,75 | 10 | 40 % | 34 % | 37 % |
| Midrange | 0,75 | 100 | 58 % | 40 % | **49 %** |
| Midrange | 0,75 | 1000 | 50 % | 42 % | 46 % |

Cuadro C.5: *MCTS1 Agent* vs *Greedy Agent*

| Score | C | Tiempo (ms) | Victorias Local | Victorias Visitante | Victorias |
|---|---|---|---|---|---|
| Utility | 2,4 | 1 | 58 % | 58 % | 58 % |
| Utility | 2,4 | 10 | 66 % | 34 % | 50 % |
| Utility | 2,4 | 100 | 64 % | 60 % | **62 %** |
| Utility | 2,4 | 1000 | 64 % | 48 % | 56 % |
| Midrange | 2,4 | 1 | 56 % | 48 % | 52 % |
| Midrange | 2,4 | 10 | 56 % | 42 % | 49 % |
| Midrange | 2,4 | 100 | 62 % | 50 % | 56 % |
| Midrange | 2,4 | 1000 | 68 % | 54 % | **61 %** |
| Utility | 2,9 | 1 | 46 % | 36 % | 41 % |
| Utility | 2,9 | 10 | 56 % | 48 % | 52 % |
| Utility | 2,9 | 100 | 66 % | 48 % | **57 %** |
| Utility | 2,9 | 1000 | 62 % | 44 % | 53 % |
| Midrange | 2,9 | 1 | 52 % | 56 % | 54 % |
| Midrange | 2,9 | 10 | 50 % | 40 % | 45 % |
| Midrange | 2,9 | 100 | 66 % | 52 % | 59 % |
| Midrange | 2,9 | 1000 | 70 % | 54 % | **62 %** |

Cuadro C.6: *MCTS2 Agent* vs *Greedy Agent*

| Score | C | Tiempo (ms) | Victorias Local | Victorias Visitante | Victorias |
|---|---|---|---|---|---|
| Utility | 1,5 | 1 | 16 % | 12 % | 14 % |
| Utility | 1,5 | 10 | 58 % | 50 % | 54 % |
| Utility | 1,5 | 100 | 62 % | 56 % | 59 % |
| Utility | 1,5 | 1000 | 70 % | 50 % | **60 %** |
| Midrange | 1,3 | 1 | 22 % | 18 % | 20 % |
| Midrange | 1,3 | 10 | 42 % | 32 % | 37 % |
| Midrange | 1,3 | 100 | 60 % | 34 % | 47 % |
| Midrange | 1,3 | 1000 | 58 % | 38 % | **48 %** |
| Utility | 3 | 1 | 18 % | 28 % | 23 % |
| Utility | 3 | 10 | 60 % | 36 % | 48 % |
| Utility | 3 | 100 | 68 % | 70 % | 69 % |
| Utility | 3 | 1000 | 72 % | 68 % | **70 %** |
| Midrange | 2,4 | 1 | 26 % | 18 % | 22 % |
| Midrange | 2,4 | 10 | 44 % | 28 % | 36 % |
| Midrange | 2,4 | 100 | 40 % | 34 % | **37 %** |
| Midrange | 2,4 | 1000 | 40 % | 32 % | 36 % |

Cuadro C.7: Tabla del torneo de 1 ms

| | **Greedy** | **SimTree 0,5 midrange** | **SimTree 0,75 utility** | **MCTS1 2,4 utility** | **MCTS1 2,9 midrange** | **MCTS2 1,5 utility** | **MCTS2 3,0 utility** |
|---|---|---|---|---|---|---|---|
| **Greedy** | 0 | 59 | 56 | 52 | 44 | 74 | 78 |
| **SimTree 0,5 midrange** | 41 | 0 | 41 | 35 | 39 | 67 | 56 |
| **SimTree 0,75 utility** | 43 | 59 | 0 | 46 | 53 | 73 | 61 |
| **MCTS1 2,4 utility** | 48 | 65 | 54 | 0 | 55 | 67 | 75 |
| **MCTS1 2,9 midrange** | 56 | 61 | 47 | 45 | 0 | 81 | 73 |
| **MCTS2 1,5 utility** | 26 | 33 | 27 | 33 | 19 | 0 | 47 |
| **MCTS2 3,0 utility** | 22 | 44 | 39 | 25 | 27 | 53 | 0 |

Cuadro C.8: Tabla del torneo de 10 ms

| | **Greedy** | **SimTree 0,5 midrange** | **SimTree 0,75 utility** | **MCTS1 2,4 utility** | **MCTS1 2,9 midrange** | **MCTS2 1,5 utility** | **MCTS2 3,0 utility** |
|---|---|---|---|---|---|---|---|
| **Greedy** | 0 | 59 | 53 | 45 | 52 | 48 | 45 |
| **SimTree 0,5 midrange** | 41 | 0 | 44 | 39 | 42 | 43 | 38 |
| **SimTree 0,75 utility** | 47 | 56 | 0 | 46 | 60 | 61 | 45 |
| **MCTS1 2,4 utility** | 55 | 61 | 54 | 0 | 53 | 48 | 45 |
| **MCTS1 2,9 midrange** | 48 | 58 | 40 | 47 | 0 | 47 | 48 |
| **MCTS2 1,5 utility** | 52 | 57 | 38 | 52 | 53 | 0 | 47 |
| **MCTS2 3,0 utility** | 55 | 62 | 55 | 55 | 52 | 53 | 0 |

Cuadro C.9: Tabla del torneo de 100 ms

| | **Greedy** | **SimTree 0,5 midrange** | **SimTree 0,75 utility** | **MCTS1 2,4 utility** | **MCTS1 2,9 midrange** | **MCTS2 1,5 utility** | **MCTS2 3,0 utility** |
|---|---|---|---|---|---|---|---|
| **Greedy** | 0 | 49 | 39 | 47 | 44 | 40 | 33 |
| **SimTree 0,5 midrange** | 51 | 0 | 42 | 39 | 55 | 41 | 29 |
| **SimTree 0,75 utility** | 61 | 58 | 0 | 41 | 55 | 42 | 34 |
| **MCTS1 2,4 utility** | 53 | 61 | 59 | 0 | 55 | 55 | 44 |
| **MCTS1 2,9 midrange** | 56 | 45 | 45 | 45 | 0 | 39 | 43 |
| **MCTS2 1,5 utility** | 60 | 59 | 58 | 45 | 61 | 0 | 39 |
| **MCTS2 3,0 utility** | 67 | 71 | 66 | 56 | 57 | 61 | 0 |

Cuadro C.10: Tabla del torneo de 1000 ms

| | **Greedy** | **SimTree 0,5 midrange** | **SimTree 0,75 utility** | **MCTS1 2,4 utility** | **MCTS1 2,9 midrange** | **MCTS2 1,5 utility** | **MCTS2 3,0 utility** |
|---|---|---|---|---|---|---|---|
| **Greedy** | 0 | 47 | 48 | 42 | 42 | 44 | 34 |
| **SimTree 0,5 midrange** | 53 | 0 | 35 | 41 | 44 | 43 | 39 |
| **SimTree 0,75 utility** | 52 | 65 | 0 | 46 | 49 | 42 | 37 |
| **MCTS1 2,4 utility** | 58 | 59 | 53 | 0 | 56 | 57 | 41 |
| **MCTS1 2,9 midrange** | 58 | 56 | 51 | 44 | 0 | 44 | 36 |
| **MCTS2 1,5 utility** | 56 | 57 | 58 | 43 | 56 | 0 | 44 |
| **MCTS2 3,0 utility** | 66 | 61 | 63 | 59 | 64 | 56 | 0 |

Cuadro C.11: *MCTS2 Agent* vs *Greedy Agent*

| **C** | **Victorias Evo1** | **Victorias Evo2** | **Victorias Evo3** |
|---|---|---|---|
| **1** | 60 % | 40 % | 41 % |
| **1,1** | 62 % | 39 % | 37 % |
| **1,2** | 49 % | 42 % | 40 % |
| **1,3** | 51 % | 38 % | 36 % |
| **1,4** | 62 % | 49 % | 40 % |
| **1,5** | 52 % | 48 % | 42 % |
| **1,6** | 53 % | 36 % | 38 % |
| **1,7** | 54 % | 38 % | 37 % |
| **1,8** | 61 % | 38 % | 35 % |
| **1,9** | **66 %** | 43 % | 34 % |
| **2** | 54 % | 37 % | 36 % |
| **2,1** | 55 % | 46 % | 33 % |
| **2,2** | 56 % | 38 % | 33 % |
| **2,3** | 55 % | 41 % | 40 % |
| **2,4** | **66 %** | 41 % | 32 % |
| **2,5** | 53 % | 40 % | 43 % |
| **2,6** | 53 % | 39 % | 41 % |
| **2,7** | 55 % | 39 % | 38 % |
| **2,8** | 51 % | 37 % | 36 % |
| **2,9** | 52 % | 37 % | 35 % |
| **3** | 50 % | 35 % | 33 % |

Cuadro C.12: Tabla del torneo de 1 ms

| | **MCTS2 3,0 utility** | **MCTS2 1,9 evo1** | **MCTS2 2,4 evo1** |
|---|---|---|---|
| **MCTS2 3,0 utility** | 0 | 60 | 55 |
| **MCTS2 1,9 evo1** | 40 | 0 | 54 |
| **MCTS2 2,4 evo1** | 45 | 46 | 0 |

Cuadro C.13: Tabla del torneo de 10 ms

| | **MCTS2 3,0 utility** | **MCTS2 1,9 evo1** | **MCTS2 2,4 evo1** |
|---|---|---|---|
| **MCTS2 3,0 utility** | 0 | 50 | 59 |
| **MCTS2 1,9 evo1** | 50 | 0 | 56 |
| **MCTS2 2,4 evo1** | 41 | 44 | 0 |

Cuadro C.14: Tabla del torneo de 100 ms

| | **MCTS2 3,0 utility** | **MCTS2 1,9 evo1** | **MCTS2 2,4 evo1** |
|---|---|---|---|
| **MCTS2 3,0 utility** | 0 | 44 | 58 |
| **MCTS2 1,9 evo1** | 56 | 0 | 52 |
| **MCTS2 2,4 evo1** | 40 | 45 | 0 |

Cuadro C.15: Tabla del torneo de 1000 ms

| | **MCTS2 3,0 utility** | **MCTS2 1,9 evo1** | **MCTS2 2,4 evo1** |
|---|---|---|---|
| **MCTS2 3,0 utility** | 0 | 46 | 42 |
| **MCTS2 1,9 evo1** | 54 | 0 | 53 |
| **MCTS2 2,4 evo1** | 56 | 47 | 0 |

# Bibliografía


[1] Conference on Games (CoG) ieee. https://ieee-cog.org/2020/. Accedida en: 11-06-2020.

[2] Hearthstone-AI competition. https://dockhorn.antares.uberspace.de/wordpress/. Accedido en: 06-05-2020.

[3] Hearthstone Card Library. https://playhearthstone.com/en-us/cards?collectible=1&set=wild. Accedido en 14-04-2020.

[4] Hearthstone deck tracker. https://github.com/fatheroctopus/hdt-deck-predictor. Accedido en 16-05-2020.

[5] Información de Wikipedia sobre Gwent: The Witcher Card Game. https://es.wikipedia.org/wiki/Gwent:_The_Witcher_Card_Game. Accedida en: 11-06-2020.

[6] Información de Wikipedia sobre Magic: The Gathering. https://es.wikipedia.org/wiki/Magic:_El_encuentro. Accedida en: 11-06-2020.

[7] Jade Shaman - Hearthstone Decks. https://www.hearthpwn.com/decks/1311850-jadeshamanai. Accedida en: 23-04-2020.

[8] SabberStone Simulator. https://github.com/HearthSim/SabberStone. Accedido en 23-04-2020.

[9] Rajeev Agrawal. Sample mean based index policies by $O(n \log n)$ regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078, 1995.

[10] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[12] Guillaume M. J-B. Chaslot, Mark H. M. Winands, H. Jaap van den Herik, JOS W.H.M. UITERWIJK, and Bruno Bouzy. Progressive strategies for monte-carlo tree search, 2008.

[13] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[14] Alexander Dockhorn, Max Frick, Ünal Akkaya, and Rudolf Kruse. Predicting opponent moves for improving hearthstone ai. In *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pages 621–632. Springer, 2018.

[15] Alexander Dockhorn and Sanaz Mostaghim. Introducing the Hearthstone-AI Competition. pages 1–4, 5 2019.

[16] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, et al. Building watson: An overview of the DeepQA project. *AI magazine*, 31(3):59–79, 2010.

[17] Matthew C Fontaine, Scott Lee, LB Soros, Fernando De Mesentier Silva, Julian Togelius, and Amy K Hoover. Mapping hearthstone deck spaces through map-elites with sliding boundaries. In *Proceedings of The Genetic and Evolutionary Computation Conference*, pages 161–169, 2019.

[18] Pablo García-Sánchez, Alberto Tonda, Antonio J Fernández-Leiva, and Carlos Cotta. Optimizing hearthstone agents using an evolutionary algorithm. *Knowledge-Based Systems*, 188:105032, 2020.

[19] Pablo García-Sánchez, Alberto Tonda, Antonio M Mora, Giovanni Squillero, and Juan Julián Merelo. Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone. *Knowledge-Based Systems*, 153:133–146, 2018.

[20] Pablo García-Sánchez, Alberto Tonda, Giovanni Squillero, Antonio Mora, and Juan J Merelo. Evolutionary deckbuilding in hearthstone. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[21] David E. Goldberg. *Genetics Algorithms in Search, Optimization & Machine Learning*. Adison-Wesley Publishing Company, Inc., 1989.

[22] Randy L Haupt and Sue Ellen Haupt. *Practical genetic algorithms*. Wiley Online Library, 2 edition, 2004.

[23] Philip Hingston. A new design for a turing test for bots. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 345–350. IEEE, 2010.

[24] Jean-Baptiste Hoock, Chang-Shing Lee, Arpad Rimmel, Fabien Teytaud, Mei-Hui Wang, and Oliver Teytaud. Intelligent agents for the game of Go. *IEEE Computational Intelligence Magazine*, 5(4):28–42, 2010.

[25] Feng-Hsiung Hsu. *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton University Press, 2004.

[26] Ejnar Håkonsen. System - and AI design of a digital collectible card game. Master's thesis, University of Copenhagen, 2 2015.

[27] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[28] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1, 2006.

[29] Tze Leung Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[30] Alan Levinovitz. The mystery of Go, the ancient game that computers still can't win. *Wired Magazine*, 2014.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[32] J v Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1):295–320, 1928.

[33] Nisha Saini. Review of selection methods in genetic algorithms. 6:22261–22263, 12 2017.

[34] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[35] André Santos, Pedro A Santos, and Francisco S Melo. Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 272–279. IEEE, 2017.

[36] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[37] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21–21, 1996.

[38] Jacob Schrum, Igor V Karpov, and Risto Miikkulainen. Ut 2: Human-like behavior via neuroevolution of combat behavior and replay of human traces. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 329–336. IEEE, 2011.

[39] Noor Shaker, Julian Togelius, Georgios N Yannakakis, Likith Poovanna, Vinay S Ethiraj, Stefan J Johansson, Robert G Reynolds, Leonard K Heether, Tom Schumann, and Marcus Gallagher. The turing test track of the 2012 mario ai championship: entries and evaluation. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[40] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[41] Nitasha Soni and Tapas Kumar. Study of various mutation operators in genetic algorithms. *International Journal of Computer Science and Information Technologies*, 5(3):4519–4521, 2014.

[42] Maciej Świechowski, Tomasz Tajmajer, and Andrzej Janusz. Improving hearthstone ai by combining mcts and supervised learning algorithms. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[43] Gerald Tesauro. Practical issues in temporal difference learning. In *Advances in neural information processing systems*, pages 259–266, 1992.

[44] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[45] Alan M Turing. Digital computers applied to games. *Faster than thought*, 1953.

[46] Georgios N Yannakakis and Julian Togelius. *Artificial intelligence and games*, volume 2. Springer, 2018.

[47] Shuyi Zhang and Michael Buro. Improving hearthstone ai by learning high-level rollout policies and bucketing chance node events. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 309–316. IEEE, 2017.

[48] Jiren Zhu. Will our new robot overlords play hearthstone with us? *CS 229 Final Report*, 2016.