Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Sistema de Decisión Inteligente. Teoría de Juegos. Diseño, aplicación y evaluación.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Daniel Diosdado López

Tutor: Federico Barber Sanchís

Curso 2017 - 2018

Resumen

Este trabajo describe el diseño, implementación y evaluación de la aplicación de la técnica de *Monte Carlo Tree Search* (MCTS) al juego Gwent: The Witcher Card Game. Gwent es un complejo juego de cartas con información oculta y elementos estocásticos. A fin de realizar un sistema de computación capaz de jugar a este juego, con una cierta competencia, se desarrolla un simulador de Gwent y se implementa un agente capaz de jugar usando MCTS. Se aplica la técnica MCTS debido a la dificultad de definir una adecuada función de evaluación de estados, lo que es requerido en técnicas más convencionales de IA (basadas en Minimax) para su aplicación a juegos con adversario. Tras el desarrollo del método base, se propone una heurística para mejorar el funcionamiento del agente y se realizan pruebas para ajustar las prioridades de la heurística. Finalmente, a fin de evaluar los resultados obtenidos, se enfrenta al jugador heurístico con un jugador MCTS puro y se analizan los resultados. Los resultados obtenidos permiten concluir la adecuación del planteamiento realizado y permiten sugerir algunas alternativas de mejora. El desarrollo del TFG ha permitido aplicar diversos conocimientos adquiridos durante el grado en la resolución de un problema complejo con resultados contrastables.

Palabras clave: Inteligencia artificial, Teoría de juegos, Monte Carlo Tree Search, Gwent.

Abstract

This paper describes the design, implementation and evaluation of the application of the Monte Carlo Tree Search (MCTS) technique to the game Gwent: The Witcher Card Game. Gwent is a complex card game with hidden information and stochastic elements. In order to make a computer system capable of playing this game, with a certain competence, a Gwent simulator is developed and an agent capable of playing using MCTS is implemented. The MCTS technique is applied due to the difficulty of defining an adequate state evaluation function, which is required in more conventional AI techniques (based on Minimax) for its application to games with adversary. After the development of the base method, a heuristic is proposed to improve the performance of the agent and tests are carried out to adjust the priorities of the heuristic. Finally, in order to evaluate the results obtained, the heuristic player faces a pure MCTS player and the results are analyzed. The results obtained allow to conclude the adequacy of the approach taken and allow to suggest some alternatives for its improvement. The development of the TFG has allowed to apply various knowledge acquired during the degree in the resolution of a complex problem with testable results.

Keywords: Artificial Intelligence, Game Theory, Monte Carlo Tree Search, Gwent.

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

1.1. Contexto y motivación

La industria del videojuego es uno de los sectores del entretenimiento que más crecimiento ha experimentado en los últimos años. El sector del videojuego generó en el año 2017 unos ingresos de 121,7 mil millones de dólares en el mercado global (Wijman, 2018).

Los juegos presentan al jugador un desafío que debe superar, normalmente dentro de un tiempo límite. Por este motivo, crear una inteligencia artificial que sea capaz, no solo de resolver el desafío presentado por el juego, sino que lo haga dentro del tiempo límite, es una tarea mucho más complicada e interesante.

Dentro de los juegos, aquellos con adversario plantean una mayor dificultad ya que nuestras decisiones se ven influidas por las que tome el otro jugador, sea este un jugador real o un agente inteligente.

En los videojuegos, los pertenecientes a géneros como el RTS (*Real Time Strategy*), los CCG (*Collective Card Game*) o los juegos de estrategia en general, presentan características que facilitan la implementación de agentes inteligentes, así como medir la eficacia de los mismos. Estás características son:

- Reglas concretas y bien definidas
- Estrategias complejas
- Objetivos concretos
- Resultados medibles

El videojuego Gwent: The Witcher Game (CD PROJEKT RED, 2018) es un CCG desarrollado por la empresa CD Projekt Red en el que dos jugadores compiten entre ellos con barajas diseñadas por ellos mismos con las cartas del juego. Gwent es un juego estrictamente competitivo con las características anteriormente mencionadas, con un fuerte componente estratégico y de gestión de recursos: es tan importante decidir qué cartas jugar y en qué orden como saber cuándo pasar de ronda es la mejor opción. Además, los jugadores no pueden ver las cartas del rival hasta que son jugadas.

Por eso, considero que implementar un agente inteligente capaz de jugar a Gwent es un desafío interesante que me permitirá aprender y profundizar en la teoría de juegos.

1.2. Objetivos

A continuación, se enumeran los objetivos que el presente proyecto pretende abordar:

- Implementar un simulador del videojuego Gwent: The Witcher Card Game con sus principales características e implementar un número suficiente de cartas y de barajas, atendiendo a las restricciones temporales de realización del trabajo.

- Implementar un agente que juegue al Gwent razonablemente bien, usando para ello el método *Monte Carlo Tree Search* (MCTS en adelante).

- Diseñar e implementar una función heurística que guíe la fase de selección de MCTS hacia mejores estados.

- Ajustar la función heurística y analizar el impacto en el índice de victorias de cada peso de la función heurística.

1.3. Estructura de la memoria

En el apartado dos se describe el videojuego Gwent, se caracteriza y se muestra la nomenclatura del juego.

En el apartado tres describe el diseño de la solución, se introduce el método MCTS y se comentan sus fases.

En el apartado cuatro describe la implementación del simulador, la implementación del jugador que usa MCTS para decidir sus movimientos y se diseña una función heurística para mejorar la fase de selección de MCTS.

En el apartado cinco describe el ajuste de la función heurística y los resultados obtenidos al enfrentar al jugador heurístico con un jugador MCTS puro.

En el apartado seis se concluye y valora el trabajo.

2. Descripción del juego Gwent

Gwent: The Witcher Card Game es un juego de cartas coleccionables actualmente en desarrollo por CD Projekt Red. El juego tiene su origen en un juego de cartas presente en las novelas de Andrzej Sapkowski, The Witcher y de la versión jugable del Gwent en el videojuego The Witcher 3: Wild Hunt de CD Projekt Red, donde tan solo era un minijuego. A petición de los aficionados, que llegaron a crear versiones físicas y organizar torneos, CD Projekt Red decidió desarrollar el minijuego hasta convertirlo en un videojuego propio.

Gwent: The Witcher Card Game	
 *1 Logotipo del Gwent*	
Desarrollador	CD Projekt Red
Publicador	CD Projekt
Serie	The Witcher
Motor Gráfico	Unity
Plataformas	• Microsoft Windows • PlayStation 4 • Xbox One
Año de Salida	2018
Géneros	Juego de cartas coleccionables
Modos de juego	Un solo jugador, multijugador

Tabla 1 Información básica del videojuego Gwent

2.1. Jugabilidad

Gwent es un juego de cartas por turnos entre dos jugadores con información oculta, puesto que desconocemos las cartas que tiene nuestro rival hasta que las juegue o se muestren debido a la habilidad de una carta, con excepción de su carta de líder. La partida tiene un máximo de tres rondas. Los jugadores comienzan la partida con el mazo que hayan construido previamente. El mazo debe tener entre veintidós y cuarenta cartas, además de contener como máximo cuatro cartas de oro y seis de plata sin repeticiones, y hasta tres copias de las cartas de bronce. Esta limitación es debida a que las cartas doradas y plateadas tienen efectos más poderosos que las cartas de bronce. Cada mazo solo puede estar compuesto por cartas de la facción a la que pertenece la carta líder y cartas neutrales. Las cartas de líder son cartas de oro que se pueden jugar en cualquier momento, y cada una tiene una habilidad única.

Al comienzo de la partida, se roban diez cartas del mazo y se permite hacer tres cambios. En la segunda ronda, los jugadores roban 2 cartas y pueden cambiar una, y en la tercera ronda roban una carta. Además de estas cartas, también está

disponible desde el comienzo la carta del líder. Cada turno, el jugador correspondiente debe jugar una de sus cartas o pasar hasta el final de la ronda. En el caso de que el jugador no tenga cartas para jugar, se verá obligado a pasar. Ganará la ronda el jugador con más puntuación en el tablero en el momento en el que hayan pasado ambos jugadores, y el primero en ganar dos rondas gana la partida.

2.2. Campo de batalla

El campo de batalla se compone de tres filas por jugador: *Melee, Ranged y Siege*. En versiones anteriores de Gwent, la gran mayoría de cartas estaban limitadas a jugarse en una fila concreta, pero se decidió eliminar esta restricción de colocación en la versión 0.9.11 y permitir que todas las cartas se puedan colocar en cualquier fila.

Además de las filas del campo, también está el cementerio y, aunque no sea un elemento visible, las cartas desterradas. La diferencia principal entre las cartas que están en el cementerio y las desterradas es que las cartas del cementerio pueden resucitarse para devolverlas al campo con efectos de otras cartas, mientras que las cartas desterradas no pueden jugarse de ninguna forma en el resto de la partida.

2 Partida de Gwent

2.3. Nomenclatura de Gwent

A continuación, se adjunta la lista de todos los términos usados en el juego de cartas Gwent en orden alfabético y su significado:

Absorber	Infligir daño y potenciarse con la misma cantidad.
Apuesta	Jugar un juego al azar.
Armadura	Absorbe cierta cantidad de daño y luego se elimina.
Beneficios	Efecto positivo en la fila. Se sustituye por otros efectos de fila y se elimina al final de la ronda.
Bloqueo	Estado que inhabilita las habilidades de una carta y elimina otros estados. Inhabilita y da la vuelta a las unidades boca abajo.
Cautivar	Mover a un enemigo a la fila opuesta.
Consumir	Destruir una unidad comiéndosela. Si está en el cementerio, se elimina de la partida en su lugar.
Crear	Generar una de tres cartas elegidas al azar, excluyendo los agentes, de la facción de la carta y las cartas neutrales, a menos que se indique lo contrario
Curar	Restaurar el poder base de una unidad, si su poder actual es inferior.
Debilitar	Disminuir el poder base de una unidad. Si se reduce por debajo de 1, se destierra de la partida.
Descartar	Enviar una carta al cementerio desde la mano o baraja.
Desechable	La habilidad de esta carta solo se puede usar una vez por partida.
Desterrar	Eliminar de la partida.
Dotación	Esta unidad activa la habilidad de las unidades con Relevo jugadas en posiciones adyacentes a ella.
Duelo	Las unidades se turnan para infligirse un daño igual a su poder hasta que una de ellas es destruida.
Emboscada	Se juega boca abajo. Puede desactivarse y darse la vuelta si se bloquea.
Espía	Estado de una unidad que se juega en el lado opuesto del tablero
Exento	No puede servir de objetivo.
Fortalecer	Aumentar el poder base de una unidad
Generar	Añadir una carta a la partida y jugarla
Inferior	La unidad con menor poder. Los empates se resuelven al azar.
Intercambiar	Mover una carta de la mano a la baraja y sacar otra en su lugar.
Invocar	Jugar automáticamente desde la baraja.
Ocultar	Dar la vuelta a una carta boca arriba de la mano.
Peligro	Efecto negativo en la fila. Se sustituye por otros efectos de fila y se elimina al final de la ronda.
Relevo	Cuando se juega, se activa esta habilidad una vez por cada unidad adyacente con Dotación.
Resistente	Estado que permite a una unidad permanecer en el tablero al final de la ronda y restablece su poder base si está potenciada.
Restablecer	Restaurar el poder base de una unidad.
Resucitar	Jugar desde tu cementerio.
Revelada	Una carta de la mano a la que se le ha dado la vuelta.
Revelar	Dar la vuelta a una carta de la mano.
Sueño mortal	Activa esta habilidad cuando se destruye durante una ronda.
Superior	La unidad con mayor poder. Los empates se resuelven al azar.
Tregua	No ha pasado ningún jugador.

Tabla 2 Vocabulario del videojuego Gwent

2.4. Versión implementada

Debido a que Gwent: The Witcher Card Game es un juego en desarrollo, se ha tomado el juego en su implementación 0.9.23.5.430.3, al ser esta la última versión disponible en el momento de comenzar con el proyecto, y debido a limitaciones de tiempo, se han implementado las características más representativas del juego. Los detalles completos de la implementación se proporcionan en el apartado de implementación.

La elección de esa versión de Gwent se debe, en primer lugar, a que es posterior a las actualizaciones que introdujeron grandes cambios en el juego: por un lado, la eliminación de la inmunidad general a los efectos de las cartas de oro, y por otro la eliminación de las restricciones de colocación de todas las cartas, que antes estaban limitadas a poder colocarse en una fila.

2.5. Caracterización de Gwent

Aquí se va a caracterizar el juego como un problema con las siguientes características:

- Información imperfecta: Ningún jugador tiene acceso a toda la información de la partida. Los jugadores no conocen la composición de las barajas de sus rivales.

- Resultados estocásticos: Hay bastante incertidumbre en todas las fases de la partida. Desde las cartas que se roban al inicio de cada ronda, al resultado de efectos aleatorios de cartas. Esto crea incertidumbre y la imposibilidad de controlar el resultado, además de aumentar mucho el coste computacional la simulación de todos los posibles resultados.

- Parcialmente observable: Los jugadores no pueden ver la mano ni la baraja del rival en condiciones normales hasta que juegan esa carta, salvo por efectos que revelen cartas. Además, al finalizar la partida no se revelan las cartas del adversario, lo que dificulta el aprendizaje de las estrategias de los rivales al no poder conocer la composición de su baraja.

- Suma cero: Gwent es un juego competitivo para dos jugadores. Toda ventaja de un jugador es una pérdida para el otro y viceversa.

- Complejidad: al haber cientos de cartas en Gwent, la complejidad de la simulación puede aumentar exponencialmente dependiendo de qué cartas se añadan a la simulación. Además, se añaden cartas nuevas al juego de forma periódica.

- Por turnos: El juego se divide en turnos de forma secuencial y los jugadores van jugando cartas de forma alternativa hasta que termina la partida o pasan.

- No reactivo: Los jugadores no pueden reaccionar a la jugada del rival durante el turno de éste.

- Cronometrado: Cada turno de una partida de Gwent entre dos jugadores tiene un límite de tiempo de un minuto para jugar una carta o pasar. Si el jugador no juega ninguna carta durante ese minuto, pasan automáticamente hasta la siguiente ronda.

- Finito: Al haber una cantidad limitada de cartas para poder jugar, es imposible que la partida dure infinitamente.

2.6. Estimación de movimientos posibles

Dado un estado de juego el número de posibles jugadas que puede hacer el jugador dependerá de si las cartas en mano necesitan posición y/u objetivo para su efecto, y, si tienen un efecto que juegue otra carta (resurrección, invocar o crear), de si la segunda carta necesita posición y/u objetivo; las posiciones dependerán de la lealtad de la carta, es decir, si se juega en nuestro campo o en el del rival, y los objetivos dependerán de la zona de efecto de los mismos y del alcance (una carta, una fila, el tablero rival,…).

El número posible de movimientos de una carta puede definirse como:

Movimientos(i) = Posición(i) X Objetivos(i)

El número de posiciones posibles se puede calcular mediante siguiente formula, dependiendo de si la carta es especial o no:

Posición(i) =

Especial: 1

No Especial: 3 + N, siendo N el número de cartas en el lado del tablero en el que se juega la carta.

El número de objetivos posibles se puede calcular mediante la siguiente formula, dependiendo del alcance del efecto:

Objetivos(i) =

Sin efecto: 1

Efecto aleatorio a 1 carta: X, siendo X el número de cartas dentro del alcance a las que puede afectar el efecto.

Efecto aleatorio a N cartas: Todas las posibles combinaciones o permutaciones (dependiendo del efecto) de X cartas en la zona de

alcance con repetición, siendo X el número de cartas a las que puede afectar el efecto.

Carta: Número de cartas en la zona de efecto

Fila: 3

Fila opuesta: 1

Tablero: 1 o 2, dependiendo de si afecta a todo el tablero o solo a un lado.

Efectos *Resurrect*, *Summon*, *Spawn* o *Create*:

$$\sum_{x}^{N} Movimientos(x)$$

Siendo N el conjunto de cartas dentro del alcance del efecto y a las que puede afectar. Hay que tener en cuenta que el número de cartas en el tablero puede haber variado: Si colocamos una carta no especial en el tablero cuyo efecto es resucitar una carta de bronce no especial, las posiciones donde se puede colocar la carta resucitada son distintas, puesto que ya hemos colocado la primera carta. A efectos de calcular los movimientos posibles, sea A la primera carta y B la segunda, Posición(A) + 1 = Posición(B).

Por tanto, el número posible de movimientos del jugador 2 vendrá dado por:

$$\sum_{i}^{N} (Movimientos(i)) + 1$$

Siendo N las cartas en la mano del jugador e i la i-ésima carta. Se suma al final puesto que los jugadores siempre pueden pasar.

Por ejemplo, el número de movimientos que puede ejecutar el jugador que juega el primer turno de la partida si tiene las siguientes cartas:

Carta Líder (Rey Foltest)

7 Unidades: 2 Invocan copias iguales

3 Dañan cartas rivales

1 se pone armadura a si misma

1 carta no tiene efecto

3 Especiales: 1 Dañan una en el tablero

1 Invoca una carta de oro

1 Aumenta el valor de 5 cartas adyacentes en nuestro campo.

Al ser el primer turno de la partida, el tablero está vacío. Por tanto, hay tres posiciones para colocar la carta líder o las unidades, y todos los efectos que se apliquen a otras cartas no tienen objetivo, aunque para el cálculo consideramos que tienen un objetivo, nadie.

A continuación, se muestran los cálculos de todos los posibles movimientos que puede hacer el jugador en este turno, descompuestos por tipo de carta:

Líder: Movimientos(Líder) = 3 x 1 = 3

Unidades: Invocan: Movimientos(carta) = 3 x 1 x 4 x 1 x 5 x 1 = 60

Dañan: Movimientos(carta) = 3 x 1 = 3

Armadura: Movimientos(carta) = 3 x 1 = 3

Sin efecto: Movimientos(carta) = 3 x 1 = 3

Especiales: Dañan: 1 x 1 = 1

Invoca: $\sum$ (3 x Y)

Aumenta: 1 x 1 = 1

Puesto que la carta especial invoca una carta de oro de la baraja, se asume en este caso que quedan 2 cartas de oro en la baraja y ambas son unidades. Asumiremos en este ejemplo que una daña a una carta rival y otra aumenta el valor de una carta aliada, quedando el cálculo así:

Invoca: 3 x 1 + 3 x 1 = 6

Por tanto, el número de posibles movimientos para este jugador con estas cartas es de 146 movimientos.

Supongamos ahora que tras varios turnos este jugador ha jugado una unidad con efecto de armadura y otra sin efecto, y el rival ha jugado dos unidades y ninguna ha sido destruida. Ahora, el cálculo quedaría así:

Líder: Movimientos(Líder) = 5 x 1 = 5

Unidades: Invocan: Movimientos(carta) = 5 x 1 x 6 x 1 x 7 x 1 = 210

Dañan: Movimientos(carta) = 5 x 2 = 10

Especiales: Dañan: 1 x 2 = 2

Invoca: 5 x 2 + 5 x 2 = 20

Aumenta: 1 x 2 = 2, o 1 x 1, dependiendo de si hemos colocado las cartas en la misma fila o no.

Ahora el jugador puede ejecutar 479 movimientos diferentes (478 si ambas cartas aliadas se colocaron en la misma fila).

3. Diseño de la solución

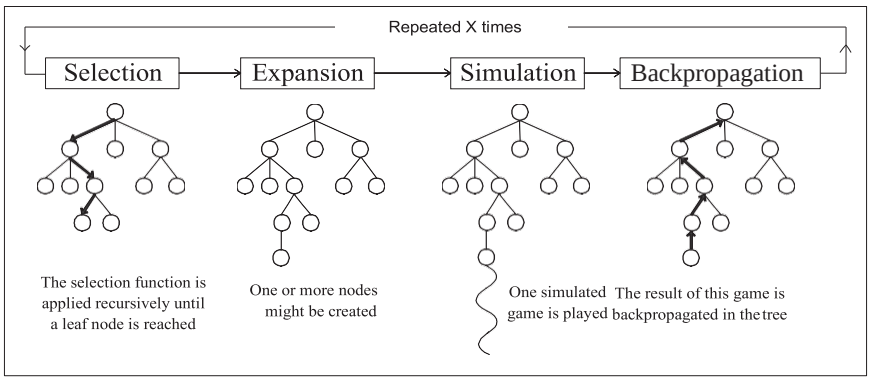
3.1. Introducción a *Monte Carlo Tree Search*

Árbol de búsqueda Monte Carlo, o *Monte Carlo Tree Search* (Abramson, 1987)es un método de toma óptima de decisiones en un dominio dado, usando los resultados de simulaciones aleatorias para construir un árbol de búsqueda.

La aplicación de este método se puede aplicar a otros campos, puesto que MCTS puede aplicarse a cualquier dominio que se pueda describir en pares (estado, acción) y pueda simularse para predecir resultados.

3.2. Funcionamiento de *Monte Carlo Tree Search*

MCTS se compone de cuatro etapas, que se repiten de forma cíclica mientras la búsqueda no supere la restricción temporal. Estas cuatro etapas son:



3 Algoritmo MCTS

- Selección: Se recorre el árbol desde el nodo raíz seleccionando uno de los nodos hijos mediante la estrategia de selección hasta alcanzar un nodo que no haya sido explorado completamente.

- Expansión: Se añaden nodos hijo al nodo seleccionado según la estrategia de expansión.

- Simulación: Se simula una partida completa partiendo del estado del nodo hoja, seleccionando los movimientos de forma aleatoria o pseudo-aleatoria.

- Retropropagación: El resultado de la simulación se propaga desde el nodo hoja hasta el nodo raíz pasando por todos los padres y actualiza el número de veces que se ha visitado el nodo y el número de victorias de dicho nodo.

Tras finalizar la construcción del árbol y agotar el tiempo disponible para decidir, se debe seleccionar el mejor movimiento que puede hacer el jugador según el árbol que ha construido, dependiendo de qué estrategia de selección del mejor movimiento se emplee para ello.

3.3. Selección

La etapa de selección consiste en recorrer el árbol de búsqueda hasta encontrar el nodo hoja más prometedor para expandir. El proceso comienza en el nodo raíz y va seleccionando de forma recursiva el nodo hijo más prometedor mediante la estrategia de selección hasta llegar a un nodo que no haya sido explorado completamente, es decir, un nodo en el que aún queden posibles movimientos por expandir.

Las estrategias de selección se enfocan generalmente en balancear la exploración y la explotación, entendiendo por exploración seleccionar más los nodos que menos se hayan visitado y por explotación seleccionar más los nodos más prometedores. Una estrategia de selección bastante común es *Upper Confidence Bounds for Trees* (UCT en adelante) (Kocsis, 2006) por ser simple y eficiente.

$$\frac{v_i}{n_i} + C * \sqrt{\frac{\ln N}{n_i}}$$

Donde v_i es el número de victorias del nodo, n_i es el número de veces que el nodo ha sido visitado, N el número de veces que se ha visitado el nodo padre y C es el parámetro que ajusta la exploración y la explotación.

3.4. Expansión

La etapa de expansión de MCTS consiste en añadir nuevos nodos al árbol de búsqueda, después de determinar el nodo a expandir en la etapa de selección.

La cantidad de nodos que se añaden al árbol depende de la estrategia de expansión, siendo las más comunes expandir un único nodo de los posibles movimientos del nodo seleccionado aun sin explorar, normalmente de forma aleatoria; o expandir todos los posibles movimientos desde el nodo seleccionado. La primera estrategia gasta menos memoria que la segunda a cambio de un ligero decremento en el nivel de juego.

En general, el efecto de las estrategias de expansión en el nivel de juego es pequeño, y la estrategia de expandir un solo nodo es suficiente en la mayoría de los casos (Chaslot, 2010).

3.5. Simulación

La simulación es la etapa en la que se simula una partida completa desde el estado de juego actual, eligiendo los movimientos de ambos jugadores de forma totalmente aleatoria o pseudo-aleatoria siguiendo una estrategia de simulación.

Una estrategia de simulación totalmente aleatoria es rara vez óptima, puesto que asume que ambos jugadores juegan aleatoriamente. Ha sido demostrado que el uso de una estrategia de simulación pseudo-aleatoria mejora de forma significativa el nivel de juego (Sylvain, 2006). A pesar de esto, una estrategia de simulación totalmente aleatoria, con un número suficiente de simulaciones, aproxima un nivel de juego avanzado.

Aplicar conocimiento del dominio a la estrategia de simulación ocasiona un intercambio entre la velocidad y la precisión de la simulación; se reduce la aleatoriedad de la simulación y, por tanto, se vuelve más realista a cambio de un mayor coste computacional, ocasionado por la heurística que guía la simulación.

También se produce un intercambio entre exploración y explotación: Si no hay estrategia o la estrategia es muy aleatoria, se beneficia la exploración. Si la estrategia es muy determinista la simulación pierde variabilidad y las simulaciones estarán sesgadas por la heurística que se emplee.

3.6. Retropropagación

La etapa de retropropagación consiste en actualizar los valores de número de visitas y número de victorias de los nodos, desde el nodo expandido hasta el nodo raíz, pasando por todos los nodos padre entre ambos. El valor de victorias depende del resultado de simulación, siendo generalmente 1 para victoria, 0 para derrota y 0.5 para el empate.

3.7. Selección del mejor movimiento

Una vez se agota el tiempo disponible para decidir el movimiento, la búsqueda termina y MCTS debe seleccionar el mejor nodo hijo del nodo raíz, usando los valores del número de visitas y de victorias para ello.

Las estrategias más comunes para seleccionar el mejor nodo hijo son:

- *Max Child*: Seleccionar el nodo hijo con mayor número de victorias.
- *Robust Child*: Seleccionar el hijo con mayor número de visitas.

- *Max-Robust Child*: Seleccionar el hijo con mayor número de victorias y visitas.

La estrategia más común es *Robust Child*, puesto que MCTS visitará el nodo más prometedor el máximo número de veces, siempre y cuando el sesgo se haya ponderado de forma correcta (Browne, 2012). *Max Child* debería ser equivalente a *Robust Child* debido a la explotación en UCT, sin embargo, *Max Child* con pocas iteraciones selecciona peores movimientos que las otras opciones (Chaslot, 2007). *Max-Robust Child* es una buena opción, pero no hay garantía de que exista tal nodo hijo y obtener dicho nodo puede requerir una cantidad de tiempo indeterminada.

3.8. Características

Las principales características de MCTS son:

- Aheurístico: MCTS tiene la ventaja de no depender de una función de evaluación heurística en su forma básica, y esto le permite poder ser aplicado a cualquier dominio que se pueda modelar como un árbol sin requerir ningún conocimiento específico del dominio en sí para funcionar.

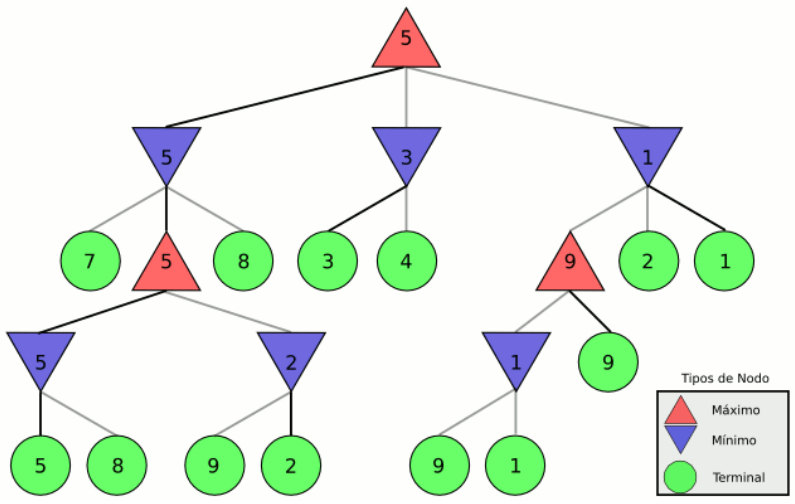
- Asimétrico: La función de selección de MCTS hace que los mejores movimientos se evalúen más que los peores movimientos.

- En cualquier momento: Puesto que tras cada ciclo de ejecución del algoritmo se retropropagan los resultados, MCTS puede decidir el mejor movimiento con el árbol que haya generado hasta el momento.

3.9. Ventajas respecto a otros algoritmos

Otras técnicas usadas típicamente en juegos son Minimax y la poda Alfa-Beta (Russell, 2008).

Minimax establece que, en los juegos de dos jugadores de suma cero e información perfecta, existe una estrategia que permite a ambos jugadores minimizar la pérdida máxima esperada. En concreto, para cada posible movimiento de un jugador, este debe considerar todas las posibles respuestas del adversario y la perdida máxima que puede acarrear cada uno de sus movimientos. El jugador elige el movimiento que minimiza la máxima pérdida. Esta estrategia es óptima para ambos jugadores si sus valores Minimax son iguales en valor absoluto y contrarios en signo. Estos valores se obtienen con una función heurística que valora los estados terminales del juego. A continuación se adjunta una imagen para ilustrar el funcionamiento del algoritmo.

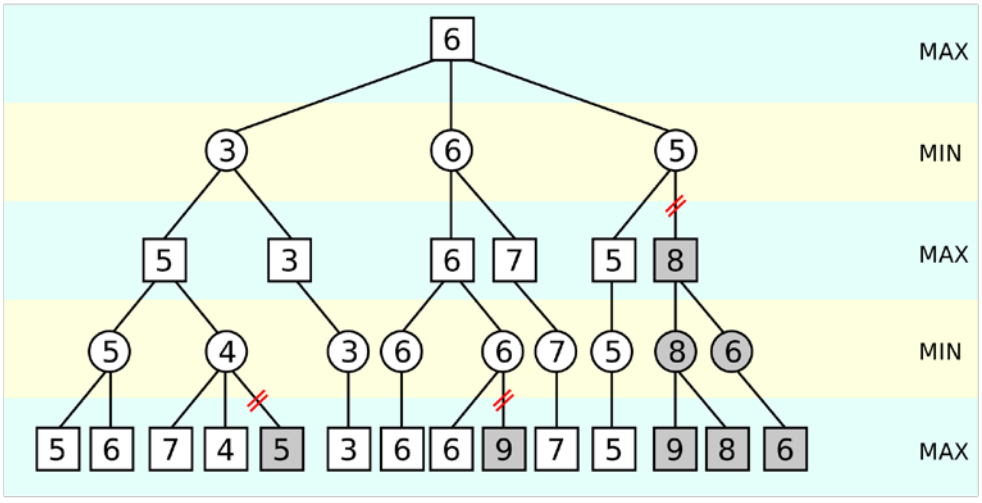


4 Algoritmo Minimax

La poda Alfa-Beta consiste en reducir el número de nodos evaluados en el árbol generado por el algoritmo Minimax podando ramas que no pueden mejorar los valores obtenidos hasta el momento. Para ello usa dos parámetros:

- α: es el valor de la mejor opción hasta el momento a lo largo del camino para MAX, esto implicará por lo tanto la elección del valor más alto
- β: es el valor de la mejor opción hasta el momento a lo largo del camino para MIN, esto implicará por lo tanto la elección del valor más bajo.

Durante la búsqueda se van actualizando los parámetros conforme se recorre el árbol y el método realizará la poda cuando el valor que se esté examinando sea peor que α para Max o que β para Min. Se adjunta a continuación una imagen para ilustrar la poda de ramas del árbol de juego.



5 Poda Alfa-Beta

A diferencia de estos algoritmos, MCTS no necesita una función de evaluación de estados al ser aheurístico en su forma base, puesto que la exploración del árbol usa los resultados obtenidos hasta el momento para guiarse, lo que lo convierte en un método especialmente interesante para dominios donde es complicado evaluar los estados.

4. Implementación

4.1. Elección del lenguaje de programación Python

Se decidió emplear Python para la realización de este trabajo por la familiaridad con el mismo y la experiencia adquirida durante los estudios, así como la facilidad para leer y escribir código en Python, dado que su estructura se asemeja bastante al lenguaje natural, y la abundancia de bibliotecas disponibles para Python.

Otro factor importante a la hora de escoger Python para el proyecto fue la gran base de usuarios existente y la facilidad para acceder a la documentación del lenguaje.

4.2. Implementación del simulador

Al no existir un simulador de Gwent: The Witcher Card Game de la versión actual en el momento de comenzar el proyecto, ni de una versión posterior a la actualización del 9 de febrero de 2018 (correspondiente a la versión de Gwent 0.9.20) donde se introdujeron grandes cambios en el juego, se decidió implementar un simulador de Gwent que pudiese representar las principales características del juego.

Con este fin, se han implementado dos clases para el simulador:

- Clase CartaGwent: clase que representa una carta con todos sus atributos en el simulador.

- Clase EstadoGwent: clase que representa un estado de juego del Gwent, así como las funciones necesarias para poder implementar el árbol de búsqueda Monte Carlo.

4.2.1. Clase CartaGwent

La clase CartaGwent se usa en el simulador para representar cada una de las cartas que hay en la partida. Se han implementado los siguientes atributos en las cartas de Gwent:

- Id: El id es un número que se asigna en el momento en el que se inicializa la partida y que sirve para diferenciar las cartas que son idénticas en la partida.
- Nombre: El nombre de la carta.
- Etiquetas: Las etiquetas de la carta. Necesarias para el correcto funcionamiento de algunos efectos que se limitan a cartas con ciertas etiquetas.
- Facción: La facción de la carta.

- V_Original: Valor original de la carta. Se asigna en el momento de creación de la carta y no se vuelve a modificar.
- V_Base: Valor base de la carta. Se asigna en el momento de creación y puede ser modificado por los efectos Weaken o Strengthen.
- V_Actual: Valor actual de la carta. Puede ser modificado por los efectos Damage, Boost, Weaken o Strengthen.
- V_Armadura: Valor de armadura de la carta. Se inicializa a cero y solo puede ser modificado por los efectos Armor o Damage.
- Color: El color de la carta. Las cartas pueden ser de oro, plata o bronce.
- Especial: Indica si la carta es especial o no.
- Leal: Indica si la carta es leal o no, es decir, si la carta se juega en el campo del propio jugador o en el campo del contrario.
- Resistente: Indica si la carta es resistente. Se inicializa a cero y solo puede ser modificado por el efecto Resilience o por la finalización de una ronda de la partida.
- Exento: Indica si la carta es exenta o no.
- Izq: Se inicializa a None. Indica que carta tiene este a su izquierda en la fila del tablero.
- Der: Se inicializa a None. Indica que carta tiene este a su derecha en la fila del tablero.
- Efecto: Contiene el efecto de la carta, así como su zona de efecto.

Se ha implementado una función auxiliar para cambiar el atributo Id de la carta, el cual se asigna a cada carta al comienzo de la partida.

```
1.  class CartaGwent:
2.      def __init__(self, identificador, nombre, faccion, etiquetas, valor_base, color, especial, lealtad,
    exento, dotacion, relevo, desechable, efecto):
3.          self.Id = identificador
4.          self.Nombre = nombre
5.          self.Etiquetas = etiquetas
6.          self.Faccion = faccion
7.          self.V_Original = valor_base
8.          self.V_Base = valor_base
9.          self.V_Actual = valor_base
10.         self.V_Armadura = 0
11.         self.Color = color
12.         self.Especial = especial
13.         self.Leal = lealtad
14.         self.Resistente = 0
15.         self.Exento = exento
16.         self.izq = None
17.         self.der = None
18.         self.Efecto = efecto
19.
20.     def __repr__(self):
21.         return self.Nombre
22.
23.     def change_id(self, new_id):
24.         self.Id = new_id
```

Tabla 3 Fragmento de código de CartaGwent

4.2.2. Clase EstadoGwent

La clase EstadoGwent es la clase principal del simulador. Representa el estado de la partida de Gwent e implementa todos los atributos necesarios para ello, así como las funciones necesarias para poder aplicar el algoritmo MCTS de forma correcta.

Se han implementado los siguientes atributos para cada jugador, siendo N 1 o 2:

- Mano_posN: Contiene todas las posibles cartas que podría tener el jugador N.
- Lider_N: Contiene la carta líder del jugador N. Cuando se usa el líder se deja a None.
- Mano_N: Contiene las cartas que tiene en mano el jugador N.
- N_Cartas_Mano_N: El número de cartas que le quedan en la mano al jugador N. Se inicializa a 11.
- Baraja_Actual_N: La baraja actual del jugador N.
- Cementerio_N: El cementerio del jugador N.
- Destierro_N: Las cartas desterradas del jugador N.
- Melee_N: La fila Melee del lado del jugador N.
- Ranged_N: La fila Ranged del lado del jugador N.
- Siege_N: La fila Siege del lado del jugador N.
- P_Melee_N: La puntuación de la fila Melee del jugador N.
- P_Ranged_N: La puntuación de la fila Ranged del jugador N.
- P_Siege_N: La puntuación de la fila Siege del jugador N.
- Puntuacion_N: La puntuación total del jugador N.
- Rondas_Ganadas_N: El número de rondas que ha ganado el jugador N a lo largo de la partida.
- Pasar_N: Indica si el jugador ha pasado en la ronda actual o no.
- Inicio_Turno_N: Lista que contiene los efectos que se ejecutan en el inicio de turno del jugador N, antes de que juegue.
- Fin_Turno_N: Lista que contiene los efectos que se ejecutan al final del turno del jugador N, después de que juegue.

Además, se han implementado los siguientes atributos:

- CONST_LIMIT_CARDS: El número máximo de cartas por fila del tablero. Es un valor constante que no cambia a lo largo de la partida. Se inicializa a 9.
- Ronda: Indica el número de ronda actual. Necesario para saber cuántas cartas roban los jugadores al finalizar la ronda.
- Turno: Indica a qué jugador le corresponde el turno.
- Player: Indica la identidad del jugador que ejecuta Monte Carlo.

```
class EstadoGwent:
    def __init__(self, bar_ini_1, bar_ini_2, player_id, list_all_pos_cards1, list_all_pos_cards2):

        i = 0
        for card in bar_ini_1:
            card.change_id(i)
            i += 1
        self.Last_Id1 = i

        i = 100
        for card in bar_ini_2:
            card.change_id(i)
            i += 1
        self.Last_Id2 = i

        i = 200
        for card in list_all_pos_cards1:
            card.change_id(i)
            i += 1
        self.Last_Id_Pos1 = i

        i = 300
        for card in list_all_pos_cards2:
            card.change_id(i)
            i += 1
        self.Last_Id_Pos2 = i

        self.Mano_pos1 = copy.deepcopy(list_all_pos_cards1)
        self.Mano_pos2 = copy.deepcopy(list_all_pos_cards2)

        self.Baraja_Inicial_1 = copy.deepcopy(bar_ini_1)
        self.Baraja_Inicial_2 = copy.deepcopy(bar_ini_2)

        self.Lider_1 = bar_ini_1[0]
        self.Lider_2 = bar_ini_2[0]

        bar_ini_1.pop(0)
        bar_ini_2.pop(0)

        self.Mano_1 = []
        self.Mano_2 = []

        for i in range(10):
            choice = random.choice(bar_ini_1)
            self.Mano_1.append(choice)
            bar_ini_1.remove(choice)

        for i in range(10):
            choice = random.choice(bar_ini_2)
            self.Mano_2.append(choice)
            bar_ini_2.remove(choice)


        self.N_Cartas_Mano_1 = 11
        self.N_Cartas_Mano_2 = 11


        self.Baraja_Actual_1 = list(bar_ini_1)
        self.Baraja_Actual_2 = list(bar_ini_2)
        random.shuffle(self.Baraja_Actual_1)
        random.shuffle(self.Baraja_Actual_2)

        self.Cementerio_1 = []
        self.Cementerio_2 = []
        self.Destierro_1 = []
        self.Destierro_2 = []
```

```
67.
68.         self.Melee_1 = []
69.         self.Melee_2 = []
70.         self.Ranged_1 = []
71.         self.Ranged_2 = []
72.         self.Siege_1 = []
73.         self.Siege_2 = []
74.
75.         self.CONST_LIMIT_CARDS = 9
76.
77.         self.P_Melee_1 = 0
78.         self.P_Melee_2 = 0
79.         self.P_Ranged_1 = 0
80.         self.P_Ranged_2 = 0
81.         self.P_Siege_1 = 0
82.         self.P_Siege_2 = 0
83.
84.         self.Puntuacion_1 = 0
85.         self.Puntuacion_2 = 0
86.
87.         self.Ronda = 1
88.
89.         self.Rondas_Ganadas_1 = 0
90.         self.Rondas_Ganadas_2 = 0
91.
92.         self.Pasar_1 = 0
93.         self.Pasar_2 = 0
94.
95.         self.Turno = random.randint(1, 2)
96.         self.Player = player_id
97.
98.         self.Inicio_Turno_1 = []
99.         self.Inicio_Turno_2 = []
100. self.Fin_Turno_1 = []
101. self.Fin_Turno_2 = []
```

Tabla 4 Fragmento de código de EstadoGwent: inicialización

Las funciones necesarias para poder aplicar MCTS son:

- __init__: Recibe las barajas reales de los jugadores y las barajas posibles de ambos, asigna un id a cada carta e inicializa el estado de la partida. El turno del primer jugador se asigna aleatoriamente, así como las diez cartas que se roban al principio de la partida.
- Clone: Devuelve una copia profunda del estado actual.
- Change_player: Este método sirve para cambiar la identidad del jugador que está ejecutando Monte Carlo, necesario para distinguir si se obtienen los movimientos de un jugador de su mano real o de su mano posible.
- Play: Este método recibe un movimiento como parámetro y lo ejecuta. Comprueba, en caso de que el movimiento sea Pasar, si el rival ha pasado también y por tanto ha finalizado la ronda.
- Get_moves: Este método devuelve todos los posibles movimientos que puede ejecutar el jugador al que le corresponde el turno. Usa el atributo del estado Player_Id para determinar si debe devolver los movimientos reales, en el caso de que el turno y jugador coincidan, o los

movimientos posibles, en caso de que se esté simulando los movimientos del jugador contrario.

- Get_random_move: Este método se implementó para mejorar la eficiencia de la simulación. En lugar de obtener todos los posibles movimientos de un estado, elige una carta de la mano del jugador al azar, genera todos los posibles movimientos de esa carta y elige uno al azar.
- Result: Este método devuelve el resultado de un estado terminal desde la perspectiva del jugador que está ejecutando Monte Carlo, devolviendo 1 si gana la partida en dicho estado terminal, 0 si pierde la partida y 0.5 si empata.

```
1.  def clone(self):
2.      s = copy.deepcopy(self)
3.      return s
4.
5.  def result(self, player):
6.      if self.Rondas_Ganadas_1 == 2 and self.Rondas_Ganadas_2 == 2:
7.          return 0.5
8.      elif player == 1 and self.Rondas_Ganadas_1 == 2:
9.          return 1
10.     elif player == 2 and self.Rondas_Ganadas_2 == 2:
11.         return 1
12.     else:
13.         return 0
```

Tabla 5 Fragmento de código de EstadoGwent: funciones clone y result

También se han implementado las siguientes funciones auxiliares, necesarias para la correcta simulación de una partida de Gwent:

- Get_all_possible_actions: función que obtiene todas las posibles acciones de la lista de cartas que recibe como parámetro.
- Colocar_carta_mano_fila: función que elimina la carta de la mano del jugador correspondiente y la coloca en el tablero mediante la función colocar_carta.
- Colocar_carta: función que coloca la carta en la posición indicada como parámetro e invoca a la función ejecutar_efectos si la carta tiene efecto.
- Usar_especial: función invoca a ejecutar_efectos y que envía la carta especial al cementerio.
- Pasar: cambia el atributo Pasar del jugador que corresponda a 1.
- Usar_lider: Usa el líder del jugador que corresponda, actualiza el atributo lider del jugador a None y lo coloca en el campo mediante la función colocar_carta.
- Ejecutar_efectos: Ejecuta los efectos que correspondan al objetivo que se le pase como parámetro.
- Turn_start: Ejecuta todos los efectos guardados en la lista Inicio_Turno del jugador correspondiente.
- Turn_end: Ejecuta todos los efectos guardados en la lista Fin_Turno del jugador correspondiente.

- Round_end: comprueba quien ha ganado la ronda, modifica el turno para que la siguiente ronda la empiece el ganador de la ronda y vacía el tablero de cartas enviándolas al cementerio, a menos que sean resistentes, en cuyo caso elimina su resistencia.
- Get_all_positions: Devuelve todas las posiciones existentes en el lado del tablero que corresponda.
- Get_all_possible_targets: Devuelve todos los posibles objetivos para el efecto y la zona de efecto que recibe como parámetro.
- Leftmost_card: Devuelve la carta más a la izquierda de la fila, si hay cartas en la fila. Función auxiliar de get_all_positions.
- Clear_hazard_or_boon: Elimina efectos Hazard o Boon de la lista de Inicio_Turno o Fin_turno del jugador que corresponda y que afectasen a la fila que reciba como parámetro.
- Hazard_or_boon: Añade un efecto Hazard o Boon a la lista De Inicio_Turno o Fin_Turno del jugador que corresponda que afecte a la fila que se le pasa como parámetro.
- Armadura: Aumenta el valor de armadura de la carta que recibe como parámetro.
- Resilience: Activa la resistencia de la carta que recibe como parámetro.
- Damage: Reduce el valor actual de la carta que recibe como parámetro, y si este es menor a 0 destruye la carta.
- Weaken: Reduce el valor base de la carta que recibe como parámetro, y si este es menor a 0 destierra la carta.
- Boost: aumenta el valor actual de la carta que recibe como parámetro.
- Strengthen: aumenta el valor base de la carta que recibe como parámetro.
- Heal: Iguala el valor actual al valor base, en el caso de que el actual sea menor al base, de la carta que recibe como parámetro.
- Reset: Iguala el valor actual al valor base de la carta que recibe como parámetro.
- Destruir_carta: Destruye la carta que recibe como parámetro, actualiza los atributos izq y der de las cartas que estuviesen al lado de esta en el caso de que las hubiese, iguala el valor actual al valor base, elimina la carta del tablero y la añade al cementerio correspondiente.
- Banish: Destierra la carta que recibe como parámetro, actualizando los parámetros izq y der de las cartas que estuviesen al lado de esta en el caso de que las hubiese, iguala el valor actual al base, elimina la carta del tablero y la añade a la lista de cartas desterradas que corresponda.
- Resurrect: Resucita una carta del cementerio y la añade al tablero en la posición indicada, y ejecuta el efecto de esta carta en el objetivo que recibe como parámetro.
- Summon: Invoca una carta de la baraja y la añade al tablero en la posición indicada, y ejecuta el efecto de esta carta en el objetivo que recibe como parámetro.

4.3. Implementación de MCTS

Se ha implementado, para la búsqueda en árbol Monte Carlo, la clase Nodo para representar los nodos en el árbol de búsqueda y una función para generar el árbol de búsqueda de Monte Carlo.

4.3.1. Nodo

La clase nodo tiene los siguientes atributos:

- Movimiento: Contiene el movimiento que nos conduce a este estado desde el nodo padre. En el nodo raíz del árbol es None.
- Nodo_Padre: Referencia al nodo padre de este nodo. En el nodo raíz del árbol es None.
- Nodos_Hijos: Una lista con referencias a todos los nodos hijos que se han expandido previamente. Si el nodo es una hoja del árbol entonces la lista estará vacía.
- Victorias: Número de victorias obtenidas en las simulaciones de este nodo y de todos sus hijos.
- Visitas: Número de veces que se ha explorado este nodo y sus hijos.
- Movimientos_sin_probar: Lista que contiene todos los movimientos posibles desde este estado de la partida.

Se han añadido los siguientes atributos para poder implementar el cálculo de la función heurística (explicada en el apartado 4.2.2.1):

- Player_id: Identidad del jugador que ejecuta la simulación.
- Valor_heuristica: valor que la función heurística asigna a este nodo.

```
1.  def __init__(self, heuryes_or_no=None, movimiento=None, padre=None, estado=None, alpha1=
    None, alpha2=None, alpha3=None, alpha4=None, alpha5=None, alpha6=None):
2.          self.movimiento = movimiento
3.          self.Nodo_Padre = padre
4.          self.Nodos_Hijos = []
5.          self.victorias = 0
6.          self.visitas = 0
7.          self.movimientos_sin_probar = estado.get_moves()
8.          self.player_id = estado.Player
9.          if heuryes_or_no == 1:
10.             self.valor_heuristica = self.heuristica(estado.Player, estado, alpha1, alpha2, alpha3, alpha
    4, alpha5, alpha6)
```

Tabla 6 Fragmento de código de Nodo: inicialización

Se han implementado las siguientes funciones:

- UCTSeleccionarHijo: Función que usa UCT o UCT y la función heurística para seleccionar un nodo hijo.
- AñadirHijo: Función que crea el nodo hijo y lo añade al árbol de búsqueda.
- Update: Función para actualizar el número de visitas y de victorias del nodo.
- Heurística: Función que calcula el valor heurístico del estado de la partida de Gwent del nodo.

```
1.  def UCTSeleccionarHijo(self, heuryes_or_no=None):
2.      # Heuristica
3.      if heuryes_or_no == 1:
4.          s = sorted(self.Nodos_Hijos, key=lambda c: c.victorias / c.visitas + sqrt(2 * log(self.visitas) / c.visitas) + self.valor_heuristica)[-1]
5.      else:
6.          s = sorted(self.Nodos_Hijos, key=lambda c: c.victorias / c.visitas + sqrt(2 * log(self.visitas) / c.visitas))[-1]
7.      return s
8.
9.  def AñadirHijo(self, m, s, heuryes_or_no, alpha1, alpha2, alpha3, alpha4, alpha5, alpha6):
10.     n = Nodo(movimiento=m, padre=self, estado=s, heuryes_or_no=heuryes_or_no, alpha1=alpha1, alpha2=alpha2, alpha3=alpha3, alpha4=alpha4, alpha5=alpha5, alpha6=alpha6)
11.     self.movimientos_sin_probar.remove(m)
12.     self.Nodos_Hijos.append(n)
13.     return n
14.
15. def Update(self, result):
16.     self.visitas += 1
17.     self.victorias += result
```

Tabla 7 Fragmento de código de Nodo: funciones UCTSeleccionarHijo, AñadirHijo y Update

4.3.2. Monte Carlo

A continuación, se va a explicar el funcionamiento de la función que ejecuta la búsqueda en árbol de Monte Carlo.

```
1.  def UCT(player_identity, estado_raiz, tiempo_ejecucion, verbose=False, heuryes_or_no=None, alpha1=None, alpha2=None, alpha3=None, alpha4=None, alpha5=None, alpha6=None):
2.      # tiempo_ejecucion en segundos
3.      # Cambiamos la identidad del jugador que simula.
4.      estado_raiz.change_player(player_identity)
5.
6.      nodo_raiz = Nodo(estado=estado_raiz,heuryes_or_no=heuryes_or_no, alpha1=alpha1, alpha2=alpha2, alpha3=alpha3, alpha4=alpha4, alpha5=alpha5, alpha6=alpha6)
7.      timeout_start = time.time()
8.
9.      while time.time() < timeout_start + tiempo_ejecucion:
10.         nodo = nodo_raiz
11.         estado = estado_raiz.clone()
12.
13.         # Seleccion
14.         while nodo.movimientos_sin_probar == [] and nodo.Nodos_Hijos != []:
15.             nodo = nodo.UCTSeleccionarHijo(heuryes_or_no)
16.             estado.play(nodo.movimiento)
```

```
17.
18.         # Expansion
19.         if nodo.movimientos_sin_probar:
20.             m = random.choice(nodo.movimientos_sin_probar)
21.             estado.play(m)
22.             nodo = nodo.AñadirHijo(m, estado, heuryes_or_no, alpha1=alpha1, alpha2=alpha2, alpha
    3=alpha3, alpha4=alpha4, alpha5=alpha5, alpha6=alpha6)
23.         # Simulacion
24.         while estado.Rondas_Ganadas_1 < 2 and estado.Rondas_Ganadas_2 < 2:
25.             random_move = estado.get_random_move()
26.             estado.play(random_move)
27.
28.         # Retropropagacion
29.         while nodo is not None:
30.             nodo.Update(estado.result(nodo.player_id))
31.             nodo = nodo.Nodo_Padre
32.
33.     return sorted(nodo_raiz.Nodos_Hijos, key=lambda c: c.visitas)[-1].movimiento
```

Tabla 8 Fragmento de código de Monte Carlo

4.3.2.1. Selección

Se han implementado dos estrategias de selección: UCT y UCT + Heurística.

En la función UCT se ha fijado el parámetro C a $\sqrt{2}$, puesto que es el valor teórico que minimiza la pérdida (Kocsis, 2006).

La función heurística evalúa el estado de juego del nodo mediante seis factores: la diferencia entre el número de cartas que tenemos en mano y las que tenga el rival, la diferencia entre el número de efectos adversos en el lado del jugador y del contrario, la diferencia entre el número de efectos beneficiosos en nuestro campo y en el del adversario, la diferencia entre el número de rondas ganadas, si hemos pasado en la ronda actual o no y la diferencia entre las puntuaciones de cada jugador.

$$\alpha_1 * \left(\frac{C_j - C_a}{C_j + C_a}\right) + \alpha_2 * \left(\frac{H_a - H_j}{3}\right) + \alpha_3 * \left(\frac{B_j - B_a}{3}\right) +$$

$$\alpha_4 * (R_j - R_a) + \alpha_5 * (P_a - P_j) + \alpha_6 * \left(\frac{V_j - V_a}{V_j + V_a}\right)$$

Donde C es el número de cartas en mano, H el número de efectos adversos en el lado del tablero, B el número de efectos beneficiosos en el lado del tablero, R el número de rondas ganadas, P si se ha pasado o no y V la puntuación total del jugador; donde *j* es el jugador y *a* es el adversario.

El objetivo de la heurística es dirigir la búsqueda para aumentar la explotación de los mejores estados y reducir la explotación de los peores estados.

4.3.2.2. Expansión

La estrategia de expansión implementada consiste en añadir al nodo seleccionado un único nodo hijo, seleccionado de forma aleatoria de entre todos movimientos sin probar.

4.3.2.3. Simulación

La estrategia de simulación implementada consiste en simular una partida donde ambos jugadores juegan de forma aleatoria. Para mejorar el comportamiento de esta estrategia se implementó la función get_random_move en EstadoGwent para reducir el coste computacional de obtener el movimiento aleatorio.

4.3.2.4. Retropropagación

La estrategia de retropropagación implementada consiste en recorrer el árbol desde el nodo en el que se inicia la simulación hasta el nodo raíz, pasando por todos los padres intermedios y se actualiza su número de visitas sumando uno y su número de victorias dependiendo del resultado de la simulación: 1 si el jugador ganó la partida, 0 si la perdió y 0.5 si la partida terminó en empate.

4.3.2.5. Selección del movimiento final

La estrategia de selección final implementada es *Robust Child*, el nodo hijo que ha sido visitado más veces a lo largo de la búsqueda.

4.4. Versión de Gwent implementada

Debido a la complejidad del juego, la necesidad de determinar empíricamente algunas reglas del Gwent y las restricciones temporales, la versión del Gwent que soporta el simulador está limitada en los siguientes aspectos:

Se han implementado los efectos más comunes de las cartas, que son los siguientes:

- Damage: Reduce el valor actual de la carta en la cantidad indicada.

- Armor: Añade la cantidad de armadura correspondiente.
- Boost: Aumenta el valor actual de la carta en la cantidad indicada.
- Strengthen: Aumenta el valor base de la carta en la cantidad indicada.
- Weaken: Reduce el valor base de la carta en la cantidad indicada.
- Reset: Iguala el valor actual de la carta a su valor base.
- Heal: Iguala el valor actual de la carta a su valor base, solo si el valor actual es menor al valor base.
- Resilience: Añade el efecto resistencia a la carta.
- Destroy: Destruye una carta y la envía al cementerio.
- Banish: Destierra una carta de la partida.
- Resurrect: resucita una carta del cementerio y la añade al tablero.
- Summon: invoca una carta de la baraja y la añade al tablero.

Y los modificadores de efecto siguientes:

- Area: aplica el efecto a la carta objetivo y a las cartas adyacentes que correspondan.
- Random: aplica el efecto a cartas de forma aleatoria dentro de la zona de efecto.
- And: aplica varios efectos a la misma carta o a distintas.
- Or: aplica un efecto de la lista de efectos de la carta.

Se han limitado los efectos de resucitar e invocar, de forma que no se permite que se encadene el efecto: Una carta puede resucitar o invocar a otra carta cuyo efecto sea resucitar o invocar, pero el efecto de la segunda carta no se ejecuta.

Se han implementado 46 cartas de las 480 que tiene Gwent actualmente, perteneciendo estas a cuatro facciones de las cinco que tiene el juego, además de las cartas neutrales. Se eligió implementar estas cartas porque son las que componen las barajas iniciales de cada facción en la versión de Gwent 0.9.23.

Se ha tenido que modificar los efectos de algunas cartas implementadas porque eran demasiado complejos para el simulador, como aquellas que requerían combinar los modificadores And y Or.

5. Evaluación y ajuste

En este apartado se va a hablar de la evaluación y ajuste de los pesos de la función heurística para maximizar el número de victorias obtenidas respecto a otro jugador que emplea UCT como su estrategia de selección.

5.1. Descripción de las pruebas realizadas

Se han implementado dos jugadores para la evaluación de la función heurística: el jugador 1, que emplea la función heurística en la etapa de selección y el jugador 2, que emplea UCT en dicha etapa.

Para ajustar los pesos de la función heurística se han ejecutado 400 partidas entre ambos jugadores, dando 1 minuto a cada jugador para elegir su jugada. Se han implementado 4 barajas, siendo cada una de una facción diferente, y se han jugado 16 partidas para cada combinación posible de pesos, tal y como se muestra en la tabla 9 a continuación. Las barajas implementadas se corresponden con las barajas iniciales de las facciones Reinos del Norte, Scoia'tael, Mostruos y Skellige, modificando y/o simplificando los efectos más complejos, puesto que no han sido implementados en el simulador.

Los pesos elegidos para las pruebas son 0, 2.5, 5, 7.5 y 10, significando el valor 5 una importancia media del parámetro, 2.5 una importancia baja, 0 que el parámetro carece de importancia, 7.5 una importancia elevada y 10 que el parámetro es muy importante.

El primer turno en las partidas se asigna al azar mediante la función random.randint() de Python.

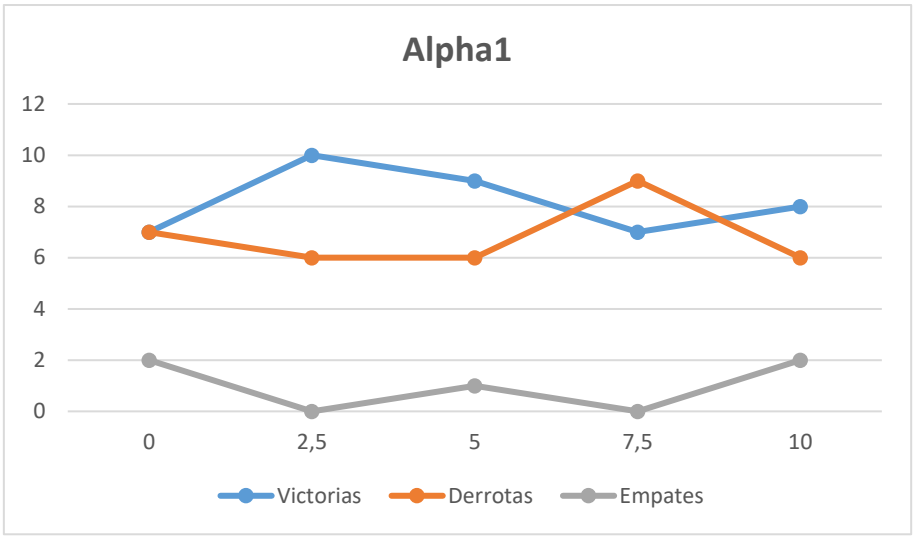
El equipo usado para ejecutar todas las pruebas es un ordenador con un procesador AMD RYZEN 5 1600X a 3925 MHZ y con un voltaje de 1.526V, 2 módulos de memoria RAM G.Skill Trident Z RGB DDR4 2400 PC4-19200 8GB CL15 y un voltaje de 1.351V y una placa base Asus Crosshair VI Hero.

5.2. Análisis de la eficacia del jugador heurístico respecto a la variación de los pesos

A continuación, se muestra una tabla con los resultados obtenidos para cada conjunto de pesos, para acto seguido analizar la variación de la eficacia del jugador al variar cada peso de forma individual:

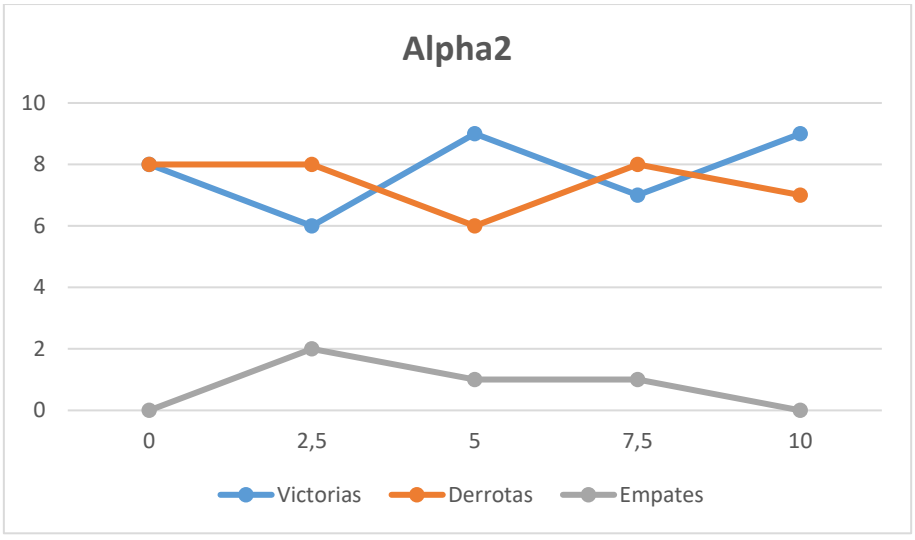
Alpha1	Alpha2	Alpha3	Alpha4	Alpha5	Alpha6	Victorias	Derrotas	Empates
5	5	5	5	5	5	9	6	1
0	5	5	5	5	5	7	7	2
2,5	5	5	5	5	5	10	6	0
7,5	5	5	5	5	5	7	9	0
10	5	5	5	5	5	8	6	2
5	0	5	5	5	5	8	8	0
5	2,5	5	5	5	5	6	8	2
5	7,5	5	5	5	5	7	8	1
5	10	5	5	5	5	9	7	0
5	5	0	5	5	5	8	7	1
5	5	2,5	5	5	5	9	6	1
5	5	7,5	5	5	5	9	6	1
5	5	10	5	5	5	8	8	0
5	5	5	0	5	5	6	10	0
5	5	5	2,5	5	5	5	9	2
5	5	5	7,5	5	5	6	9	1
5	5	5	10	5	5	10	5	1
5	5	5	5	0	5	9	7	0
5	5	5	5	2,5	5	11	5	0
5	5	5	5	7,5	5	7	8	1
5	5	5	5	10	5	8	6	2
5	5	5	5	5	0	9	7	0
5	5	5	5	5	2,5	4	10	2
5	5	5	5	5	7,5	8	7	1
5	5	5	5	5	10	7	9	0

Tabla 9 Resultados de las partidas para cada combinación de pesos probada



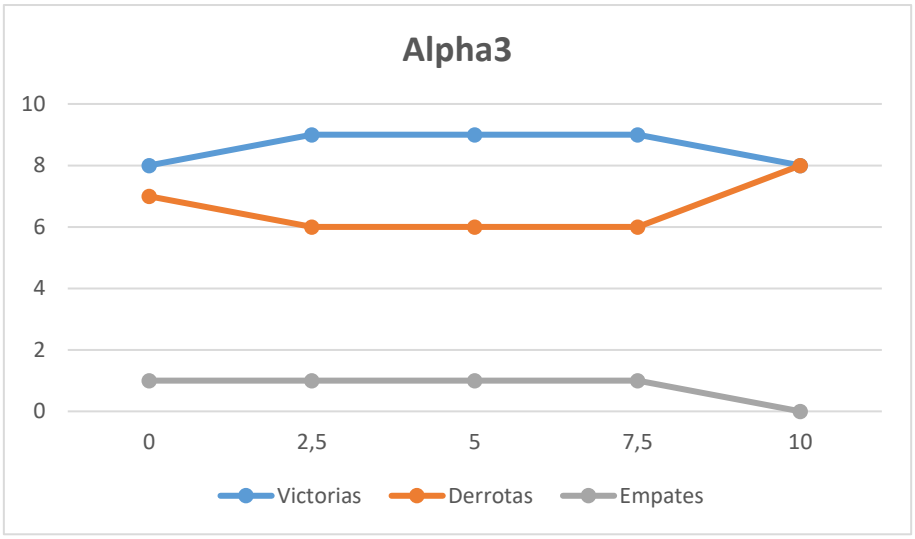
6 Gráfico de los resultados al variar el peso 1

El primer peso pondera la diferencia entre el número de cartas que tiene cada jugador. Como se puede observar en el gráfico X, el valor 7.5 para el primer peso obtiene el peor resultado en estas pruebas con 7 victorias y 9 derrotas, seguido del valor 0 con 7 derrotas, 7 victorias y 2 empates. Los valores que mejores resultados obtienen son 2.5 y 5 con 10 victorias y 9 victorias y un empate respectivamente. Por tanto, podemos concluir que la importancia de la diferencia entre el número de cartas en mano de cada jugador tiene una importancia medio-baja.



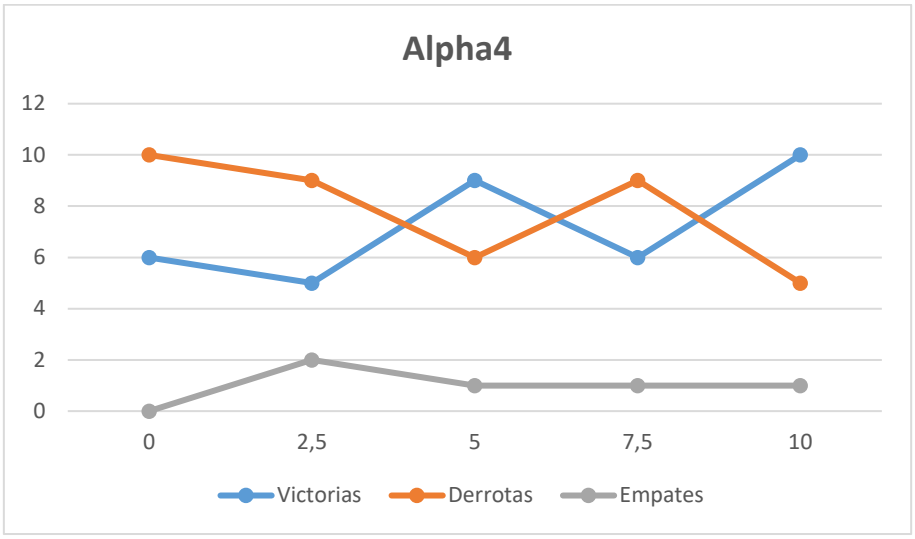
7 Gráfico de los resultados al variar el peso 2

El segundo peso pondera el número de efectos adversos presentes en la parte del tablero de cada jugador. Como se puede observar en el gráfico X, el valor 2.5 para el segundo peso obtiene el peor resultado con 6 victorias, 8 derrotas y 2 empates, seguido del valor 7.5 con 8 derrotas, 7 victorias y 1 empate. Los valores que mejores resultados obtienen son 5 y 10 con 9 victorias ambos, pero teniendo una derrota menos el valor 5. Por tanto, podemos concluir que la importancia de los efectos adversos en el campo tiene una importancia media.



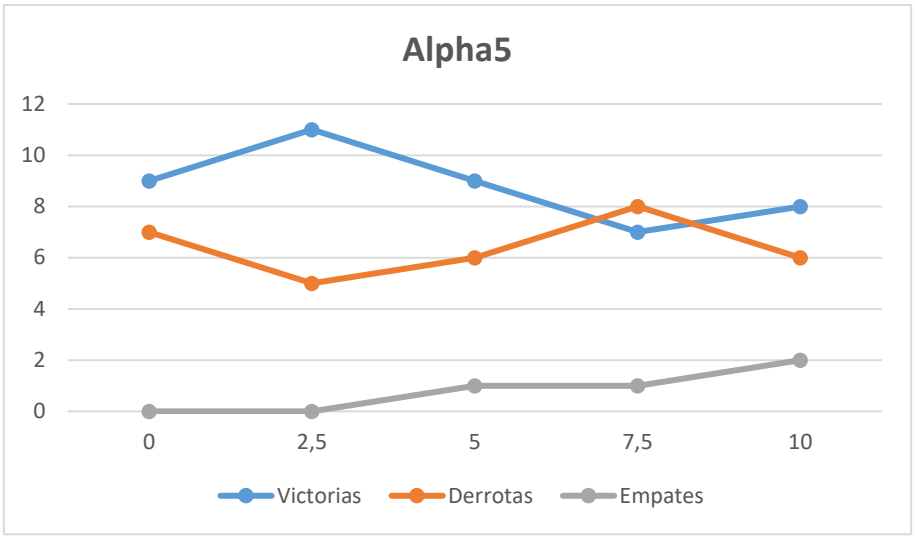
8 Gráfico de los resultados al variar el peso 3

El tercer peso pondera el número de efectos beneficiosos presentes en la parte del tablero de cada jugador. En las barajas implementadas no hay ninguna carta que añada efectos beneficiosos al tablero, de ahí la casi nula variación del número de victorias, derrotas y empates. El valor para este peso se ha asignado finalmente a 5, como el segundo peso, puesto que los efectos beneficiosos son inversos a los efectos adversos y, por tanto, su importancia debería ser similar.



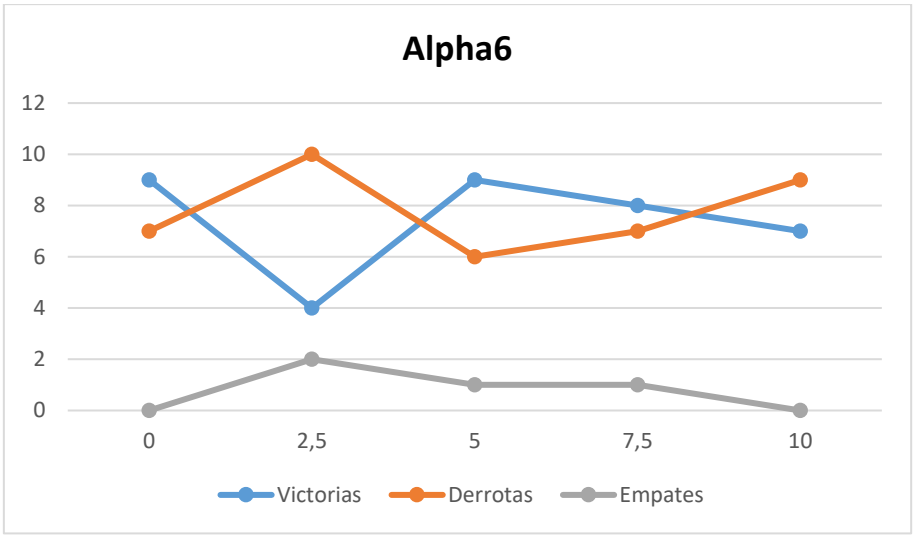
9 Gráfico de los resultados al variar el peso 4

El cuarto peso pondera el número de rondas que ha ganado cada jugador. Se puede observar el crecimiento del número de victorias y el decrecimiento del número de derrotas conforme se incrementa el valor de este parámetro. La importancia de este parámetro es máxima, puesto el número de rondas ganadas indica lo cerca que se está de ganar o perder la partida.



10 Gráfico de los resultados al variar el peso 5

El quinto peso pondera si los jugadores han pasado o no. Se puede observar que los valores 7.5 y 10 son los que obtienen menor números de victorias, siendo 2.5 el valor que más victorias obtiene, concretamente 11. Por tanto, la importancia de pasar es medio-baja.

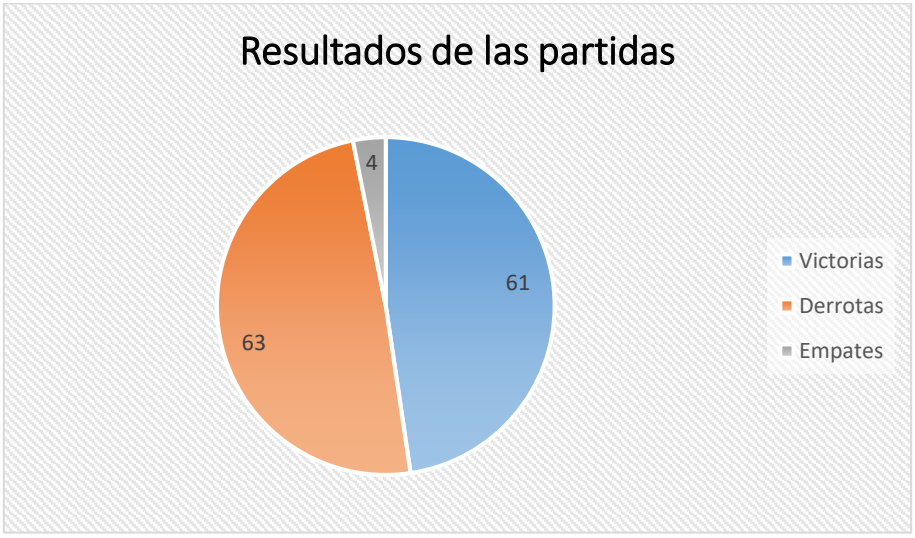


11 Gráfico de los resultados al variar el peso 6

El sexto peso pondera la diferencia de puntuación de los jugadores en la ronda actual. Como puede observarse en la gráfica, el valor 2.5 obtiene el peor resultado con 4 victorias y 10 derrotas, y el valor 5 obtiene el mejor resultado con 9 victorias, 6 derrotas y 1 empate.

5.3. Análisis de la función heurística ajustada

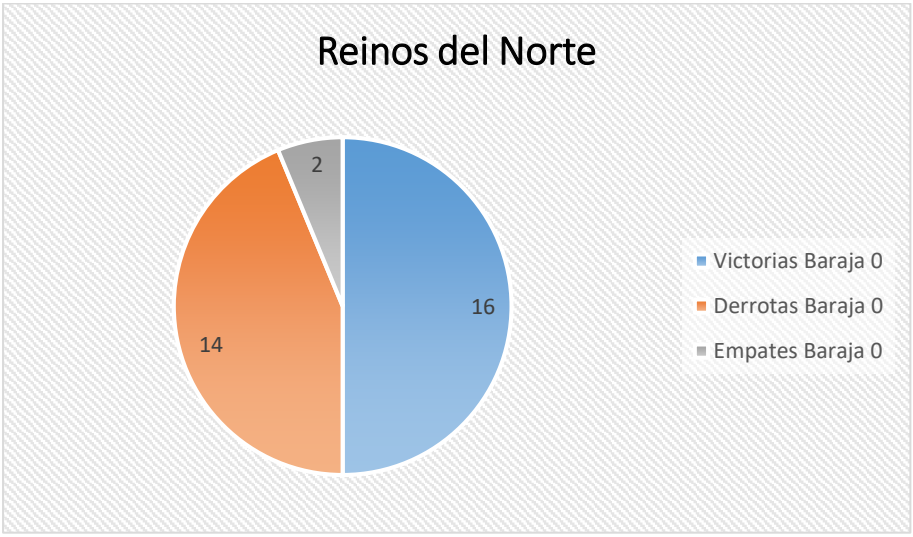
A continuación, se han ejecutado 128 partidas entre el jugador 1 con la función heurística con los pesos ajustados y el jugador 2, siendo estos los resultados obtenidos:



12 Resultados del primer ajuste

Como puede observarse en el gráfico, el jugador 1 ha ganado un 47,656% de las partidas, perdido el 49,219% y empatado 3,125%. Se procede a analizar los resultados obtenidos por el jugador 1 con cada baraja implementada.

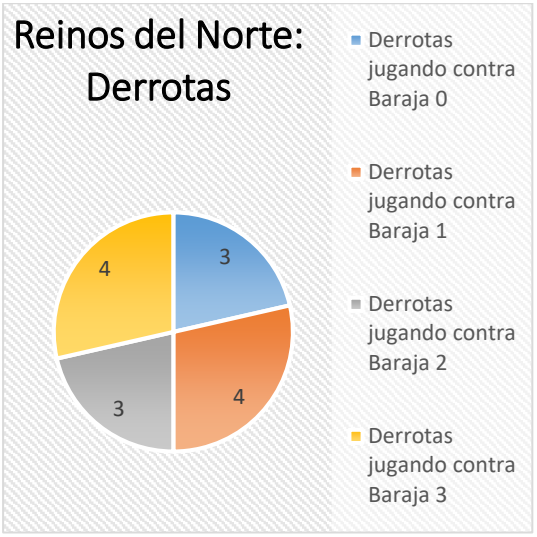
5.3.1. Análisis de los resultados de la baraja 0



13 Resultados de la baraja 0 con el primer ajuste

Como podemos observar, el jugador 1 gana aproximadamente la mitad de las partidas que juega con la baraja 0. Se procede a desglosar las victorias y las derrotas.

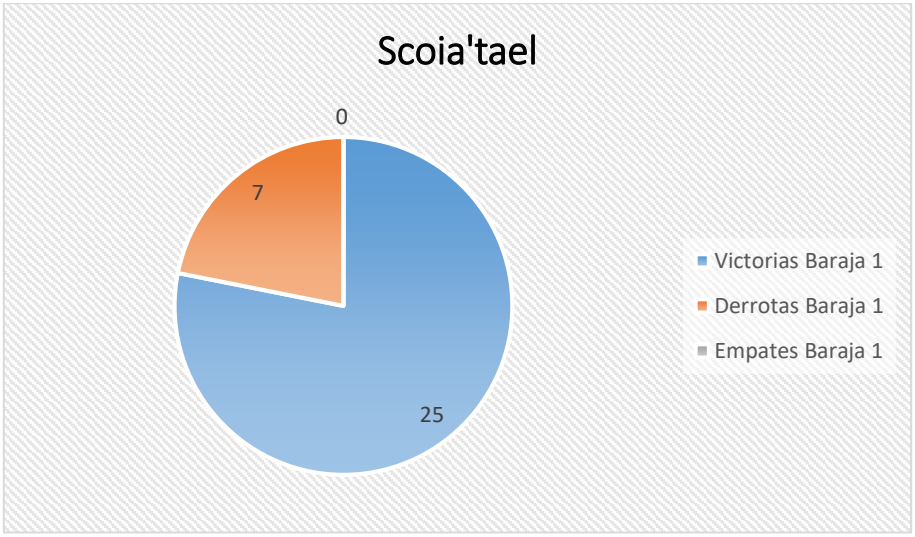
14 Victorias de la baraja 0 con el primer ajuste



15 Derrotas de la baraja 0 con el primer ajuste

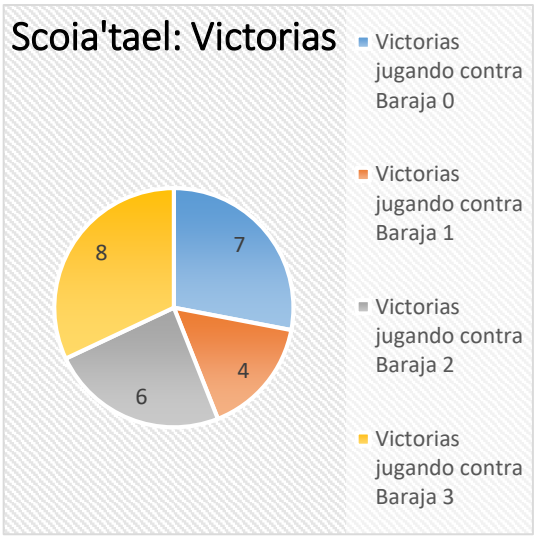
Por los resultados, se puede observar que el jugador heurístico con la baraja 0 gana la mitad de sus partidas, independientemente de qué baraja emplee el contrincante. Esto nos indica que el jugador heurístico, con esta baraja, tiene un nivel de juego similar al del jugador 2.

5.3.2. Análisis de los resultados de la baraja 1

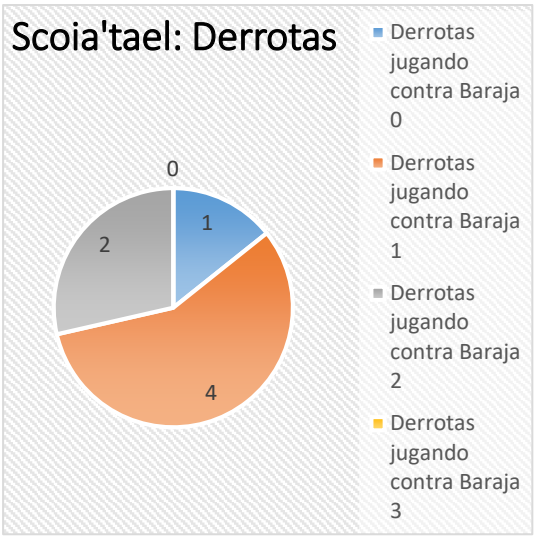


16 Resultados de la baraja 1 con el primer ajuste

A diferencia de la baraja 0, al jugar con la baraja 1, el jugador heurístico gana un 78,125% de las partidas que juega. Se procede a desglosar las victorias y las derrotas.



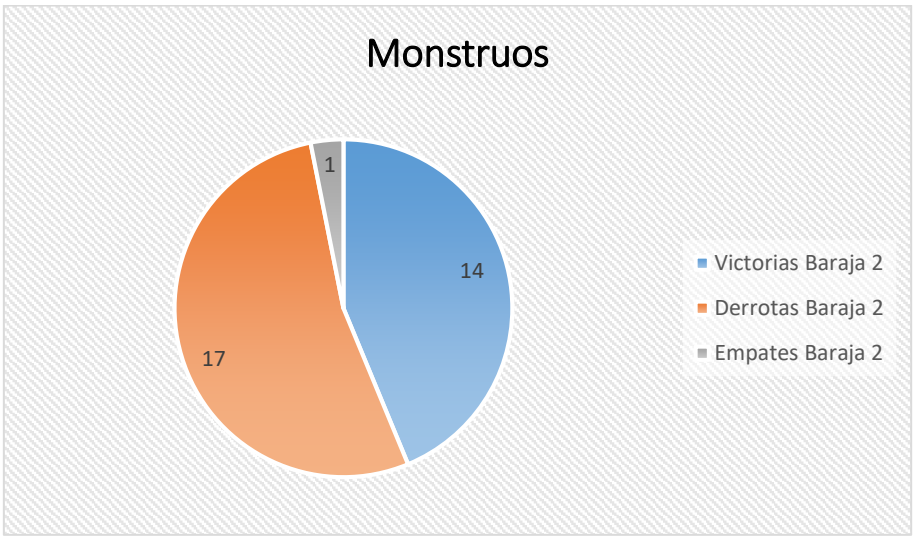
17 Victorias de la baraja 1 con el primer ajuste



18 Derrotas de la baraja 1 con el primer ajuste

Lo primero que se puede observar, es que el jugador heurístico con esta baraja no pierde ninguna partida contra el jugador 2 con la baraja 3, y solo pierde una partida contra la baraja 0. También se puede observar que, cuando ambos jugadores usan la baraja 1, el jugador heurístico gana la mitad de las veces. Esto puede deberse, principalmente, a dos factores: que la baraja 1 sea más fuerte que el resto de las barajas implementadas, o que la función heurística favorezca un estilo de juego agresivo, que sería el más apropiado para esta baraja en concreto, al contener cartas resistentes.

5.3.3. Análisis de los resultados de la baraja 2

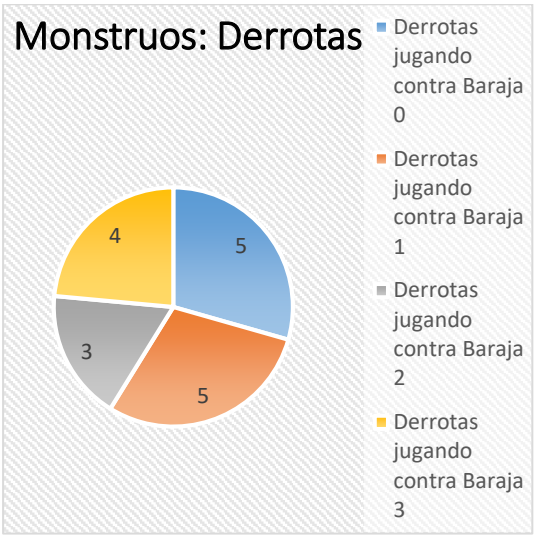


19 Resultados de la baraja 2 con el primer ajuste

Los resultados del jugador heurístico al jugar con la baraja 2 son similares a los obtenidos con la baraja 0, aunque en este caso pierde un 53,125% de las

partidas contra el jugador 2. Procedemos a desglosar las victorias y las derrotas.

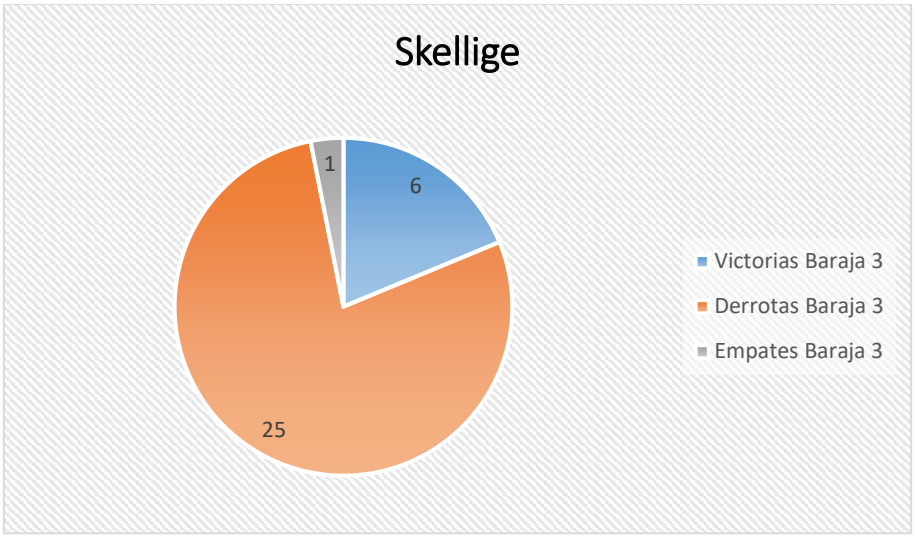
20 Victorias de la baraja 2 con el primer ajuste



21 Derrotas de la baraja 2 con el primer ajuste

Se puede observar que el jugador heurístico gana 5 veces y pierde 3 contra el segundo jugador cuando ambos usan la misma baraja. También se puede observar que los resultados obtenidos al jugar contra la baraja 0 y la 1 son de 3 victorias y 5 derrotas. Esto, unido a la información analizada anteriormente, induce a pensar que el jugador heurístico juega ligeramente mejor que el jugador 2 cuando ambos usan la misma baraja, y que juega ligeramente peor cuando el jugador 2 usa otra baraja distinta.

5.3.4. Análisis de los resultados de la baraja 3

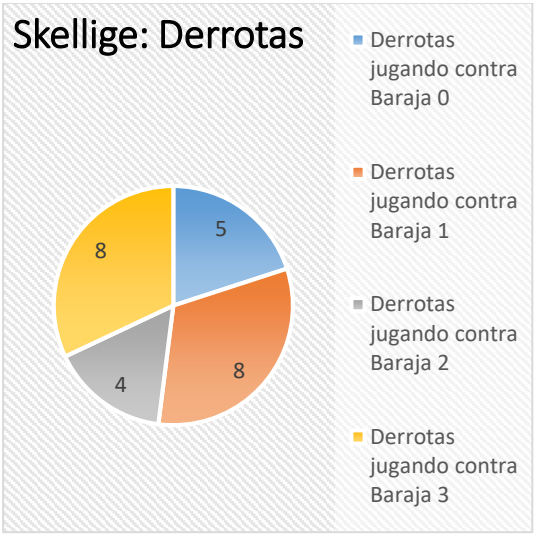


22 Resultados de la baraja 3 con el primer ajuste

Los resultados obtenidos por el jugador heurístico al jugar con la baraja 3 son desastrosos: 6 victorias, 1 empate y 25 derrotas. Esto puede ser debido a

dos factores principalmente: que la baraja 3 sea la más débil de todas las implementadas o que el ajuste de la función heurística no sea adecuado para esta baraja en concreto. Se procede a desglosar las victorias y las derrotas de la baraja 3 para hallar el motivo.

23 Victorias de la baraja 3 con el primer ajuste

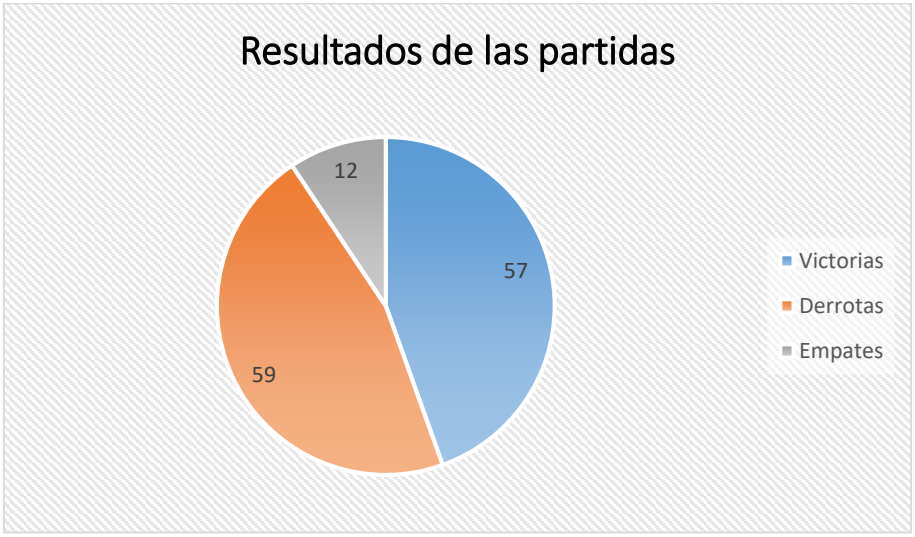


24 Derrota de la baraja 3 con el primer ajuste

Podemos observar que gana 4 partidas y pierde 4 al enfrentarse a la baraja 2, lo cual nos indica que, si bien la baraja podría ser la peor de las cuatro implementadas, el hecho de que empate con dicha baraja y que pierda todas las partidas contra el jugador cuando este usa la baraja 1 o la 3 indica que el ajuste de pesos de la heurística no es adecuado para esta baraja.

Habiendo analizado los resultados obtenidos con el ajuste de la función heurística, se propone repetir las pruebas realizadas al jugador heurístico, cambiando los pesos para mejorar el rendimiento obtenido al jugar con la baraja 3. Los valores elegidos son todos a 5 salvo el peso 4, que pondremos a 10.

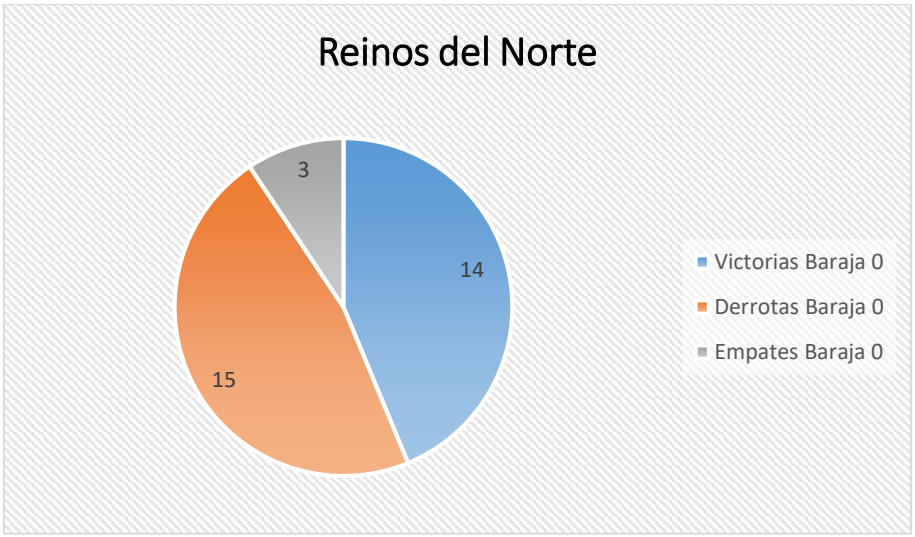
5.4. Análisis de la heurística con el segundo ajuste



25 Resultados del segundo ajuste

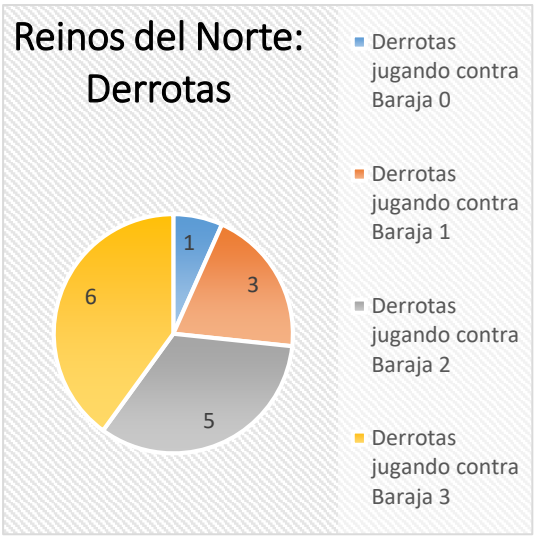
Como podemos observar, el número de empates ha aumentado, pero sigue habiendo un equilibrio entre las victorias y las derrotas cercano al 50%, tal y como ocurría con el ajuste anterior. Pasaremos a desglosar y analizar los resultados.

5.4.1. Análisis de los resultados de la baraja 0



26 Resultados de la baraja 0 con el segundo ajuste

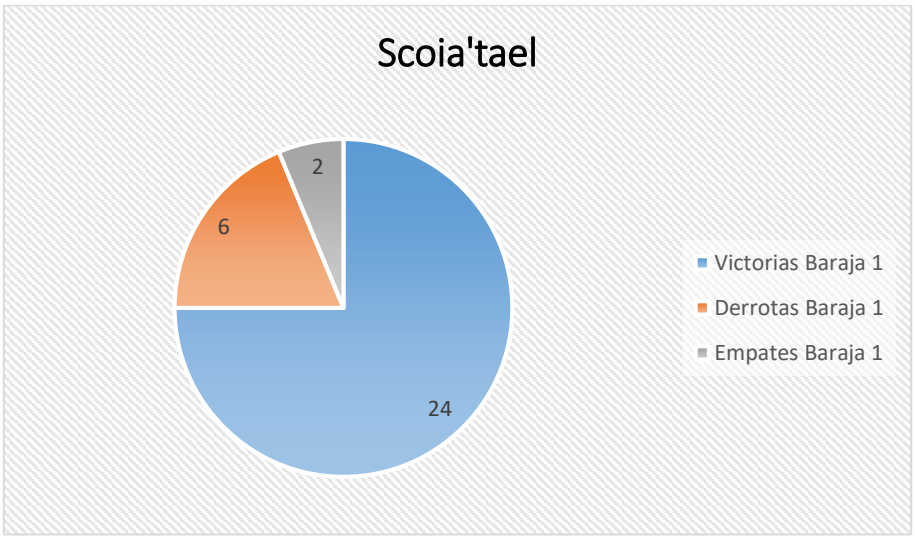
27 Victorias de la baraja 0 con el segundo ajuste



28 Derrotas de la baraja 0 con el segundo ajuste

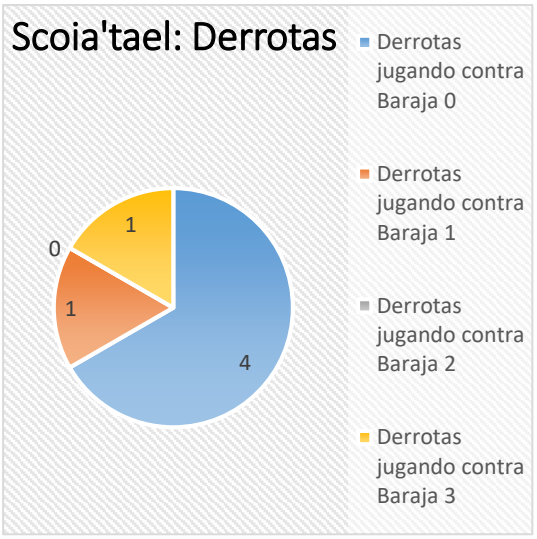
Aunque el resultado final de victorias y derrotas es similar al obtenido con el primer ajuste, con este segundo vemos que el jugador heurístico gana al jugador dos cuando este usa la baraja 0 o la 1, pero apenas gana cuando el jugador 2 usa la baraja 2 o la 3. Esto parece indicar que no solo es necesario ajustar la función para la baraja que use el jugador, sino también en función de la baraja del rival. En la práctica, esto no es factible, puesto que el número de barajas posibles que se pueden crear es intratable con este tipo de ajuste, por lo tanto, se debería ajustar la función dependiendo de la baraja con la que se juegue y de la que use el rival.

5.4.2. Análisis de los resultados de la baraja 1



29 Resultados de la baraja 1 con el segundo ajuste

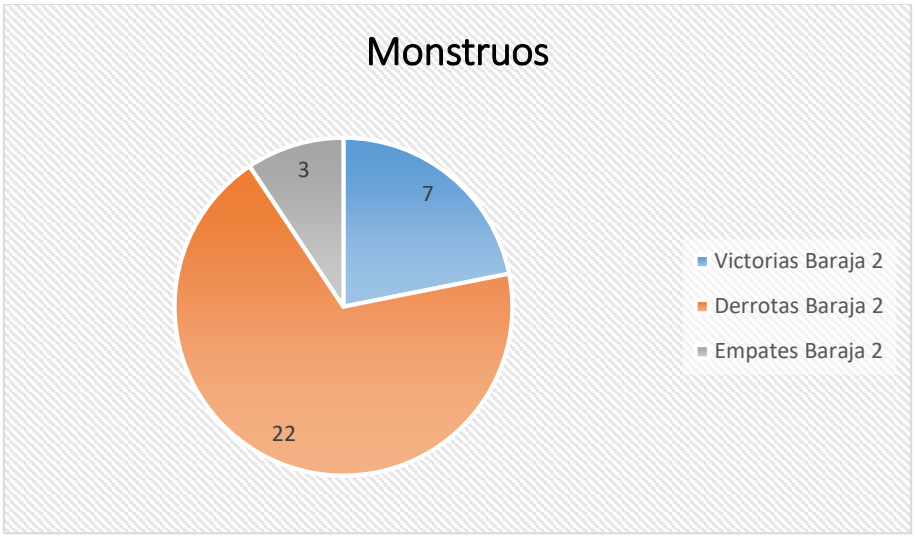
30 Victorias de la baraja 1 con el segundo ajuste



31 Derrotas de la baraja 1 con el segundo ajuste

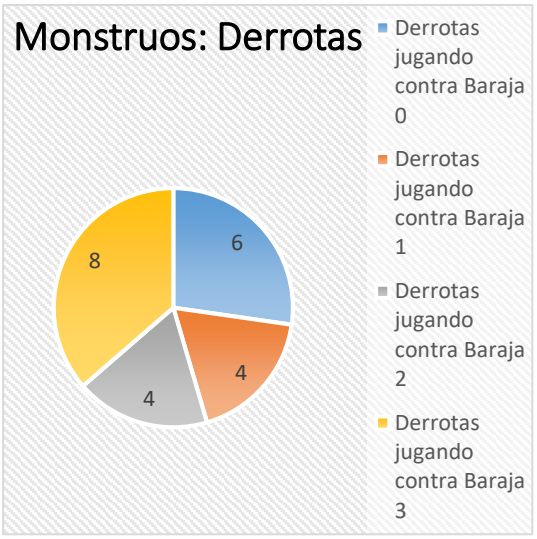
El principal cambio que podemos observar en los resultados es que ahora el jugador heurístico empata en número de victorias y derrotas con el jugador 2 cuando este usa la baraja 0, mientras que con el anterior ajuste ese empate se producía cuando el jugador 2 usaba la baraja 1. Esto refuerza la idea de que, para maximizar la eficacia de la función heurística, el ajuste debe hacerse teniendo en cuenta la baraja que usa el contrincante.

5.4.3. Análisis de los resultados de la baraja 2



32 Resultados de la baraja 2 con el segundo ajuste

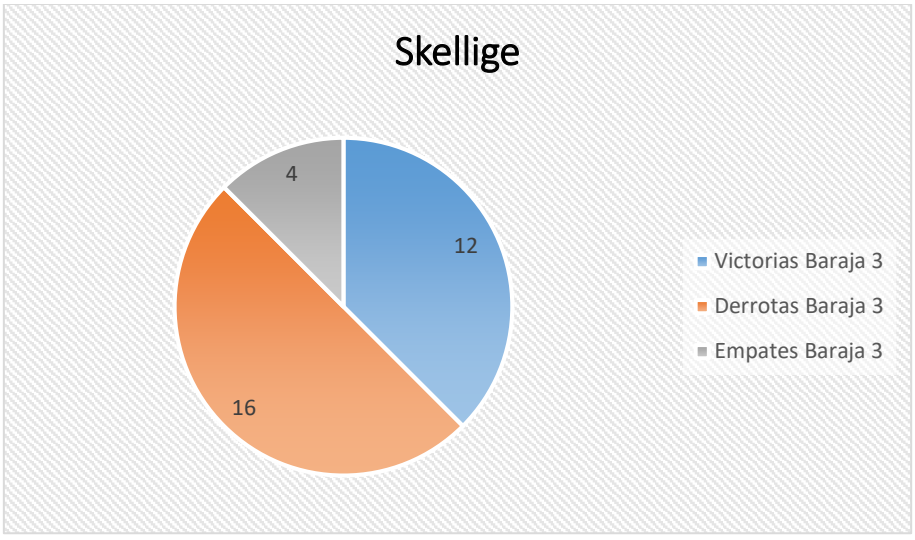
33 Victorias de la baraja 2 con el segundo ajuste



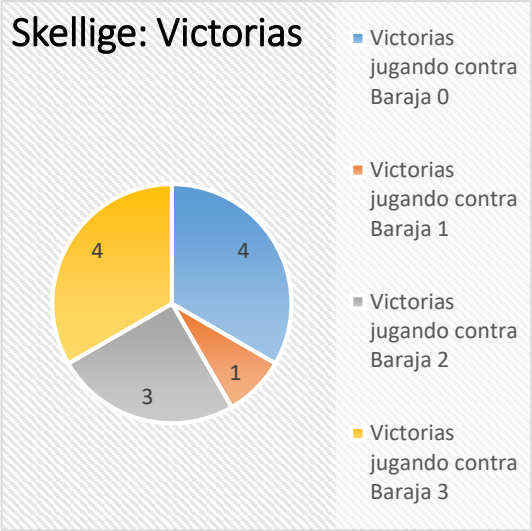
34 Derrotas de la baraja 2 con el segundo ajuste

Como podemos ver, con este ajuste el rendimiento del jugador heurístico al usar la baraja 2 es muy inferior al que obteníamos con el primer ajuste. Destaca el hecho de que no ha ganado ni una vez al jugador 2 cuando este usa la baraja 3, mientras que los resultados al enfrentarse a la baraja 0 son similares. Esto refuerza la idea de que el ajuste debe tener en cuenta la baraja del contrincante.

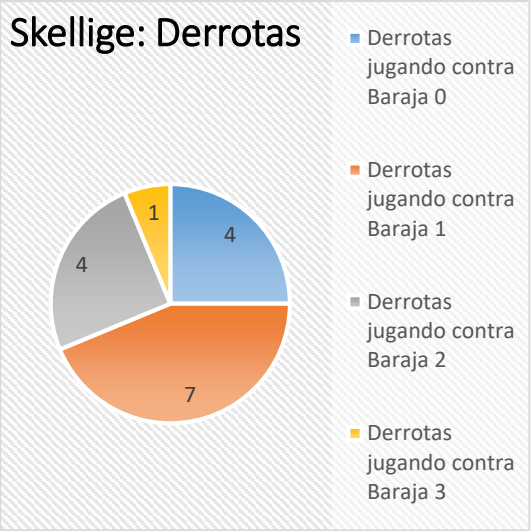
5.4.4. Análisis de los resultados de la baraja 3



35 Resultados de la baraja 3 con el segundo ajuste



36 Victorias de la baraja 3 con el segundo ajuste



37 Derrotas de la baraja 3 con el segundo ajuste

Como se puede observar, los resultados de la baraja 3 han mejorado de forma notable respecto del primer ajuste. Destaca que apenas hayan variado los resultados al enfrentarnos al jugador 2 cuando usa una baraja distinta de la 3, que es donde se ve la mejora. Esto refuerza la idea de que el ajuste debe tener en cuenta la baraja del rival.

5.5. Análisis de los resultados

Del análisis de los datos de las partidas, se puede concluir que la función heurística propuesta es capaz de mejorar el nivel de juego y superar al jugador que utiliza UCT en la etapa de selección; sin embargo, la función heurística depende de un ajuste no solo para la baraja que vaya a usar el jugador, sino también para la baraja a la que se vaya a enfrentar. Esto en la práctica no es viable, puesto que el número de barajas posibles es demasiado elevado como para compensar la mejora obtenida en el nivel de juego, pero si se limita a ajustar la heurística a formas o estilos de juego, seria asumible el coste temporal de obtener dicho ajuste.

Sería interesante comprobar el funcionamiento y compararlo con los jugadores ya implementados, de un tercer jugador que use la misma función heurística, pero incorporé el valor proporcionado por la misma usando la técnica de sesgo progresivo (Chaslot, 2007) para que el peso de la heurística en la fase de selección descienda conforme más se explore el nodo y sus hijos. Con esta técnica, podría evitarse en cierta medida la elevada dependencia de la heurística de tener un ajuste tan especifico y mejorar el funcionamiento de MCTS puro.

6. Conclusiones

Los objetivos propuestos en el apartado 1.2 fueron:

- Implementar un simulador del videojuego Gwent: The Witcher Card Game con sus principales características e implementar un número suficiente de cartas y de barajas, atendiendo a las restricciones temporales de realización del trabajo.
- Implementar un agente que juegue al Gwent razonablemente bien, usando para ello el método de árbol de búsqueda Monte Carlo.
- Diseñar e implementar una función heurística que guíe la fase de selección del árbol de búsqueda Monte Carlo hacia mejores estados.
- Ajustar la función heurística y analizar el impacto en el índice de victorias de cada peso de la función heurística.

Como se ha mostrado, se ha implementado de forma satisfactoria un simulador con las principales características de Gwent, con 46 cartas implementadas de las 480 que tiene Gwent actualmente y 4 barajas, pertenecientes cada una a una facción distinta.

También se ha implementado un agente capaz de jugar a Gwent usando MCTS de forma satisfactoria, que juega de forma coherente sin hacer jugadas incorrectas.

Acto seguido, se ha diseñado e implementado una heurística que, atendiendo a seis parámetros del estado de la partida, retorna una valoración de lo bueno que es ese estado al jugador.

Finalmente, se ha hecho un ajuste de los pesos de la función para maximizar el nivel de juego del jugador heurístico, pero los resultados no han sido todo lo buenos que cabría esperar. Tal y como se ha mostrado en el apartado 5, el nivel de juego del agente heurístico varía en función de qué baraja use y de a qué baraja se enfrente. Este hecho indica que el ajuste de la función debe tener en cuenta qué baraja usa el jugador heurístico y a qué baraja se enfrenta. Cómo obtener este ajuste es computacionalmente inviable, se propone ajustar los pesos dependiendo del estilo de juego que favorezcan las cartas del jugador y las del contrincante. También se propone implementar otro jugador que utilice la técnica de sesgo progresivo para reducir el peso de la función heurística en la fase de selección conforme avanza la ejecución de MCTS.

Para concluir, el resultado obtenido de este trabajo es la aplicación de la técnica MCTS con unos resultados razonables teniendo en cuenta la dificultad del juego y del tiempo disponible. Este diseño, implementación y los resultados obtenidos cubren los objetivos marcados y han permitido aplicar diversas técnicas vistas en el grado. Además, se plantean posibles desarrollos y mejoras que podrían estudiarse en el futuro.

Bibliografía

Abramson, B. 1987. *The Expected-Outcome Model of Two-Player Games.* Columbia University. 1987.

Andersson, M. H., Hesselberg, H. H. 2016. *Programming a Hearthstone agent using Monte Carlo Tree Search.* Department of Computer and Information Science, Norwegian University of Science and Technology. 2016.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games.* 2012, Vol. 4.

CD PROJEKT RED. 2018. Gwent: The Witcher Card Game. [En línea] CD PROJEKT, 2018. https://www.playgwent.com/es/.

Chaslot, G. M. J. B. 2010. *Monte-Carlo Tree Search.* Universiteit Maastricht. 2010. pág. 22. ISBN 978-90-8559-099-6.

Chaslot, G.M.J.B., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J., Bouzy, B. 2007. Progressive Strategies for Monte-Carlo Tree Search. 15 de Mayo de 2007.

Kocsis, L., Szepesvári, C. 2006. *Bandit based Monte-Carlo Planning.* Computer and Automation Research Institute, Hungarian Academy of Sciences. 2006.

Russell, S. J., Norvig, P. 2008. *Inteligencia artificial. Un enfoque moderno.* Segunda. s.l. : Pearson Prentice Hall, 2008. págs. 181-207. ISBN 978-84-205-4003-0.

Santos, A., Santos, P. A., Melo, F. S. 2017. *Monte Carlo Tree Search Experiments in Hearthstone.* Instituto Superior Técnico/INESC-ID, Universidade de Lisboa. 2017.

Sylvain, G., Wang, Y., Munos, R., Teytaud, O. 2006. Modification of UCT with Patterns in Monte-Carlo Go [Research Report] RR-6062. 20 de Diciembre de 2006.

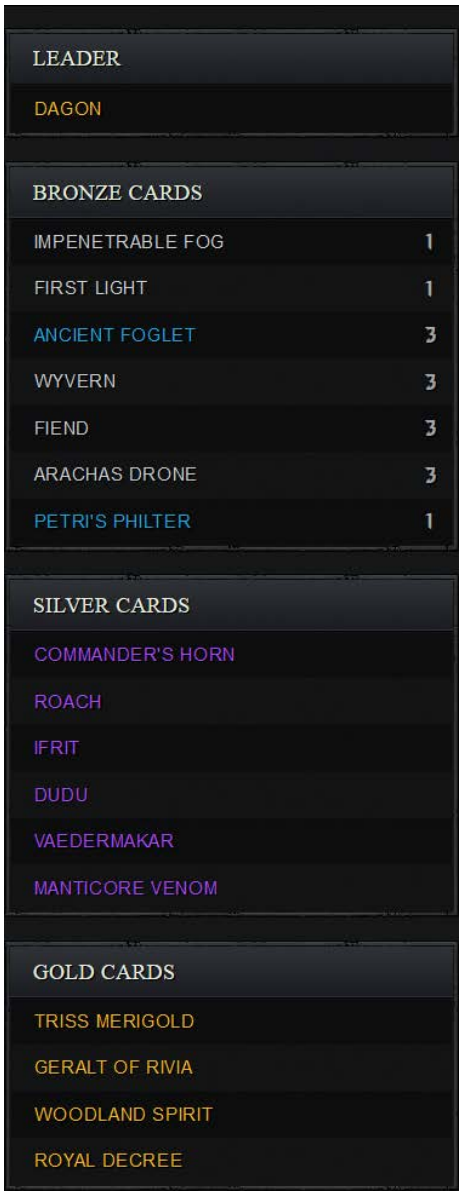
Wijman, T. 2018. Newzoo. [En línea] Newzoo, 30 de Abril de 2018. https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/.

Anexo

Composición de las barajas utilizadas

38 Composición de la baraja 0

39 Composición de la baraja 1



40 Composición de la baraja 2

41 Composición de la baraja 3

Datos de las partidas con el primer ajuste

Todos los resultados son desde la perspectiva del jugador 1.

Alpha1	Alpha2	Alpha3	Alpha4	Alpha5	Alpha6	Victorias	Derrotas	Empates	Partidas
2,5	5	5	10	2,5	5	61	63	4	128

Tabla 10 Resultados del primer ajuste

	Victorias jugando contra			
	Baraja 0	Baraja 1	Baraja 2	Baraja 3
Baraja 0	4	4	4	4
Baraja 1	7	4	6	8
Baraja 2	3	3	5	3
Baraja 3	2	0	4	0

Tabla 11 Victorias del primer ajuste

	Derrotas jugando contra			
	Baraja 0	Baraja 1	Baraja 2	Baraja 3
Baraja 0	3	4	3	4
Baraja 1	1	4	2	0
Baraja 2	5	5	3	4
Baraja 3	5	8	4	8

Tabla 12 Derrotas del primer ajuste

	Empates jugando contra			
	Baraja 0	Baraja 1	Baraja 2	Baraja 3
Baraja 0	1	0	1	0
Baraja 1	0	0	0	0
Baraja 2	0	0	0	1
Baraja 3	1	0	0	0

Tabla 13 Empates del primer ajuste

Datos de las partidas con el segundo ajuste

Todos los resultados son desde la perspectiva del jugador 1.

Alpha1	Alpha2	Alpha3	Alpha4	Alpha5	Alpha6	Victorias	Derrotas	Empates	Partidas
5	5	5	10	5	5	57	59	12	128

Tabla 14 Resultados del segundo ajuste

	Victorias jugando contra			
	Baraja 0	Baraja 1	Baraja 2	Baraja 3
Baraja 0	6	5	2	1
Baraja 1	4	5	8	7
Baraja 2	2	2	3	0
Baraja 3	4	1	3	4

Tabla 15 Victorias del segundo ajuste

	Derrotas jugando contra			
	Baraja 0	Baraja 1	Baraja 2	Baraja 3
Baraja 0	1	3	5	6
Baraja 1	4	1	0	1
Baraja 2	6	4	4	8
Baraja 3	4	7	4	1

Tabla 16 Derrotas del segundo ajuste

	Empates jugando contra			
	Baraja 0	Baraja 1	Baraja 2	Baraja 3
Baraja 0	1	0	1	1
Baraja 1	0	2	0	0
Baraja 2	0	2	1	0
Baraja 3	0	0	1	3

Tabla 17 Empates del segundo ajuste